Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems

Yong Yan, *Member, IEEE*, Canming Jin, and Xiaodong Zhang, *Senior Member, IEEE*

Abstract—Using runtime information of load distributions and processor affinity, we propose an adaptive scheduling algorithm and its variations from different control mechanisms. The proposed algorithm applies different degrees of aggressiveness to adjust loop scheduling granularities, aiming at improving the execution performance of parallel loops by making scheduling decisions that match the real workload distributions at runtime. We experimentally compared the performance of our algorithm and its variations with several existing scheduling algorithms on two parallel machines: the KSR-1 and the Convex Exemplar. The kernel application programs we used for performance evaluation were carefully selected for different classes of parallel loops. Our results show that using runtime information to adaptively adjust scheduling granularity is an effective way to handle loops with a wide range of load distributions when no prior knowledge of the execution can be used. The overhead caused by collecting runtime information is insignificant in comparison with the performance improvement. Our experiments show that the adaptive algorithm and its five variations outperformed the existing scheduling algorithms.

Index Terms—Adaptive scheduling algorithms, dynamic information, load balancing, parallel loops, processor affinity, shared-memory systems.

1 Introduction

Loops are the richest sources of parallelism and are widely used in scientific application programs. In many scientific applications, a set of independent tasks typically exists in a *parallel loop*, called a DoAll loop, where the processing of each element in each iteration is independent of the others.

The performance of a loop scheduling algorithm is mainly affected by three overhead sources: *synchronization and loop allocation*, *load imbalance*, and *data communication*. Although it is desirable for an efficient algorithm to minimize the above three sources of overhead, it is usually impossible because conflicts can arise among them. Exploiting processor affinity (processor affinity refers to certain data access dependence of a task to a specific processor; a more precise definition is given in Section 3.1) favors the allocation of loop iterations close to their data, which tends to cause load imbalance. Load balance favors the "fine grain" allocation of loop iterations (where a small number of iterations are allocated) in order to minimize the effects of uneven assignment. However, the "fine grain" allocation tends to increase synchronization overhead and loop allocation overhead. In different applications, each overhead source affects performance differently. Hence, an efficient loop scheduling algorithm should optimize its performance by adaptively trading off synchronization overhead, loop allocation overhead, load imbalance overhead, and data-communication overhead. Moreover, a dynamic scheduling algorithm should not assume any prior knowledge of the execution times of the loop iterations because the execution of the loop usually is unpredictable in practice.

So far, many novel dynamic scheduling algorithms have been proposed, e.g., [2], [4], [5], [6], [8], [9], [11], [10]. These algorithms fall into two distinct classes: central queue based and distributed queue based. In central queue based algorithms [2], [8], [11], [10], 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 execute. The major advantage of using a central queue is the possibility of evenly balancing the workload. While keeping a good load balance, the central queue based algorithms differ in the way they reduce synchronization and loop allocation overheads. However, three limitations are associated with the use of a central queue:

1) An iteration in the central queue is likely to be dynamically allocated to execute on any processor, which does not facilitate the exploitation of processor affinity;
2) During allocation, all the processors but one should remotely access the central work queue, and thereby generate heavy network traffic;
3) Because all the processors contend for the central queue, the central queue tends to be a performance bottleneck, resulting in a longer synchronization delay.

In order to exploit the processor affinity inherent in the parallel execution of many loops and to eliminate the central bottleneck, the affinity scheduling algorithm proposed in [6] distributes the central queue to be local to each proc-

- *Y. Yan and X. Zhang are with the High Performance Computing and Software Laboratory, the University of Texas at San Antonio, San Antonio, TX 78249-0664.*
  E-mail: yyan@dragon.cs.utsa.edu; zhang@ringer.cs.utsa.edu.
- *C. Jin is with InterVoice Inc., Dallas, Texas. E-mail: cjin@intervoice.com.*

Manuscript received Feb. 19, 1996.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D96040.

essor, and the algorithm partitions iterations of a parallel loop statically into local queues so that each processor only is involved in remote access while load imbalance occurs. Markatos and Leblanc [6] show that affinity scheduling almost achieves the best performance in all tested cases when compared with those central queue based algorithms. To enhance the affinity scheduling algorithm in the presence of large, correlated imbalance in loop execution time, Subramaniam and Eager [9] propose two loop partition methods: dynamic partition and wrapped partition. These two partition methods, however, only improve affinity scheduling for some specific applications because both of them execute under some specific assumptions about the distribution of loop execution time.

In the design of distributed queue based algorithms, we have no reason to prefer other kinds of loop partition methods to uniform partition, due to the uneven and unpredictable execution time of loop iterations. Hence, it is crucial for a distributed queue based algorithm to be able to dynamically and efficiently schedule tasks at run-time to even the load imbalance caused by the static partition. In existing affinity scheduling algorithms [6], [9], all the processors schedule loop iterations in their local queues using the same allocation scheme where, at each time, $1/P$ of the remaining iterations in the local queue are allocated (P is the number of processors). This iteration allocation scheme may not be efficient. For example, if the initial loop partition is balanced then all processors will complete the execution of iterations in their local queues at the same time, and each processor should grab all the iterations in the local queue in an allocation, instead of only $1/P$ of the remaining iterations. On the other hand, if the initial loop partition is not balanced, those lightly loaded processors should finish execution of the iterations in their local queue as soon as possible so that they can immediately turn to help heavily loaded processors. Hence, processors should be able to dynamically increase or decrease their allocation granularity based on runtime information to reduce synchronization and loop allocation overhead and balance load more evenly. This motivates us to design adaptive scheduling algorithms to further improve existing affinity scheduling algorithms.

The major objective of this paper is to exploit the potential of dynamic information to reduce loop execution time. We propose an adaptive scheduling algorithm and its five variations. Our algorithms dynamically adjust allocation granularity according to a program's execution history. We classify parallel loop execution patterns and fairly select a set of applications to experimentally verify the effectiveness of our algorithms in comparison with the proposed affinity scheduling algorithms in [6], [9].

