

## LOAD BALANCING IN HOME-BASED SOFTWARE DSMS\*

WEISONG SHI<sup>†</sup> and ZHIMIN TANG

*Division of Computer Systems  
Institute of Computing Technology  
Chinese Academy of Sciences  
100080, P.O.Box 2704-25  
Beijing, P.R.China*

*Email: {weisong,tang}@water.chpc.ict.ac.cn*

Received (received date)

Revised (revised date)

Communicated by Editor's name

### ABSTRACT

Load balancing is a critical issue for achieving good performance in parallel and distributed systems. However, this issue is neglected in the research area of software DSMS in the past decade. Based on the observation that scientific applications can be classified into two categories: *iterative* and *non-iterative*, we propose two dynamic scheduling schemes for these two cases respectively in this paper. For iterative scientific applications, a dynamic task migration technique is proposed which characterizes itself with integrating computation migration and data migration together. An affinity-based self scheduling (ABS) is proposed for non-iterative scientific applications, which take both the static and dynamic processor affinity into consideration when scheduling. The target experiment platform is a state-of-the-art home-based DSM system named *JIAJIA*. Performance evaluation results show that the novel task migration scheme improves the performance ranging from 36% to 50% compared with a static task allocation scheme in a metacomputing environment, and performs better than traditional task (computation-only) migration approach about 12.5% for MAT, and 37.5% for SOR and EM3D. Higher resource utilization is achieved via the new task migration scheme too. In comparison with other loop scheduling schemes, the ABS achieves the best performance among all scheduling schemes in a metacomputing environment because of the reduction of synchronization overhead and the great improvement of waiting time resulting from load imbalance.

*Keywords:* Load Balancing, Dynamic Task Migration, Affinity-based Self Scheduling, Software DSM Systems

## 1. Introduction

---

\*The work of this paper is supported by the CLIMBING Program, and the National Natural Science Foundation of China under Grant No. 69896250-1.

<sup>†</sup>Present at Department of Computer Science, New York University, 715 Broadway Room 719, New York NY 10003, Tel:(212) 998-3493, FAX: (212) 995-4123, <http://www.cs.nyu.edu/~weisong>, [weisong@cs.nyu.edu](mailto:weisong@cs.nyu.edu)

*Load balancing* plays an important role in parallel and distributed systems in order to achieve good performance. This issue is even more critical in a meta-computing environment where the machines might be shared among many users. In order to maximize performance based on available resources, the parallel system must not only optimally distribute the work according to the inherent computation and communication demands of the application, but also according to the available computation resources dynamically. Within the context of dynamic load balancing scheme, the run time system needs an appropriate way to change the amount of work assigned to each processor. Dynamic task scheduling is an effective strategy to attack this problem, and it has been widely studied in the literature[20, 14, 19, 3, 10, 2, 22, 12, 23]. However, most of these previous work focus either on shared memory multiprocessors or on distributed systems. Furthermore, many evaluation environments are dedicated, which can not reflect the dynamic behaviors of real world.

According to the granularity of a task, previous work can be classified into three categories: *loop-level*, *thread-level*, and *process-level*. Loop-level and thread-level task migration schemes are widely used in shared memory environments, such as shared memory multiprocessors and software DSM systems. Whereas process migration is generally adopted in message passing environments, such as distributed or network operating systems. In many scientific applications, loops are the richest source of parallelism, therefore, changing the number of the loop iterations performed by each processor can balance the load. We will restrict our attention to this kind of program in this paper.

Based on the observation that scientific applications can be classified into two categories: *iterative*<sup>a</sup> and *non-iterative*<sup>b</sup>, we propose two dynamic scheduling schemes for these two cases respectively in this paper. For iterative scientific applications, a dynamic task migration technique is proposed which characterizes itself with integrating computation migration and data migration together. An affinity-based self scheduling (ABS) is proposed for non-iterative scientific applications, which take both the static and dynamic processor affinity into consideration when scheduling.

The experiment platform is a state-of-the-art home-based DSM system named *JIAJIA*[5]. Performance evaluation results show that the novel task migration scheme improves the performance ranging from 36% to 50% compared with a static task allocation scheme in a metacomputing environment, and performs better than traditional task (computation-only) migration approach about 12.5% for MAT, and 37.5% for SOR and EM3D. Higher resource utilization, 53% for MAT, 71% for SOR, 74% for EM3D, In comparison with other loop scheduling schemes, the ABS achieves the best performance among all scheduling schemes in metacomputing environment because of the reduction of synchronization overhead and the great improvement of waiting time resulting from load imbalance.

The rest of the paper is organized as follows. A overview of our target software

---

<sup>a</sup>Here, we use iterative applications to represent the application which has a sequential outer loop.

<sup>b</sup>Here, non-iterative application means there is no outer sequential loop in the application, but the application itself has many loop iterations.

DSM system JIAJIA is described in the following Section. In Section 3, a novel dynamic task migration scheme is proposed and evaluated for iterative applications. While a new affinity-based self scheduling scheme is designed and implemented in Section 4. Related work are presented in Section 5. Finally, conclusion remarks are listed in Section 6.

## 2. JIAJIA Software DSM System

JIAJIA[5] is characterized itself with a new lock-based cache coherence protocol and home-based memory organization scheme which combines the physical memories of multiple workstations to form a large shared address space. JIAJIA is implemented entirely as a user-level library and currently runs on many mainstream UNIX platforms. Multiple writer technique is employed to alleviate false sharing. JIAJIA implements the scope consistency (ScC) memory model[7].

The unique characteristic of JIAJIA is the use of home concept not only for data information, but also for coherence information, such as *write notices*. Generally, home-based software DSM systems[7] propagate data information and apply them at homes eagerly, but this information is fetched by others lazily. On the other hand, these systems propagate coherence information and apply them either eagerly or lazily. In our lock-based cache coherence protocol, coherence information is processed in a similar way to data information, and each coherence information has a static home according to the corresponding synchronization object (e.g., a lock or a barrier manager). So, coherence information is propagated to the corresponding home at release time, and is lazily fetched by the next acquirer of the same synchronization object. Compared with directory-based protocols, all coherence related actions in the protocol are applied at synchronization points. In this way, the lock-based protocol has least coherence related overheads for ordinary read or write misses. Moreover, the lock-based protocol is free from the overhead of maintaining the directory.

