OPTIMIZING PARALLEL APPLICATIONS

by

Shih-Hao Hung

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1998

Doctoral Committee:

Professor Edward S. Davidson, Chair
Professor William R. Martin
Professor Trevor N. Mudge
Professor Quentin F. Stout

For my family.

ACKNOWLEDGMENT

I owe a special thanks to my advisor, Professor Edward S. Davidson. Without his support, guidance, wisdom, and kindly revising my writing, my study toward Ph.D degree and writing this dissertation would have been much more painful. I would like to thank Prof. Gregory Hulbert, Prof. William Martin, Prof. Trevor Mudge, and Prof. Quentin Stout for their advises that helped improve this dissertation.

I would like to thank Ford Motor Company, Automative Research Ceter (U.S. Army-TACOM), and the National Partnership for Advanced Computational Infrastructure (NPACI, NSF Grant No. ACI-961920) for their generous financial support of this research. Parallel computer time was provided by the University of Michigan Center of Parallel Computing which is sponsored in part by NSF grant CDA-92-14296. Thanks to CPC staff, particularly Paul McClay, Andrew Caird, and Hal Marshall, for providing excellent technical support on the KSR, HP/Convex Exemplar, IBM SP2, and SGI PowerChallenger.

I have been fortunate to meet numerous marvelous fellow students in the Parallel Performance Project (formerly the Ford Project). Lots of my work in this dissertation has been inspired or based on their research. These are what I learned from these friends who have been my officemates in 2214 EECS: hierarchical performance bounds models (Tien-Pao Shih, Eric Boyd), synthetic workload (Eric Boyd), domain decomposition (Karen Tomko), relaxed synchronization (Alex Eichenberger), and machine performance evaluation (Gheith Abandah). Thanks to Ed Tam and Viji Srinivasan for letting me use their cache simulation tools. Jude Rivers has been a bright, delightful fellow of mine, whose constant skeptical attitude toward parallel computing has been one motivation for this research.

Thanks to many people, Ann Arbor has been a very wonderful place to me in the last 5 years. To people who I have played volleyball, basketball, softball, and majong with. To people who lend me tons of anime to kill countless hours. To my family back in Taiwan for their full support of my academic career over the years. To our cute little black cat, Ji-Ji, for bringing lots of laughters to our home. Finally, to my lovely wife, Weng-Ling, for always finding ways to make our lives here more interesting and meaningful.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1. INTRODUCTION

"A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem" [1]. Today, various types of parallel computers serve different usages ranging from embedded digital signal processing to supercomputing. In this dissertation, we focus on the application of parallel computing to solve large computational problems fast. Highly parallel supercomputers, with up to thousands of microprocessors, have been developed to solve problems that are beyond the reach of any traditional single processor supercomputer. By connecting multiple processors to a shared memory bus, parallel servers/ workstations have emerged as a cost-effective alternative to mainframe computers.

While parallel computing offers an attractive perspective for the future of computers, the parallelization of applications and the performance of parallel applications have limited the success of parallel computing. First, applications need to be parallel or parallelized to take advantage of parallel machines. Writing parallel applications or parallelizing existing serial applications can be a difficult task. Second, parallel applications are expected to deliver high performance. However, more often than not, the parallel execution overhead results in unexpectedly poor performance. Compared to uniprocessor systems, there are many more factors that can greatly impact the performance of a parallel machine. It is often a difficult and time-consuming process for users to exploit the performance capacity of a parallel computer, which generally requires them to deal with *limited inherent parallelism* in their applications, *inefficient parallel algorithms*, *overhead of parallel execution*, and/or *poor utilization of machine resources*. The latter two problems are what we intend to address in this dissertation.

Parallelization is a state-of-the-art process that has not yet been automated in general. Most programmers have been trained in and have experience with serial codes and there exist many important serial application codes that could well benefit from the performance increases offered by parallel computers. Automatic parallelization is possible for loops where data dependency can be analyzed by the compiler. Unfortunately, the complexity of interpro-

cedural data flow analysis often limits automatic parallelization to basic loops without procedure calls. Problems can occur even in these basic loops if, for example, there exist indirect data references such as pointers or indirectly-indexed arrays. Fortunately, many of those problems are solvable with some human effort, especially from the programmers themselves, to assist the compiler.

Regardless of whether parallelization is automatic or manual, high performance parallel applications are needed to better serve the user community. So far, while some parallel applications do successfully achieve high delivered performance, many others only achieve a small fraction of peak machine performance. It is often beyond the compiler's or the application developer's ability to accurately identify and consider the many machine-application interactions that can potentially affect the performance of the parallelized code. Over the last several decades, during which parallel computer architectures have constantly been modified and improved, tuned application codes and software environments for these architectures have had to be discontinued and rebuilt. Different application development tools have not been well integrated or automated. The methodology to improve software performance, i.e. performance tuning, like parallelization, has never been mature enough to reduce the tuning effort to routine work that can be performed by compilers or average programmers. Poor performance and painful experiences in performance tuning have greatly reduced the interest of many programmers in parallel computing. These problems must be solved to permit routine development of high performance parallel applications.

Irregular applications [2] (Section 1.3), including sparse problems and those with unstructured meshes, often require more human parallel programming effort than regular applications, due to their *indirectly indexed* data items and *irregular load distribution* among the problem subdomains. Most full scale scientific and engineering applications exhibit such irregularity, and as a result they are more difficult to parallelize, load balance and optimize. Due to the lack of systematic and effective performance-tuning schemes, many irregular applications exhibit deficient performance on parallel computers. In this dissertation, we aim to provide a unified approach to addressing this problem by integrating performance models, performance-tuning methodologies and performance analysis tools to guide the parallelization and optimization of irregular applications. This approach will also apply to the simpler problem of parallelizing and tuning regular applications.

In this chapter, we introduce several key issues in developing high performance applications on parallel computers. Section 1.1 classifies parallel architectures and parallel programming models and describes the parallel computers that we use in our experiments. Section 1.2 describes the process and some of the difficulties encountered in the parallelization of applications. In Section 1.3, we discuss some aspects of irregular applications and the performance problems they pose. In Section 1.4, we give an overview of current application development environments and define the goals of our research. Section 1.5 summarizes this chapter and overviews the organization of this dissertation.

1.1 Parallel Machines

There are various kinds of parallel computers, as categorized by Flynn [3]. We focus on *multiple instruction stream, multiple data stream (MIMD)* machines. MIMD machines can be further divided into two classes: *shared-memory* and *message-passing*. These two classes of machines differ in the type and amount of hardware support they provide for interprocessor communication. Interprocessor communications are achieved via different mechanisms on shared-memory machines and message-passing machines. A shared-memory architecture provides a globally shared physical address space, and processors can communicate with one another by sharing data with commonly known addresses, i.e. *global variables*. There may also be *local* or *private* memory spaces that belong to one processor or cluster of processors and are protected from being accessed by others. In *distributed shared-memory* (DSM) machines, logically shared memories are physically distributed across the system, and *non-uniform memory access* (NUMA) time results as a function of the distance between the requesting processor and the physical memory location that is accessed. In a message-passing architecture, processors have their own (disjoint) memory spaces and can therefore communicate only by passing messages among the processors. A message-passing architecture is also referred to as a *distributed-memory* or a *shared-nothing* architecture.

In this section, we briefly discuss two cases that represent current design trends in shared-memory architectures (the *HP/Convex Exemplar*) and message-passing architectures (the *IBM SP2*). The Center for Parallel Computing of the University of Michigan (CPC) provides us with access to these machines.

1.1.1 Shared-Memory Architecture - HP/Convex Exemplar

The HP/Convex[1] Exemplar SPP-1000 shared memory parallel computer was the first model in the Exemplar series. It has 1 to 16 hypernodes, with 4 or 8 processors per hypernode, for a total of 4 to 128 processors. Processors in different hypernodes communicate via four *CTI* (*Coherent Toroidal Interconnect*) rings. Each CTI ring supports global memory accesses with the IEEE SCI (Scalable Coherent Interface) standard [4]. Each hypernode on the ring is connected to the next by a pair of unidirectional links. Each link has a peak transfer rate of 600MB/sec.

Within each hypernode, four functional blocks and an I/O interface communicate via a 5-port crossbar interconnect. Each functional block contains two Hewlett-Packard PA-RISC7100 processors [5] running at 100MHz, 2 banks of memory and controllers. Each processor has a 1MB instruction cache and a 1MB data cache on chip. Each processor cache is a

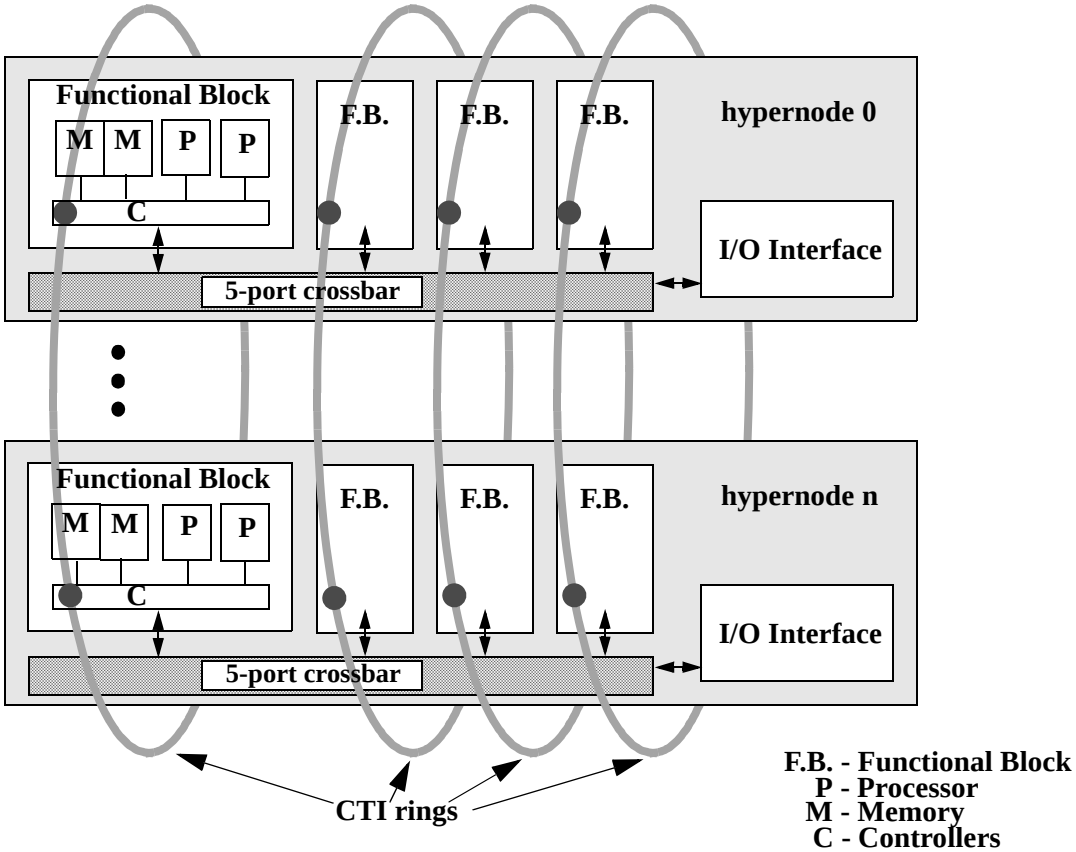


Figure 1-1: HP/Convex Exemplar System Overview.

1. Convex Computer Company was acquired by Hewlett-Packard (HP) in 1996. As of 1998, Convex is a division of HP and is responsible for the service and future development of the Exemplar series.

direct-mapped cache with a 32-byte line size. Each hypernode contains 256MB to 2GB of physical memory, which is partitioned into three sections: *hypernode-local*, *global*, and *CTI-cache*. The hypernode-local memory can be accessed only by the processors within the same hypernode as the memory. The global memory can be accessed by processors in any hypernode. The CTIcaches reduce traffic on the CTI rings by caching global data that was recently obtained by this hypernode from remote memory (i.e. memory in some other hypernode). The eight physical memories in each hypernode are 8-way interleaved by 64-byte blocks. The basic transfer unit on the CTI ring is also a 64-byte block.

The CPC at the University of Michigan was among the first sites with a 32-processor SPP-1000 as the Exemplar series was introduced in 1994. In August, 1996, the CPC upgraded the machine to an SPP-1600. The primary upgrade in the SPP-1600 is the use of more advanced HP PA-RISC 7200 processors [6], which offer several major advantages over the predecessor, PA-RISC 7100: (1) Faster clock rate (the PA7200 runs at 120MHz), (2) the Runway Bus - a split transaction bus capable of 768 MB/s bandwidth, (3) an additional on-chip 2-KByte fully-associative assist data cache, and (4) a four state cache coherence protocol for the data cache.

The newest model in the Exemplar series is the SPP-2000, which incorporates HP PA-RISC 8000 processors [7]. The SPP-2000 has some dramatic changes in the architectural design of the processor and interconnection network, which provide a substantial performance improvement over the SPP-1600. Using HP PA-RISC family processors, the Exemplar series runs a version of UNIX, called SPP-UX, which is based on the HP-UX that runs on HP PA-RISC-based workstations. Thus, the Exemplar series not only maintains software compatibility within its family, but can also run sequential applications that are developed for HP workstations. In addition, using mass-produced processors reduces the cost of machine development, and enables more rapid upgrades (with minor machine re-design or modification), as new processor models become available.

1.1.2 Message-Passing Architecture - IBM SP2

The IBM *Scalable POWERparallel* SP2 connects 2 to 512 RISC System/6000 POWER2 processors via a communication subsystem, called the *High Performance Switch (HPS)*. Each processor has its private memory space that cannot be accessed by other processors. The HPS is a bidirectional multistage interconnect with wormhole routing. The IBM SP2 at CPC has 64

nodes in four towers. Each tower has 16 POWER2 processor nodes. Each of the 16 nodes in the first tower has a 66 MHz processor and 256MB of RAM. Each node in the second and third towers has a 66 MHz processor and 128MB of RAM. The last 16 nodes each have a 160 MHz processor and 1GB of RAM. Each node has a 64KB data cache and 32 KB instruction cache. The line size is 64 bytes for the data caches and 128 bytes for the instruction caches.

The SP2 runs the AIX operating system, a version of UNIX, and has C, C++, Fortran77, Fortran90, and High Performance Fortran compilers. Each POWER2 processor is capable of performing 4 floating-point operations per clock cycle. This system thus offers an aggregate peak performance of 22.9 GFLOPS. However, the fact that the nodes in this system differ in processor speed and memory capacity results in a *heterogeneous* system which poses additional difficulties in developing high performance applications. Heterogeneous systems, which often exist in the form of a network of workstations, commonly result due to incremental machine purchases. As opposed to heterogeneous systems, a *homogeneous* system, such as the HP/Convex SPP-1600 at CPC, uses identical processors and nodes, and this is easier to program and load-balance for scalability. In this dissertation, we focus our discussion on homogeneous systems, but some of our techniques can be applied to heterogeneous systems as well.

1.1.3 Usage of the Exemplar and SP2

The HP/Convex Exemplar and IBM SP2 at the CPC have been used extensively for developing and running production applications, as well as in performance evaluation research. Generally, shared-memory machines provide simpler programming models than message-passing machines, as discussed in the next section. The interconnect of the Exemplar is focused more on reducing communication latency, so as to provide faster short shared-memory communications. The SP2 interconnect is focused more on high communication bandwidth, in order to reduce the communication time for long messages, as well as to reduce network contention.

Further details of the Convex SPP series machines can be found in [8][9]. The performance of shared memory and communication on the SPP-1000 is described in detail in [10][11][12]. A comprehensive comparison between the SPP-1000 and the SPP-2000 can be found in [13]. Detailed performance characterizations of the IBM SP2 can be found in [14][15][16].

1.2 Parallelization of Applications

Parallelism is the essence of parallel computing, and parallelization exposes the parallelism in the code to the machine. While some algorithms (i.e. parallel algorithms) are specially designed for parallel execution, many existing applications still use conventional (mostly sequential) algorithms and parallelization of such applications can be laborious. For certain applications, the lack of parallelism may be due to the nature of the algorithm used in the code. Rewriting the code with a parallel algorithm could be the only solution. For other applications, limited parallelism is often due to (1) insufficient parallelization by the compiler and the programmer, and/or (2) poor runtime load balance. In any case, the way a code is parallelized is highly related to its performance.

To solve a problem by exploiting parallel execution, the problem must be decomposed. Both the *computation* and *data* associated with the problem need to be divided among the processors. As the alternative to *functional decomposition*, which first decomposes the computation, *domain decomposition* first partitions the data domain into disjoint subdomains, and then works out what computation is associated with each subdomain of data (usually by employing the "owner updates" rule). Domain decomposition is the method more commonly method used by programmers to partition a problem, because it results in a simpler programming style with a parallelization scheme that provides straightforward scaling to different numbers of processors and data set sizes.

In conjunction with the use of domain decomposition, many programs are parallelized in the *Single-Program-Multiple-Data (SPMD)* programming style. In a SPMD program, one copy of the code is replicated and executed by every processor, and each processor operates on its own data subdomain, which is often accessed in globally shared memory by using index expressions that are functions of its *Processor IDentification (PID)*. A SPMD program is *symmetrical* if every processor performs the same function on an equal-sized data subdomain. A *near-symmetrical* SPMD program performs computation symmetrically, except that one processor (often called the *master* processor) may be responsible for extra work such as executing serial regions or coordinating parallel execution. An asymmetrical SPMD program is considered as a Multiple-Programs-Multiple-Data (MPMD) program whose programs are packed into one code.

In this dissertation, we consider symmetrical or near-symmetrical SPMD programs that employ domain decomposition, because they are sufficient to cover a wide range of applications. Symmetrical or near-symmetrical SPMD programming style is favored not only because it is simpler for programmers to use, but also because of its scalability for running on different numbers of processors. Usually, a SPMD program takes the machine configuration as an input and then decomposes the data set (or chooses a pre-decomposed data set if the domain decomposition algorithm is not integrated into the program) based on the number of processors, and possibly also the network topology. For scientific applications that operate iteratively on the same data domain, the data set can often be partitioned once at the beginning and those subdomains simply reused in later iterations. For such applications, the runtime overhead for performing domain decomposition is generally negligible.

Parallel programming models refer to the type of support available for interprocessor communication. In a shared-memory programming model, the programmers declare variables as *private* or *global*, where processors share the access to global variables. In a message-passing programming model, the programmers explicitly specify communication using calls to message-passing routines. Shared-memory machines support shared-memory programming models as their native mode; while block moves between shared memory buffer areas can be used to emulate communication channels for supporting message-passing programming models [8]. Message-passing machines can support shared-memory programming models via software-emulation of shared virtual memory [17][18]. Judiciously mixing shared-memory and message-passing programming models in a program can often result in better performance. The HP/Convex Exemplar supports shared-memory programming with automatic parallelization and parallel directives in its enhanced versions of C and Fortran. Message-passing libraries, PVM and MPI, are also supported on this machine. The SP2 supports Fortran 90 and High-Performance Fortran (HPF) [19] parallel programming languages, as well as the MPI [20], PVM [21], and MPL message-passing libraries. Generally, parallelization with shared-memory models is less difficult than with message-passing models, because the programmers (or the compilers) are not required to embed explicit communication commands in the codes.

Direct compilation of serial programs for parallel execution does exist today [22][23][24], but the state-of-the-art solutions are totally inadequate. Current parallelizing compilers have some success parallelizing loops where data dependency can be found . Unfortunately, prob-

lems often occur when there exist indirect data references or function calls within the loops, which causes the compiler to make safe, conservative assumptions, which in turn can severely degrade the attainable parallelism and hence performance and scalability. This problem limits the use of automatic parallelization in practice. Therefore, most production parallelizing compilers, e.g. KSR Fortran [25] and Convex Fortran [26][27], are of very limited use for parallelizing application codes.

Interestingly, many of those problems can be solved by trained human experts. Use of conventional languages enhanced with parallel extensions, such as Message-Passing Interface (MPI), are commonly used by programmers to parallelize codes manually, and in fact many manually-parallelized codes perform better than their automatically-parallelized versions. So far, parallel programmers have been directly responsible for most parallelization work, and hence, the quality of parallelization today usually depends on the programmer's skill. Parallelizing large application codes can be very time-consuming, taking months or even years of trial-and-error development, and frequently, parallelized applications need further fine tuning to exploit each new machine effectively by maximizing the application performance in light of the particular strengths and weakness of the new machine.

Unfortunately, fine tuning a parallel application, even when code development and maintenance budgets would allow it, is usually beyond the capability of today's compilers and most programmers. It often requires an intimate knowledge of the machine, the application, and, most importantly, the machine-application interactions. *Irregular* applications are especially difficult for the programer or the compiler to parallelize and optimize. Irregular application and their parallelization and performance problems are discussed in the next section.

1.3 Problems in Developing Irregular Applications

1.3.1 Irregular Applications

Irregular applications are characterized by indirect array indices, sparse matrix operations, nonuniform computation requirements across the data domain, and/or unstructured problem domains [2]. Compared to regular applications, irregular applications are more difficult to *parallelize*, *load balance* and *optimize*. Optimal partitioning of irregular applications is an NP-complete problem. Compiler optimizations, such as cache blocking, loop transforma-

tions, and parallel loop detection, cannot be applied to irregular applications since the indirect array references are not known until runtime and the compilers therefore assume worst-case dependence. Interprocessor communications and the load balance are difficult to analyze without performance measurement and analysis tools.

For many regular applications, domain decomposition is straightforward for programmers or compilers to apply. For irregular applications, decomposition of unstructured domains is frequently posed as a *graph partitioning* problem in which the data domain of the application is used to generate a graph where *computation* is required for each data item (*vertex* of the graph) and *communication dependence* between data items are represented by the *edges*. The vertices and edges can be weighted to represent the amount of computation and communication, respectively, for cases where the load is nonuniform. Weighted graph partitioning is an *NP-complete* problem, but several efficient *heuristic* algorithms are available. In our research, we have used the *Chaco* [28] and *Metis* [29] domain decomposition tools, which implement several algorithms. Some of our work on domain decomposition is motivated and/or based on profile-driven and multi-weight weighted domain decomposition algorithms developed previously by our research group [2][30].

1.3.2 Example - CRASH

In this dissertation, an example application, *CRASH*, is a highly simplified code that realistically represents several problems that arise in an actual vehicle crash simulation. It is used here for demonstrating these problems and their solutions. A simplified high level sketch of the serial version of this code is given in Figure 1-2. *CRASH* exhibits irregularity in several aspects: indirect array indexing, unstructured meshes, and nonuniform load distribution. Because of its large data set size, communication overhead, multiple phase and dynamic load balance problems, this application requires extensive performance-tuning to perform efficiently on a parallel computer.

CRASH simulates the collision of objects and carries out the simulation cycle by cycle in discrete time. The vehicle is represented by a finite element mesh which is provided as input to the code, such as illustrated in Figure 1-3. Instead of a mesh, the barrier is implicitly modeled as a boundary condition. Elements in the finite-element mesh are numbered from **1** to

```
      program CRASH

      integer Type(Max_Elements),Num_Neighbors(Max_Elements)
      integer Neighbor(Max_Neighbors_per_Element,Max_Elements)
      real_vector Force(Max_Elements),Position(Max_Elements),
            Velocity(Max_Elements)
      real t, t_step
      integer i,j,type_element

      call Initialization

c Main Simulation Loop
      t=0

c First phase: generate contact forces
100   do i=1,Num_Elements
        Force(i)=Contact_force(Position(i),Velocity(i))
        do j=1,Num_Neighbors(i)
            Force(i)=Force(i)+Propagate_force(Position(i),Velocity(i),
              Position(Neighbor(j,i),Velocity(Neighbor(j,i))
        end do
      end do

c Second phase: update position and velocity
200   do i=1,Num_Elements
        type_element=Type(i)
        if (type_element .eq. plastic) then
            call Update_plastic(i, Position(i), Velocity(i), Force(i))
        else if (type_element .eq. glass) then
            call Update_elastic(i, Position(i), Velocity(i), Force(i))
        end if
      end do

      if (end_condition) stop
      t=t+t_step
      goto 100
      end
```

Figure 1-2: Example Irregular Application, CRASH.

Num_Elements. Depending on the detail level of the vehicle model, the number of elements varies.

Variable **Num_Neighbors(i)** stores the number of elements that element **i** interacts with (which in practice would vary from iteration to iteration). Array **Neighbors(*,i)** points to the elements that are connected to element **i** in the irregular mesh as well as other elements with which element **i** has come into contact during the crash. **Type(i)** specifies the type of material of element **i**. **Force(i)** stores the force calculated during contact that will be applied to element **i**. **Position(i)** and **Velocity(i)** store the position and velocity of element **i**. Force, position, and velocity of an element are declared as type **real_vector** variables, each of which is actually formed by three double precision (8-byte) floating-point numbers representing a three dimensional vector. Assuming the integers are 32-bits (4-bytes)

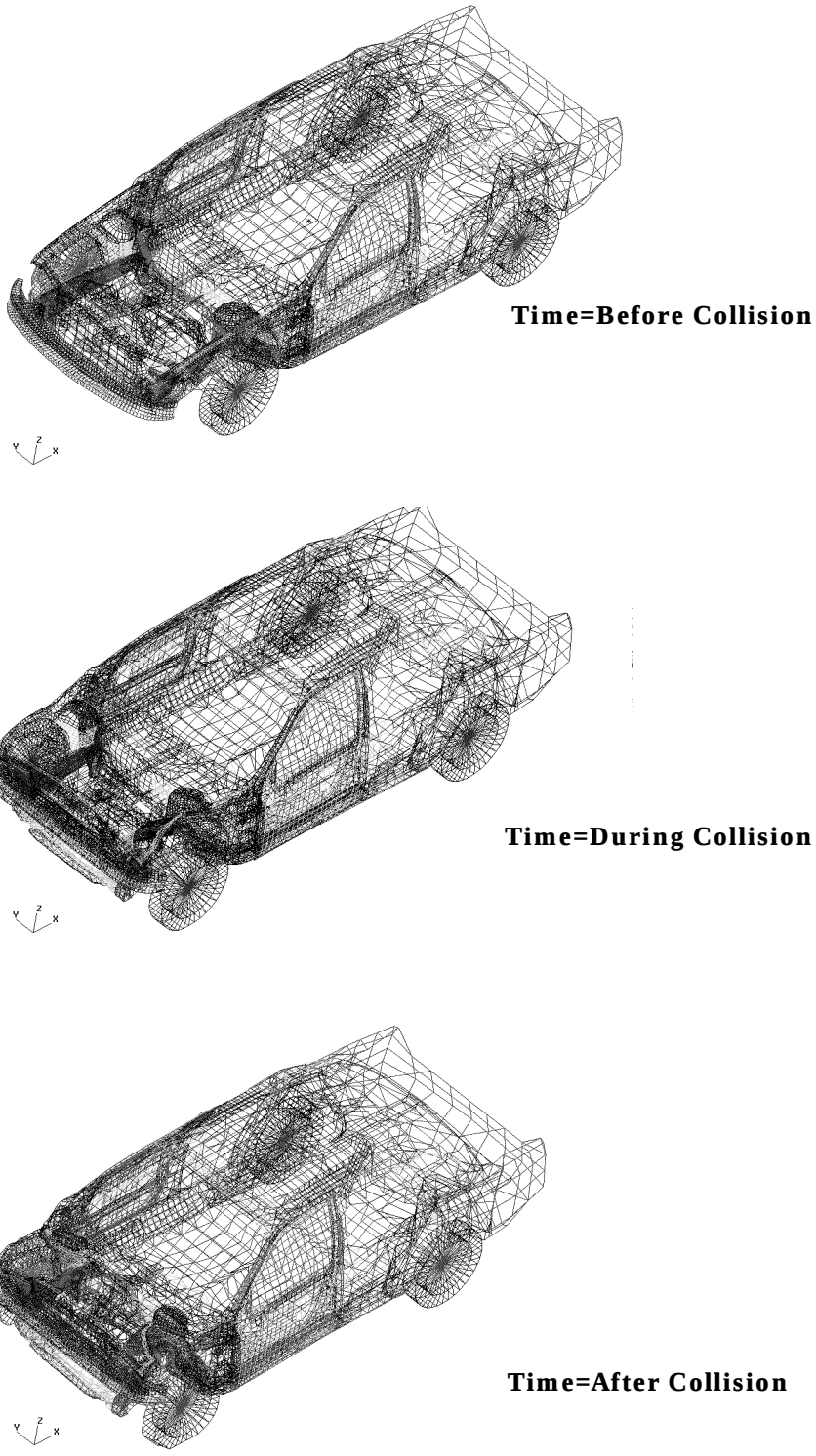


Figure 1-3: Collision between the Finite-Element Mesh and an Invisible Barrier.

Vehicle Model	Num_Elements	Max_Neighbors_ per_Element	Working Set (Mbytes)
Small	10000	5	1
Large	100000	10	12

Table 1-1: Vehicle Models for CRASH.

each, the storage requirement, or the *total data working set*, for CRASH is about *(84+4*Max_Neighbor_per_Element)*Num_Elements* bytes. In this dissertation, we assume two different vehicle models; their properties are shown in Table 1-1.

The program calculates the forces between elements and updates the status of each element for each cycle. In the first phase, the *Contact phase*, the force applied to each element is calculated by calling **`Contact_force()`** to obtain and sum the forces between this element and other elements with which it has come into contact. In second phase, the *Update phase*, the position and velocity of each element are updated using the force generated in the contact phase. Depending on the type of material, the Update phase calls **`Update_Plastic()`** or **`Update_Elastic()`** for updating the position and velocity as a function of **`Force(i)`**. Each cycle thus outputs a new finite-element mesh which is used as input to the next cycle.

This example program shows irregularities in several aspects. First, objects are represented by *unstructured meshes*. Second, in the Contact phase, properties of neighbor elements are referenced with *indirect array references*, e.g. **`Velocity(Neighbor(j,i))`**, referring to the velocity of the j-th neighbor of element i. Third, the load is *nonuniform* because the load of calculating the force for an element during the Contact phase depends on how many neighbors each element has, and the load of updating the status of an element during the Update phase depends on the type of element being updated.

1.3.3 Parallelization of CRASH

Even for codes as simple as CRASH, most parallelizing compilers fail to exploit the parallelism in CRASH because of the complexity of analyzing indirect array references, such as **`Velocity(Neighbor(j,i))`**, and procedure calls. The communication pattern needs to be explicitly specified for a message-passing version, yet determining the pattern is not a trivial

task. Fortunately, a shared-memory parallelization does not require the specification of an explicit communication pattern, and hence is initially much easier to develop. Performance tuning does, however, require some understanding of the communication pattern, in both cases.

The parallelism in CRASH can quite easily be recognized by a parallel programmer: the calculations for different elements within each phase can be performed in parallel, because they have no data dependence. Manual parallelization of CRASH can be implemented by parallelizing the major loop in each phase (indexed by i). A straightforward, simple parallel version of CRASH on HP/Convex Exemplar, *CRASH-SP*, is illustrated in Figure 1-4. Note that the *parallel directive*, **`c$dir loop_parallel`**, is inserted ahead of each parallel loop.

By default, loop_parallel instructs the compiler to parallelize the loop by partitioning the index into p subdomains: elements $\{1,2, \ldots \lfloor N/p \rfloor\}$ are assigned to processor 1, elements $\{\lfloor N/p \rfloor+1 \ldots \lfloor 2N/p \rfloor\}$ are assigned to processor 2, etc., where N is **`Num_Elements`** and p is the number of processors used in the execution. Since this parallelization partitions the domain into subdomains of nearly equal size, the workload will be evenly shared among the processors, if the load is evenly distributed over the index domain. However, for irregular applications like CRASH, this simple decomposition could lead to enormous communication traffic and poor load balance due to the unstructured meshes and nonuniform load distribution. More sophisticated domain decomposition algorithms are commonly used for partitioning the unstructured meshes so that the communication traffic is reduced [30].

1.3.4 Performance Problems

In this dissertation, we focus on solving major performance problems resulting from irregular applications because they pose more difficult optimization problems and lack an effective general method to guide the programmers toward achieving high performance. Once such methods exist, regular applications may be handled as a degenerate case. These problems are described in the next chapter and are addressed throughout this dissertation.

```
      program CRASH-SP

      integer Type(Num_Elements),Num_Neighbors(Num_Elements)
      integer Neighbor(Max_Neighbor_per_Elements,Num_Elements)
      real_vector Force(Num_Elements),Position(Num_Elements),
           Velocity(Num_Elements)
      real t, t_step
      integer i,j,Num_Neighbors,type_element

      call Initialization

c Main Simulation Loop
      t=0

c First phase: generate contact forces
c$dir loop_parallel
100   do i=1,Num_Elements
        Force(i)=Contact_force(Position(i),Velocity(i))
        do j=1,Num_Neighbors(i)
            Force(i)=Force(i)+Propagate_force(Position(i),Velocity(i),
              Position(Neighbor(j,i),Velocity(Neighbor(j,i))
        end do
      end do

c Second phase: update position and velocity
c$dir loop_parallel
200   do i=1,Num_Elements
        type_element=Type(i)
        if (type_element .eq. plastic) then
            call Update_plastic(i, Position(i), Velocity(i), Force(i))
        else if (type_element .eq. glass) then
            call Update_glass(i, Position(i), Velocity(i), Force(i))
        end if
      end do

      if (end_condition) stop
      t=t+t_step
      goto 100
      end
```

Figure 1-4: CRASH with Simple Parallelization (CRASH-SP).

1.4 Developing High Performance Parallel Applications

Often, performance tuning is called hand tuning, which emphasizes the programer's central role in most performance tuning work. While many people hope that someday compiler technology will automate most performance tuning work, most parallel programmers today are directly responsible for a large portion of the performance tuning work, which often requires extensive knowledge of the underlying hardware characteristics. Parallel programmers seem to spend more time poring over performance information and repeatedly modifying the source code in a cut-and-try approach, rather than simply deciding how best to tune the performance, and then simply doing it in one pass. In this section, we discuss this conven-

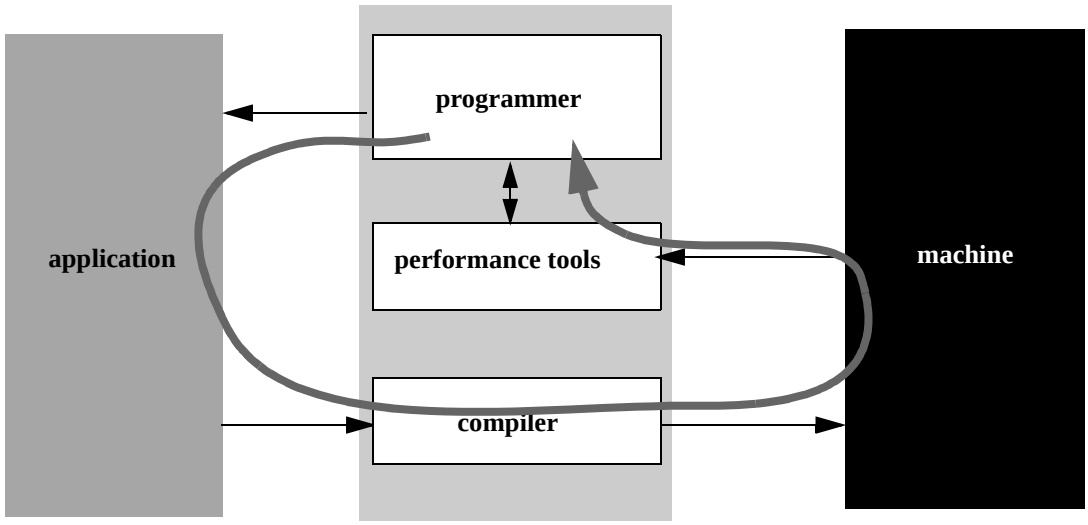


Figure 1-5: Typical Performance Tuning for Parallel Applications.

tional scheme, recent trends, current problems, and our approach for developing high performance applications.

1.4.1 Conventional Scheme

Figure 1-5 shows how performance tuning is typically done today. The compiler transforms the source code into an executable code with optimizations derived from source-code analysis and programmer written directives. Although, theoretically, compilers could accept feedback from the execution and use it when deciding whether and how to apply certain optimizations, such as during instruction scheduling [31][32], no effective feedback mechanism exists today in any compiler that provides a sufficient amount of information to guide its optimizations. Performance enhancement is the programmer's responsibility to carry out, primarily by revising the source code and modifying directives, perhaps with some help from performance assessment tools.

The hardware details of a computer are usually hidden from the user by commonly used programming models. Programmers may ignore many hardware issues that can greatly degrade a code's performance, even when substantial effort is spent varying the directives and flag settings for an optimizing compiler. As a result, hand tuning is subsequently involved to improve code performance, inefficient performance is accepted, or the parallel execution is

abandoned altogether. Under this conventional scheme, the effort to tune a parallel application, especially an irregular application, can be very time-consuming and/or ineffective. However, without proper tuning, peak performance on parallel computers, or even a respectable fraction of peak, can be very difficult to achieve. It should be no surprise that few tools can be used to facilitate the tuning, because for a long time, development of tools has been neglected or discontinued due to the typically short life times of parallel machines. Poor performance and painful experiences in performance tuning have greatly reduced the interest of many programmers in parallel computing. These are the major problems in a conventional parallel application development environment.

1.4.2 Recent Trends

Until the early '90s, some optimistic researchers believed that parallelization could and would soon be fully automated and fairly well optimized by future parallelizing compilers, and some believed that parallel machines would then be able to carry out parallel execution fairly efficiently with a minimum of hand tuning. Unfortunately, parallelization and performance tuning turned out to be more difficult than they thought, and none of those expectations have been satisfied. More and more researchers realize that (1) manual parallelization is necessary and should be properly supported, and (2) the key to develop high performance applications lies in a strong understanding of the interactions between the machine and the application, i.e. *machine-application interactions*. Given these awareness, the parallel computing community started several projects (e.g. the *Parallel Tools Consortium*, est. 11/93) to promote the development of standardized application environments and performance tools to ease programmer's burden and study machine-application interactions. Since then, more research has been focused on improving parallel tools, such as the following, to form a better application development environment:

- *Standard Parallel Languages or Message-Passing Libraries*, such as *HPF* [19] and *MPI* [20], which enable programmers to develop machine-independent codes and encourage vendors to optimize their machines for these languages. Among these new standards, MPI has been widely supported on almost all message-passing machines and even most shared-memory machines (e.g. HP/Convex Exemplar).

- *Parallelized and Tuned Libraries*, such as *Convex LAPACK* [33], which ease the programmers' burden by parallelizing and fine-tuning commonly-used routines. Many scientific/ engineering applications can benefit from the availability of such libraries.

- *Domain Decomposition Packages*, such as *Chaco* [28] and *Metis* [29], which provide various algorithms for decomposing irregular problem domains. With such packages, programmers are likely to save a considerable amount of time that would otherwise be spent writing and maintaining their own domain decomposition routines for irregular applications.

- *Source-code Analyzers or Visualizers*, such as *APR Forge* [34], which assist the users in parallelizing existing applications by revealing the data dependencies or data flow in the codes. Such tools directly assist the users, instead of attempting to accomplish automatic parallelization with a parallelizing compiler.

- *Performance Assessment Tools*, such as *CXpa* [35] and *IBM PV* [36], which assist the users in exploring the machine-application interactions in their applications. Performance assessment tools can be further categorized into three groups: performance extraction/ measurement, performance evaluation/characterization, and performance visualization. Performance extraction/measurement tools focus on finding specific events in the application and measuring their impact on the performance. Performance evaluation/characterization tools analyze the application performance, characterize the performance, and roughly identify the performance problems. Performance visualization tools provide a friendly graphics interface to help the users access and understand performance information.

Note that some of these parallel tools address the problems in parallelizing applications, while some others, also known as *performance tools*, address the problems which limit the application performance. Often, these two goals are both considered by one tool, since the way that the code is parallelized is highly correlated with the application performance.

1.4.3 Problems in Using Parallel Tools

Recent development of parallel tools has considerably improved the application development environment. However, current application development environments are still far from satisfactory. Typical problems that we have experienced are described below:

1. *Lack of Tools and Difficulty of Using the Tools:* While many tools have resulted from academic research, machine vendors have not been aggressively developing performance tools. According to the Ptools Consortium (http://www.ptools.org/): "Current parallel tools do not respond to the specific needs of scientific users who must become quasi-computer scientists to understand and use the tools".

2. *Lack of Tool Integration:* Combining the use of different tools can be painful and tricky. For example, feeding performance information from the CXpa to Metis requires a mechanism to convert the performance data output from CXpa to a form that is accepted as input to Metis. Due to the lack of integration, programmers still need to spend a substantial amount of time in interfacing the use of tools. Such routine work should be automated as much as possible, allowing the programmers simply to monitor the process and make high-level decisions.

3. *Unpredictable and Unreliable Tuning Methodologies:* Programmers have been relying for better or worse on their personal experiences and trial-and-error processes in their practice of performance tuning. The learning curve is tedious and time consuming with no guarantee of how much if any performance improvement to expect for a particular application. Furthermore, because the side-effects of one optimization applied to solve one problem can expose or aggravate other problems, there are many cases where performance problems cannot be solved individually.

4. *Unsuitable for Dynamic Applications:* For one application, performance tuning may need to be repeated for different input data sets and different machines. Although integrating the performance tuning mechanism into the runtime environment would ameliorate this problem, runtime performance tuning is difficult to implement because performance tuning is far from automatic. Most performance tuning tools, either cannot be, or are not integrated into the runtime environment.

1.4.4 The Parallel Performance Project

The *Parallel Performance Project* (PPP) research group was established at the University of Michigan (*http://www.eecs.umich.edu/PPP/PPP.html*) in 1992. The objective of our work is to develop and implement automated means of assessing the potential performance of applications on targeted supercomputers, high performance workstations and parallel computer systems, identifying the causes of degradations in delivered performance, and restructuring the application codes and their data structures so that they can achieve their full

performance potential. Many of the techniques presented in this dissertation are based on or motivated by some previous or concurrent work in the PPP group, including:

- *Experiences in Hand-tuning Applications*: We have worked jointly with scientists and engineers in developing high performance applications in various fields, including: (1) vehicle crash simulation with the Ford Motor Company, (2) ocean simulation for the Office of Naval Research, (3) modeling and simulation of ground vehicle with the Automotive Research Center (*http://arc.engin.umich.edu/*) for the U.S. Army (TACOM), and are continuing work of this kind on (4) antenna design and scattering analysis with the Radiation Laboratory at U. of Michigan (*http://www.eecs.umich.edu/RADLAB/*), (5) modeling soil vapor extraction and bioventing of organic chemicals in unsaturated geological material with the Environmental and Water Resources Engineering program at U. of Michigan (*http://www-personal.engin.umich.edu/~abriola/simulation.html*) for the U.S. Environmental Protection Agency, and (6) computer simulation of manufacturing processes of materials and structures with the Computational Mechanics Laboratory (*http://www-personal.engin.umich.edu/~kikuchi/*) at U. of Michigan. This ongoing work is supported in part by the *National Partnership for Advanced Computational Infrastructure* (NPACI) (*http://www.npaci.edu/*) of the National Science Foundation.

- *Machine Performance Evaluation:* The PPP group has developed performance evaluation methodologies (e.g. [37]) and tools (e.g. [38]) and has evaluated targeted high performance workstations (IBM RS/6000 [39][40], HP PA-RISC, DEC Alpha) and parallel computer systems (KSR1/2 [41][42], IBM SP2 [14][15], HP/Convex Exemplar [11][12][13], Cray T3D). The knowledge gained regarding target machines helps develop future machines and machine-specific performance tuning techniques [43].

- *Goal-Directed Performance Tuning:* A machine-application hierarchical performance bounds methodology has been developed to characterize the runtime overhead and guide optimizations (e.g. [44][45][46][47][48][49]). Major optimization techniques developed in the PPP group include data layout (e.g. [45][50]), restructuring data access patterns (e.g. [51]), communication latency hiding (e.g. [52]), instruction scheduling (e.g. [53][54]), and register allocation (e.g. [55][56]).

- *Domain Decomposition Techniques and Advanced Synchronization Algorithms:* Our experiences show that irregular problem domains commonly exist in many scientific/engineering applications. The PPP group has evaluated the performance of several domain decomposition packages, including Chaco [28] and Metis [29], for balancing the load in tar-

get applications. A profile-driven domain decomposition technique [30] and a multiple weight (multiphase) domain decomposition algorithm [2] have been developed for balancing non-uniformly distributed computations in single-phase as well as multiple-phase applications. Advanced synchronization algorithms, such as fuzzy barriers [57] and point-to-point synchronization [58], have been studied and implemented to improve the load balance in target applications.

These techniques, together with other available tools, such as CXpa, CXtrace, and IBM PV, constitute the current application development environment for our targeted machines. While these techniques and tools have helped our performance tuning work in the PPP group, the resulting application development environment still suffered from the problems that we mentioned in Section 1.4.3 and led to the goals of this research.

1.4.5 Goals of this Dissertation

In this dissertation, we discuss several new techniques that have been developed to address the weaknesses within this application development environment. More importantly, we present a unified approach to developing high performance applications by linking most of these known techniques within a unified environment. We believe that this unified approach will eventually lead to a well-integrated and fairly automated application development environment. The following statements describe the general goals of our research and the scope of this dissertation:

1. *Develop New Tools and Improve Existing Tools:* More techniques and tools are needed to characterize the machine and the machine-application interactions. The user interface of each tool should be designed so that it can be used by non-computer scientists with minimal knowledge about the underlying computer architecture.

2. *Develop a Unified Performance Tuning Methodology:* To speed up the performance tuning process and aim at making it more automated in the future, it would be best to develop a systematic method for selecting among optimization techniques and applying them in a proper order.

3. *Provide an Environment for Programmers and Computer Architects to Work Jointly in Developing Applications:* Programmers often possess an intimate knowledge of the inherent behavior of the underlying algorithms of their codes, while computer architects are

generally more familiar with machine characteristics. For tuning a large application, direct communications between these two groups of people are generally both time-consuming and inefficient. We thus need an environment that helps programmers expose their application behavior to computer architects.

4. *Incorporate Runtime Performance Tuning within Applications*: For optimizing applications whose behavior may not be discoverable at compile time, some performance tuning should be carried out during runtime. To achieve this goal, proper runtime performance monitoring support, well-integrated performance tools, and automatic performance tuning algorithms are necessary. Although we cannot satisfy this goal without the cooperation of computer manufacturers and software providers, this dissertation will lay the groundwork for satisfying this goal.

1.5 Summary and Organization of this Dissertation

In this chapter, we have overviewed parallel machines, parallelization of applications, common performance problems, and approaches to developing high performance parallel applications, and we have discussed the weaknesses in current application development environments. While most researchers focus on developing more parallel tools to help solve individual parallelization/performance problems, we aim at an integrated suite of existing and new tools and a unified approach to considering and solving the performance problems that they expose. In the following chapters, we discuss the techniques that we have developed and integrated to pursue these goals:

- *Chapter 2 Assessing the Delivered Performance* discusses the machine-application interactions in a parallel system, how machine performance characterization and performance assessment techniques can expose these relatively subtle interactions to help programmers visualize performance problems in their applications. We also present some innovative techniques that we developed for analyzing communication patterns in cache-based distributed shared-memory applications.

- *Chapter 3 A Unified Performance Tuning Methodology* summarizes our performance tuning methodology: a unified, step-by-step performance tuning approach that logically integrates various useful techniques that may be used to solve specific performance problems in an irregular application. For each step, we identify certain key issues that concern the

performance tuning at that step, as well as specific tuning actions that are most appropriate to solving these issues. The interactions among these issues, and actions that are vital to effectively tuning the application, are clarified in this chapter.

- *Chapter 4 Hierarchical Performance Bounds and Goal-Directed Performance Tuning* further extends previously developed goal-directed performance tuning work to more completely characterize a parallel application. This new performance characterization scheme is partly automated on the HP/Convex Exemplar with the CXbound tool that we developed. We explain how the performance bounds and the gaps between successive bounds can be used in conjunction with the step-by-step performance tuning scheme discussed in Chapter 3 to tune an application more efficiently.

- *Chapter 5 Model-driven Performance Tuning* describes our approach to facilitating the communications between programmers and performance tuning specialists and speeding up the performance tuning process. We discuss how the application behavior exposed by the programmers and performance assessment tools can be integrated to form application models. These application models can be analyzed by the model-driven simulation tools that we developed to model the application performance. Model-driven analysis guides performance tuning specialists in tuning the application models and greatly reduces the number of tuning iterations that must be carried out explicitly on the actual application code. This model-driven performance tuning methodology thus serves as a means to shorten the application development time, and hopefully in the future, to tune the code dynamically during runtime.

- *Chapter 6 Conclusion* concludes this dissertation by summarizing the key contributions of this research and their significance. Some topics of further research that may be investigated in the future are also presented.

CHAPTER 2. ASSESSING THE DELIVERED PERFORMANCE

In the world of scientific computing, *peak performance* is an upper bound on the performance of a computer system. However, users often find that there is a large *gap* between the peak performance and the performance that their applications actually see, the *delivered performance*. Without proper utilization of the hardware resources, applications can perform very inefficiently. As the peak performance of parallel computers is approaching TeraFLOPS (10 billion *FL*oating-*P*oint *O*perations per *S*econd), the computer architecture is increasingly complex, and the performance gap is growing. Current compiler technology makes an effort to close this gap, but still leaves much to be desired, especially for parallel computing.

The performance gap between peak performance and delivered performance is a complex function of the machine-application interactions. The ability to observe detailed machine-application interactions is necessary in order to determine the primary causes of the performance gap, and subsequently resolve them by boosting delivered performance, reducing cost, and finally accepting the remaining gap. Various public domain and commercial performance assessment tools have been developed for gathering, analyzing, and visualizing application performance. However, we have not seen any single tools or tool suites that provide complete performance assessment at various needed levels of detail. In most cases that we have encountered during our practice of performance tuning, we have had to find ways to integrate the use of existing tools and often develop new tools in order to gain a sufficient assessment of the delivered performance, relative to the aspects shown in Figure 2-1.

This chapter focuses on analyzing machine-application interactions for parallel applications. While the performance of a modern processor is relatively complex, the performance of a parallel system, with the addition of communication and synchronization, is even more mysterious for most users. For a user to gain a sufficient understanding of the performance of an application, the target machine must be well-characterized. For tuning the applications, the

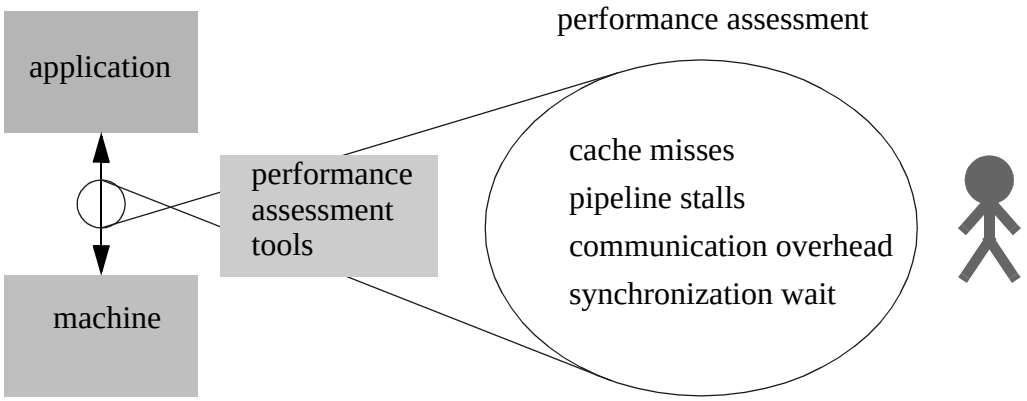


Figure 2-1: Performance Assessment Tools.

characterization of the target machine forms a basis by providing the target machine's performance metrics and performance constraints. In Section 2.1, we discuss approaches to characterizing the machine performance.

In Section 2.2, we categorize the machine-application interactions and discuss their correlation with common performance problems. Section 2.3 categorizes existing performance assessment techniques and discuss how they apply to observe specific machine-application interactions and trace the performance problems in the application. Our research on efficient communication performance analysis using trace-driven analysis is described in Section 2.4.

2.1 Machine Performance Characterization

A well-characterized machine makes it less difficult for programmers to develop high performance applications on the machine. A good machine performance characterization generates useful information in two respects: (1) The characterization serves to identify *performance metrics* that are important for modeling application performance on this machine. Performance metrics, such as memory access time and communication latency, provide the basis for modeling the machine and thence for quantitative analysis of application performance. (2) By analyzing the performance metrics, one may gain a sufficient understanding of the machine performance and develop certain *guidelines* on how to utilize the machine effectively.

Processor clock rate/cycle	100 MHz / 10 ns
Peak floating-point performance	200 M FLOPS
Processor cache size (data or instruction)	1 MByte
Processor cache line size	32 Bytes
Processor cache associativity	1
Processor cache access latency (data or instruction)	~ 10 ns
Number of processors, per hypernode	8
Local hypernode memory access latency	~ 500 ns
Processors-to-hypernode memory interconnection bandwidth	1.25 GByte/s
Remote hypernode memory access latency	~ 2000 ns
CTI bandwidth, per ring	600 MByte/s
Hypernode memory line size	64 Bytes

Table 2-1: Some Performance Specifications for HP/Convex SPP-1000

Number of Hypernodes	4
Hypernode memory size, per hypernode	1 GByte
Hypernode-local memory size, hypernode 0	578 MByte
Hypernode-global memory size, hypernode 0	272 MByte
Hypernode-local memory size, hypernode 1-3	322 MByte
Hypernode-global memory size, hypernode 1-3	528 MByte
CTIcache size, per hypernode	166 MByte

Table 2-2: The Memory Configuration for HP/Convex SPP-1000 at UM-CPC

There are a few ways that one might obtain a performance characterization of the target machine:

1. *Manufacturer/administrator's specifications:* The machine specifications from the manufacturer usually contain basic performance information. Some useful information also comes from the system administrator, as the configuration of a particular machine may vary. Table 2-1 and Table 2-2 list some major performance metrics of the HP/Convex SPP-1000 as documented in [8] and the memory system configuration of the machine at the Center of Parallel Computing at the University of Michigan.

Transfer rate, read from processor cache	800 MByte/sec
Transfer rate, write to processor cache	400 MByte/sec
Local-hypernode memory access latency, read	55.4 cycles (average)=554ns
Local-hypernode memory access latency, write	63.3 cycles (average)=663ns
Local-hypernode memory access bandwidth, read	52 MB/sec
Local-hypernode memory access bandwidth, write	50 MB/sec

Table 2-3: Microbenchmarking Results for the Local Memory Performance on HP/Convex SPP-1000

Distance	Cached?	Latency (ms)	Transfer Rate (MByte/sec)
Near	No	1.3	23.5
	Yes	2.0	15.4
Far	No	9.1	5.0
	Yes	10.0	4.6

Table 2-4: Microbenchmarking Results for the Shared-Memory Point-to-Point Communication Performance on HP/Convex SPP-1000

2. *Microbenchmarking:* Microbenchmarking [12][59] characterizes the performance of the target system with small specific *microbenchmark* programs that are designed to isolate and characterize primitive operations in the system that commonly lie on the critical path. Various microbenchmarks have been developed to characterize the performance of particular operations or help decide on machine configurations. Results from microbenchmarks simulate the performance of primitive operations delivered by the system, including the required machine operations and related software overhead. Tables 2-3 and 2-4 list some of the performance metrics of local memory and the shared-memory communication on HP/Convex SPP-1000 that have been characterized by using microbenchmarks [12]. Note that the shared-memory performance is a function of the distance between the requesting processor and the physical memory location that is accessed. Highest shared-memory performance is attainable only if the data is present in a near memory (in the same hypernode as the requesting processor) but not cached in any processor's data cache. Acquiring data from a far memory across the CTI ring has a latency of at least 9.1 ms. Accessing a cached block requires an extra 0.7-0.9 ms for flushing the block back to the shared-memory and modifying the cache status.

3. *Benchmark suites:* Benchmark suites, such as *SPLASH/SPLASH-2* [60][61] and *NAS Parallel Benchmarks (NPB)* [62] have been developed to resemble the workload of combinations of commonly used routines (also known as *kernels*, e.g. FFT) and full applications. The characteristics of benchmarks in SPLASH-2 vary in concurrency, load balance, working sets, temporal locality, communication-to-computation ratio, communication traffic, spatial locality, and false sharing[1]. In [61], Woo et. al. show the use of SPLASH-2 for characterizing a target machine qualitatively, by observing how the machine behavior changes across different benchmarks.

4. *Synthetic workload experiments:* Methods for synthesizing parameterized, controllable workloads have been developed to resemble various application workload in a parameterized space [41][37][63]. In [41], a sparse matrix multiplication program is used to generate synthetic workloads for evaluating communication performance on the KSR1. This workload generation is controlled by a few basic parameters: *the number of processors*, *the average number of point-to-point communications per processor*, *the degree of data sharing*, and *the computation-to-communication ratio*. With this approach, the communication performance of the KSR1 was characterized by varying these four parameters.

Among these four methods, machine specifications and microbenchmarking directly characterize the target machine quantitatively in terms of performance metrics, while benchmark suites and synthetic workload experiments tend to characterize the machine qualitatively in terms of its suitability for specific modes of usage. From the performance metrics, experienced computer architects may notice weaknesses in the performance of the machine. Some qualitative characterizations of the HP/Convex SPP-1000, which should be familiar to the application developers, are as follows:

- From Table 2-1, one may notice that memory access latency varies from 10ns (hit in processor cache), to 500ns (miss in the processor cache, but found in the local hypernode memory), to 2000ns (miss in both the processor cache and the local hypernode memory, but found in a remote hypernode memory). Since the latency differs so dramatically, cache utilization should be a serious concern for achieving good application performance. The cache and memory configurations in Table 2-1 and Table 2-2 are vital for applying data layout techniques to improve the cache utilization.

1. False-sharing is further discussed in Section 3.3.

- From Table 2-4, we notice that the communication performance is highly sensitive to the distance of communication. Near communications, which take place within a hypernode, are about 4 to 7 times faster than far communications. Therefore, the programmer should try to reduce the number of far communications. In fact, many Exemplar programmers have problems scaling their applications beyond 8 processors due to this high far communication cost.

- In [12], Abandah and Davidson approximate the synchronization time required for synchronizing p processors with a barrier synchronization by:

$$T_{sync}(p) = 7.1\ p^2\ microseconds,\ for\ p \neq 8, \qquad \text{(EQ 1)}$$

  and they pointed out that the implementation of the barrier is particularly inefficient for 8 processors, which requires at least 3 times longer (> 1500μs) than synchronizing 9 or 10 processors. This characterization not only models the performance of barrier synchronization, but also reveals a weakness in the machine that may seriously affect an application's performance.

2.2 Machine-Application Interactions

In this section, we categorize common machine-application interactions, discuss how these interactions affect the application performance, speculate as to the performance problems that they may cause, and point out previous work and future directions for solving these problems.

2.2.1 Processor Performance

Here, we consider the performance of individual processor subsystems with private caches and/or private memory. Low processor performance is often the most serious performance problem even for parallel codes on parallel systems. Yet this problem is surprisingly often ignored by machine designers and users of parallel systems. Processor performance is most critical to those applications whose performance scales poorly with the number of processors. It is also important for efficient use of machine resources and for approaching potentially deliverable performance even when performance gains can be made by increasing the number of processors.

The current trend in processor design is to discover and exploit more of the *instruction-level parallelism* (ILP) in the application. With multiple pipelines, today's high performance processors can exploit a high degree of ILP (6-8 instructions per cycle as of 1997) by executing many instructions simultaneously. To fully exploit the performance capability of these processors, it is necessary to:

1. *Provide sufficient bandwidth and low latency for instruction and data supply:* The performance of the memory system is important for supplying enough instruction and data supply bandwidth. Memory systems are organized in a hierarchical fashion with faster, but costlier caches placed between the processor and main memory to reduce average latency and control cost, by exploiting data locality. Even relatively few cache misses can seriously degrade the performance of a highly concurrent processor.

   As we mentioned in Section 2.1, a processor cache miss on HP/Convex SPP-1000 requires a latency of more than 50 instruction cycles, and about 200 cycles if the requested data resides in a remote hypernode memory (see Table 2-1). Efficient utilization of caches is thus a key issue for developing applications on this machine. In fact, the performance of CRASH suffers seriously due to cache misses. In each phase, CRASH iterates through every element in its finite-element mesh. When the working set is greater than the capacity of the processor cache, the processor cache has to reload the working set for every iteration. With 32-byte cache lines, the miss ratio is at least 0.25 while loading 8-byte double precision floating-point numbers. Utilization of instruction cache can also be a problem for large application codes.

2. *Expose sufficient instruction-level parallelism in the applications:* Ideally, the compiler and/or the scheduler in the processor should exploit the parallelism in the instruction stream. In practice, they often fail to do so, even when there is abundant ILP in the application. In contrast with runtime hardware schedulers, compilers should be able to perform more sophisticated and thorough source-code analysis for exploiting ILP, but they are weak when memory references and program flow are ambiguous. Although runtime schedulers operate after some of this ambiguity has been resolved, they cannot find the parallelism between distant instructions.