Our experimental results show that using runtime information to adaptively adjust scheduling granularity is an effective way to handle a wide range of load distributions when no prior knowledge can be used. The overhead caused by collecting runtime information is insignificant in comparison with the benefit gained by this method when proper algorithms are designed. The experiments show that the more aggressive an algorithm is in using dynamic information, the more it improves the execution performance of parallel loops. The adaptive algorithm outperformed the existing scheduling algorithms experimentally.

The organization of this paper is as follows. Section 2 describes in detail the main ideas in the design of the adaptive scheduling algorithm and presents five variations that have different degrees of aggressiveness in the adjustment of loop scheduling granularities. In order to assess precisely the effectiveness of the proposed algorithms in practice, we analyze and classify the program characteristics of applications in Section 3. We selected five representative kernel applications as benchmarks for performance evaluation. In Section 4, we report our experimental results and comparisons. We summarize the paper in Section 5.

2 Adaptive Scheduling Algorithms

Similar to the affinity scheduling algorithm [6], the adaptive affinity scheduling algorithm is also constructed to have following three phases:

Initial partition phase: Deterministic assignment policy is used to partition iterations of a parallel loop into local queues of processors, which ensures that an iteration is always assigned to the same processor at the start. With this assignment scheme, if a parallel loop executes repeatedly and each parallel iteration accesses the same data set in different executions, the first execution of the parallel loop will bring data locally to processors so that the subsequent execution of the parallel loop only involves local data access.

Local scheduling phase: Based on some local scheduling policy, each processor allocates a part of the remaining iterations in a local queue to execute until the local queue is empty. Local scheduling does not cause remote access overhead. Because each local queue is shared by all processors, a critical section is used to protect the allocation of the loop iterations in the local queue. The local scheduling overheads mainly come from the synchronization overhead and the loop allocation overhead in the execution of the critical section. Reducing the number of allocations is crucial to improve the performance of the local scheduling phase.

Remote scheduling phase: When a processor finishes the execution of all the iterations in the local queue, it remotely allocates a portion of the iterations from the most loaded processor in the system to execute. The remote scheduling phase is aimed at dynamically balancing the workload. An iteration is at most reassigned once, which avoids processor thrashing. Remote scheduling causes remote data access overhead as well as synchronization overhead and loop allocation overhead.

Instead of relying on preknowledge about a loop's execution, our adaptive affinity scheduling algorithm exploits the potential of using dynamic execution history to adaptively adjust iteration chunking size to reduce synchronization and loop allocation overheads. The algorithms also maintain a better load balance. The main idea of our designs is to minimize local scheduling overhead so that the phase of dynamically balancing the workload can be

speeded up, which results in reduction of loop execution time.

In the initial phase, a loop with N iterations is partitioned into chunks of uniform size $\lceil N/P \rceil$ over P processors because we have no reason to prefer other partition methods in the absence of a precise prediction about the execution distribution of the loop's iterations. Our initial partition is identical to the one in [6].

In the local scheduling phase, a processing speed variable s_i, termed the *PS* variable, is set for each processor, which keeps track of the number of iterations the processor has executed so far ($i = 1, \cdots, P$). Variable s_i is initially set to zero and is increased by one each time processor i finishes the execution of an iteration. By comparing the local *PS* variable with other *PS* variables, a processor can observe the load distribution. At any time, the processors with smaller *PS* variable values have executed iterations that have a heavier workload than those executed by the processors with larger *PS* variable values. Because, in most applications, a load distribution state has a certain steady duration, it is feasible to speculate about the load distribution in the near future by the current observation. Although some applications really have a surge variance in the load distribution, the prediction difference can be minimized by dynamic readjustments. In order that processors response spontaneously to the dynamic changes of iteration workload, it is necessary to differentiate the workload states of processors. For fairness, we select the average number of iterations executed by all processors, i.e., $\sum_{i=1}^{P} s_i/P$, as the pivot to partition the workload states of processors into the following three types:

- heavily loaded (HL)—the processor's *PS* value is smaller than $\sum_{i=1}^{P} s_i/P - \alpha$;
- normally loaded (NL)—the processor's *PS* value is within the range of $\left(\sum_{i=1}^{P} s_i/P - \alpha,\ \sum_{i=1}^{P} s_i/P + \alpha\right)$; and
- lightly loaded (LL)—the processor's *PS* value is equal to or larger than $\sum_{i=1}^{P} s_i/P + \alpha$;

where α is a nonnegative, range control variable, which adjusts the distribution of HL processors, NL processors, and LL processors. Variations of parameter α would affect the algorithmic performance. The dynamic features in the execution of real applications' loops make it impractical or impossible to analytically determine an optimal value of α. Here, we will discuss the effect of different values of α on performance through experiments, giving an empirical method of determining a performance-efficient α.

In order to control chunk size in the allocation of loop iterations, a chunk-size control variable k_i is set for processor i ($i = 1, \cdots, P$). Each processor always removes $1/k_i$ of the remaining iterations in its local work queue for execution. In the beginning of the local scheduling phase, all the chunk-size control variables are initialized to the same value, such as P, the total number of processors. Then, each k_i ($i = 1, \cdots, P$) is adaptively and independently adjusted by a chunk-size control function Π. Function Π uses the load state and the current value of k_i of processor i as its two input parameters, and adjusts the chunk-size control variable k_i as follows:

- If processor i is heavily loaded, Π increases k_i, aiming at reducing chunk size so that more iterations remaining in the heavily loaded processor can be executed by those lightly loaded processors, therefore, balancing workload more efficiently.
- If processor i is normally loaded or lightly loaded, Π decreases k_i, aiming at increasing chunk size so that a normally loaded or lightly loaded processor can finish all the iterations in their local work queues as soon as possible, and then immediately starts to help heavily loaded processors.