JIAJIA provides three types of synchronization mechanisms: lock, barrier, and condition variables. JIAJIA also supports home migration scheme based on the information of access patterns collected at run time.

## 3. Iterative Scientific Applications

### 3.1. Motivation

The performance of a loop scheduling algorithm is mainly affected by three overhead sources: *synchronization overhead*, *load imbalance*, and *remote data access latency*, where remote data access latency can be represented by *processor affinity*, which refers to certain data dependence of a task to a specific processor. One important context in which loop iterations have an affinity for a particular processor is when:

1. the same parallel loop will be nested within a sequential loop, therefore the same data has to be repeatedly used by successive execution of an iteration

(iterative applications satisfy this condition), and

2. the cache (in software DSM this means software cache) is large enough to hold the data until it can be reused.

Many iterative scientific computing applications have these characteristics, and we restrict our attention to this specific scenario of a parallel loop nested in an outer sequential loop here.

On the other hand, among all task migration schemes proposed in the past, however, a “task” means the corresponding code of computation, i.e., the data related to this computation is totally neglected. As such, when one task is migrated from processor A to another processor B, the data required by this task remains on processor A. Thus, the processor B has to perform remote communication when it executes this task, eliminating the advantage of task migration, or even making the performance worse. Hence, the definition of traditional “task” should be revisited. We define a task as follows:

**Task = Computation subtask + Data subtask**

Computation subtask is the program code to be executed, while the data subtask is the operations to access the related data located in memory. In fact, as the speed gap between processors and memory becomes larger and larger, the importance of data subtask becomes more obvious than before. Therefore, we argue that both subtasks should be migrated to a new processor during task migration.

### 3.2. Rationale of Dynamic Task Migration

For simplicity, we define one loop iteration and its corresponding data as a task here, and we assume the tasks are allocated statically with good processor affinity, i.e., both the computation and the data subtask are allocated as close as possible.

Since the execution time of each task is nondeterministic and the practical capacity of each processor can not predetermined in metacomputing environments, static task allocation schemes do not perform well. On the other hand, in iterative applications, the available computing power which is effected by the application and processor capacities can be obtained at run time. Accordingly, tasks can be reallocated to balance the remaining iterations, thus improving the resource utilization.

In dynamic task migration schemes, however, it is difficult to determine statically the relationship between computation and data subtasks. In general, only computation tasks are migrated in such scheme, while the data remain at the original processor, resulting in remote data access latency. For example, the dynamic scheme of Subramaniam and Eager[19] belongs to this category. We use *CompMig* to represent this scheme. Although the cache of remote processor will utilize data reuse, due to the limited cache size, the contents of the cache are replaced randomly, thus, yielding inevitably many remote data misses after computation migration. In our new migration scheme, not only the computation subtasks, but also the data subtasks are migrated to appropriate processors. This scheme is represented as *TaskMig*. In order to find the relationship between computation and data subtasks, the migration of data is delayed one iteration for every  $\alpha$  iterations. The

Figure 1: Basic framework of dynamic task migration scheme

```

for (i=0; i<N; i++) {
  execute task;
  synchronization;
  if (i %  $\alpha$  == 0) {
    collect computing power of each processor;
    computation subtask migration;
  }
  if (i %  $\alpha$  == 1) {
    data subtask migration;
  }
}

```

Table 1: Definition of the symbols

Symbol	definition
$t$	the time variable
$P$	the number of processors
$W$	total number of tasks
$W_i$	the workload of processor $i$
$T_i$	the execution time of last interval* of processor $i$
$P_i$	the available computing power of processor $i$
$W_i'$	the new workload of processor $i$ after task migration
$E(t)$	expected value of execution time $T_i$
$E(t^2)$	expected value of $T_i^2$
$\sigma(t)$	variance of execution time $T_i$
$\alpha$	migration step
$\beta$	significance parameter in migration decision

\*Here, the interval is delimited by two consecutive synchronization operations.

pseudo code of our new scheme is shown in Figure 1. In this framework,  $N$  is the total number of iterations and  $\alpha$  is the migration step, which determines the frequency of task migration. Currently, the value of  $\alpha$  is set to 1/10 of  $N$ .

In iterative applications, a global synchronization (i.e., barrier) is required at the end of each iteration to maintain coherence among all the processors. Though the synchronization operation and task migration are shown separately in Figure 1, we embed all the migration work in the synchronization operation in our implementation, so that the overhead of task migration is minimized to be negligible. Details about the implementation is presented in the following section.

### 3.3. Implementation

#### 3.3.1. Computation Migration

We augmented the global synchronization operation of the JIAJIA system to count  $T_i$ ,  $W_i$ <sup>c</sup> dynamically. Based on these two variables,  $P_i$  is defined as  $\frac{W_i}{T_i}$ . Ideally, the task can be reallocated according to  $P_i$ . However, experimental results show that a small change in the load might cause the system to oscillate and can be harmful for the overall application performance. So, we borrowed the idea of the *variance*, which is used to measure the degree of the deviation of random variable to *average expected value*, to decide whether the workload should be reallocated.

<sup>c</sup>Table 1 defines all the symbols used in this paper.

The expected value and the variance of execution time  $T_i$  is defined as follows:

$$E(t) = \frac{1}{P} \sum_{i=1}^P T_i \quad \text{and} \quad \sigma(t) = \sqrt{E(t^2) - E(t)^2} \quad (1)$$

Only when the ratio between  $\sigma(t)$  and  $E(t)$  is larger than a threshold  $\beta$ , we perform task migration. The parameter  $\beta$  varies for different applications and different environments. In our tests, the value of  $\beta$  was set to 0.2. After task migration, the workload  $W_j'$  of each processor  $j$  falls into the following range:

$$W_j' = \left[ \frac{\sum_{i=1}^{j-1} P_i}{\sum_{i=1}^P P_i} W \right] \sim \left[ \frac{\sum_{i=1}^j P_i}{\sum_{i=1}^P P_i} W \right]$$

This algorithm is implemented entirely in `jia_lbarrier(&begin, &end)` which is a variation of the original `jia_barrier()` function. The parameters `begin` and `end` represent the upper and lower bounds of the task allocated to the local (caller) processor, respectively. In order to reduce extra system overhead associated with the computation migration, the computation migration is called at every  $\alpha$  steps. If  $\alpha$  is too small, then the overhead becomes non-negligible. Otherwise, the dynamic environment will not be reflected accurately, and the performance will degrade. Therefore,  $\alpha$  is another key parameter in this migration scheme.

