Locality-Preserving Dynamic Load Balancing for Data-Parallel Applications on Distributed-Memory Multiprocessors

Pangfeng Liu
Chih-Hsuae Yang
Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi, Taiwan, R.O.C.
pangfeng@cs.ccu.edu.tw

Abstract

Load balancing and data locality are the two most important factors in the performance of parallel programs on distributed-memory multiprocessors. A good balancing scheme should evenly distribute the workload among the available processors, and locate the tasks close to their data so as to reduce communication and idle time.

In this paper, we study the load balancing problem of data-parallel loops with predictable neighborhood data references. The loops are characterized by variable and unpredictable execution time due to dynamic external workload. Nevertheless the data referenced by each loop iteration exploits spatial locality of stencil references. We combine an initial static BLOCK scheduling and a dynamic scheduling based on work stealing. Data locality is preserved by careful restrictions on the tasks that can be migrated. Experimental results on a network of workstations are reported.

1 Introduction

To achieve maximum performance, a parallel program must evenly distribute the workload among the available processors, and locate the tasks close to their data. If the workload is not balanced among processors, a heavily loaded processor may be busy executing tasks assigned to it while other processors sit idle. This will degrade the overall parallel speedup since the aggregated CPU computing power is not fully utilized. If tasks are not placed close to their data, then the processors will have to synchronize and communicate with one another to acquire the necessary data for computation.

In cases where the structure of the program is fixed and the execution time of tasks are known before program execution, static scheduling policies can be used to find appropriate assignment of tasks to processors and data to memories. In many cases however, dynamic policies for load balancing and locality management must be used to determine this assignment. The challenges of load balancing for dynamic problems can be summarized as follows.

1. The amount of computation required for each task can vary tremendously between tasks and may change dynamically during program execution, so it is not sufficient to map equal numbers of tasks among processors; rather, the work must be equally distributed among processors in a dynamic manner.
2. The data reference patterns may be irregular and dynamic; as they evolve, a good mapping must change adaptively in order to ensure good data locality.
3. The system load on the parallel machine may vary dynamically and is usually unpredictable. This is true for non-dedicated parallel machines, network of workstations and PC clusters, since a parallel process on such platforms not only has to compete with other parallel processes but also with other jobs submitted by the interactive users. A change of a single processor load can easily defeat a sophisticated strategy for task (and data) assignment, if we do not take this factor into consideration.

A good balancing scheme should address all these issues satisfactorily. However, these issues are not necessarily independent; there are often conflicts between load balancing and data locality. In this paper, we study the load balancing problem of data-parallel computations which exploit neighborhood data references. These computations are usually expressed by means of parallel loops characterized by iteration costs which are variable and often unpredictable, due to the dynamically changing external load. On the other hand, the data referenced by each loop iteration exploits spatial locality of stencil references (for example, regular 5-point stencils on regular grids, irregular n-point stencils on unstructured grids), so that boundary elements can be identified once a partitioning strategy is given, and optimizations such as message vectorization and message aggregation can be applied. A large number of applications fall into this category. For example, solving PDEs and ODEs using iterative methods.

Many load-balancing schemes work in an "active" way. For example, many loop partitioning algorithms actively estimate the load of each loop iteration, then try to find out a good way to distribute them. Also in many task scheduling method, the scheduling algorithms try to partition the task DAG. Be it partitioning the loop, or the data set, or the task graph, the load distributer must know the load information about every elements it wants to partition (loop iterations, the amount of computation on a data, or on a subtask) and make correct judgment accordingly. Unfortunately, if the load balancer cannot acquire accurate load information, it could make a wrong decision. As a result the system load information must be constantly monitored and updated.

One "passive" approach of load distribution is to put all the work in a job queue and let any idle processor grab a job from this pool and execute it. This approach is conceptually simple, and can be easily implemented on a shared memory environment. However, this approach may perform poorly due to lack of management for data locality – a processor may get tasks from the pool that do not share any data, or many tasks that require remote data. Another problem the job pool approach might have is that it uses a global data structure to store the jobs. The global queue requires a centralized control on the integrity of the data structure and will create "hot spots" in data access since every processor can only get job from a single processor. In addition, this would be difficult to implement in a distributed memory environment, in which processors can only receive remote data by message passing.