When processor i completes execution of the iterations in the local queue, it turns to the remote scheduling phase. In the affinity scheduling algorithm of [6], when a processor exhausts its local work queue and starts to help other heavily loaded processors, it just removes $\lceil 1/P \rceil$ of the remaining iterations from the most heavily loaded processor. This allocation method may not be efficient when only a few processors can turn to help other processors. Here, we determine the chunking size according to the current number of lightly loaded or normally loaded processors because they are able to help those heavily loaded processors in the near future. For processor i, it determines its chunk control variable k_i as follows:

$$k_i = min\{P, n+1\}, \tag{2.1}$$

where n is the total number of lightly loaded processors and normally loaded processors ($n + 1$ means to include the most heavily loaded processor from which processor i will allocate the remaining iterations), and P is the total number of processors. Then, processor i allocates $\lceil 1/k_i \rceil$ of the remaining iterations in the most heavily loaded processor to execute. This procedure will repeat until all local work queues are empty. Initially, k_i has a smaller value than P so that a big chunk size is used to reduce the number of remote allocations. Our experiments in the next section will show that the selected big chunk size does not increase the risk of imbalancing load. Subsequently, when more processors become lightly loaded or normally loaded, k_i will increase until it reaches the maximal value, P.

In the following, a pseudocode description of the adaptive affinity scheduling algorithm is given. In implementation, this code can be automatically inserted by a compiler into application programs for each processor to dynamically schedule the execution of loops without the interference of the operating system. As with other existing work, we generate dynamic scheduling programs by hand in our experiments in order to focus on the algorithmic study.

1. Initial partition phase:

```
initial_partition(N, P) // N iterations are uniformly partitioned over P processors
{for (i = 0; i < P; i ++)
    assign_iterations(i); // assign iterations to processor i.
 for (i = 0; i < P, i ++){
```

$s_i = 0$; $k_i = P$;} / / Initialize PS variables and K_i's
}
2. Local scheduling phase on processor *i*:
loop { / / processor *i* gets $1/k_i$ of the local iterations to execute and adjusts k_i.
Lock(local_queue_*i*);
range = get_iterations(local_queue_*i*, $1/k_i$); / / allocate $1/k_i$ of the iterations.
unlock(local_queue_*i*)
While (*range*−− ! = 0) {
execute_one_iteration(); s_i ++;}
state = load_state$\left(\left(\sum_{i=1}^{P} s_i\right)\Big/P, s_i, \alpha\right)$; / /compute the load state of processor *i*.
$k_i = \Pi(state, k_i)$; / / adjust the chunking granularity
} until (local_queue_*i* = ∅)
3. Remote scheduling phase on processor *i*:
$k_i = 1$;
loop { if (k_i ! = *P*) {
k_i = find_non_heavily_loaded_processor(); k_i = min{*P*, k_i + 1}; }
j = find_most_loaded_processor();
lock(local_queue_*j*);
range = get_iterations(*j*, $1/k_i$); / / get $\lceil 1/k_i \rceil$ of the iterations on processor *j*.
unlock(local_queue_*j*);
execute(*range*);
} until (all iterations have been finished).

Adaptively changing loop scheduling granularity is the major characteristic which distinguishes our adaptive affinity scheduling from the affinity scheduling algorithm in [6]. Remotely reading the *PS* variables of other processors is the overhead caused by our adaptive scheduling algorithm in collecting execution history of other processors. If the increased overhead nullifies the benefit of adaptively varying loop scheduling granularity, the adaptive affinity scheduling algorithm may not exhibit a performance improvement over existing affinity scheduling algorithms.

Different variations of the adaptive affinity scheduling algorithm can be constructed by designing different chunk-size control protocols for the function Π. Here, we propose four mechanisms for our adaptive algorithm. Let k_i be the chunk size control variable of processor *i* in the local scheduling phase.

- **Exponentially Adaptive (EA) Mechanism**
  In the EA mechanism, a processor increases or decreases the value of its chunk-size control variable k_i by a factor at each time according to the current load state. The chunk-size control function Π is formally defined as

$$\Pi(\text{state}, k_i) = \begin{cases} k_i * base & \text{if state = HL} \\ \lceil k_i / base \rceil & \text{if state = NL or LL} \end{cases}$$

  where *base* is an integer constant. Here, we choose two for *base*. Initially, k_i is set to *P* in our adaptive algorithm.
- **Linearly Adaptive (LA) Mechanism**
  In the LA mechanism, a processor increases or decreases its chunk size control variable k_i by a constant at each time interval according to the current load state. The chunk-size control function Π is formally defined as

$$\Pi(\text{state}, k_i) = \begin{cases} k_i + con & \text{if state = HL} \\ max\{1, k_i - con\} & \text{if state = NL or LL} \end{cases}$$

  where *con* is a constant specified by users. We choose 1 in our experiments. Because the LA mechanism changes the chunk size at a slower pace than the EA algorithm, it has less risk of imbalancing the workload, but larger synchronization and loop allocation overhead.
- **Conservatively Adaptive (CA) Mechanism**
  A careful selection of the chunking size in a loop scheduling algorithm is crucial to find a compromise between synchronization overhead and load imbalance. Allocating a bigger chunk of the iterations of a loop tends to reduce synchronization and loop allocation overhead, but increase the risk of imbalancing load. Previous work in [5] shows that in order to have reasonable load imbalance and synchronization overhead, it is safe for the chunk size control variable k_i to choose a value in [*P*, 2*P*]. The CA mechanism is constructed by restricting the varying range of the chunk-size control variables of the LA mechanism within [*P*/2, 2*P*]. The chunk size control function is defined as follows:

$$\Pi(\text{state}, k_i) = \begin{cases} \min\{2P, k_i + con\} & \text{if state = HL} \\ max\{P/2, k_i - con\} & \text{if state = NL or LL} \end{cases}$$

  where *con* is a constant in [0, *P*]. We will use one in our experiments.
- **Greedily Adaptive (GA) Mechanism**
  The GA mechanism employs a two-phase consensus method to greedily enlarge the chunking size on non-heavily loaded processors. The GA mechanism records the previous load state of the processor. If a processor finds it is in a nonheavily loaded state in two consecutive allocations, it greedily reduces the chunk-size control variable to 1, i.e., it grabs all the remaining iterations in the local work queue to execute. Otherwise, the processor increases or decreases the chunking size by using the conservation method in the CA mechanism with *con* = 1 experimentally.