### 3.3.2. Data Migration

As shown in Table 1, data migration is called after computation migration is iterated once. During this iteration, the relationship between migrated tasks and their corresponding data is detected by run time system. According to this relationship, the related data are sent to the appropriate processors transparently.

The heuristic idea about relationship between data and computation is based on the observation that the computation task always processes (including read and write) the data exclusively and appears as a single writer in cache coherence protocols[1]. In the context of a single writer, the data can be migrated close to computation task so that all the future accesses will hit locally. This can be implemented in underlying cache coherence protocol automatically. However, we basically assumed that all the data and computation are allocated in home statically. Therefore, the migration of data necessitates the support of home migration, which has not been implemented in any previous home-based software DSM systems. As a matter of fact, we recently proposed a simple but efficient home migration scheme in the JIAJIA system[4]. With this migration scheme, all the data with single writer characteristic will migrate to the new home automatically, while the ease of locating data is kept. The data migration is implemented by a new function called `jia_config(HMIG, ON)`. Here, the first parameter indicates the home migration scheme, and the second parameter controls the beginning and the end of our home migration scheme.

In the current implementation, both migration schemes are embedded in the barrier operation at the end of each iteration, to reduce the system overhead.

### 3.4. Experimental Results and Analysis

#### 3.4.1. Experiment Platform

We conducted our evaluation on the Dawning-1000A parallel machine developed by the National Center of Intelligent Computing Systems of China. This machine has eight nodes, each with a PowerPC 604 processor and 256MB memory. The nodes are connected through a 100Mbps switched Ethernet. During the experiments, all libraries and applications (except a FORTRAN application) are compiled by `gcc` with the `-O2` optimization option. The metacomputing environment is assumed by adding artificial loads to some processors in order to have a fair comparison. For the experiments, we added 2, 1, 3, 1 loads to the second, fourth, sixth, and eighth processors, respectively.

#### 3.4.2. Applications

Three iterative applications were used to evaluate our task migration scheme: A locally developed inner-product of matrix multiplication (MAT) code, with matrix size of  $1024 \times 1024$  and repeated 100 times. SOR (Successive Over-Relaxation), with a matrix size of  $4096 \times 4096$  floats, and 100 iterations. SOR is an iterative method for solving partial differential equations, with nearest neighbor 5 point difference as the main computation. finally, a real application (EM3D) from Institute of Electronics, Chinese Academy of Sciences[17]. EM3D is the parallel implementation of Finite Difference Time Domain (FDTD) algorithm to compute the resonant frequency of a waveguide loaded cavity. The electronic and magnetic field components are updated alternatively in each iteration. Barriers are used for synchronization. The grids used in this test is  $60 \times 30 \times 832$ , and 100 iterations are run in our test, while the real run requires 12000 iterations.

#### 3.4.3. Performance Evaluation and Analysis

Figure 2(a) shows the relative performance, with the largest values listed on the top of the bar. For comparison, we listed the execution time in dedicated *Dedi* and metacomputing *Meta* environment without task migration on the left. The execution time with computation migration *CompMig* and task migration *TaskMig* schemes are listed on the right of each bar groups. The breakdown of system overhead under different schemes are shown in Figure 2(b). System overhead is divided into three parts: *Segv*, *Syn*, and *Serv*. *Segv* represents the data miss penalty, including local and remote misses. *Syn* and *Serv* represent the time spent on synchronization and servicing remote requests, respectively. Detailed analysis about the breakdown can be found in [16]. The leftmost bar reflects the system overhead of *Meta*, while the overhead of *CompMig* and *TaskMig* are shown as the middle and the right bars, respectively.

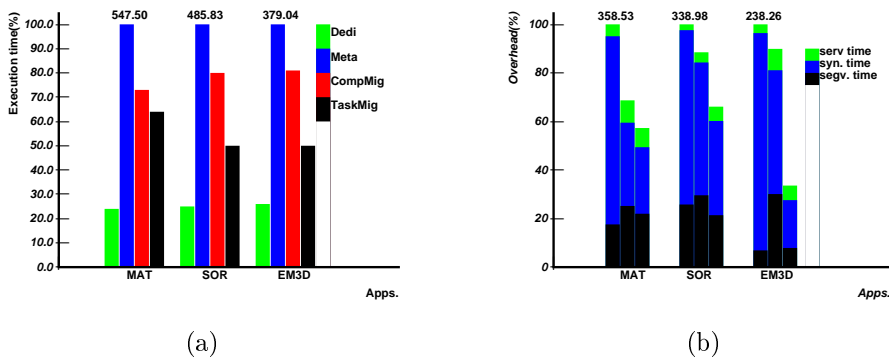


Figure 2: Performance comparison: (a) execution time, (b) system overhead

Table 2: System overheads in unbalanced environment

Overhead	1	2	3	4	5	6	7	8
Segv.	0.43	15.74	12.94	13.51	12.88	25.22	25.24	26.17
Syn.	288.82	123.94	<b>294.80</b>	211.81	291.04	<b>23.57</b>	284.35	187.90
Serv.	19.01	9.64	5.29	5.89	10.00	10.96	5.25	1.70

Figure 2(a) shows that *TaskMig* improves the performance significantly compared with *Meta*, ranging from 36% for MAT to 50% for SOR and EM3D. We ascribe this to the great reduction of waiting time at synchronization points, as shown in Figure 2(b).