   Compilers may have difficulty exposing the ILP in CRASH, due to the ambiguity in indirect array references. Fortunately, this is a minor problem as far as the overall performance is concerned, because most of the runtime is spent in the subroutines whose memory references are not ambiguous.

3. *Optimize the instruction schedules:* Given adequate ILP and instruction/data supply, a processor approaches its peak performance by maintaining a sufficient number of independent instructions in flight within each critical functional unit. For real applications, in light of limited ILP and imperfect instruction/data supply, optimizing the instruction schedules can be very difficult. First, the complexity of finding an optimal schedule grows quickly as the number of instructions that are considered for scheduling increases. Second, accurate information regarding the code's cache behavior, data dependencies, and control flow are needed for optimizing the schedule, which is an area in which performance tools can help.

   For a large application code, it is helpful to identify the code portions that deserve more elaborate effort from the compiler. The code schedule can be characterized for this purpose, and then fed back for recompilation. Without proper tools, it is extremely difficult to characterize the schedule.

A failure in any of the above three categories can easily degrade the processor performance. Observing the application's behavior in terms of the performance impact of cache misses, degree of ILP, and inefficient scheduling will help to pinpoint these processor performance problems.

Various well established *optimizing compiler techniques* for uniprocessors can help optimize the processor performance. For example, loop unrolling, software pipelining, and trace scheduling can be used to expose ILP [64][31], data prefetching can be used to hide memory access latencies [52][65], loop blocking can be used to reduce cache capacity misses [66][67], and data layout methods can be used for reducing conflict misses [45]. *Performance information* is useful for clarifying the ambiguity of data and control flow, and is widely used in the areas of hand-tuning of programs [68], trace scheduling [31], superblock scheduling [32], data preloading [69], branch prediction [70] and improving instruction cache performance [32]. *Hierarchical machine-application performance bounds* models have been developed that also exploit performance information to analyze the performance of scientific codes and suggest where, in what respect, and how much improvement may be possible [44][45][46][47][48][49]. This performance bounds methodology is further developed in Chapter 4.

2.2.2 Interprocessor Communications

Communications between processors need to be performed efficiently, in the light of *latency* and *bandwidth* limitations. Communications impact the performance of a parallel program as the execution is delayed due to communication latency and/or network contention. Proper organization of communications is important to reduce the communication latency, enhance overlap, and keep the required bandwidth well below the bandwidth limitations of the system.

Communications are explicitly specified by the programmers in message-passing codes. Organizing the communication patterns properly is often a challenge for the programer. Poor communication performance can result from:

1. *Improper scheduling of communications* can result in an uneven distribution of communications which can cause network contention and waste bandwidth; communications may be serialized in some parts of the network due to contention, while bandwidth in the other parts may be wasted. Accurately scheduling the communication tasks in the code requires information regarding the execution time of computation and communication tasks.

2. *Improper domain decomposition* can cause excess communications. To select a better domain decomposition algorithm, the domain's geometrical properties and data dependencies among the elements should be well-analyzed, which is usually a difficult process since it requires knowledge of the application as well as the available domain decomposition algorithms.

Shared-memory programming helps relieve the programmers of having to explicitly specify communications in their codes. However, the actual communication patterns are then only implicit and hence more difficult to identify and analyze. Although application developers like to view the shared memory as a unified centralized memory where multiple processors can access any memory location with some fixed latency (as in the Parallel Random Access Machine, *PRAM*, model [71]), in scalable shared-memory machines (KSR1, HP/Convex Exemplar, Cray T3D, SGI Origin, etc.), the shared memory is distributed over the system and the communication latency can be highly unpredictable. As a result, shared-memory programmers or compilers need to expose the hidden communication patterns in order to solve performance problems due to memory access latency.

The use of coherent caches complicates the analysis of communication patterns for a shared-memory application. The cache coherence protocol tends to move or replicate data to caches near the processors that have recently requested the data. This is done automatically by hardware on the HP/Convex Exemplar, the KSR1, and many other modern DSM machines. Sometimes, the presence of caches degrades the application performance by causing superfluous communication traffic due to false-sharing and unnecessary coherence operations (further discussed in Section 3.3). To address such problems, we would like to characterize and control the data movement and the resulting coherence operations (invalidates, updates, etc.), collectively referred to as *memory traffic*, in the application. However, exposing the memory traffic for CRASH-like irregular applications with source code analysis is very difficult.

Better decomposition of the problem domain is a key to reducing communication traffic, and this has been a consideration in most domain decomposition packages. The *cache coherence protocol* can have a strong impact on the performance of a *distributed-cache* shared-memory system [72][73][74][75]. Choosing an appropriate cache coherence protocol is an important issue in machine design, but has not yet become a fine-tuning option that end users can vary statically for particular data structures or dynamically in time within their particular applications. In addition, *prefetching/pre-updating* techniques have been developed for hiding communication latencies in shared-memory codes [52]. *Asynchronous (nonblocking) communications* can be used for overlapping communication and computation in message-passing codes. *Array grouping* can change the data layout to improve the efficiency of communications by reducing superfluous information within communication transactions and the false sharing that it can cause [45]. The communication patterns in message-passing applications can be traced by instrumentation/visualization tools such as PICL/Paragraph [76][77], AIMS [78] and CXtrace [79]. However, exposing communication patterns in sufficient detail for tuning shared-memory applications is relatively time consuming and was not available before this dissertation work, which is further addressed in Section 2.4.

2.2.3 Load Balancing, Scheduling, and Synchronization

Even with abundant exposed parallelism, the degree of parallelism achieved in an execution may be limited by the load imbalance and scheduling constraints in the execution. The performance of a parallel region is limited by the processor with the most work, and the

degree of parallelism achieved is reduced if some processors are idle waiting for the slowest processor. Scheduling constraints often force some processors to be idle, waiting for the arrival of schedulable tasks.

Load imbalance affects the *degree of parallelism* (i.e. *efficiency*) achieved in a parallel execution. The performance of a parallel region is limited by the processor with the most work; the degree of parallelism is imperfect if some processors are idle waiting for the slowest processor. The metric

$$Load\ Imbalance = (T_{max}-T_{avg})/T_{avg} \quad \text{(EQ 2)}$$

is often used to describe the load balance of a parallel region, where T_{max} is the execution time for the processor with the most work and T_{avg} is the average execution time. An application using *p* processors will have an imbalance of 0 when it is perfectly load balanced or an imbalance of *p-1* when all the work is performed by one processor. The degree of parallelism is defined as

$$Degree\ of\ Parallelism = p*T_{avg}/T_{max} = p/(1+imbalance), \quad \text{(EQ 3)}$$

ranging from 1 (total serial) to *p* (fully parallel).

Synchronization/scheduling overhead reduces the efficiency of a parallel execution. Synchronization overhead includes the latency for performing the synchronization operations (*synchronization costs*) and the idle time that processors spend waiting at the synchronization points (*synchronization idle time*). Scheduling overhead includes the latency for scheduling tasks onto processors (*scheduling costs*) and the idle time that processors spend waiting for their tasks to become ready for scheduling (*scheduling idle time*). For loop-based applications, a large load imbalance often results in long synchronization idle time, due to the need to synchronize with the most heavily loaded processor. Inefficient scheduling can cause unnecessary scheduling idle time due to unnecessary blocking of tasks.

Scheduling constraints are imposed by the application to ensure correct execution, usually enforced by various forms of synchronizations. Scheduling constraints enforce a proper ordering of memory references and instruction executions to satisfy the data and control dependence in the code. Some synchronization operations are required for correct execution, but some forms of synchronization, such as barriers, often impose excessive scheduling con-

straints, which result in extra synchronization overhead. The parallelization in CRASH-SP (Figure 1-4) implicitly employs barriers to isolate two individual phases, which illustrates such a case. For multiple-phase applications like CRASH, if the load is distributed differently for different phases, finding a decomposition that balances the load individually for each of the multiple phases is more difficult to determine.

Some schemes, such as *fuzzy barriers*, have been proposed to relax barrier synchronizations by scheduling unrelated ready tasks whenever a wait occurs [80][81]. Point-to-point synchronizations reduce synchronization wait time by minimizing scheduling constraints. Long idle times are often due to *load imbalance*, in which case domain decomposition techniques can be used to better balance the load by adjusting the sizes of the sub-problems handled by individual processors. *Static* balancing techniques are most efficient for problems whose load distribution does not change during the runtime. *Dynamic* balancing techniques that monitor and periodically rebalance the load as the run progresses can be used effectively within programs that exhibit dynamic load behavior. For better load balance, an accurate performance assessment of the load distribution is needed [30]. *Multiple phase* load balance problems, where multiple parallel regions are divided by synchronization operations in a program, are more difficult to solve; the load must be balanced in each such region by either a static multi-objective decomposition [2] or by dynamically rebalancing as each new and different phase is entered.

Measuring load imbalance is discussed in Section 2.3. Load balancing techniques are further discussed in Sections 3.5-3.8.

2.2.4 Performance Problems in CRASH-SP

Here we summarize the performance problems related to running CRASH-SP on the HP/Convex SPP-1000/1600, based on the previous discussions:

1. Parallelizing this program by partitioning the loop indices (`i`) normally results in poor *load balance* and *excessive communication overhead*. A graph-partition algorithm is required to decompose the unstructured finite-element meshes.

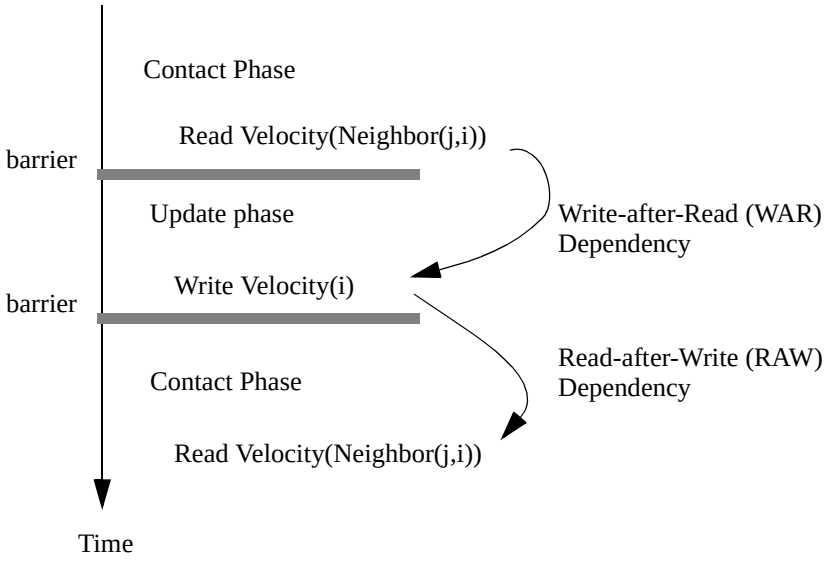


Figure 2-2: Dependencies between Contact and Update

2. The *communication pattern* needs to be explicitly specified for a message-passing version, yet determining the pattern is not a trivial task. While a shared-memory version does not require an explicit communication pattern, analyzing its communication overhead is more difficult.

3. In order to balance the load among the components of a partition, performance assessment is needed to extract the *load* associated with individual elements of various types in the finite mesh, for each of the phases in the application code.

4. Orchestrating the code in two phases, which appears to have no impact on serial performance, can significantly degrade parallel performance, e.g. by doubling the number of cache capacity misses if the *data working set* is much larger than the cache size. Note that the 1MB processor cache on HP PA-RISC7100, although quite large as a first level cache, is relatively small, compared to the desired data set size of many parallel codes, such as CRASH and NPB[62].

5. Loop fusion cannot be applied to help combine the workload or reduce capacity misses for this example, because execution of one phase cannot start until the other is completely executed. As shown in Figure 2-2, array **`Velocity`** is read in the *Contact* phase and written in the *Update* phase, which creates *Write-After-Read* (WAR) and *Read-After-Write*

(RAW) dependencies between these two phases. Since **`Velocity`** is referenced indirectly in the Contact phase, it is difficult to determine when an arbitrary array element, e.g. **`Velocity(k)`**, is no longer referenced in the *Contact* phase. Therefore, some sort of mechanism, such as a synchronization barrier, is used to conservatively enforce all these potential data dependencies.

6. Since the *Contact* phase and the *Update* phase cannot be fused, they are orchestrated in the code as two phases. It is generally difficult to balance two program phases with one static domain decomposition, because their load distributions can be different; however, use of two different domain decompositions can result in excessive communication between phases.

7. As the program's iterations proceed, the load distribution may vary considerably as contacts occur (and cease) during the runtime. *Dynamic load balancing* techniques should be considered in this case.

2.2.5 Overall Performance

The performance problems in CRASH are *interrelated*, it is almost impossible to solve one problem without affecting the other problems. A complete performance tuning scheme should consider them jointly, so as to optimize the *overall performance*, which makes optimization of parallel applications *complex*. For example, changing domain decomposition affects both communication traffic and load balance in the application. To efficiently optimize the overall performance, a *complete* and *global* performance assessment is often necessary for identifying performance problems and orchestrating performance tuning actions.

In the next section, we discuss various performance assessment techniques for characterizing a wide range of performance problems as well as for exposing detailed machine-application interactions. In Chapter 3, we optimize the overall performance by carefully characterizing the interrelationship among major performance problems and suggest a logically-ordered sequence of performance tuning steps. In Chapter 4 and Chapter 5, we use high-level performance assessment and program abstraction respectively to reduce the complexity of optimizing the overall performance.

2.3 Performance Assessment Techniques

In this section, we discuss performance assessment techniques for characterizing the performance problems as well as exposing machine-application interactions. Performance assessment techniques are categorized into three basic types: source code analysis, profile-driven analysis, and trace-driven analysis. We discuss how these different types of tools can be applied to characterize different performance problems.

2.3.1 Source Code Analysis

Source code analysis, including the use of compilers, is useful for extracting an application's high-level semantics that may be lost after compilation. Most parallelizing compilers perform limited data and control dependency analysis to parallelize the code and attempt to optimize its performance. Some attempt to optimize without such knowledge simply by trying a series of cut-and-try options, which is usually very tedious and achieves only mixed results. Users often need to assess the code performance independently in order to decide how to set the many compiler switches appropriately for good optimization. Even when the requisite knowledge is obtained, the compiler switches are often defined at too coarse a granularity for effective control of the process.

Besides conventional compilers, pre-compilers (a.k.a. pre-processors or source-to-source translators), such as *Parafrase-2* [23] and *APR Forge* [34], specialize in data and/or control flow analysis and are designed primarily to assist manual parallelization by exposing the results of their analysis to the users. Some of them even interact with the users by providing a visualization the results and accepting user commands using graphical user interfaces.

Nevertheless, difficulties in analyzing indirect references and procedure calls often limit a source code analyzer's ability to pursue global analysis of the application. Global events, such as shared-memory accesses, communications, and synchronizations, often interact with the application beyond the scope of the loops, the procedures, or even the processors where they reside. As a result, current source code analyzers are fairly ineffective in providing vital performance assessment for irregular parallel applications.

An experienced programmer can, with sufficient effort, carry out a source code analysis quite effectively, as is done in practice by hand-tuning today. People, especially the code authors, seem to be able (with practice) to learn how to envision their high-level codes and data structures relatively well in cases where most compilers fail. For example, it should quite easy for a programmer to examine the procedure **`Contact_force(Position(i), Velocity(i))`** in our simple CRASH example and verify whether there might be any side-effects within the procedure, such as when referencing variables other than **`Position(i)`** and **`Velocity(i)`**.

Our performance tuning experience in the Parallel Performance Project shows that the knowledge, insights, and statements of purpose from a code's author are extremely useful when carrying out source code analysis. It is doubtful that this sort of human *art* can be eliminated from performance tuning at any time in the near future. We believe, however, that such human effort should be further assisted, utilized, and integrated into the application development environment. In Chapter 5, we show how our performance modeling methodology addresses such issues.

2.3.2 Profile-Driven Analysis

A profiling tool, or profiler, counts the occurrence of specific performance-related events in the application. These counts, also known as performance metrics, provide a quantitative characterization of certain aspects of the application performance as well as a description of application behavior which might be useful for analyzing the application. For example, the cache miss ratio generally characterizes the utilization (efficiency) of a cache. Communication latency is essential to characterization of the communication performance. The runtime distribution over the processors in each program region is a good indication of load balance. The control flow profile provides the user with the execution counts for loops and subroutines.

Hardware-only profiling was intended for evaluating the machine performance by monitoring the machine operation at the system component level [82]. A hardware-only approach, however, is improper for application development since it is not controllable by software, nor can the profiles be associated with the application in a way that exposes machine-application interactions. *Software-only* profilers, such as *gprof* [68] and *QPT* [83], are capable of collecting the control flow and the execution time profile at some coarse resolution by a user-defined

instrumentation of the application. While such software-only profilers are relatively machine-independent, they are of limited use because (1) the code instrumentation may be intrusive to the application performance, (2) some important, but detailed machine operations, such as cache misses, operation system activities, and library routines, generally cannot be profiled, and (3) the profiling process is usually time-consuming, since the profile data are collected by software via interrupting the execution of the target application.

It has become a trend in processor design to support hardware performance monitoring (event counters), such as KSR's *PMON* [84] and HP/Convex's *CXpa* [35]. Using the hardware performance monitors, profiling software can selectively probe some detailed machine operations with less intrusiveness to the application, yet still provide a user-controllable profiling process and expose machine-application interactions. Such *hybrid* approaches tend to provide the most accurate, detailed, and efficient profiling process today.

Profiling is an important skill to master for characterizing application performance. Recent profilers, such as *CXpa*, are quite useful and user-friendly for this purpose. In this section, we illustrate the role of profilers in application development with *CXpa* as a case study. After this case study, we discuss the weaknesses and the desired features of profiling tools.

2.3.2.1 The Convex Performance Analyzer (CXpa)

The *Convex Performance Analyzer (CXpa)* is a HP/Convex software product for profiling application performance on HP/Convex Exemplar machines. *CXpa* is capable of profiling performance for selected *routines* and *loops* of a shared-memory or message-passing program written in *C* or *Fortran*. Thanks to the performance monitoring hardware embedded in the HP PA-RISC 7100/7200 processors, *CXpa* gives accurate profiles with very low interference to the regular execution. Detailed user references about *CXpa* can be found in [35].

Performance Metrics for the Application

Use of *CXpa* basically consists of three major stages: *instrumentation*, *profile collection*, and *profile visualization/report*. First, using a compiler directive, **`-cxpa`**, the users can instruct the compiler to instrument their programs. Then, execution of a *CXpa-instrumented* program generates a performance profile (**`*.pdf`**). Finally, the users can review the profile

with a utility called **cxpa**. Advanced users can use the **cxpa** utility to select the performance metrics and the portions of the code that need to be profiled.

CXpa can instrument the following program regions: (1) routines, (2) loops, (3) parallel loops, and (4) basic blocks. Profiling basic blocks is incompatible with profiling the other program structures in the same run. Loops that are automatically parallelized by the compiler or parallelized by the programmer using compiler directives are *parallel regions* that are recognized by *CXpa*, while other parallel regions that are formed by direct use of the parallel libraries (the *CPSLib*) may or may not be recognized by *CXpa*. For users who are interested in profiling library routines, pre-instrumented libraries are available and should be linked at compile-time.

For the HP/Convex Exemplar SPP-1600, *CXpa* provides the performance metrics listed in Table 2-5 for each user-selected *program region* (loop or routine). Performance assessment can be performed by identifying the most time-consuming program regions or other individual regions. Note that *local memory accesses* are the memory accesses for data not found in the processor cache that was found in local memory (the portion of memory allocated to that processor's hypernode), and *remote memory accesses* are the memory accesses for data not found in the processor cache that was found in a remote memory (memory in some other hypernode). Also note that while various memory access events can be profiled by *CXpa*, only one type of on-processor events and one type of off-processor events can be chosen for *CXpa* to profile within a run.

CXpa includes a performance visualizer that provides three different views for a profile: (1) text report, (2) 2-dimensional visualization, and (3) 3-dimensional visualization. Examples of the 2-D and 3-D views are shown in Figure 2-4. Below we discuss how the performance metrics collected by *CXpa* can be used to analyze the machine-application interactions in the target application.

Assessing the Processor Performance

The overall processor performance is indicated by the average *instructions per clock cycle* (IPC) or average *million instructions per second* (MIPS), calculated by

Wall Clock Time	The wall clock time measures the time duration from the beginning to the end of workload in a profiled program region.
CPU Time	The CPU time measures the execution time that each processor spends in performing computation and/or memory access (*not* including the time the processor spends in the OS routines or outside the profiled program).
Dynamic Call Graph	CXpa profiles the count and time for every caller-callee (a.k.a. parent-child) pair to generate a call graph or report. A call graph helps the user understand the distribution of runtime and pinpoint time-consuming program regions.
Execution Counts	The number of times that a profiled program region is executed.
On-Processor Event Counts and Latency	The number and latency of specific on-processor memory access events, which can be one of the following four categories: (1) executed instructions, (2) instruction cache misses, (3) data access count and cache misses, or (4) instruction and data TLB (translation lookaside buffer) misses.
Off-Processor Memory Access Event Counts and Latency	The number and latency of specific off-processor memory access events, which can be one of the following three categories: (1) local memory accesses, (2) remote memory access, or (3) both local and remote memory accesses. For each category, CXpa can be configured to profile: (1) read accesses, (2) write accesses, or (3) both read and write accesses.

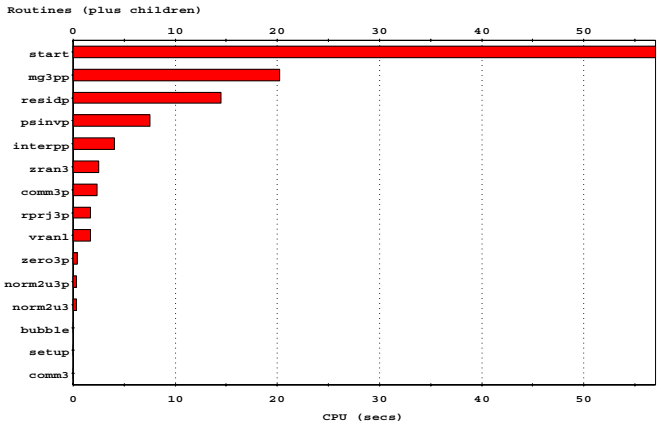
Table 2-5: Collectable Performance Metrics with CXpa, for SPP-1600

$$IPC = (Number\ of\ instructions) / (Execution\ time\ in\ cycles), \qquad (EQ\ 4)$$

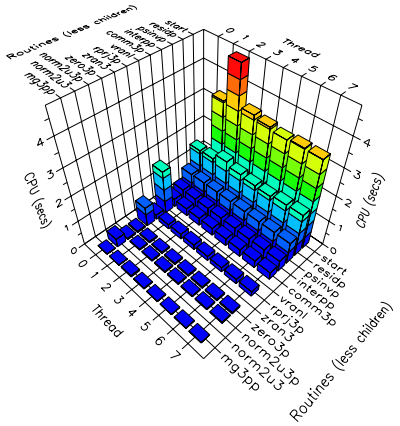
or

$$MIPS = (Number\ of\ instructions) / (Execution\ time\ in\ microseconds), \qquad (EQ\ 5)$$

where the number of instructions is obtained by configuring the on-processor counters to measure the executed instructions. A larger IPC (MIPS) indicates higher processor performance, where the maximum achievable IPC (MIPS) is 2 (240) on each 120MHz HP PA-RISC 7200 processor in the SPP-1600. Low IPC indicates inefficient processor utilization.



(a) Two-Dimensional Visualization



(b) Three-Dimensional Visualization

Figure 2-3: CPU Time of MG, Visualized with CXpa.

The on-processor counters can also be configured to monitor instruction or data cache misses. A large cache miss ratio and/or long cache miss latency indicates inefficiency in caching. The cache miss ratio for CRASH-SP reveals one problem that we mentioned in Section 2.2.4 for the large vehicle model (see Table 1-1) whose *data working set* is larger than the processor data cache (1MB). In this case, the miss ratio of the processor data cache is relatively high. Increasing the number of processors from 1 to 8 does not reduce the number of cache

misses, instead, we observe an increase in the average cache miss latency. The increase in the average cache miss latency is possibly due to the increasing degree of sharing and/or degree of memory contention. The high miss ratio and high cache miss latency appear to be major reasons that the processors deliver poor IPC, but the *CXpa* profiles alone do not suffice to determine the exact causes of the performance problems.

Assessing the Communication Time

The off-processor memory access counters can be configured to acquire one specific type of communication events in one profiling. The number and latency of *local memory accesses* characterize the *communications* together with the *processor cache misses* within each hypernode. Since cache misses and communications are usually treated differently in performance tuning, further characterization is necessary to differentiate between the overhead of these two. When the working data set is much greater than the processor cache, such as for the large vehicle model running on 1 to 8 processors, the communications are overshadowed by the cache capacity misses. In fact, the number of local memory accesses (cache misses) in CRASH-SP increases slightly while increasing from 1 to 8 processors.

The number and latency of *remote memory accesses* characterize the (long distance) communications among the processors in different hypernodes via the CTI rings. Running CRASH-SP on 9 or more processors results in a large amount of remote memory accesses, which indicates that the *domain decomposition* in CRASH-SP performs poorly. Longer average local (remote) memory *latency* per access indicates a higher degree of sharing or contention in the local (remote) memory system or the interconnection network.

Assessing the I/O and OS Time

For shared-memory parallel programs, memory access includes cache misses and coherence communications. Computation, memory accesses (including communication), I/O and OS events can be overlapped during the execution, as illustrated in Figure 2-4. The *wall clock time* on a processor measures the total elapsed time in these activities whereas the *CPU time* excludes the time that the processor spends exclusively in the I/O and OS routines, i.e. the CPU time is the wall clock time minus the time during which the processor is not performing

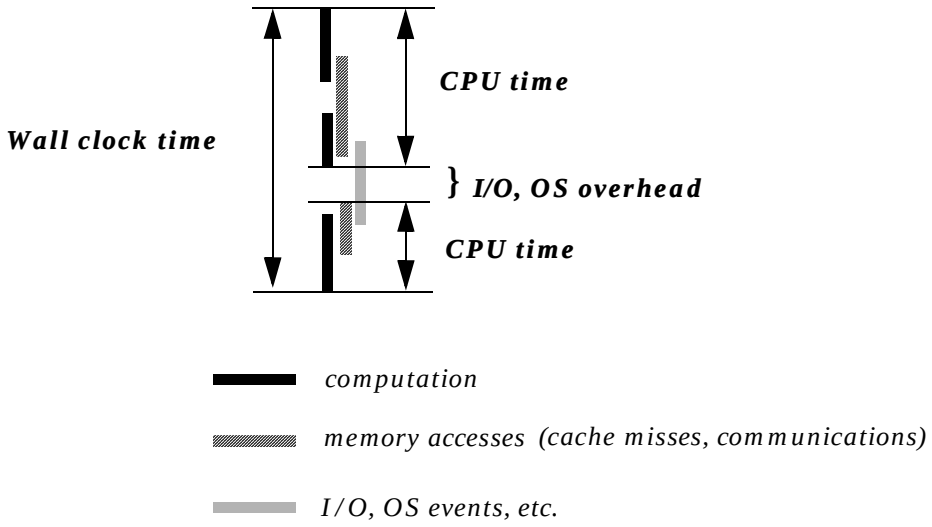


Figure 2-4: Workload Included in CPU Time and Wall Clock Time.

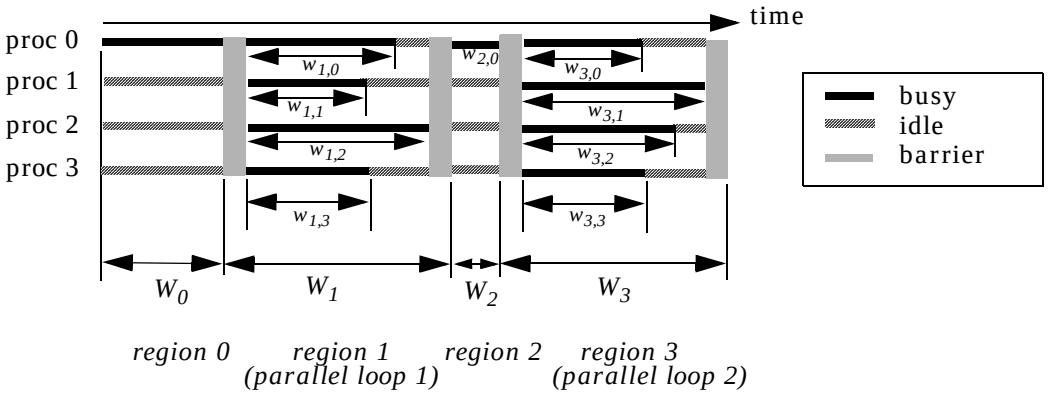
either computation or memory accesses. Wall clock minus CPU time can be used to characterize the overhead due to I/O and other OS events.

CRASH performs I/O sequentially only at the beginning and the end of the simulation. The I/O time is not a serious problem in CRASH, as it is relatively small compared to the computation time.

Assessing the Load Imbalance

On the HP/Convex SPP, when a parallel loop is executed, multiple *threads* are spawned and assigned to the processors. Barriers are mandatory to synchronize the processors at the beginning and at the end of the parallel loop. Between the barriers, each thread is essentially the same as the original sequential loop but the loop index values are partitioned among the processors. As illustrated in Figure 2-5, the wall clock time reported for each processor (e.g. $w_{1,0}$, $w_{1,1}$,...) in the parallel loop is the wall clock time required by that processor to perform its thread. The wall clock times of parallel loops (W_1,W_3) measure the time from the beginning to the end of the parallel loop, including the overhead of barrier synchronizations, i.e.

$$W_i = Max_p\{w_{i,p}\} + (Synchronization\ Cost\ in\ the\ Parallel\ Loop) \qquad \text{(EQ 6)}$$



W_r : Total wall clock time the processors spent on program region r.
$w_{r,p}$: Total wall clock time that processor p spent in program region r

Figure 2-5: Wall Clock Time Reported by the *CXpa*.

Some processors may be idle while waiting for the slowest processor. The amount of work performed in a parallel loop is indicated by the total wall clock time (busy time) on individual processors. For example, the total busy time in parallel loop 1 is $\Sigma_q\, w_{1,p} = (w_{1,0} + w_{1,1} + w_{1,2} + w_{1,3})$, and the processor utilization (efficiency) in this parallel loop can be characterized by

$$Efficiency = (total\ workload)/(total\ time) = (\Sigma_p\, w_{1,p})/(4*Max_p\{w_{1,p}\})\,, \qquad \text{(EQ 7)}$$

or

$$Degree\ of\ Parallelism = (total\ workload)/(wall\ clock\ time) = (\Sigma_p\, w_{1,p})/Max_p\{w_{1,p}\}\,, \quad \text{(EQ 8)}$$

or

$$Load\ Imbalance = (wall\ clock\ time - average\ workload)/(average\ workload)$$
$$= (4 * Max_p\{w_{1,p}\} - \Sigma_p\, w_{1,p})\ /\ (\Sigma_p\, w_{1,p})\,. \qquad \text{(EQ 9)}$$

Despite interpreting the performance slightly differently from one another, these metrics provide essentially the same performance assessment regarding the degree of parallelism achieved in the parallel loop and can be used interchangeably.

Note that *CXpa* does not report any of the above metrics, instead, it reports a *CPU time/ wall clock time ratio*, $(\Sigma_p\, c_{i,p})/W_i$, for loop *i*, where $c_{i,p}$ is the computation time that processor *p* spent in the loop. This particular performance metric views both computation and com-

munication time as useful work only the additional I/O and OS time included in the wall clock time is viewed as overhead. This metric (CPU time/wall clock time ratio) often yields similar value for the degree of parallelism of a parallel loop, but significant differences can emerge when a parallel loop spends a considerable amount of time in OS events or synchronization operations.

We notice that the load imbalance for CRASH-SP generally increases as the number of processors increases, which is typical for irregular applications.

2.3.2.2 Limitations of Profile-Driven Analysis

Profiling tools are generally useful in permitting the user to probe various machine-application interactions, as we have demonstrated in Section 2.3.2.1. However, the user usually has to reason carefully about what the performance metrics actually represent in order to gain realistic insights about the application performance. In addition to counting events, more sophisticated analysis should be integrated into profiling tools in order to permit their effective use by inexperienced programmers.

To save hardware cost, some profilers may use the same set of counters for profiling various types of events. For example, in a particular run, the HP PA7100/7200 on-processor counters can be configured to profile one of the four types of memory access events described in Table 2-5, thus one would need four profile runs in order to acquire the counts of all those four types of events. This results in an inconvenient and less consistent performance assessment; profiles obtained from different runs may not be consistent, especially for applications with dynamic behavior.

Profilers do not record performance metrics of different iterations separately; they simply accumulate the time in each region over all iterations and report the sum. As a result, dynamic behavior from iteration to iteration cannot be ascertained simply by using a profiler. However, if a profiler allows the profiled program to turn profiling on and off during runtime, a user with a great deal of patience could profile part of the execution in each run and capture some of the dynamic behavior.

Even with hardware support, profiling can be very intrusive to application performance, particularly when the application is profiled in detail. The accuracy of profiling can also be

influenced by other programs in a multitasking environment. Unfortunately, since current profile-driven analysis tools are far from foolproof, one should be cautious in using profiles until they are well-validated.

2.3.3 Trace-driven Analysis

Trace-driven analysis allows detailed assessment of the application's runtime behavior by performing post-run analysis of the event traces recorded during a run of the application. A trace contains a sequence of events, such as instructions, memory references, communications, and synchronizations, which may immediately interest the user, or help to define a more focused trace-driven simulation to derive a more detailed performance assessment.

Trace-driven analysis has conventionally been used by computer architects for studying processor performance in earlier design stages (e.g. [85][86][87][88]) and characterizing existing machines (e.g. [38][89]). In such usages, traces are often collected, sampled, or synthesized from benchmark programs. The storage and time required for collecting and analyzing traces are usually costly. For accurately assessing the application performance, trace collection needs to be less intrusive. Furthermore the functionality and complexity of the analysis need to be considered when making trade-offs in the design of trace-driven analysis tool. For example, a precise cycle-by-cycle simulation that models detailed machine-application interactions may be too complex to be applied to the entire application. Simplified machine or application models are often used to reduce the complexity and hence time requirements of trace-driven analyses, and yet provide useful results.

2.3.3.1 Assessing the Processor Performance

An instruction trace driven processor simulation that models major operations of the processor core, i.e. the datapath, yields a first-order approximation for assessing the processor performance on this application (e.g. [87][88]), as illustrated in Figure 2-6(a), which is useful for analyzing the instruction schedule, the instruction-level parallelism, or the efficiency of branch prediction. Traditionally, other factors, such as cache and memory effects on performance, are investigated separately to reduce the complexity of the processor simulation. Behavioral cache simulators, such as *Dinero* [90], examine the memory reference trace and

report the number of references and cache misses in the application, as illustrated in Figure 2-6(b).

Various schemes, such as *out-of-order execution*, *trace caches* (e.g. [91]), *prefetching* (e.g. [65]), and *multithreading* (e.g. [92]), have been employed in recent high-performance processor designs to reduce the performance impact of cache and memory. Simulating the processor and/or cache individually may not produce satisfactory results for studying the performance of these designs. A timing simulation that models the datapath, the cache, and the memory (e.g. [93][94]), as illustrated in Figure 2-6(c), can provide better accuracy in one unified simulation, yet the complexity and cost of such a simulation is higher than simulating the processor and the cache individually.

In our Parallel Performance Project, we organize individual simulation modules, *K-LCache* (see Section 2.4.4) or *CIAT/CDAT* [38], *mlcache* [93], and *SimpleScalar* [94], in a hierarchical fashion to reduce the complexity of the simulator, as illustrated in Figure 2-6(d):

1. First, the memory reference trace is examined by a shared memory simulator (*K-LCache* or *CIAT/CDAT*) to expose and flag the shared-memory communications in the trace;

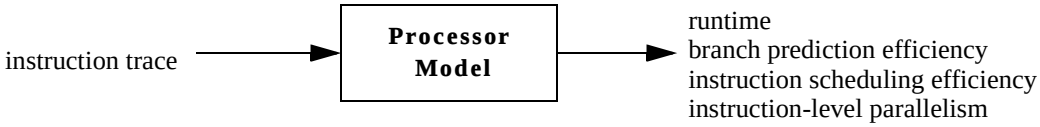
2. Then, the memory reference trace, with exposed communications flagged, is sent to a cache simulator (*mlcache*) to expose and flag the cache misses in the trace;

3. Finally, the memory reference trace, with the exposed communications and cache misses flagged, is used together with the instruction trace to perform a timing simulation of the processor-memory system using *SimpleScalar*.

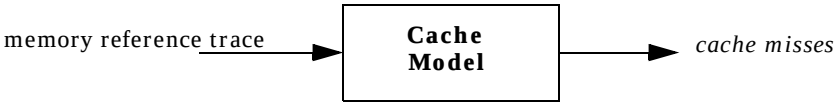
This approach simplifies the implementation in each of these modules, and also allows us to use a variety of existing cache and memory simulators.

2.3.3.2 Assessing Shared-Memory Communication Performance

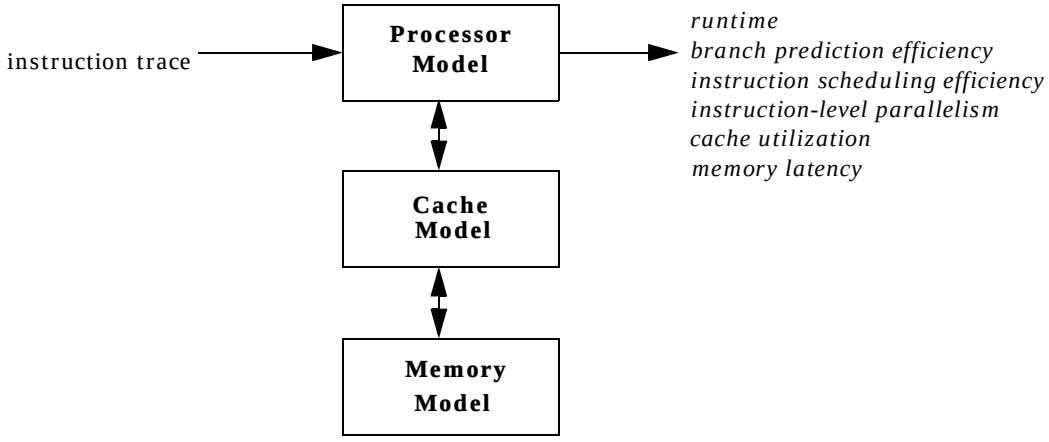
For a shared-memory application, we first focus on exposing its communication pattern, which is the key to detecting and solving performance problems in communications. Simulating the shared-memory with the memory reference trace is sufficient to expose this communication pattern. Tools of this kind and performance assessment examples can be found in [11][38][44]. This topic, as well as our shared-memory simulator, K-LCache, are further dis-



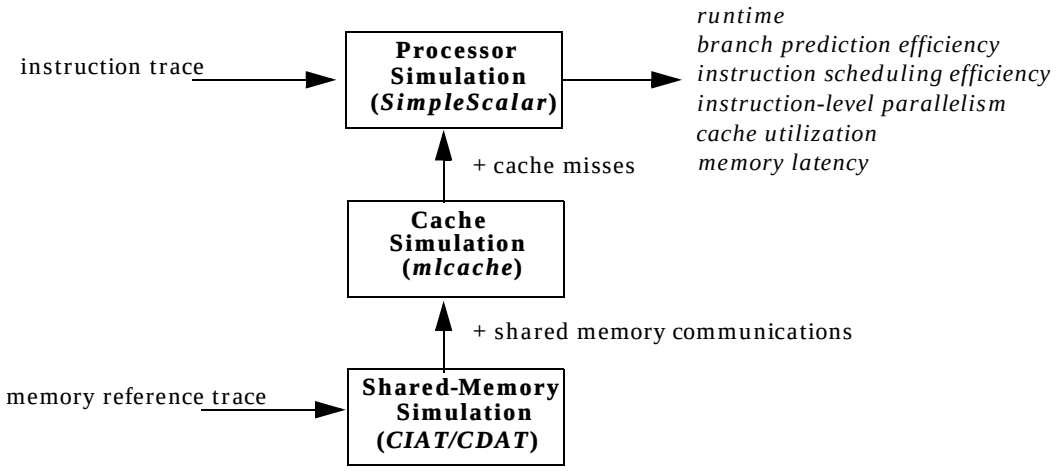
(a) First order approximation of the processor performance



(b) Simulating the cache



(c) Simulating the processor, the cache, and the memory system simultaneously



(d) Simulating the processor, the cache, and the memory system in a hierarchical fashion

Figure 2-6: Examples of Trace-Driven Simulation Schemes.

cussed in Section 2.4. K-LCache provides innovative techniques for categorizing shared-memory cache misses, and speeds up its simulation by using parallel execution.

After the communication pattern is exposed, we can further analyze the timing of the communication events by carrying out a processor-cache simulation, as mentioned in Section 2.3.3.1.

2.3.3.3 Assessing Message-Passing Performance

The performance of communication and synchronization in a message-passing application is explored by examining messages. As opposed to shared-memory applications, messages can be more conveniently exposed by instrumenting the message-passing library itself, or calls to the message-passing library. The source, destination, length, type, as well as time-stamps are commonly attached to each exposed message event that is recorded in the message trace. While time-stamps are prohibitively costly for instruction or memory reference traces, the cost of time stamping is small for message traces since the time between messages usually consists of a great many (normally more than thousands) instructions.

PICL/ParaGraph [76][77], *AIMS* [78], and *CXtrace* [79] are representative tools that employ this approach. In these tools, the runtime is partitioned into the *computation* (*busy*) time between messages, *message* time (time for *transferring/receiving* messages), and *idle* time (time spent in message routines *waiting* to communicate/synchronize). The runtime and the network traffic are analyzed to characterize the load imbalance (degree of parallelism) and the communication performance of the application.

2.3.4 Other Approaches

An approach similar to trace-driven analysis, *execution-driven simulation,* such as Proteus [95], Tango/Tango-lite [96][97], or Sim-OS [98], extracts the behavior of an application by running its executable code on a *simulating* machine with instrumentations that *emulate* the machine-application interactions between the application and a *target* machine. This approach allows one to analyze the application performance with an existing or hypothetical machine model. It also saves the space and time required for storing and retrieving traces. However, the implementation of execution-driven simulation is more complicated and error-

prone, thus, execution-driven simulation has not been a popular tool for application development.

Mixing profiling and tracing, hybrid approaches (such as IBM's *Performance Visualizer (PV)* [36]) monitor and trace selected machine-application events (e.g. messages) and performance metrics (e.g. number of cache misses). With the hardware performance monitors integrated into the RS/6000 POWER2 processor, PV can visualize selected runtime events and various performance metrics for the user in real-time during an application run. These visualized events and metrics can also be recorded in a trace and played back for post-execution analyses.

2.4 Efficient Trace-Driven Techniques for Assessing the Communication Performance of Shared-Memory Applications

In this section, we present techniques for accelerating the analysis of communication patterns. The tools that we developed for the KSR may have become obsolete, but most of the techniques used in these tools are applicable to newer machines. We summarize the KSR1/2 *Cache-Only Memory Architecture* (COMA) in Section 2.4.1. The communication patterns in this COMA machine are more subtle and less predictable than for most *Uniform Memory Access* (UMA) or the *Non-Uniform Memory Access* (NUMA) machines that prevail today.

In Section 2.4.2, we categorize and identify communications in a distributed cache system. In Section 2.4.3 and Section 2.4.4, we present the techniques and tools that employ trace collection and trace-driven simulation to expose the implicit communications that occurs in a parallel execution. Analysis of the communication patterns in the codes are important for reducing the communication overhead. Section 2.4.5 shows a case study how the tools can be used for analyzing the communication patterns.

2.4.1 Overview of the KSR1/2 Cache-Only Memory Architecture

The Kendall Square Research KSR1 was the first commercially available shared-memory parallel system to employ a COMA design. The memory system of the KSR1/2[1], also known as

1. The KSR1 and its successor KSR2 shared the same memory system design.

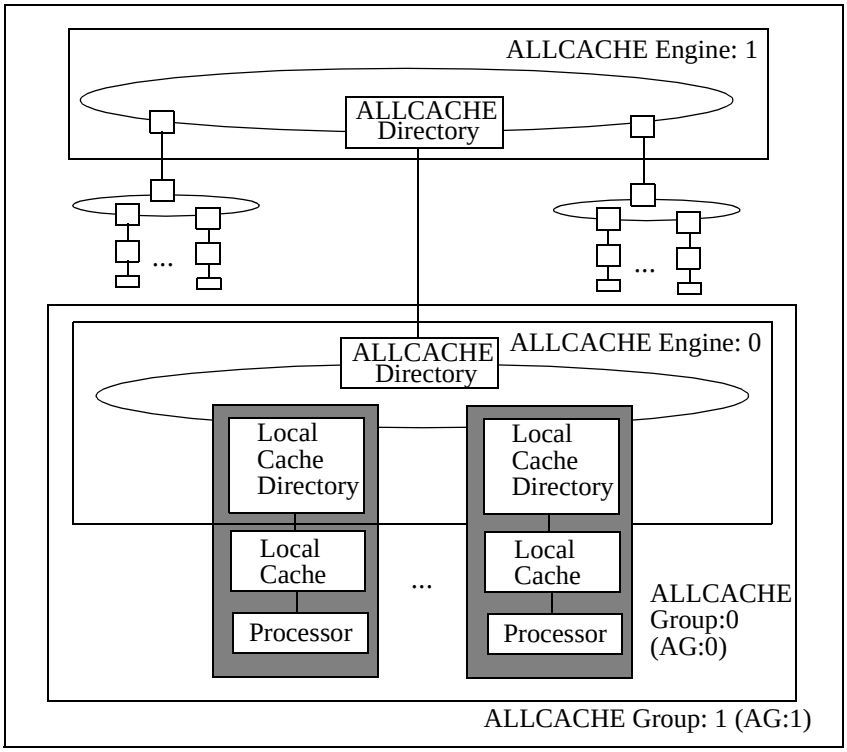


Figure 2-7: KSR1/2 ALLCache.

ALLCACHE, was conceived to be a group of ALLCACHE engines, connected in a fat tree hierarchy of rings, as shown in Figure 2-7. In practice, up to 34 rings could be connected by a single second-level ring for a maximum configuration of 1088 processors. Each single ring consists of up to 32 processor cells and up to two ALLCACHE directories. The ALLCACHE memory system provides programmers with a uniform 64-bit address space for instructions and data, which is called the *System Virtual Address space (SVA).*

The contents of SVA locations are physically distributed among the caches situated in the single-processor cells. An ALLCACHE Engine 0 is associated with each lower level ring and is physically comprised of a set of *local caches*, each capable of storing 32 MB. There is one local cache for each processor in the system. Hardware cache management mechanisms cause the page containing a referenced SVA address to be allocated, and to contain a valid copy of the referenced *subpage*, in the local cache of the referencing processor. That subpage remains in that local cache until the page is replaced or the subpage is marked invalid by some other processor.

Data items are dynamically moved and replicated, updated and invalidated, among the local caches based on the reference patterns in the parallel program. The programmer does not have to explicitly control the placement of data in the system. Besides specifying parallelism, the programmer's only concern is the work distribution and scheduling which implicitly controls the location of data and hence the traffic on the network. However, COMA systems require a run time mechanism to search for copies of valid cache lines in the system and manage them properly with a complex cache coherence protocol, which adds runtime overhead. Nevertheless, the KSR1, as opposed to traditional supercomputers, offered scalable performance, a competitive price, a familiar UNIX (OSF/1) operating system, an easy-to-use shared-memory programming model, and a parallelizing Fortran compiler. Upon its introduction in 1991, the KSR1 became popular as a low-cost supercomputer.

Each KSR1 node contains a 64-bit custom VLIW RISC processor with a 20 MHz clock. The processor allows a two-instruction issue per clock cycle: one address calculation, branch, or memory instruction and one integer or floating-point calculation instruction. The instruction scheduling on the KSR1 is done purely by the compiler, including *nop* (no operation) instructions placed for hazard-protection since there is no hardware interlocking mechanism on the pipelines. Cache misses generate exceptions that invoke system routines (firmware) to perform the cache coherence protocol and memory accesses. Floating-point multiply-add triad instructions allow a peak performance rating of 40 MFLOPS. However, as microprocessor performance continued to evolve rapidly in the early 90's, the custom-designed processors of the KSR1 quickly fell far behind high performance commodity microprocessors, such as the IBM RS6000, HP PA-RISC and DEC Alpha. Due to the inability to provide new generation custom processors in time, the KSR2, which incorporated processors twice fast (40 MHz) as the KSR1, was not released until 1994, and its single processor performance was quite far behind many of its competitors at that time. Mediocre single processor performance, combined with financial problems caused KSR to go bankrupt in 1995.

A 32-processor KSR1 system was the first parallel machine installed at the CPC of the University of Michigan in 1991. A 64-processor KSR2 system was later installed at the CPC in 1993. References [84][99] provide detailed descriptions of the KSR1. Performance evaluations of the KSR1 can be found in [44][84][41][52].

2.4.2 Categorizing Cache Misses in Distributed Shared-Memory Systems

In *UMA* machines, there is no need to distinguish communications from memory accesses since they have the same latency. In *NUMA* machines without caches, communications are said to occur when processors access remote memory locations. The remote memory accesses can usually be identified by their addresses. However, in a *distributed cache shared-memory* or *cache-only* machine, such as the KSR1/2, the data movement in the system is hidden from the user because the cache system moves data among the caches automatically.

In the KSR1/2, each subpage that is cached in a local cache has an associated state. As viewed by the requesting processor, a memory access is a *hit* if the subpage that holds the requested data is cached in the local cache of the requesting processor *and* the state of the subpage satisfies the type of the memory access; otherwise, it results in a *local cache miss* that requires *copy*, *invalidation*, or *update* of the requested subpage from/to remote caches [84].

There are several types of local cache misses. Besides *compulsory*, *capacity*, and *conflict* misses that occur in single caches, *coherence misses* are references to subpages that are not in whose state in the local cache is not appropriate for that reference, according to the cache coherence protocol. In a parallel execution, these misses are intermixed and difficult to identify during runtime. Thus, we developed a *post-execution* methodology to categorize cache misses in a distributed cache system by extending Sugumar's *OPT* methodology [100]. This new cache miss categorization system, called *D-OPT* model, includes a hierarchy of compulsory, coherence, capacity, mapping, and replacement misses, as defined below: (Note that a cache *line* in this definition is equivalent to a subpage in the KSR1/2).

Definition 1. Categories of Misses in Distributed-Cache Systems (*D-OPT* Model): Given a target distributed-cache system, where each individual cache has size C, associativity A, line size L, and replacement policy R, the cache misses generated in an application are categorized as:

- *Compulsory miss*: the misses that would still occur for a *unified cache* of infinite size. A compulsory miss occurs when the program accesses a cache line for the first time.

Cache Miss Category	Cache Misses in the Category
Compulsory	*Misses(L, ∞, ∞, - , unified)*
Coherence	*Misses(L, ∞, ∞, - , distributed) - Misses(L, ∞, ∞, - , unified)*
Capacity	*Misses(L, C, C/L, opt, distributed) - Misses(L, ∞, ∞, - , distributed)*
Mapping	*Misses(L, C, A, opt, distributed) - Misses(L, C, C/L, opt, distributed)*
Replacement	*Misses(L, C, A, R, distributed) - Misses(L, C, A, opt, distributed)*

Table 2-6: D-OPT Model for characterizing a Distributed Cache System.

- *Coherence miss*: the additional misses that would occur for a *distributed* cache system where each individual cache is of infinite size. The number of coherence misses is the total number of misses in this system minus the compulsory misses above.

- *Capacity miss*: the additional misses that would occur for a distributed cache system where each individual cache has *size C*, full associativity, and an optimal replacement policy, i.e. all misses in this system minus the compulsory and coherence misses above.

- *Mapping miss*: the additional misses that would occur for a distributed cache system where each individual cache has size *C*, *associativity A*, and optimal replacement policy, i.e. all misses in this system minus the compulsory, coherence, and capacity misses above.

- *Replacement miss*: the additional misses that occur for the actual distributed cache system, where each individual cache has size *C*, associativity *A*, and *replacement policy R*, i.e. all misses minus all the above types of misses.

Given a memory access pattern, let *Misses(l, c, a, r, d)* represent the cache miss pattern in a cache where *l*, *c*, *a*, *r*, and *d* are the line size, cache size, associativity, replacement policy, and distribution (unified or distributed) of that cache. Table 2-6 shows the characterization of the cache misses in the target distributed cache system. In this table, note that full-associativity is represented as *C/L*, *opt* replacement indicates an *optimal replacement policy with bypass* [100], and a replacement policy is not necessary (indicated as -) for a cache of infinite size.

The *D-OPT* model represents a hierarchical view of the target cache system. From compulsory misses to replacement misses, each cache miss category characterizes a level of hierarchy with respect to idealization of the cache system from a perfect, but not preloaded,

system (most idealized view), then successively adding individual caches, finite capacity, associativity, and the actual replacement policy (most detailed view).

The *D-OPT* model is not suitable for categorizing cache misses in real time, because it requires evaluations of multiple cache configurations and an optimal replacement policy, but it can be carried out by using trace-driven simulation. We have developed tools to extract the coherence misses in an application by simulating an infinite-size distributed-cache system and an infinite-size unified cache system given an the application's memory reference trace. Such a simulation is much faster than conventional cache simulation schemes, yet it is sufficient to provide the *coherence misses* that are *inherent* to a given parallel application, as categorized in our *D-OPT* model. Once compulsory and coherence misses are identified and marked in the trace, the capacity, mapping, and replacement misses of each individual cache can be extracted by simulating each cache individually.

2.4.3 Trace Generation: K-Trace

K-Trace is a tracing tool for the KSR1/2 parallel computer; it modifies an assembly program by inserting tracing instructions so that the program can automatically generate a memory reference trace during the run-time. K-Trace has successfully generated traces for our performance studies on the KSR1 at the University of Michigan.

Simulation is an important technique in hardware and software studies, providing measurements to evaluate the performance of computer systems [85][86][89]. In studies of the KSR1, the traces generated by K-Trace can be used by trace-driven simulators to simulate processor caches, local caches, and communication traffic. Thus, K-Trace enables detailed studies of program behavior, compiler optimizations, and the architecture of the KSR1, in a way that is helpful for tuning the performance of practical applications.

K-Trace is based on *in-line tracing*, a method that has been broadly used in other tracing and performance tools [101][83]. However, it is not straightforward to port tracing tools from other platforms to the KSR1 because of some unique hardware and software features of the KSR1. Some difficulties we encountered in developing K-Trace are machine-specific problems which require detailed or undocumented information about the KSR1. For example, assembly language programming was essential to developing K-Trace, but was not supported by KSR.

2.4.3.1 The Design of K-Trace

In-Line Tracing

K-Trace employs in-line tracing techniques which consist of the following two phases:

- *Instrumentation*: K-Trace reads the assembly version of a source program and produces a modified source to be executed. The instrumentation includes the insertion of tracing code and transformations of conditional branch instructions. Figure 2-8 shows an example inline code. The instrumentation of a file is relatively fast, taking less time than compiling the source program. The K-Trace preprocessor is written in a mixture of C and *lex*, a compiler generating tool under Unix

- *Trace Generation*: The modified program is then linked with K-Trace run-time routines and executed. The execution produces a trace that includes the addresses, instruction ID's and processor ID's that identify memory references. The *synchronization points* in the program are also marked in the trace. Figure 2-9 shows an example trace.

Since K-Trace directly modifies assembly programs without any information from compilers, it is independent of compilers and can be used to trace C, Fortran and assembly programs on the KSR1/2. The instructions inserted during instrumentation do not affect the state of the processors in the trace generation phase. The state of a processor is composed of registers and condition codes. The results of an instrumented program should remain the same as the original program, although its timing behavior may be different from the original programs because of the run-time dilation introduced by instrumentation.

Basically, K-Trace scans the original assembly code, looks for memory instructions, and inserts a trap code before each memory instruction. The trap codes that are inserted by K-Trace will be executed to record the memory references in the trace. Each trap code starts with **`#Trap`**, and ends with **`#EndTrap`**. Each trap is assigned a number, location id, which can be used to identify the location of the trapped instruction. A trap code calls the *add_trace* subroutine in the *K-Trace runtime library* to record the location id, address, and type of a memory reference.

```
# Trap 2:
   finop                                ;  cxnop
   finop                                ;  cxnop
   finop                                ;  st8          %sp, -64(%sp)
   finop                                ;  st8          %i6, -32(%sp)
   finop                                ;  st8          %c6, -24(%sp)
   finop                                ;  st8          %c10, -8(%sp)
   movi8        104, %i6                ;  ld8          MAIN_$TRAP-MAIN_+8(%cp), %c6
   finop                                ;  st8          %i6, -72(%sp)
   finop                                ;  st8          %c14, -16(%sp)
   finop                                ;  jsr          %c14, 16(%c6)
   movi8        20002, %i6              ;  ld8          MAIN_$TRAP-MAIN_(%cp), %c10
   finop                                ;  st8          %i6, -80(%sp)
   finop                                ;  cxnop
   finop                                ;  cxnop
# EndTrap 2:
   finop                                ;  st8          %c14, 104(%sp)
```

Figure 2-8: An Example Inline Tracing Code.

The inserted trap codes add to the length of the instrumented code. Because conditional branches have limited branch offsets, K-Trace replaces conditional branch instructions with jump instructions. The constant tables of instrumented assembly subroutines also have to be modified so that the **`add_trace`** subroutine can be called within those subroutines.

In a uniprocessor system, the *run-time dilation* does not affect the accuracy of the collected traces because the memory references are recorded in the same order as in the original program. In a shared-memory multiprocessor system, the timing of memory references in a set of processors is very important because the order of all references among all processors to a particular global address can easily affect the amount of interprocessor communications.

Parallel Traces and Synchronization Points

On an MIMD machine like the KSR1, the overall instruction sequence of a parallel program is not deterministic. In an MIMD machine, each processor executes its own instruction stream at its own rate until a synchronization point is reached, therefore the order of the instructions, and more particularly the memory references, between synchronization points is not deterministic over the whole system. A trace generated by K-Trace may be just one case from numerous possible traces of the same program. One may argue that using only one trace is not enough to characterize the behavior of a program; however, it is almost impossible to explore all the possibilities.

A trace format has been developed to solve this problem. By modifying the KSR1 *Presto* thread-control library, synchronization points are automatically inserted into a trace by the

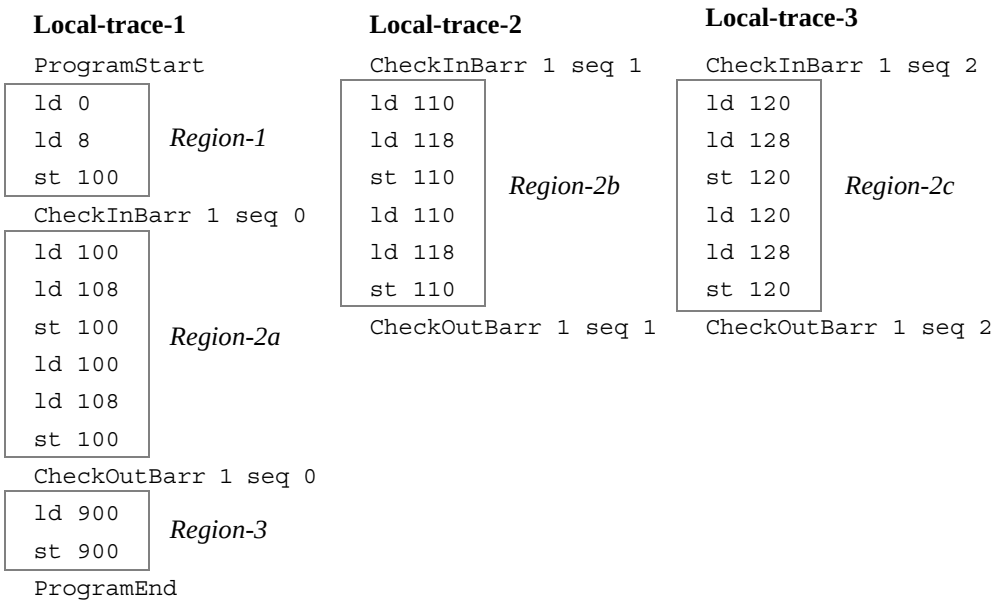


Figure 2-9: A Parallel Trace Consisting of Three Local Traces.

K-Trace runtime library. Each processor sequentially outputs its memory references and synchronization points to a separate file, which is called a *local trace*. This format improves the performance of trace generation with parallel trace recording and provides more complete information as to the order of memory referencing for trace-driven simulation tools, such as K-LCache.