  Let S_{pre}^{i} record the previous load state of processor *i*, and let S_c record the current load state of processor *i*. The chunk size control function is

$$\Pi(S_c, k_i) = \begin{cases} \min\{2P, k_i + con\} & \text{if } S_c = \text{HL} \\ max\{P/2, k_i - con\} & \text{if } S_c \neq \text{HL and } S_{pre}^{i} = \text{HL} \\ 1 & \text{if } S_c \neq \text{HL and } S_{pre}^{i} \neq \text{HL} \end{cases}$$

Keeping and maintaining the *PS* variable for each processor allows the above four adaptive mechanisms to know exactly the current workload of each processor; thereby, the *PS* variables can be used to adjust the speed for each proc-

essor, and as a consequence, to adjust the workload among the processors. But it also introduces loop allocation overhead. Here, we design a heuristic variation, denoted by HA, which still adopts the framework of our adaptive scheduling algorithm. Instead of using *PS* variables to determine workload distribution among processors, we use the number of iterations actually executed by each processor to guide the adjustments of scheduling granularities. Initially, a parallel loop is uniformly distributed to processors. Each processor i repeats grabbing $1/k_i$ of the remaining iterations in its local queue to execute without doing any adjustment to k_i. If processor i finishes all the iterations in its local work queue, and turns to get iterations from the most heavily loaded processor j, processor i is said to be lightly loaded and processor j is heavily loaded. Then processor i increases its scheduling granularity and processor j decreases its scheduling granularity. Hence, the lightly loaded processors can turn as early as possible to help heavily loaded processors, and the heavily load processors can remain as much workload as possible to even those lightly loaded processors. At the end of each execution of the parallel loop, processors check whether they have executed approximately the same number of iterations, i.e., a balanced workload. If so, processors increase their scheduling granularities to speed up subsequent executions of the parallel loop.

Comparing with the four variations of the adaptive algorithm: EA, LA, CA, and GA, the HA variation differs in several aspects:

1) Instead of determining the load state at each time of local scheduling that is used by the adaptive algorithms, the HA variation updates the load states of processors only in the remote scheduling phase and at the end of one execution of the parallel loop, so that it causes less scheduling overhead than the adaptive algorithm.
2) The HA variation works by requiring that the parallel loop be nested in a sequential loop to execute repeatedly.

When the parallel loop only executes once, the HA variation becomes the affinity algorithm [6].

The pseudocode of the heuristic variation of the adaptive affinity scheduling algorithm (HA variation) is shown as follows.

1. Initial partition phase:

```
initial_partition(N, P) // N iterations are uniformly partitioned over P processors
{for (i = 0; i < P, i ++)
    assign_iterations(i); // assign iterations to processor i.
 for (i = 0; i < P; i ++)
    k_i = P;
}
```

2. Local scheduling phase on processor *i*:

```
loop { Lock(local_queue_i);
       range = get_iterations(i, 1/k_i); // allocate 1/k_i of the remaining iterations
       unlock(local_queue_i);
       execute(range);
} until (local_queue_i = ∅)
```

3. Remote scheduling phase on processor *i*:

```
loop {j = find_most_loaded_processor();
      lock(local_queue_j);
      range = get_iterations(j, 1/k_j); // get ⌈1/k_j⌉ of the iterations on processor j.
      unlock(local_queue_j);
      if (k_i > 1) k_i = k_i − 1; // increase the chunk size for processor i.
      if (k_j < 2 * P) k_j = k_j + 1; // decrease the chunk size for processor j.
      execute(range); }
```

4. The program section at the end of each parallel loop:

```
end_para_loop
{     barrier(&barrier, &P);
      if (tid == 0) // only processor 0 execute the code.
        find_maximum_and_minimum_of_chunk_
          sizes(kmax, kmin);
      if ((kmax − kmin < P/2) // if the workload is balanced, increase chunk size.
        for (each processor i with k_i > 1) k_i = k_i/2;
      barrier(&barrier, &P);
}
```

3 Experimental Evaluation Methods

Markatos and Leblanc [6] show that the affinity scheduling algorithm (hereafter, simplified as the ML algorithm) outperforms other algorithms that do not exploit processor affinity. Hence, we focus on comparing the variations of the adaptive scheduling algorithm with the affinity scheduling algorithm and its two variations in [9]. The scheduling algorithms we have evaluated and compared are:

1) the ML affinity scheduling algorithm,
2) the SE dynamic initial partition affinity scheduling algorithm,
3) the adaptive affinity algorithm with the exponential adaptive mechanism (EA),
4) the adaptive affinity algorithm with the linearly adaptive affinity mechanism (LA),
5) the adaptive affinity algorithm with the conservatively adaptive mechanism (CA),
6) the adaptive affinity algorithm with the greedily adaptive mechanism (GA), and
7) the heuristic adaptive variation (HA).

The experiments were conducted on two machines: the KSR-1, a hierarchical-ring-based, cache coherent shared-memory system and the Convex Examplar, a crossbar and ring-based cache coherent shared-memory system. Here, we address our methods of selecting application kernels and of evaluating the algorithms.

3.1 Principles for Selecting Application Kernels

Considering the effects of program features on scheduling algorithms, we characterize parallel loops by three factors: the affinity of loop iterations to processors, the distribution of loop execution time, and the granularity of loop iterations.