The difference between *CompMig* and *Meta* manifests that the effectiveness of *CompMig* is limited and that it relies greatly on the granularity of the application. For coarse-grain applications, such as MAT, the induced remote data miss penalty of *CompMig* is compensated by the reduction of the long computation time. Hence, *CompMig* performs well. However, for relatively fine grain applications, such as SOR and EM3D, the remote data miss can not be compensated by computation, and thus, the performance of *CompMig* is not satisfactory. The results shown in Figure 2(b) confirm our analysis.

In comparison with *CompMig*, our new task migration scheme takes the data migration into account. As such, not only the synchronization overhead, but also the remote data miss penalty *Segv* are reduced significantly, as shown in Figure 2(b). So the total execution time of *CompMig* are improved about 12.5%, 37.5% and 37.5% for MAT, SOR, and EM3D, respectively.

In order to clarify the effectiveness of our task migration scheme, we now argue about the system overhead of all of the 8 processors using EM3D results. Table 2 lists the results in metacomputing environment without load balancing, while Table 3 shows the case with new task migration scheme.

It can be seen from Table 2 that the main difference is induced by synchronization. For example, *Syn* overhead of the slowest processor (#6) is 23.57 seconds, while in the fastest processor (#3) is 294.80 seconds. In other words, the faster

Table 3: System overheads in unbalanced environment with task migration

Overhead	1	2	3	4	5	6	7	8
Segv.	20.25	15.90	21.27	20.14	30.43	19.40	16.70	5.59
Syn.	36.59	<b>25.21</b>	58.84	39.94	50.13	26.01	<b>76.32</b>	62.94
Serv.	7.32	19.57	12.49	15.42	15.90	22.61	14.67	6.96

processor is idle for 76% due to the computing power imbalance. However, this problem is solved significantly with the new task migration scheme. As shown in Table 3, the gap between the maximal and minimal values of *Syn* is 51.11, i.e., only 19% with respect to the counterpart in metacomputing environment with static task allocation scheme. From the tables we find that the overhead incurred by *Segv* and *Serv* are slightly increased with new task migration method. This is because the data migration is performed one iteration later than the computation migration, and there were some remote data misses in that iteration. However, compared with the decrease in *Syn* overhead, this minor increase is negligible.

Furthermore, in order to evaluate the resource utilization, we define the efficiency  $E$  of the new task migration scheme as the ratio between the ideal execution time ( $T_{Ideal}$ ) and that of *TaskMig* scheme ( $T_{TaskMig}$ ). Since the total workload is fixed, we have the following equation:

$$T_{Ideal} \times Power' = T_{Static} \times Power \quad (2)$$

where  $Power$  represents the total computing power of 8 processors,  $Power'$  represents the available computing capacity of these processors. In our test environment,  $Power' = (1 + 1 + 1 + 1 + 0.5 + 0.5 + 0.33 + 0.25)/8 = 0.6975 \times Power^d$ . As such, the efficiency  $E$  is defined as follows:

$$E = \frac{T_{Ideal}}{T_{TaskMig}} = \frac{1.43 \times T_{Static}}{T_{TaskMig}} \quad (3)$$

Given the above, the efficiencies of *TaskMig* for three benchmarks are 53%, 71% and 74%, respectively. Therefore, we conclude that our new task migration scheme can achieve a satisfactory resource utilization as well.

## 4. Non-Iterative Applications

### 4.1. Motivation

Above all, only iterative applications are considered. In fact, for applications without an sequential loop outside the parallel loop, there are many different loop scheduling algorithms were proposed, such as Self Scheduling(SS) [20], Block Self Scheduling (BSS), Guided Self Scheduling (GSS)[14], Factoring Scheduling (FS) [6], Trapezoid Self Scheduling (TSS) [22], Affinity Scheduling (AFS) [12], Safe Self

<sup>d</sup>The available power of a processor with  $n$  loads will be  $1/(n + 1)$  of it's original power.

```

void worker()
{
    jia_lock(0);
    /* get the new boundary from master */
    /* update the global boundary variable */
    jia_unlock(0);
    if (there is left work) {
        /* computing */
    } else {
        /* exit of worker() */
    }
}

```

Figure 3: Framework of self scheduling algorithm for single iteration applications

Scheduling (SSS) [11], Adaptive Affinity Scheduling (AAFS) [23] etc.. For comparison, all of them are implemented in JIAJIA system. The basic framework of application is shown in Figure 3.

All those loop scheduling algorithms discussed above fall into two distinct classes: *central queue based* and *distributed queue based*. In central queue based algorithms, such as SS, BSS, GSS, FS, TSS, SSS, iterations of a parallel loop are all stored in a shared central queue and each processor exclusively grabs some iterations from the central queue to execution. The major advantage of using a central queue is the possibility of optimally balancing the load. While keeping a good load balance, these central queue based algorithms differ in the way of reducing loop allocation overhead. However, three limitations are associated with the use of these traditional central queue method<sup>e</sup>:

- An iteration in the central queue is likely to be dynamically allocated to execute on any processor since the order of each processor visiting this central queue is random, which does not facilitate the use of processor affinity<sup>f</sup>. In distributed memory systems processor affinity is more important than load balancing.
- During allocation, all the processors but one should remotely access the central work queue, and thereby generate heavy network traffic;
- Because all the processors contend for the central queue, the central queue tends to be a performance bottleneck, which results in a longer synchronization delay.

The limitation 2 and 3 can be solved by using distributed queue methods as AFS and AAFS. Furthermore, since the local work queue of AFS and AAFS assign the

<sup>e</sup>In order to distinguish those central queue methods from our new ABS scheme, we use traditional central queue method to represent them.

<sup>f</sup>*Processor Affinity* means the relationship between processor and the data set it processes. For example, if one task can be processed more efficiently on processor A than on processor B, due to facts such as the presence of required data in its local cache, the task is said to have an “affinity” with processors A.

same iterations statically for repeated execution loops, the data required by each processor will be cached or moved<sup>§</sup> to related processor so that the processor affinity can be used. However, the extra synchronization overhead associated with AFS and AAFS when load imbalance occurs are neglected in previous work. In distributed systems, this operation requires at least  $2P + 2$  messages and  $P + 1$  synchronization operations, where  $P$  is the number of processors. This operation requires much more time and will affect the final execution time greatly. On the other hand, the allocation overhead associated with each local allocation does not reduce at all in AFS and AAFS because the local queue must be accessed exclusively too.