In this paper, we propose a passive scheduling system, WBRT, that achieves load balancing and preserves data locality at the same time. We combine static scheduling and dynamic scheduling such that initially data are BLOCK distributed to preserve data locality for stencil-type data references, while dynamic load balancing is activated only when load imbalance occurs during program execution. Furthermore, to avoid synchronization overhead required in dynamic scheduling based on centralized dispatcher, we employ a fully distributed scheduling policy that constantly monitors and updates the system load information. Furthermore, to preserve data locality during program execution, tasks are migrated only if they form a contiguous domain. As a result an overloaded processor transfers tasks and data in the way that preserves the BLOCK distribution as much as possible, so that subsequent communication for executing the program can be minimized. Finally, we also duplicate boundary elements (shadowing) between adjacent processors to avoid inter-processor communication for computation of boundary elements and also to improve vectorization of the loop body, hence reducing computation time of each processor.

The rest of the paper is organized as follows. Section 2 reviews some related works. Section 3 outlines the implementations of the WBRT system. Section 4 reports our experimental results on a network of UltraSPARC workstations, and Section 5 concludes.

2 Related Work

Many studies have been carried out on various dynamic load balancing strategies for distributed-memory parallel computing platforms. Rudolph and Polychronopoulos [9] implemented and evaluated share-memory scheduling algorithms in the iPSC/2 hypercube multicomputer. It was not until early 90's that load balancing algorithms for distributed-memory machines were reported in literature [5, 10, 4, 8, 2, 3]. Liu et. al. proposed a two-phase Safe Self-Scheduling (SSS) [5]. In the first (compile-time) phase, a subset of the loop iterations is distributed uniformly among the processors. In the second (run-time) phase, an idle processor sends task request to the scheduler. The scheduler then chooses and assigns dynamically a chunk of not yet executed iterations to each requesting processor. Distributed Saft Self-Scheduling (DSSS), a distributed version of SSS, is reported in [10]. The data is partitioned into small blocks of the same size and distributed with partial redundancy among all the processors. Unlike SSS, DSSS assigns the chunks of iterations to the processors that have the corresponding data. DSSS is further generalized in [4]. Plata and Rivera [8] proposed a two-level scheme (SDD) in which static scheduling and dynamic scheduling overlap. In SDD, data-locality is considered, and the scheduler tries to predict in advance the workload unbalance in order to obtain a better load-balance and to reduce the communication overhead. While the processors are executing their statically distributed workload, a dynamic redistribution of the rest of the workload is in action. A similar approach focusing on adaptive data placement for load balancing is reported in [6].

The difficulty of load balancing is in deciding whether work migration is beneficial or not. None of the above balancing strategies has addressed this issue however. The SUPPLE system [7] determines whether work migration is profitable by comparing current load with a machine-dependent threadhold. When migration is determined, an unloaded processor chooses a victim from overloaded processors using a round-robin policy. Upon receiving job request, a victim processor chooses the appropriate number of loop iteraions to be moved using a modified *Factoring* scheme.

The idea of work stealing is not new. Cilk [1, 11] is a multi-thread language with runtime support for dynamic load balancing. At runtime, an idle processor steals work from a *random* victim processor by migrating a task from the top of the job queue in the victim processor. On a shared memory environment Cilk reported good speedup for various applications. However, the randomized work stealing strategy may perform poorly for data-parallel applications, where data locality is a critical factor in code performance. Cilk does not make an attempt to analyze the profitability of work stealing either. A processor begins to steal works when it becomes idle, and work migration is always carried out whenever there are unprocessed jobs on overloaded processors.

3 WBRT System

WBRT system is a runtime environment for parallel programming on distributed networks. The main features of WBRT includes a high-level programming interface for array-based parallel programming, a partitioner and a WBRT scheduler for automatic data distribution and load balancing, and a communicator for data delivery and low-level message-passing over the network.

WBRT provides a global view of the data, in which the data structure is treated as a whole, with operators that manipulate individual elements and implicitly iterate over substructures. When a global array is instantiated, it creates a constituent local array on each processor. Whenever a kernel operator associated with global array is invoked, the operation is delegated to each local array. If communication is required, it is performed through the communicator.