Local traces, rather than a single unified trace, are used because sequential file access to a unified trace would be a bottleneck in trace generation, since only one processor at a time can access the next element of a unified trace file. Figure 2-9 shows an example with three local traces. *CheckInBarr* and *CheckOutBarr* are synchronization points, which separate the traces into *regions*. The execution order of the regions is fixed by the synchronization operations. For example, Region-1 must complete before the other regions begin. The Region-3 can begin only after Region-2a, Region-2b, and Region-2c all complete. However, the actual memory reference sequence among the three processors within Region-2 is uncertain.

2.4.3.2 Generating Traces with K-Trace

There are five steps for generating a trace for a program with K-Trace. As shown in Figure 2-10, these steps are:

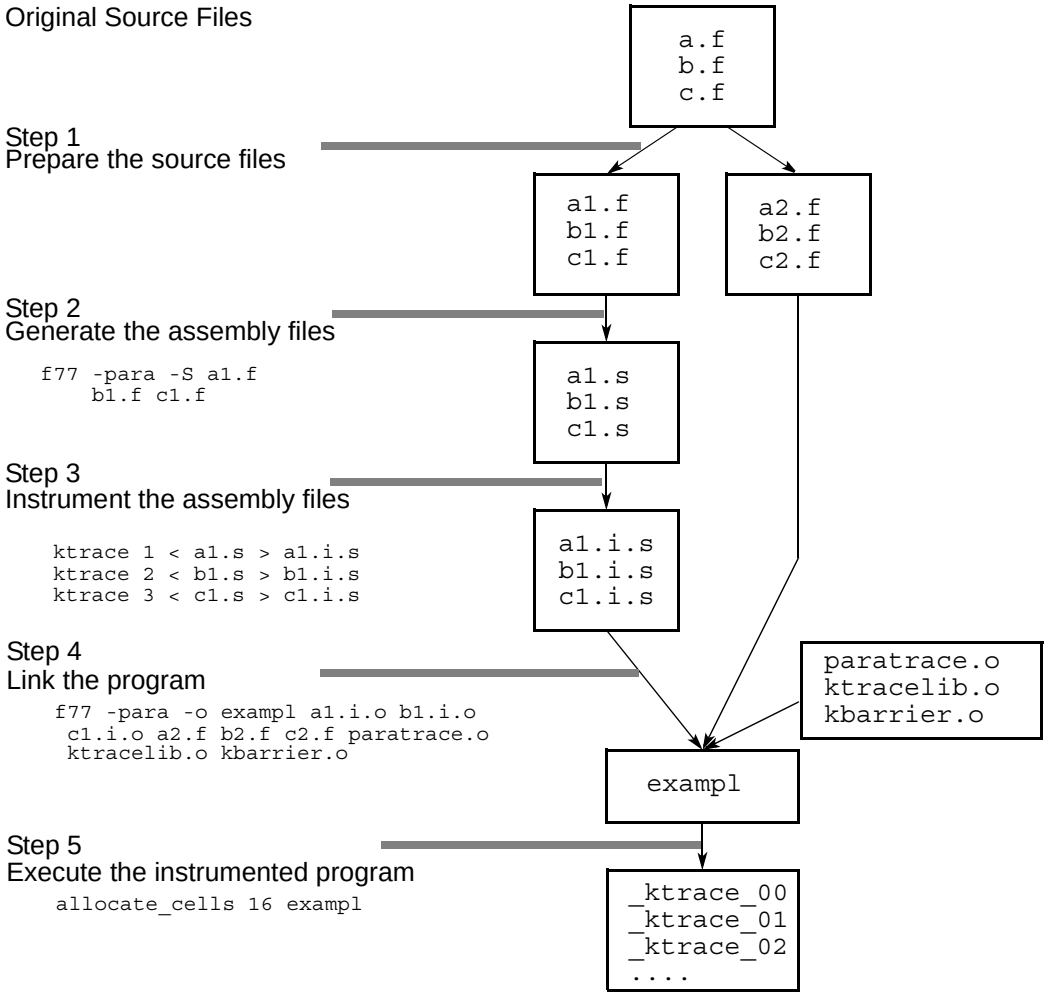


Figure 2-10: Trace Generation with K-Trace.

1. *Prepare the source files*: Users of K-Trace can decide which portions of the source code to trace and instrument those portions only. For example, in Figure 2-10, *a1.f, b1.f, c1.f* are the portions to be instrumented. K-Trace provides a set of *tracing directives,* listed in Table 2-7, which control the trace generation during run time.

2. G*enerate the assembly files*: The source code portions that need to be instrumented are compiled into assembly language.

3. *Instrument the assembly files*: Each assembly file generated in step 2 is assigned a different file identification number and instrumented independently. The file identification number will be recorded with the memory references in the corresponding file.

Directive	Comments
ktrace_on	Turn on tracing.
ktrace_off	Turn off tracing.
ktrace_end	Dump the references in buffer and stop the program execution.
ktrace_mark_location	Output a location identification number to the trace file.
ktrace_mark_variable	Output the identification number and the address of a variable to the trace file.

Table 2-7: Tracing Directives.

4. *Link the program*: Link the instrumented assembly files with the un-instrumented files and the K-Trace runtime library files.

5. *Execute the instrumented program*: The memory references in the instrumented portions will be recorded in a trace when the instrumented program is executed. Multiple trace files will be generated to store the memory references on different processors.

2.4.3.3 Limitations of K-Trace

A common disadvantage for inline tracing is that the *library* routines cannot be traced without access to their source codes. Also, the mapping between data structures and addresses is not available in the trace file because the KSR Fortran and C Compilers do not generate a memory allocation map file when optimization is turned on, so the users need to manually insert function calls (mark directives) to record key addresses of interesting data structures in the trace file. These addresses can then be used to construct the map.

2.4.4 Local Cache Simulation: K-LCache

K-LCache takes a parallel trace generated by K-Trace, simulates the local cache coherence protocol of the KSR1/2, and reports *compulsory and coherence misses* in the parallel execution. In general, simulation of a distributed-cache system is very time-consuming and memory-demanding. As mentioned in Section 2.4.2, we focus on the coherence misses, which represent true interprocessor communication. By simulating infinite distributed caches, K-LCache extracts the compulsory and coherence misses much more efficiently than conventional cache simulators, and the simulation can be performed on portions of the trace address

space to reduce the memory requirements of K-LCache. We also introduce two types of communications, *required* and *optional*, for characterizing the effect of cache line sharing. We shall show how further speedups of the simulation can be achieved by decomposing the traces and simulating disjoint portions of address space in parallel.

Categorizing Communications

In an MIMD shared-memory computer, the order of memory references in a parallel execution may be nondeterministic. Nondeterministic communication patterns can occur whenever two processors access the same cache line in their execution of a particular parallel region. We identify two types of communications in a parallel region:

- *Required (minimum, inherent) communication*, which occurs regardless of the actual memory reference order that occurs in a particular parallel execution run.
- *Optional communication*, which occurs only when one of a particular proper subset of the memory reference orders occurs among accesses from multiple processors to the same memory block.

Consider the example trace shown in Figure 2-9. A communication pattern obtained from simulating that trace is shown in Figure 2-11. The labels R.C. indicate the required communications, and O.C. indicate optional communications. O.C. are due to subpage sharing, where one processor invalidates the useful data in other local caches by writing to the same subpage. Recall that a subpage is 128 bytes, so addresses **`100-17F`** are in the same subpage. For example, in local-trace 1, *Region-1* ends with a store to address **`100`** and *Region-2a* begins with a load from **`100`**. However, either of other two processors may write the subpage where 100 resides by executing the "**`st 110`**" instruction in *Region-2b* or the "**`st 120`**" instruction in *Region-2c* before the "**`ld 100`**" instruction in *Region-2a* is executed. The load could generate communication if the subpage **`100-17F`** is written by another processor, so this communication is optional. On the other hand, the required communications are independent of the order of execution in the parallel region.

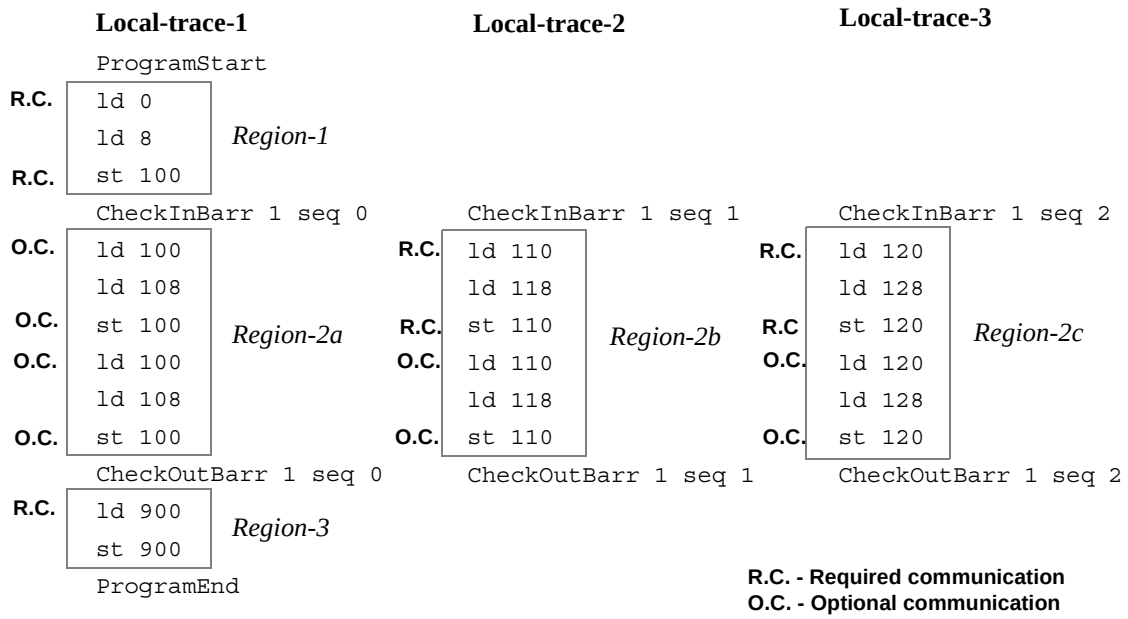


Figure 2-11: Communications in a Sample Trace.

Finding Required Communications

For finding required communications, K-LCache simulates each local cache independently of the others within each parallel region. The status of a set of local caches only needs to be reconciled when a synchronization among them is found in their associated traces. As parallel regions can be nested in a program, K-LCache maintains cache state accordingly. Whenever processors check in at a region, K-LCache creates a next-level *subpage state table* to record the local cache state transitions within this region for each processor. The local cache state transitions for a processor are driven by the corresponding local trace and recorded in the corresponding subpage state table. By searching the corresponding subpage state table and the previous-level subpage state table (which holds the cache state before entering the region), K-LCache checks if a memory reference can be satisfied by the local cache. When those processors check out of that region, the previous-level subpage state tables are updated coherently by reconciling the information in all the subpage state tables in that region. K-LCache uses a recursive algorithm to deal with nested parallel regions. Dynamically allocated subpage tables are used for storing local cache states at each nesting level. This simulation reports the required communications for each parallel region.

Finding Optional Communications

The optional communications are detected by comparing the subpage state tables when the processors check out of the region. The simulator reports the addresses, the processors, the number of reads and the number of writes associated with the optional communications. The user can locate shared objects in the source code by using the addresses and the processor identifiers. The amount of optional communication can be estimated and bounded by the number of reads and writes. Possible patterns for the optional communications can be extracted by performing simulation on the shared subpages.

A very loose upper bound of the optional communications generated by a subpage accessed in a parallel region is 0 if $P_{RW} = 1$, or $Min\{W+R, W*(P_R+1)\}$ if $P_{RW} > 1$, where W is the number of writes to that subpage in that parallel region, R is the number of reads to that subpage in that region, P_R is the number of processors that have read that subpage in that region, and P_{RW} is the number of processors that have read and/or written that subpage in that region. When $P_{RW} = 1$, there is no optional communication since the access pattern of the subpage is fixed. When $P_{RW} > 1$, the subpage is shared by multiple processors. It is clear that the number of misses is not larger than the number of memory references, i.e. $R+W$. When R is larger than $W*P_R$, each write to the subpage in the parallel region can generate at most one write miss for the writing processor and P_R read misses for the reads following this write, and hence the number of misses should not exceed $W*(P_R+1)$. Note that when W=0, no optional communications can occur in the parallel region, since the reads to the subpage in the region can only result in required communications.

A large upper bound on optional communications can be the result of a large W and/or a large P_R for shared ($P_{RW} > 1$) subpages, which indicates a potential performance hazard or even (in the presence of insufficient synchronization in the code) an incorrect execution that could occur when a particular memory reference order is executed in the parallel region. There are two kinds of sharing that have been generally used to refer to such a problem.

- *True-sharing*: The sharing of a memory location is referred to as *true-sharing* if the location could potentially be accessed by multiple processors. When true-sharing occurs within one parallel region, the shared memory location could be accessed in a nondeterministic order and could cause *incorrect execution* if at least one of these accesses is a *write*. For

example, if two processors both execute *x=x+1* in the same parallel region, since (1) the order of true-sharing accesses is uncertain, and (2) the operation is not atomic, the execution may not produce consistent results and thus should not be allowed in a well-orchestrated code, unless the sharing accesses are protected with locks or orchestrated within a reduction operation. K-LCache issues a warning message that flags the sharing accesses when it detects true-sharing (with at least one write) within one parallel region. Given a warning message, the programmer can then verify whether these sharing accesses comply with the intended program semantics or are due to programming errors.

- *False-sharing*: The sharing of a subpage is referred to as *false-sharing* if there is no true-sharing within that subpage, but that subpage is accessed by multiple processors. False-sharing within in one parallel region can cause a *performance hazard* if some of the accesses are *writes*. For example, the memory accesses labeled as optional communications in Figure 2-11 are false-sharing accesses. False-sharing accesses can impose unnecessary coherence misses on a parallel execution. The programmer may reduce false-sharing accesses by reducing the number of falsely shared subpages using data layout techniques, and/or by making local copies of falsely shared subpages (privatization), as further discussed in Section 3.3.

Parallel Execution of the Simulation with Multiple Processor Traces

Since K-LCache simulates each local cache independently of the others within each synchronization region, simulation for memory references in different local traces can be performed independently. Given the K-Trace output from an instrumented run of an application code on *P* processors, K-LCache can simulate the *P* local traces concurrently on *P* processors. The speedup of the parallel simulation is good when the parallel regions of the program are long.

Accelerating the Simulation with Address Filtering

For extracting compulsory and coherence misses, simulation of memory references to different subpages can be performed independently because references to different subpages do not interfere with one another in infinite caches. K-LCache is capable of simulating a filtered trace, which contains only the memory references in a specific address range. The user can focus on certain data structures by filtering out references to the others. We have developed a filter program that is capable of reporting the address histogram of a trace and partitioning

trace files into evenly-sized subtrace files that contain different ranges of addresses. With their smaller sizes and smaller address ranges, a separate simulation on each trace portion done in parallel with the other portions can be performed much faster and requires less storage than one simulation done on the original trace.

2.4.5 Analyzing Communication Performance with the Tools

For irregular applications, compile-time analysis of the communication overhead may not be possible due to indirect array references. In this section, we show how our tools can be used for analyzing the performance of an ocean simulation code, *OCEAN*, running on the KSR2. We analyze the communication patterns, estimate the execution time, and propose ways to reduce the communication overhead of the program, using the tools that we developed. We found that more than 90% of the communication is of *multicast*, *producer-consumer* type, where a data item is written by one processor, but read by multiple processors during the program. This type of communication is commonly found in many parallel programs, such as CRASH; the NAS Parallel Benchmarks [11] are also of this type.

OCEAN has indirectly indexed arrays, but the computation load is uniformly distributed over the problem domain. The problem domain is evenly partitioned for the parallel execution. The domain decomposition results in good load balance, yet the parallel speedups on the KSR2 are 3.685 for 16 processors, and 3.74 for 25 processors. Using the *KSR Performance MONitor* (PMON), we identify that the cause of this poor scalability is essentially due to the local cache miss latency.

We collected a parallel trace of the first 3 iterations using K-Trace. There are 7 synchronization regions in each iteration and 120M memory references in 25 sub-trace files. The actual working set size is about 80 MB. To reduce the memory requirement for carrying out a trace-driven simulation with K-LCache, the set of sub-traces is partitioned into 70 portions, with a 4MB address range for each portion. The simulation reports the number of compulsory misses and coherence misses for every address over the first 3 iterations.

Analyzing the Communication Patterns

First, we identify the data structures that cause coherence misses, as shown in Figure 2-12(a). The number of misses is high in the first iteration due to compulsory misses. The numbers in 2nd and 3rd iterations indicate the communication overhead. To analyze the communication pattern further, we categorize the coherence misses by the degree of sharing of the data item in the program. A data item can be (1) read and/or written only by one processor (*private*), (2) read by multiple processors (*read-only*), (3) written by one and read by multiple processors (*producer-consumer*), or (4) both read and written by multiple processors (e.g. *synchronization variables*, *data migration*, *reduction variables*, or *false sharing*). Only the last two types of data items cause coherence communication.

As Figure 2-12(b) shows, 97.6% of the data items are of producer-consumer type, and they are the major source of the communication overhead. The communication pattern for a producer-consumer type data item is visualized as shown in Figure 2-12(c). In this case, processors *P2* and *P3* are consumers that read the data item during *consuming* region. Processor *P1* updates the data item during *producing* region. Consuming regions and producing regions are separated by barriers to ensure correct access order. With the KSR2 write-invalidate protocol, the first write to a subpage by *P1* in each producing region invalidates the copies on *P2* and *P3*, and the first reads by *P2* and *P3* in consuming regions need to copy the updated subpage from *P1*. These invalidation and copy operations compose the communication pattern. This producer-consumer communication pattern is due to the data dependency between boundary elements of the domain decomposition. The boundary elements are updated by their owners in one parallel region, and some other processors request copies of the updated boundary elements in some later parallel region.

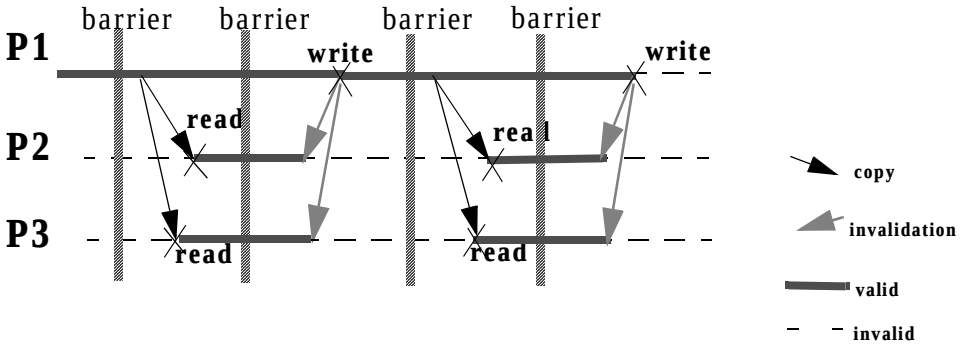
The above analysis and visualization can also be performed automatically by our tools, without knowledge of the original program. Actually, K-LCache and the analysis can also be integrated into K-Trace for runtime analysis, thereby saving trace storage. Compared to other profiling tools, our tools provide more complete and detailed information about the objects, their locations, and the patterns of communications.

Data Structure	Number of Cache Misses		
	Iteration 1	Iteration 2	Iteration 3
ETA	11584	1320	1320
VELD	30256	7630	7630
VELD2	30240	7626	7626
HOLD	15438	4206	4206
V	67231	25670	7674
U	98655	25621	7635
H	108509	66829	48885
Others	128309	1023	749

(a) Statistics of Cache Misses

Category	Number of Subpages	Percentage
Producer-Consumer	59882	97.6%
Multiple Read/Write	1486	2.4%

(b) Categorizing Coherence Communications



(c) Producer-Consumer Communication

Figure 2-12: Coherence Misses and Communication Patterns in an Ocean Simulation Code on the KSR2.

Knowing the sources of communications and the communication pattern, we can be more accurate in orchestrating the communications. For example, to hide the communication latency in OCEAN, a *prefetch* instruction can be issued by the consumer to request the data before it is needed; or a *post-store* instruction can be executed immediately after the producer writes the data to force the subpage that contains the data to be sent to the consumers, giving them an opportunity to achieve an early update (a more detailed discussion of latency hiding techniques on the KSR1/2 can be found in [52]). Since the invalidation traffic is an artifact of the cache coherence protocol, its impact might be reduced by using a different protocol, e.g. *write-update*, and/or using a *relaxed memory consistency model* that allows the invalidates to be carried out after the writes. Information about communication patterns is useful in selecting and applying certain performance-tuning techniques as well as in automatic program conversion from shared-memory to message-passing [102]. The above techniques, as well as some other techniques that concern communication performance, are discussed further in Section 3.3.

2.5 Summary

In Section 2.1, we have introduced several aspects of machine performance characterization and discussed its usefulness in assessing application performance. The machine model that results from machine characterization can be used to roughly estimate the application performance and expose problems in the performance.

In Section 2.2, we have discussed a range of machine-application interactions that are often be associated with major performance problems. In particular, in Section 2.2.4, we described the performance problems that we have speculated about with respect to running CRASH-SP on the HP/Convex SPP-1000/1600. With the intuition provided by this discussion, talented programmers may be able to envision specific machine-application interactions and identify some major problems in their applications; but for most programmers, better tools are needed to expose various run time events and assess application performance in a detailed, accurate, and timely fashion.

In Section 2.3, we surveyed a range of existing performance assessment tools. Source code analysis usually provides information that is most directly related to the source code, and is thus particularly useful for hand-tuning the source code. Unfortunately, source code analysis

is often limited by the ambiguity of data and control flow in the application that cannot be resolved before the actual execution. Profiling tools provide performance information by directly measuring specific machine-application interactions during runtime. Such information is useful for characterizing performance problems quantitatively, yet the exact cause of a problem and locations where it arises may not be accurately pinpointed. Trace-driven analysis can be used to study specific event patterns in the application. The trade-offs among functionality, cost, and accuracy of performance assessment are major concerns in designing and using trace-driven tools. Various trace-driven simulation schemes have been discussed.

In Section 2.4, we described K-Trace and K-LCache, which were, to our knowledge, the only trace-driven simulation tools available on the KSR1/2 for analyzing shared-memory communications. Based on our new D-OPT cache miss categorization system and communication categorization (required/optional communications), we have reduced the complexity of simulating the KSR1/2 distributed cache system, and furthermore, we have provided two mechanisms (multiple traces and multiple address ranges) to accelerate the simulation by executing K-LCache on parallel processors. Although these tools are obsolete now, most techniques mentioned in this section can be applied in the design of similar tools for recent shared-memory machines, such as the HP/Convex Exemplar and SGI Origin2000.

CHAPTER 3. A UNIFIED PERFORMANCE TUNING METHODOLOGY

In Chapter 2, we discussed some common performance problems in irregular applications. As these problems are often correlated, solving one problem may expose or aggravate in other problems that reduce the overall improvement, or actually make the performance worse. Unfortunately, most performance-tuning techniques previously developed tend to address performance problems in regular applications or focus on solving individual performance issues. For example, Tseng et. al. [103] have addressed these performance issues with unified compilation techniques, but their work focuses on compiler techniques for regular applications.

Currently, irregular applications are for the most part tuned manually by programmers, as illustrated in Figure 3-1. Our work aims at techniques for irregular applications that will

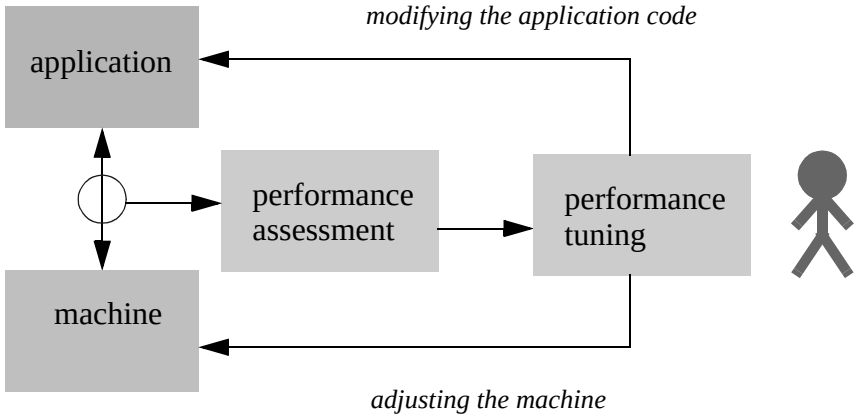


Figure 3-1: Performance Tuning.

initially be used to aid in the hand-tuning effort, and hopefully, after more experience is gained, will find their way into future compilers. In this chapter, a unified performance-tuning methodology is presented for solving these performance problems. We show how performance tuning can be systematically conducted by addressing individual performance issues in a logically-ordered sequence of steps.

3.1 Step-by-step Approach

Figure 3-2 summarizes the performance-tuning scheme that we use to improve the application performance. The numbers show the order of the steps, and the arrows show the dependence between the steps. When the program is modified in a certain step, the earlier steps found by following the backward arrows may need to be performed again as they may conflict

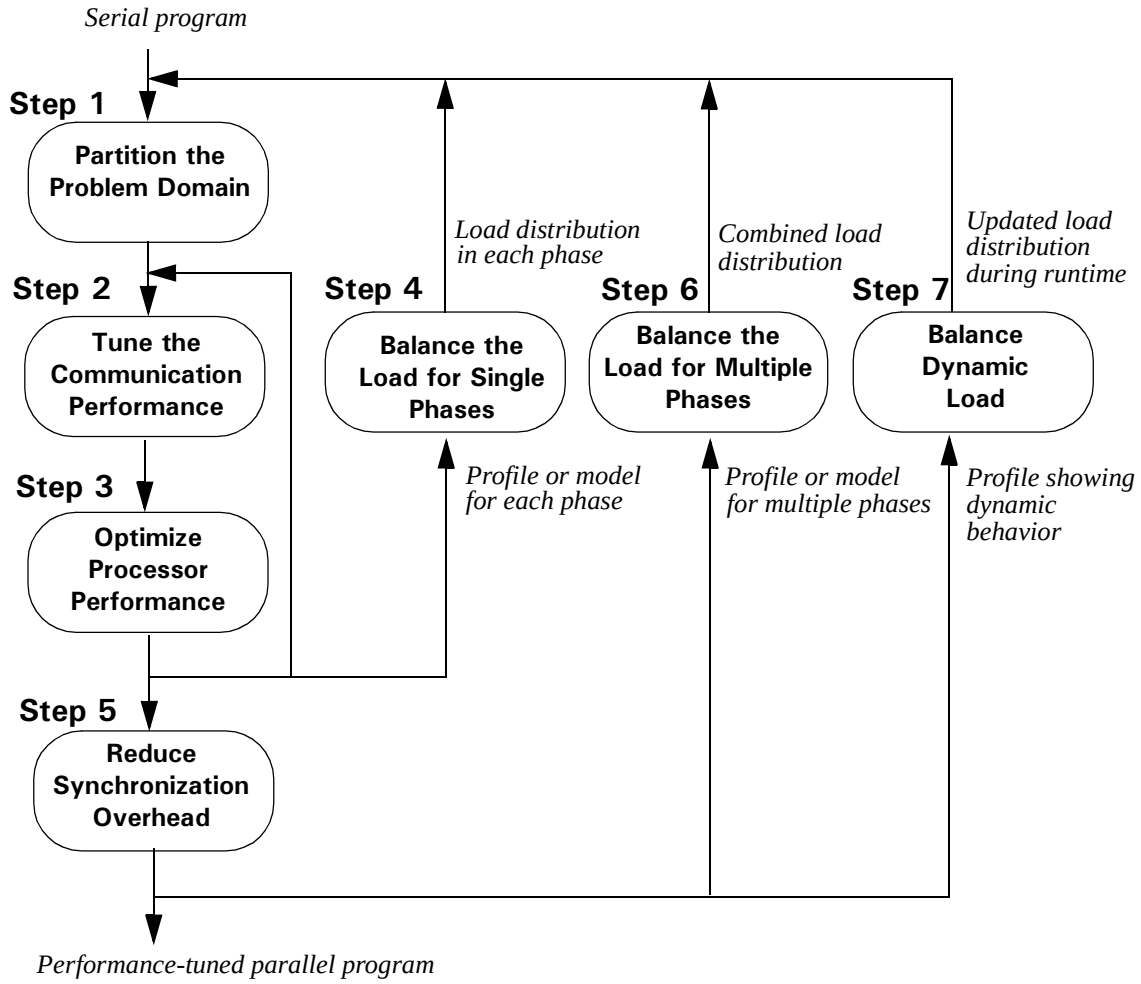


Figure 3-2: An Ordered Performance-tuning Methodology

with or be able to take advantage of the modification. For example, load balancing techniques in Steps 4, 6 and 7 may suggest different partitionings of the domain, which would cause different communication patterns that may need to be re-optimized in Step 2. Changing the memory layout of arrays to eliminate false sharing in Step 2 might conflict with certain data layout techniques that improve processor performance in Step 3. Changing communication and processor performance may affect the load distribution which then needs to be re-balanced. In general, this graph detects various types of performance problems in an ordered sequence, and a step needs to be repeated only if particular problems are detected and dealt with. Less aggressive optimization techniques that are more compatible with one another are better choices in the earlier phases of code development.

Through the following sections, we classify the performance issues and possible actions to address them in each step. The performance issues and actions are numbered sequentially as they arise in the discussion. Issue *x* and action *y* are referred as (IS *x*) and (AC *y*) respectively. At the end of this chapter, we summarize the correlations among these tuning steps, performance issues, and actions.

3.2 Partitioning the Problem (Step 1)

In Step 1, the problem domain is partitioned for parallel execution. We assume that parallelization of the application is achievable, and, as mentioned in Section 1.2, domain decomposition is used for partitioning.

Issue 1 - Partitioning an Irregular Domain

If the load is evenly distributed over the data domain, the decomposition would best partition the domain into subdomains of equal size. One straightforward way is to assign elements {1,2, .. $\lfloor N/p \rfloor$} to processor 1, elements {$\lfloor N/p \rfloor$+1 ... $\lfloor (2N)/p \rfloor$} to processor 2, etc., where N is the number of elements in the data domain and p is the number of processors in the system. However, for irregular applications, such as CRASH-SP (see Section 1.4), this simple decomposition often leads to enormous communication traffic and poor load balance. Hence, more sophisticated domain decomposition algorithms are often used to partition the domain for better performance.

Action 1 - Applying a Proper Domain Decomposition Algorithm for (I-1)

Optimal partitioning schemes are computationally difficult to find, and most domain decomposition algorithms use heuristics that attempt to partition the problem into load balanced subdomains and minimize the communications between subdomains. Some example decomposition algorithms are: *binary decomposition, recursive coordinate bisection (RCB), recursive graph bisection (RGB), recursive spectral bisection (RSB), greedy algorithms* and *simulated annealing*. References [104][105][106] provide comparisons of several algorithms. None of the algorithms consistently produces the best partitioning.

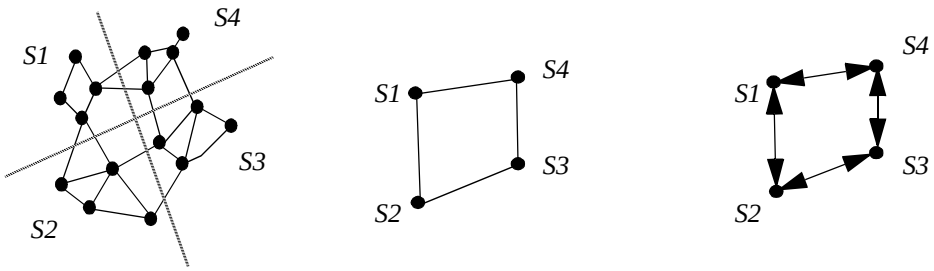
In this step, domain decomposition is employed primarily for the purpose of partitioning the problem. It can be difficult for the user to optimize domain decomposition initially without going through the later steps. Fine tuning of domain decomposition in the later steps is usually required to remedy the imperfections of the partition. Further discussion is provided in the sections devoted to later steps.

3.2.1 Implementing a Domain Decomposition Scheme

In this subsection, we address the following questions. What information do domain decomposition algorithms require us to extract from the application? How do we implement a domain decomposition scheme in the application? Does the domain decomposition scheme cause extra run time overhead?

First, the elements of the data structures associated with the domain and the connections among these elements are identified and abstracted to a graph (referred to hereafter as a *domain graph*) that captures most of the aspects of the data domain that are most relevant to decomposition. A domain decomposition algorithm partitions the domain graph into sub-graphs by providing the user with descriptions of these sub-graphs as well as the communication dependence between sub-graphs which are represented by cut edges, as shown in Figure 3-3(a).

For static applications whose load distribution does not vary during the runtime, domain decomposition needs to be performed only once per application, and the cost for domain decomposition is usually negligible. For dynamic applications that require periodic re-partitioning of the domain during a run, the frequency (and hence cost) of domain decomposition



(a) domain decomposition *(b) connectivity graph* *(c) communication dependence graph*

Figure 3-3: Domain Decomposition, Connectivity Graph, and Communication Dependency Graph

```
      program CRASH-SD

      ....
      call Domain_decomposition_algorithm(Neighbor, Num_Elements,
             Num_Subdomains, Num_Elements_in_subdomain, global_id)
      t=0

c First phase: generate contact forces
100   doall d=1,Num_Subdomains
      do ii=1,Num_Elements_in_subdomain(d)
        i=global_id(ii,d)
        Force(i)=Contact_force(Position(i),Velocity(i))
        do j=1,Num_Neighbors(i)
            Force(i)=Force(i)+Propagate_force(Position(i),Velocity(i),
              Position(Neighbor(j,i),Velocity(Neighbor(j,i))
        end do
      end do
      end do

c Second phase: update position and velocity
200   doall d=1,Num_Subdomains
      do ii=1,Num_Elements_in_subdomain(d)
        i=global_id(ii,d)
        type_element=Type(i)
        if (type_element .eq. plastic) then
            call Update_plastic(i, Position(i), Velocity(i), Force(i))
        else if (type_element .eq. glass) then
            call Update_glass(i, Position(i), Velocity(i), Force(i))
        end if
      end do
      end do

      if (end_condition) stop
      t=t+t_step
      goto 100
      end
```

Figure 3-4: A Shared-memory Parallel CRASH (CRASH-SD)

should be traded off against the performance gained by the improved balance. This is further discussed in Section 3.8 (Step 7).

In Figure 3-4, we show a shared-memory parallel version of CRASH (CRASH-SD) with domain decomposition (SD stands for **S**hared-memory **D**omain-decomposed). Note that the **doall** statements are equivalent to the **c$dir loop_parallel** directives in Convex Fortran, but **doall** is used hereafter for its simplicity. CRASH-SD calls **Domain_decomposition_algorithm()** to partition the domain graph, which is specified by array **Neighbor(*,*)**, into **Num_Subdomains** subdomains. The decomposition returns the number of elements in each subdomain in the array **Num_Elements_in_subdomain**. Subdomain *d* owns **Num_Elements_in_subdomain(d)** elements, and the original/global identifier of *ii-th* element of subdomain *d* is stored in **global_id(ii,d)**. The **doall** parallel directives perform the loops in parallel such that each processor handles the computation of an equal (or nearly equal) number of subdomains, usually one. A loop with a doall directive forms a *parallel region*, where the processors all enter and leave the region together. This program runs on shared-memory machines without the need to specify interprocessor communications explicitly.

For message-passing programming, the communication dependence information from the domain decomposition algorithm is used to specify explicit messages. Such communication dependence information can be considered as a *connectivity graph*, where each subdomain (sub-graph) is represented by a vertex and the communications between subdomains are represented by edges, as shown in Figure 3-3(b). Given the data access behavior of the program, a connectivity graph is transformed to a *communication dependence graph* (CDG), as shown in Figure 3-3(c), which determines the communication between each pair of processors. There are elements near the boundaries of subdomains that need to be referenced by processors other than their owners.

In this section, a pseudo message-passing code is used to illustrate message-passing, as shown in Figure 3-5. The communication is orchestrated in three steps: (1) *gathering*, (2) *exchanging*, and (3) *scattering*. Gathering and scattering is used to improve the efficiency of data exchange. In the gathering step, each processor uses a list **boundary(*,my_id)** to gather the contact forces of the *boundary elements* in its subdomain into its *gathering buffer*,

```
c First phase: generate contact forces
100   do ii=1,Num_Elements_in_subdomain(my_id)
        i=global_id(ii,my_id)
        Force(i)=0
        do j=1,Num_Neighbors(i)
            Force(i)=Force(i)+Contact_force(Position(i),Velocity(i),
              Position(Neighbor(j,i),Velocity(Neighbor(j,i))
        end do
      end do

c Gather contact forces
      do ii=1,Num_boundary_elements(my_id)
        buffer(ii,my_id)=Force(boundary(ii,my_id))
      end do
c Exchange contact forces
      do p=1,Num_Proc
        if (p .eq. my_id) then
            broadcast(buffer(1,p),1,Num_boundary_elements(my_id))
        else
            receive(p,buffer(1,p))
        end if
      end do
c Scatter contact forces
      do p=1,Num_Proc
        if (p .ne. my_id) then
          do ii=1,Num_boundary_elements(p)
            Force(boundary(ii,p))=buffer(ii,p)
          end do
        end if
      end do
```

Figure 3-5: A Message-passing Parallel CRASH (CRASH-MD), A Psuedo Code for First Phase is shown.

buffer(*,my_id), where **my_id** is the processor ID, ranging from **1** to **Num_Proc**. During the exchanging step, each processor counts synchronously with variable **p** from **1** to **Num_Proc**. At any time, only the processor whose processor ID (**my_id**) matches **p** broadcasts the data in its own gathering buffer to all the other processors, and all the other processors receive. At the end of the exchanging step, all the processors should have identical data in their gathering buffers. Then in the scattering step, each processor updates its copy of array **Force** by reversing the gathering process, i.e. *scattering* **buffer(*,p)** to **Force**. Broadcasting is used in this example for simplicity, but it can be replaced with other communication mechanisms for better performance, as discussed in Section 3.3.

3.2.2 Overdecomposition

To solve a problem efficiently on *N* processors, we need to decompose the problem into at least *N* subdomains. The term, *overdecomposition*, is used here to refer to a domain decomposition that partitions the domain into M (M>N) subdomains. A symmetric overdecomposition

contains a multiple of *N* subdomains, i.e. *k*N*, where *k* subdomains are associated with each processor. For *multithreaded processors*, overdecomposition can be a convenient and efficient strategy for generating multiple threads for each processor. Overdecomposition is also used in the following steps.

3.3 Tuning the Communication Performance (Step 2)

3.3.1 Communication Overhead

From a processor's point of view, the communication overhead, without overlap, is approximately: *(communication traffic)*(average communication latency)*. Reducing the communication traffic and reducing the average communication latency should result in less communication overhead. Also, since the communication network may provide limited connectivity and bandwidth, the user should try to avoid causing high communication traffic to the network or to any processor.

In this section, we discuss the techniques for reducing the communication overhead. We consider three classes of approaches: (1) *reducing the communication traffic*, (2) *reducing the communication latency*, and (3) *avoiding network contention*. We discuss code/data restructuring techniques and investigate possible machine enhancement features that can help reduce the communication overhead. It is important to analyze the communication patterns and evaluate the communication costs before applying the techniques because some techniques require more programming work and some generate extra computation overhead.

3.3.2 Reducing the Communication traffic

For reducing the communication traffic, we consider two goals: *reducing the amount of data to be communicated*, and *improving the efficiency of the communication traffic*. Note that most issues and techniques in this section are discussed primarily for shared-memory applications. Message-passing applications, thanks to their explicit communication patterns, are less prone to the problems discussed here, but more dependent on the programmer's skill. An inefficiently written message-passing program can generate superfluous communications, while in a shared-memory program, some superfluous communications are eliminated, and others are generated due to cache effects.

3.3.2.1 Reducing the Amount of Data to Be Communicated

Issue 2 - Exploiting Processor Locality

Besides spatial locality and temporal locality, another form of locality that is important in a multiprocessor context is processor locality - the tendency of a processor to access a data item repeatedly before an access to this item from another processor. *Processor locality* is an important key for reducing the amount of data that needs to be communicated. For a data item that shows good processor locality, it is best distributed to a location near the processor(s) that will use the data item soonest and most frequently so as to reduce the traffic caused by accessing the item.

Action 2 - Proper Utilization of Distributed Caches for (I-2)

Distributed caches are often used as a hardware solution for facilitating processor locality by moving the data close to whichever processor accesses the data. Distributed caches provide a convenient solution that can take advantage of the processor locality in the application. With distributed caches, the programmer can pay less attention to the physical data layout and data distribution.

However, distributed caches may cause extra overhead for cache coherence operations, and thus degrade the performance unless sufficient processor locality exists. Therefore, those data items that exhibit poor processor locality, e.g. data that migrate frequently from one processor to another, may yield better performance if not cached.

Action 3 - Minimizing Subdomain Migration for (I-2)

Subdomain migration may occur if the processor assignment for a subdomain is not specified by the program. For example, a doall statement can arbitrarily assign the tasks in the loop to the processors. In a self-scheduling or dynamic scheduling scheme, the task-processor mapping is decided by the runtime system. In our experiences with KSR1/2 and HP/Convex Exemplar, we found that subdomain migration can seriously degrade processor locality. To avoid unnecessary subdomain migration, an explicit specification of the mapping of subdomains to processors, which permanently binds data items to specific processors under program control is best used.

Issue 3 - Minimizing Interprocessor Data Dependence

Even with good processor locality, processors still need to communicate in order to satisfy interprocessor data dependence. When a program is parallelized, some of the data dependencies in the serial program become interprocessor data dependencies in the parallel version. For reducing the amount of communication, interprocessor data dependency should be minimized.

Action 4 - Minimizing the Weight of Cut Edges in Domain Decomposition for (I-3)

Most domain decomposition algorithms attempt to minimize the number, or better yet, the total weight, of the cut edges. Minimizing cut edges reduces the data dependencies between subdomains, which in turn reduces interprocessor data dependencies between the corresponding processors.

3.3.2.2 Improving the Efficiency of the Communication Traffic

Issue 4 - Reducing Superfluity

A block is defined here as the unit of data transfer for communications. For example, the block size is 256 bytes (subpage) in the KSR1/2, or 64 bytes in the Convex Exemplar. A large block size can take advantage of the spatial locality in the code and amortizes the latency overhead if the entire block will eventually be used by the processor. However, it is inefficient for a processor to acquire a block from another processor if only part of the block is used before it is invalidated (or replaced) in a finite size cache, which shows *superfluity*. Superfluity can be measured by averaging the number of unused words in an invalidated block. Reference [107] has defined a metric, the *superfluity coefficient*, for measuring the efficiency of cache-memory traffic in uniprocessor systems. We extended the use of the superfluity coefficient for measuring the efficiency of communication traffic in parallel systems:

$$SFC = \frac{\sum_{i=0}^{s-1} i \times Freq_i}{s \times \sum_{i=0}^{s} Freq_i}$$

where $Freq_i$ is the number of block invalidations that contain i unused words and s is the

block size. *SFC* equals 0 if every word in each loaded block is referenced at least once before the block is invalidated. *SFC* equals *(s-1)/s* if only one word is referenced in each loaded block. High superfluity coefficients indicate poor efficiency of the communication traffic. Rivers and Davidson [107] have shown the use of the superfluity coefficient for locating spatial locality problems in serial executions. In this research, we propose to use the superfluity coefficient for exposing *spatial and processor locality problems*.

Serious superfluity problems may be caused for two major reasons:

- *Inefficient layout of the data structures* may result in poor spatial locality. *Array padding* is often used to eliminate false-sharing and align memory references on 32-bit, 64-bit, or cache-line-size boundaries, but it causes superfluity since cache blocks that contain pad elements are not entirely used.

- *Frequent invalidation/update traffic*: False-sharing occurs if different portions of a cache block are owned by different processors. When one processor writes to a false-shared block, it may be required to invalidate (or update) copies of the cache block in the caches of other processors, which invalidates, or redundantly updates, portions of the block that may still be needed by those processors. In addition, the acquisitions and reaquisitions of such blocks cause superfluity in the caches of the acquiring processor, since some portions of false-shared blocks are not needed by the acquiring processor.

Issue 5 - Reducing Unnecessary Coherence Operations

Unnecessary coherence operations occur when processor locality in a code is different from what the cache coherence protocol expects. Under write-update protocols, consecutive writes to the same block by a processor may generate unnecessary update traffic if no other processors read that block during those writes. Write-invalidate protocols may result in redundant invalidation traffic, e.g. for the producer-consumer sharing patterns that we discussed in Section 2.4.5. Thus, depending on the pattern of accesses to shared data in the program, certain cache coherence protocols may result in more efficient communication traffic than others. Some adaptive protocols have been proposed that combine invalidation and update policies in a selectable fashion [108]. Eliminating some unnecessary coherence operations may also reduce superfluity because of less frequent invalidate/update traffic.

Action 5 - Array Grouping for (I-4)

A common solution for reducing superfluity is to group certain data structures that are used by a processor in the same program regions. Grouping several arrays into one interleaved layout can be done statically by redefining the arrays in the source code, or dynamically by gathering and scattering the arrays during runtime. Static methods usually rely on source code analysis and may require permanently changing the layout of global variables. Dynamic methods reduce superfluity locally without interfering with other program regions, but introduce extra overhead in gathering and scattering. A systematic method can be found in [45][50].

Action 6 - Privatizing Local Data Accesses for (I-5)

Making a *private copy*[1] of data items in false-shared blocks can avoid false-sharing, if the item is repeatedly accessed solely by one processor in some portion of the program. Figure 3-6 shows the use of private copies to enhance data locality within a processor and reduce false-sharing. Before the main loop starts, each processor makes a private copy of the data structures with local indices, e.g. *p_Force(i)* is a private copy of *Force(global(i,d))* for subdomain *d*. Before computing the forces, each processor acquires remote data (*Input*) by copying the data from globally shared variables to private variables. After the forces are computed, each processor updates the shared forces (*Output*) by copying the data from private forces (*p_Force*) to globally shared forces (*Force*). The arrays, *Input* and *Output*, list the input variables and output variables.

Since the private data structures are indexed with local indices, the *spatial locality* is improved for the access patterns in the loops that iterate with local indices. *False-sharing* is reduced because accesses to private data structures do not cause communications. If each data element is smaller than the block size and the access is nonconsecutive, as mentioned in Section 3.2.1, gathering and scattering can be used to improve the efficiency. Figure 3-7 shows an example of using a set of buffers (gather_buffer) for gathering and scattering forces.

Since the integrity of private data copies is no longer protected by the coherence mechanism, updates are now explicitly specified in the program in a fashion similar to those used in the message-passing code in Figure 3-5. It should be noticed that privatization,

1. *Convex Fortran* allows the programmer to use *thread-private* directives to generate private copies of variables so that when different threads access the same variable name, they actually access their private copy of that variable.

```
c$dir thread_private(p_Force,P_Position,p_Velocity,p_No_of_neighbors)
c$dir thread_private(p_Neighbor, Input, output)

c Make private copies
      doall d=1,Num_Subdomains
      do ii=1,No_of_elements_in_subdomain(d)
        i=global_id(ii,d)
        p_Force(ii)=Force(i)
        p_Position(ii)=Position(i)
        p_Velocity(ii)=Velocity(i)
        p_No_of_neighbors(ii)=No_of_neighbors(i)
        do j=1,p_No_of_neighbors(ii)
            p_Neighbor(j,ii)=Neighbor(j,ii)
        end do
      end do
      call Initialize_Input_Output(Input,Output)
      end do

c First phase: generate contact forces
100   doall d=1,Num_Subdomains
c Acquire input elements
      do i=1,No_of_input_elements(d)
        p_Position(Input(i))=Position(global_id(Input(i)))
        p_Velocity(Input(i))=Velocity(global_id(Input(i)))
      end do
      do i=1,No_of_elements_in_subdomain(d)
        p_Force(i)=Contact_force(p_Position(i),p_Velocity(i))
        do j=1,p_No_of_neighbors(i)
           p_Force(i)=p_Force(i)+
             Propagate_force(p_Position(i),p_Velocity(i),
             p_Position(p_Neighbor(j,i),p_Velocity(p_Neighbor(j,i))
        end do
      end do
c Make output available to other processors
      do i=1,No_of_output_elements(d)
        Force(Output(i))=p_Force(global_id(Output(i)))
      end do
      end do
      .....
      goto 100
```

Figure 3-6: Using Private Copies to Enhance Locality and Reduce False-Sharing

gathering and scattering generates extra computation overhead for calculating indices and local copies, and an extra burden on the software to assure their coherence with respect to the original globally shared structure. Therefore, copying should not be used when it is not necessary, i.e. when the communication overhead is not a serious problem or can be solved sufficiently by other means with less overhead.

```
      ....
c Make output available to other processors
      do dd=1,Num_Subdomains
        if (dd.ne.d) then
            do i=1,No_of_comm_elements(dd,d)
               gather_buffer(i,dd,d)=p_Force(Output(i))
            end do
        end if
      end do
      end do

c Second phase: update position and velocity
200   doall d=1,Num_Subdomains
c Acquire input elements
      do dd=1,Num_Subdomains
        if (dd.ne.d) then
            do i=1,No_of_comm_elements(dd,d)
               p_Force(Input(i))=gather_buffer(i,dd,d)
            end do
        end if
      end do
      ....
```

Figure 3-7: Using Gathering Buffers to Improve the Efficiency of Communications

Action 7 - Optimizing the Cache Coherence Protocol for (I-5)

Figure 2-12(c) shows an example of producer-customer communication patterns under a write-invalidate protocol. For the same program running on a system using a *write-update* protocol, the communication pattern would look like Figure 3-8, where each copy-invalidation pair is replaced with a single update. Suppose each of the consumers reads the production variable (*x*) once after the producer writes *x*, and then the producer writes *x* once before the consumers read *x* again, using a write-update cache coherence protocol may result in better performance than using a write-invalidate protocol, since the number of transactions is less when a write-update protocol is used. Also, update can be performed more efficiently if the system supports multicasting and/or asynchronous (nonblocking) updates. However, if the producer writes *x* multiple times consecutively, a write-update protocol may generate more transactions than a write-invalidate protocol does, since the producer is required to update *x* for each write to *x* under a write-update protocol. Although there are variations or hybrids of the above two protocols that have been developed to solve the above problems, none of them constantly performs best for all kinds of applications.

It should be noted that *privatization* (see (A-6)) can significantly simplify the access patterns of global variables and hence reduce the chances of consecutive writes to one variable. Thus, the performance of a write-update protocol may be improved by privatizing

those shared variables and updating the global copy only once prior to the accesses by readers. In addition, the release consistency memory model (see (A-9)) can help reduce the number of updates required by consecutive writes.

It is possible to reduce the invalidation/update traffic if the user is allowed to choose a suitable protocol, either for an entire application, block by block, or even dynamically during the run. Some machines have certain features, such as automatic update in the KSR1/2, which may automatically update invalidated copies that lie somewhere along the return path of a read miss. For such machines, it is possible to make the communication traffic more efficient by reallocating the threads among the processors or rearranging the access patterns so as to automatically update as many copies as possible [48]. With trace-driven simulation tools like K-Trace/K-LCache, the performance of each protocol can be evaluated and used to help choose suitable protocols.

Action 8 - Cache-Control Directives for (I-5)

If the communication patterns are known during compile-time, unnecessary coherence operations may be reduced by overriding the regular coherence protocol with *cache-control directives*, when such directives are supported by the system.

For example, we can design a machine could be designed to support *noncoherent memory instructions* that load or store data that reside in a remote cache without bringing the data into the local cache and modifying the status of the data in the remote cache. Some systems have *noncacheable (or cache-bypass)* memory regions, e.g. for memory-mapped I/O addresses, whose contents are not cached, primarily because the data in those areas are updated frequently by different processors (or I/O devices) between consecutive accesses. For certain shared-memory machines whose hardware does not enforce cache coherence, such as *Cray T3D*, programmers must manually insert cache control directives, such as *flush*, to periodically ensure the cache coherence in their programs.

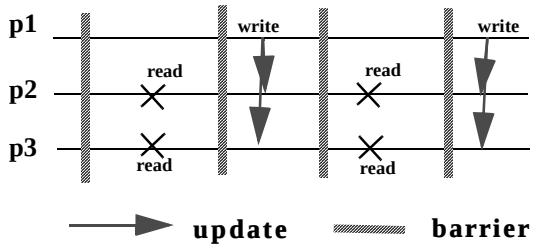


Figure 3-8: Communication Patterns of the Ocean Code with Write-Update Protocol.

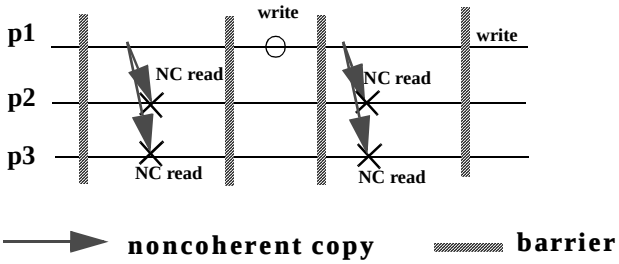


Figure 3-9: Communication Patterns of the Ocean Code with Noncoherent Loads.

Noncoherent memory accesses and *noncacheable data structures*, if they are supported by the system, may be used to reduce unnecessary invalidation/update traffic in producer-consumer patterns. Figure 3-9 shows how the invalidation in a producer-consumer pattern can be eliminated with noncoherent reads. An *NC* (*N*on*C*oherent) read always copies a block from the owner, *p1*, but does not change the status of the block in *p1*. The programmer is responsible for the use of any data items in that block in *p2* and *p3* since writes from *p1* will neither update nor invalidate that block. In this case, the noncoherent copies can be safely used in those regions where updates of the block do not occur.

At one extreme, the programmer can use noncoherent memory instructions (or cache-bypass) and privatization to completely and explicitly control the coherence traffic, as if programming for a non-cache-coherent or message-passing machine. Such a programming style is similar to what we have shown in Figure 3-6 and Figure 3-7, except that the program carries out interprocessor communications by using noncoherent instructions. For example, the copying from **`gather_buffer`** to **`p_Force`** in Figure 3-7 can be implemented by NC reads, which read the contents of **`gather_buffer`** without affecting the exclusive ownership of **`gather_buffer`**. Thus, writing gather_buffer later by owners would not generate invalidation traffic.

However, cache-control directives are not popular in commercial machines, due to their complexity of implementation and difficulty of use. The instruction *ld.ex* in the KSR1/2 performs a load, but also acquires the exclusive ownership of the cache block for the load, which is useful when the load is soon followed by a store to the same memory location. A study of the Convex SPP-1000 by Abandah and Davidson [10] shows that shared-memory point-to-point communication performance could be improved by 53% if the communication variables are not cached. In addition to source code analysis, the communication patterns extracted by K-Trace/K-LCache can be used to decide where to apply such instructions in the program.

Action 9 - Relaxed Consistency Memory Models for (I-5)

Most distributed shared-memory machines today, like KSR1/2 and Convex Exemplar, maintain a *sequential consistency* memory model, where coherence operations are issued, roughly speaking, for every write to a shared memory that has other valid copies outstanding. For many applications, sequential consistency is unnecessarily restrictive, and more relaxed memory consistency models can be used to reduce unnecessary coherence operations.

One of the most relaxed memory models, the *release consistency* model, issues coherence operations for every synchronization operation instead, generally generates fewer coherence operations (mainly for *writes*) than sequential consistency. *Eager release consistency* models send coherence operations to all processors to inform them that cache data that has been modified by the releasing processor. In *lazy release consistency*, however, messages travel only between the last releaser and the new acquirer, which results in even fewer coherence operations.

However, release consistency does increases hardware and programming complexity, thus it is not popular for implementation on commercial hardware cache coherent shared-memory machines. On the other hand, studies show that the performance of some *software-emulated virtual shared-memory systems* benefit greatly from relaxing their memory consistency models [109][110]. Depending on the application, the performance improvement resulting from using release consistency ranges from a few percent to several hundred percent, on systems where coherence operations usually take milliseconds to complete.

Action 10 - Message-Passing Directives for (I-5)

The code in Figure 3-6 is programmed in a *message-passing* style. If the machine supports both message-passing and shared-memory programming, this capability raises a question: Which communication mechanism leads to better communication performance? Usually, the block size and latency of shared-memory accesses are smaller than typical message sizes and latencies, but the bandwidth of the message channels may be higher. Communicating via messages can eliminate unnecessary cache coherence traffic, if the message-passing library is optimized by using a proper coherence protocol or cache control directives, as discussed in (A-7) and (A-8). Using the communication patterns exposed by the K-trace/K-LCache tools, automatic program conversion from shared-memory to message-passing has been experimented with on the KSR1/2 with encouraging results [102]. A judicious mixture of shared-memory accesses and messages in a program would generally result in the best communication performance; performance models of each communication mechanism should be used to determine where each mechanism should be used.

3.3.3 Reducing the Average Communication Latency

For reducing the average communication latency, we consider three approaches: (1) *reducing the distances of communication*, (2) *hiding the communication latencies*, and (3) *reducing the number of transactions*. The first approach minimizes latencies of communications by moving data to locations near the processor that accesses the data. The second approach overlaps communication with computation (and with other communications) either by starting communications earlier than when the information is required, or by starting independent computation when the processor is waiting for the communication to complete. We also discuss how reducing the number of transactions reduces communication overhead, such as the use of gathering and scattering in message-passing programming to reduce the number of times that the overhead of setting up communication channels is incurred. It is useful for shared-memory programs if the machine supports efficient accesses to long-blocks in memory. In this section, we consider the communications that remain after applying the communication traffic reduction techniques discussed in the previous section.

Issue 6 - Reducing the Communication Distance

The communication distance may significantly affect communication latency in some interconnection topologies; in others, it is an insignificant factor. Topologies such as *buses*, *crossbars*, *multistage networks*, and *unidirectional rings* have near constant communication latency overall processor-memory pairs, provided that network contention effects are minimal (the effects of network contention are discussed in Section 3.3.4). Communication latency on some topologies like *meshes*, *hypercubes* and *bidirectional rings*, or *hierarchical* topologies, depends on the distance (or the number of hops) in the path. For such systems, reducing the distance of communication may be of interest when hop times are significant.

Action 11 - Hierarchical Partitioning for (I-6)

In machines with hierarchical communication structures, the interprocessor data dependencies that can cause higher communication latencies should be minimized first. For example, on a 32-processor Convex Exemplar, domain decomposition should be performed by first partitioning the domain into 4 large subdomains for the 4 hypernodes, and then partitioning each large subdomain into 8 small subdomains for the processors within each hypernode.

Action 12 - Optimizing the Subdomain-Processor Mapping for (I-6)

Given a decomposition and a communication dependence graph, finding a mapping that optimizes the performance for the network topology of interest can be computationally expensive if all possible mappings need to be evaluated. There are p^n possible mappings for n subdomains onto p processors. For each mapping, the total communication latency for each pair of subdomains is calculated by multiplying the distance between the two processors where two subdomains reside by the number of communications between those two subdomains.

For large systems, heuristic algorithms may be used to optimize the mappings. Some domain decomposition packages, e.g. Chaco [28], use heuristics to improve mappings to hypercube and mesh architectures. For a particular domain, they first perform decomposition and then attempt to optimize the subdomain-processor mapping.

However, when decomposition and mapping are performed separately, the “best” decomposition followed by the best mapping for that partitioning does not necessarily result in the best overall performance. Finding an optimal mapping that assigns each domain ele-

ment to a processor is more complex than grouping elements into subdomains and then assigning the subdomains to processors, which is why domain decomposition packages normally choose to decompose into subdomains first.

Issue 7 - Hiding the Communication Latency

In an application, some communications and computations can, at least theoretically, be executed in parallel, as long as they are mutually independent. The idle time that a processor spends waiting for communication can be reduced if the communication is executed well in advance of the need, or the processor executes some independent computations during the wait.

Action 13 - Prefetch, Update, and Out-of-order Execution for (I-7)

There are various techniques for hiding communication latency on shared-memory machines. Some machines support *prefetch* and *update* instructions that can be used to move data close to a processor before it actually needs that data. Prefetching is useful for hiding communication latency when the communication pattern is predictable and the distance between the prefetch and the instructions that actually need that data is far enough [52][51]. Unfortunately, most commercial shared-memory machines we have worked with do not allow a large number of pending prefetches, which can make prefetching an inefficient technique. It has been shown that a great reduction (53%-86%) in coherence cache misses can be achieved on the KSR1 by using prefetch for a finite element code, but the total execution time still may not be significantly reduced due to the overhead of the prefetch instructions [51].

HP/Convex's latest Exemplar, SPP-2000, is capable of executing 10 outstanding cache misses and 10 prefetches simultaneously on each of its PA 8000 processors [7]. As with prefetching, supporting out-of-order execution while there are outstanding cache misses allows the processor to hide some of the cache miss latencies. However, the maximum number of outstanding misses allowed (10), the size of the instruction reorder buffer (56), and the data dependencies in the application interact to limit the effectiveness of this approach in hiding communication latency. In comparison, prefetches can be issued well ahead of the use of data to hide more latency; but prefetch instructions increase the load on the instruction fetching unit, and may pollute the data caches if used unwisely or too early.