Moreover, the processor affinity exploited in AFS and AAFS is limited to iterative applications only.

#### 4.2. Affinity-based Self Scheduling

The basic idea of affinity-based self scheduling (ABS) is to distribute the loop iterations to the processor that already has corresponding data while minimizing the loop allocation overhead (i.e., synchronization overhead to access exclusive shared variables). In traditional central queue based scheduling algorithms, the initial data distribution is separated from the scheduling algorithm so that the affinity between the initial data distribution and the following computing can not be exploited, since the order of each processor visiting the central queue is not deterministic.

ABS includes three steps: *static allocation*, *local dynamic scheduling*, and *remote dynamic scheduling*. We implement a globally shared central queue, which is partitioned into  $p$  segments evenly, and the  $i$ th segment is assigned to processor  $i$  statically according to the initial data distribution. At local dynamic scheduling phase, BSS or GSS scheduling algorithms can be employed<sup>h</sup>. When load imbalance occurs, the idle processor obtains the task from the most heavily loaded processor. It seems that there is no much difference between ABS and AFS at the later scheduling phases. However, the use of central shared queue, the lazy memory consistency model adopted by software DSM system, and the large granularity inherent in the software DSM system contributes greatly to our ABS scheduling algorithm. The large grain (generally one page is larger or equal to 4K bytes) causes the whole shared queue to be allocated in one page. Therefore there is no difference between local scheduling and remote scheduling, both of them require only one synchronization operation and one message. Table 4 compares the number of messages required and synchronization operations of ABS, AFS and AAFS. From the table, we conclude that when load imbalance occurs, more messages and synchronizations are required in AFS and AAFS to gather other processors' load information. We can also find that the synchronization overhead associated with the loop allocation of AAFS is worse than that of AFS, which will be demonstrated by the performance evaluation results in the next section.

The main difference between ABS algorithm and traditional central queue based

---

<sup>§</sup>Only systems supporting data migration can achieve this.

<sup>h</sup>Due to space limitation, we present the results of GSS here only, more evaluation about different schemes adopted in the local scheduling phase of ABS are presented in [18]

Table 4: # of messages and synchronization operations associated with loop allocation

Scheme	Local scheduling		Remote scheduling	
	# of message	sync.	# of message	sync.
AFS	0	1	$2p+2$	$p+1$
AAFS	$2p$	$p+1$	$2p+2$	$p+1$
ABS	0 or 2	1	4	2

scheduling schemes lies in the organization and management of the central queue. In our scheme, the queue is partitioned among these processors, and each processor first accesses it’s own part. In traditional central queue based scheduling algorithms, the initial data distribution is separated from the scheduling algorithm so that the affinity between the initial data distribution and the following computing can not be exploited at all.

### 4.3. Experiment Results and Analysis

#### 4.3.1. Applications

We choose five applications which cover the range of all applications used in the related literature. The selected applications includes SOR, Jacobi Iteration (JI), Transitive Closure (TC), Matrix Multiplication (MM), and Adjoint Convolution (AC). Details of these programs can be found in [23]. Table 5 illustrates the basic characteristics of these 5 applications. Among these 5 benchmarks, three of them are non-iterative applications, while the rest are iterative applications.

Table 5: Characteristics of the applications

Application	SOR	JI	TC	MM	AC
Size	4096, 10 iter.	5000, 10 iter.	$2048 \times 2048$	$2048 \times 2048$	$256^2$

For comparison, we add several artificial loads to some processors when testing. Figure 4 illustrates the execution time of different schemes under metacomputing environment. The loop allocation overhead is listed as the number of synchronization operations in Table 6. The number of remote getpages, which reflects the effect of remote data communication, is shown in Table 7. Here, the *Syn.* overhead includes two parts. One is the synchronization overhead associated with loop allocation operation, the other is the waiting time at synchronization point because of load imbalance.

#### 4.3.2. Analysis

Figure 4 illustrates the execution time of different schemes under metacomputing environment. Similar to the analysis in dedicated environment, the loop allocation overhead is listed as the number of synchronization operations in Table 6. The number of remote getpages, which reflects the effect of remote data communica-

Table 6: The number of synchronization operations of different scheduling algorithms in metacomputing environment

Apps.	Static	SS	BSS	GSS	FS	TSS	SSS	AFS	AAFS	ABS
SOR	0	41040	410	610	310	270	270	7003	14414	970
JI	0	50080	420	620	315	270	288	7056	7251	3130
TC	0	205600	2500	5500	3154	2700	2761	56924	123529	5788
MM	0	2056	41	55	31	27	26	1040	1565	227
AC	0	65544	4105	81	31	27	41	1411	2595	465

Table 7: The number of getpages of different scheduling algorithms in metacomputing environment

Apps.	Static	SS	BSS	GSS	FS	TSS	SSS	AFS	AAFS	ABS
SOR	8456	260838	67418	82892	77177	75212	81928	12091	10478	6215
JI	13820	182889	75719	70413	77531	75621	57852	9045	7251	4058
TC	2392	335924	54418	55878	82701	54153	22773	41511	49191	19129
MM	13547	33910	23843	26391	22965	21846	21935	17998	18198	17550
AC	100	58032	838	818	205	209	175	804	775	465

tion, is shown in Table 7. More intuitive effects of load imbalance, loop allocation, and remote data communication are shown as execution time in Table 8 and Table 9. Here, the *Syn.* overhead includes two parts. One is the synchronization overhead associated with loop allocation operation, the other is the waiting time at synchronization point because of load imbalance.

As analyzed in [17], load imbalance is less important than locality in software DSMs. Therefore, for three kernels with fine computation granularity, SOR, JI, and TC, Static scheduling scheme remains well since the processor affinity is maintained in Static scheme with respect to some dynamic schemes, such as SS, GSS etc. However, for coarse grain applications and applications which have limited locality, the performance of Static scheme becomes unacceptable because of the existence of load imbalance, such as MM and TC. Figure 4 demonstrates our analyses. The *Syn* overhead listed in Table 8 and Table 9 shows the effects of load imbalance.