Conceptually, WBRT array operations are decomposed into parallel tasks, which are initially distributed to available processors following the "owner-computes-rules". When the program starts execution, every processor self-schedules its own portion of the tasks, and when the need arises, tasks at processor boundaries are migrated among processors by work stealing technique.

3.1 Programming Model

Internally, WBRT implements an arrays based on the Single-Program-Multiple-Data (SPMD) model, in which every processor executes the same program operating on the portion of the array that are mapped to that processor. Since inter-processor communication may occur frequently during load balancing and actual program execution and the message start-up cost is usually high for network transmission, one-dimensional partition is adopted to minimize number of messages.

When WBRT initializes, it is given the size of the global array by the application. Then on each processor WBRT allocates a memory segment for this global data according to its current workload and memory usage, which results in a variable-size one dimensional block partitioning. After the global data structure is partitioned and mapped into local memory segments, each segment is partitioned into chunks consisting of a fixed number of adjacent elements of the global data. This chunk of data is

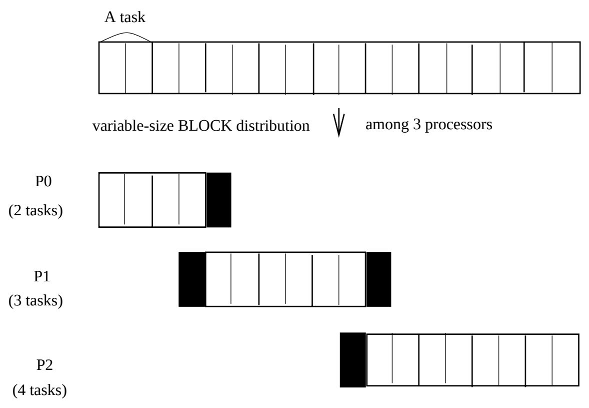

Figure 1: A mapping of data array to processors. Each task computes two data elements and the padding size is one.

called a *task* (Figure 1). The user code can retrieve a task from WBRT system, and perform operations in data-parallel fashion. The WBRT system provides data and the user provides the function that will be applied on the data.

3.2 WBRT API

WBRT provides a simple interface that the application program can retrieve the tasks and execute them. Figure 2 show how a sample code communicates with WBRT runtime system. First the application calls a WBRT initialization function `WBRT_init` collectively. The application specifies the size of the global array, the number of data in each task, the boundary width, and two function pointers that initializes and computes the data in a task (`InitData` and `DoTask` respectively). Finally the `WBRT_init` returns a WBRT *handler* by which the application can communicate with WBRT. Then the application calls `WBRT_Run` to start the execution, and `WBRT_Finalize` to finish.

A WBRT handler (`WBRT_H`) is the window that the application can communicates with the WBRT runtime system. The structure records detailed information of the runtime environment, including the geometry of the global array, the padding size, the task size. Also it keeps a list of pointers to those tasks that have not been done, so that they can be exported to other processors for load balancing purposes. Figure 3 shows the handler in details.

WBRT users can start the execution by calling `WBRT_Run`, whose default implementation is also

```
#include "WBRT.h"

#define D_ARRAY_SIZE 500
#define BOUNDARY 1
#define TASK_SIZE  5
int ARRAY_SIZE = 20000;

typedef struct{
  int org[D_ARRAY_SIZE];
  int res[D_ARRAY_SIZE] ;
} DATA;

/* Major WBRT interface */

void WBRT_Init(int argc, char **argv, int *array_size,
       int task_size, int boundary_prefetch,
               WBRT_H* rh, void(*INIT_DATA)(DATA*),
       void (*TASKFUNC)(Task*));
void WBRT_Run(WBRT_H *wrh);
void WBRT_Finalize();

/* User functions to manipulate the data in a Task */

void DoTask(Task *);
void InitData(DATA*);

int main(int argc, char *argv[])
{
  WBRT_H wrh;
  WBRT_Init(argc, argv, &ARRAY_SIZE, TASK_SIZE, BOUNDARY,
    &wrh, InitData, DoTask);
  WBRT_Run(&wrh);
  WBRT_Finalize();
}
```

Figure 2: A sample application using WBRT interface.

given in Figure 3. The `WBRT_Run` function repeatedly calls `WBRT_Gettask` to get a task for execution. This task returned by the WBRT may be a local task or a remote one that were stolen from other processors. In other words, the task stealing is transparent to the application and it does not need to know where the task came from. All the details of sending/receiving data associated with the migrating tasks are handled by WBRT.