Action 14 - Asynchronous Communication via Messages for (I-7)

It is possible to realize asynchronous communications on hierarchical shared-memory machines by, for example, dedicating one processor in a cluster for communicating with other clusters. When the traffic among clusters is heavy, dedicating one processor for communication may improve overall performance. If the machine supports message-passing in shared-memory programs, asynchronous communication directives can also be used for hiding communication latencies.

Action 15 - Multithreading for (I-7)

Multithreading can also be used to hide communication latencies [111][112]. Whenever one thread issues a communication operation, the processor then quickly switches to another thread. To be effective, the latency of thread switching should of course be much smaller than that of the communication operation. *Multithreaded* or *multiple-context processors* that minimize *thread switching* time are required for high performance shared-memory systems to utilize this multithreading solution. For software shared memory systems implemented on networks of workstations, in which shared memory operations are slower, multithreading can be competitive with less hardware support and may achieve higher relative performance improvement than for shared memory systems with faster hardware-based sharing. Multithreading can also be applied to message-passing codes.

Issue 8 - Reducing the Number of Communication Transactions

Part of the communication latency is spent in setting up the communication channel and synchronizing the senders and the receivers. Such *initialization* overhead is proportional to the number of communication *transactions*. Therefore, it would be more efficient to satisfy the communication that is required by the application with fewer transactions, e.g. by grouping multiple messages into one message or using long-block memory accesses.

Action 16 - Grouping Messages for (I-8)

To reduce the number of communication transactions, one technique that has often been employed in message-passing codes is *grouping messages*. For example, some messages that are sent and received by the same pair of processors can be combined into one message. Some messages that are passed to different processors by one processor can be *combined* and *multicasted* with one message from the sender. Some messages that are passed to the same processor by different processors can also be combined en route and received

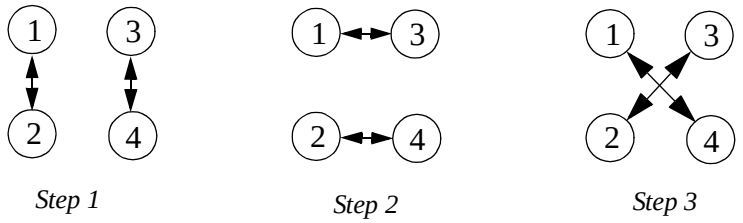


Figure 3-10: Example Pairwise Point-to-point Communication

as one message, e.g. by using *reduction* operations in *MPI*. These operations, as well as *gathering* and *scattering*, are supported in MPI for reducing the number of messages, which are also referred to as *collective communication functions* in [113].

Action 17 - Using Long-Block Memory Access for (I-8)

Unlike messages, communications in a shared-memory machine are usually performed by passing relatively short, fixed-size cache blocks. While a short block size reduces the latency for a single transaction somewhat, and helps reduce false-sharing and superfluity problems, transfering a large block of data using many transactions is rather inefficient, because there is latency overhead due to issuing the supporting instructions and to processing the cache coherence protocol for each transaction. *Long-block memory access*, if supported by the system, may be used for reducing the latency overhead of transferring large blocks of data. However, the programmer should avoid causing significant false-sharing and superfluity problems.

3.3.4 Avoiding Network Contention

Issue 9 - Distributing the Communications in Space

Network contention occurs *spatially* when a processor/memory node is simultaneously accessed by multiple processors. Therefore, to avoid network contention, it is better to distribute the communications in space.

Action 18 - Selective Communication for (I-9)

Depending on the machine and the degree of data-sharing, broadcasting can be more or less efficient than *point-to-point communication*, where each processor communicates individually with those processors that require its data (Figure 3-10). Point-to-point communication may cause more overhead if a processor often needs to communicate with

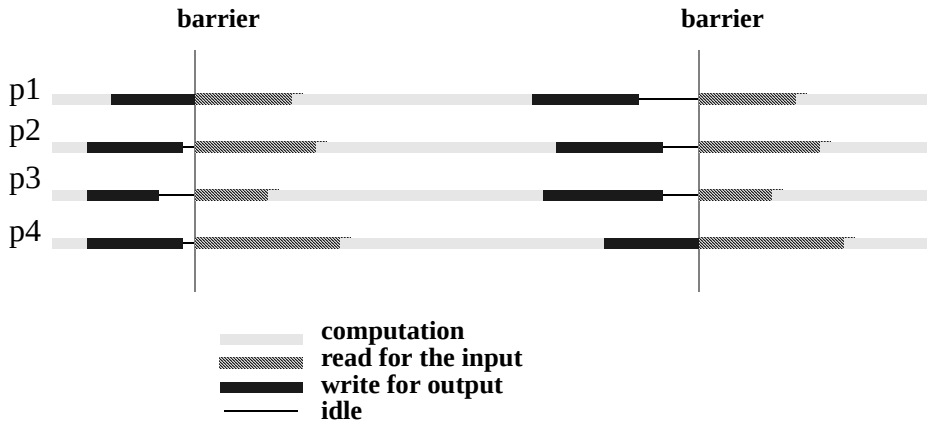


Figure 3-11: Communication Patterns in a Privatized Shared-Memory Code

many processors, but it may save bandwidth if there are only few processors that actually need its data or if other processors need small distinct portions of its data. Compared to broadcasting, pairwise point-to-point communication may perform better in many cases, but requires more complicated programming.

Issue 10 - Distributing the Communications in Time

Network contention occurs *temporally* when the network bandwidth (or number of channels) is saturated. Therefore, to avoid network contention, it is better to distribute the communications in time.

The shared-memory CRASH code examples CRASH-SP and CRASH-SD, shown in Figure 1-4 and Figure 3-4, respectively, have distributed communication patterns, where the remote loads and stores are interleaved with the computation. However, the message-passing version (CRASH-MP) in Figure 3-5, and the privatized code in Figure 3-6 group their communications near the beginning and end of parallel regions. Figure 3-11 shows a typical communication pattern for a privatized shared-memory code. At the beginning of a parallel region, every processor reads its required input data from other processors. At the end of a parallel region, every processor updates its output data that will be used by other processors in some later parallel region. Network contention may occur when the communication traffic is high during these input and output activities, while the network idles during the computation-dominated time in the middle of a parallel region.

Action 19 - Overdecomposition to Scramble the Execution for (I-10)

Overdecomposition may also serve as a method for interleaving communication with computation. When the number of subdomains assigned to one processor is increased, the degree of interleaving between communication and computation also increases. However, this approach does introduce extra communication overhead due to the increased number of messages and shorter message lengths, which may need to be traded-off against the benefit of reducing network contentions.

3.3.5 Summary

In the previous chapter, we presented the *trace-driven simulation* techniques that we developed for exploiting communications in a shared-memory code running on distributed cache systems. We also showed how *communication patterns* can be extracted from a shared-memory trace. Knowing the communication patterns, various techniques can be used for tuning communication performance. These techniques are classified into three categories: *reducing traffic*, *reducing latency*, and *avoiding network contention*. We have discussed how and when each technique can improve the performance.

Cache coherence protocols and *data layout* affect the communication patterns in subtle ways without the user necessarily being able to discover that this is happening. Trace-driven simulation exposes the communication patterns so that coherence communication and data layout can be improved by employing a suitable coherence protocol, cache-control instructions, and array grouping techniques. Trace-driven simulation also serves as a tool for evaluating the communication performance of parallel machines, and determining the magnitude of this problem.

Domain decomposition has played an important role in many performance-tuning techniques. Domain decomposition can enhance *data locality*, which reduces communication traffic. It also enables *privatization*, which may improve communication efficiency. A shared-memory code can gradually be converted, or partially converted, into a *message-passing* code by full or partial privatization, respectively. With explicit control of communication, overlapping communication and computation can be achieved using asynchronous communication directives.

For machines with hierarchical communication structures, e.g. KSR1/2 and Convex Exemplar, the communication problem should be dealt with hierarchically by first reducing the communication traffic in the level with the highest latency, and then considering the next levels in turn.

We discussed the use of *multithreading* for hiding communication latencies, as well as the use of *overdecomposition* for reducing network contention via increased interleaving of communication with computation. Overdecomposition may serve as one convenient and effective method for providing multiple threads to a multithreaded processor. More discussion about overdecomposition and multithreading is found in Sections 3.6 and 3.7, where we discuss these two approaches from the viewpoints of load balance, scheduling and synchronization.

3.4 Optimizing Processor Performance (Step 3)

Many sequential code optimization techniques have been developed, and discussion of them is beyond the scope of this dissertation. Fortunately, a large portion of these techniques have been implemented in many uniprocessor compilers, so users are less concerned with them. Here, we address some of the issues that are specifically related to hand-tuning parallel applications.

Issue 11 - Choosing a Compiler or Compiler Directive

To improve processor performance the use of an *optimizing compiler* is critical. Surprisingly, some parallel compilers sacrifice serial optimization for parallel features. For example, the *HP Fortran* sequential compiler consistently outperforms the *Convex Fortran* parallel code compiler for the Exemplar in the quality of an individual processor's machine code in many of our test cases, despite the fact that they both generate codes for the *PA-RISC* architecture. Some codes compiled by the *KSR Fortran* compiler actually produce incorrect execution results when optimization is activated.

Action 20 - Properly Using Compilers or Compiler Directives for (I-11)

Our current solution is to use parallel compilers only where necessary, e.g. the *Convex Fortran* compiler is used only for the routines with parallel directives, since a well-developed sequential compiler, such as *HP Fortran*, tends to deliver higher processor performance. We also gradually increase the compiler optimization level and carefully monitor both the delivered performance and the numerical results of the program.

Action 21 - Goal-Directed Tuning for Processor Performance for (I-11)

Hierarchical bound models could be used to estimate the lower bounds on a processor's run time [45][48]. In case the gap between two successive performance bounds in the hierarchy is large, the developer can identify specific problems and try to close the gap by solving those problems. Profiling tools are useful to measure the actual execution time, count cache misses, and identify the routines that consume the most execution time.

Issue 12 - Reducing the Cache Capacity Misses

For applications with large data sets, *loop blocking* techniques are commonly used to reduce cache capacity misses. However, for CRASH, loop-blocking cannot be applied to the main (*iter*) loop, because one phase needs to be completed before the other can start. Some of the capacity misses can be eliminated, however, by reversing the index order in the second phase, but this is unlikely to eliminate a significant fraction of the capacity misses.

Capacity misses can greatly degrade the performance of applications that repeatedly iterate through the problem domain within each phase. Since these capacity misses may be difficult to eliminate, the memory system performance often becomes the bottleneck of the application's performance. As we mentioned in Section 2.2.1, the processor cache miss ratio of CRASH-SP is about 0.25 on the HP/Convex SPP-1000, which shows that the processor caches do not effectively cache the working set of CRASH-SP.

Action 22 - Cache Control Directives for (I-12)

Some systems allow the programmer to control how to cache the data in load and store instructions. For example, *cache-bypass loads* may be used to selectively load data into the cache in order to utilize the cache more efficiently. The PA 7200 processor in HP/Convex SPP-1600 allows the programmer to specify loads with a "*spatial-locality-only*" hint to indicate which data exhibit only spatial locality, so that the spatial-locality-only data will be loaded to the assist cache, where they typically reside only briefly, instead of competing with longer-lived temporal-locality data in the main cache [6].

Action 23 - Enhancing Spatial Locality by Array Grouping for (I-12)

Array grouping [45], is a software approach to enhancing the spatial locality in the memory access pattern, by selectively grouping the elements in multiple arrays that are accessed within a short period. For example, in CRASH-SD, **`Position(i)`** and **`Posi-`**

tion(i+1) may not necessarily be accessed in consecutive iterations, but it is possible that they are brought into the cache in one cache block that is replaced before the second **Position** element is used. It is better to group **Velocity(i)** and **Position(i)** in the same cache line, because they are always used together in the same iteration.

Array grouping can improve the efficiency of cache-memory traffic by reducing the superfluity in the cache blocks. It also reduce the chances of contention in the cache-memory traffic because cache misses should occur less frequently, e.g. after **Velocity(i)** and **Position(i)** are grouped, at most one cache miss results from accessing both of them.

Action 24 - Blocking Loops Using Overdecomposition for (I-12)

To apply the idea of loop blocking to an irregular application, reference [51] proposed using the *domain decomposition algorithm* to produce subdomains within the subdomain assigned to each processor and execute each small subdomain as a block (the symmetrical overdecomposition mentioned in Section 3.2.2). Compared to the rectangular blocks that result from conventional blocking of the iteration space, such subdomains may have better data locality for irregular applications. This approach is consistent with the way that we used domain decomposition initially for partitioning the problem in order to enhance processor locality and reduce communications. It is usually convenient and efficient to partition a domain into *k*p* subdomains and assign *k* blocks to each of the *p* processors.

Issue 13 - Reducing the Impact of Cache Misses

Inevitably, some cache misses will remain. It is thus important to reduce the impact of these remaining cache misses.

Action 25 - Hiding Cache Miss Latency with Prefetch and Out-of-Order Execution for (I-13)

Prefetch techniques, as mentioned in Section 3.3.3, can be applied to hide cache miss latency. Yet, improper use of prefetch can generate superfluous cache-memory traffic that slows down demand accesses to memory. However, in our example, the capacity miss latency in CRASH can be greatly reduced by scheduling prefetch instructions to make data available in the cache before it is actually needed by the processor. Executing instructions out-of-order also helps hide memory access latency.

The PA7200 processor allows four data prefetch requests to be outstanding at one time and supports a class of load instructions and a prefetch algorithm that can automatically predict and prefetch for the next cache miss [6]. When our CPC machine was upgraded from the PA7100 (HP/Convex SPP-1000) to the PA7200 (SPP-1600), the cache misses in CRASH-SP where reduced considerably due to prefetching with this new class of load instructions.

Action 26 - Hiding Memory Access Latency with Multithreading for (I-13)

Hiding cache miss latency has been one of the major goals for *multithreaded processors*, or *simultaneous multithreaded processors*. When cache misses stall the execution of the running thread, the processor can quickly switch to work on a thread that is available for execution. While compilers may have difficulty decomposing sequential applications into threads, threads can be made plentiful in a parallel application. As mentioned above, overdecomposition is one convenient and efficient strategy to generate enough threads for multithreading within each processor.

Issue 14 - Reducing Conflicts of Interest between Improving Processor Performance and Communication Performance

Certain processor performance optimization techniques may conflict with techniques for tuning communication performance, especially for shared-memory machines. For example, one data layout may be preferred for computation, and another for communication performance. Uniprocessor compilation techniques are often unaware of communications, e.g. messages, cache line invalidations and prefetch instructions, which can change the status of the caches.

Action 27 - Repeating Steps 2 and 3 for (I-14)

The optimization of overall uniprocessor computation and communication performance is more difficult than computation performance alone and requires knowledge of communication patterns, which is why we readdress processor performance issues after tuning the communication patterns. Given knowledge of the resulting communication pattern and the data access pattern in the application, we may be able to optimize the overall communication and processor performance. So far such optimizations are highly case-sensitive, and more research is needed to address this issue better in the future.

3.5 Balancing the Load for Single Phases (Step 4)

Load imbalance affects the *degree of parallelism* (i.e. *efficiency*) achieved in a parallel execution. The performance of a parallel region is limited by the processor with the most work, and the degree of parallelism is imperfect if some processors are idle waiting for the slowest processor. As mentioned above, sophisticated domain decomposition algorithms are often used to balance the load for irregular applications. An optimal partitioning would be a partitioning whose T_{max} (defined for each parallel region in Section 2.2.3) summed over all parallel regions is the smallest among all possible partitions.

For multiple-phase applications, if the node and edge weights are distributed differently in different phases, finding a partition that balances the load for multiple phases is more difficult. We discuss the multiple phase load balancing problem in Section 3.7. In addition, if the load distribution varies dynamically over the runtime, static load balancing techniques may be ineffective. We discuss balancing the dynamic load in Section 3.8.

Issue 15 - Balancing a Nonuniformly Distributed Load

Since finding an optimal partitioning is an NP-complete problem for applications with a nonuniform load distribution, most domain decomposition algorithms use heuristics to partition the problem into reasonably load balanced subdomains and also attempt to minimize the communications between subdomains. Some decomposition algorithms have been mentioned in Section 3.2, yet none of these algorithms consistently produces the best partition. In addition to the selected decomposition algorithm, an accurate *profile* of the load distribution (the node and edge weights provided as input to the decomposition algorithm) determines the quality of a decomposition.

Action 28 - Profile-Driven Domain Decomposition for (I-15)

Tomko [2] has developed a profile-driven approach for determining the weights and improving the quality of the resulting decomposition. Experiments have shown that the profile-driven approach results in good load balance for single-phase irregular applications.

Action 29 - Self-Scheduling for (I-15)

Self-scheduling techniques [114][115] have been used to balance the load of a parallel region. In these approaches, the parallel region contains extra tasks that can be dynamically assigned to balance the load among the processors. However, such dynamic task assignment shifts the locality of the data referenced by each processor, and disrupts the working sets built up in their caches, thereby causing increased communication overhead as the tasks migrate.

3.6 Reducing the Synchronization/Scheduling Overhead (Step 5)

Synchronization/scheduling overhead reduces the efficiency of a parallel execution. Synchronization overhead includes the latency for performing the synchronization operations (*synchronization costs*) and the idle time that processors spend waiting at the synchronization points (*synchronization idle time*). Scheduling overhead includes the latency for scheduling tasks onto processors (*scheduling costs*) and the idle time that processors spend waiting to begin their tasks (*scheduling idle time*). For loop-based applications, a large load imbalance often results in long synchronization idle time before synchronizing with the most heavily loaded processor. Inefficient scheduling causes unnecessary scheduling idle time due to unnecessary blocking of task acquisition.

Issue 16 - Reducing the Impact of Load Imbalance

Barrier synchronization is often used to parallelize loops, especially in shared-memory programs. When a program reaches a barrier synchronization point, it must wait there until all the processors reach that point. A long wait time may occur when one processor arrives much later than others due to *load imbalance*; the most heavily loaded processor dominates the execution time. Such overhead may not be necessary in many cases. A waiting processor could be allowed to cross the barrier if there are no *data dependencies* between the tasks of the processors that have not yet reached the barrier and the next waiting task of the processor.

Action 30 - Fuzzy Barriers for (I-16)

This overhead may be reduced with more efficient synchronization schemes and/or schedules. Some schemes, such as *fuzzy barriers*, have been proposed [80][81] for relaxing barrier synchronization by performing unrelated *useful work* on the processors which are waiting for other processors to arrive at a synchronization point. However, since the useful work must be independent of the work on any other processor prior to the fuzzy barrier, it may be difficult for the compiler or the user to find suitable tasks that can be efficiently inserted to relax the synchronization sufficiently.

Action 31 - Point-to-Point Synchronizations for (I-16)

Replacing barrier synchronization with *point-to-point synchronization* may reduce the overhead in certain cases when the data dependencies between processor tasks are known [58]. For example the pairwise point-to-point communication pattern shown in Figure 3-10 can also be used for pairwise synchronization of the processors in a producer-consumer fashion. Point-to-point synchronization requires extra programming effort because it is generally difficult for a compiler to extract the data dependencies in sufficient detail.

The communication dependency graph produced by *Domain decomposition* algorithms can be used to implement point-to-point synchronization. There is no need to synchronize two processors if there is no communication between them. We will refer to this scheme as *communication-dependency-graph-directed (CDG-directed) synchronization*. In case one processor needs to communicate with only a few others that own its neighbor subdomains, point-to-point synchronization may be more efficient than barrier synchronization.

We use Figure 3-12 to illustrate the CDG-directed synchronization. Figure 3-12(a) shows a two phase parallel execution with a different load distribution for each phase. Note that the two phases combined show perfect load balance, although neither phase is individually balanced. Figure 3-12(b) shows simple CDG-directed synchronization with 4 subdomains, whose dependency graph is shown in Figure 3-3(c). Processor 3 can begin the second phase of its computation as soon as processor 2 and processor 4 have both finished their first phase computation. However, point-to-point synchronization cannot effectively reduce the synchronization overhead in this example because processor 4 cannot begin its second phase until processor 1 completes its first phase, and this dependence serializes the execution of the two longest tasks.

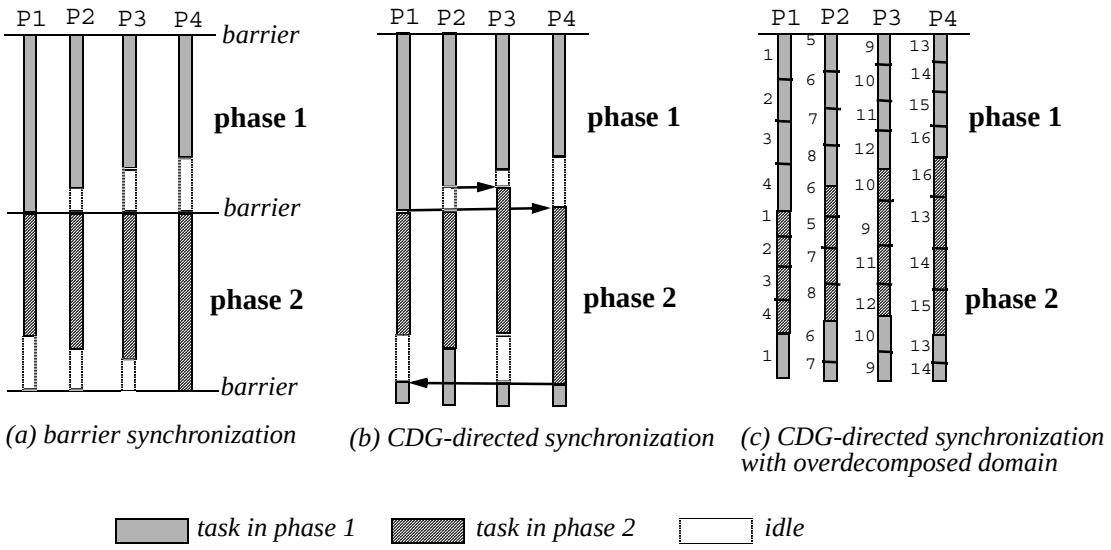


Figure 3-12: Barrier, CDG-directed Synchronization, and the Use of Overdecomposition

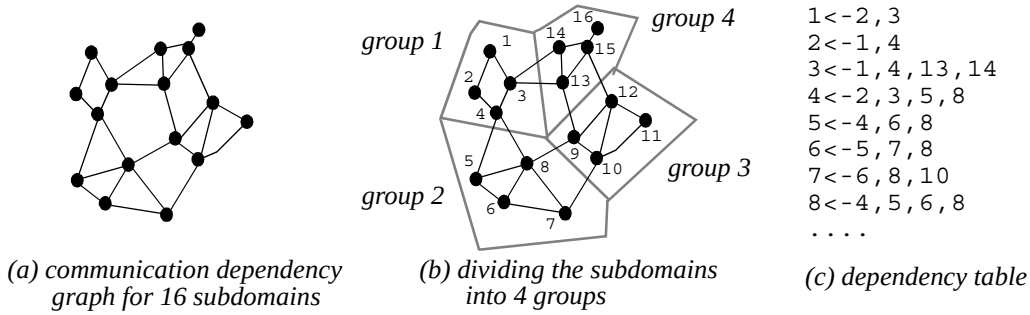


Figure 3-13: Overdecomposition and Dependency Table

Action 32 - Self-scheduling of Overdecomposed Subdomains for (I-16)

The communication dependence graph (CDG) can be also be used with overdecomposition to implement a variant of the self-scheduling schemes, hereafter referred to as *CDG-directed overdecomposed self-scheduling* (CDG-OSS), by imposing a set of scheduling constraints that are derived from the CDG. This scheme is described as follows:

- To eliminate or reduce the task (subdomain) migration caused by self-scheduling, all or most of the *overdecomposed subdomains* must be bound to particular processors. As shown in Figure 3-13, for solving the example problem of Figure 3-3 with *p=4* proces-

sors, the domain is overdecomposed into p groups of $k=4$ subdomains, and each group of k subdomains is assigned to a processor.

- As in a conventional self-scheduling scheme, each processor has an assigned pool of tasks waiting to be scheduled on the processors. Instead of using a barrier, CDG-directed self-scheduling performs point-to-point synchronization to decide if a task can be executed. If a task does not need communication from any unfinished task, it is ready to be scheduled on its processor; otherwise, it is not ready to be scheduled at that moment. The data dependencies between subdomains across the synchronization are listed in a dependency table (Figure 3-13(c)), e.g. subdomain 1 has depends on data from subdomain 2 and 3. Whenever a processor finishes one of its tasks, it finds a new ready task in its task pool and begins its execution. If no task remaining in its pool are ready, it waits.

- When a processor finishes all its tasks in the computation phase, CDG-OSS determines, by checking the dependency table, if the computation for any of its subdomains in the next phase can be scheduled. Because the processor has freedom to choose any unblocked task from among its k subdomains, its wait time may be reduced.

Figure 3-12(c) shows how overdecomposition can reduce the synchronization/scheduling overhead. When the first phase computation is finished on processor 4, an unblocked subdomain is scheduled to be executed on processor 4 for its second phase. At the moment, subdomains 13, 14, and 15 are blocked because they depend on unfinished phase 1 computations, so the unblocked subdomain, 16, is scheduled on processor 4. When subdomain 16 finishes its second phase, subdomain 13 is ready for execution. In this example, the CDG-OSS eliminates the overhead due to the load imbalance in these two phases.

A special form of overdecomposition assigns one subdomain to each processor and then divides the elements in each subdomain into two groups: *internal* and *boundary* [51]. Since the computation for the internal elements in one phase depends on other processors only for the boundary elements computed in the previous phase, the internal computations can be executed without communicating/synchronizing with any other processors. This overdecomposition strategy may be a good choice if it provides a number of internal elements in the subdomains that is sufficient to provide enough useful computation to significantly reduce the wait time for boundary element values.

Issue 17 - Reducing the Overall Scheduling/Synchronization Overhead

While various scheduling/synchronization schemes can be used to reduce the impact of load imbalance, the costs for these schemes should also be considered. For example, the cost of fine-grain self-scheduling may not be acceptable in message-passing applications. Point-to-point synchronization can cause more overhead than barrier synchronization if one processor needs to synchronize with too many other processors. Therefore, the *dependency graph*, the *load imbalance*, and the *cost of synchronization operations* should be considered for determining which scheduling/synchronization scheme to use.

3.7 Balancing the Combined Load for Multiple Phases (Step 6)

Issue 18 - Balancing the Load for a Multiphase Program

Different phases may have different sets of weights for the nodes and edges of the problem domain. Load balancing for multiple phases is more difficult than balancing one phase only.

Action 33 - Balancing the Most Critical Phase for (I-18)

One common solution is to balance the most *critical phase* with one domain decomposition and let the other phases simply use the resulting decomposition. This solution may not work well for programs with multiple critical phases.

Action 34 - Multiple Domain Decompositions for (I-18)

Another common solution for load balancing a multiple phase application is, to perform a separate domain decomposition for each phase of the program and *redistribute* data between phases. However, this redistribution can be costly since there may be little locality between consecutive phases.

Action 35 - Multiple-Weight Domain Decomposition Algorithms for (I-18)

Finding one partition that balances each of several phases has been viewed as a difficult problem. A heuristic algorithm, called a *dual-weight* algorithm [2] has been proposed to load balance two phases with one partition that approximately equally divides each of two sets of weights with a greedy bisection algorithm.

Action 36 - Fusing the Phases and Balancing the Total Load for (I-18)

As shown in Section 3.6, the use of *CDG-OSS* allows the system to tolerate some degree of temporal load imbalance in each phase. In Figure 3-12(c), the CDG-directed synchronization/overdecomposition scheme has *virtually fused* the two phases. Instead of balancing the load for each phase, balancing the *total load* for the effectively fused phases is much easier and may lead to a better overall solution.

Deciding which actions to choose for (I-18)

Action 33 (Balancing the Most Critical Phase) is the easiest solution, but is also least effective among these four actions. *Action 34 (Multiple Domain Decomposition)* should be used only if the data redistribution overhead due to subdomain migration is justified with respect to the improved load balance. *Action 35 (Multiple-Weight Domain Decomposition Algorithms)* can often effectively balance multiple phases with one domain decomposition, which eliminates the need for redistributing data. *Action 36 (Fusing the Phases and Balancing the Total Load)* can be used alone or in conjunction with the other approaches to help to tolerate the load imbalance in individual phases.

3.8 Balancing Dynamic Load (Step 7)

Issue 19 - Reducing the Dynamic Load Imbalance

Some applications have dynamic runtime behavior that changes the load distribution over time. An initially well-balanced data decomposition may become unbalanced some time later. For example, CRASH exhibits such behavior as the vehicle is progressively distorted. To balance the dynamic load, we may need to find a way to re-balance the load whenever it has changed significantly.

Action 37 - Dynamically Redecomposing the Domain for (I-19)

A common solution is to dynamically redecompose the domain and redistribute the data as the load changes. Some implementations perform a completely new decomposition (Action 37a), while others simply adjust the boundaries between subdomains (Action 37b). The performance can be improved if the overhead for domain re-decomposition and data

redistribution is smaller than the saved execution time. Some solutions require constantly monitoring the load on every processor, which may cause some modest extra overhead and require some modest hardware support.

Action 38 - Dynamic/Self-Scheduling for (I-19)

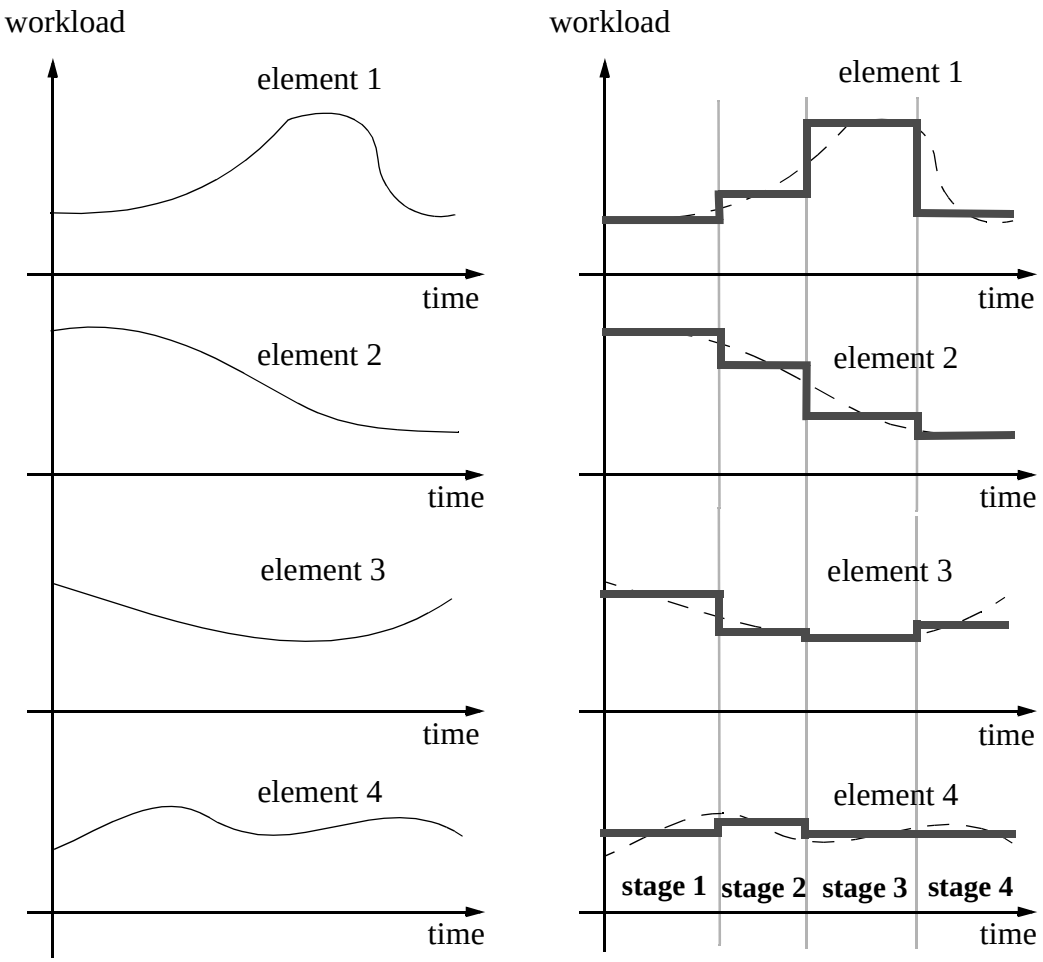
Dynamic or self-scheduling schemes may also be used to balance load dynamically without re-decomposing the domain. While these schemes save the cost for redecomposition, the data redistribution cost for task migration can be significant.

Action 39 - Multiple-Weight Domain Decomposition for (I-19)

Dynamic load distribution within some applications can be predicted from previous experience. Many engineering design applications are performed many times for slightly different data sets, where the load behavior may be predictable fairly well. If the load does not change frequently and dramatically, the load distribution function can be approximated with a few representative sets of weights, which represent the load distribution at certain stages. As a phase that is executed repeatedly over the course of run, if the behavior of that phase varies dynamically, it may be convenient to group the instances of that phase into stages where the instances within a stage have fairly similar behavior.

Figure 3-14 shows an example of such an approximation for a domain consisting of four elements: 1, 2, 3, and 4, where the workload associated with each element changes over time, as shown in Figure 3-14(a). Consequently, the workload distribution over the domain also changes over time. In Figure 3-14(b), we divide the execution into four stages, and approximate the workload distribution in each stage by choosing a representative workload value for each element.

This concept of multiple stages is thus similar to the concept of multiple phases. Thus, some of the techniques that balance the load for multiple phases may be able to be successfully applied to solve this multiple stage balancing problem. For example, the dual weight domain decomposition algorithm developed by Tomko [2] could be extended for finding a good partitioning that balances multiple stages.



(a) different dynamic workload behavior for 4 elements

(b) approximating the dynamic workload with 4 stages, yielding 4 sets of weights

Figure 3-14: Approximating the Dynamic Load in Stages.

Issue 20 - Tolerating the Impact of Dynamic Load Imbalance

As mentioned in (I-16), load imbalance impacts performance via synchronization/scheduling wait. As the load behavior changes over time, the load imbalance may change dynamically. In some cases, load imbalances that occur in different iterations may cancel each other, if the load imbalance can be tolerated temporally.

.

Predictable?	Frequently?	Significantly?	Primary/ Effective Actions	Secondary Actions
Yes	No	No	Action 37a Action 37b Action 39	Action 38 Action 40
Yes	No	Yes	Action 37a Action 39	Action 38 Action 40
Yes	Yes	No	Action 37b	Action 38 Action 40
Yes	Yes	Yes	N/A	Action 37a Action 38 Action 40
No	N/A	N/A	N/A	Action 38 Action 40

Table 3-1: Comparison of Dynamic Load Balancing Techniques

Action 40 - Relaxed Synchronizations for (I-20)

The relaxed synchronization schemes we mentioned earlier, *Action 30 (Fuzzy Barriers)*, *Action 31 (Point-to-Point Synchronizations)*, *Action 32 (Self-scheduling of Overdecomposed Subdomains)*, may also be used to tolerate highly unpredictable loads, but they may not work as well for this problem because these schemes cannot tolerate a large temporal load imbalance that may be accumulated in one stage after some number of iterations.

The load distribution function(s) in a dynamic application can change infrequently or frequently, gradually or significantly, and predictably or unpredictably. The user needs to understand the nature of the dynamic behavior of the application before applying the solutions. Possible actions for solving a specific class of dynamic applications are summarized in Table 3-1 and discussed below. Note that runtime performance monitoring is necessary if the way in which load distribution changes cannot be discovered at compile or load time.

If changes of the load distribution are predictable, dynamic domain decomposition may be applied periodically, depending on the frequency and the magnitude of the change. If the load distribution changes frequently but gradually, it may be possible to find some heuristics for adjusting the domain decomposition appropriately, e.g. by shifting subdomain boundaries, or domain redecomposition can be performed when the load imbalance exceeds a certain limit. However, if the load distribution changes frequently and significantly, then dynamic decom-

position may need to be applied whenever the load distribution changes, although this can cause unacceptable decomposition and data redistribution costs. If the load distribution is unpredictable (nondeterministic), then dynamic decomposition may not improve the load balance, and dynamic self-scheduling or simply suffering the imbalance may be the only ways of dealing with this situation.

In most cases, dynamic scheduling of fine-grain parallel tasks may help balance the load in a parallel region at the costs of scheduling and data redistribution. Relaxed synchronization can be used for tolerating a certain degree of temporal load imbalance, hoping that the imbalance can eventually be compensated for either naturally or with the help of dynamic load balancing techniques. Data redistribution costs can be modeled or measured to determine the benefit of applying the dynamic load balancing techniques. It is also possible that redistribution costs can be hidden by overlapping data redistribution with computation.

3.9 Conclusion

We have discussed various performance-tuning actions for solving common performance problems in irregular applications, including existing techniques and techniques derived from our work. More importantly, we provide a unified methodology to classify and utilize these performance-tuning techniques. Through the classification of performance issues, correlations among the tuning actions become more clarified, which is extremely important for optimizing the overall performance. The performance issues and their most appropriate tuning actions, as well as the effects of the actions discussed in this chapter are summarized at the end of this chapter in Table 3-2. In conjunction with the step-by-step performance tuning scheme described in Figure 3-2, the issues in Table 3-2 are addressed for resolution in our performance tuning methodology. While this is still preliminary work, we believe that, after more experience is gained, better resolution of these issues, supported by more performance tuning techniques, will find their way into future compilers and be applied in a manner such as we have articulated here.

Domain decomposition is critically involved in the parallelization of irregular applications, as well as many of the solutions for each of those performance issues. We provide a consistent way of using domain decomposition to solve individual problems, for example, parallelizing programs, organizing communications, blocking loops, balancing the load, and

implementing point-to-point synchronization. Some of the techniques, such as multithreading, prefetching, and point-to-point synchronization, can be further enhanced by overdecomposition. The CDG-directed overdecomposed self-scheduling scheme can tolerate some temporal load imbalance so that, instead of balancing the load for each phase, the total load can be balanced for multiple phase applications. Our approach provides a unified methodology for integrating various performance-tuning techniques with domain decomposition and aplying them selectively in a well-ordered sequence.

On a shared-memory machine, communications and the memory system are closely coupled, which provides both advantages and disadvantages for the programmers of these machines. While shared-memory machines are more powerful (and have more costly hardware) than message-passing machines, and although creating a functional code for a shared-memory machine is easier than orchestrating a functional message-passing code, tuning the code for efficiency on a distributed shared-memory machine is a far more sophisticated and subtle process than the corresponding tuning process for a message-passing code. The issues and techniques have been discussed in Section 3.3 and summarized in Section 3.3.5.

For improving the processor performance, our discussion in Section 3.4 focuses on issues that concern the parallel execution of irregular applications. As the speed gap between the memory and the processor continues to increase today, the data supply problem should receive more attention. For large irregular applications whose working set exceed the cache capacity, efficiently caching the data, e.g. (A-22)(A-23)(A-24), and hiding the memory latency, e.g.(A-25)(A-26), can be vital to the processor performance. The application of these techniques may need to be considered in conjunction with the communication pattern for improving the overall communication and computation performance.

Sections 3.5 to 3.8 collectively address the issue of improving the degree of task-level parallelism. Although the load balance, the schedule, and the synchronization in an application are often programmed and/or perceived differently by the programmer, they are all involved in this complex issue. Below we summarize the way we approach this problem:

- *Reducing synchronization/scheduling overhead for single phase applications*

  For applications whose performance is dominated by a single program phase, static scheduling and static load balancing, e.g. (A-1)(A-28)(A-33), are usually used when the load distribution does not vary much during the runtime. The quality of domain decomposition determines the load imbalance in the parallel region. In this case, relaxed synchronization e.g. (A-30)(A-31), may not be useful since the performance is dominated by the heaviest load, unless dynamic scheduling or dynamic decomposition is used to change the load on processors. However, self-scheduling (A-29), multiple domain decomposition (A-34), and dynamic decomposition (A-37) incur extra scheduling/decomposition costs and data redistribution costs, which offset the performance gained from achieving a balanced load, or even degrades the overall performance.

- *Reducing synchronization/scheduling overhead for multiple phase applications*

  For applications with multiple program phases and static load distribution, static load balancing techniques with a single domain decomposition (A-33)(A-35) may or may not be able to balance the load for multiple phases simultaneously. As in the case for single phase applications, dynamic scheduling/decomposition (A-29)(A-34) requires extra costs for scheduling/decomposition and data redistribution. Fuzzy barriers (A-30) may be effective, but will not be able to relax the barrier unless a sufficient amount of independent computation can be found.

  We use point-to-point synchronization (A-31) as a means of virtually fusing regions (A-36), if the dependence information between the phases can be determined. However, point-to-point synchronization alone may not improve the performance, as we have shown in Figure 3-12(b). In such cases, we employ dynamic scheduling to schedule the overdecomposed subdomains assigned to each processor. In (A-32), we have discussed how combining overdecomposition, localized self-scheduling, and point-to-point synchronization can reduce the synchronization/scheduling overhead more effectively than point-to-point synchronization alone.

- *Reducing synchronization/scheduling overhead for dynamic applications*

  Tuning dynamic applications is highly case-dependent and requires extensive information about the application performance. For predictable load changes, optimizing the decomposition, e.g. (A-37) or (A-39), may effectively balance the load, in conjunction with the use of

self-scheduling (A-38) and/or relaxed synchronization (A-40). For highly dynamic or unpredictable load changes that are difficult to track, (A-38) and (A-40) may be the only effective ways to reduce the overhead.

There are several issues that we have not fully addressed regarding this combined scheme. First, we have not characterized the specific strategies for domain decomposition and overdecomposition. For example, dynamic scheduling and point-to-point synchronization are more efficient if each subdomain depends on only a few other subdomains. Second, we have not measured the performance gain versus the synchronization/scheduling costs. The synchronization/scheduling costs can offset the performance gain if the domain is excessively overdecomposed and/or the degree of dependence is high for the average subdomain. Third, we have not characterized how large a temporal load imbalance this scheme can actually tolerate. The tolerance of temporal load imbalance can be critical to the strategies of domain decomposition and overdecomposition for balancing multi-phase applications. These are issues that need to be addressed in the future.

Tuning Step	Associated Performance Issues	Tuning Action	Positive for Solving Issues	Negative for Solving Issues	Other Related Issues
Partitioning the Problem (Step 1)	(I-1) Partitioning an Irregular Domain	(A-1) Applying a Proper Domain Decomposition Algorithm for (I-1)	(1)	-	(3)(6) (15)(18) (19)
Tuning the Communication Performance (Step 2)	(I-2) Exploiting Processor Locality	(A-2) Proper Utilization of Distributed Caches for (I-2)	(2)	-	(4)(5)(12) (14)
		(A-3) Minimizing Subdomain Migration for (I-2)	(2)	-	(1)(4)(12) (15)(18)(19)
	(I-3) Minimizing Interprocessor Data Dependence	(A-4) Minimizing the Weight of Cut Edges in Domain Decomposition for (I-3)	(3)	-	(1)(6)(15) (18)(19)
	(I-4) Reducing Superfluity	(A-5) Array Grouping for (I-4)	(4)	-	(5)(12)
	(I-5) Reducing Unnecessary Coherence Operations	(A-6) Privatizing Local Data Accesses for (I-5)	(4)(5)	(12)	(2)(14)
		(A-7) Optimizing the Cache Coherence Protocol for (I-5)	(5)	-	(2)(4)(12)
		(A-8) Cache-Control Directives for (I-5)	(5)	-	(2)(4)(12)
		(A-9) Relaxed Consistency Memory Models for (I-5)	(5)	-	(2)(4)(12)
		(A-10) Message-Passing Directives for (I-5)	(5)	-	(4)(7)(9)
	(I-6) Reducing the Communication Distance	(A-11) Hierarchical Partitioning for (I-6)	(6)	-	(1)(2)(3) (15)(18) (19)
		(A-12) Optimizing the Subdomain-Processor Mapping for (I-6)	(6)	-	(15)(16)(18) (19)(20)
	(I-7) Hiding the Communication Latency	(A-13) Prefetch, Update, and Out-of-order Execution for (I-7)	(7)	-	(12)(14)
		(A-14) Asynchronous Communication via Messages for (I-7)	(7)	-	(4)(5)(9)(14)
		(A-15) Multithreading for (I-7)	(7)(13)	-	(9)(12)(14)
	(I-8) Reducing the Number of Communication Transactions	(A-16) Grouping Messages for (I-8)	(8)	(9)	(14)
		(A-17) Using Long-Block Memory Access for (I-8)	(8)	(4)(5)(9)	(14)
	(I-9) Distributing the Communications in Space	(A-18) Selective Communication for (I-9)	(8)	-	(14)
	(I-10) Distributing the Communications in Time	(A-19) Overdecomposition to Scramble the Execution for (I-10)	(9)	-	(14)

Table 3-2: Performance Tuning Steps, Issues, Actions and the Effects of Actions. (1 of 2)

Tuning Step	Associated Performance Issues	Tuning Action	Positive for Solving Issues	Negative for Solving Issues	Other Related Issues
Optimizing Processor Performance (Step 3)	(I-11) Choosing a Compiler or Compiler Directive	(A-20) Properly Using Compilers or Compiler Directives for (I-11)	(11)	-	-
		(A-21) Goal-Directed Tuning for Processor Performance for (I-11)	(11)	-	-
	(I-12) Reducing the Cache Capacity Misses	(A-22) Cache Control Directives for (I-12)	(12)	(2)	(4)(14)
		(A-23) Enhancing Spatial Locality by Array Grouping for (I-12)	(12)	(2)	(4)(14)
		(A-24) Blocking Loops Using Over-decomposition for (I-12)	(12)	-	(14)
	(I-13) Reducing the Impact of Cache Misses	(A-25) Hiding Cache Miss Latency with Prefetch and Out-of-Order Execution for (I-13)	(13)	-	(7)
		(A-26) Hiding Memory Access Latency with Multithreading for (I-13)	(13)	-	(7)
	(I-14) Reducing Conflicts of Interest between Improving Processor Performance and Communication Performance	(A-27) Repeating Steps 2 and 3 for (I-14)	(14)	-	(2)-(13)
Balancing the Load for Single Phases (Step 4)	(I-15) Balancing a Nonuniformly Distributed Load	(A-28) Profile-Driven Domain Decomposition for (I-15)	(15)	-	(1)(3) (18)(19)
		(A-29) Self-Scheduling for (I-15)	(15)(16)(18) (19)(20)	(2)(17)	-
Reducing the Synchronization/ Scheduling Overhead (Step 5)	(I-16) Reducing the Impact of Load Imbalance	(A-30) Fuzzy Barriers for (I-16)	(16)(18) (19)(20)	-	(17)
		(A-31) Point-to-Point Synchronizations for (I-16)	(16)(18) (19)(20)	-	(17)
		(A-32) Self-scheduling of Overdecomposed Subdomains for (I-16)	(2)(16)	-	(16)(17) (18)(19)
	(I-17) Reducing the Overall Scheduling/ Synchronization Overhead	-	-	-	-
Balancing the Combined Load for Multiple Phases (Step 6)	(I-18) Balancing the Load for a Multiphase Program	(A-33) Balancing the Most Critical Phase for (I-18)		(18)	(3)(15)
		(A-34) Multiple Domain Decompositions for (I-18)	(18)	-	(3)
		(A-35) Multiple-Weight Domain Decomposition Algorithms for (I-18)	(2)(18)	-	(3)(15)(19)
		(A-36) Fusing the Phases and Balancing the Total Load for (I-18)	(16)(18)	-	(15)(20)
Balancing Dynamic Load (Step 7)	(I-19) Reducing the Dynamic Load Imbalance	(A-37) Dynamically Redecomposing the Domain for (I-19)	(19)(20)	(2)	(3)(14)
		(A-38) Dynamic/Self-Scheduling for (I-19)	(16)(19) (20)	(2)	(3)(15) (17)(18)
		(A-39) Multiple-Weight Domain Decomposition for (I-19)	(2)(19)	-	(3)(15)(19)
	(I-20) Tolerating the Impact of Dynamic Load Imbalance	(A-40) Relaxed Synchronizations for (I-20)	(16)(19)	-	(14)(17)(18)

Table 3-2: Performance Tuning Steps, Issues, Actions and the Effects of Actions. (2 of 2)

CHAPTER 4. HIERARCHICAL PERFORMANCE BOUNDS AND GOAL-DIRECTED PERFORMANCE TUNING

Hierarchical machine-application bounds models [39][40][42][44][45][46][47][49], collectively called the *MACS* bounds hierarchy, have been used to characterize application performance by exposing performance gaps between the different levels of the hierarchy. The MACS bounds hierarchy successively includes performance constraints of *Machine* peak performance, an *Application's* essential computation workload, the additional workload in the *Compiler* generated code, and instruction *Scheduling* constraints caused by data, control, and structural hazards. Modeling methodologies and specific models have been developed and presented for evaluating processor performance a variety of systems. The MACS bounds hierarchy has been extended to characterize application performance on the KSR1 shared-memory parallel computer. The extended hierarchy, called *MACS12*B* [48], addresses *cache misses* in the shared-memory system and the runtime overhead due to *load imbalance*.

Several important performance issues remained unaddressed in the previous work, namely, *degree of parallelization*, *multiple program regions with different workload distributions*, *dynamic load imbalance*, and *I/O and operating system interference*. For *irregular* applications, *I/O-intensive* applications, or *interactive* applications, these unaddressed issues can greatly affect the performance. By adapting the existing hierarchies and incorporating new bounds as described in this chapter, the performance bounds methodology now has a more complete hierarchy for characterizing a broader range of applications on parallel machines.

With the new bounds hierarchy and our new automatic bounds generation tool, *CXbound*, complicated application performance profiles on the HP/Convex Exemplar can be converted into a simple set of performance bounds, which provide more effective high level performance visualization and insights into program behavior.

In this chapter, we explain the hierarchical machine-application-performance bounds models we have developed for characterizing the performance gaps between ideal and delivered performance. The previously developed bounds models are introduced in Section 4.1. Our recent extension of the bounds models and our goal-directed performance tuning scheme for parallel environments are described in Section 4.2. In Section 4.3, we discuss the acquisition of the performance bounds within *CXbound*. In Section 4.4, we use *CXbound* in several case studies and demonstrate the effectiveness of hierarchical bounds analysis. Finally, in Section 4.5, we review the development of the hierarchical bounds methodology and discuss possible future research topics in this area.

4.1 Introduction

A *performance bound* is an upper bound on the best achievable performance. In previous performance bounds work, *performance* has been measured by *Cycles Per Instruction* (CPI), *Cycles Per Floating-point operation* (CPF) or *Cycles Per Loop iteration* (CPL). In this dissertation, we extend the scope of performance bounds to assess the performance of entire applications. Since the goal of performance tuning is to reduce application runtime, we believe that performance is best measured by the *total runtime*. CPI, CPF, or CPL may easily be derived from runtime metrics of applications or regions of applications and provide meaningful comparisons when the number of instructions, number of floating-point operations, or the number of iterations of the target application remains constant during performance tuning. Note that an upper bound on performance is a lower bound on the runtime, CPI, CPF, or CPL.

4.1.1 The MACS Bounds Hierarchy

The MACS machine-application performance bound methodology provides a series of upper bounds on the best achievable performance (equivalently, lower bounds on the runtime) and has been used for a variety of loop-dominated applications on vector, superscalar and other architectures [39][40][42][44][45][46][47][49]. The hierarchy of *bounds equations* is based on the peak performance of a *M*achine of interest (M), the *M*achine and a high level *A*pplication code of interest (MA), the *C*omputer-generated workload (MAC), and the actual compiler-generated *S*chedule for this workload (MACS), respectively. MACS Bounds equations for IBM RS/6000 [46], Astronautics ZS-1 [46], Convex C-240 [47], KSR1 [42][44], IBM SP2 [15], and other systems, have been developed.

Use of the hierarchical bounds analysis for performance tuning on scientific code has been presented in [44][45][48][49]. Tools that automate the acquisition of the performance bounds were developed for the KSR1 [44][116]. The MACS bounds hierarchy is generally described below in Sections 4.1.1.1 through 4.1.1.4.

4.1.1.1 Machine Peak Performance: M Bound

The Machine (M) bound is defined as the minimum run time if the application workload were executed at the peak rate. The minimum workload required by the application is indicated by the total number of operations observed from the high-level source code of the application. The machine peak performance is specified by the maximum number of operations that can be executed by the machine per second. The M bound (in seconds) can be computed by

M Bound = (Total Number of Operations in Source Code)/(Machine Peak Performance). (EQ 10)

4.1.1.2 Application Workload: MA Bound

The MA bound considers the fact that an application usually has various types of operations that have different execution times and use different processor resources (functional units). Functional units are selected for evaluation if they are deemed likely to be a performance bottleneck in some common situations. The MA bound of an application counts the operations for each selected function unit from the high level code of the application, the utilization of each functional unit is calculated, and the MA bound is determined by the execution time of the most heavily utilized functional unit. The MA bound thus assumes that no data or control dependencies exist in the code and that any operation can be scheduled at any time during the execution, so that the function unit(s) with heaviest workload is fully utilized.

4.1.1.3 Compilation: MAC Bound

The MAC bound is similar to MA, except that it is computed using the actual operations produced by the compiler, rather than only the operations counted from the high level code. Thus MAC still assumes an ideal schedule, but does account for redundant and unnecessary operations inserted by the compiler as well as those that might be necessary to add to the MA

operations counts in order to orchestrate a particular machine code. MAC thus adds one more constraint to the model by using an actual rather than an idealized workload.

4.1.1.4 Instruction Scheduling: MACS Bound

The MACS bound, in addition to using the actual workload, adds another constraint by using the actual schedule rather than an ideal schedule. The data and control dependencies limit the number of valid instruction schedules and may result in pipeline stalls (bubbles) in the functional units. A valid instruction schedule can require more time to execute than the idealized schedules we assumed in the M, MA, and MAC bounds.

4.1.2 The MACS12*B Bounds Hierarchy

In his Ph.D dissertation [48], Eric L. Boyd extended the MACS bounds hierarchy to characterize application performance in parallel environments, by using the KSR1 parallel computer as a case study. Boyd's MACS12*B bounds hierarchy subdivides the gap between the actual runtime and the MACS bound with intermediate bounds that model *data subcache misses* (MACS1), *local cache misses* (i.e. interprocessor communication) (MACS12), *inserted instructions*[1] (MACS12*), and the *load imbalance* (MACS12*B).

The calculation of MACS1, MACS12, and MACS12* relies heavily on data gathered from KSR's performance analyzer, PMON. The number of data subcache misses, local cache misses, and CEU_STALL time reported by PMON are used to compute the MACS bound and to obtain the MACS1, MACS12, and MACS12* bounds. Instead of generating a different bounds hierarchy for each processor, the bounds hierarchy from M through MACS12* is calculated for the "average" processor, assuming that the workload is perfectly balanced. The MACS12*B bound adds the load imbalance effect, by using the processor with the largest runtime to bound the performance of an application.

The MACS12*B bounds hierarchy has proved to be effective for performance characterization of certain scientific applications, e.g. some of the NAS Parallel Benchmarks [62].

1. In addition to the constraints that are modeled in the MACS12 bound, there are a variety of reasons that may cause extra instructions to be inserted into the instruction stream for such purposes as timer interrupts and I/O services. Refer to [48] for more details.

4.1.3 Performance Gaps and Goal-Directed Tuning

In ascending through the bounds hierarchy from the M bound, the model becomes increasingly constrained as it moves in several steps from potentially deliverable toward actually delivered performance. Each *gap* between successive performance bounds exposes and quantifies the performance impact of specific runtime constraints, and collectively these gaps identify the bottlenecks in application performance. Performance tuning actions with the potential greatest performance gains can be selected according to which gaps are the largest, and their underlying causes. This approach is referred to as *goal-directed performance tuning* or *goal-directed compilation* [45][48], which can be used to assist hand-tuning, or implemented within a goal-directed compiler for general use.

4.2 Goal-Directed Tuning for Parallel Applications

4.2.1 A New Performance Bounds Hierarchy

Several important performance issues remained unaddressed in the previous work, namely, *degree of parallelization*, *multiple program regions with different workload distributions*, *dynamic load imbalance*, and *I/O and operating system interference*. For *irregular* applications, *I/O-intensive* applications, or *interactive* applications, these unaddressed issues can greatly affect the performance. By adapting the existing hierarchies and incorporating new bounds as described in this chapter, the performance bounds methodology now has a more complete hierarchy for characterizing a broader range of applications on parallel machines.

Our new performance bounds hierarchy, as shown in Figure 4-1, successively includes major constraints that often limit the delivered performance of parallel applications. These constraints are considered in the order of: *machine peak performance* (*M* bound), *mismatched application workload* (*MA* bound), *compiler-inserted operations* (*MAC* bound), *compiler-generated instruction schedule* (*MACS* bound), *finite cache effect* (*MACS$* or *I* bound), *partial application parallelization* (*IP* bound), *communication overhead* (*IPC* bound), *I/O and operating system interference* (*IPCO* bound), *overall load imbalance* (*IPCOL* bound), *multiple phase load imbalance* (*IPCOLM* bound), *dynamic load imbalance* (*IPCOLMD* bound). We have found this

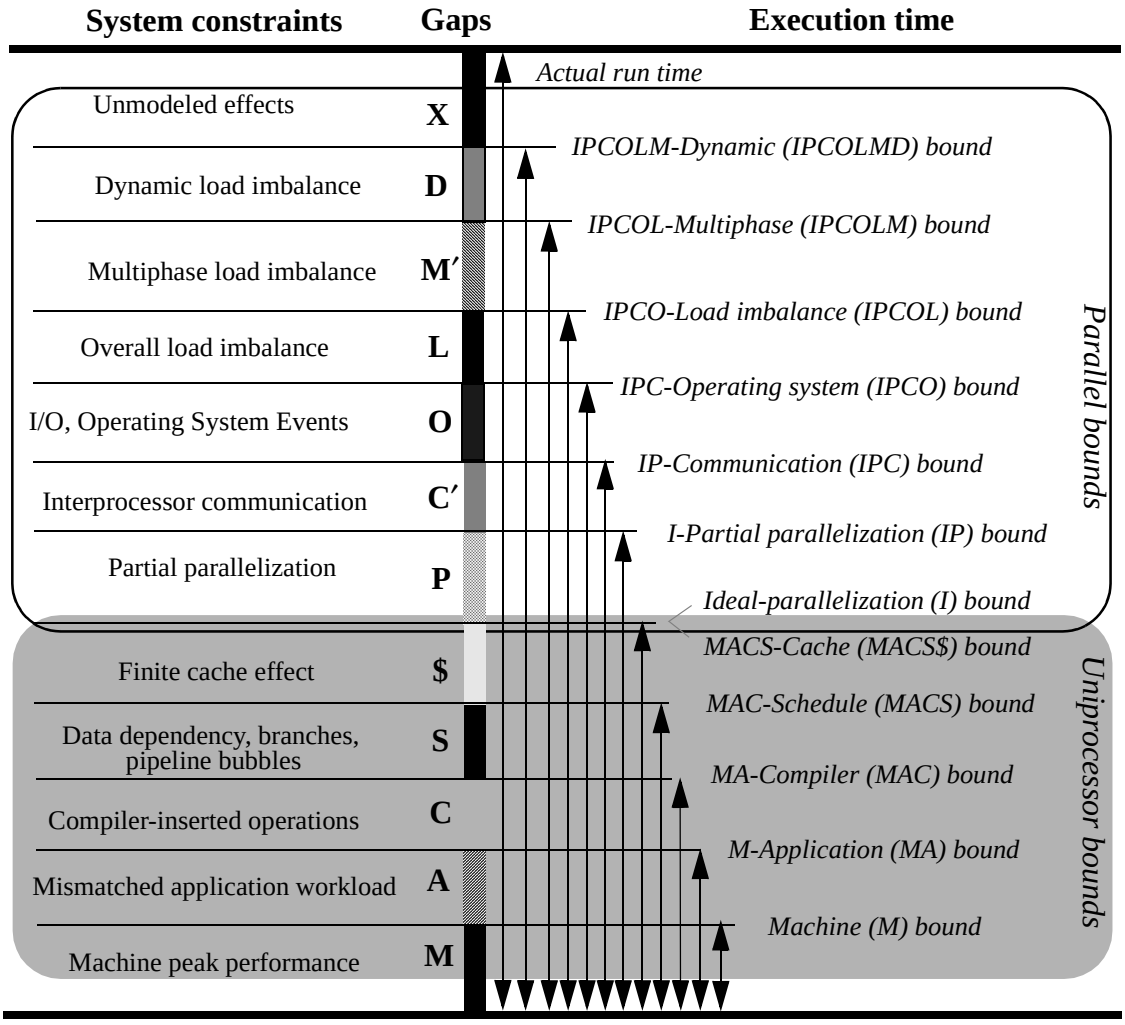


Figure 4-1: Performance Constraints and the Performance Bounds Hierarchy.

ordering to be intuitive and useful in aiding the performance tuning effort; however, we do not claim that it is unique or optimal. Other variations or refinements could be considered.