Similar to the dedicated environment, though SS promises the perfect load balance among all the nodes, SS remains the worst scheme for all applications except MM because of the large loop allocation overhead, as shown in Table 6. Furthermore, Table 7 shows the inherent drawback of traditional central queue based scheduling algorithm, i.e., potential to violate processor affinity, results in large count of remote getpage operations. Two columns, *Data* and *Syn*, listed in Table 8, reflect the effects of these two aspects obviously. SS performs better than Static for MM because the synchronization waiting time dominates the whole synchronization overhead here. The corresponding *Syn* value reduces from 157.1 in Static to 8.1 in SS in MM kernel.

BSS, GSS, FS, TSS, and SSS make some progress compared with SS, especially for MM kernel. The performance of these 5 scheduling schemes is acceptable. However, as discussed in the last subsection, due to the large extra overhead resulting from loop allocation and corresponding potential of violating processor affinity associated with BSS, GSS, FS, TSS, and SSS scheduling schemes, as listed in Table 6 and Table 7, the performance of these 5 schemes remains unacceptable with respect

Table 8: System overhead of different scheduling algorithms in metacomputing environment(I)

Apps.	Static			SS			BSS			GSS			FS		
	Data	Syn.	Sever.	Data	Syn.	Sever.	Data	Syn.	Sever.	Data	Syn.	Sever.	Data	Syn.	Sever.
SOR	24.3	65.4	5.3	954.5	834.5	259.5	192.7	49.0	36.5	181.8	89.3	37.1	154.6	32.8	32.9
JI	29.8	26.0	4.3	601.0	518.9	169.7	214.4	52.2	34.1	173.0	85.7	34.2	175.9	65.0	36.2
TC	21.8	116.2	4.9	345.1	2885.0	305.3	93.2	159.3	42.6	141.6	102.1	39.6	204.6	144.4	57.3
MM	22.9	157.1	13.3	79.3	8.1	20.2	56.7	18.3	16.6	47.8	65.3	19.8	42.1	54.4	16.4
AC	0.3	39.1	0.1	35.6	777.3	89.0	0.5	0.5	0.2	0.5	45.6	0.2	0.4	27.2	0.2

\* For save line width, we reduce the number of digit after dot to 1 here.

to Static scheduling scheme in metacomputing environment for three fine grain kernels, which is illustrated in Figure 4. Among these five scheduling schemes, SSS is the worst due to the large chunk size allocated in the first phase, which results in large amount of waiting time at synchronization points. For example, the *Syn* value of SSS is (*SOR*, 122.5), (*JI*, 138.7), while the corresponding values in FS are (*SOR*, 32.8), (*JI*, 65.0)<sup>1</sup>. The rest are presented in Table 8 and Table 9 respectively. Therefore, the conclusions proposed in [11] are no longer available. However, from the Figure 4, we find the FS is prior to GSS in metacomputing environment as claimed in [6].

For all five applications but TC, the performance of AFS is improved significantly compared with the former 6 dynamic scheduling schemes, as shown in Figure 4. Though the number of synchronization operations of AFS increases about one order of magnitude, as shown in Table 6, the number of remote getpage reduces about one order of magnitude, as listed in Table 7, which amortizes the effects of loop allocation overhead. Surprisingly, the number of remote getpage operations of AFS in TC leads to the contrary conclusion, i.e., it does not reduce at all with respect to former scheduling schemes. The main culprit here is the relaxed memory consistency model used in state-of-the-art software DSM system, where all the data associated with one synchronization object will be invalidated when requesting this synchronization object. Therefore, the large number of synchronization operations of AFS will lead to large amount of invalidation, which in turn leads to large number of remote fetchs. So synchronization overhead becomes unacceptable in AFS for TC, as shown in Table 9.

As described before, AAFS increases the number of synchronization overhead in local scheduling phase in order to collect the states of other nodes. Therefore, the corresponding synchronization overhead becomes larger than that of AFS. For example, in TC, the number of synchronizations increases from 56924 of AFS to 123529 of AAFS, the corresponding synchronization time increases from 212.9 seconds in AFS to 541.2 seconds in AAFS. Other results are presented in Figure 4, Table 6 and Table 7, and Table 9. Our results give the contrary conclusion to that presented in [23] where AAFS was better than AFS for all five applications in their two hardware platforms.

The ABS scheduling scheme achieves the best performance among all of the dynamic scheduling schemes, as shown in Figure 4, and is superior to Static for all

<sup>1</sup>Since the algorithm of FS and SSS is very similar except the value of chunk size in allocation phase, we compare it with FS here.

Table 9: System overhead of different scheduling algorithms in metacomputing environment (II)

Apps.	TSS			SSS			AFS			AAFS			ABS		
	Data	Syn.	Sever.	Data	Syn.	Sever.	Data	Syn.	Sever.	Data	Syn.	Sever.	Data	Syn.	Sever.
SOR	148.3	42.5	28.9	157.8	122.5	38.7	70.0	29.6	15.9	44.0	51.9	16.4	30.3	23.7	10.2
JI	165.5	86.8	34.1	126.8	138.7	31.0	40.6	28.6	11.7	15.2	61.2	11.7	3.9	32.0	5.4
TC	98.1	146.6	41.2	53.1	93.2	19.4	64.1	212.9	41.9	72.2	541.2	81.0	38.9	99.8	19.3
MM	39.8	59.2	15.9	40.8	137.5	19.1	55.2	2.2	11.1	49.8	3.4	14.5	50.9	1.1	12.0
AC	0.5	28.7	0.1	0.4	46.1	0.1	0.8	2.0	0.7	0.7	4.5	1.3	0.5	1.5	0.6

\* For save line width, we reduce the number of digit after dot to 1 here.