The iterations of a parallel loop may exhibit affinity to processors only when the loop is nested in a sequential loop to be executed repeatedly. One fact in parallel processing is that the dominant overhead source in many applications is

communications, not synchronization. Hence, we first classify parallel loops into two classes: potential affinity parallel loops that are nested in a sequential loop, and nonaffinity parallel loops that are only executed once. How strong the iterations of a potential affinity parallel loop that exhibits affinity to processors is significantly affected by the sizes of data sets accessed by iterations, and by the data locality of iterations.

A better data locality of an iteration means that the data set accessed by the iteration changes less significantly in different executions of the iterations. Data locality determines the affinity of iterations to processors. On the other hand, the sizes of data sets accessed by iterations determine the benefit of exploiting processor affinity. For those parallel loops with better data locality, if their iterations have very small data sets (e.g., one integer), exploiting processor affinity will not improve the execution times of these parallel loops more than by balancing load and reducing synchronization overhead.

Let $D(i)$ be the data set of an iteration in the ith execution of a parallel loop, and let $|D(i)|$ be the size in bytes of data set $D(i)$. Then, $\overline{D} = \sum_{i=1}^{N} |D(i)| / N$ is the average size of data sets of the iteration over N executions of the parallel loop, and $\overline{\delta} = \sum_{i=1}^{N} |D(i) - D(i-1)| / N$, (D(0) = D(1)) is the average size difference of two data sets in two consecutive loop executions of an iteration, which indicates approximately how much data should be reloaded in each execution of an iteration of the parallel loop. So, the data locality of an iteration can be quantitatively evaluated by the following defined locality rate:

$$\text{locality\_rate} = 1 - \frac{\overline{\delta}}{\overline{D}},$$

where the locality rate is a value in [0, 1]. A locality rate of one means an iteration always accesses the same set of data. A larger locality rate represents a better data locality. Then, how strong an iteration has affinity to a processor can be quantitatively evaluated by $|\text{locality\_rate} \times \overline{D}|$, the average number of data sets that will be accessed repeatedly (these data sets may be always stored in a local cache). In the selection of potential affinity parallel loops, we use data locality and data size to differentiate the affinity of iterations to processors.

The unpredictable variance in the execution times of parallel loops is a major obstacle for loop scheduling algorithms to work efficiently. In order to show how much parallel loop scheduling algorithms can tolerate different distributions of workload among iterations, we selected parallel loops by load distribution to cover three distinguished types of loops:

1) balanced loops where each iteration has the same amount of computation time,
2) predictable imbalanced loops where the computation times of iterations of a parallel loop vary as a predictable function of the loop control variable or where the load distribution in a parallel loop is fixed when it executes repeatedly, and
3) unpredictable imbalanced loops where the computation times of iterations change randomly, depending on initial input and some runtime variables (e.g., the execution time of a branch statement depends on its actual execution path).

The ML algorithm only handles load imbalance by remote scheduling. The SE algorithm improves the ML algorithm performance only for those predictable imbalanced parallel loops where the load distribution of a parallel loop is not changed in multiple executions of the parallel loop, and the execution times of the iterations of a loop increase or decrease monotonically with the loop control variable. Our adaptive algorithm and its variations dynamically adjust the loop scheduling granularity to speed up the load balance procedure based on the execution history of processors. In the following experiments, we will show that the adaptive algorithm can handle load imbalance more efficiently over a wider range than the ML and the SE algorithms.

Besides affinity and load distribution, the iteration granularity of a loop is another important factor affecting the performance of loop scheduling algorithms. For parallel loops with coarse granularity where the execution times of loop iterations are significantly larger than the overhead of remote access delay, balancing the workload is more crucial than reducing synchronization and loop allocation overheads. For parallel loops with fine granularity where the execution times of loop iterations are much smaller than the overhead of remote access delay, it is important to minimize scheduling overheads. Because the determination of the iteration granularity of a parallel loop depends on the interaction between the parallel loop and the underlying system, it is difficult to tell whether a parallel loop is coarsely grained or finely grained before execution. Instead of classifying parallel loops by granularity, we consider the effect of iteration granularity in our experiments.

Based on the above analyses, we classify parallel loops into six types by their affinity and load distributions:

I) loops with potential affinity and balanced workload,
II) loops with potential affinity and predictable workload,
III) loops with potential affinity and unpredictable workload,
IV) loops with nonaffinity and balanced workload,
V) loops with nonaffinity and predictable workload, and
VI) loops with nonaffinity and unpredictable workload.

In order to have a complete understanding of how well each scheduling algorithm works in the area of real-world applications, we select one application from each type. A loop with nonaffinity and unpredictable workload is a rare case in practice. Therefore, we only evaluate scheduling algorithms for applications of the first five types of loops.

3.2 Applications

The selected application kernels including potential affinity loops are Successive Over-Relaxation (SOR) (type I), Jacobi Iteration (JI) (type II), and Transitive Closure (TC) (type III). Matrix Multiplication (MM) (type IV), and Adjoint Convolution (AC) (type V) are application kernels including nonaffinity loops.

Type I: Balanced affinity loops in SOR.

```
DO SEQUENTIAL 1 I = 1, L
```

```
      DO PARALLEL 2 J = 1, N
         DO SEQUENTIAL 3 K = 1, N
         A(J, K) = UPDATE(A, J, K)
3        CONTINUE
2     CONTINUE
1   CONTINUE
```

All the iterations of the SOR parallel loop take about the same time to execute and each iteration always accesses the same set of data. Exploiting processor affinity may improve performance better than balancing the workload. In this application, each parallel iteration has locality rate of one and a data set of N array elements. The computational granularity of each parallel iteration is O(N).