The *uniprocessor bounds*, consisting of M, MA, MAC, MACS, and MACS$, are carried over from previous work [44][45][48] and are used to explain the performance within each processor (uniprocessor performance). The MACS$ bound is similar to the MACS1 bound in the MACS12*B hierarchy (see Section 4.1.2), but there is a subtle difference between them. Unlike the MACS1 bound, coherence misses are regarded as a form of interprocessor communication and are distinguished from the other types of cache misses (compulsory, capacity,

mapping, and replacement) which are included in the MACS$ bound for characterizing the finite cache effect.

Except for the IPCO bound, the *parallel bounds* address issues beyond the context of a uniprocessor. The *degree of parallelization* can limit the scalability of application performance on a parallel machine. *Interprocessor communication*, *I/O and OS events* add more processor workload. Although I/O and operating system events also occur in serial applications, they are addressed in the IPCO bound, because it is easier for our CXbound tool to characterize their impact with this ordering. The performance impact of an *unbalanced load* is gauged not only by *overall*[1] load imbalance in the application, but also by the load imbalance within individual parallel regions, which is extremely important for characterizing applications with multiple phases (separated by barrier synchronization) with different non-uniformly distributed loads. Finally, the effectiveness of static performance optimization can be compromised by *dynamic* application/machine behavior which varies over the course of a run and cannot be predicted at compile time.

For convenience, the uniprocessor bounds and parallel bounds follow two different naming conventions. For consistency, the names of most uniprocessor bounds, i.e. through the MACS bound, remain unchanged from previous work. Due to the change in the way that the finite cache effect is calculated, the new symbol $ and new name MACS$ are used for this level, instead of the MACS1 bound used in [48]. For the parallel bounds, we rename the MACS$ bound as the I (*I*deal parallelization) bound, followed by the IP (*P*artial parallelization), IPC (*C*ommunication), IPCO (I/O and OS events), IPCOL (overall *L*oad imbalance), IPCOLM (*M*ultiphase load imbalance), and IPCOLMD (*D*ynamic load imbalance) bounds.

The gap between two successive bounds is named after the performance constraint(s) that differentiates the two bounds. However, while we tried to assign a different letter to each new gap, the letters *C* and *M* are each repeated twice in the entire bounds hierarchy. To avoid confusion, we shall refer to the *C*ommunication gap and *M*ultiphase gap as *C′* gap and *M′* gap, respectively, to distinguish them from the *C*ompiler inserted instructions gap and the *M*achine peak performance.

1. Overall load imbalance refers to the imbalance of the distribution of the total load assigned to each processor over the entire application.

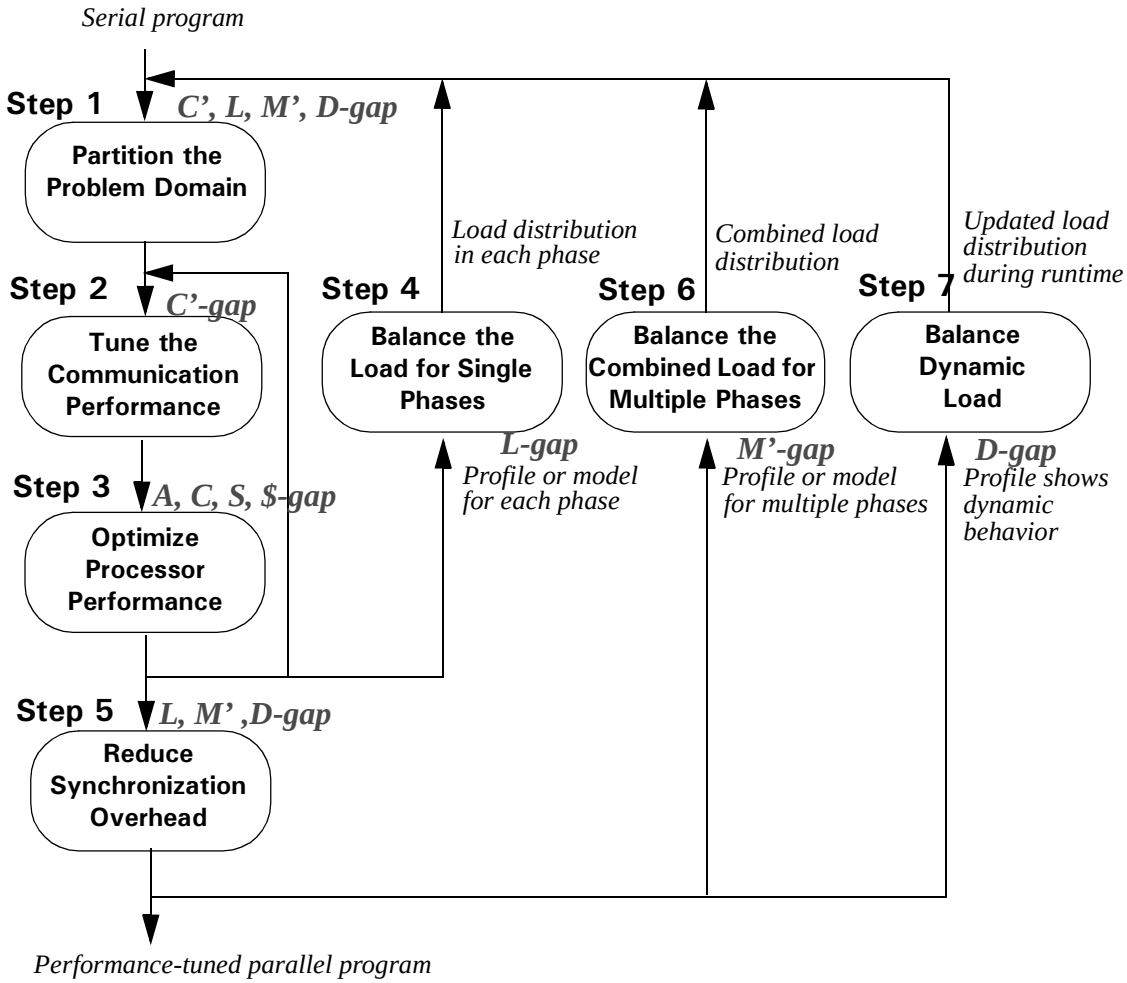


Figure 4-2: Performance Tuning Steps and Performance Gaps.

4.2.2 Goal-Directed Performance Tuning

The new bounds hierarchy matches the performance tuning methodology that we discussed in Chapter 3 and aids it in implementing a goal-directed performance tuning strategy. Figure 4-2 shows the relationship between the performance tuning steps and the performance gaps. Before each step, we consider specific gap(s). For example, the actions in Step 1 (partitioning) are associated with gaps C', L, M', and D. Significant gaps help guide what specific performance tuning actions should be considered for each step. A step may be skipped if there is no significant gap associated with that step. After one or more performance tuning actions are applied, the bounds hierarchy can be re-calculated to evaluate the effectiveness and the side-effects of these actions.

Table 4-1 starting on page 145 lists the performance issues and tuning actions, together with the performance gaps that each tuning action attempt to reduce (*targeted gaps*), and the primary performance gaps that each tuning action may affect (*primarily affected gaps*). This table is used in conjunction with Figure 4-2 for selecting appropriate tuning actions within the goal-directed performance tuning process.

4.2.3 Practical Concerns in Bounds Generation

4.2.3.1 Symmetrical Applications

Originally, performance bounds analysis was developed for characterizing performance of loop-based codes. Performance bounds analysis has been capable of explaining performance gaps for individual loops from the Livermore Fortran Kernels (LFK) [45][117] and the NAS Parallel Benchmark suite (NPB) [48]. For a perfectly parallelized Single-Program-Multiple-Data (SPMD) application, the workload should be symmetrically partitioned among the processors. Since the constraint of load imbalance is not considered until the Load Balance (IPCOL) bound, the workload is assumed to be perfectly balanced, and thus the performance bounds from M to ICPO are identical for each processor in parallel regions. Therefore, only one set of performance bounds is calculated, by averaging the bounds among the processors. For example, the M, MA, MAC, and MACS bounds for a parallel application are often approximated by dividing the bounds acquired on single-processor runs by the number of processors. If the parallel code presents a substantially different computational workload than a uniprocessor version, then these bounds should be calculated by evaluating the parallel version of the code on one processor.

4.2.3.2 Automatic Bounds Generation

Automatic performance bounds generation is essential for users to apply bounds analysis on large scale applications. As shown in [116], with automatic source-code analysis tools, complicated codes on the KSR1 can be analyzed with the MACS bounds hierarchy. We believe that an automatic tool should be able to generate the performance bounds described in Section 4.2.1, by integrating the use of source-code analysis, profiling and tracing. In pursuit of automatic bounds generation, we have successfully converted *CXpa* profiles into parallel bounds with our tool, *CXbound*, discussed in Section 4.3. In Section 4.4, we demonstrate the

use of *CXbound* to generate parallel performance bounds automatically for characterizing a large application.

4.2.3.3 Accuracy and Cost of Bounds Generation

For effective performance characterization, accurate performance bounds are necessary. The impact of a performance constraint may not be reflected accurately by a performance gap if the performance bounds are not accurate. A gap can be overestimated (or underestimated), if the bound at the bottom (or top) of the gap is over-optimistic. A good bounding mechanism should generate performance bounds that approximate their *tightest* lower bounds.

It can be very time-consuming to acquire the I (MACS$) bound precisely, since cache simulation may be necessary (see Section 4.3.1). Instead of acquiring the I bound, CXbound reports the I_a bound (also see Section 4.3.1) as an approximation for the I bound since the I_a bound is much easier to obtain. CXbound does check, however, the validity of this approximation, and notifies the user when this approximation is not accurate.

4.3 Generating the Parallel Bounds Hierarchy: CXbound

CXbound is a tool that converts profiles acquired by *CXpa* (see Section 2.3.2) into performance bounds and gaps. In Sections 4.3.1 through 4.3.7, we discuss the mechanisms that *CXbound* uses to acquire the I, IP, IPC, IPCO, IPCOL, and IPCOLM bounds of an application running on a *N*-processor HP/Convex Exemplar. The limitations and possible future development of *CXbound* are discussed in Section 4.5.

4.3.1 Acquiring the I (MACS$) Bound

The I (MACS$) bound measures the minimum run time required to execute the application under an ideal (zero communication overhead and perfectly load balanced) parallel environment with no I/O or OS interference. As mentioned in Section 4.2.3.1, the I bound for a SPMD application is the average MACS$ bound among the processors. Thus, given the number of processors involved in the execution, *N*, and the MACS$ bounds on the runtime for individual processors, Ω_1, Ω_2,..., Ω_N, the averaged I bound is calculated as:

$$I\ Bound = \left(\Sigma_{i=1..N}\,\Omega_i\right)/N. \qquad \text{(EQ 11)}$$

The I bound excludes the coherence misses in a shared-memory application, but it is relatively difficult to create a zero-communication-overhead environment for acquiring the I bounds. Thus, CXbound uses the *CXpa* profiles acquired from single processor runs to approximate the I bounds:

$$I_a \text{ Bound} = (\text{Total CPU Time from the One-Processor Profile})/N. \qquad \text{(EQ 12)}$$

Since one-processor runs do not generate coherence misses, the CPU time (as defined by CXpa, the time that the CPU spent in computation and memory accesses) in the one-processor CXpa profiles should measure the *inherent* computation and memory access time in the application. The I_a bound generated by the above equation essentially assumes *perfect parallelization* and *linear speedup*. For the I_a bound to be close to the I bound, the total computation time should be independent of the number of processors.

Due to the higher collective cache capacity of an *N*-processor execution, in a general it will generate fewer cache capacity, conflict, and replacement misses than the single-processor execution. As a result, the I_a bound may be larger than the I bound. For example, if the working set for a one-processor run far exceeds the size of a processor cache, I_a will be heavily influenced by the cache thrashing that results. As processors are added, and if eventually the working set for a particular processor fits well within its cache (as it should), the gains from the reductions in capacity, conflict, and replacement misses may far exceed the increasing penalties of compulsory and coherence misses, and other parallel overhead factors which tend to grow as the number of processors increases. This situation is commonly referred to as *superlinear speedup*. To detect such cases, the cache miss information reported by CXpa is used. CXbound will notify the user if the number of cache misses reported in the one-processor profile is significantly more than the misses reported in the *N*-processor profile; this notification should be taken as a warning that the I_a bound may be too pessimistic (relative to the I bound).

4.3.2 Acquiring the IP Bound

The degree of parallelization in the application is a factor that can limit the parallelism in a parallel execution. The application may contain sequential regions that are executed sequentially by one processor. Let the total computation time in the sequential regions be Ω_s and total computation time in the parallel regions be Ω_p, the IP bound for an *N*-processor exe-

cution is defined as the minimum time required to execute the application under the assumption that the parallel regions are executed under an ideal parallel environment, i.e.

$$IP\ Bound = \Omega_s + \Omega_p / N, \quad \text{(EQ 13)}$$

which is also known as *Amdahl's Law*.

As in the I bound, the computation time is approximated by CXbound using one-processor CXpa profiles. The sequential CPU time and parallel CPU time can be separated by CXpa even on a one-processor configuration.

The gap between the I bound and the IP bound, i.e.

$$P\ Gap = (\Omega_s + \Omega_p / N) - (\Omega_s + \Omega_p) / N = \Omega_s (N-1) / N, \quad \text{(EQ 14)}$$

reflects the impact of imperfect parallelization of the application, since the P gap is proportional to the sequential workload.

The degree of parallelization, which may be a more conventional metric for parallel programmers, can be computed as:

$$Degree\ of\ Parallelization = \Omega_p / (\Omega_s + \Omega_p) = 1 - (P\ Gap) / ((N-1) * (I\ Bound^1)). \quad \text{(EQ 15)}$$

4.3.3 Acquiring the IPC Bound

The IPC bound is defined by the minimum time required to execute the application workload with actual communications on actual target processors, under the assumption that the workload in the parallel portions is always perfectly balanced. Note that communications may add extra workload to both the sequential portions and the parallel portions of the application. Amdahl's Law is reapplied to the increased sequential and parallel workload to acquire the IPC bound for a *N*-processor execution, i.e.

$$IPC\ Bound = \Omega_s' + \Omega_p' / N, \quad \text{(EQ 16)}$$

where Ω_s' and Ω_p' denote the sequential and parallel workload assumed in the IPC bound.

1. Note that the I bound may be written in terms of Ω_s and Ω_p as $(\Omega_s + \Omega_p) / N$.

CXbound uses the CPU time in *N*-processor profiles to measure the run time that processors spend on computation and communication. The parallel CPU time (Ω_p') that the processors spend in the parallel regions, is calculated by

$$\Omega_p' = \sum_q \sum_r c_{r,q} \ , \qquad \text{(EQ 17)}$$

where $c_{r,q}$ is the total CPU time that processor q spends in parallel region r. The sequential CPU time (Ω_s') is summed over the serial regions.

4.3.4 Acquiring the IPCO Bound

In many high-performance applications, input and output for a program occur mostly in the form of accessing mass storage and other peripheral devices (e.g. terminal, network, printer,...etc.). I/O events are mostly handled by the operating system (OS) on modern machines. The OS also handles many other operations, such as virtual memory management and multitasking, in the background. These background OS activities may or may not be originated by the target application, but can greatly affect the performance of the target application.

To acquire the IPCO bound for an *N*-processor execution, CXbound first calculates the sequential execution time (Ω_s'') and parallel execution time (Ω_p'') under the environment that the IPCO bound models:

$$\Omega_s'' = \sum_{q=1..N} \sum_{r \in S} w_{r,q} \ , \qquad \text{(EQ 18)}$$

$$\Omega_p'' = \sum_{q=1..N} \sum_{r \in P} w_{r,q} \ , \qquad \text{(EQ 19)}$$

where $w_{r,q}$ is the *wall clock time* that processor q spent in region r, S is the set of sequential regions, and P is the set of parallel regions. As we explained in Section 2.3.2, the wall clock time reported by CXpa additionally includes the time spent in OS routines, which is not included in the reported CPU time. Then, Amdahl's Law is reapplied to the increased sequential and parallel execution times under the environment that the IPCO bound models, i.e.

$$IPCO\ Bound = \Omega_s'' + \Omega_p'' \ / N. \qquad \text{(EQ 20)}$$

4.3.5 Acquiring the IPCOL Bound

Load imbalance affects the degree of parallelism in the parallel execution. The execution time of an application with load imbalance is bounded by the time required to execute on the most heavily loaded processor. The IPCOL bound is defined as the minimum time required to execute the largest load assigned to one processor, under the assumption that the load from different parallel regions and iterations that is assigned to a particular processor can simply be combined.

The total wall clock time that processor q spent in parallel regions is calculated by summing processor q's wall clock time over the parallel regions, i.e.

$$\Omega_{p,q}'' = \sum_{r \in P} w_{r,q} . \qquad \text{(EQ 21)}$$

The IPCOL bound for the parallel regions is determined by the heaviest parallel workload among the processors; the IPCOL bound for the sequential region is carried over from the IPCO bound (Ω_s''). The IPCOL bound is thus

$$IPCOL\ Bound = \Omega_s'' + Max_{q=1..N} \{\Omega_{p,q}''\}. \qquad \text{(EQ 22)}$$

In Figure 4-3, we illustrate how the IPCO and IPCOL bounds are calculated from a performance profile. The example run consists of a two-iteration loop, in which two parallel regions are both executed on two processors. The workload distribution for this example is shown in Figure 4-3(a). Since this example contains no sequential region, the IPCO bound (41) is essentially the average workload over the two processors, and the IPCOL bound (42) is the maximum overall workload between the two processors, as calculated in Figure 4-3(b). As indicated by the L gap, the load imbalance of overall workload causes an overhead of 1, which amounts to a 2.43% increase in execution time over a perfectly balanced execution.

4.3.6 Acquiring the IPCOLM Bound

The IPCOLM bound characterizes the multiphase load imbalance in the application. Multiphase load imbalance usually results from different workload distributions in different program phases of the application that are separated by barrier synchronizations. The execution time for each parallel region is determined by the most heavily loaded processor (the longest

Iteration/Region	Load on Processor 0	Load on Processor 1
1/1	10	5
1/2	10	15
2/1	5	6
2/2	15	16

(a) A Profile Example.

Iterations/Regions	Load on Processor 0	Load on Processor 1
All/All	40	42
IPCO Bound = (40 + 42)/2 = 41		
IPCOL Bound = Max{40, 42} = 42		
Load Imbalance Gap = IPCOL - IPCO = 42 - 41 = 1		

(b) Calculation of the IPCO and IPCOL Bounds.

Iterations/Region	Load on Processor 0	Load on Processor 1	Max. Load
All/1	15	11	15
All/2	25	31	31
IPCOLM Bound = (Max. Load of Phase 1) + (Max. Load of Phase 2) = 15+31 = 46			
Multiphase Gap = IPCOLM - IPCOL = 46 - 42 = 4			

(c) Calculation of the IPCOLM Bound.

Iteration/Region	Load on Processor 0	Load on Processor 1	Max Load
1/1	10	5	10
1/2	10	15	15
2/1	5	6	6
2/2	15	16	16
IPCOLMD Bound = Σ(Max. Load in each region for each iteration) = 10+15+6+16 = 47			
Dynamic Gap = IPCOLM - IPCOL = 47 - 46 = 1			

(d) Calculation of the IPCOLMD Bound.

Figure 4-3: Calculation of the IPCO, IPCOL, IPCOLM, IPCOLMD Bounds.

running thread) in that region. The IPCOLM bound is calculated by summing the execution time of the longest thread over the individual program regions, namely

$$IPCOLM\ Bound = \Omega_s'' + \sum_{r \in P} Max_{q=1..N} \{w_{r,q}\}. \qquad \text{(EQ 23)}$$

where $w_{r,q}$ is the *wall clock time* that processor q spent in region r, Ω_s'' is the lower bound for the sequential workload carried over from the IPCO bound, P is the set of parallel regions, and N is the number of processors.

The *Multiphase (M′) gap* (IPCOLM - IPCOL) characterizes the performance impact of multiphase load imbalance. Note that an application can pose serious multiphase load imbalance and still be well balanced in terms of total workload. As we illustrate in Figure 4-3(c), the calculation of the IPCOLM bound finds the local maxima for individual parallel regions and hence is never smaller than the IPCOL bound. The multiphase load imbalance in this example causes a Multiphase gap of 4, which equals 4/42 = 9.5% runtime increase over the IPCOL bound.

4.3.7 Actual Run Time and Dynamic Behavior

The actual run time is measured by the wall clock time of the entire application. The gap between the actual run time and the IPCOLM bound (*unmodeled gap*) should characterize both dynamic behavior and other factors that have not been modeled in the IPCOLM bound, e.g. the cost of spawn/join and synchronization operations.

Dynamic workload behavior can occur if the problem domain or the workload distribution over the domain changes over time. This happens often in programs that model dynamic systems. Dynamic behavior can result in an unpredictable load distribution and renders static load balancing techniques ineffective. An IPCOLMD bound could be generated to model the dynamic workload behavior *if* the execution time for each individual iteration could be separately reported by CXpa, i.e.

$$IPCOLMD\ Bound = \Omega_s'' + \sum_{r \in P} \sum_{i=1,Num\_Iter} Max_{q=1,N} \{w_{r,q,i}\}. \qquad \text{(EQ 24)}$$

where $w_{r,q,i}$ is the *wall clock time* that processor q spent in region r for iteration i, Ω_s'' is the lower bound for the sequential workload carried over from the IPCO bound, P is the set of parallel regions, Num_Iter is the number of iterations, and N is the number of processors.

Iteration/Region	Load on Processor 0	Load on Processor 1
1/1	15	5
1/2	5	15
2/1	5	15
2/2	15	5

(a) A Profile Example

Bound	Value	Gap from the Previous Bound
IPCO	40	N/A
IPCOL	40	0
IPCOLM	40	0
IPCOLMD	50	10

(b) Calculation of the IPCO and IPCOL Bounds

Figure 4-4: An Example with Dynamic Load Imbalance.

Unfortunately, CXpa is not suitable for measuring the execution time for each individual iteration, and hence CXbound cannot generate the IPCOLMD bound. So far, we have not found a proper tool to solve this problem on the HP/Convex Exemplar[1]. Thus in the later case studies of actual codes in this dissertation, the dynamic behavior effects are lumped together with the other "unmodeled effects" as the *unmodeled* (*X*) gap which is then calculated as (*Actual Execution Time) - (IPCOLM Bound).*

For applying the above equation to our example in Figure 4-3, the maximum workload in each instance of a parallel region is first computed, as shown in the 'Max. Load' column of Figure 4-3(d). Then, the IPCOLMD bound is calculated by summing the maximum load in each region for each iteration. The Dynamic (D) gap characterizes the performance impact of the dynamic load imbalance in the application. The D gap in the above example is primarily due to the change of load distribution in region 1 from iteration 1 to iteration 2. A more dynamic example is given in Figure 4-4(a), and the performance problem, i.e. the dynamic behavior, is revealed via the bounds analysis shown in Figure 4-4(b).

1. Two tracing tools are available on the HP/Convex Exemplar. CXtrace [79], a tracing tool developed by Convex, supports PVM programming only. SMAIT [38] was not designed to directly measure application performance.

4.4 Characterizing Applications Using the Parallel Bounds

In this section, we demonstrate the use of the hierachical bounds to characterize application performance in a user-friendly and effective fashion. First, in Section 4.4.1, to prove these concepts, we characterize a few matrix operation programs with *CXbound*. These examples illustrate the correlation between the performance gaps and the weakness in the programs. Then, in Section 4.4.2, to show the effectiveness of this characterization methodology, we characterize a large finite-element application and show how well the performance behavior of this relatively complex application is captured by these easily understood performance bounds.

4.4.1 Case Study 1: Matrix Multiplication

4.4.1.1 Baseline Matrix Multiplication

Matrix multiplication, as shown in Figure 4-5, is one of the most frequently used computer algorithms. In this characterization, the subroutine *MULT1* was executed 100 times *statically* without any change. The initialization and multiplication loop were automatically parallelized by the Convex Fortran compiler [26][27] with optimization level 3. *Spawn* and *join* were inserted at the beginning and the end of each parallel loop by the compiler.

The results of the bounds analysis on *MM1* for configurations of 1 to 8 processors are shown in Figure 4-6, and Figure 4-7. From Figure 4-7, It is clear that communication becomes a more serious performance bottleneck as the number of processors grows. Focusing on the communication pattern, we found that the compiler parallelized the initialization loop on index *i*, while the multiplication loop is parallelized on index *j*. The compiler interchanges the loop indices for the multiplication loop to optimize the data access pattern. As we have discussed in Chapter 3, multiple domain decompositions incur data redistribution cost. Thus, by parallelizing on two different indices, the code generated more coherence communication than it would if those two loops were both parallelized on index *j*.

To correct this problem, we manually interchanged the indices *i* and *j* for the two loops[1], as shown in Figure 4-8. The two loops in this new program, *MM2*, are now both parallelized

```
1          program MM1
2          integer iter
3          do iter = 1, 100
4             call MULT1
5          end do
6          end

7          subroutine MULT1
8          parameter (N=512)
9          double precision a(N,N), b(N,N), c(N,N)
10         integer i, j, k

11c initialization loop
12         do i=1, N
13          do j=1, N
14                 a(i,j) = sin(i*j*0.1)
15                 b(i,j) = cos(i*j*0.1)
16           end do
17         end do

18c multiplication loop
19         do i=1, N
20           do j=1, N
21                 c(i,j) = 0.0
22                 do k=1, N
23                  c(i,j) = c(i,j) + a(i,k) * b(k,j)
24                 end do
25           end do
26         end do

27         end
```

Figure 4-5: The Source Code for MM1.

on index *j*. The performance bounds of *MM2*, shown in Figure 4-9, indicate that the performance of *MM2* is generally better than *MM1*, primarily because the *communication* gap (gap between the IPC and IP bound) is reduced. This difference is more visible in Figure 4-10 which shows the performance comparison of *MM1* and *MM2* for an 8-processor configuration. Since the two loops in *MM2* are parallelized consistently, MM2 maintains a better cache utilization (processor locality), and hence *MM2* has less communication overhead.

4.4.1.2 Load Unbalanced Matrix Multiplication

To illustrate the effects of load imbalance, we split the matrix multiplication loop into two poorly balanced loops, as shown in program *MM_LU* in Figure 4-11. For each of these two loops, the workload for a particular *j* depends on the value of *j*, since the iteration space of the

1. Actually, we only need to interchange the initialization loop. In contrast to *MM1*, the multiplication loop in *MM2* is explicitly interchanged.

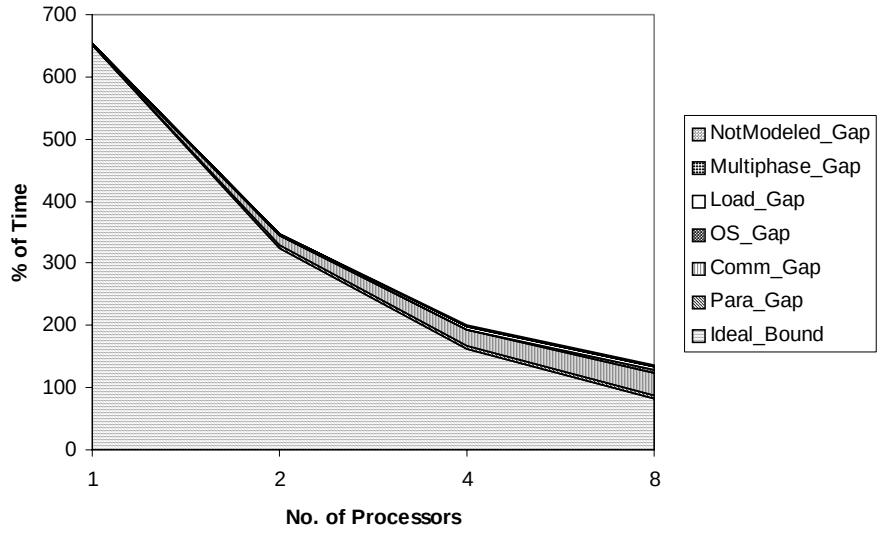


Figure 4-6: Parallel Performance Bounds for MM1.

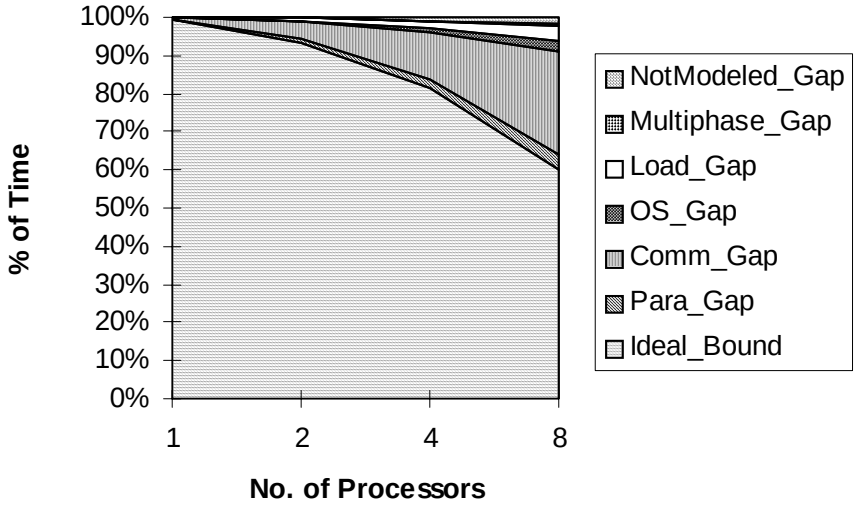


Figure 4-7: Performance Gaps for MM1.

```
1          program MM2
2          integer iter
3          do iter = 1, 100
4             call MULT2
5          end do
6          end

7          subroutine MULT2
8          parameter (N=512)
9          double precision a(N,N), b(N,N), c(N,N)
10         integer i, j, k

11c initialization loop
12         do j=1, N
13          do i=1, N
14                a(i,j) = sin(i*j*0.1)
15                b(i,j) = cos(i*j*0.1)
16           end do
17         end do

18c multiplication loop
19         do j=1, N
20           do i=1, N
21                c(i,j) = 0.0
22                do k=1, N
23                 c(i,j) = c(i,j) + a(i,k) * b(k,j)
24                end do
25           end do
26         end do

27         end
```

Figure 4-8: The Source Code for MM2.

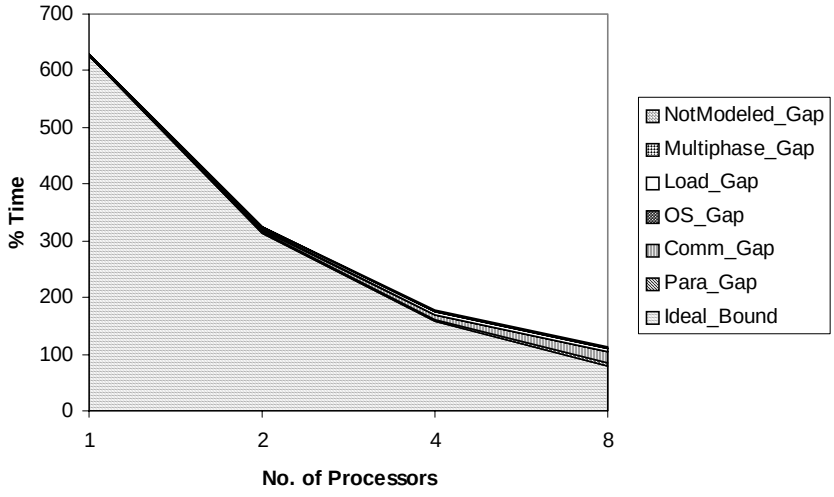


Figure 4-9: Parallel Performance Bounds for MM2.

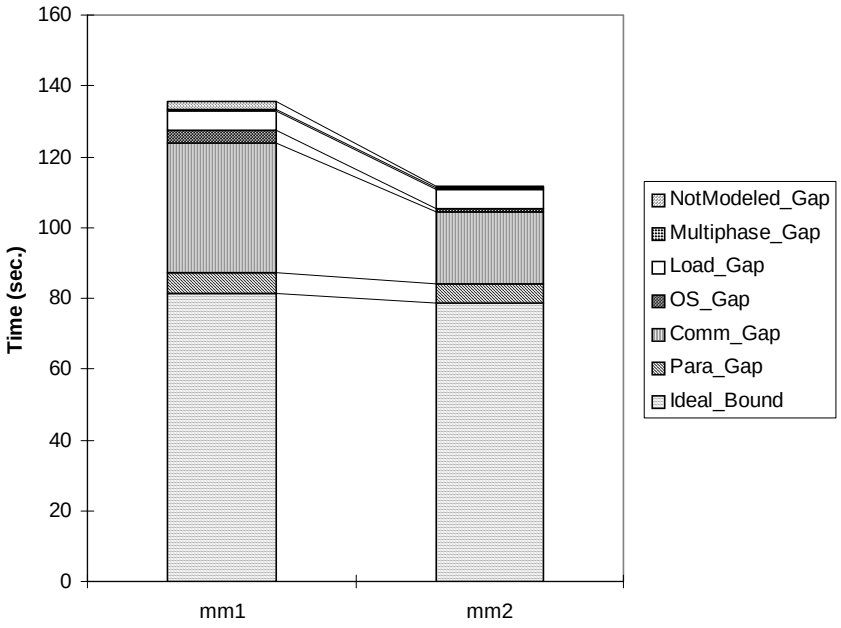


Figure 4-10: Comparison of MM1 and MM2 for 8-processor Configuration.

innermost loop *k* depends on the value of *j*. It should not be surprising that the compiler failed to balance the load distribution in these two loops. The compiler simply parallelizes the iteration space of loop *j* evenly. As a result, each of these two loops exhibits poor load balance. The bounds analysis of the program, shown in Figure 4-12, correctly indicates that the cause of the load imbalance is the multiphase load imbalance (not the overall load imbalance, since the load summed over both loops is in fact fairly well balanced).

We note that the I bound of *MM_LU* (shown in Figure 4-12) is much larger than that of *MM1* due to its poor cache behavior. The compiler will not apply loop blocking for any loop whose iteration space is not constant. Consequently, it did not apply loop blocking within the *k*-indexed loop of MM_LU (shown in Figure 4-11).

4.4.1.3 Performance in a Multitasking Environment

The performance of a parallel program can be very sensitive to interference from other programs running on the same machine. Here we study such a case by characterizing the performance of *MM2* when it was performed on a heavily loaded multitasking system. Figure 4-13 shows the performance comparison of *MM2* on dedicated and multitasking configurations

```
1          program MM_LU
2          integer iter
3          do iter = 1, 100
4             call MULT_LU
5          end do
6          end

7          subroutine MULT_LU
8          parameter (N=512)
9          double precision a(N,N), b(N,N), c(N,N)
10         integer i, j, k

11c initialization loop
12         do j=1, N
13          do i=1, N
14                a(i,j) = sin(i*j*0.1)
15                b(i,j) = cos(i*j*0.1)
16           end do
17         end do

18c multiplication loop 1
19         do j=1, N
20           do i=1, N
21                c(i,j) = 0.0
22                do k=1, j
23                 c(i,j) = c(i,j) + a(i,k) * b(k,j)
24                end do
25           end do
26         end do

27c multiplication loop 2
28         do j=1, N
29           do i=1, N
30                do k=j+1, N
31                 c(i,j) = c(i,j) + a(i,k) * b(k,j)
32                end do
33           end do
34         end do

35         end
```

Figure 4-11: Source Code for MM_LU

using four processors. The execution time on the multitasking processors is about 8 times longer than on the dedicated processors.

The bounds analysis shows that most performance gaps are larger with multitasking. The Parallelization gap is increased primarily because the interference results in cache pollution and memory contention which in turns increases the sequential workload. The Communication gap is slightly increased due to cache pollution and memory contention. The OS gap is much larger because of the increased wall clock time caused by the interruptions from the OS for task switching. The Load Balance gap indicates that multitasking might affect the execu-

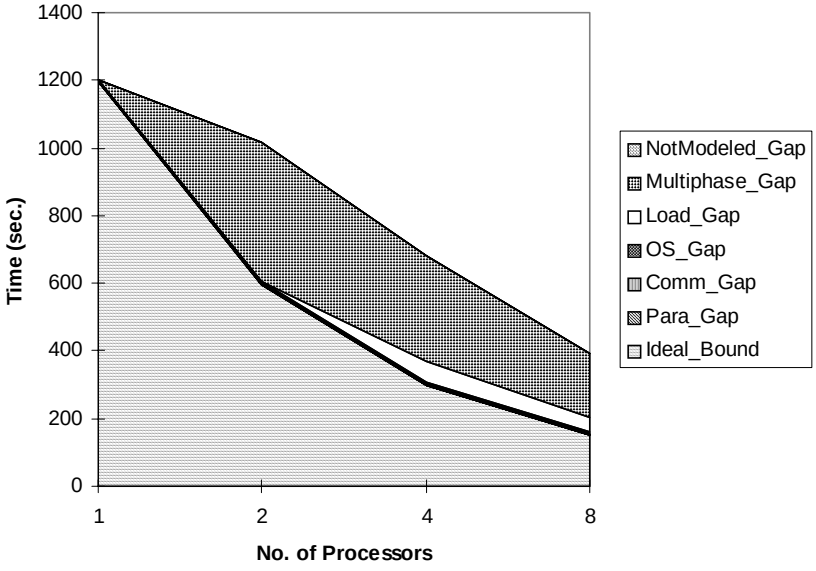


Figure 4-12: Performance Bounds for MM_LU

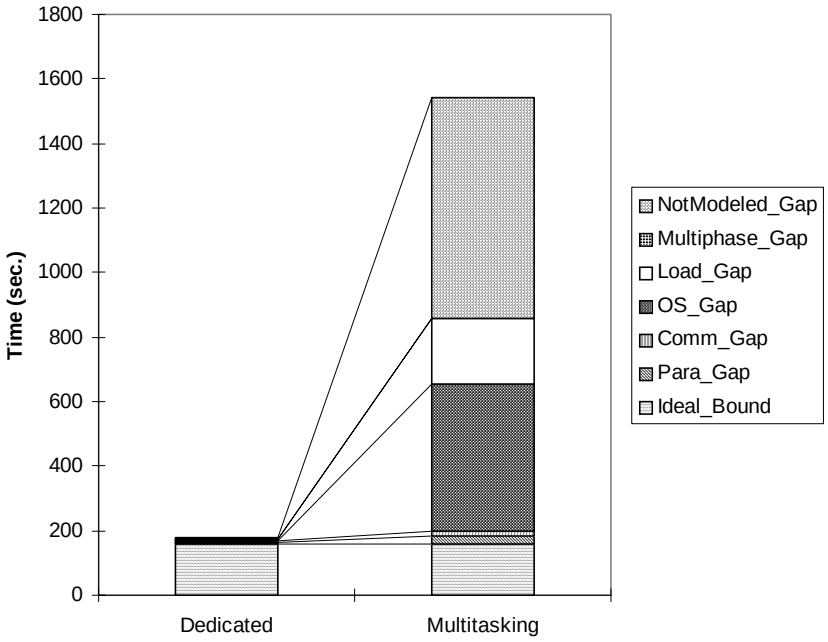


Figure 4-13: Performance Comparison of MM2 on Dedicated and Multitasking Systems

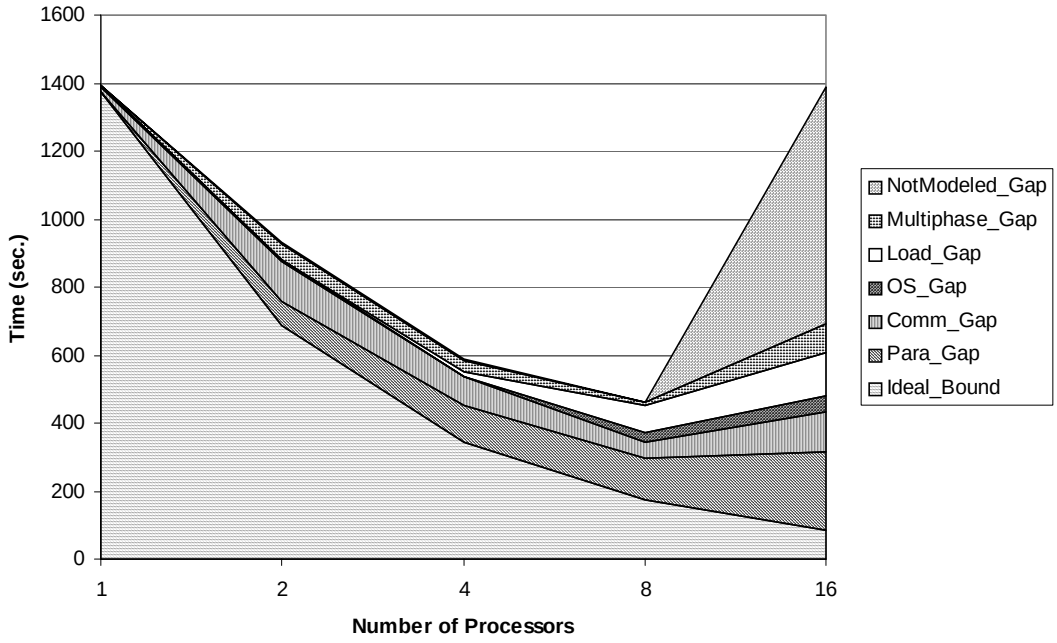


Figure 4-14: Performance Bounds for the *Ported* Code

tion differently on individual processors, e.g. one processor might be affected more than the others. The Multiphase gaps are virtually nonexistent because MM2 contains only one dominant loop. The large unmodeled gap is possibly due to dynamic behavior caused by sporadic interference in the multitasking environment.

4.4.2 Case Study 2: A Finite-Element Application

In this section, we demonstrate the effectiveness of the performance bounds analysis on a representative portion of a commercial full vehicle crash simulation code. *Ported* is the version of the application that was ported to the HP/Convex Exemplar without machine-dependent performance tuning. *Ported* was converted from serial code by manually parallelizing the most time consuming loops, which collectively account for about 90% of the workload on a one-processor Exemplar SPP-1600 run.

The performance of *Ported* is characterized by the bounds and gaps in Figure 4-14. The actual performance is effectively accelerated within one 8-processor hypernode, thanks to low intra-hypernode communication latency. However, as the number of processors increases beyond one hypernode, the combined effects of partial parallelization, inter-hypernode com-

munication, load imbalance, and unmodeled behaviors prohibit the code from further speedup. Certain initial performance tuning directions are suggested by these bounds:

- *P-gap*: the degree of parallelization should be improved
- *C', L, M' gaps*: better domain decomposition schemes should be adapted to control the communication and improve load balance.
- *Unmodeled gap*: a static domain decomposition should be used to reduce dynamic task migration by binding the computations and data to the processors permanently.

We improved the degree of parallelization and the domain decomposition with a version called *Tuned*. A domain decomposition tool, *Metis* [29], and a weighted domain decomposition technique [2][51] were adapted for decomposing the finite-element graphs and the computation associated with the graph. This explicit domain decomposition has the desirable side-effect of exposing the data dependencies between subdomains in *Tuned*. This information helps the programmer to manually parallelize more loops. Also, each processor is now responsible for the computation associated with one subdomain with a permanent assignment, which minimizes unfortunate task migrations during runtime.

The performance characterization of *Tuned* is shown in Figure 4-15. A comparison of performance on the 16-processor configuration (Figure 4-16) shows that *Tuned* achieves better performance, due to a higher degree of parallelization (P gap), less communication overhead (C′ gap), and better load balance (L gap). The unmodeled gap is not noticeable for *Tuned*, because *Tuned* uses a permanent task assignment scheme.

Nevertheless, *Tuned* still runs slower on 16 processors than it runs on 8 processors. Imperfect parallelization, communication overhead, overall load imbalance, and multiphase load imbalance all contribute to the poor performance on 16 processors. Further performance tuning techniques, as discussed in Chapter 3, are necessary to address these problems.

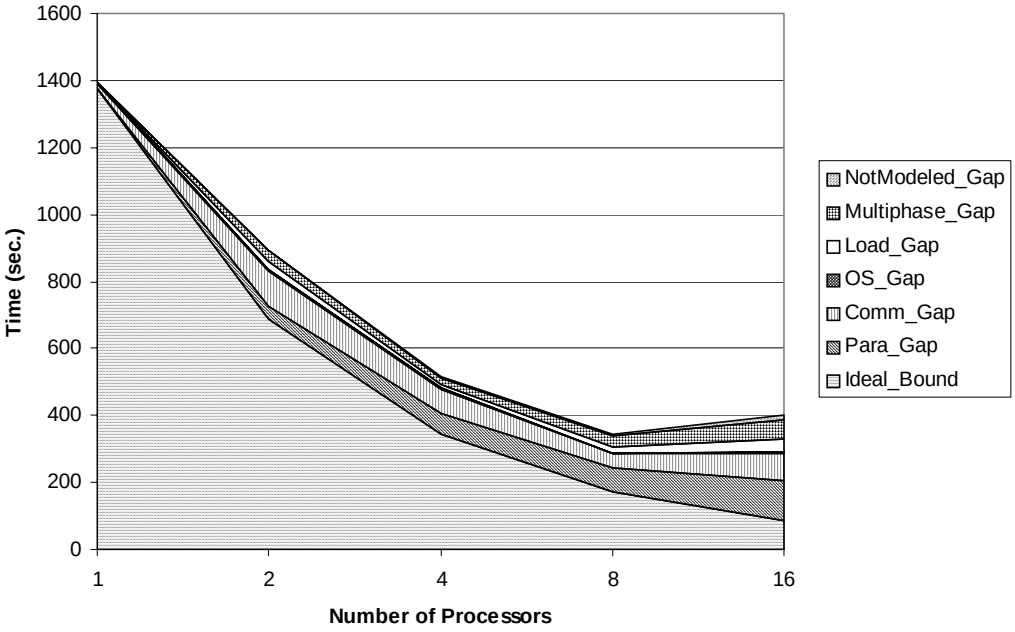


Figure 4-15: Performance Bounds for the *Tuned* Code

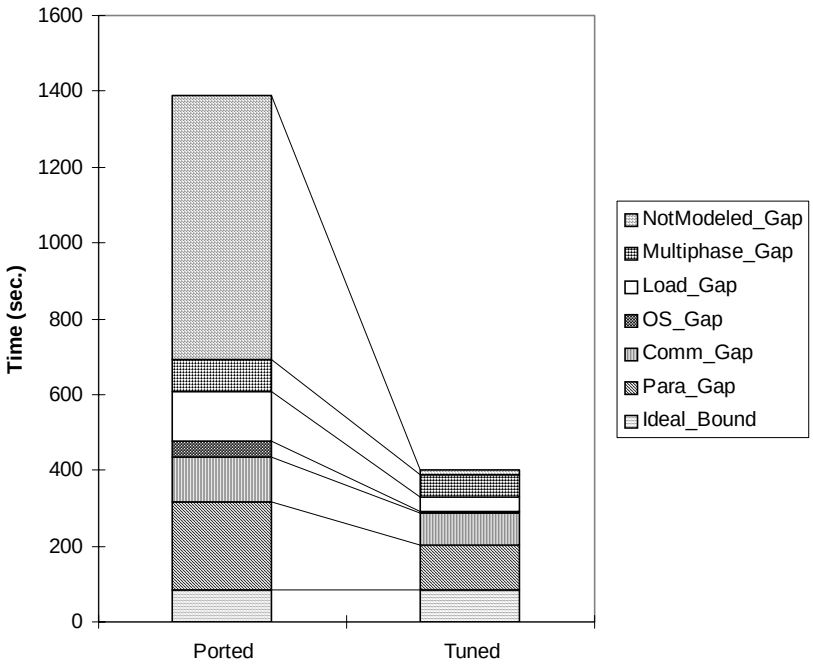


Figure 4-16: Performance Comparison between *Ported* and *Tuned* on 16-processor Configuration

4.5 Summary

As we have demonstrated in Section 4.4, relatively complicated performance profiles can be summarized with a relatively simple set of performance bounds to provide more effective performance visualization and insights into program behavior. The performance bounds methodology implemented in our automatic tool, *CXbound*, has successfully pinpointed performance weaknesses of the parallel applications in our case study. Our experiences show that the bounds analysis can effectively detect performance bottlenecks, guide the use of performance tuning techniques, and evaluate the results of performance tuning.

The bounding mechanism in *CXbound* depends on the performance profiles provided by *CXpa*. As a result, the limitations of *CXpa* also affect the implementation of *CXbound*. Major limitations in the current *CXbound* implementation are discussed below:

- Proper utilization of *CXpa* is necessary to ensure accurate performance bounds. The program should be profiled on machines as *clean* as possible to avoid disturbance from other tasks, unless such disturbance is of interest to the performance study. Since the bounds obtained from *CXbound* are based on the profile of individual runs, one set of bounds may not characterize the performance of other runs.

- As mentioned in Section 4.3.1, the I bound and the IP bound do not consider the increased cache capacity as more processors are used. Therefore they can be misleading if the cache behavior on the one-processor configuration is very different from that on an *N*-processor configuration. Hence, cache misses reported in the *CXpa* profiles are used to assess the accuracy of these two bounds. To gain better accuracy for the I and IP bounds, trace-driven simulation tools can be used to isolate the coherence misses in the *N*-processor execution. For the same reason, the gap between the IPC and IP bounds characterizes the negative effect of increased communication plus the beneficial effect of reduced cache misses. Again, cache miss reports should be used to assess whether the effect of increased cache capacity is negligible.

- Since *CXpa* is not designed to profile individual iterations, *CXbound* cannot automatically generate the IPCOLMD bound. Thus, the performance impact of dynamic behavior is included in the gap between the actual runtime and the IPCOLM bound, but cannot be differentiated from other unmodeled factors with this tool.

- *CXbound* does not recognize parallel program structures that *CXpa* does not recognize. This limits automatic bounds generation to parallel structures that are formed by compiler directives. For the user to profile unrecognized parallel structures, such as those that are formed by hand-coded synchronizations, manual instrumentation may be necessary. We found that a pseudo-loop can often be used to identify a code segment. By enclosing a code segment in a pseudo-loop that iterates only once, the enclosed code segment is recognized and is therefore profiled by *CXpa* as a non-parallel loop. However, special instructions will be needed by *CXbound* to handle such loops.

- The current implementation of *CXbound* does not support *message-passing* codes. Message-passing codes perform communications via message-passing libraries, and the communication time spent in the library is not reported by *CXpa* unless the library itself is instrumented.

Although these limitations may compromise the usefulness of *CXbound* in some cases, the current implementation of *CXbound* has demonstrated the potential of the hierachical performance bounds analysis and has been quite useful in our case studies (as illustrated in Section 4.4).

There are several possible directions for the future development of *CXbound*. Some of the *CXbound* limitations can be eliminated by adding a few enhancements to *CXpa*, such as recognizing more parallel structures and profiling individual iterations. While such enhancements can be very difficult for the user to implement, the vendor (HP/Convex) should be able to accomplish them with modest effort. Also, porting *CXbound* to other platforms should be straightforward if the target machines are made to provide similar profile and/or trace tools (e.g. [119][120]). As we have shown in this section, calculation of the bounds is simple, given proper support from the machine.

Step	Performance Issue	Tuning Action	Targeted Performance Gap(s) in this Step	Primarily Affected Performance Gaps
Partitioning the Problem (Step 1)	(I-1) Partitioning an Irregular Domain	(A-1) Applying a Proper Domain Decomposition Algorithm for (I-1)	P	$, P, C', L, M', D
Tuning the Communication Performance (Step 2)	(I-2) Exploiting Processor Locality	(A-2) Proper Utilization of Distributed Caches for (I-2)	C'	$, C'
		(A-3) Minimizing Subdomain Migration for (I-2)	C'	C', L, M', D
	(I-3) Minimizing Interprocessor Data Dependence	(A-4) Minimizing the Weight of Cut Edges in Domain Decomposition for (I-3)	C'	C', L
	(I-4) Reducing Superfluity	(A-5) Array Grouping for (I-4)	C'	C', $
	(I-5) Reducing Unnecessary Coherence Operations	(A-6) Privatizing Local Data Accesses for (I-5)	C'	C', $, M
		(A-7) Optimizing the Cache Coherence Protocol for (I-5)	C'	C', $
		(A-8) Cache-Control Directives for (I-5)	C'	C', $
		(A-9) Relaxed Consistency Memory Models for (I-5)	C'	C', $
		(A-10) Message-Passing Directives for (I-5)	C'	C', $
	(I-6) Reducing the Communication Distance	(A-11) Hierarchical Partitioning for (I-6)	C'	C', $, L
		(A-12) Optimizing the Subdomain-Processor Mapping for (I-6)	C'	C'
	(I-7) Hiding the Communication Latency	(A-13) Prefetch, Update, and Out-of-order Execution for (I-7)	C'	C', $, M
		(A-14) Asynchronous Communication via Messages for (I-7)	C'	C', $
		(A-15) Multithreading for (I-7)	C'	C', $, S
	(I-8) Reducing the Number of Communication Transactions	(A-16) Grouping Messages for (I-8)	C'	C', S
		(A-17) Using Long-Block Memory Access for (I-8)	C'	C', S, $
	(I-9) Distributing the Communications in Space	(A-18) Selective Communication for (I-9)	C'	C'
	(I-10) Distributing the Communications in Time	(A-19) Overdecomposition to Scramble the Execution for (I-10)	C'	C', S, $

Table 4-1: Performance Tuning Actions and Their Related Performance Gaps. (1 of 2)

Step	Performance Issue	Tuning Action	Targeted Performance Gap(s) in this Step	Primarily Affected Performance Gaps
Optimizing Processor Performance (Step 3)	(I-11) Choosing a Compiler or Compiler Directive	(A-20) Properly Using Compilers or Compiler Directives for (I-11)	C, S, $	C, S, $, C'
		(A-21) Goal-Directed Tuning for Processor Performance for (I-11)	-	-
	(I-12) Reducing the Cache Capacity Misses	(A-22) Cache Control Directives for (I-12)	$	$, C
		(A-23) Enhancing Spatial Locality by Array Grouping for (I-12)	$	$, C
		(A-24) Blocking Loops Using Overdecomposition for (I-12)	$	$, C
	(I-13) Reducing the Impact of Cache Misses	(A-25) Hiding Cache Miss Latency with Prefetch and Out-of-Order Execution for (I-13)	$	$, C, M, S
		(A-26) Hiding Memory Access Latency with Multithreading for (I-13)	$	$, C, S
	(I-14) Reducing Conflicts of Interest between Improving Processor Performance and Communication Performance	(A-27) Repeating Steps 2 and 3 for (I-14)	C, S, $, C'	C, S, $, C'
Balancing the Load for Single Phases (Step 4)	(I-15) Balancing a Nonuniformly Distributed Load	(A-28) Profile-Driven Domain Decomposition for (I-15)	L	L, C'
		(A-29) Self-Scheduling for (I-15)	L	L, C', M', D
Reducing the Synchronization/ Scheduling Overhead (Step 5)	(I-16) Reducing the Impact of Load Imbalance	(A-30) Fuzzy Barriers for (I-16)	L, M'	L, M', D
		(A-31) Point-to-Point Synchronizations for (I-16)	L, M'	L, M', D
		(A-32) Self-scheduling of Overdecomposed Subdomains for (I-16)	L, M'	L, M', D
	(I-17) Reducing the Overall Scheduling/ Synchronization Overhead		L, M', D	L, C', M', D
Balancing the Combined Load for Multiple Phases (Step 6)	(I-18) Balancing the Load for a Multiphase Program	(A-33) Balancing the Most Critical Phase for (I-18)	M'	M', L, D
		(A-34) Multiple Domain Decompositions for (I-18)	M'	M', C', L, D
		(A-35) Multiple-Weight Domain Decomposition Algorithms for (I-18)	M'	M', L, D
		(A-36) Fusing the Phases and Balancing the Total Load for (I-18)	M'	M', L, D
Balancing Dynamic Load (Step 7)	(I-19) Reducing the Dynamic Load Imbalance	(A-37) Dynamically Redecomposing the Domain for (I-19)	D	D, C', L, M'
		(A-38) Dynamic/Self-Scheduling for (I-19)	D	D, C', L, M'
		(A-39) Multiple-Weight Domain Decomposition for (I-19)	D	D, C', L, M'
	(I-20) Tolerating the Impact of Dynamic Load Imbalance	(A-40) Relaxed Synchronizations for (I-20)	D	D, C', L, M'

Table 4-1: Performance Tuning Actions and Their Related Performance Gaps. (2 of 2)

CHAPTER 5. MODEL-DRIVEN PERFORMANCE TUNING

From our experience with performance tuning, we have found that a high-level abstraction of the application is quite helpful for assessing application performance. In this chapter, we discuss our application performance modeling methodology, *Model-Driven Analysis* (MDA), which analyzes an application's performance with an intermediate-level abstraction of the machine-application behavior. With MDA, we proposed a methodology, called *Model-Driven Performance Tuning* (MDPT) to facilitate the conventional performance tuning process by employing the application model as the object of performance tuning. We developed several key tools and conducted a preliminary study to evaluate this novel approach.

In this preliminary study, we target at providing a fast and robust mechanism for estimating the effectiveness of the applied tuning actions, as well as resolving the conflicts among these actions. Our case study shows that high-level application models can be well-built by programmers using their knowledge about the application and basic performance assessment tools, especially when the knowledge and tools that are required to build application models are already extensively used in performance tuning. Implementing performance tuning with the aid of these models is fairly straightforward, and MDA is useful in guiding the use of performance tuning techniques as well as resolving the conflicts among them.

5.1 Introduction

The machine-application interactions in a parallel system can be very complicated, even to a performance tuning expert. While various performance assessment tools exist to help expose the problems in the application performance and extract specific machine-application interactions, the performance data collected by previously existing tools can be very complicated and time-consuming to analyze. In a hand tuning situation, the programmer has to carry out this work repeatedly since most high-level program analysis and performance tuning decisions are done by the programmer.

Unfortunately, most *application developers* are unable or unwilling to deal with such a complex performance tuning process, even when provided with various tools and a systematic tuning paradigm. In our experience working with application developers, as computer architects and performance tuning specialists, we often have to spend a considerable amount of time learning and analyzing the behavior of the application. Ironically, we have spent much time learning what application developers already knew about their applications, such as program flow, data access patterns, etc. We believe that a better way of collaboration is needed to improve the communications between application developers and performance tuning specialists so that each side can focus more on their specialities.

We address this issue by forming an *intermediate representation* of the application to facilitate the communications between the application developer and the performance tuning specialist, as well as to simplify the problems from each side's point of view. We call this intermediate representation an *application model*. Our methodology, called *Model-Driven Analysis* (MDA), was developed for analyzing the application performance by firstly transforming the application into a model and simulating the machine-application interactions on the model.

The first phase in MDA is called *Application Modeling* (AM), which generates specifications of the application behavior, including the application's *control flow*, *data dependence*, *domain decomposition*, and the *weight distribution* over the domain. This phase can be carried out by the application developer with minimal knowledge about machine-application interactions. We have designed a language, called the *Application Modeling Language* (AML) for the user to specify the application model and incorporate results from performance assessment tools, such as profiling.

The second phase in MDA, called *Performance Modeling* (PM), derives performance information based on the application model. In PM, the application model is analyzed via simulation. A simulation tool, called the *Model-Driven Simulator* (MDS), is developed to analyze the data flow, working set, cache utilization, workload, degree of parallelism, communication pattern, and the hierarchical performance bounds. MDS performs a broad range of analysis that would require combinations of conventional performance assessment tools. Results from MDS are used to validate the application model by comparing results with those of previous perfor-

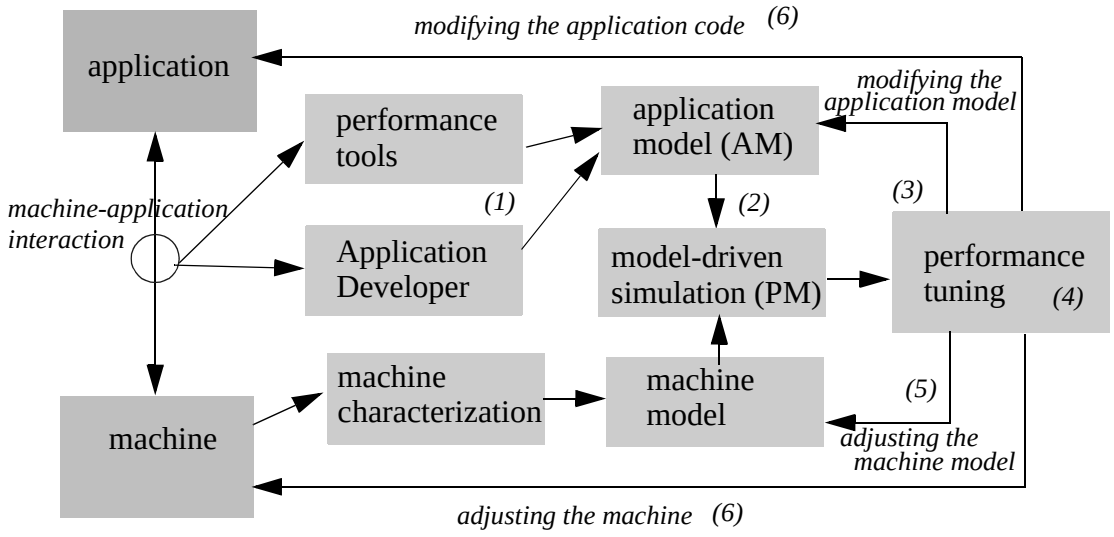


Figure 5-1: Model-Driven Performance Tuning.

mance assessments in known cases (both cases that were previously used to generate the model, as well as new cases with new profiles).