5 kernels except TC. From Table 6 and Table 7, we can find that the great reduction of the number of getpages, an obvious result of the exploitation of processor affinity, amortizes the negative effects of the increasing of synchronization overhead. Table 8 and Table 9 show this case. For TC kernel, the advantage of ABS does not manifest with respect to the best Static scheduling scheme. There is only 8% performance gap between them. Compared with AFS, the synchronization overhead resulting from loop allocation in ABS reduces about one order of magnitude or more, as shown in Table 6. Furthermore, the number of getpages reduces significantly because of the reduction of synchronization operations and the lock-based cache coherence protocol adopted in JIAJIA. In this protocol, all the write notices are accompanied by lock. Therefore, the larger the number of lock operations, the more the possible is the invalidation and the getpage operation. So the corresponding values of *Data*, *Syn* and *Server* all reduce greatly from AFS to ABS, as listed in Table 9. Due to space limit, the evaluation of the ABS in dedicated environment is not described here, more details can be found at[15].

## 5. Related Work

Many loop scheduling schemes were proposed to exploit parallelism. *Static scheduling* minimizes run time synchronization overhead, but does not balance the load dynamically. Dynamic scheduling algorithms include processor self-scheduling [20], guided self-scheduling[14], adaptive guided self-guiding[2], trapezoid self-scheduling [22], affinity scheduling[12], adaptive affinity scheduling[23], etc.. These schemes work well in the context of multiprocessors, however, their performance is not desirable in software DSMs[18].

Ioannidis and Dwarkadas[8] addressed the load balancing in software DSM systems too. Their scheme requires compiler support to extract access pattern from the source code, which is different from our scheme. Furthermore, the redistribution scheme of our schemes is different with theirs too.

Markatos and Le Blanc in [13] analyzed the importance of load balancing and locality management in shared memory multiprocessors. They argued that locality management is more important than load balancing in thread assignment. They introduced a policy called *memory conscious scheduling* that assigns threads to processors whose local memory holds most of the data the thread will access. Their results showed that the looser the interconnection work, the more important the locality management. Our algorithm is greatly inspired by this conclusion.

Hsieh et. al. proposed computation migration to enhancing locality for dis-

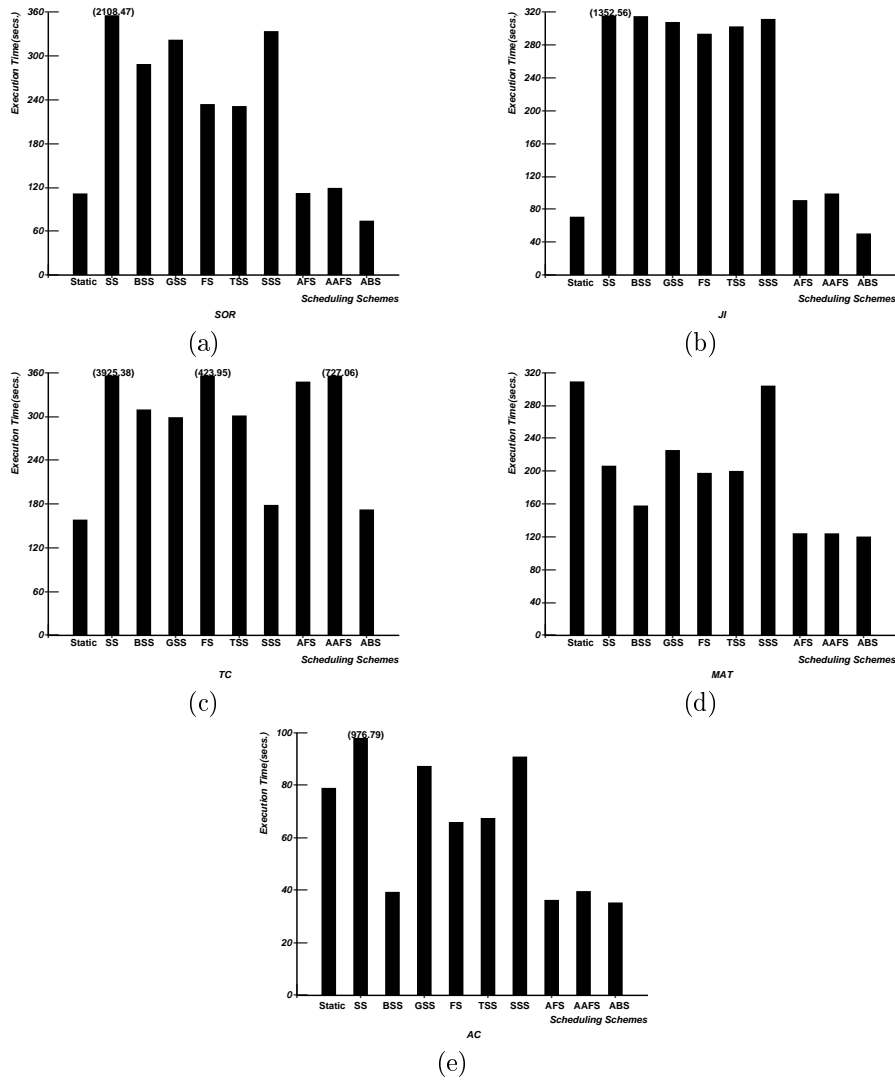


Figure 4: Comparison of execution time of different scheduling schemes in meta-computing environment:(a)SOR, (b) JI, (c) TC, (d) MM, and (e) AC

tributed memory parallel systems[3]. They compared the performance of RPC and data migration. Since the goal of their computation migration is locality, it may result in load imbalance, which is different from our goal, which uses computation migration to balance the load, and try to maintain processor locality at the same time. Furthermore, the implementation methods we used is also different. They implemented computation migration in kernel level, while we implement it in user level.

Subramaniam and Eager[19] proposed dynamic and wrapped schemes for task migration, however, their scheme is computation migration, not task migration.

Thread migration is widely used to balance the load and improve the locality in software DSM, systems such as Millipede[9] and D-CVM[21]. Compared with thread migration, our scheme has two different characteristics: (1) Lower overhead. Migrating an entire thread is expensive, since there may be a large amount of state to move. Moreover, thread migration would lead to inconsistencies in the shared memory state[21]. While in our task migration, the computation migration is embedded in the synchronization operation with minimal additional overhead. (2) Ease of use. Task migration proposed in this paper is performed automatically without requiring any extra information about application characteristics. While in D-CVM's thread migration scheme, the static relation among different threads and the cost of each thread are required in order to make an appropriate decision. Though Millipede uses access histories in choosing threads for automatic migration too, it lacks the global view since it only considers the interactive between any two nodes and neglects the communication used to keep the shared data coherent between the processors.