Type II: Predictable affinity loops in a Jacobi Iteration (JI).

```
    DO SEQUENTIAL 1 I = 1, L /* L controls
        iteration precision. */
       DO PARALLEL 2 J = 1, N
         X1[J] = 0
         DO SEQUENTIAL 3 K = 1, N
            IF (A[J][K] .NE. 0).AND.(J .NE.
K)
               X1[J] = X1[J] + A[J][K] * X0[K]
3          CONTINUE
         X1[J] = (B[J] - X1[J])/A[J][J]
2        CONTINUE
       DO SEQUENTIAL 4 L = 1, N
         X0[L] = X1[L]
4        CONTINUE
1    CONTINUE
```

In the JI program, the top 20% of rows of elements in the nonsingular matrix A are nonzero elements, which are generated by a random number generator. The iterations of the parallel loop have a different workload which is determined by the distribution of nonzero elements in A, so exploiting load imbalance would improve performance. However, the workload of each parallel iteration is not changed when it is executed repeatedly.

The jth iteration of the parallel loop always accesses the jth row of the matrices A, B[j] and x0[j]. When the jth iteration is fixed to be executed repeatedly on a processor, it only needs to reload x0[j] into a cache because x0[j] is updated after each execution of the parallel loop. Hence, this application kernel exhibits good processor affinity. Each iteration has a data set of the size of N + 2 elements, and data locality close to one. The average computational granularity of each iteration is smaller than that in the SOR kernel.

Type III: Unpredictable imbalanced affinity loops in the Transitive Closure (TC) kernel.

```
   DO SEQUENTIAL 1 I = 1, N
      DO PARALLEL 2 J = 1, N
         IF (A[J][I] .EQ. TRUE) THEN
            DO SEQUENTIAL 3 K = 1, N
               IF (A[I][K] .EQ. TRUE)
                  A[J][K] = TRUE
3              CONTINUE
2        CONTINUE
1   CONTINUE
```

The TC program may exhibit more serious load imbalance than JI, where each iteration of the parallel loop and each execution of a parallel iteration may have computational granularity of O(1) or O(N), depending on the input matrix A. The iterations exhibit a weaker affinity to processors than SOR and JI. Due to the random computation feature, it is difficult to quantify the data locality and affinity of each parallel iteration.

Type IV: Balanced nonaffinity loops in Matrix Multiplication (MM).

```
   DO PARALLEL 1 I = 1, N
      DO PARALLEL 2 J = 1, N
         DO SEQUENTIAL 3 K = 1, N
           C[I][J]=C[I][J]+A[I][K]*
B[K][J]
3          CONTINUE
2        CONTINUE
1    CONTINUE
```

The MM program does not have affinity to exploit. All the parallel iterations have computational granularity of O(N). So, reducing synchronization and loop allocation overhead is the only way to improve performance. This application is used to investigate whether the adaptive algorithm has a lower scheduling overhead than the ML algorithm.

Type V: Predictable imbalanced nonaffinity loops in Adjoint Convolution (AC).

```
   DO PARALLEL 1 I =1, N * N
      DO SEQUENTIAL 2 K = I, N * N
         A[I] = A[I] + X * B[K] * C[I - K]
2        CONTINUE
1    CONTINUE
```

Similar to the matrix multiplication application, the parallel loop in the AC kernel only executes once, hence it does not exhibit processor affinity. However, the computational granularity of the ith parallel iteration is $O(N^2 - i)$, changing as a specific function of the control variable i to produce a significant imbalanced load distribution (a triangular pattern). This kernel is used to examine how efficiently the adaptive algorithms can handle the load imbalance caused by a uniform partition.

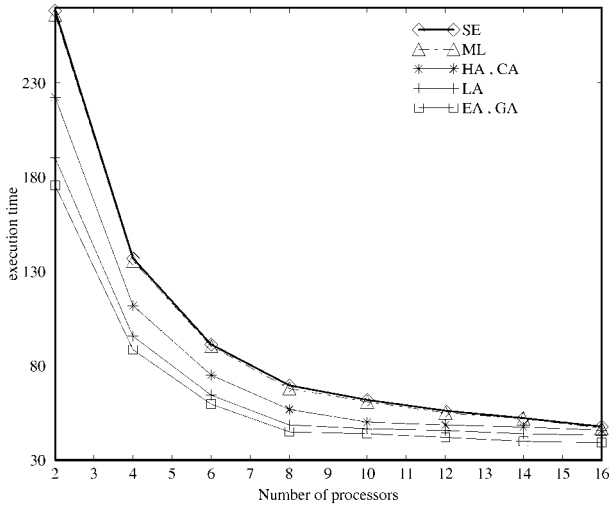
4 Experimental Results

The performance metric we use to evaluate algorithms is execution time. Execution time measures how differently the scheduling algorithms work for different types of applications for given problem size.

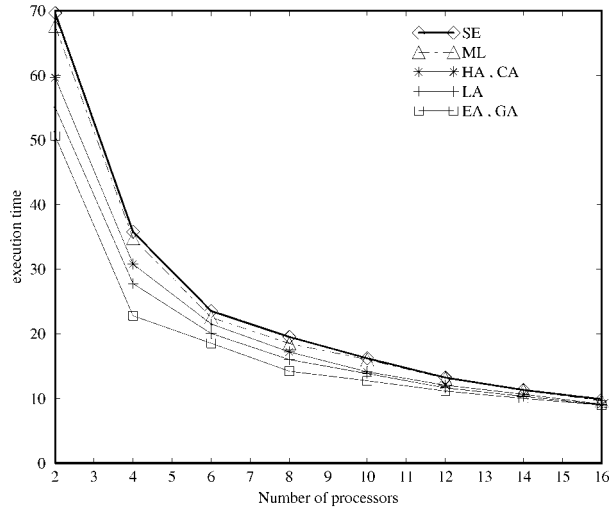
4.1 Comparisons of Loop Scheduling Algorithms

First we use N/P^2 as the α value in our four adaptive scheduling variations CA, LA, EA, and GA. We shall discuss the effect of the α value on performance in a later section, and discuss why $\alpha = N/P^2$ is cost-effective in the next section.