The MDA methodology supports the notion of *Model-Driven Performance Tuning* (MDPT), where the application model, instead of the application, becomes the object of performance tuning. In MDPT, proposed performance tuning actions are first installed in the application model and evaluated via MDS to assist the user in making tuning decisions. This concept of the MDPT approach, the capabilities it provides, and its potential are illustrated in Figure 5-1 and discussed below: (each of the following paragraphs is indicated by a number in the figure)

1. Various sources of performance assessment and program analysis contribute to the AM phase for providing a more complete, accurate model. Performance assessment tools and application developers both contribute to creating the application model.

2. In the PM phase, MDS is carried out to derive information by analyzing the machine-application interactions between the application model and the machine model. The machine model is based on the machine characterization techniques discussed in Section 2.1.

3. The application model serves as a medium for experimenting with the application of performance tuning techniques as well as resolving the conflicts among them. In MDPT, performance tuning techniques are first applied and evaluated on the application model using MDA and then ported to the code, which can potentially shorten the application development time.

4. The application model can be tuned by either the programmer or the compiler. A properly abstracted application model helps the user or the compiler assess the application performance at an adequate level, without the overkill burden of tuning by carrying out transformations and performance analysis directly on the application and repeatedly handling the high volume of raw performance data that is produced.

5. In addition to tuning the application code, the machine model can be tuned to improve the application performance. Using MDA, the users are given the opportunity to evaluate various machine configurations or different machines for specific applications without actually reconfiguring or building the machine.

6. After tuning actions are evaluated with MDA, they are applied to the application code and/or the target machine to assess the actual improvement, validate, and possibly recalibrate the models.

In this chapter, we propose an application development environment for experimenting with the concept of MDPT, as illustrated in Figure 5-2. We gather program/performance information from the programmer, and from existing performance tools such as CXpa. In Section 5.2, we discuss how to model an application and how we build the application model using our application modeling language (AML). We have developed an MDS that incorporates machine models based on the HP/Convex Exemplar SPP-1000/1600. In Section 5.3, we show how the data flow, the communication pattern, the load imbalance, and the execution time are estimated by incorporating several techniques discussed in the previous chapters into MDS. In Section 5.4, we illustrate MDA and MDPT with our example irregular application, CRASH (see Section 1.3.2). Section 5.5 summarizes the results.

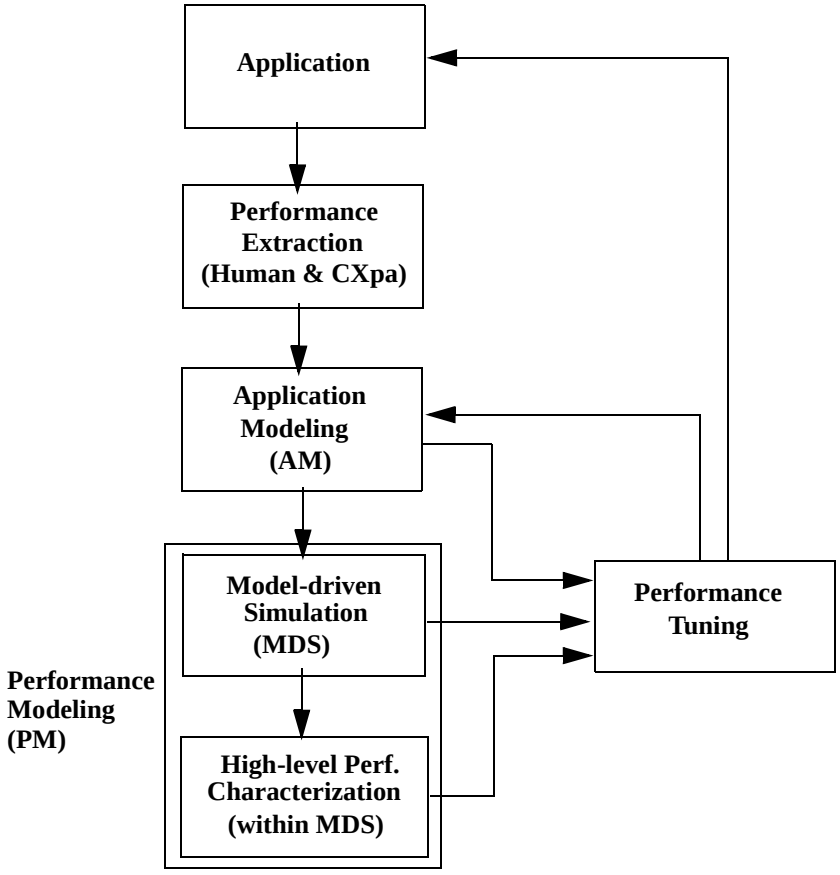


Figure 5-2: Model-driven Performance Tuning.

5.2 Application Modeling

The performance of an application is fundamentally governed by (1) the program (algorithm), (2) the input (data domain/structures), and (3) the machine that are used to execute the application. It is relatively difficult to observe machine-application interactions at this level, since detailed machine operations are often hidden from the programming model that is available to the programmer.

We would like to model the application at a level that provides us with more precise information on how the application behaves, especially the behavior that directly affects performance. The *control flow* and the *data dependence* in the application are modeled because they limit the instruction schedule and determine the data access pattern for the application. The *decomposition* of the input data determines the decomposition of the workload (for a SPMD

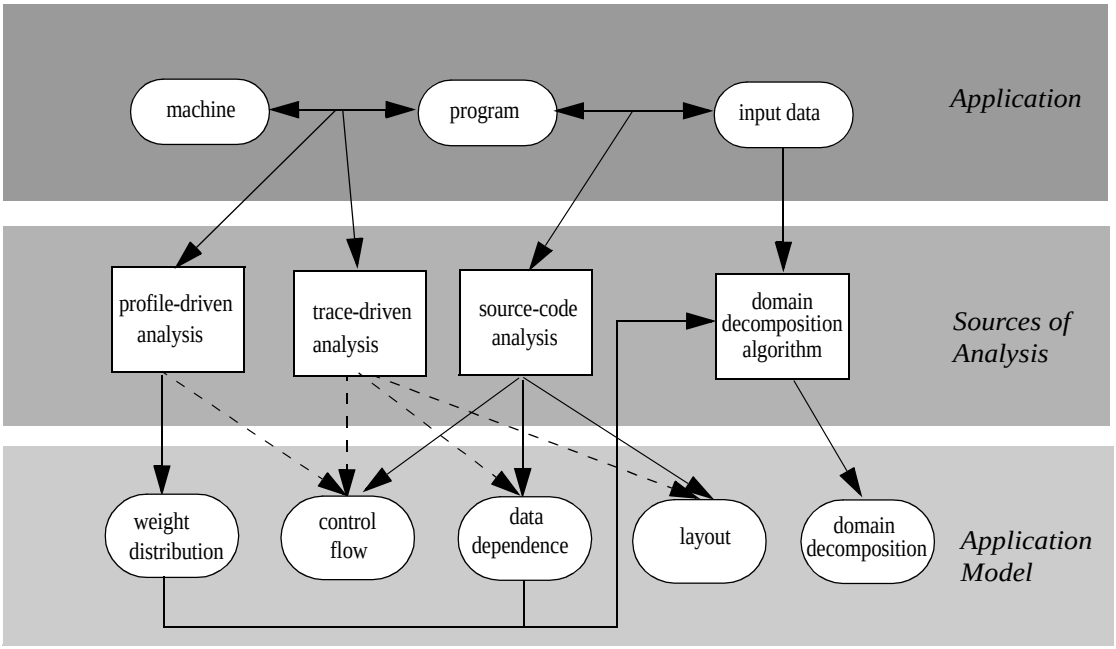


Figure 5-3: Building an Application Model.

application). The *layout of the data structure* determines the data allocation and affects the actual data flow in the machine, especially for a cache-based, distributed shared-memory application. The *workload* in the application certainly requires resources from the processors and hence needs to be modeled for addressing load balance problems. An application model is acquired by abstracting (1) control flow, (2) data dependence, (3) domain decomposition, (4) data layout, and (5) the workload from the application. These five components are hereafter referred to as *modules* of the application model.

Figure 5-3 illustrates how we model a CRASH-like application on the HP/Convex Exampler via the use of source code analysis (mostly done by the programmer), profiling (CXpa), and trace-driven simulation tools (e.g. Smait, Dinero, CIAT/CDAT). In this flow chart, a solid line indicates a path that we currently employ to create a particular module, and a dashed line indicates an additional path that might be useful for creating the module. We briefly describe the process used to create these modules for a CRASH-like application as follows:

- The control structure, the data dependence, and the data layout that are encoded in the program are abstracted via source code analysis. While analyzing irregular applications can be difficult for compilers, it can be done by humans, especially the author(s) of the

code. However, since compilers are useful for analyzing most regular applications, we assume that the generation of these modules can be done mostly by converting the results of compiler analysis (from the internal representation used by the compiler), thereby freeing the programmer to focus on the irregularities in the application.

- We use weights to represent the application workload in different code sections. Although the instruction sequence in a code section can be extracted to model the workload, accurately predicting the execution time of the code section based on the instruction sequence can be rather complicated and difficult. Profile-driven analysis can be used straightforwardly by the user for extracting the weights where the load is uniformly-distributed. For non-uniformly-distributed cases, techniques such as the weight classification and predication method developed by Tomko and Davidson [30] may be needed.

- In an SPMD application, the computation is decomposed by decomposing the domain. The domain decomposition can be implicitly specified in the application by DOALL statements, or explicitly programmed into the code according to the output of a domain decomposition package such as Metis [29]. As mentioned in Section 1.3.1, the data dependence and the weight distribution of the application are given as inputs to the domain decomposition package.

- Ambiguities in the control flow and the memory references, e.g. IF-THEN statements and indirect array references, reveal major weaknesses in compiler analysis. It is possible to clarify these ambiguities by profiling or tracing the application, which can be used to automate the specifications of control flow and data dependence or validate the model.

We believe that building such an application model is highly feasible for the application developers with the programming tools available today. Most of the above procedures involved in modeling an application require very little knowledge about the target machine, and tools, such as profiling, provide additional help in measuring the workload and also helping the programmer to extract the application behavior.

Depending on the user, some of the modules can be modeled in detail, while some may be relatively simplified. More precise modeling of application behavior may occasionally be needed in some situations for carrying out specific performance tuning techniques and solving particular performance problems when the simplified modules are shown to be inadequate. In general, however, it is most efficient to use a model that is as simple as possible for dealing with the major problems being addressed. Simplifying the modeling of less relevant applica-

```
INTEGER Num_Elements;
INTEGER Type (1:Max_Elements) ALIGNED_32;
INTEGER Num_Neighbors (1:Max_Elements) ALIGNED_32;
INTEGER Neighbor (1:Max_neighbor_per_Element,1:Max_Elements) ALIGNED_32;
VECTOR Force (1:Max_Elements) ALIGNED_32;
VECTOR Position (1:Max_Elements) ALIGNED_32;
VECTOR Velocity (1:Max_Elements) ALIGNED_32;
```

Figure 5-4: An Example Data Layout Module for CRASH.

tion behavior may reduce the amount of work required to model the application and the complexity of analyzing the model. The design of AML intends to provide the user with flexibility in modeling of the application at various levels of detail.

We incrementally extend the capability of AML to model the application behavior that we have encountered in our test cases. So far, we have successfully modeled several programs from the NAS benchmarks as well as CRASH-SP and CRASH-SD. The syntax of our preliminary version of AML is quite simple and limited, because it is intended to prove the concept of model-driven analysis and model-driven performance tuning. As currently implemented, AML input is actually written in tabular form. However, AML is being revised to add some capabilities and improve its usability, primarily by implementing a parser that can interpret the more humanly readable (and writable) input format used in this dissertation, and convert it to the tabular form that AML uses internally.

5.2.1 The Data Layout Module

The data layout module declares the type, size, and layout for the major data structures used in the application. Figure 5-4 shows a data layout module for CRASH. The first argument in each line defines the type of the data structure, the second argument is the name of the data structure, then the dimension of the data structure is specified (if it is an array), and finally the alignment or specific layout method can be attached.

Currently, AML supports most Fortran data structure types and a few user-defined types, such as VECTOR in the above example. More sophisticated C language data structures, such as structures, unions, and pointers, add considerable complexity to the parser of MDS, so they are not present in the current version, but they are certainly realizable in a future version.

The array dimension by default uses Fortran language's row-major convention. MDS can be configured to use C language's column-major convention. AML allows the user to align data structures to begin at 2^n-byte boundaries. The arrays in the above example are all aligned to 32-byte boundaries to ensure that they do not reside in the same cache line together with any other data structures.

As far as performance is concerned, modeling the layout of all data structures is not necessary. Data structures of minimum impact on the layout of other data structures or the performance, such as local, temporal, or scalar variables, may be ignored.

The memory layout for data structures is used by MDS to generate the memory reference addresses for the data accesses in the application. As we noted in previous chapters, data layout can affect the application performance in terms of shared-memory communication and cache-memory traffic. By modeling the data layout of the application, we can study performance problems related to the data layout via simulating the memory reference pattern in MDS.

Data structures declared in the data layout module need not to be bound with values unless the values in the data structures are to be used by MDS. For example, the values of the **`Force`**, **`Position`**, and **`Velocity`** arrays are not related to the performance; in contrast, the values of **`Num_Elements`**, **`Type`**, **`Num_Neighbors`**, and **`Neighbor`** are needed for controlling the program flow in CRASH, hence their values should be specified. Binding values to these data structures is carried out in the data dependence module discussed in Section 5.2.3.

5.2.2 The Control Flow Module

5.2.2.1 Control Flow Graph

Most compilers decompose a program into a set of *basic blocks*, "each of which is a sequence (zero or more) of instructions with no branch instructions, except perhaps the last instruction, and no branch targets or labels, except perhaps the first instruction," as described by Wolfe [121]. Compilers often use a directed graph, called the *control flow graph* (CFG), to represent the control flow in the program. In a CFG, each vertex represents a basic block, and each edge represents the *potential* control flow between two basic blocks.

While basic blocks are commonly used in compilation, they are usually restricted to represent small code sections. Often, we intend to model the application at a level that treats certain program constructs, such as loops or subroutines, as a unit. We would like to treat these program constructs similarly to the way basic blocks are used in a compiler. Therefore, we use *tasks*, as basic units to specify the control flow in application modeling.

Definition 2. Task: A task is an arbitrarily defined sequence (zero or more) of instructions that contain no branch instructions that jump out of the task, except perhaps the last instruction, and no branch targets that can be hit from branch instructions in other tasks, except perhaps the first instruction. In addition, a task contains no instructions that synchronize with other tasks or force the task to migrate onto another processor, except perhaps the first and/or the last instruction.

A task can be a basic block, a loop, or a subroutine, as long as its execution is neither interrupted by synchronization nor further broken to be executed by multiple processors. This gives us more freedom in defining the program units for studying the application behavior. Once a task is running on a processor, the task cannot be stopped or rescheduled to run on another processor by the program. Using tasks, we can specify the control flow by a CFG where each vertices now represents a task.

Figure 5-5 shows a CFG for CRASH, where four tasks are defined: (1) **`Initialization`**, (2) **`Contact_force`**, (3) **`Loop_contact_neighbor`**, and (4) **`Update_material`**. The code section represented by each task is shown in Figure 5-6. We choose not to model the details within these tasks rather we focus on the interactions among them. The CFG can be enhanced to reflect the actual execution by labeling each edge or vertex with the number of times that it is executed. In this case, tasks **`Contact_force`** and **`Loop_contact_neighbor`** (which constitute the *Contact* phase) are executed **`Num_Elements*Num_iterations`** times and each phase is executed **`Num_Iterations`** times over the entire run, where **`Num_Iterations`** is obtained by profiling the run.

So far, the exact task sequence in the codes traversal of the CFG is not recoverable from the profile counts. For example, Figure 5-5 does not tell us exactly how many times task **`Contact_force`** is executed within one iteration; it shows only total executions. Since the

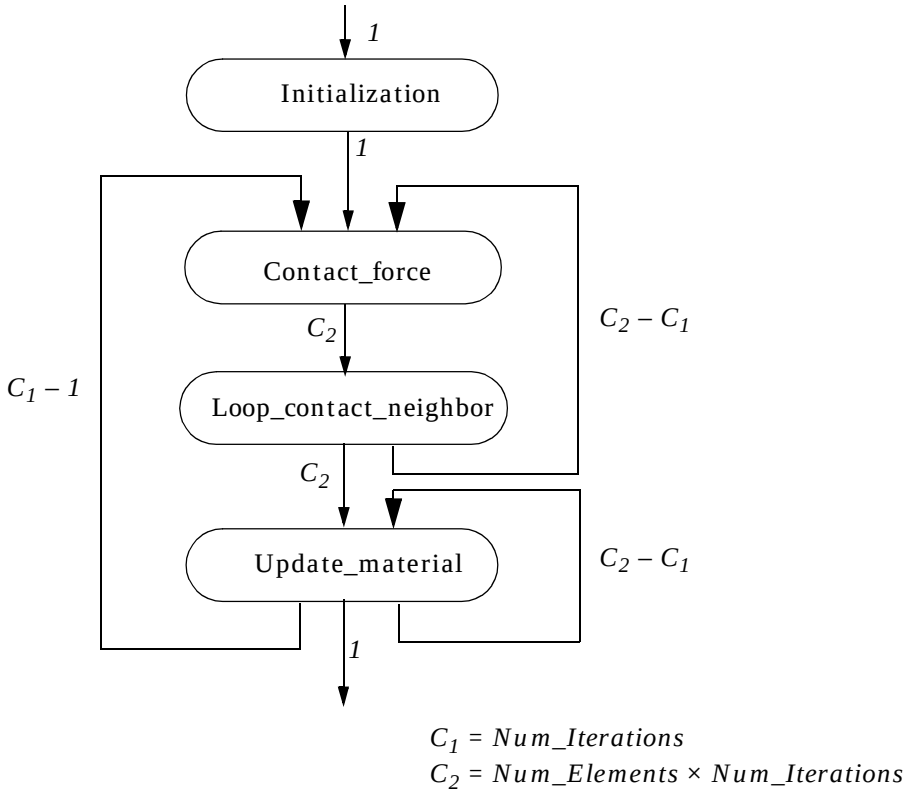


Figure 5-5: A Control Flow Graph for CRASH.

task sequence can be extremely important to performance analysis, we further specify the task sequence by defining the CFG hierarchically, as illustrated in Figure 5-7. This set of CFGs collectively provides the same control information as the CFG shown in Figure 5-5, yet each CFG is now a *directed acyclic graph* (DAG) that contains no multiple-vertex cycles. While decomposing the CFG into a set of DAGs disambiguates some of the control flow, conditional statements, such as IF-THEN-ELSE, are still difficult to model unless trace information is available. For example, task **`Update_material`** can be further divided into two tasks **`Update_plastic`** and **`Update_elastic`** that model the subroutines called in the IF-THEN-ELSE statement. The CFG at this level is shown in Figure 5-8, whose actual flow depends on **`Type(i)`**. We have two ways to address this issue: (1) record the sequence of outcomes of the conditional statement by tracing the application, or (2) model the conditional statement as a task with nonuniform workload and associate the task with a function. We model the IF-THEN-ELSE statement in CRASH as one task, **`Update_material`**, with the second approach, as shown in Figure 5-7. The workload of **`Update_material`** is now a function of **`Type(i)`**. The workload is further defined in the weight distribution module (Section 5.2.5).

```
      program CRASH

      integer Type(Max_Elements), Num_Neighbors (Max_Elements)
      integer Neighbor(Max_Neighbor_per_Elements, Max_Elements)
      real_vector Force(Max_Elements),Position(Max_Elements),
            Velocity(Max_Elements)
      real t, t_step
      integer i,j,type_element

(1)   call Initialization

c Main Simulation Loop
      t=0

c First phase: generate contact forces
100   do i=1,Num_Elements
(2)     Force(i)=Contact_force(Position(i),Velocity(i))

(3)     do j=1,Num_Neighbors(i)
          Force(i)=Force(i)+Propagate_force(Position(i),Velocity(i),
            Position(Neighbor(j,i),Velocity(Neighbor(j,i))
        end do
      end do

c Second phase: update position and velocity
200   do i=1,Num_Elements
(4)     type_element=Type(i)
        if (type_element .eq. plastic) then
          call Update_plastic(i, Position(i), Velocity(i), Force(i))
        else if (type_element .eq. glass) then
          call Update_elastic(i, Position(i), Velocity(i), Force(i))
        end if
      end do

      if (end_condition) stop
      t=t+t_step
      goto 100
      end
```

(1) **Initialization**
(2) **Contact_force**
(3) **Loop_contact_neighbor**
(4) **Update_material**

Figure 5-6: The Tasks Defined for CRASH

5.2.2.2 Control Structures

We use *control structures* to specify the control flow in AML. Figure 5-9 shows a control flow module for CRASH specified by control structures that define the control flow hierarchically. Each control flow hierarchy is defined as a *program construct*, such as program, loop, and complex. The program constructs currently supported in AML are defined below.

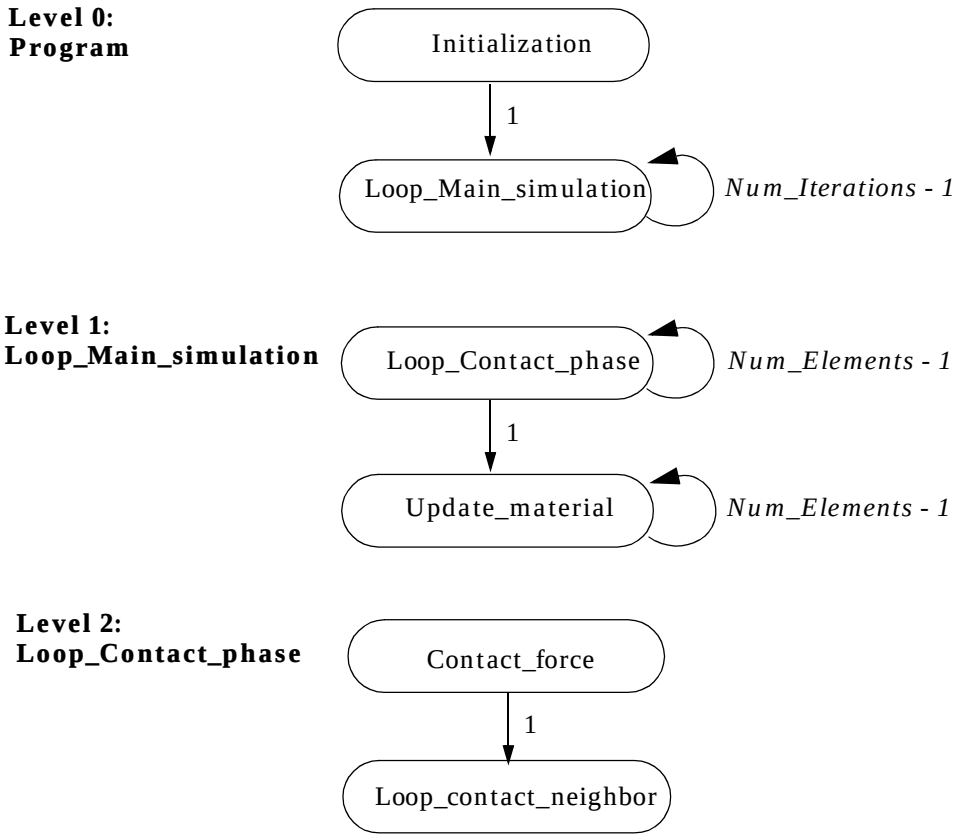


Figure 5-7: A Hierarchical Control Flow Graph for CRASH.

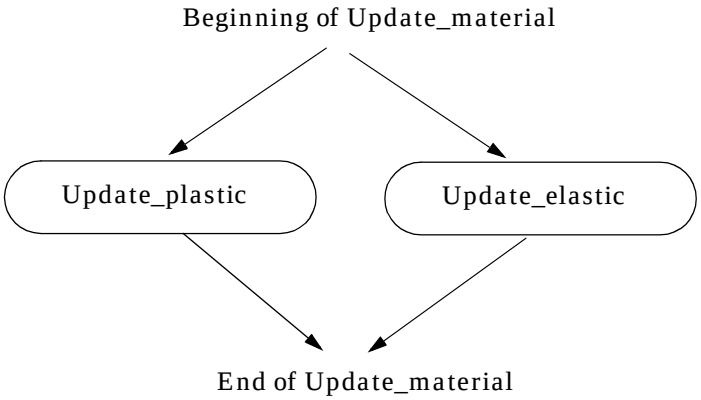


Figure 5-8: An IF-THEN-ELSE Statement Represented in a CFG.

```
Line
1       program CRASH:
2            task Initialization
3            loop Main_simulation;
4
5       loop Main_simulation: Num_Iterations
6            complex Contact_phase
7            complex Update_phase;
8
9       complex Contact_phase:
10           loop Contact_element;
11
12      loop Contact_element: Num_Elements
13           task Contact_force
14           task Loop_contact_neighbor;
15
16      complex Update_phase:
17           loop Update_element;
18
19      loop Update_element: Num_Elements
20           task Update_material;
```

Figure 5-9: A Control Flow Module for CRASH.

Definition 3. Complex: A complex, or complex task, $C=T_1->T_2-> \ldots ->T_N$ is an arbitrarily defined sequence of tasks T_i, $i=1..N$, that contains no branch instructions that jump out of the complex (except perhaps for the last instruction in T_N) no branch targets that can be hit from branch instructions outside the complex (except perhaps for the first instruction in T_1), and T_i must follow T_{i-1}, for $2 \leq i \leq N$.

Definition 4. Loop: A loop $L=C_1->C_2-> \ldots ->C_N$ is a sequence of complexes C_i, $i=1..N$, where each C_i is an instance of the loop body, i.e. C_i, $i=1..N$, all share the same code. C_i contains no branch instructions that jump out of the loop (except perhaps for the last instruction in C_N), and no branch targets that can be hit from branch instructions outside the loop (except perhaps for the first instruction in C_1). The loop count, N, is a non-negative integer, which specifies the number of times that the loop body is executed during runtime.

Note that an irregular loop may contain branches anywhere within the loop or may allow branches to jump into the middle of the loop body. Irregular loops are not accepted by this definition, but can be transformed into regular loops (many compilers do this). Details of such a transformation are beyond the scope of this dissertation. In this chapter, we assume that the loops have been transformed into regular loops, as defined here.

Definition 5. Program: A program is a complex that contains all the tasks executed in the application.

Note that current version of AML does not support conditional statements. This is because (1) scientific/engineering applications are often loop-based, where conditional statements do not frequently appear in high-level control flow, (2) knowing exact flow of conditional statements would require tracing, which adds complexity to the modeling, and (3) we model the conditional statements in our target applications quite well with the nonuniform workload approach mentioned previously.

The tasks or program constructs defined in a hierarchy are executed in order, as if they were in a Fortran or C program. The control flow module in Figure 5-9 should be relatively simple for a Fortran or C programmer to read or write:

- "**Program**" corresponds to the *program* statement in Fortran, which points to the beginning of the program. Lines 1-3 define **program CRASH**, which contains **task Initialization** and **loop Main_simulation**.

- In lines 4-6, **loop Main_simulation** contains two phases, **Contact_phase** and **Update_phase**, each of which is a *complex*, as defined in lines 9 and 16, respectively. In line 5, **Num_Iterations** specifies how many times this loop is executed.

- **complex Contact_phase** and **complex Update_phase** are used here for illustrative purposes. Usually, a complex contains more than one program construct or task.

Figure 5-10 maps the source code of CRASH to the program constructs defined above in Figure 5-9. The tasks are defined in Section 5.2.2.4.

5.2.2.3 Modeling the Control Flow in a SPMD Parallel Execution

In a *Single-Program-Multiple-Data* (SPMD) program, the same code is performed by multiple processors, which carry out their computations on different data subdomains. In our application modeling, we model SPMD parallelization by specifying the domain decomposition that is associated with the tasks.

Figure 5-11 shows a CFG that represents a 2-processor execution of the two DOALL loops in CRASH-SP. Such a CFG can easily be derived from the sequential CFG (Figure 5-7) by replicating the parallelized tasks and associating each instance with a distinct subdomain.

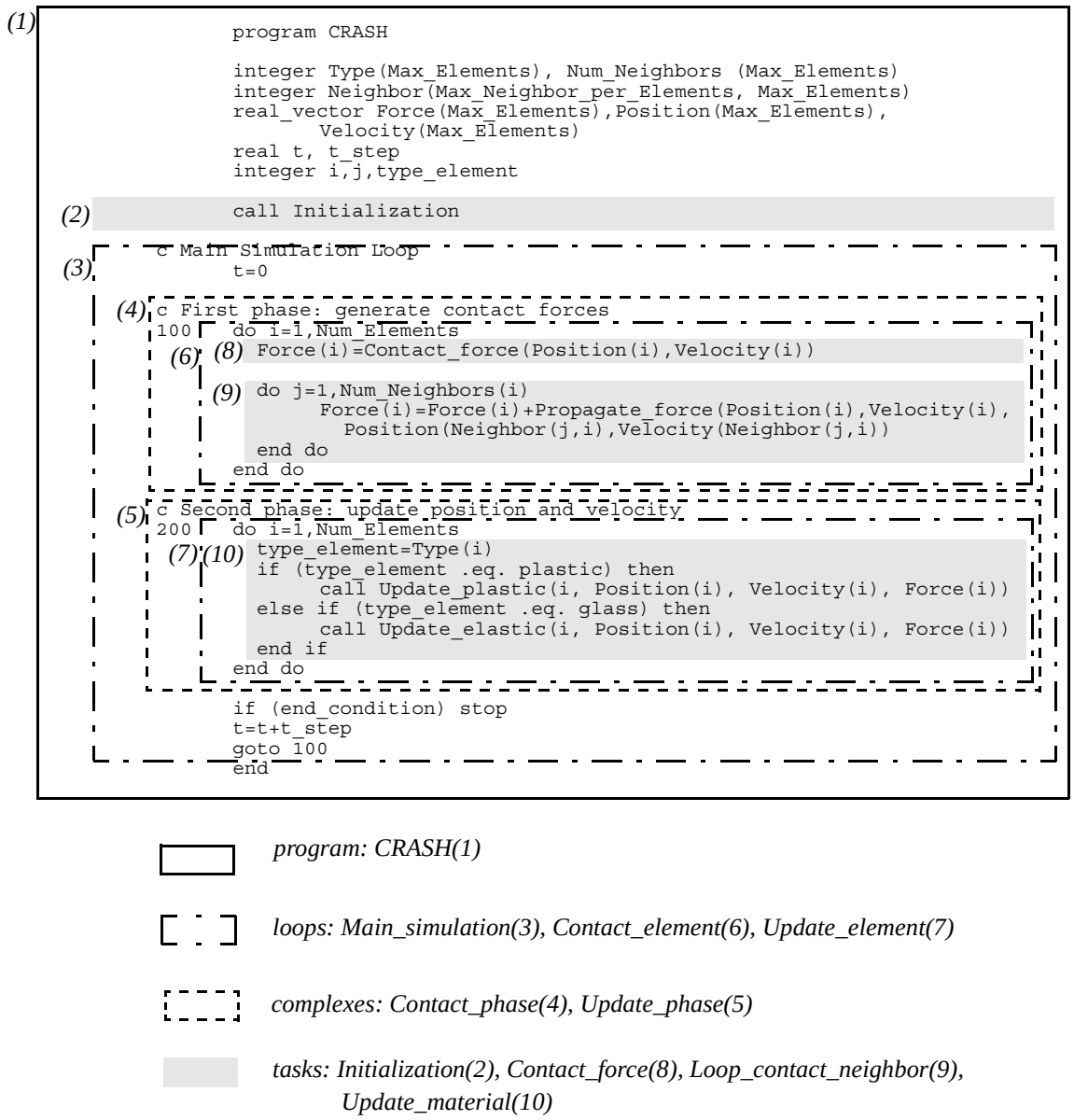


Figure 5-10: Program Constructs and Tasks Modeled for CRASH

Therefore, for a SPMD application, a sequential CFG should be sufficient to specify the control flow regardless of the number of processors that are used in the execution.

Now that we are dealing with sub-tasks that are executed in parallel, we must also model the scheduling of these sub-tasks and the synchronization among them. The scheduling and synchronization shown in Figure 5-11 for DOALL loops are quite common and simple. The control dependencies (arrows) in the CFG represent the scheduling constraints, and the synchronization operations (in this case, the barriers) further enforce the execution order. Sub-

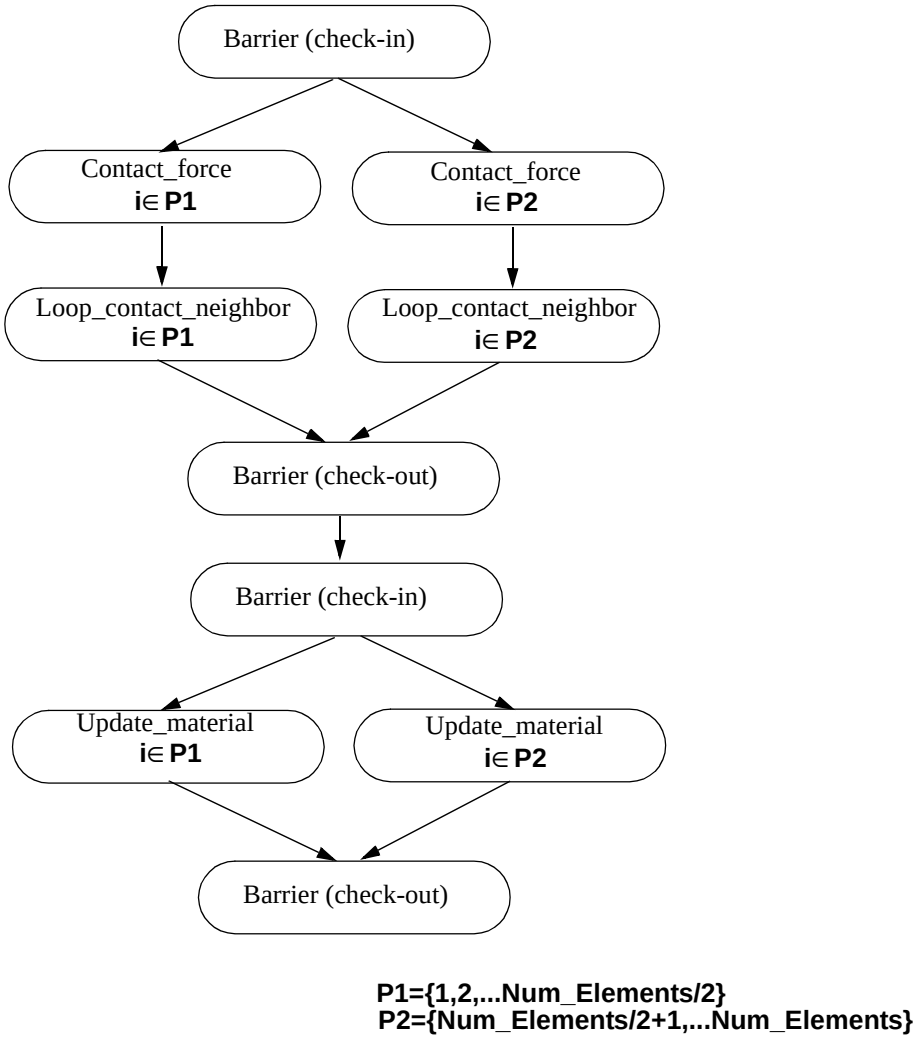


Figure 5-11: A Control Flow Graph for a 2-Processor Parallel Execution in CRASH-SP.

tasks within the DOALL loops, however, can be executed in any order because there is no control dependence or synchronization among those sub-tasks.

In addition to domain decomposition and scheduling/synchronization, each task is associated with a data dependence module and a weight distribution module. The data dependence and weight distribution modules of a task are used in conjunction with its domain decomposition module to specify the input/output data and the computational weight for the sub-tasks generated for that task in a SPMD parallel execution.

```
Line
1       Task Initialization:
2            DD_1
3            Dat_Init
4            Wei_Init
5            Serial;
6
7       Task Contact_force:
8            DD_4
9            Dat_Cforce
10           Wei_Cforce
11           Consecutive(4);
12
13      Task Loop_contact_neighbor:
14           DD_4
15           Dat_Cneighbor
16           Wei_Cneighbor;
17           Consecutive(4);
18
19      Task Update_material:
20           DD_4
21           Dat_Update
22           Wei_Update;
23           Consecutive(4);
```

Figure 5-12: Associating Tasks with Other Modules for Modeling CRASH-SP on 4 Processors.

5.2.2.4 Defining the Tasks

Figure 5-12 shows how we use AML to associate the tasks in CRASH with the domain decomposition, data dependence, weight distribution and scheduling policy modules for modeling the behavior of CRASH-SP executed with 4 processors. Lines 1-5 generally define **`Task Initialization`**: The first argument (line 2) specifies its data domain (**`DD_1`**), the second argument specifies its data dependence (**`Dat_Init`**), the third argument specifies its workload (**`Wei_Init`**), and the fourth argument specifies its scheduling policy (**`Serial`**). These modules are discussed in Sections 5.2.3 through 5.2.5. Before looking into these modules individually, we would like to further discuss the notion of workload decomposition and non-uniform workload.

The *domain decomposition module* decomposes the domain of a task. The way in which most programs decompose the domain is by grouping the loop indices, or associating a function with the loop indices to identify the workload. For example, the DOALL loops in CRASH-

SP divide the loop indices into *N* groups, where *N* is the number of processors. On the other hand, CRASH-SD uses a function (actually an array which serves as a look-up table), **`global_id(ii,d)`**, to identify the **`ii`**-th component of the workload in subdomain **`d`**. The domain decomposition module describes how these subdomains are mapped into the indices.

The workload in a loop can be *uniformly distributed* (*homogenous*) or *non-uniformly distributed* (*heterogeneous*) over the loop indices. Extracting the workload function and utilizing the workload function to calculate the workload of each sub-task is a focus in MDA for studying several important problems, such as load imbalance and irregular communication patterns. A non-uniform workload distribution often results due to a conditional statement (e.g. **`Update_material`**) or variable loop counts (e.g. **`Loop_contact_neighbor`**).

5.2.2.5 Scheduling of Sub-Tasks

A scheduling policy is associated with a task to model the scheduling of the sub-tasks decomposed from that task. Currently, the following scheduling policies are implemented in MDS for scheduling *M* sub-tasks on *N* processors, where the sub-tasks are labeled $T_1..T_M$, processors are labeled *0..N-1*, and *M=k*N* for simplicity:

- *Serial*: all the sub-tasks are assigned to processor 0.
- *Consecutive(N)*: processor *p* is assigned $\{T_{p*k+i} \mid i=1..k\}$.
- *Interleaving(N)*: processor *p* is assigned $\{T_{p+1+i*N} \mid i=0..k-1\}$.
- *List*: the mapping is defined by a *M*-ary list, whose *i*-th element specifies the processor that the *i*-th task is assigned to.

One of the above scheduling policy specifications can be assigned to each task. For example, in Figure 5-12, the sequential task **`Initialization`** is attributed with **`Serial`**, while the DOALL loops (**`Contact_force`**, **`Loop_contact_neighbor`**, and **`Update_material`**) are attributed with **`Consecutive(4)`**. Additional scheduling policies, such as self-scheduling, could easily be incorporated in future AML/MDS versions.

Note that the number of sub-tasks generated for a task depends on the domain decomposition. For example, in the contact phase of CRASH-SD (Figure 3-4), the data domain is explic-

```
Line
1       loop-doall Contact_element: Num_Elements
2            task Contact_force
3            task Loop_contact_neighbor;
4
5       loop-doall Update_element: Num_Elements
6            task Update_material;
```

(a) Using DOALL Statements

```
Line
1       loop Contact_element: Num_Elements
2            barrier-check-in;
3            task Contact_force
4            task Loop_contact_neighbor;
5            barrier-check-out;
6
7       loop Update_element: Num_Elements
8            barrier-check-in;
9            task Update_material;
10           barrier-check-out;
```

(b) Using Barrier Statements

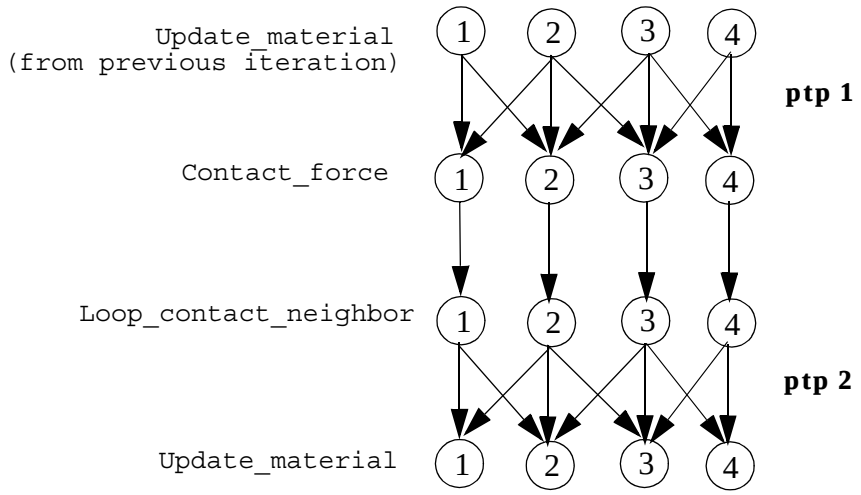
Figure 5-13: Modeling the Synchronization for CRASH-SP.

itly decomposed into **Num_Subdomains** subdomains, which maps the **Num_Elements** loop index values into **Num_Subdomains** sub-tasks, and then a **DOALL** statement schedules these sub-tasks on the processors. Another way to model this scheduling process is to list the processor assignment for each iteration in the DOALL loop, without grouping the iterations in the domain decomposition. These two ways should produce the same results in performance modeling, so it is up to the users to choose which way they wish to model their applications.

5.2.2.6 Synchronization

Three mechanisms are provided in AML to model the synchronization schemes that are most commonly found in a parallel application: (1) DOALL, (2) barriers, and (3) point-to-point synchronization. Since a DOALL loop implicitly uses barriers at the beginning and the end, the synchronization invoked in the DOALL loops in CRASH-SP can be modeled in AML with DOALL statements or barriers as shown in Figure 5-13(a) and (b), respectively.

To model point-to-point synchronization, the user specifies the pairs of sub-tasks that require synchronization. Our example in Figure 5-14(b) shows the use of point-to-point syn-



(a) Control Dependencies

```
Line
1       loop Contact_element: Num_Elements
2            ptp 1;
3            task Contact_force
4            task Loop_contact_neighbor;
5
6       loop Update_element: Num_Elements
7            ptp 2;
8            task Update_material;
9
10      ptp 1: task Contact_force <- task Update_material
11           (1,1),(1,2),
12           (2,1),(2,2),(2,3),
13           (3,2),(3,3),(3,4),
14           (4,3),(4,4);
15
16      ptp 2: task Update_material <- task Loop_contact_neighbor
17           (1,1),(1,2),
18           (2,1),(2,2),(2,3),
19           (3,2),(3,3),(3,4),
20           (4,3),(4,4);
```

(b) Using Point-to-point Synchronization Statements

Figure 5-14: Modeling the Point-to-Point Synchronization, an Example.

chronization to model the control flow in Figure 5-14(a). Point-to-point synchronization operations, **ptp 1** and **ptp 2**, are placed in lines 2 and 7, respectively, where synchronization is required. **ptp 1**, defined in lines 10 to 14, synchronizes the sub-tasks in **Contact_force** with the prior sub-tasks in **Update_material**. Although a synchronization point can appear at either the beginning or the end of a sub-task, a synchronization pair *(u,v)* synchronizes the beginning of *u* with the end of *v*.

5.2.3 The Data Dependence Module

The data dependence module specifies the data dependence of each elementary component of the workload (e.g. the i-the iteration of a loop) by listing the input and output data elements of that component. For example, to perform **task Contact_force** for a particular index value **i**, **Position(i)** and **Velocity(i)** are read and **Force(i)** is written. To express such data dependence, we can use the following statement in AML:

```
Force(i) <- Position(i), Velocity(i);
```

The output is expressed on the left-hand side of the arrow, and the input is expressed on the right-hand side. There may be multiple variables listed on each side.

Modeling the data dependence for tasks with a heterogeneous workload is more complicated, yet their data dependence functions can be often expressed quite straightforwardly using AML, as long as the data dependence function is known. For example, the data dependence function for **task Loop_contact_neighbor** can be expressed by the following statement:

```
Force(i) <- Force(i), Position(i), Velocity(i),
      Position(Neighbor(1..Num_Neighbors(i),i),
      Velocity(Neighbor(1..Num_Neighbors(i),i);
```

In Section 5.2.1, we mentioned that some data structures can be bound with values. Binding values to variables can be carried out by binding statements in a data dependence module, such as:

```
Num_Neighbors(1) = 3;
Num_Neighbors(2) = 4;
...
```

```
Dat_Init:
     Num_Elements = 100000; Plastic=1; Elastic=2;
     Num_Neighbors(1) = 3;
     Num_Neighbors(2) = 4;
     ... (more value binding statements omitted)

Dat_Cforce:
     Force(i) <- Position(i), Velocity(i);

Dat_Cneighbor:
     Force(i) <- Force(i), Position(i), Velocity(i),
           Position(Neighbor(1..Num_Neighbors(i),i),
           Velocity(Neighbor(1..Num_Neighbors(i),i);

Dat_Update:
     Position(i), Velocity(i) <- Position(i),
           Velocity(i), Force(i);
```

Figure 5-15: A Data Dependence Module for CRASH.

which are used in the data dependence module of **task Initialization**. When **task Initialization** is referenced in MDS, these binding statements are carried out.

To summarize the above discussion, we show an example data dependence module for CRASH in Figure 5-15. Note that the same module applies to CRASH-SP or CRASH-SD since the parallelization in those two codes does not affect the data dependence.

The user can choose to provide a more precise data access sequence by defining the tasks in a more fine-grain fashion. For example, the functions **Contact_force()**, **Propagate_force()**, **Update_Plastic()**, and **Update_Elastic()** may all be modeled in further detail. More precise modeling may reveal some details in the data access pattern, such as multiple accesses to one data item. For example, **Position(i)** may be referenced multiple times for computing **Force(i)** in **Contact_force()**. However, while more detailed modeling can produce more accurate results, we also think that unnecessary information should be filtered out to reduce the complexity of model-driven analysis. The following few guidelines are suggested:

- *Local* or *read-only* variables which are of no interest in data flow analysis should be removed from the data dependence module. Generally, index variables are not modeled because they usually reside in registers. Read-only data may be removed from the module for data flow analysis, yet they can affect working set analysis.

- Detailed modeling of the data access sequence can be critical for analyzing the performance of the processor cache. A precise data access sequence is useful for exposing the conflict and capacity misses, while a less precise sequence may be sufficient for analyzing the data flow and working set.

- As discussed in Section 2.4.4, the amount of optional communications (e.g. false sharing) for one shared-memory block depends on the number of accesses and the access sequence to the block. Thus, modeling the data access sequence precisely helps MDS estimate optional communications more accurately, yet the required communication as well as the locations that cause false sharing can be identified with much less a precise sequence.

5.2.4 The Domain Decomposition Module

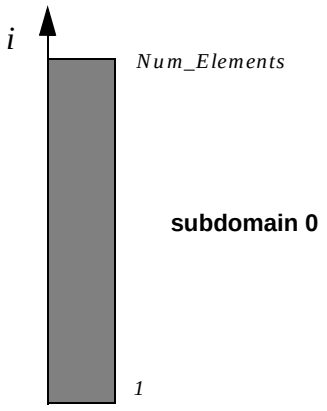
For a task to be processed efficiently by N processors in parallel, it needs to be decomposed into at least N sub-tasks. For a task that consists of M ($M \geq N$) elementary components of independent workload, the decomposition amounts to *grouping* these components into N sub-tasks. In practice, this grouping is typically performed in one of two ways:

1. by partitioning the code (e.g. the iteration space in an SPMD code) into N sub-tasks, and then deciding which elements of the data domain are to be associated with each sub-task (i.e. which data are local and which require a remote reference if needed), or

2. by partitioning the data domain into N subdomains using some domain decomposition algorithm, and then associating each workload component of the code with one subdomain (usually with an “owner updates” rule which requires that a sub-task must be associated with the subdomain that contains the elements that it updates).

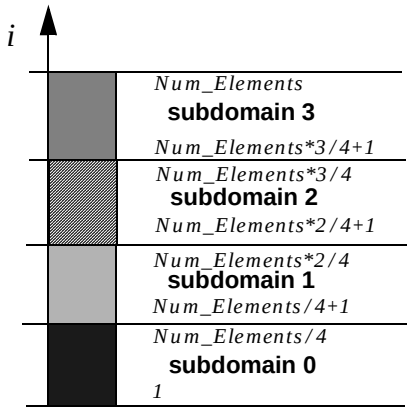
Sometimes, we overdecompose the domain to create more than N sub-tasks so that the processors have more freedom in scheduling the sub-tasks. In some cases, when the parallelism is insufficient ($M < N$), or limited parallelization is preferred, less than N sub-tasks (subdomains) are created and some processors are simply left idle.

```
Decomposition DD_1
 Number_of_Subdomains 1
 Subdomain 0
  i 1:Num_Elements
end Decomposition
```



Decomposition DD_1

```
Decomposition DD_4
 Number_of_Subdomains 4
 Subdomain 0
  i 1:Num_Elements/4
 Subdomain 1
  i Num_Elements/4+1:Num_Elements*2/4
 Subdomain 2
  i Num_Elements*2/4+1:Num_Elements*3/4
 Subdomain 3
  i Num_Elements*3/4+1:Num_Elements
end Decomposition
```



Decomposition DD_4

Figure 5-16: An Example Domain Decomposition Module for CRASH-SP.

Figure 5-16 shows two decomposition schemes for CRASH-SP: **`DD_1`** and **`DD_4`**. **`DD_1`** represents the domain decomposition in a sequential region which assigns all the loop iteration space to subdomain 0. **`DD_4`** decomposes the domain into four disjoint, equally sized subdomains by partitioning **`i`** in a linear consecutive fashion.

Several common decomposition schemes for partitioning a one-dimensional domain can be modeled in AML, including *consecutive*, *interleave*, and *list*, as illustrated in Figure 5-17. These decomposition schemes are defined below, for decomposing a one-dimensional space *1..M* to subdomains *0..N-1* (assuming *M=k*N* for simplicity):

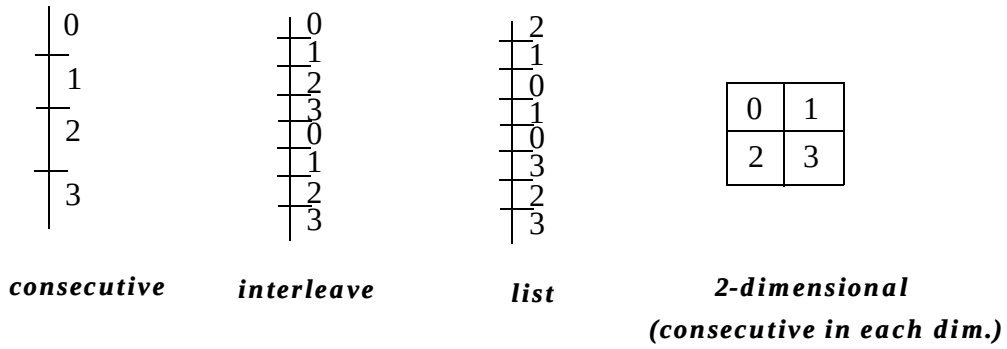


Figure 5-17: Domain Decomposition Schemes Supported in MDS.

```
Decomposition DD_4
 Number_of_Subdomains 4
 Subdomain 0
  i {1,2,3,6}
 Subdomain 1
  i {4,5,7,8}
 Subdomain 2
  i {9,10,12,14}
 Subdomain 3
  i {11,13,15,16}
end Decomposition
```

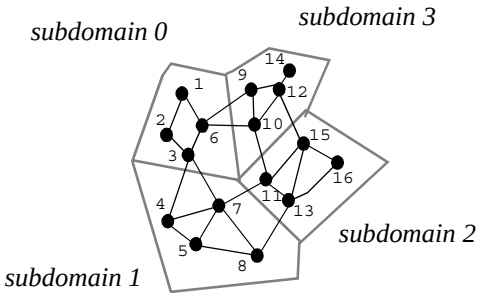


Figure 5-18: An Example Domain Decomposition for CRASH-SD.

- *Consecutive (default)*: subdomain *j* is assigned {*j*k+i* | *i=1..k*}.
- *Interleave*: subdomain *j* is assigned {*j+1+i*N* | *i=0..k-1*}.
- *List*: the mapping is specified by exhaustively listing the elements that are assigned to each subdomain.

Multiple decomposition schemes can be combined to decompose a multi-dimensional domain. A list decomposition scheme is used to model the domain decomposition in CRASH-SD using the information generated by the domain decomposition algorithm, as illustrated in Figure 5-18. This list is generated and used similarly to array **`global(ii,d)`** in CRASH-SD.

```
Wei_Init:
     weight 7.3 ms;

Wei_Cforce:
     for each i {weight 4.075 us};

Wei_Cneighbor:
     for each i {weight Num_Neighbors(i)*1.398 us};

Wei_Update:
     for each i {weight $WC(Type(i))};

$WC(Plastic)=5.7 us;
$WC(Elastic)=7.7 us;
```

Figure 5-19: An Example Workload Module for CRASH/CRASH-SP/CRASH-SD.

5.2.5 The Weight Distribution Module

The weight for a task is normally defined as the *time* required for performing the computation in the task. The weight distribution module of a task defines the weight distribution over the domain for the task.

Figure 5-19 shows a weight distribution module for CRASH/CRASH-SP/CRASH-SD, where we specify the weight distribution function for each task. For **`task Initialization`**, we treat the task as one unit and assign it a weight of 7.3 ms (7.3e-3 seconds). Since this is a serial task, we simply specify its execution time. For parallel tasks, we specify the weight for each instance of the loop index **`i`**.

Direct measurement of the execution time for one iteration of a loop can be difficult, since the loop body can be relatively small and the instrumentation required to time the loop body individually can be very intrusive to the execution. Most profiling tools report the total execution time for the loop, but not for individual iterations. In case individual iterations are difficult to measure, a first-order-approximation can be made by averaging the total execution time. For example, assuming that the load is uniformly distributed in this case, we use the average execution time, 4.075 μ*s* (4.075e-6 seconds) per iteration, to specify the weight for **`task Contact_force`** in **`Wei_Cforce`**.

For loops where load is non-uniformly distributed, estimating the weights may require some heuristics. While source code analysis may not accurately predict the weight, it can be useful for classifying the workload. For example, for each **i**, the workload of **task Update_material** depends on **Type(i)**. It is logical to assume that there are two weight classes, since there are two types of material in this example application. These two weight classes are defined as **$WC(Plastic)** and **$WC(Elastic)** in the weight distribution module. After the weight classes are identified, systematic methods can be used to estimate the weights for performing the task for each type of material by profiling the application with several sets of input data, as demonstrated in [30]. Here, **$WC(Plastic)** and **$WC(Elastic)** are estimated to be 5.7 and 7.7 μs, respectively.

It is a reasonable guess that the workload of **task Loop_contact_neighbor** roughly depends on the number of neighbors that the element interacts with. Assuming that the total time of executing **task Loop_contact_neighbor** for all elements is 1.398 seconds, and the total number of neighbors is 1,000,000, the average time required to perform the calculation for one neighbor is 1.398/1,000,000 = 1.398 μs. Therefore, the weight function in **Wei_Cneighbor** is specified as **Num_Neighbors(i)*1398**.

5.2.6 Summary of Application Modeling

In this section, we have discussed how to generate modules using AML to model an application. Although these modules are related, one module can often be independently generated or modified without affecting the other modules. This is important for meeting our goal of accepting results from different sources of performance assessment. We also believe that separated modules are easier to access, and helps facilitate our development of model-building and model-driven analysis tools.

We would like to point out that the application model can be very fine detailed, with the tasks defined at basic block level or even instruction level. However, overly detailed models are overkill for performance modeling and add cost to MDA. Therefore, in some cases, users may consider merging tasks, such as merging **task Contact_force** and **task Loop_contact_neighbor** in the above examples. The level of abstraction is a subjective choice of the user. Our application models illustrated in this section are abstracted at a relatively high level, yet this level of abstraction suffices to reveal important performance problems and investigate performance tuning in the manner developed in the previous chapters.

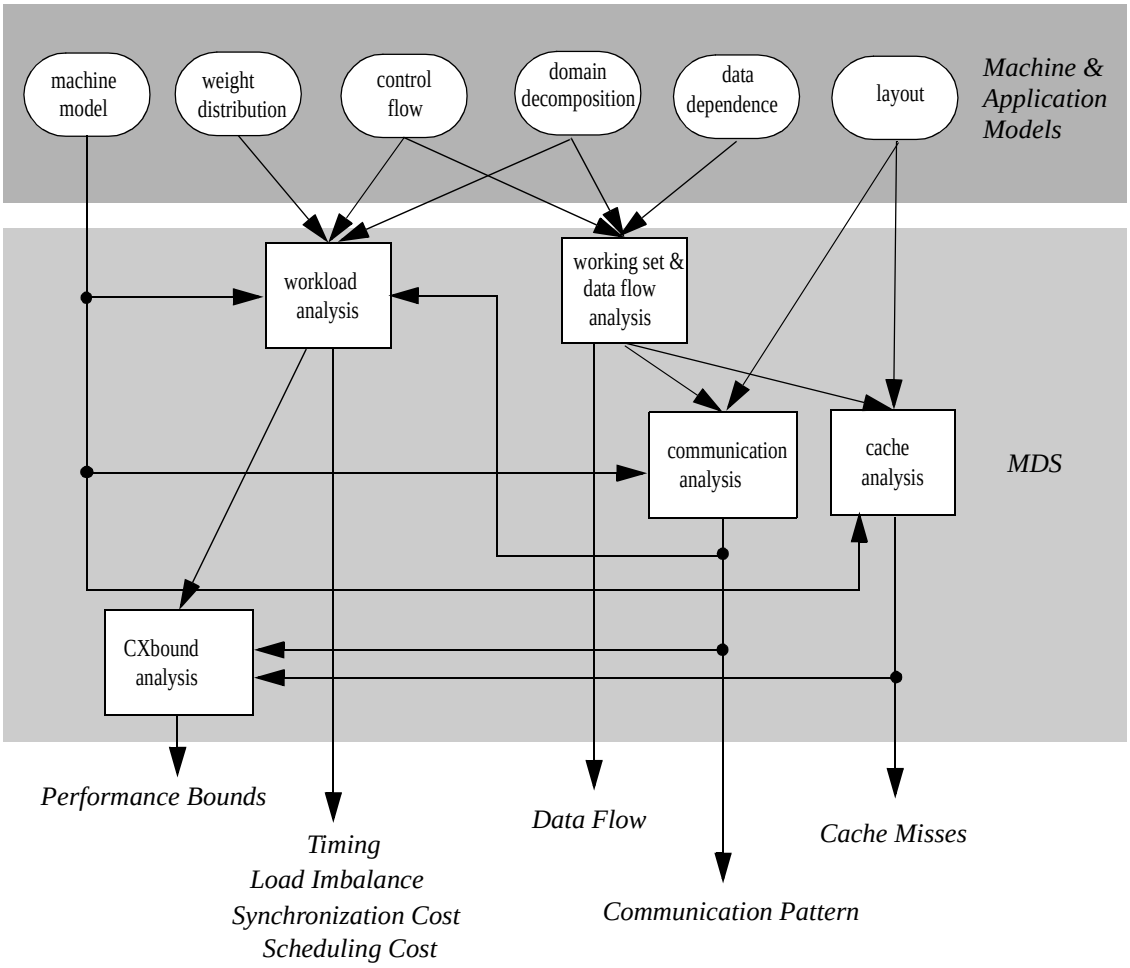


Figure 5-20: Model-driven Analyses Performed in MDS.