3.3 Implementation Details

This subsection describes the details of the WBRT implementation. A WBRT execution consists of two threads – AP and RT running on every processor in the system. The AP is the user application thread and the RT is the runtime system thread. An AP can only request tasks from the corresponding RT on the same processor. RTs work together to handle all the low level details of work stealing and task migration, without the involvement of APs.

```
/* WBRT Handler */

typedef struct {
  int id;                 /* MPI processor ID */
  int proc_num;           /* MPI total processor number */
  pthread_t wbrt_thr_id;  /* Runtime system thread id */
  pthread_t app_thr_id;   /* Application thread id */

  int array_len;          /* Array total data element number */
  int task_len;           /* Task data element number */
  int boundary;           /* Boundary effective data number */

  int total_task_num;     /* Runtime total task number */

  DATA *data_array;       /* Data pointer array */
  char *dirty_array;      /* Dirty bit array */

  TaskList* finished_tasks;   /* Link list of finished tasks */
  TaskList* local_tasks;      /* Link list of local tasks */
  Task* working_task;     /* Pointer to current working task */

  void (*InitData)(DATA*);
  void (*DoTask)(Task*);  /* Function pointer of DoTask */
} WBRT_H;

Task *WBRT_Gettask(WBRT_H* wrh);

void WBRT_Run(WBRT_H *wrh)
{
  Task *task ;
  while((task = WBRT_Gettask(wrh)) != NULL)
    (wrh->DoTask)(task);
}
```

Figure 3: major data structures in WBRT and the default implementation of implementation of `WBRT_Run`.

Initial tasks assignment

At the beginning of execution WBRT distributes the workload according to the initial load of processors. First each processor test-runs a task to determine its current load, then the processors distribute all the tasks among themselves accordingly. If a processor is overloaded, it will be given less tasks. The load information obtained this way is most accurate so that actual workload, not simply the number of tasks, is evenly distributed. Formally let the load on the i-th processor P_i be $\frac{1}{L_i}$ and N_{all} be the total number of tasks. The number of tasks given to the i-th processor P_i, denoted by N_i, is given as follows.

$$N_i = N_{all} * (\frac{\frac{1}{L_i}}{\sum \frac{1}{L}}) \tag{1}$$

During the initial distribution of tasks to processors we also measure the network transmission time. This information is used to determine the communication overheads in sending a task to a remote processor, so that we can decide if it is worthwhile to perform work-stealing. We obtain this information at this stage for free since the data have to be transmitted to individual processor anyway.

Work Stealing

The most important function of WBRT is *work stealing*. When an AP requests work from a corresponding RT by `WBRT_Gettask`, the corresponding RT will give it a local task if one is available, otherwise the RT will try to steal a set of contiguous tasks from other processors. Note that these stolen tasks may not be adjacent to those in the thief processors. By forcing the processors to steal only contiguous tasks, we prevent the data fragmentation and extra overhead in both computation and communication. See Figure 4 for an illustration.

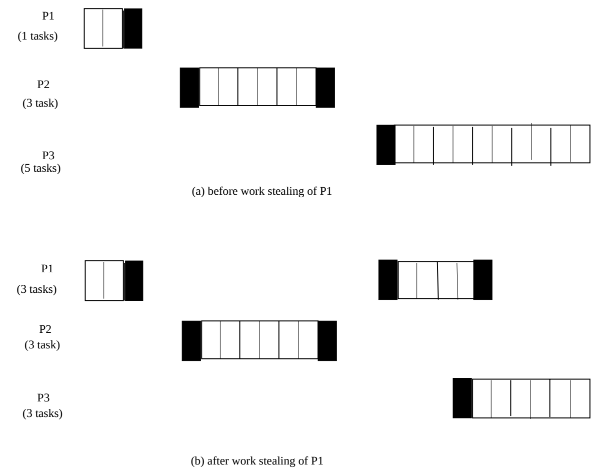

Figure 4: A mapping of 9 tasks to three processors. (a) and (b) show the situation before and after processor 1 steals one task from processor 2.