Fig. 1 presents the execution time (in seconds) of SOR (L = 500, N = 1,024) running on two to eight processors on both the KSR-1 and the Convex Exemplar. Since SOR is a perfectly balanced application kernel, the dynamic partition of the SE algorithm did not improve the performance of the ML affinity algorithm. On the other hand, it introduced some overhead into the ML algorithm. As the result, the ML and the SE perform the worst among them all, due to the overhead caused by more loop allocation and synchronization steps. By adaptively increasing the chunk size each

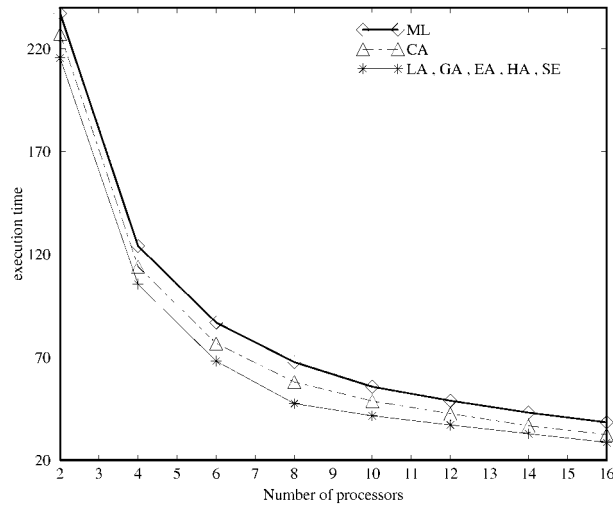


(a)

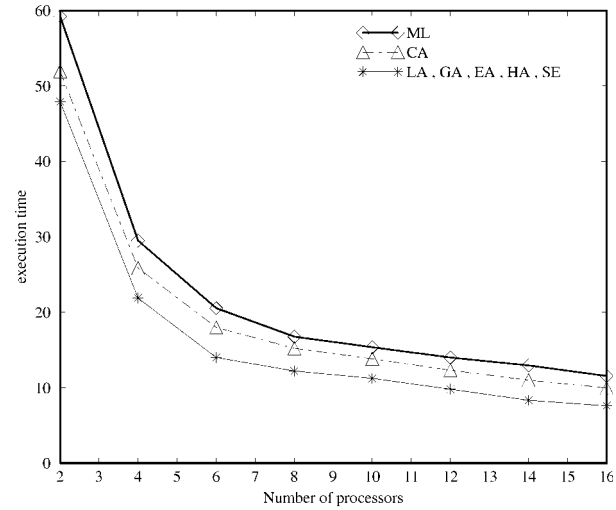


(b)

Fig. 1. Performance of SOR on the KSR-1 (a) and EXEMPLAR (b).



(a)



(b)

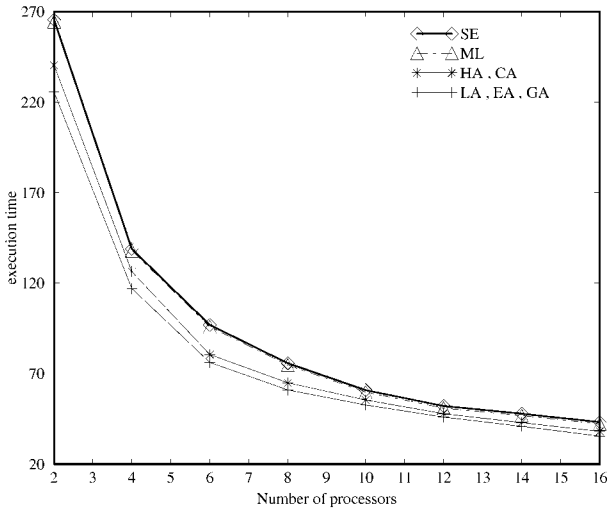
Fig. 2. Performance of JI on the KSR-1 (a) and the EXEMPLAR (b).

time when a processor accesses the local work queue, the adaptive algorithms reduce the times that processors need for accessing the local work queues, therefore, scheduling and synchronization overhead is reduced. All our five adaptive algorithms outperformed the ML and the SE algorithms. The EA and GA performed the best among them all, since they take no more than three steps to adjust their chunk size to finish the remaining iterations. The LA variation needs more allocation steps than the EA and the GA need. The HA and the CA variations change the chunk size in a limited range; therefore, they could not get the best benefit by reducing the synchronization and loop allocation overhead for perfectly balanced applications.

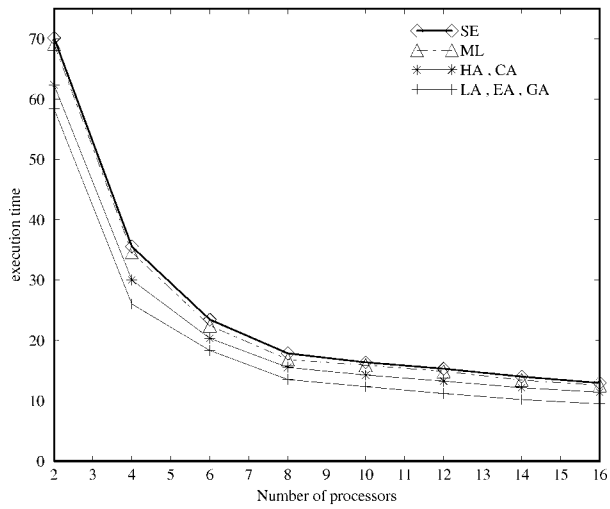
Fig. 2 plots the execution time of the Jacobi Iteration (JI) (L = 500, N = 1,024) for the different scheduling algorithms on the KSR-1 and on the Convex Exemplar. JI should be an application that fits the SE algorithm best. Since the workload distribution illustrates a "rectangular" shape—the leftmost 20% having a very heavy load and the remaining 80% having almost zero workload. The SE algorithm can readjust the initial partition to balance the workload for each processor to improve the execution time. The lower execution time curves of the SE algorithm confirm this.