5.3 Model-Driven Performance Analysis

MDS is a performance analyzer that derives performance information for an application by simulating the application's model with a machine model. MDS is a model-driven simulator that *executes* the tasks in the application model as if executing a Fortran or C program. Figure 5-20 shows the performance analyses that are carried out in MDS. Most of the techniques used in these analyses have been discussed in Chapter 2, because we selectively ported a few performance assessment techniques from K-LCache, CIAT/CDAT, CXpa, and CXbound into MDS. Therefore, the details of MDS are not discussed in this section. In this section, we present a machine model and explain the generation scheme of MDS.

```
Processor_clock_cycle: 1e-8
Processor_data_cache_size: 1e6
Processor_data_cache_line_size: 32
Processor_data_cache_associativity: 1
Processor_data_cache_access_latency: 1e-8
Processor_instruction_cache_size: 1e6
Processor_instruction_cache_line_size: 32
Processor_instruction_cache_associativity: 1
Processor_instruction_cache_access_latency: 1e-8
Number_processors_per_hypernode: 8
Number_processors: 32
Local_memory_line_size: 64
Local_memory_read_access_latency: 5.54e-7
Local_memory_write_access_latency: 6.33e-7
Remote_memory_access_latency: 2e-6
Local_memory_size: 578e6
Global_memory_size: 272e6
CTI_cache_size: 166e6
```

Figure 5-21: An Example Machine Description of HP/Convex SPP-1000 for MDS.

5.3.1 Machine Models

A machine model is needed by MDS to carry out machine-dependent analyses. MDS accepts a fairly simplified machine description as its the machine model input. For example, HP/Convex SPP-1000 is described as in Figure 5-21. Some minor changes in the processor cache and the cache coherence protocol are made to model to the SPP-1600 (see Section 1.1.1). This machine model incorporates the information that is collected from the machine specifications and the microbenchmarking results (see Tables 2-1, 2-2 and 2-3, Section 2.1).

MDS also accepts the machine configuration files for the CDAT trace-driven shared-memory machine simulation tool [38], but MDS only uses a subset of the CDAT machine configuration parameters.

5.3.2 The Model-Driven Simulator

MDS follows the flow defined in the control flow module. When a task is executed, MDS performs the operations required by the task according to the modules associated with the task. MDS handles a task according to the following steps:

1. The *domain decomposition* module (Section 5.2.4) is used to group the iteration space into sub-domains. The workload for one subdomain forms a sub-task.

2. The *scheduling policy attribute* (part of the control flow, see Section 5.2.2.5) of the task is used to map each sub-task to one processor, say P_i, which is responsible for executing the sub-task.

3. Find the sub-task's *data dependence* statement in its data dependence module (Section 5.2.3) and mark the data read and/or written in the sub-task. The user can configure MDS to perform the following inherent data analyses:

   - *Working set analysis*: MDS calculates the volume of data accessed in the task.

   - *Data flow analysis*: MDS records a *Read-after-Write (RAW)* transaction if the subtask reads a data item which was a written by a previous sub-task.

4. The user can configure MDS to analyze the data accesses with memory addresses generated using the (sub)task's data layout module. Using the addresses, MDS can perform the following functions:

   - *Memory reference trace generation*: MDS outputs the addresses and the types of the data references in the task to the trace file associated with P_i.

   - C*oherence communication analysis*: a shared-memory simulation is carried out to identify the memory references that would cause interprocessor communications under the infinite-cache assumption, as discussed in Section 2.4.

   - C*ommunication latency analysis*: the communication latency required for the (sub)task is estimated based on the distance and type of communications, as characterized in the machine model.

5. The *weight distribution* module (Section 5.2.5) is used to calculate the *computational weight* for the sub-task.

6. The execution time counter for P_i, denoted $T_{exec}(P_i)$ is updated by adding the computational weight and the communication latency of the (sub)task to the previous $T_{exec}(P_i)$.

When a synchronization point is reached, MDS finishes executing all the (sub)tasks that are prior to the synchronization and may selectively execute independent (sub)tasks if a

relaxed synchronization is used. At the synchronization point, MDS calculates a few time stamps for the synchronization:

1. For each processor, P_i, in the group of processors, $\{P_0, P_1,..P_{N-1}\}$, that are involved in the synchronization, the time when P_i left its last synchronization point is denoted as $T_{last\_sync}(P_i)$.

2. The *busy time* for P_i, between this synchronization point and the last synchronization on P_i, is calculated by subtracting $T_{last\_sync}(P_i)$ from $T_{exec}(P_i)$, i.e. $T_{busy}(P_i) = T_{exec}(P_i) - T_{last\_sync}(P_i)$.

3. The time that the last processor in $\{P_0, P_1,...,P_{N-1}\}$ enters the synchronization, denoted as T_{enter_sync}, is calculated as $T_{enter\_sync} = Max\{T_{exec}(P_i) \mid i=0..N-1\}$.

4. The *idle time* (*synchronization wait time*, as defined in Section 3.6) for P_i is estimated by subtracting $T_{exec}(P_i)$ from T_{enter_sync}, i.e. $T_{idle}(P_i) = T_{enter\_sync} - T_{exec}(P_i)$.

5. The time that the synchronization is ended is estimated by adding the *synchronization cost* (as defined in Section 3.6, denoted as T_{sync_cost}) to T_{enter_sync}, and this time is used to update $T_{last\_sync}(P_i)$, i.e. $T_{last\_sync}(P_i) = T_{enter\_sync} + T_{sync\_cost}$.

6. For a parallel region, i.e. a region between two barriers, MDS computes the *load imbalance* by using the equation in (EQ 2), i.e. $Load\ Imbalance = (T_{max}-T_{avg})/T_{avg}$, where $T_{max} = Max\{T_{busy}(P_i) \mid i=0..N-1\}$ and $T_{avg} = \Sigma\{T_{busy}(P_i) \mid i=0..N-1\}/N$.

MDS can generate a *profile* of the simulated performance by recording the computation time, communication event counts and latency for each task, complex, or loop on each processor, as well as the load imbalance for each parallel region. Based on this profile, MDS calculates the *parallel hierarchical performance bounds* for the application, using the same methodology that has been implemented in CXbound (Section 4.3).

5.3.3 Current Limitations of MDS

MDS is still in a preliminary development stage. While it is now capable of generating a wide range of performance metrics, there is more work that can be done to improve the functionality of MDS. In particular, MDS does not currently model the processor at all and does not model the memory system in detail. In this section, we discuss how those features can limit the use of MDS and how they could be modeled in the future.

5.3.3.1 Modeling the Processor

In our current performance modeling scheme, a processor model is not required because the computation time (weight) is extracted from the application run by profiling. While this approach provides an accurate and simple mechanism to extract the computation time, it also results in two limitations of the model-driven analysis:

1. *Less portability*: Since profiling is required for modeling the weight distribution of an application, accurate simulation of application performance for an arbitrary machine may not be possible.

2. *Inability to address detailed processor performance issues*: Since instructions are not modeled, MDS cannot analyze the instruction scheduling and possible overlapping of memory access latency and instruction execution.

It is possible to model the computation workload of a task with the task's instructions extracted from its source or assembly code. MDS can be enhanced with a processor module that estimates a task's run time by simulating its instructions. However, extracting and simulating instructions would add considerable cost and complexity to the existing performance modeling scheme.

5.3.3.2 Modeling the Processor Cache and Shared-Memory Network

For simplicity, the current MDS assumes that each processor has an infinite cache. In addition, MDS assumes that contentions do not occur in accessing the shared memory. Consequently, MDS tends to report communication overhead optimistically.

Currently, MDS can generate memory reference traces for the user to further analyze the behavior of the processor caches and the shared-memory network with other trace-driven simulation tools. However, trace generation increases the run time of the performance modeling. To improve the efficiency of modeling cache and shared-memory, we plan to integrate mlCache and CDAT into a future version of MDS.

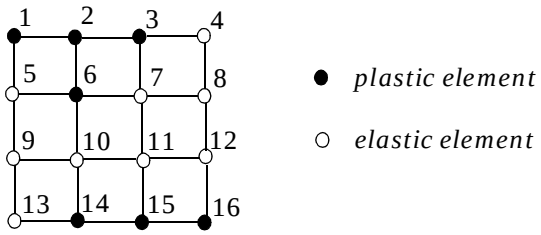


Figure 5-22: A Sample Input for CRASH.

5.3.3.3 Serialized Memory Access Patterns

MDS currently estimates the communications in a parallel region based on a serialized memory access pattern. When each parallel region is simulated, MDS finishes simulating the memory accesses for one processor before it starts to simulate the memory accesses for the next processor. In this fashion, MDS captures the required communications and some of the optional communications in the execution (required and optional communications are defined in Section 2.4.4).

In an undisturbed parallel execution, the memory accesses from different processors are most likely to be interleaved. Therefore, interleaving the memory accesses from different processors may form more realistic memory access patterns for MDS, which is relevant for accurately simulating the effect of sharing.

5.4 A Preliminary Case Study

In this section, we illustrate the concept of model-driven analysis by using AML and MDS to analyze the performance of different versions of CRASH running on the HP/Convex SPP-1600 using four processors. We show that the use of models in our model-driven approach facilitates the selection, the application, and the evaluation of a series of code modifications. Head-to-head comparisons with CXpa shows that MDS yields a more comprehensive and deterministic performance evaluation that is more useful for performance tuning.

For simplicity of discussion, the input is a 2-D structured mesh that consists of 16 elements (**`Num_Elements=16`**), as shown in Figure 5-22. The main simulation loop is executed

10,000 times in each run (`Num_Iterations=10000`). Interestingly, using such a simple input does not reduce the performance problems of CRASH; on the contrary, it raises several problems, such as imprecise CXpa profiling and synchronization costs, that may be less obvious or less significant in a CRASH run with a much larger input. On the other hand, some opportunities for small improvements highlighted by MDS analysis of this example may result in much larger improvements when running on more processors with larger data sets.

In Section 5.4.1, we describe the modeling of three basic versions of CRASH, CRASH (referred to as *CRASH-Serial* hereafter), CRASH-SP, and CRASH-SD. In Section 5.4.2, we characterize the performance of these three versions using MDS and compare the results with their CXpa profiles. In Section 5.4.3, CRASH is further tuned by improving its data layout, fixing workload-thread assignment, and reducing the number of synchronization barriers. Section 5.4.4 summarizes the case study.

5.4.1 Modeling CRASH-Serial, CRASH-SP, and CRASH-SD

In Section 5.2, we have already shown the modules that we use to model CRASH-Serial, i.e. the data layout (Figure 5-4), control flow (Figure 5-9), data dependence (Figure 5-15), and weight distribution (Figure 5-19) modules. Since CRASH-Serial is a serial code, its domain decomposition module has only one subdomain consisting of the entire domain. Besides the weight distribution module, each of the above modules can be created manually in a straightforward fashion by a programmer who is familiar with the code.

The generation of the weight distribution module requires profiling the run time of CRASH-Serial. In Section 5.4.1.1, we describe how we use CXpa profiles to estimate the weight distribution for the tasks in CRASH-Serial.

Domain decomposition is what essentially distinguishes CRASH-Serial, CRASH-SP, and CRASH-SD. In Section 5.4.1.2, we discuss the domain decomposition schemes in these codes.

5.4.1.1 Acquiring the Weight Distribution Functions for CRASH-Serial

To acquire the weight function for a task, the run time for the task needs to be measured. We use CXpa to profile the run time for three major parts of the code, `initialization`,

Program Region	Execution Count	CPU Time (sec.)	% of Total CPU Time
Initialization	1	0.00731	0.3
Contact_phase	10000	1.323	54.6
Update_phase	10000	1.092	45.1
Total	N/A	2.422	100

Table 5-1: A Run Time Profile for CRASH

contact_phase, and **update_phase**, as shown in Table 5-1. The run time dilation due to the CXpa instrumentation is about 0.3 seconds (11%), primarily because the **contact_phase** and **update_phase** are iterated and profiled 10,000 times. The execution time for each iteration is relatively short with the small input data set used in this case study.

Further profiling the routines within the "**do i=1,Num_Elements"** loops yields excessive run time dilation. Consequently, we cannot directly extract the weight for each individual task within the phases. Instead, we estimate the weights using the following heuristics:

1. To separate the run time between **task Contact_force** and **task Loop_contact_neighbor**, we modify the code to skip the execution of **task Loop_contact_neighbor**. The profiled run time of the **contact_phase** for this modified code is 0.652 seconds, which should be approximately the run time for **task Contact_force**, and the run time of **task Loop_contact_neighbor** is approximately 1.323 minus 0.652, which is 0.671 seconds.

2. Assuming that the weight in **task Contact_force** is uniformly distributed over the elements, the weight for carrying out **task Contact_force** per element per iteration is approximately 0.652/16/10000 seconds = *4.075* μs, i.e. **Wei_Cforce=4.075 us**[1].

3. Assuming that the weight for carrying out **task Loop_contact_neighbor** for element *i* is proportional to **Num_neighbors(i)**, the weight for carrying out **task Loop_contact_neighbor** per neighbor per iteration is approximately

1. We use "us" instead of μs to refer to microsecond in the weight distribution modules.

0.671/(Σ*Num_neighbors(i),i=1..16)/10000 seconds = 1.398* μ*s,*

where *(*Σ*Num_neighbors(i),i=1..16) = 48.*

That is, **`Wei_Cneighbor=1.398 us*Num_neighbors(i)`**.

4. To identify the weights for **`Update_Plastic()`** and **`Update_Elastic()`**, we run the code with two sets of inputs. The elements of *input-1* are all plastic, and the elements of *input-2* are all elastic. By profiling the run time of CRASH-Serial with these two inputs, we obtain the run time of **`update_phase`**: 0.912 and 1.232 seconds for *input-1* and *input-2*, respectively. Thus, the weight for carrying out **`Update_Plastic()`** per element per iteration is approximately 0.912/16/10000 seconds = 5.7 μs, and the weight for carrying out **`Update_Elastic()`** per element per iteration is approximately 1.232/16/10000 seconds = 7.7 μs. That is,

```
Wei_Update=5.7 us if Type(i)=Plastic;
or Wei_Update=7.7 us if Type(i)=Elastic.
```

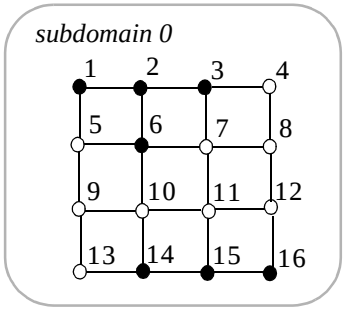
The above results were implemented in the weight distribution module (Figure 5-19) illustrated in Section 5.2.5.

5.4.1.2 Domain Decomposition Schemes for CRASH-Serial, CRASH-SP, and CRASH-SD

Since CRASH-Serial is a serial code, its domain decomposition module has only one subdomain consisting of the entire domain, as shown in Figure 5-23(a). To be executed on four processors, the domains of CRASH-SP and CRASH-SD are each partitioned into 4 subdomains using decomposition scheme DD_4A and DD_4B respectively, as shown in Figure 5-23(b) and Figure 5-23(c).

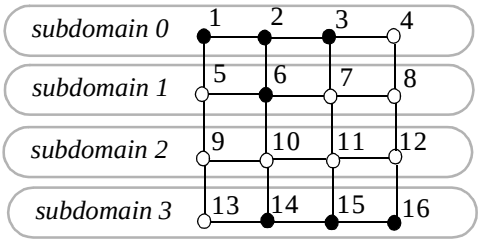
Each processor will be scheduled to execute the computation for one subdomain. For better performance, the processor-subdomain mapping should remain fixed during the entire run (i.e. 10,000 iterations), in order to minimize unnecessary communication overhead due to subdomain migration (see Action 3 in Section 3.3.1). Presumably, processor 0 executes subdomain 0, processor 1 executes subdomain 1, and so on.

DD_4A partitions the domain in a linear consecutive fashion, which is the default option for most DOALL loops. The cut edges generate interprocessor communications. For example,



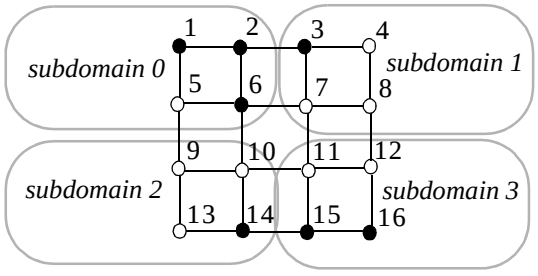

```
Decomposition DD_1
 Number_of_Subdomains 1
 Subdomain 0
  i 1:16
end Decomposition
```

(a) Domain Decomposition Scheme DD_1 (CRASH-Serial)




```
Decomposition DD_4A
 Number_of_Subdomains 4
 Subdomain 0
  i 1:4
 Subdomain 1
  i 5:8
 Subdomain 2
  i 9:12
 Subdomain 3
  i 13:16
end Decomposition
```

(b) Domain Decomposition Scheme DD_4A (CRASH-SP)




```
Decomposition DD_4B
 Number_of_Subdomains 4
 Subdomain 0
  i {1,2,5,6}
 Subdomain 1
  i {3,4,7,8}
 Subdomain 2
  i {9,10,13,14}
 Subdomain 3
  i {11,12,15,16}
end Decomposition
```

(c) Domain Decomposition Scheme DD_4B (CRASH-SD)

Figure 5-23: Decomposition Scheme Used in CRASH-Serial, SP, and SD.

edges {(1,5),(2,6),(3,7),(4,8)} would generate communications between processors 0 and 1. Processor 1 needs to read elements {1,2,3,4} from processor 0 and {9,10,11,12} from processor 2, etc. In contrast, using DD_4B, each processor needs to read 2 elements from each of two other processors. DD_4A thus results in an unbalanced communication pattern, since processors 2 and 3 are required to read more elements than processors 1 and 4, while DD_4B partitions the domain in a symmetrical fashion, which should yield a more balanced communication pattern, as well as less overall communication. This benefit will increase with larger data sets. In the next section, we use MDS and CXpa to verify the above observations and speculations.

5.4.2 Analyzing the Performance of CRASH-Serial, CRASH-SP, and CRASH-SD

5.4.2.1 Simulated Performance Reported by MDS

We simulate the performance of CRASH-Serial, CRASH-SP, and CRASH-SD by running the three application models described in the previous section on MDS with an HP/Convex SPP-1600 machine model. We selectively present some performance metrics of the simulated performance in Tables 5-2 to 5-5, and discuss them below:

- *Computation Time* (Table 5-2): The total computation time remains the same for these three versions. Note that the computation time is calculated by accumulating the weights of the tasks that are executed in the simulated run.

- *Communication Time* (Table 5-3): The (required) communication time in CRASH-SD is less than CRASH-SP, as we speculated in the previous section. However, MDS also reports that some sharing of cache blocks occurs in CRASH-SD, which may cause extra (optional) communications.

- *Barrier Synchronization Time* (Table 5-4): The barrier synchronization time is the time that each processor spends in the barrier synchronization routine. In CRASH-SP and CRASH-SD, there are four barriers per iteration, and the barrier synchronization time accumulates to 1.164 seconds after 10,000 iterations. Note that MDS estimates the barrier synchronization time based on the microbenchmarking results in [12]

Program Region	Computation Time (sec.)		
	CRASH-Serial	CRASH-SP	CRASH-SD
Initialization	0.00731	0.00731	0.00731
Contact_phase	1.323	1.323	1.323
Update_phase	1.092	1.092	1.092
Total	2.422	2.422	2.422

Table 5-2: Computation Time Reported by MDS.

Program Region	Communication Time (sec.)		
	CRASH-Serial	CRASH-SP	CRASH-SD
Initialization	0	0	0
Contact_phase	0	0.184	0.147
Update_phase	0	0.139	0.118
Total	0	0.322	0.265

Table 5-3: Communication Time Reported by MDS.

Program Region	Synchronization Time (sec.)		
	CRASH-Serial	CRASH-SP	CRASH-SD
Initialization	0	0	0
Contact_phase	0	0.582	0.582
Update_phase	0	0.582	0.582
Total	0	1.164	1.164

Table 5-4: Barrier Synchronization Time Reported by MDS

Program Region	Wall Clock Time (sec.)		
	CRASH-Serial	CRASH-SP	CRASH-SD
Initialization	0.00731	0.00731	0.00731
Contact_phase	1.323	1.003	0.957
Update_phase	1.092	0.925	0.906
Total	2.422	1.935	1.871

Table 5-5: Wall Clock Time Reported by MDS.

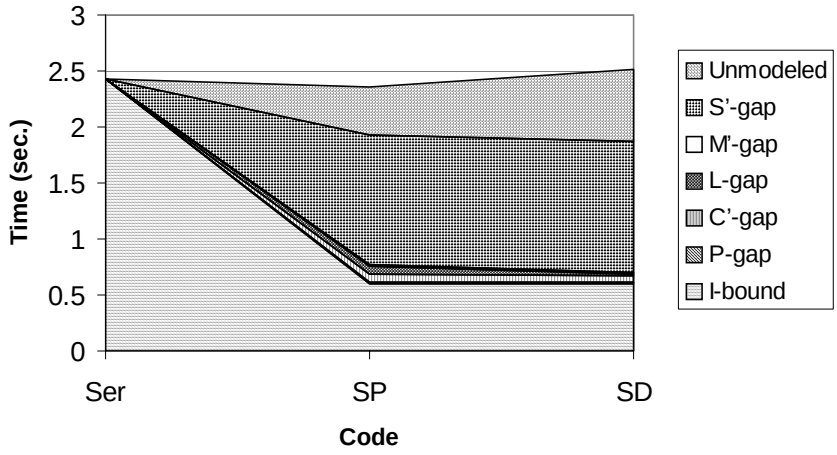


Figure 5-24: Performance Bounds and Gaps Calculated for CRASH-Serial, CRASH-SP, and CRASH-SD.

- *Wall Clock Time* (Table 5-5): The wall clock time report shows that CRASH-SD should run faster than CRASH-SP, if optional communications were not of concern. However, the parallel speedup for both parallel versions is poor, only 1.252 for CRASH-SP and 1.294 for CRASH-SD.

We are mostly concerned about the poor scalability in the parallel versions. The cause of this poor scalability is primarily due to the high barrier synchronization cost relative to the run time. The barrier synchronization time consists of about 60% of the wall clock time for the parallel versions. This is a classic example of the conflict of interest between fine-grain parallelization and synchronization cost; the percentage of synchronization overhead time should decrease as subdomain size per processor increases. The ratio between the required communication time and the computation time is in the range of 10-13% for the parallel versions, which is a significant, but less serious problem for this case.

Another way, and perhaps a better way, for novice programmers to visualize the performance problems is to analyze the simulated performance using hierarchical performance bounds. MDS reports hierarchical performance bounds for these three versions, as shown in Figure 5-24 and Tables 5-6 and 5-7. We notice that the synchronization gaps are the main

Performance Bounds	Time (sec.) (Percentage)		
	CRASH-Serial	CRASH-SP	CRASH-SD
I	2.422 (100%)	0.606 (25.7%)	0.606 (24.1%)
IP	2.422 (100%)	0.611 (25.9%)	0.611 (24.3%)
IPC	2.422 (100%)	0.692 (29.4%)	0.677 (27.0%)
IPCL	2.422 (100%)	0.764 (32.4%)	0.679 (27.0%)
IPCLM	2.422 (100%)	0.770 (32.6%)	0.707 (28.2%)
IPCLMS	2.422 (100%)	1.935 (82.2%)	1.871 (74.5%)
Actual Runtime	2.422 (100%)	2.355 (100%)	2.51 (100%)

Table 5-6: Hierarchical Parallel Performance Bounds (as reported by MDS) and Actual Runtime (Measured).

Performance Bounds or Gap	Time (sec.) (Percentage)		
	CRASH-Serial	CRASH-SP	CRASH-SD
I-Bound	2.422 (100%)	0.606 (25.7%)	0.606 (24.1%)
Parallelization-Gap	0 (0%)	0.005 (0.2%)	0.005 (0.2%)
Communication-Gap	0 (0%)	0.081 (3.4%)	0.066 (2.6%)
Load Balance-Gap	0 (0%)	0.072 (3.0%)	0.002 (0.1%)
Multiphase-Gap	0 (0%)	0.007 (0.3%)	0.027 (1.1%)
Synchronization-Gap	0 (0%)	1.164 (49.4%)	1.164 (16.4%)
Unmodeled Gap	0 (0%)	0.42 (17.8%)	0.639 (25.4%)

Table 5-7: Performance Gaps (as reported by MDS).

Working Set Analysis	CRASH-Serial	CRASH-SP	CRASH-SD
Basic Working Set Characterization			
1. Working Set, Accessed in the Program (Bytes)	1152	1152	1152
2. Working Set, Read from Memory (Bytes)	1152	1152	1152
3. Working Set, Written by Processor(s) (Bytes)	1152	1152	1152
Degree of Sharing			
4. Working Set, Accessed by 1 Processor (Bytes)	1152 (100%)	384 (33%)	576 (50%)
5. Working Set, Accessed by 2 Processors (Bytes)	0	384 (33%)	384 (33%)
6. Working Set, Accessed by 3 Processors (Bytes)	0	384 (33%)	192 (17%)
7. Working Set, Accessed by 4 Processors (Bytes)	0	0	0
False-Sharing of Cache Blocks			
8. Number of False-Shared Cache Blocks	0	0	12

Table 5-8: Working Set Analysis Reported by MDS

cause for the poor scalability in CRASH-SP and CRASH-SD. The communication (C) gap and load balance (L) gap in CRASH-SD are smaller than in CRASH-SP, which conforms to our expectations.

The results of working set analysis are shown in Table 5-8. We instruct MDS to analyze three major data structures, **`Position`**, **`Velocity`**, and **`Force`**. Each of these data structures is a 3-by-16 array of 8-byte real numbers. Three data structures form a working set whose size is 1152 bytes (3*16*8*3).

In CRASH-SP, 33% (384 bytes) of the working set is accessed by one processor, 33% by two processors, and 33% by three processors. A detailed report reveals that array **`Force`** is not shared by multiple processors. But internal the boundary elements of the **`Position`** and **`Velocity`** arrays are accessed by multiple processors. The domain decomposition in CRASH-SP causes each element of subdomains 0 and 3 to be accessed by two processors, while each element of subdomains 1 and 2 is accessed by three processors. On the other hand, for CRASH-SD, we expect a lower degree of sharing than for CRASH-SP, since four elements of the two shared arrays {1,4,13,16} are accessed by one processor only, and only four elements {6,7,10,11} are accessed by three processors. MDS does in fact report a lower degree of sharing in CRASH-SD.

However, MDS detects 12 cache blocks with false-sharing in CRASH-SD. False-sharing occurs in CRASH-SD because data from different subdomains share the same cache lines. For example, **`Position(i)`** is a vector consisting of three double real (8-byte) components, so it is actually a 2-dimensional array that is declared as **`real*8 Position(3, Max_Elements)`**[1]. The layout of array **`Position`** in the processor cache is shown in Figure 5-25. **`Postion(2)`** and **`Position(3)`** share one cache line, although they are assigned to different subdomains, 0 and 1 respectively. In all, four cache blocks are shared by different subdomains of **`Position`**. The same false-sharing occurs in accessing the **`Velocity`** and **`Force`** arrays.

1. For simplicity, when only one index is used to refer to a vector (Position, Velocity, or Force), it is meant to refer to all three elements in the referenced column. For example, Position(2) is often referred to Position(1:3,2).

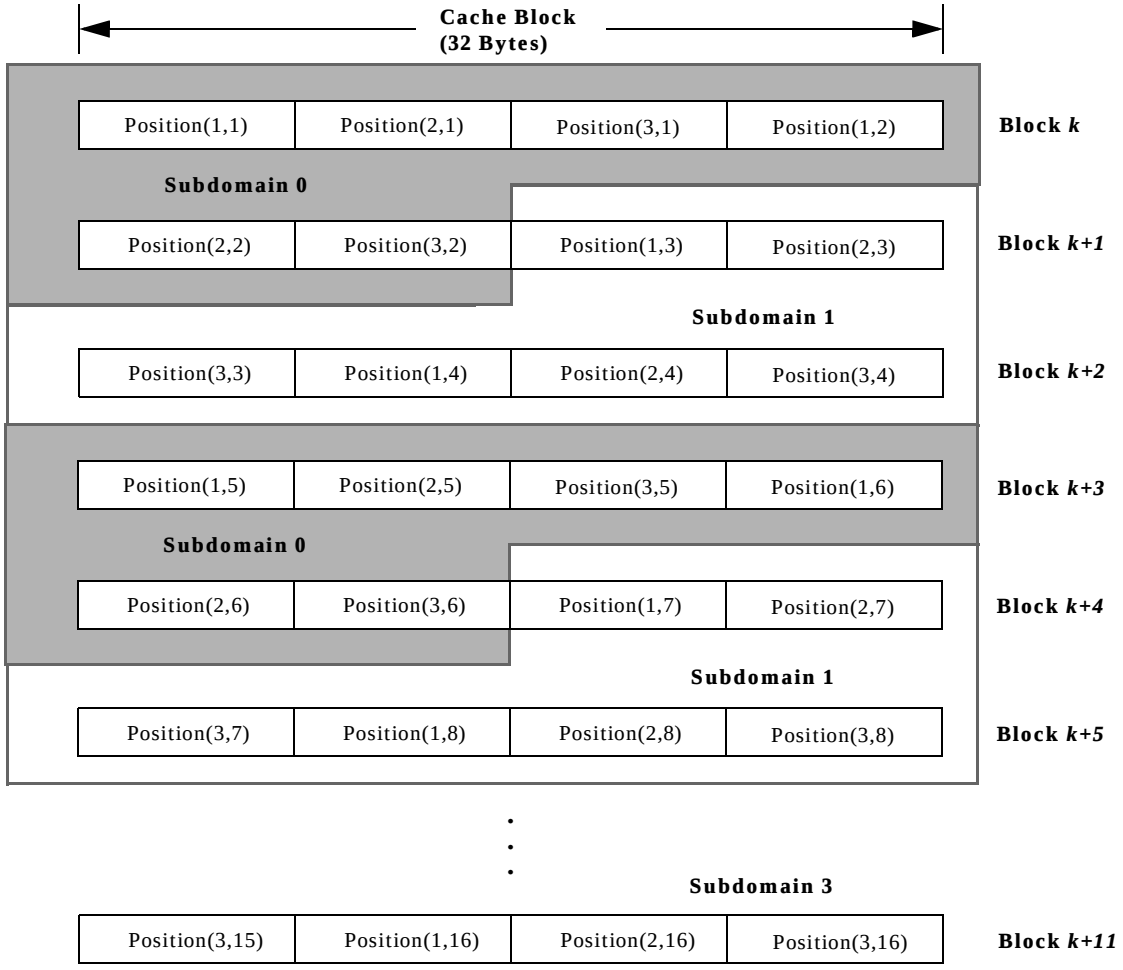


Figure 5-25: The Layout of Array `Position` in the Processor Cache in CRASH-SD.

5.4.2.2 Profiled Performance Reported by CXpa

We use CXpa to profile the performance of CRASH-Serial, CRASH-SP, and CRASH-SD running on HP/Convex SPP-1600. The wall clock time, CPU time, and cache miss latency for these three versions are shown in Tables 5-9 to 5-12. Table 5-9 shows the wall clock time when minimum profiling is applied. As mentioned above, the CXpa instrumentation may cause more dilation of the profiled performance when more routines are profiled. When the Contact phase and Update phase are profiled, the dilation of wall clock time is about 11% for CRASH-Serial, 23% for CRASH-SP, and 19% for CRASH-SD. Other performance metrics, including CPU time, cache miss latency, and cache miss count, are dilated as well.

	CRASH	CRASH-SP	CRASH-SD
Wall Clock Time	2.344	2.355	2.510
CPU Time	2.317	5.251	6.091
Cache Miss Latency	0.000267	1.294	2.397
Cache Miss Count	2020	1717976	2617441
Average Latency per Miss	132 ns	753 ns	916 ns

Table 5-9: Performance Metrics Reported by CXpa (Only the Main Program Is Instrumented).

Program Region	Wall Clock Time (sec.)		
	CRASH	CRASH-SP	CRASH-SD
Initialization	0.00817	0.015	0.015
Contact_phase	1.328	1.428	1.475
Update_phase	1.096	1.271	1.302
Total	2.604	2.906	2.983

Table 5-10: Wall Clock Time Reported by CXpa.

Program Region	CPU Time (sec.)		
	CRASH	CRASH-SP	CRASH-SD
Initialization	0.00731	0.00733	0.00733
Contact_phase	1.323	3.078	3.497
Update_phase	1.096	2.506	2.823
Total	2.588	5.77	6.505

Table 5-11: CPU Time Reported by CXpa.

Program Region	Total Cache Miss Latency (sec.)		
	CRASH	CRASH-SP	CRASH-SD
Initialization	0.000016	0.000031	0.000031
Contact_phase	0.023	0.842	1.416
Update_phase	0.023	0.547	1.019
Total	0.092	1.445	2.492

Table 5-12: Cache Miss Latency Reported by CXpa.

Program Region	Zero-Workload CRASH-SP	
	Cache Miss Latency (sec.)	Wall Clock Time (sec.)
Contact_phase	0.436	0.993
Update_phase	0.428	0.991
Total	0.864	1.984

Table 5-13: Cache Miss Latency and Wall Clock Time for a Zero-Workload CRASH-SP, Reported by CXpa

At first look, these CXpa profiles seem very different from the simulated performance obtained via MDS. CRASH-SD has the worst performance among these three versions, with the highest wall clock time, CPU time, and total cache miss latency. The CPU time in the two parallel versions is 2.2 to 2.6 times higher than the serial version. High total cache miss latencies contribute to the high CPU time in the parallel versions, but do not fully explain the increase in the CPU time. The average latency per cache miss (shown in Table 5-9) for the parallel versions is significantly (30% and 58%) higher than the cache miss latency modeled by MDS, where 500 ns and 580 ns are used to model a read miss and a write miss respectively. The increased average cache miss latency is possibly due to memory contention.

In the CXpa *parallel loop* reports (not shown), we find that the wall clock time within the parallel loops (excluding the spawn/join overhead) is considerably less than the wall clock time that includes the spawn/join overhead. For example, the heaviest loaded processor spent 0.443 seconds *within* the doall loop in **`Update_phase`**, but it spent 2.367 seconds executing **`Update_phase`** overall. Therefore, we believe that the spawn/join overhead is also a primary cause of the increase in CPU time for the parallel versions. To measure the spawn/join overhead, we run a zero-workload version of CRASH-SP, which does not perform any computation at all within each phase. The results, shown in Table 5-13, measure the total cache miss latency and wall clock time for spawning and joining 20,000 empty doall loops.

The total cache miss latency is unexpectedly high for CRASH-SP (1.445 seconds) and CRASH-SD (2.492 seconds). Some of the cache misses come from the barriers for spawning and joining parallel loops, which contribute approximately 0.864 seconds of cache miss latency, estimated from the total cache miss latency of the zero-workload CRASH-SP in Table 5-13. However, we cannot identify the cause(s) of the rest of cache miss latency based on these CXpa profiles.

5.4.2.3 Comparing MDS and CXpa Results

Even with the time dilation, CXpa generally gives more accurate performance profiles than MDS does for these three CRASH codes. But, based on the profiles, we are still unable to identify the exact causes for the performance overheads. As discussed in Section 5.4.2.2, we speculate that a large portion of the run time comes from spawning and joining parallel loops, but we still cannot identify the cause(s) of the excessive cache miss latencies in the parallel codes.

On the other hand, MDS faithfully exposes the machine-application interactions based on the machine and application models provided by the user. MDS reports relatively high barrier synchronization cost for spawning and joining the parallel loops. MDS captures the required communications in the codes and simulates the performance based on these required communications. Consequently, the simulated performance tends to be optimistic. Note that MDS does report the number of memory blocks that are false-shared in CRASH-SD. These false-sharing memory blocks may explain why CRASH-SD actually performs worse than CRASH-SP.

Fortunately, we do not see any conflicts between the results reported by these two tools. Instead, these two tools complement one another in our performance analysis. The overall differences in the wall clock time reported by MDS and CXpa are shown in Figure 5-26. MDS reports longer wall clock time than CXpa for CRASH-Serial, but shorter for CRASH-SP and CRASH-SD. MDS reports longer wall clock time for CRASH-Serial because the CXpa profile that we used to model CRASH-Serial is more dilated, since we have to profile the run time for each phase. For CRASH-SP and CRASH-SD, MDS reports more optimistic wall clock time because there are some performance constraints that are not modeled in MDS.

The extra wall clock time in CRASH-SD is partly due to false-sharing and memory contention, as we explained previously. However, MDS does not detect any false-sharing in CRASH-SP and, based on the previous analysis, at this point we still cannot fully explain the 0.42 seconds difference between MDS-simulated wall clock time (1.935 seconds) and the CXpa-profiled wall clock time (2.355 seconds) for CRASH-SP (even after we doubled the cache miss latency per miss in MDS in order to account for the effect of memory contention).

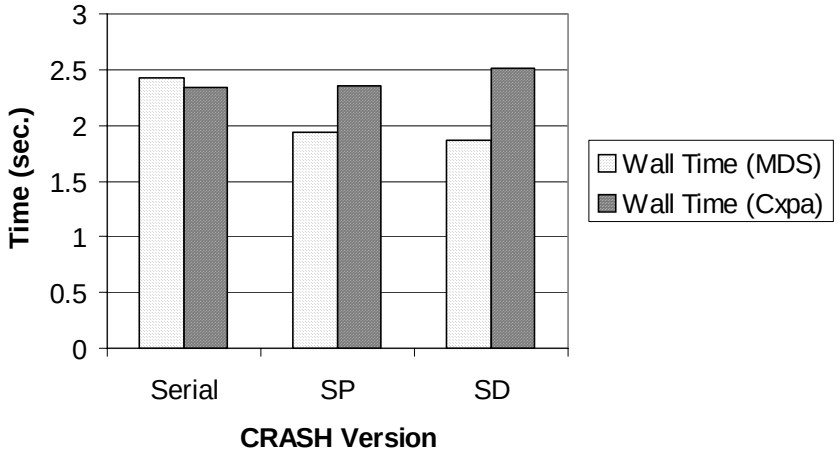


Figure 5-26: Comparing the Wall Clock Time Reported by MDS and CXpa.

We therefore considered the performance constraints that are not modeled in MDS to find possible causes for this unidentified overhead, and we discovered that thread migration is in fact responsible for this overhead. We had expected the parallel loops in CRASH-SP and CRASH-SD to be spawned with a consistent thread-processor assignment over the entire run, which was not the case. By binding the threads to processors permanently in CRASH-SP3 (Section 5.4.3.4), we were able to explain and eliminate the 0.42 seconds difference between MDS and CXpa wall clock time for CRASH-SP. Further performance tuning of CRASH is discussed in the next section.

5.4.3 Model-Driven Performance Tuning

In this section, we improve the performance of CRASH by applying our goal-directed performance tuning scheme in conjunction with our model-driven performance tuning approach. Below, we describe a series of tuning actions and their results.

5.4.3.1 First Parallelized Version: CRASH-SP

The performance bounds analysis for CRASH-SP is shown in Figure 5-27. As described in Chapter 4, we should apply tuning actions in a logical sequence to reduce the magnitude of significant gaps in each step. The major gaps in CRASH-SP and their causes are:

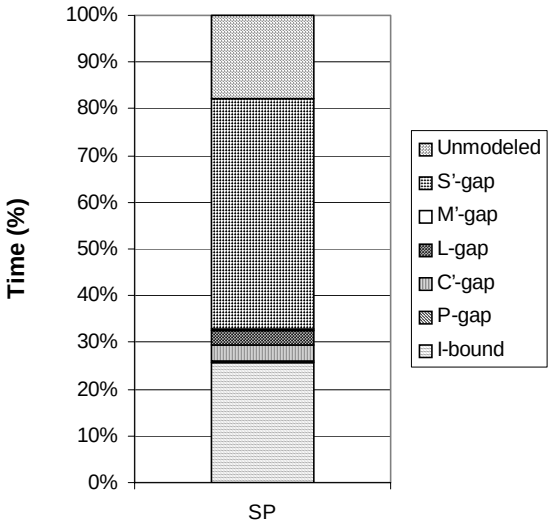


Figure 5-27: Performance Bounds Analysis for CRASH-SP.

- *S'-gap* (49.4% of the runtime): synchronization cost for executing the barriers.
- *Unmodeled gap* (17.8% of the runtime): false-sharing (optional) communications and other unknown factors.
- *C'-gap* (3.4% of the runtime): required communications.
- *L-gap* (3.0% of the runtime): overall load imbalance.

5.4.3.2 Better Domain Decomposition: CRASH-SD

Initially, as in Step 1: Action 1 (Section 3.2) of our tuning methodology, we would like to improve the domain decomposition in CRASH-SP. As mentioned in Section 5.4.1, the domain decomposition scheme incorporated in CRASH-SD is supposed to reduce the overhead due to communication and load imbalance. As shown in Figure 5-28, the C'-gap (required communication) and L-gap (overall load imbalance) are, in fact, reduced. However, the unmodeled gap is increased due to false-sharing communications, as explained in Section 5.4.2.1.

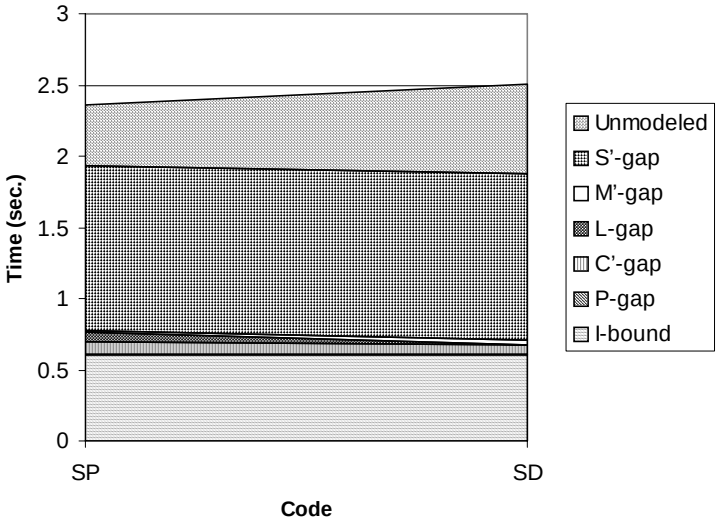


Figure 5-28: Performance Bounds Analysis for CRASH-SP and SD.

5.4.3.3 Eliminating False-sharing: CRASH-SD2

Since the C'-gap and the expanded unmodeled gap caused by the false-sharing communications in CRASH-SD are both significant, we chose to reduce the communication overhead as our next performance tuning step.

We believe that CRASH-SD should perform better than CRASH-SP if false-sharing can be eliminated from CRASH-SD. To eliminate false-sharing, we use padding to adjust the size of array **`Position`**, **`Velocity`**, and **`Force`**. In CRASH-SD, each of these arrays is defined as a one-dimensional array of vectors, where each vector consists of three 8-byte real numbers. Therefore, each is in fact a two-dimensional array declared as *(3,Max_Elements)*. In CRASH-SD2 we eliminate false-sharing by increasing the size of these arrays to *(4,Max_Elements)*, as illustrated in Figure 5-29. The expanded data layout, however, is less efficient in memory usage and has eight superfluous bytes in each cache block, which affects storage space and communication.

Consequently, the unmodeled gap, as shown in Figure 5-30, is reduced on CRASH-SD2 to about the same size as that of CRASH-SP. The C'-gap, however, is increased due to the

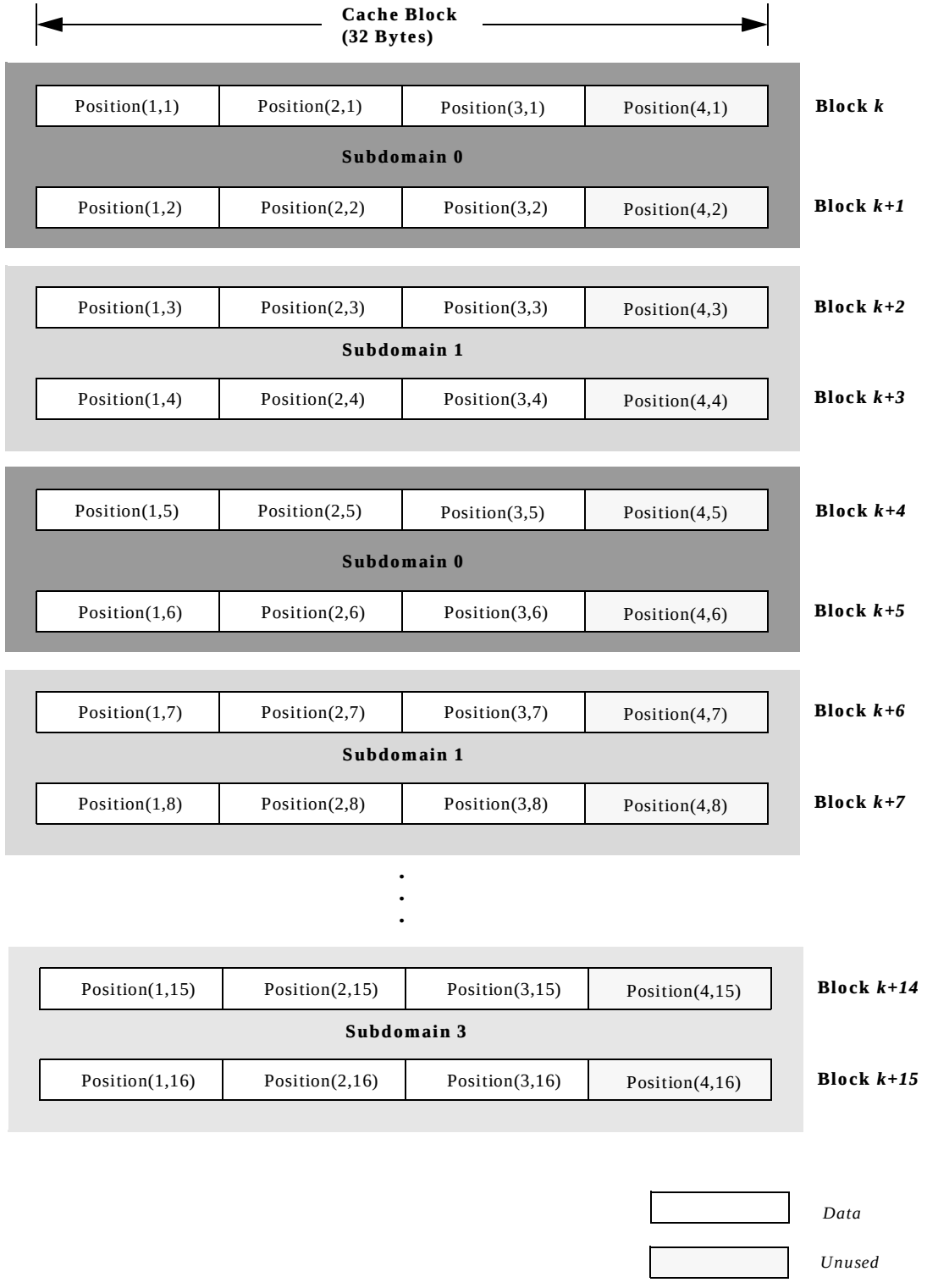


Figure 5-29: Layout of the Position Array in the Processor Cache in CRASH-SD2.

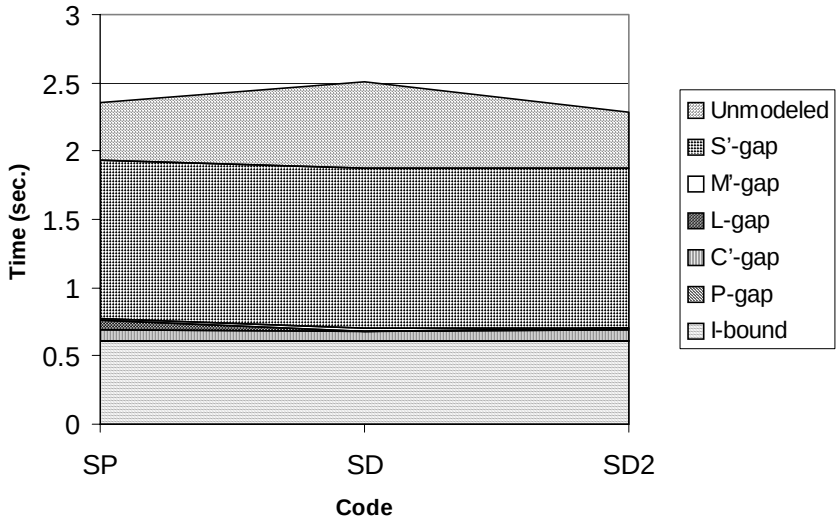


Figure 5-30: Comparing the Performance Gaps of CRASH-SD2 to Its Predecessors.

expanded data layout. Nevertheless, the overall performance of CRASH-SD2 is better than its predecessors. The remaining unmodeled gap and the expanded C'-gap are addressed further in the next version.

The simulated and profiled performance of CRASH-SD2 is shown (with the previous versions) in Figure 5-31. The profiled performance shows that CRASH-SD2 does indeed perform better than its predecessors in terms of CXpa wall clock time (2.288 seconds), and the profiled cache miss latency of CRASH-SD2 is significantly less than CRASH-SD due to the elimination of false-sharing. The working set analysis performed by MDS, shown in Table 5-14, confirms that the false-sharing is eliminated. Note that the communication time reported by MDS in Figure 5-31 is far less than the profiled cache miss latency, because MDS does not count the communication time spent within the synchronization routines, but CXpa does, as explained above in Section 5.4.2.2.

Judging from the CXpa reported cache miss latency in Figure 5-31, CRASH-SD2 causes more cache miss latency than CRASH-SP. This is due to the increase (33%) in the data structure size. MDS reports an increase in the working set size of CRASH-SD2 relative to CRASH-SP and CRASH-SD (see Table 5-14). Therefore, array padding, while it is a simple solution for

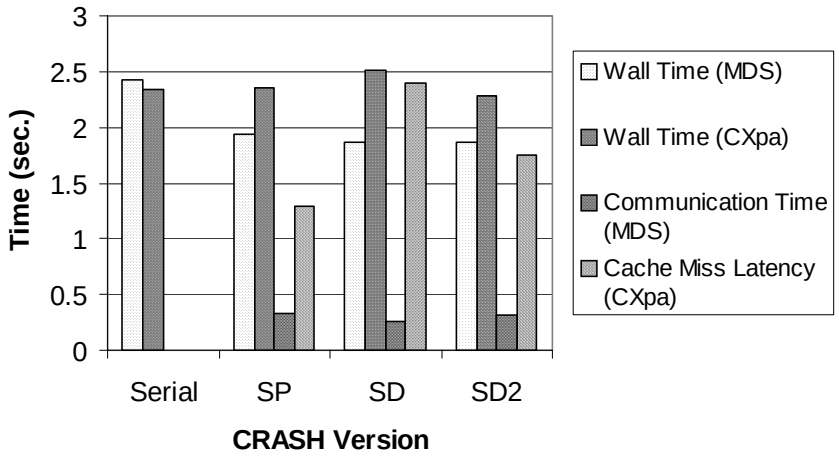


Figure 5-31: Comparing the Performance of CRASH-SD2 to Its Predecessors.

Working Set Analysis	CRASH-SP	CRASH-SD	CRASH-SD2
Basic Working Set Characterization			
9. Working Set, Accessed in the Program (Bytes)	1152	1152	1536
10. Working Set, Read from Memory (Bytes)	1152	1152	1536
11. Working Set, Written by Processor(s) (Bytes)	1152	1152	1536
Degree of Sharing			
12. Working Set, Accessed by 1 Processor (Bytes)	384 (33%)	576 (50%)	768 (50%)
13. Working Set, Accessed by 2 Processors (Bytes)	384 (33%)	384 (33%)	512 (33%)
14. Working Set, Accessed by 3 Processors (Bytes)	384 (33%)	192 (17%)	256 (17%)
15. Working Set, Accessed by 4 Processors (Bytes)	0	0	0
False-Sharing of Cache Blocks			
16. Number of False-Shared Cache Blocks	0	12	0

Table 5-14: Working Set Analysis Reported by MDS for CRASH-SP, CRASH-SD, and CRASH-SD2.

eliminating false-sharing, does not result in a significant improvement over CRASH-SP in terms of overall performance.

5.4.3.4 Eliminating Subdomain Migration: CRASH-SD3

At the end of Section 5.4.2, we mentioned that the subdomains (threads) may migrate during the execution, which is one aspect of machine-application behavior that MDS does not

```
      program CRASH-SD3

      .... (Variable declaration and initialization omitted).

c$dir loop_parallel(ivar=d)
c$dir loop_private(t,ii,i,j,type_element)
      do d=1,Num_Subdomains

c Main Simulation Loop
      t=0

c First phase: generate contact forces
100   wait_barrier(barr1,Num_Subdomains)
      do ii=1,Num_Elements_in_subdomain(d)
        i=global_id(ii,d)
        Force(i)=Contact_force(Position(i),Velocity(i))
        do j=1,Num_Neighbors(i)
            Force(i)=Force(i)+Propagate_force(Position(i),Velocity(i),
              Position(Neighbor(j,i),Velocity(Neighbor(j,i))
        end do
      end do
      wait_barrier(barr2,Num_Subdomains)

c Second phase: update position and velocity
200   wait_barrier(barr3,Num_Subdomains)
      do ii=1,Num_Elements_in_subdomain(d)
        i=global_id(ii,d)
        type_element=Type(i)
        if (type_element .eq. plastic) then
            call Update_plastic(i, Position(i), Velocity(i), Force(i))
        else if (type_element .eq. glass) then
            call Update_glass(i, Position(i), Velocity(i), Force(i))
        end if
      end do
      wait_barrier(barr4,Num_Subdomains)

      if (end_condition) stop
      t=t+t_step
      goto 100

      end do
      end
```

Figure 5-32: A Pseudo Code for CRASH-SD3

model. Accordingly, we selected Action 3, as discussed in (Section 3.3.2), in an attempt to minimize subdomain migration by permanently binding subdomains to processors. The resulting pseudo code for CRASH-SD3 is shown in Figure 5-32. CRASH-SD3 spawns threads using the **loop_parallel** compiler directive before the main simulation loop starts. Since each of these threads is responsible for one subdomain throughout the main simulation loop, the subdomain cannot migrate during the execution. Explicit barriers (**wait_barrier()**) are placed at the beginning and the end of each phase to ensure correct parallel execution. Note that

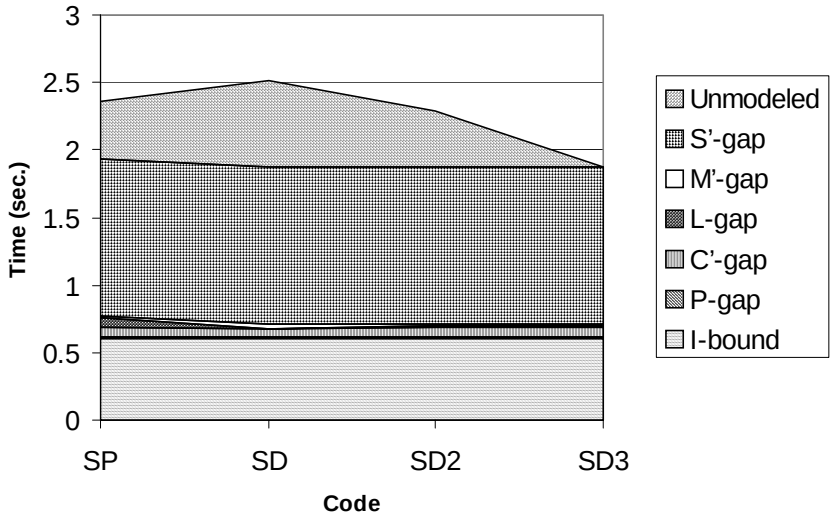


Figure 5-33: Comparing the Performance Gaps of CRASH-SD3 to Its Predecessors.

explicit barriers were not needed in CRASH-SP/SD/SD2 since their phases were formed by DOALL loops and barriers will be invoked when the DOALL loops are spawned and joined.

The performance bounds analysis (Figure 5-33) shows that the unmodeled gap has now been eliminated from CRASH-SD3, due to permanently binding subdomains to processors. The simulated and profiled performance of CRASH-SD3 are compared with previous versions in Figure 5-34. The MDS simulated wall clock time of CRASH-SD3 is now very close to the CXpa profiled performance of CRASH-SD3 (due to the elimination of thread migration which, as stated above, is accounted for in CXpa, but not modeled in MDS). MDS wall clock time is slightly longer than CXpa wall clock time because of the time dilation in the application model used in MDS. Elimination of thread migration causes a visible decrease in cache miss latency.

5.4.3.5 Reducing Synchronization Cost: CRASH-SD4

In this case study, we skip Step 3 (Optimizing Processor Performance), because we are not interested in tuning processor performance. Since the L-gap (overall load imbalance) is not significant in CRASH-SD3, we also skip Step 4 (Balancing the Load for Single Phases). In this version, we attempt to reduce the S'-gap by reducing the number of barriers.

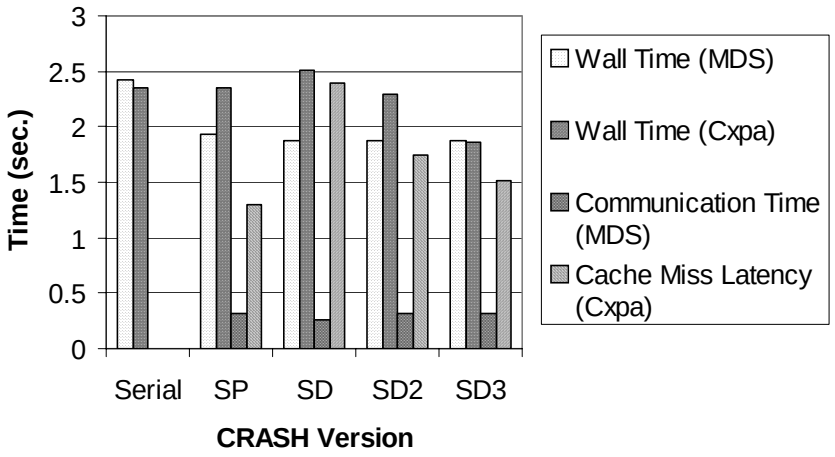


Figure 5-34: Comparing the Performance of CRASH-SD3 to Its Predecessors.

According to hierarchical performance bounds analysis performed by MDS (Figure 5-24), the synchronization time is obviously the most significant performance problem exhibited by the parallel CRASH codes introduced so far. In CRASH-SD3, we have a better view of this problem because the barriers are explicitly placed in the code. From Figure 5-32, we notice that some of the barriers, namely, **`barr2-barr3`** and **`barr4-barr1`**, are placed consecutively and hence cause redundant synchronization time. This is typical when DOALL loops or automatic parallelization are used in a code, and most compilers today do not attempt to eliminate the redundant barriers.

We remove redundant barriers by replacing consecutive barriers with one barrier. Consequently, **`barr3`** and **`barr4`** are removed[1], as shown in Figure 5-35. The performance bound analysis reported by MDS shows that this new version, CRASH-SD4, is significantly improved over CRASH-SD3 due to reduced synchronization overhead (smaller S'-gap), as shown in Figure 5-36. Approximately 50% of the S' gap is eliminated as a result of removing two of the four barriers.

1. Alternatively, we could choose to remove (barr1,barr2), (barr1,barr3) or (barr2,barr4), which all yield the same performance.

```
      program CRASH-SD4

      .... (Variable declaration and initialization omitted).

c$dir loop_parallel(ivar=d)
c$dir loop_private(t,ii,i,j,type_element)
      do d=1,Num_Subdomains

c Main Simulation Loop
      t=0

c First phase: generate contact forces
100   wait_barrier(barr1,Num_Subdomains)
      do ii=1,Num_Elements_in_subdomain(d)
        i=global_id(ii,d)
        Force(i)=Contact_force(Position(i),Velocity(i))
        do j=1,Num_Neighbors(i)
            Force(i)=Force(i)+Propagate_force(Position(i),Velocity(i),
              Position(Neighbor(j,i),Velocity(Neighbor(j,i))
        end do
      end do

c Second phase: update position and velocity
200   wait_barrier(barr3,Num_Subdomains)
      do ii=1,Num_Elements_in_subdomain(d)
        i=global_id(ii,d)
        type_element=Type(i)
        if (type_element .eq. plastic) then
            call Update_plastic(i, Position(i), Velocity(i), Force(i))
        else if (type_element .eq. glass) then
            call Update_glass(i, Position(i), Velocity(i), Force(i))
        end if
      end do

      if (end_condition) stop
      t=t+t_step
      goto 100

      end do
      end
```

Figure 5-35: A Pseudo Code for CRASH-SD4

At this point, the S'-gap is the only performance gap that is still critical to the overall performance for this particular case study. It is possible that the S'-gap will become less significant and some other gaps will become more critical if the computation workload between the barriers increased with a larger input data set. Thus, we continue to fine-tune the code in areas where we can apply further tuning actions. The remaining tuning actions demonstrate the ability of MDS to provide subtle performance assessment for evaluating the results of fine-tuning actions and thereby prepare the code for larger input data sets. Furthermore, some innovative tuning actions that are not immediately available, such as the use of double buffering to reduce the number of barriers in CRASH-SD8 (Section 5.4.3.9), may be inspired after some other actions are applied.