A processor must determine if it is underloaded before performing work stealing. Let S_i be the time for P_i to finish a task under current workload, R_i be the number of remaining tasks in P_i, then we define W_i to be the amount of time to P_i to finish its tasks.

$$W_i = R_i * S_i \tag{2}$$

We define that a processor P_i is underloaded if $W_i < k * W_{avg}$, where k is a constant that we can tune by experiments, and W_{avg} is the sum of all workload.

A underloaded processor locates its victim for task stealing by message passing. A underloaded processor sends a *request* messages to each of the other processors. When a processor receives this request, it will return an *reply* message, by which the requesting processor can determine if it wants to steal tasks from the replying processor.

A reply message from P_j consists of S_j, R_j, and T_j, namely the current computation cost per task, the number of remaining tasks, and the time for P_i to send a task to other processors. The requesting processor P_i uses these information to select the victim processor to steal tasks from. First P_i compares the W_j received with its own W_j, then consider only those processors that have more workload than itself as possible victims. In other words, an overloaded processor will never try to steal from a underloaded one. In addition, after P_i realizes that P_j has less workload, it *will not* send any further request message to P_j in order to reduce communication costs.

To determine the most suitable victim processor we define a *cost ratio* for a possible victim processor P_j.

$$C_j = \frac{T_j}{W_j} \tag{3}$$

This ratio indicates the relative cost of stealing tasks from P_j among other choices. A processor with a small C_j ratio is either overloaded or can send tasks to others very quickly, both indicate that it is a good candidate for work stealing. As a result, we compare C_j with a fixed threshold δ and ignore those processors that have high C_j values, since they are either underloaded or will incur high migration costs. From the remaining possible victim processors, we choose the one with the smallest C_j to steal from. However, if there is no possible victim, i.e. all the processors have C_j larger than δ, the requesting processors will *not* try to steal workload from others. We argue that under such a circumstance it will not be beneficial to migrate the tasks despite of load imbalance, since the communication overheads will be high.

To prevent unnecessary task migrations, a underloaded processor can issue "false" reply messages. For example, to prevent the tasks of a underloaded processor P_j from being stolen by another processor that has even less workload, P_j will issue reply message with R_j set to 0. In other words, when a processor decides that its workload is under a certain threshold and it will not be beneficial to migrate the tasks, it issues this message so that other processors will not bother it again. After issuing such reply, all the local tasks of P_j will stay in P_j until the execution ends.

After having located the victim processor, the underloaded processor will transfer tasks from the victim to itself. The overloaded processor makes sure the tasks that will be sent out form a *contiguous* block so that data locality is preserved, and only up to half of the tasks are allowed to be transferred. In addition, we restrict the number of contiguous blocks a processor domain can have. it will be impractical, if not impossible, to maintain a block partition at all times, since the most suitable victim processor for P, as far as the C cost is concerned, may not be adjacent to P. To distribute workload and maintain data locality simultaneously, we make the following comprise that each processor can have up to a small number of contiguous segments. If the number of segments in a processor P reaches the upper bound, any further tasks that P wants to steal must be adjacent to the existing segments in P. These restrictions reduce high communication costs in task migration, and data fragmentation due to work stealing.

To further reduce the communication cost we reduce the number of request messages being sent. When a remote processor return a high C_i value, the requesting processor notes the fact and will not send any further request for a given period of time. We argue that since P_j is now underloaded or having large communication costs, the situation will not improve in a very short period of time.

Boundary Padding and Direct Reference

WBRT also maintains a read-only padding (or shadowing) around the processor domain boundary. This size of this padding can be set during `WBRT_init`. This padding is maintained by WBRT to simplify the application code. Each task returned by `WBRT_gettask` is automatically padded by WBRT so that user code can access the data in the padding. In addition, when the boundary between two processors is changed, (e.g. due to work stealing), the padding is automatically adjusted by WBRT. The existing of padding is completely transparent to applications.