The dynamic task scheduling technique proposed by Liang et al. [10] is similar to our work. Their scheduling strategy also includes two steps: *Migration* and *Exchange*. However, there are three substantial differences: (1) In their migration stage, the decision of migration is according to computation time only, and the processor power is assumed to be fixed, which does not reflect the real world of scenarios. (2) The exchange stage is used according to access patterns, i.e., in their scheme, the pair of the tasks with the most number of shared pages will be reallocated onto the same processor in the exchange stage. In our scheme, the data, not the computation task, is migrated according to access patterns. (3) Their scheme is thread-level, while ours is loop-level.

## 6. Conclusions

As the prevalence of network of workstations, load balancing plays more and more important role in improving the utilization of computer systems. In this paper, we proposed two efficient scheduling schemes for iterative applications and non-iterative applications respectively. For iterative applications, we identified the disadvantages of traditional task migration methods firstly, and presented a new definition of a task, which takes into account both computation and data. Based on this observation, a new dynamic task migration was proposed and evaluated it within the context of the JIAJIA software DSM system. The evaluation results

show that new task migration scheme improves the performance ranging from 36% to 50% compared with a static task allocation scheme, and performs better than traditional task (computation-only) migration approach about 12.5% for MAT, and 37.5% for SOR and EM3D, and hence high resource utilization is achieved.

Furthermore, a new affinity-based self scheduling scheme was proposed for non-iterative applications. Based on the evaluation results and analyses, we have observed that ABS performs well for both iterative and non-iterative applications, and ABS is the best scheduling scheme among the eight previous scheduling schemes in the context of software shared memory systems because of the reduction of synchronization overhead and the great improvement of waiting time resulting from load imbalance. In this paper, we assume the execution time of loads will be long enough for us to scheduling, how ABS adapt itself to the more dynamic environment, for example short-lived loads, will be our future work.

## Acknowledgements

We would like to thank Prof. Weiwu Hu for providing invaluable suggestions to our work. We also show great thanks to the anonymous referees of I-SPAN'99 and IFJCS, whose feedback allowed us to improve the quality of the paper. An allocation of computer time from the National Center of Intelligent Computing (NCIC) of China is gratefully acknowledged.

## References

1. C. Amza, A. L. Cox, S. Dwarkadas, L-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):467–475, March 1999.
2. D. L. Eager and J. Zahorjan. Adaptive guided self-scheduling. Technical Report CS-87-11, Department of Computer Science and Engineering, University of Washington, January 1987.
3. W. C. Hsieh, P. Wang, and W. E. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proc. of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'93)*, May 1993.
4. W. Hu, W. Shi, and Z. Tang. Home migration in home-based software dsms. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, June 1999.
5. W. Hu, W. Shi, and Z. Tang. JIAJIA: An svm system based on a new cache coherence protocol. In *Proc. of the High-Performance Computing and Networking Europe 1999 (HPCN'99)*, pages 463–472, April 1999.
6. S. E. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A practical and robust method for scheduling parallel loops. *Communication of ACM*, 35(8):90–101, August 1992.
7. L. Iftode. *Home-based Shared Virtual Memory*. PhD thesis, Dept. of Computer Science, Princeton University, June 1998.
8. S. Ioannidis and S. Dwarkadas. Compiler and run-time support for adaptive load balancing in software distributed shared memory systems. In *Proc. of the Fourth*

9. A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its applications in distributed shared memory systems. *The Journal of Systems and Software*, 47(1):71–87, July 1998.
10. T-Y. Liang, C-K. Shieh, D-C. Liu, and W. Zhu. Dynamic task scheduling on multithreaded distributed shared memory systems. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume II, pages 1058–1065, July 1998.
11. J. Liu, V. A. Salestora, and T. G. Lewis. Safe self-scheduling: A parallel loop scheduling scheme for shared memory multiprocessors. *Int'l Journal of Parallel Programming*, 22(6):589–616, June 1994.
12. E. Markatos and T. Le Blanc. Using processor affinity in loop scheduling on shared memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 5(4):379–400, April 1994.
13. E. P. Markatos and T. J. LeBlanc. Load balancing vs locality management in shared memory multiprocessors. In *Proc. of the 1992 Int'l Conf. on Parallel Processing (ICPP'92)*, August 1992.
14. C. Polychronopoulos and D. Kuck. Guided self-scheduling: A practical self-scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987.
15. W. Shi. *Improving the Performance of Software DSM Systems*. PhD thesis, Institute of Computing Technology, Chinese Academy of Sciences, available at <http://www.cs.nyu.edu/weisong/pubs.html>, November 1999.
16. W. Shi, W. Hu, and Z. Tang. Where does the time go in svm system?—experiences with JIAJIA. *Journal of Computer Science and Technology*, 14(3):193–205, May 1999.
17. W. Shi and J. Ma. High efficient parallel computation of resonant frequencies of waveguided loaded cavities on JIAJIA software dsm system (poster). In *Proc. of the High-Performance Computing and Networking Europe 1999 (HPCN'99)*, April 1999.
18. W. Shi, Z. Tang, and W. Hu. A more practical loop scheduling for home-based software dsms. In *Proc. of the ACM-SIGARCH Workshop on Scheduling Algorithms for Parallel and Distributed Computing—From Theory to Practice*, June 1999.
19. S. Subramaniam and D. L. Eager. Affinity scheduling of unbalanced workloads. In *Proc. of Supercomputing'94*, pages 214–226, November 1994.
20. P. Tang and P.-C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proc. of the 1986 Int'l Conf. on Parallel Processing (ICPP'86)*, pages 528–535, August 1986.
21. K. Thitikamol and P. Keleher. Thread migration and communication minimization in dsm systems. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):487–497, March 1999.
22. T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):87–98, January 1993.
23. Y. Yan, C. Jin, and X. Zhang. Adaptively scheduling parallel loops in distributed shared memory systems. *IEEE Trans. on Parallel and Distributed Systems*, 8(1):70–81, January 1997.