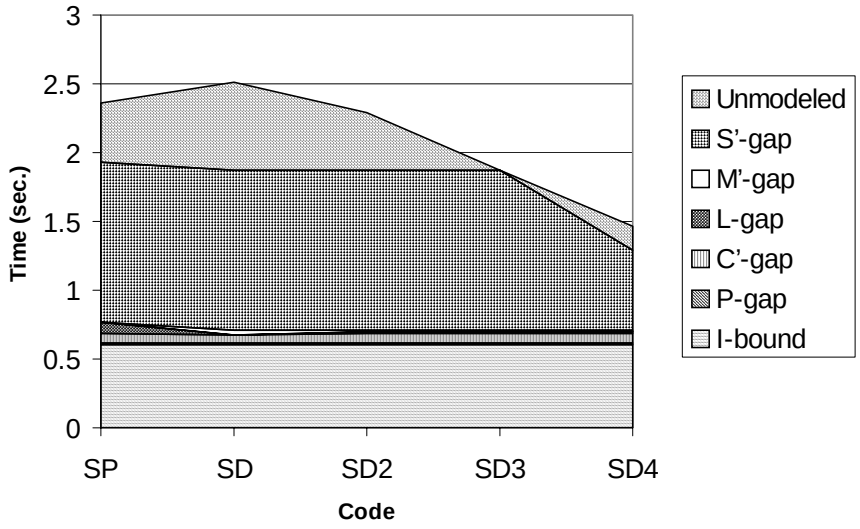
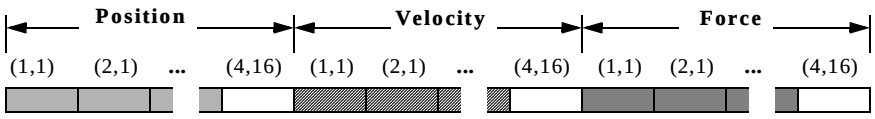


Figure 5-36: Comparing the Performance of CRASH-SD4 to Its Predecessors.

5.4.3.6 Array Grouping (1):CRASH-SD5

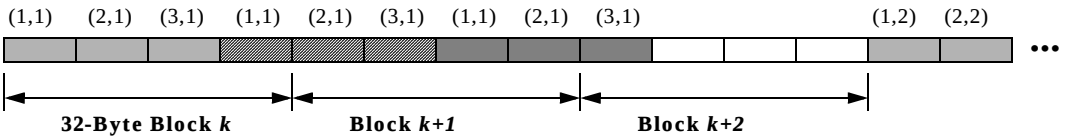
The array padding method did eliminate false-sharing for CRASH-SD2, SD3, and SD4, but it also increased the required communication time because accessing each 24-byte vector brings 8 bytes of unused data into the cache. The data layout in these codes is as shown in Figure 5-37(a), and their working set size is 1536 bytes. One better, but more sophisticated way is to apply array grouping (Action 5), as discussed in Section 3.3.2.

First, we attempt to tune CRASH by grouping arrays **`Position`**, **`Velocity`**, and **`Force`**. For a new version of CRASH, CRASH-SD5, we place vectors **`Position(i)`**, **`Velocity(i)`**, and **`Force(i)`** consecutively in the memory space, as shown in Figure 5-37(b). Unfortunately, this modification does not reduce the memory requirement, instead, it results in a different sort of false-sharing. MDS reports that there are 12 memory blocks that are false-shared in CRASH-SD5. False-sharing occurs because part of **`Velocity(i)`** and part of **`Force(i)`** share the same block. During *Contact*, **`Force(i)`** is written by its owner processor, but **`Velocity(i)`** can be read by one or two other processors if element i is at the boundary of two or three subdomains.



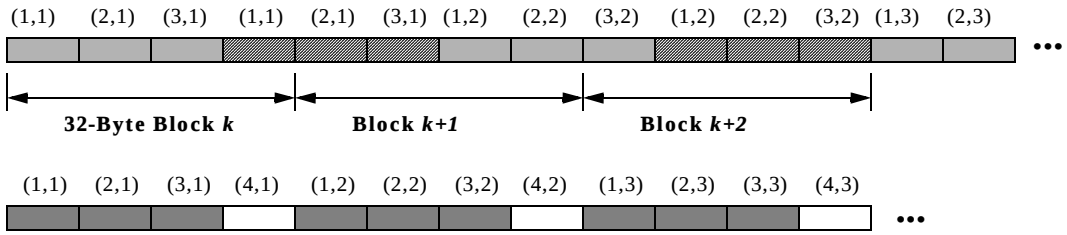
Total Size; 4*16*8*3=1536 Bytes

(a) The Data Layout in CRASH-SD2, SD3, and SD4



Total Size; 32 * 6 =1536 Bytes

(b) The Data Layout in CRASH-SD5



Total Size; 3*16*2*8+4*16*8=1280 Bytes

(c) The Data Layout in CRASH-SD6

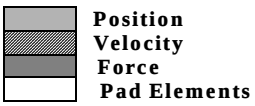


Figure 5-37: The Layout in CRASH-SD2, SD3, SD4, SD5, and SD6.

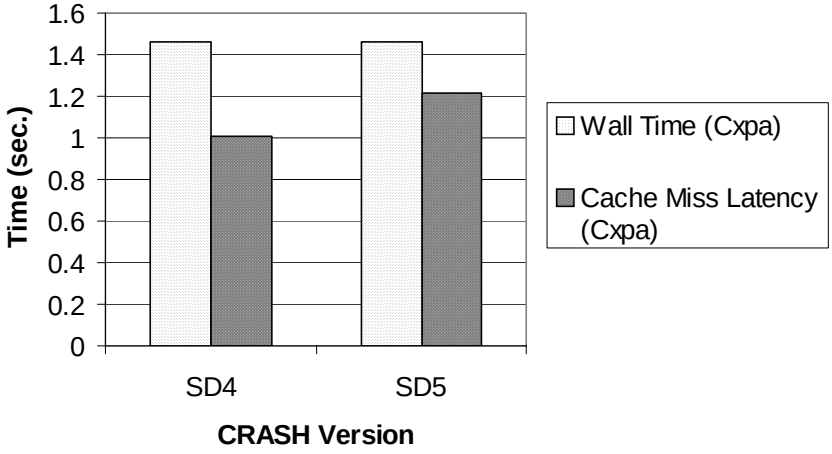


Figure 5-38: Comparing the Performance of CRASH-SD5 to CRASH-SD4.

The profiled performance of CRASH-SD5 is shown in Figure 5-38 along with CRASH-SD4. Although the cache miss latency of CRASH-SD5 is higher than CRASH-SD4, interestingly, the profiled wall clock times of these two versions are almost identical.

The failure to improve performance with CRASH-SD5 is a result of blindly applying array grouping, which in this case introduces a different form of false sharing without eliminating pad elements. Unfortunately, while some theories (e.g. [50]) have been proposed to guide the use of array grouping, we have not seen any of those theories implemented in any compiler or public domain tool. We chose to present this case to demonstrate that failure of an ill-conceived tuning step is common, and we need proper tools to help us minimize such mistakes along with the wasted effort in implementing them and the performance degradation and other complications that they can cause.

5.4.3.7 Array Grouping (2): CRASH-SD6

To properly apply array grouping, the program's data access pattern must be considered. In [50], Shih and Davidson provide a systematic method of array grouping to reduce communication and improve the locality of parallel programs. Choosing arrays for grouping is a critical step. It is determined by analyzing the array reference patterns recorded in the *Program*

Program Region	Position	Velocity	Force
Contact_phase	RI^2*(global),* RI^2*(neighbor)*	RI^2*(global),* RI^2*(neighbor)*	WI^2*(global)*
Update_phase	RI^2*(global),* WI^2*(global)*	RI^2*(global),* WI^2*(global)*	RI^2*(global)*

Table 5-15: PSAT of Arrays Position, Velocity, and Force in CRASH.

Section Array Table (PSAT)[1]. The PSAT of CRASH, for any of the versions discussed here, is shown in Table 5-15 and explained below:

- Each reference pattern recorded in the PSAT (e.g. RI^2*(global)*) starts with a letter *R* (Read) or *W* (Write) to indicate the type of access.

- Reference patterns are classified as *C* (consecutive), I^d*(X)* (indirect on dimension *d* through array *X*), and F^d*(f)* (non-unit stride on dimension *d* using a function *f* of loop index variables). For example, one reference pattern of array **`Position`** in **`Contact_phase`** is specified by I^2*(global)*, which indicates that the second dimension of **`Position`** is indexed indirectly though **`global(ii)`**.

- One array can have more than one reference pattern in a particular program section.

It is quite simple for us to create the above PSAT using the information in the data dependence module of CRASH. Based on the array grouping method of [50], or intuitively, grouping arrays **`Position`** and **`Velocity`** should improve the code performance since these arrays share identical reference patterns, but grouping all three arrays as we did in CRASH-SD5 may cause problems since their reference patterns are quite different.

The results of the above discussion is implemented in new version of CRASH, CRASH-SD6, which uses the data layout scheme shown in Figure 5-37(c). Both the simulated and profiled performance of CRASH-SD6, as shown in Figure 5-39, show a significant reduction in the communication overhead (MDS) and cache miss latency (CXpa), because 25% of the communications of array **`Position`** and **`Velocity`** are eliminated. Although the CXpa-profiled

1. The term *program section* is equivalent to the term *program region* used in this dissertation. To be consistent with the original work, we use *program section* in the discussion of array grouping here.

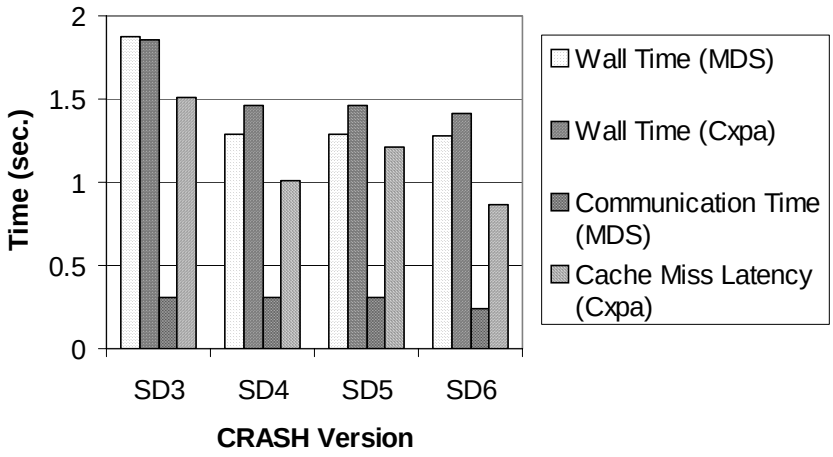


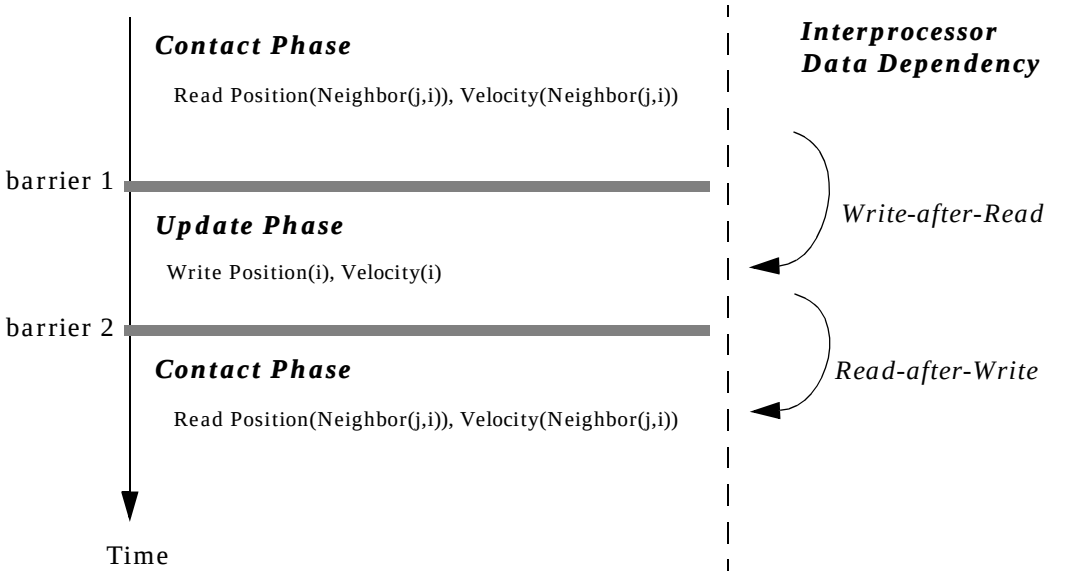
Figure 5-39: Comparing the Performance of CRASH-SD6 to Previous Versions.

cache miss latency is reduced by 0.346 seconds, the CXpa-profiled wall clock time is only reduced by 0.05 seconds, because (i) the reduction is shared by four processors, and (ii) some of the reduced cache miss latency was tolerated by the PA7200 processor's out-of-order execution capability in CRASH-SD5.

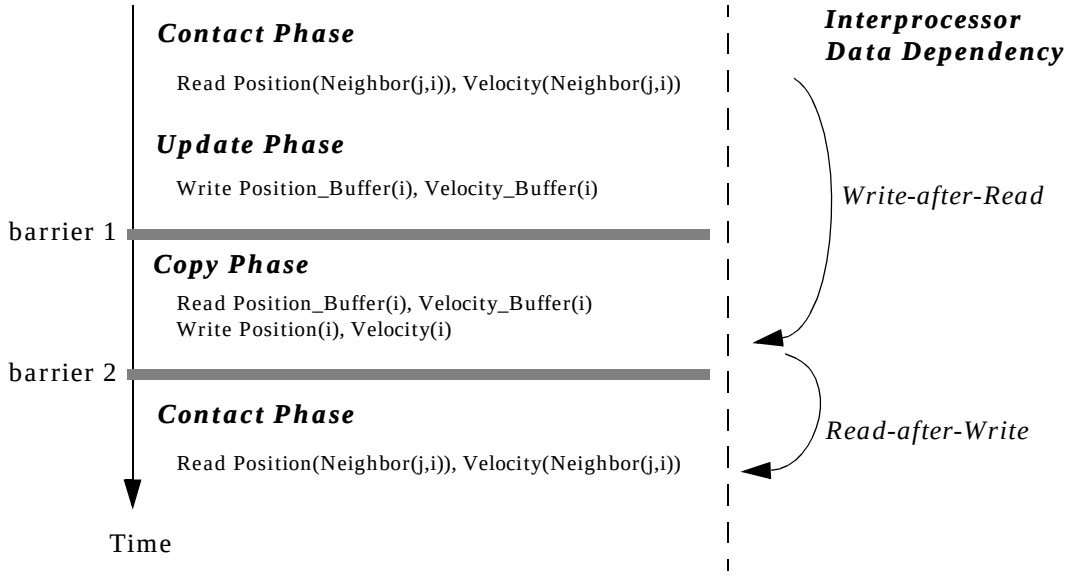
5.4.3.8 Fusing Contact and Update: CRASH-SD7

In the previous versions of CRASH, the execution is separated into two phases because of the *interprocessor* data dependencies, as discussed in Section 2.2.4 and illustrated in Figure 5-40(a). One type of the data dependence that exists in CRASH is *Write-After-Read* (WAR), which prohibits us from writing arrays **`Position`** and **`Velocity`** in *Update* before *Contact* finishes reading them. MDS performs data flow analysis to help the user to distinguish different types of data dependencies (RAW, WAR, or WAW), as well as to visualize the RAW dependencies between program regions and between processors. Figure 5-41(a) shows the data dependence graph of CRASH-SD6.

WAR dependencies can be *delayed* by using buffers, as shown in Figure 5-40(b). Instead of writing the results to **`Position`** and **`Velocity`**, the *Update* phase in CRASH-SD7, as shown in Figure 5-42, now writes to buffers **`Position_Buffer`** and **`Velocity_Buffer`**. An additional phase, the *Copy* phase, is performed after *Update* to copy the results from the buffers

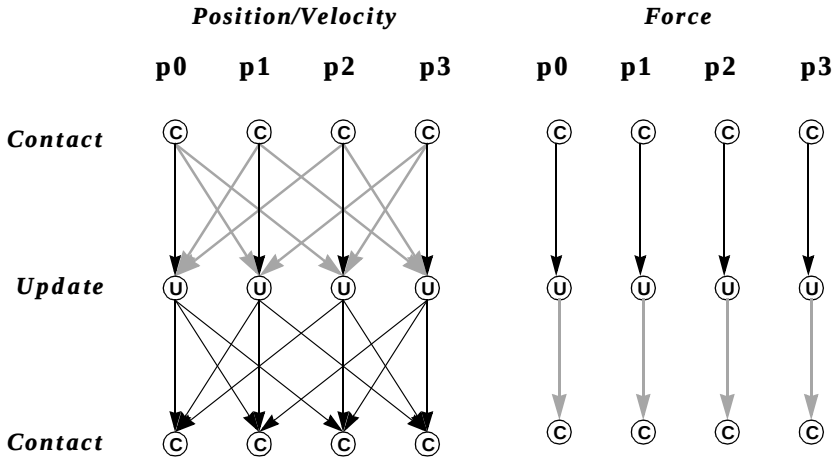


(a) Data Accesses that Cause Interprocessor Data Dependencies in CRASH-SD6.

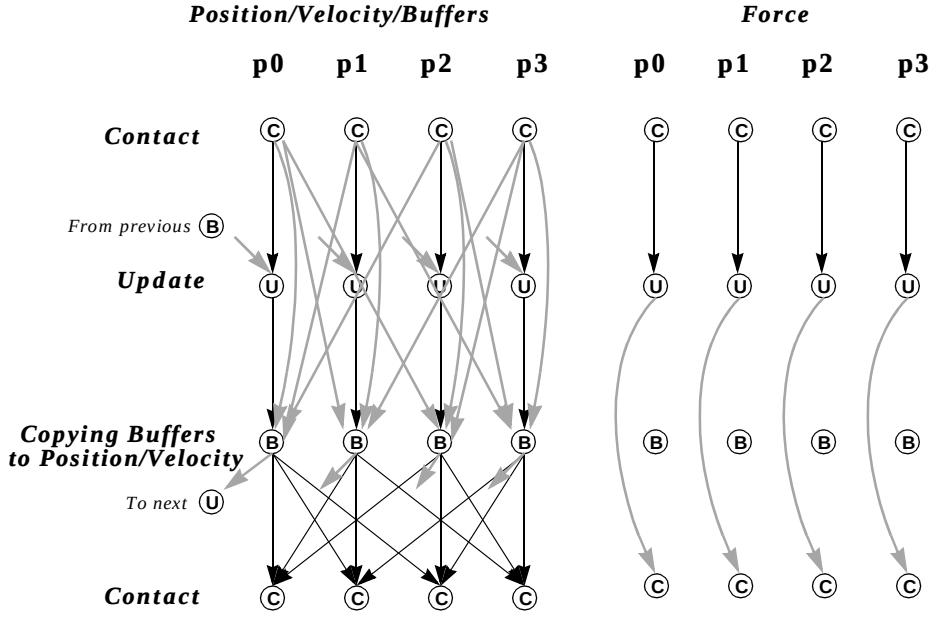


(b) Data Accesses that Cause Interprocessor Data Dependencies When Buffers Are Used in CRASH-SD7.

Figure 5-40: Delaying Write-After-Read Data Dependencies By Using Buffers.



(a) The Data Dependence Graph of CRASH-SD6



(b) The Data Dependence Graph after Buffers Are Used to Remove the WAR dependencies from Contact to Update by Accessing Position and Velocity.

RAW *WAR*
(WAW dependencies are not shown.)

Figure 5-41: The Results of Delaying Write-After-Read Data Dependencies By Using Buffers.

```
      program CRASH-SD7

      .... (Variable declaration and initialization omitted).

c$dir loop_parallel(ivar=d)
c$dir loop_private(t,ii,i,j,type_element)
      do d=1,Num_Subdomains

c Main Simulation Loop
      t=0

100   wait_barrier(barr1,Num_Subdomains)

      do ii=1,Num_Elements_in_subdomain(d)
        i=global_id(ii,d)

c Contact phase
        Force(i)=Contact_force(Position(i),Velocity(i))
        do j=1,Num_Neighbors(i)
            Force(i)=Force(i)+Propagate_force(Position(i),Velocity(i),
              Position(Neighbor(j,i),Velocity(Neighbor(j,i))
        end do

c Update phase
        type_element=Type(i)
        if (type_element .eq. plastic) then
            call Update_plastic(i, Position_Buffer(i),
                  Velocity_Buffer(i), Force(i))
        else if (type_element .eq. glass) then
            call Update_glass(i, Position_Buffer(i),
                  Velocity_Buffer(i), Force(i))
        end if

      end do

200   wait_barrier(barr2,Num_Subdomains)

c Copy phase
      do ii=1,Num_Elements_in_subdomain(d)
        i=global_id(ii,d)
            Position(i)=Position_Buffer(i)
            Velocity(i)=velocity_Buffer(i)
      end do

      if (end_condition) stop
      t=t+t_step
      goto 100

      end do
      end
```

Figure 5-42: A Pseudo Code for CRASH-SD7

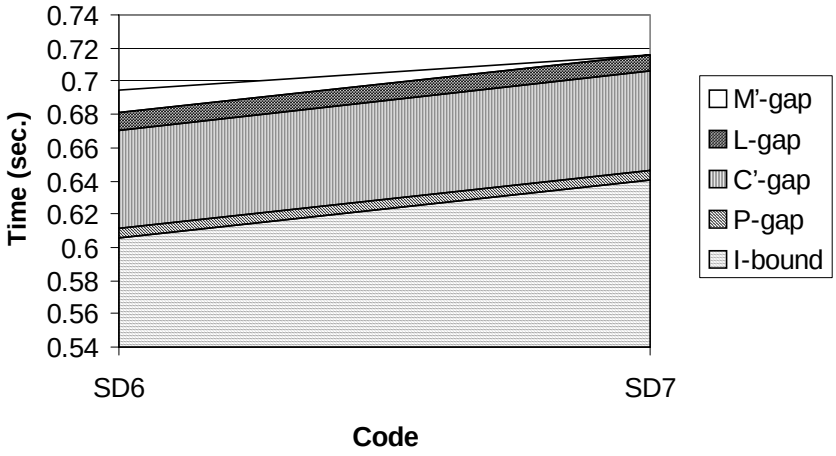


Figure 5-43: Performance Bounds Analysis for CRASH-SD5, SD6, and SD7.

back to **`Position`** and **`Velocity`**. Now there is no interprocessor data dependency between *Contact* and *Update*, hence *Contact* and *Update* need not be separated by a barrier. The WAR dependencies are delayed and occur between the *Contact* and *Copy* phases, so a barrier is needed to separate *Contact* and *Copy*. Figure 5-41(b) shows the data dependence graph that results from when these buffers are used. Since this code transformation adds a new routine for copying the buffers, the run time of the new code needs to be profiled to update the weight module, and to capture the effect of the new code in the control flow and data dependence modules.

What benefit can the code realize by delaying the WAR dependencies? As shown in Figure 5-40(b), the workload of *Contact* and *Update* in one iteration are no longer separated by a barrier, which eliminates the multiphase load imbalance due to that barrier. But note that we still need one barrier to ensure that *Contact* is finished before the buffers are copied back to **`Position`** and **`Velocity`**. Thus we can we cannot reduce the number of barriers with this approach.

Performance bounds analysis from MDS, as shown in Figure 5-43, reveals the changes in performance that results from CRASH-SD7. Since the run time overhead due to multiphase load imbalance is relatively low for the particular input used in this case study, the benefit of

this transformation is overshadowed by the added workload for performing the *Copy phase*, as reflected in the increased I-Bound of CRASH-SD7. Hence the performance of CRASH-SD7 is worse than CRASH-SD6 performance. However, CRASH-SD7 might perform better than CRASH-SD6 given a different input or machine configuration, particular when multiphase load imbalance is more critical to performance.

5.4.3.9 Double Buffering: CRASH-SD8

The extra overhead from the *Copy* phase in CRASH-SD7 might be eliminated by using a technique called *double buffering*, which is done by alternating the arrays that store the results between iterations. A new code, CRASH-SD8, which uses double buffering, is shown in Figure 5-44. The main loop in CRASH-SD7 is unrolled twice to form CRASH-SD8. In the first half, *Contact* reads (**`Position1`**, **`Velocity1`**) and *Update* writes (**`Position2`**, **`Velocity2`**); in the second half, *Contact* reads (**`Position2`**, **`Velocity2`**) and *Update* writes (**`Position1`**, **`Velocity1`**). Thus, there is no need for a copy phase between iterations.

In CRASH-SD8, interprocessor RAW data dependencies appear between *Update1* and *Contact2*, and between *Update2* and *Contact1*, and interprocessor WAR data dependencies appear between *Contact1* and *Update2*, and between *Contact2* and *Update1*, as shown in Figure 5-45. In one (unrolled) iteration, two barriers are sufficient to satisfy these interprocessor data dependencies. Each of these barriers separates one *Contact*-*Update* pair from the next. Compared to previous versions, the number of barriers per *Contact*-*Update* pair is reduced from two to one.

The performance bounds analysis from MDS, as shown in Figure 5-46, reveals the results of this code restructuring. CRASH-SD8 maintains the benefit of fusing the Contact and Update phases, as in CRASH-SD7, which results in a zero Multiphase (M') gap. The I bound of SD8 (the same as SD6) is smaller than that of SD7, due to the elimination of the Copy phase. Most importantly, the synchronization (S' gap) is reduced to 1/2 that of SD4 through SD7, and 1/4 that of SD through SD3.

```
      program CRASH-SD8

      .... (Variable declaration and initialization omitted).

c$dir loop_parallel(ivar=d)
c$dir loop_private(t,ii,i,j,type_element)
      do d=1,Num_Subdomains

c Main Simulation Loop
      t=0

100   wait_barrier(barr1,Num_Subdomains)
      do ii=1,Num_Elements_in_subdomain(d)
        i=global_id(ii,d)
c Contact1 phase
        Force(i)=Contact_force(Position1(i),Velocity1(i))
        do j=1,Num_Neighbors(i)
            Force(i)=Force(i)+Propagate_force(Position1(i),Velocity1(i),
              Position1(Neighbor(j,i),Velocity1(Neighbor(j,i))
        end do
c Update1 phase
        type_element=Type(i)
        if (type_element .eq. plastic) then
            call Update_plastic(i, Position2(i),
                  Velocity2(i), Force(i))
        else if (type_element .eq. glass) then
            call Update_glass(i, Position2(i),
                  Velocity2(i), Force(i))
        end if
      end do

      if (end_condition) stop
      t=t+t_step

200   wait_barrier(barr2,Num_Subdomains)
      do ii=1,Num_Elements_in_subdomain(d)
        i=global_id(ii,d)
c Contact2 phase
        Force(i)=Contact_force(Position2(i),Velocity2(i))
        do j=1,Num_Neighbors(i)
            Force(i)=Force(i)+Propagate_force(Position2(i),Velocity2(i),
              Position2(Neighbor(j,i),Velocity2(Neighbor(j,i))
        end do
c Update2 phase
        type_element=Type(i)
        if (type_element .eq. plastic) then
            call Update_plastic(i, Position1(i),
                  Velocity1(i), Force(i))
        else if (type_element .eq. glass) then
            call Update_glass(i, Position1(i),
                  Velocity1(i), Force(i))
        end if
      end do

      if (end_condition) stop
      t=t+t_step
      goto 100

      end do
      end
```

Figure 5-44: A Pseudo Code for CRASH-SD8

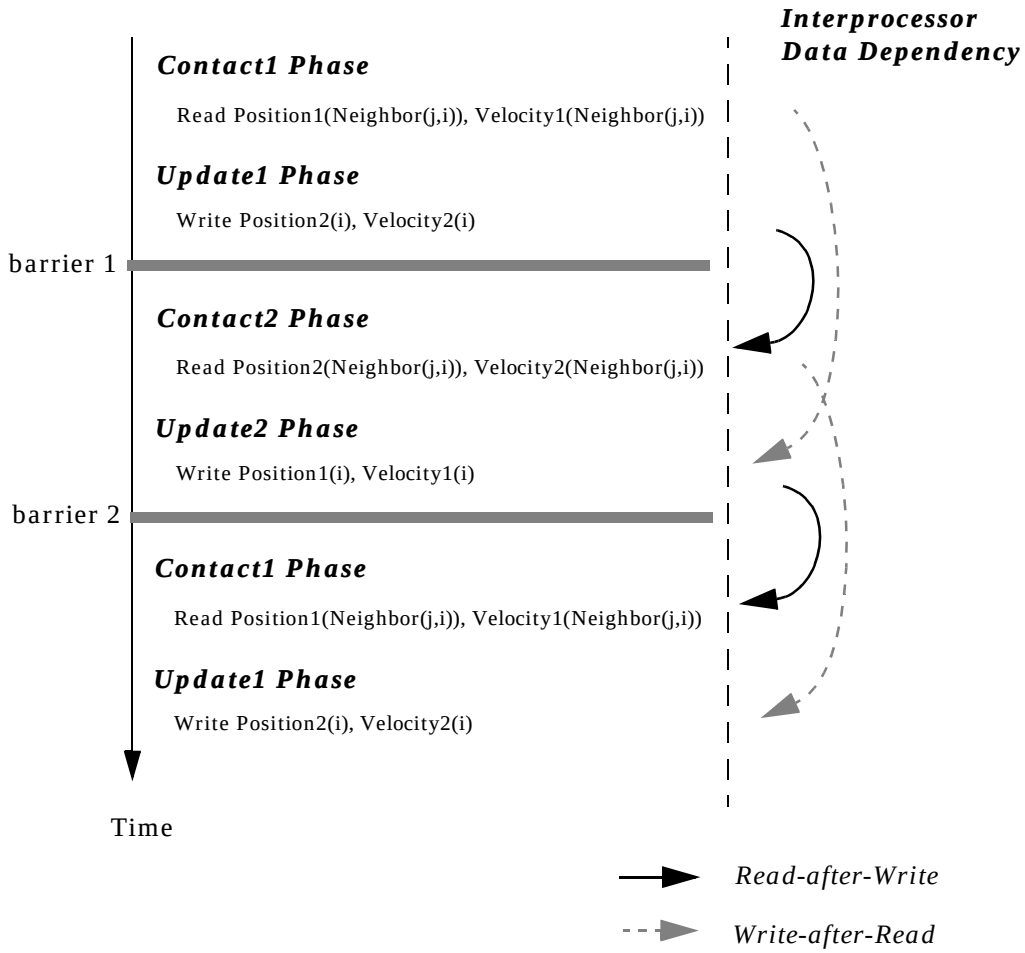


Figure 5-45: Data Accesses and Interprocessor Data Dependencies in CRASH-SD8.

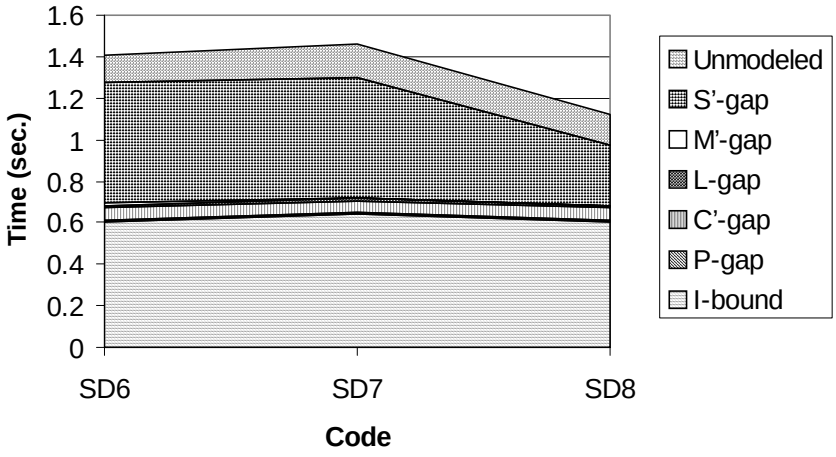


Figure 5-46: Performance Bounds Analysis for CRASH-SD6, SD7, and SD8.

5.4.4 Summary of the Case Study

In this section, we have used our model-driven simulator (MDS) to illustrate the use of application models in predicting and analyzing the performance of various CRASH versions and to guide the selection of tuning actions that move the code from one version to the next. We have applied a series of performance tuning actions in various goal-directed attempts to improve the performance of CRASH. The performance analysis reports from MDS, in conjunction with the performance profiles from CXpa, provided valuable guidance and evaluation in this performance tuning process. The overall results of performance tuning process are summarized in Figure 5-47. From the hierarchical performance bounds analysis shown in Figure 5-48, the cost of synchronization remains as the primary performance bottleneck in the parallel versions, but as the problem size increases, the synchronization cost will be decreased relative to computation time.

5.5 Summary

In this chapter, we have described an application modeling process that is capable of abstracting important application behavior into a set of programmer-friendly, easy to access modules. While this application modeling process is designed to be carried out by programmers, we also discussed how tools can be applied to help clarify ambiguous application behavior and automate routine work. The use of application models provides a common language and medium that programmers, performance tuning specialists, and programming tools all can access.

This explicit use of application and machine models is an innovative approach to performance tuning. Most compilers form intermediate representations of applications and analyze application performance with simplified machine models (if not explicitly, then at least implicitly through their defined heuristics). Our model-driven approach provides a paradigm for compilers to extend their capability by interacting with programmers, performance tuning specialists, and programming tools, and by performing more sophisticated analyses.

With our model-driven simulator, we can carry out numerous analyses of machine-application interactions that are difficult or impossible to do on real machines. Several key tech-

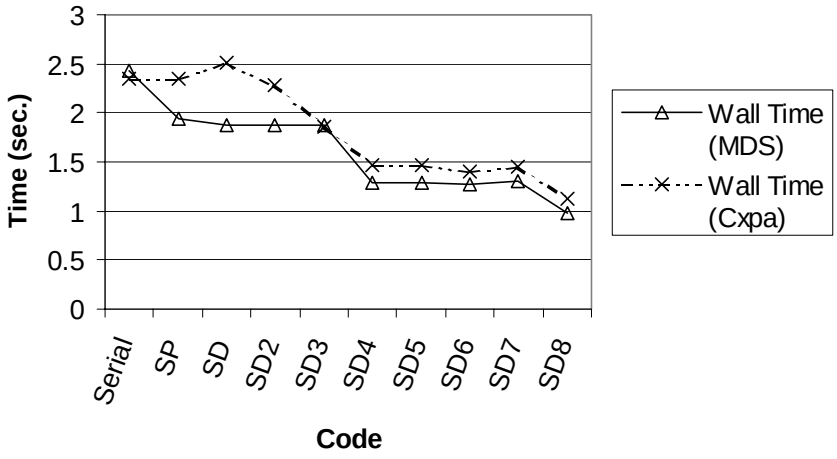


Figure 5-47: Summary of the Performance of Various CRASH Versions.

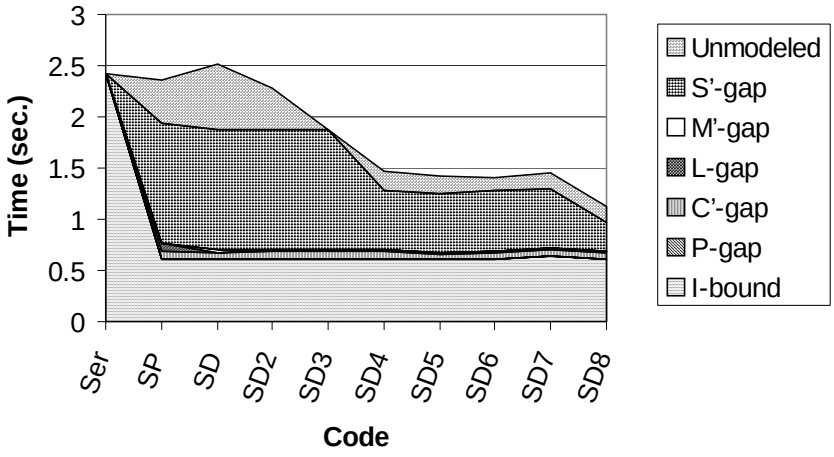


Figure 5-48: Performance Gaps of Various CRASH Versions.

niques and tools that have been widely used in the Parallel Performance Project have been integrated into MDS (e.g. K-LCache, CXBound,...) or interfaced with MDS (e.g. Dinero, CIAT/ CDAT,...). With this rich set of tools and techniques and its simple, yet flexible model description language, AML, MDS provides a variety of system-level and global event counts and performance metrics that are more sophisticated than those provided by existing performance assessment tools. In our preliminary case study, we showed that MDS analyzes data flow and

data working sets, exposes required communications, detects false sharings, and calculates load imbalances as well as performance bounds, and illustrated how the performance information provided by MDS is useful for guiding performance tuning.

While adding more performance assessment features to MDS seems quite straightforward, our ultimate goal of automating the performance tuning process requires extensive further research effort. With MDS and its application models, we have substantially reduced the difficulty of integrating existing techniques and carrying out performance tuning in a new and powerful application development environment.

CHAPTER 6. CONCLUSION

Performance tuning is very important to many real-time scientific/engineering applications as state-of-the-art compilers often fail to adequately exploit the peak performance and scalability of parallel computers. Very few application developers or computer architects are capable and willing to spend so much time in this tedious performance tuning process. Thousands of hours spent on hand tuning parallel applications have motivated us to search for practical and effective solutions to improve current application development environments.

In Chapter 1, we presented an overview of modern parallel architectures, typical current parallel application development environments and their weaknesses, and major problems that can result in significant gaps between the peak machine performance and the delivered performance on these machines. The delivered performance can often be improved by tuning the applications, and the secret of performance tuning lies in having an intimate knowledge of the machine-application pair and using that knowledge to achieve a proper orchestration of the machine-application interactions.

To understand the behavior of the machine-application pair, a fairly complete and accurate assessment of the delivered performance is necessary. In Chapter 2, we discussed the performance characterization of modern parallel machines, problems that can affect their delivered performance, important machine-application interactions that are related to these problems, and techniques to expose these interactions, assess performance problems, and gain insights regarding the machine-application pair. We also presented several innovative techniques that allow the memory traffic and communication patterns in large applications to be effectively exposed and analyzed in distributed shared memory systems via trace-driven simulation.

What often drives many programmers away from parallel computing today is the complexity of the performance tuning process required to develop a reasonably efficient parallel

code. Given an example complex as a full vehicle crash simulation, it can take researchers years to fine-tune the code for one representative data input and machine configuration. In Chapter 3, we presented a general performance tuning scheme that can be used for systematically applying a broad range of performance tuning actions to solve major performance problems in a well-ordered sequence of steps. The discussion in this chapter covers numerous performance issues, the interrelationship of these performance issues, and the positive, as well as negative effects of performance tuning actions. This innovative performance tuning scheme, along with the intuitions presented in the discussion and several new performance tuning techniques, provides an important new paradigm that unifies the performance tuning processes and reduces the complexity and mystery of the overall process.

To further guide programmers through the performance tuning process, we have successfully extended and automated the hierarchical performance bounds methodology that was previously developed in the Parallel Performance Project. In Chapter 4, we described an extended performance bounds hierarchy that matches our systematic performance tuning scheme, a tool (CXBound) that automatic generates parallel bounds on HP/Convex Exemplar, and case studies to show the effectiveness of this methodology in assessing and visualizing application performance for programmers. Our hierarchical performance bounds methodology is one of the most comprehensive and effective tools to date for assessing parallel application performance.

In Chapter 5, we proposed the use of application models to further reduce the complexity of a performance tuning process. We described an application modeling process that creates intermediate representations for abstracting performance-related application behavior. Given a machine model, our model-driven simulator (MDS) exposes important machine-application interactions and assesses the delivered performance; it is thus highly useful for conducting performance tuning with application models described above. These innovations in model-driven performance tuning should provide an efficient mechanism for reducing the complexity and cost of performance tuning of parallel applications and specifying the designs of future parallel machines in an application-sensitive manner.

In summary, the major contributions of this dissertation are: (1) a performance tuning paradigm that systematically addresses important performance problems, (2) a goal-directed scheme that guides performance tuning with hierarchical performance bounds analysis, and

(3) a model-driven methodology that eases the performance tuning process by quickly estimating the results of tuning actions via model-driven simulation.

In this dissertation, although we have contributed numerous insights and various innovations for optimizing parallel applications, we feel that this is only one more step toward fully understanding and solving an extremely subtle and complex problem. This dissertation unifies a range of research work done within the Parallel Performance Project at the University of Michigan; it also provides a solid foundation for us to pursue the remaining unsolved aspects of this problem. Much work remains to be done for further formalizing, automating, and optimizing the approaches we have developed. To further enhance our application development environment, we are interested in learning, developing, and integrating new techniques that would help us assess, tune, and model machine-application performance. We are currently extending our goal-directed and model-driven tuning methodology to experiment with automatic/dynamic performance tuning as well as advanced computer architecture designs. Eventually, we would like to integrate the techniques developed in this research into future compilers.

REFERENCES

[1] Ian Foster. *Design and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley, 1994.

[2] Karen A. Tomko. *Domain Decomposition, Irregular Applications, and Parallel Computers*. Ph.D Thesis, The University of Michigan, 1995.

[3] M. J. Flynn. Very high-speed computing systems. In *Proc. IEEE 54:12*. pages 1901-1909, December, 1966.

[4] IEEE Computer Society. *IEEE Standard for Scalable Coherence Interface (SCI)*, IEEE Standard 1596-1992, Aug. 1993.

[5] T. Asprey, G. Averill, E. Delano, R. Mason, B. Weiner, and J. Yetter. Performance features of the PA7100 microprocessor. In *IEEE Micro*, pp. 22-34, June 1993.

[6] Kenneth K. Chen, Cyrus C. Hay, John R. Keller, Gorden P. Kurpanek, Francis X. Schumacher, and Jason Zheng. Design of the HP PA 7200 CPU, *Hewlett-Packard Journal*, Feburary 1996.

[7] Doug Hunt. Advanced performance features of the 64-bit PA-8000, In *Digest of Papers, COMPCON '95*, pp. 123-129, March 1995.

[8] Convex Computer Corp. *Convex Exemplar SPP1000-Series Architecture*. 4th Ed., HP Convex Technology Center, May 1996.

[9] T. Brewer and G. Astfalk. The evolution of the HP/Convex Exemplar. In *Digest of papers, COMPCON'97*, pp. 81-86, Feb. 1997.

[10] Gheith A. Abandah and Edward S. Davidson, *Characterizing Shared Memory and Communication Performance: A Case Study of the Convex SPP-1000*, Technical Report, Dept. of Electrical Engineering and Computer Science, The University of Michigan, Jan 1996.

[11] Gheith A. Abandah. *Characterizing Shared-Memory Applications: A Case Study of the NAS Parallel Benchmarks*. Technical Report HPL-97-24, HP Laboratories, January 1997.

[12] Gheith A. Abandah and Edward S. Davidson. Characterizing shared memory and communication performance: a case study of the Convex SPP-1000, To appear in *IEEE Transcations of Parallel and Distributed Systems*.

[13] Gheith A. Abandah and Edward S. Davidson. Effect of architectural and technological advances on the HP/Convex Exemplar's memory and communication performance, To

appear in *Proc. 25nd Ann. International Symposium on Computer Architecture*, June 1998.

[14] Gheith A. Abandah and Edward S. Davidson. Modeling the communication performance of the IBM SP2. In *Proc. 10th International Parallel Processing Symposium*, April 1996.

[15] Eric L. Boyd, Gheith A. Abandah, Hsien-Hsin Lee, and Edward S. Davidson. *Modeling Computation and Communication Performance of Parallel Scientific Applications: A Case Study of the IBM SP2.* Technical Report CSE-TR-236-95, The University of Michigan, Ann Arbor, May 1995.

[16] Theodore B. Tabe, Janis P. Hardwick, Quentin F. Stout. Statistical analysis of communication time on the IBM SP2, In *Computing Science and Statistics* **27**, pp. 347-351, 1995.

[17] K. Li. IVY: A shared virtual memory system for parallel computing. In *Proc. International Conference on Parallel Processing*. pages 94-101, 1988

[18] Steven K. Reinhardt, James R. Larus, David A. Wood. Typhoon and tempest: user-level shared memory, *ACM/IEEE International Symposium on Computer Architecture*, April 1994.

[19] *High Performance FORTRAN Language Specification*. Technical Report, Rice University, 1993.

[20] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1995.

[21] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience, 2(4):315-339*, 1990.

[22] J. R. Allen and K. Kennedy. Automatic Translation of Fortran Programs to Vector Form. In *ACM Trans. Programming Languages and Systems*, Vol. 9, No. 4, Oct. 1987.

[23] Constantine Polychronopoulos, Milind B. Girkar, Mohammad R. Haghighat, Chia L. Lee, Bruce P. Leung, Dale A. Schouten. The structure of parafrase-2: an advanced parallelizing compiler for C and Fortran. *Languages and Compilers for Parallel Computing*, MIT Press, 1990

[24] S. P. Amarasinghe, J. M. Anderson, M. S. Lam and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February, 1995.

[25] Kendall Square Research. *KSR Fortran Programming.* July 1993.

[26] Convex Computer Corp. *CONVEX Fortran User's Guide.* Oct. 1994.

[27] Convex Computer Corp. *CONVEX Fortran Language Reference, 11th Ed.* Convex Press, Oct. 1994.

[28] Bruce Hendrickson and Robert Leland. *The Chaco User's Guide Version 2.0.* Technical Report SAND94-2692, Sandia National Laboratory, Albuquerque, NM, July 1995.

[29] George Karypic and Vipin Kumar. *METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System Version 2.0*. Technical Report, The University of Minnesota, 1995.

[30] Karen A. Tomko and Edward S. Davidson. Profile driven weighted decomposition. In *Proc. 1996 ACM International Conference on Supercomputing*, May 1996.

[31] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, vol. C-30, No. 7, pages 478-490, July 1981.

[32] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. In *Software-Practice and Experience*, Vol. 21, pages 1301-1321, Dec. 1991.

[33] CONVEX Computer Corp.,*ConvexMLIB User's Guide: LAPACK*, Document No. 720-005630-003, June 1994.

[34] Applied Parallel Research. *FORGE Shared Memory Parallelizer (spf) User's Guide 2.0*.

[35] Convex Computer Corp., *CONVEX CXpa Reference*, 2nd Ed., Convex Press,1994.

[36] IBM. *Program Visualizer (PV) Tutorial and Reference Manual*. Feb. 1995.

[37] Eric L. Boyd, John-David Wellman, Santosh G. Abraham, and Edward S. Davidson. Evaluating the communication performance of MPPs using synthetic sparse matrix multiplication workloads. In *Proceedings of the International Conference on Supercomputing*, pp. 240-250, Tokyo, Japan, November 93.

[38] Gheith A. Abandah. *Tools for Characterizing Distributed Shared Memory Applications*. Technical Report HPL-96-157, HP Laboratories, December 1996.

[39] William H. Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. A Performance Comparison of the IBM RS/6000 and the Astronautics ZS-1. *IEEE Computer*, Vol 24(1), pp 39-46, January 1991.

[40] William H. Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. Architectural vs. Delivered Performance of the IBM RS/6000 and the Astronautics ZS-1. *Proc. Twenty-Fourth Hawaii International Conference on System Sciences*, pp 397-408, January 1991.

[41] Eric L. Boyd and Edward S. Davidson. Communication in the KSR1 MPP: performance evaluation using synthetic workload experiments. In *Proc. 1994 International Conference on Supercomputing*, pages 166-175, July 1994.

[42] Waqar Azeem. *Modeling and Approaching the Deliverable Performance Capability of the KSR1 Processor*. Technical Report CSE-TR-164-93, The University of Michigan, Ann Arbor, June 1993

[43] Gheith A. Abandah. *Reducing Communication Cost in Scalable Shared Memory Systems*. Ph.D. Dissertation, Technical Report CSE-TR-362-98, Department of EECS, University of Michigan, April 1998.

[44] Eric L. Boyd, Waqar Azeem, Hsien-Hsin Lee, Tien-Pao Shih, Shih-Hao Hung, and Edward S. Davidson. A hierarchical approach to modeling and improving the perfor-

mance of scientific applications on the KSR1. In *Proceeding of the 1994 International Conference on Parallel Processing*, Vol. III, pp. 188-192, 1994.

[45] Tien-Pao Shih. *Goal-Directed Performance Tuning for Scientific Applications*. Ph.D Dissertation, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan, June 1996.

[46] William H. Mangione-Smith. *Performance Bound and Buffer Space Requirements for Concurrent Processors*. Ph.D. Thesis (Technical Report CSE-TR-129-92), The University of Michigan, Ann Arbor, 1992.

[47] Eric L. Boyd and Edward S. Davidson. Hierarchical Performance Modeling with MACS: A Case Study of the Convex C-240. *Proceedings of the 20th International Symposium on Computer Architecture*, pp 203-212, May 1993.

[48] Eric L. Boyd. *Performance Evaluation and Improvement of Parallel Applications on High Performance Architectures*. Ph.D dissertation, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, 1995.

[49] William H. Mangione-Smith, Tien-Pao Shih, Santosh G. Abraham, and Edward S. Davidson. Approaching a machine-application bound in delivered performance on scientific code. *Proceedings of the IEEE: Special Issue on Computer Performance Evaluation*, 81(8):1166-1178, Aug. 1993.

[50] Tien-Pao Shih and Edward S. Davidson. Grouping array layouts to reduce communication and improve locality of parallel programs, In *1994 International Conference on Parallel and Distributed Systems*, pages 558-566, Hsinchu, Taiwan, R.O.C., December 1994.

[51] Karen A. Tomko and Santosh G. Abraham, Data and program restructuring of irregular applications for cache-coherent multiprocessors, In *1994 Proc. International Conference on Supercomputing*, pages 214-255, Manchester, England, July 1994.

[52] Daniel Windheiser, Eric L. Boyd, Eric Hao, Santosh G. Abraham, Edward S. Davidson. KSR1 Multiprocessor: Analysis of Latency Hiding Techniques in a Sparse Solver. In *Proc. 7th International Parallel Processing Symposium*, Newports Beach, California, April, 1993.

[53] Alexandre E. Eichenberger and Edward S. Davidson. Efficient Formulation for optimal modulo schedulers, In *Proc. Conference on Programming Language Design and Implementation*, June 1997.

[54] Alexandre E. Eichenberger. *Modulo Scheduling, Machine Representations, and Register-Sensitive Algorithms*, Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, University of Michigan, December 1996.

[55] Alexandre E. Eichenberger and Edward S. Davidson. Register allocation for predicated code, In *Proc. 28th Annual International Symposium on Microarchitecture*, pp 180-191, November 1995.

[56] Waleed M. Meleis and Edward S. Davidson. Optimal local register allocation for a multiple-issue machine, In *Proc. International Conference on Supercomputing*, pp 107-116, July 1994.

[57] Alexandre E. Eichenberger and Santosh G. Abraham. Modeling load imbalance and fuzzy barriers for scalable shared-memory multiprocessors, In *Proc. 28th Hawaii International Conference on System Sciences*, pp 262-271, January 1995.

[58] John Nguyen, *Compiler Analysis to Implement Point-to-Point Synchronization in Parallel Programs*, Ph.D Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1993.

[59] R. Saavedra, R. Gaines, and M. Carlton. Micro benchmark analysis of the KSR1. In *Supercomputing*, pp. 202-213, November, 1993.

[60] J. P. Singh, W. Dietrich-Webber, and A. Gupta. *Splash: Stanford Parallel Application for Shared-Memory*. Technical Report. 596, Stanford, April 1991.

[61] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh,, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations, In *Proc. 22nd Ann. International Symposium on Computer Architecture*, pp.24-36, 1995.

[62] D. Bailey, et al. *The NAS Parallel Benchmark*. Technical Report RNR-94-07, NASA Ames Research Center, March 1994.

[63] A. Nanda and L. M. Ni. Benchmark workload generation and performance characterization of multiprocessors, In *Supercomputing '92*, pp. 20-29, 1992.

[64] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.

[65] Tien-Fu Chen. *Data Prefetching for High-Performance Processors*. Ph.D dissertation, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, July 1993.

[66] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed Sameh. *The Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design*. Technical report, Center for Supercomputing Research and Development, University of Illinois, September 1987, CSRD Rpt. No. 625.

[67] Michael Wolfe. Optimizing supercompilers for supercomputers. *Research Monographs in Parallel and Distributed Computing*. The MIT Press, Cambridge, Massachusetts, 1989.

[68] S. L. Graham, P. B. Kessler, and M. K. McKusock. Gprof: a call graph execution profiler. In *Proc. 1982 SIGPLAN Symp. Compiler Construction*, pages 120-126, June 1982.

[69] W. Y. Chen. *Data preload for superscalar and VLIW processors*. Ph.D thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, Illinois, 1993.

[70] S. McFarling and J. L. Hennessy. Reducing the cost of branches. In *Proc. 13th Annual International Symposium Computer Architecture*. Tokyo, Japan, pages 396-403, June 1986.

[71] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. ACM Symp. on Theory of Computing*, pages 114--118. ACM Press, 1978.

[72] Michel Dubios and Faye A. Briggs. Effect of cache coherency in multiprocessors. In *IEEE Transactions on Computers*, Vol. C-31, No. 11, pages 1083-1099, November 1982.

[73] Per Stenstrom, A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, Vol. 23, No.6, June 1990, pp.12-24.

[74] Milo Tomasevic and Veljko Milutinovic (editors), *The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions*, IEEE Computer Society Press, 1993.

[75] Milo Tomasevic and Veljko Milutinovic (editors), *The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions*, IEEE Computer Society Press, 1993.

[76] *A User's Guide to PICL: A Portable Instrumented Communication Library*. Technical Report, ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, October 1990.

[77] Michael T. Heath and Jennifer E. Finger. *ParaGraph: A Tool for Visualizing Performance of Parallel Programs*. User's Guide, Oak Ridge National Laboratory, Oak Ridge, June 1994.

[78] J. C. Yan, S. R. Sarukkai, and P. Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit". *Software Practice & Experience*. April 1995. Vol. 25, No. 4, pages 429-461

[79] Convex Computer Corp. *CONVEX CXtrace User's Guide*, 1st Ed., Convex Press, March 1994.

[80] Rajiv Gupta. Synchronization and communication costs of loop partitioning on shared-memory multiprocessor systems. In *Proceedings of the International Conference on Parallel Processing*, pp. 23-30, 1989.

[81] Alexandre E. Eichenberger and Santosh G. Abraham. Impact of load imbalance on the design of software barriers. In *Proceedings of the International Conference on Parallel Processing*, volume II, pp. 63-72, 1995.

[82] J. S. Emer and D. W. Clark. A Characterization of Processor Performance in the VAX-11/780, in *Proc. of the International Symposium on Computer Architecture*, pp. 301-309, June 1984.

[83] James R. Larus, Efficient program tracing, *Computer*, pages 52,-61, IEEE, May 1993.

[84] Kendall Square Research. *KSR1 Principles of Operation*. 1992.

[85] Eric J. Koldinger, Susan J. Eggers, and Henry M. Levy, On the validity of trace-driven simulation for multiprocessors, In *Proc. 18th Ann. International Symposium on Computer Architecture*, pages 244-253, 1991.

[86] Anoop Gupta and Wolf-Dietrich Weber, Cache invalidation patterns in shared-memory multiprocessors, *IEEE Transactions on Computers*, pages 794-810, Vol. 41, No. 7, July 1992.

[87] J-D Wellman and E. S. Davidson. The Resource Conflict methodology for Early-Stage Design Space Exploration of Superscalar RISC Processors, In *Proceedings of the 1995 International Conference on Computer Design* , Oct 2-4, 1995, pp. 110-115.

[88] J-D Wellman. *Processor Modeling and Evaluation Techniques for Early Design Stage Performance Comparison*, Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan, 1996.

[89] Andrew W. Wilson Jr. Multiprocessor cache simulation using hardware collected address traces, In *Proc. 23rd Annual Hawaii International Conference on System Sciences*, Vol. I, pages 252-260, IEEE Computer Society Press, 1990.

[90] Mark D. Hill, Dinero IV Trace-Driven Uniprocessor Cache Simulator, http:// www.cs.wisc.edu/~markhill/DineroIV, January 1998.

[91] Sanjay J. Patel, Marius Evers, and Yale N. Patt, Improving trace cache effectiveness with branch Promotion and trace packing, In P*roceedings of the 25th International Symposium on Computer Architecture*, Barcelona, June 1998.

[92] D.M. Tullsen, S.J. Eggers, and H.M. Levy, Simultaneous multithreading: maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pp. 392-403, June 1995.

[93] Edward S. Tam and Edward S. Davidson. *Early Design Cycle Timing Simulation of Caches.* Technical Report CSE-TR-317-96, University of Michigan, 1996.

[94] D. Burger, T. Austin, and S. Bennett. *The SimpleScalar tool set, version 2.0*. Technical Report #1342, University of Wisconsin - Madison Technical Report, June 1997.

[95] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. *PROTEUS: A High-Performance Parallel-Architecture Simulator*. Technical Report MIT/LCS/TR-516, MIT, September, 1991.

[96] S. R. Goldschmidt and H. Davis. *Tango Introduction and Tutorial*. Technical Report CSL-TR-90-410, Stanford University, 1990.

[97] Stephen Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance*. Ph.D. Thesis, Stanford University, June 1993.

[98] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer Simulation: The SimOS Approach, In *IEEE Parallel and Distributed Technology*, Fall 1995.

[99] *KSR1 Technical Summary*, Kenall Square Research Corporation, Waltham, MA, 1992.

[100] Rabin A. Sugaumar and Santosh Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 24-35, Santa Clara, California, May 1993.

[101] Craig B. Stunkel, Bob Janssens and W. Kent Fuchs. Address tracing for parallel machines. *Computer*, Vol.24, No.1, Jan. 1991, pp. 31-38.

[102] Hsien-Hsin Lee and Edward S. Davidson. *Automatic Parallel Program Conversion from Shared-Memory to Message-Passing.* Technical Report CSE-TR-263-95, Department of Electrical Engineering and Computer Science, University of Michigan, October, 1995.

[103] Chau-Wen Tseng, Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam, Unified compilation techniques for shared and distributed address space machines, In *Proc. 1995 International Conference on Supercomputing*, pages 67-76, Barcelona, Spain, July 3-7, 1995.

[104] Horst Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135-148, 1991.

[105] George Karypis and Vipin Kumar. *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*, Technical Report TR 95-035, Dept. Computer Science, University of Minnesota, 1995.

[106] Robert Leland and Bruce Hendrickson. An empirical study of static load balancing algorithms. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 682-685, 1994.

[107] Jude A. Rivers and Edward S. Davidson, *Sectored Cache Peformance Evaluation: A Case Study on the KSR-1 Data Subcache*, Technical Report CSE-TR-303-96, University of Michigan, September 1996.

[108] Milo Tomasevic and Veljko Milutinovic, Hardware solutions for cache coherence in shared-memory multiprocessor systems, In *The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions*, pages 57-67, IEEE Computer Society Press, 1993.

[109] A. Gupta, J. L. Hennessy, K. Gharachorloo, T. Mowry, and W.D. Weber, Computative evaluation of latency reducing and tolerating techniques, In *Proc. 18th Annual International Symposium on Computer Architecture*, pages 254-263, Toronto, May 1991.

[110] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, Techniques for reducing consistency-related information in distributed shared memory systems, *ACM Transactions on Computer Systems*, pages 205-243, Vol. 13, No. 3, August 1995.

[111] David Culler, Jaswinder P. Singh, and Anoop Gupta. *Parallel Computer Architecture - Hardware Software Interactions*, Alpha Draft, Morgan Kaufmann Publishers, 1997.

[112] Guang R. Gao, Lubomir Bic, and Jean-Luc Gaudiot. *Advanced Topics in Dataflow Computing and Multithreading*, IEEE Computer Science Press, 1995.

[113] *MPI: A Message-Passing Interface Standard.* The Message Passing Interface Forum (MPIF), June 1995.

[114] Ten H. Tzen and Lionel M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transcations on Parallel and Distributed Systems*, pp. 97-98, March 1993.

[115] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transcations on Computers*, C-36(12):1425-1439, December 1987.

[116] Hsien-Hsin Lee and Edward S. Davidson. *Automatic Generation of Performance Bounds on the KSR1.* Technical Report CSE-TR-256-95, The University of Michigan, August 1995.

[117] F. H. McMahon. *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*, Technical Report UCRL-5357, Lawrence Livermore National Laboratory, December, 1986.

[118] J. C. Yan and S. R. Sarukkai. Analyzing parallel program performance using normalized performance indices and trace transformation techniques". *Parallel Computing. Vol. 22, No. 9,* November 1996. pages 1215-1237

[119] S. R. Sarukkai, J. C. Yan and M. Schmidt. "Automated Instrumentation and Monitoring of Data Movement in Parallel Programs". In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA. April 25-28, 1995. pages 621-630

[120] P. Mehra, B. VanVoorst, and J. C. Yan. "Automated Instrumentation, Monitoring and Visualization of PVM Programs". In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing.* San Francisco, CA. February 15-17, 1995. pages 832-837

[121] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.