WBRT also supports direct access to remote data. User application can call `WBRT_request` to request a data in the global array by index. Similar to the padding, this access is read-only.

Synchronization

The scheduling between AP and RT is important. We implement AP and RT as two Pthreads in one processor. The RT will wake up every 500 micro second to see if there is anything it needs to do. If there is then it will do the work, possibly communicate with other RTs in order to finish it. Otherwise it will go back to sleep so that AP can use the CPU. There is a tradeoff between shorter response time and better CPU utilization by AP in picking the length of the sleep. We are working on a new version in which RT will wake up on demand. That is, RT will only wake up when it is interrupted by its AP or other RT. There will be extra programming effort and a careful investigation of the possible performance gain is needed.

4 Experimental Results

We design a series of experiments to evaluate the efficiency of WBRT. These experiments are conducted on a cluster of four Sun Dual UltraSPARC II workstations. Each Dual UltraSPARC II is running at 296 with 1Gega bytes of memory. All workstations run SunOS release 5.6, and we use mpich 1.1.2 for message-passing, and POSIX thread for multi-threading. The application is a graphic relaxation process that computes the value of every pixel as the average of its four neighbors. The computation domain is a N by 500 matrix where N is between 1000 and 10000.

WBRT runtime overheads

First we examine the overheads due to WBRT. We run the graphic relaxation code sequentially and compare the results with WBRT with/without workload stealing. Different problem sizes are tested. Figure 5 shows that the speedup on a four node cluster, with WBRT work stealing, is between 3.46 to 3.81, depending on the problem size, the speedup number improves. This high speedup number indicates that WBRT API only introduces a very small amount of overhead.

Effectiveness in load balancing

The second set of experiments examine if work-stealing can effectively balance the load on a real cluster system. We run the same relaxation code

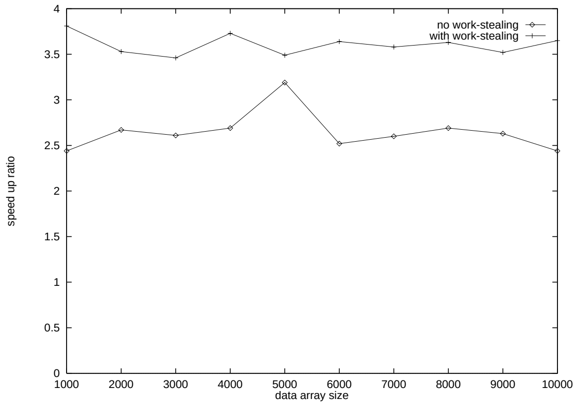

Figure 5: Speedup of the relaxation code with and without work stealing the relaxation code sequentially, on a cluster of four Dual UltarSPARC II workstations.

on the UltraSPARC cluster, and compare the timing from with and without work stealing. The cluster is located at Academia Sinica and is heavily used from time to time. Figure 6 shows the load on this cluster in a 24 hour period.

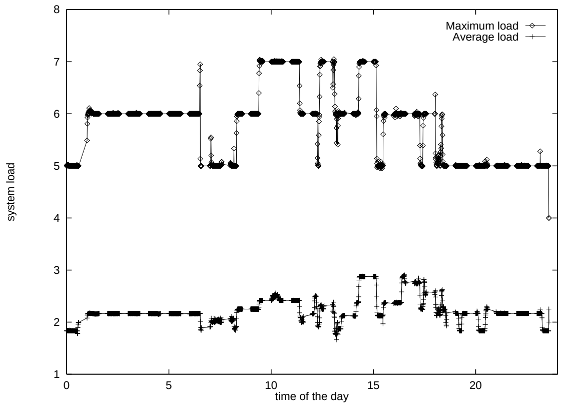

Figure 6: Maximum and average system load on a cluster of four Dual UltraSPARC II workstations during a 24 hour period.

On this cluster we run the relaxation code with WBRT. Each timeing data point is the average from ten test runs. The experiments show the the same code with work-stealing runs about 1.5 time as fast as without (Figure 7). This significant improvement indicates that WBRT does reduce the parallel execution time on a multiprocessor with dynamic external workload.

Figure 7 also indicates that the execution time with work stealing increases more smoothly than without. In a system with dynamic workload, the execution time can be greatly affected by the fluctuation of system load. Despite the fact that we have taken multiple samples in time and compute the average, we can still see the timing fluctuates slightly when work stealing is disabled. In contrast WBRT with work stealing gives much predictable, and also shorter, execution time.

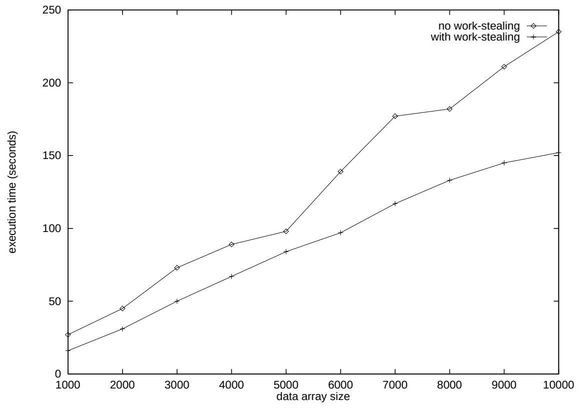

Figure 7: Timing results from running the relaxation code on four processors with and without work-stealing respectively.

5 Conclusion

In this paper we show that the simple technique of work-stealing improves parallel efficiency. The key observation is that by letting the idle processors steal tasks from busy processors, every processor can be busy all the time. Data locality is important to parallel execution efficiency. By restricting that tasks must be migrated as contiguous blocks, data locality can be preserved. This is as important as even distribution of workload, especially in a distributed memory multiprocessor.

Preliminary experiments show that WBRT work stealing effectively balances the load on a real cluster system. We run a relaxation code on a four node UltraSPARC II cluster, and report that the same code with work-stealing runs about 1.5 time as fast as without. This improvement indicates that WBRT does reduce the parallel execution time on a multiprocessor with dynamically changing external workload. In addition, WBRT with work stealing also gives much more predictable execution time than without.

References

[1] R. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, Nov 1994.

[2] M. Hamdi and C.-K. Lee. Dynamic load balancing of data parallel applications on a distributed network. In *ACM International Conference on Supercomputing*, pages 170–179, Barcelona, Spain, 1995.

[3] T. Y. Lee, C. S. Raghavendra, and H. Sivaraman. A practical scheduling scheme for non-uniform parallel loops on distributed-memory parallel computers. In *Proceedings of HICSS-29*, pages 243–250, Jan 1996.

[4] J. Liu and V. A. Saletore. Self-scheduling on distributed-memory machines. In *IEEE Supercomputing Conference*, pages 814–823, Nov. 1993.

[5] J. Liu, V. A. Saletore, and T. G. Lewis. Scheduling parallel loops with variable length iteration execution times on parallel computers. In *Proceedings of 5th IASTED-ISMM International Conference on Parallel and Distributed Computing and Systems*, Oct. 1992.

[6] D. K. Lowenthal and G. R. Andrews. Adaptive data placement for distributed-memory machines. In *Internationl Parallel Processing Symposium (IPPS96)*, Honolulu, Hawaii, 1996.

[7] S. Orlando and R. Perego. A support for non-uniform parallel loops and its application to a Flame simulation code. In *Irregular'97*, pages 186–197, 1997.

[8] O. Plata and F. Rivera. Combining static and dynamic scheduling on distributed-memory multiprocessors. In *the 1994 ACM International Conference on Supercomputing*, pages 186–195, July 1994.

[9] D. C. Rudolph and C. D. Polychronopoulos. An efficient message-passing scheduler based on guided self-scheduling. In *ACM International Conference on Supercomputing*, pages 50–61, July 1989.

[10] V. A. Saletore, J. Liu, and B. Y. Lam. Scheduling non-uniform parallel loops on distributed memory machines. In *IEEE International Conference on System Sciences*, pages 516–525, Jan. 1993.

[11] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.2 Reference Manual*, 1998.