Instead of readjusting the initial partition, our adaptive algorithms reduced the execution time by adjusting the chunk size for each processor. A lightly loaded processor took a larger number of iterations to execute. Then it turned to help the heavily loaded processor. The heavily loaded processor took a small number of iterations to execute, and it might leave some iterations for the other processors to finish.

For solving a linear system of size N = 1,024 by the JI kernel, our LA, GA, EA, and HA variations perform as well as the SE. The CA variation performed slightly worse than the other adaptive algorithms because, when using the CA variation, the processors with zero workload still cannot take more than $2/P$ iterations to execute. Therefore they need more time to finish their lightly loaded jobs, and turn to help the heavily loaded processor. In the meantime, the heavily loaded processor may have already taken a large number of jobs to execute and did not leave enough jobs for the idle processors.

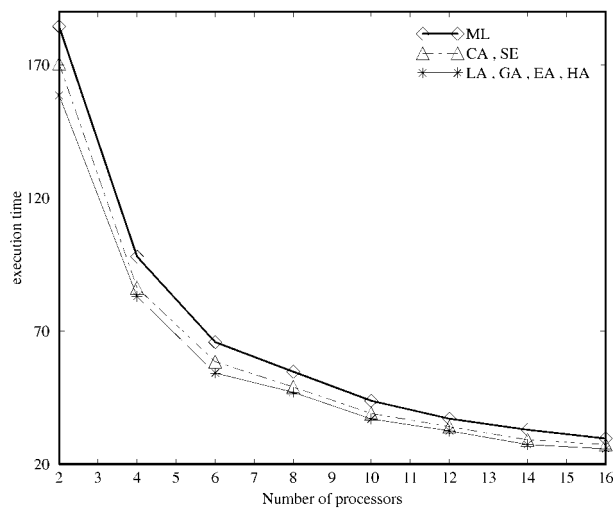


(a)

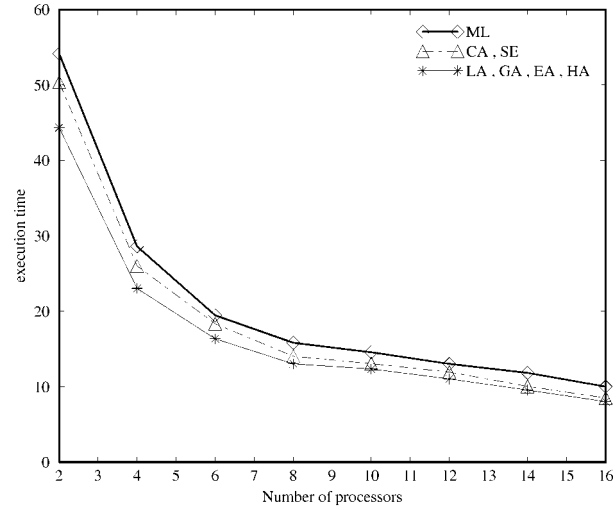


(b)

Fig. 3. Performance of TC with random input on the KSR-1 (a) and the EXEMPLAR (b).



(a)

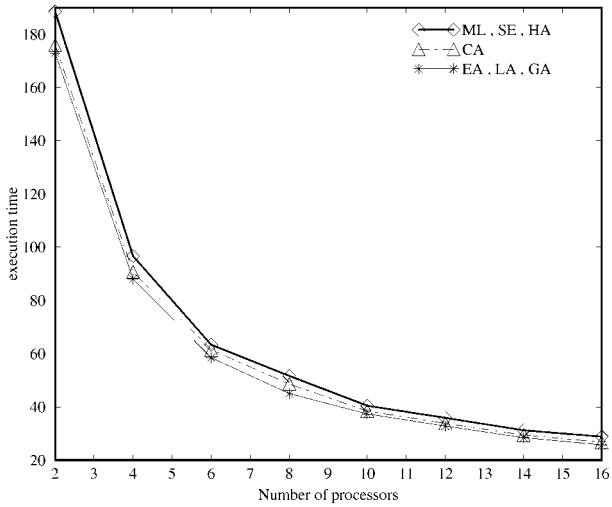


(b)

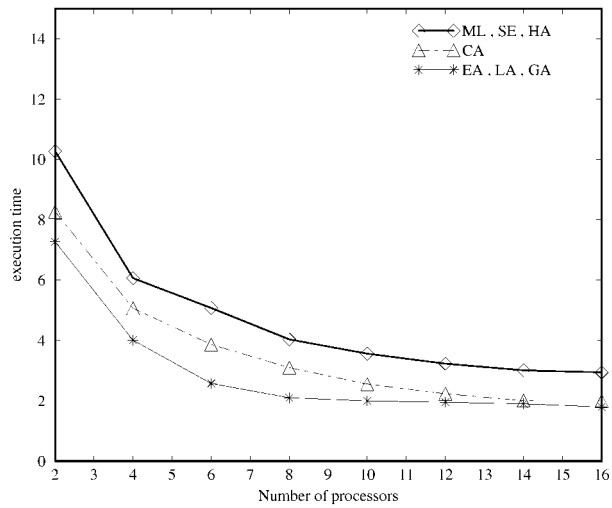
Fig. 4. Performance of TC with skewed input on the KSR-1 (a) and the EXEMPLAR (b).

Fig. 3 presents the execution time of the transitive closure kernel with a random input graph of 1,024 nodes, where about 10% of the edges are uniformly presented. In each execution of the parallel loop, the workload is uniformly distributed among iterations. However, the total workload increases at the next execution of the parallel loop. Fig. 3a and Fig. 3b show the comparative performance of seven tested algorithms respectively on both the KSR-1 machine and the Exemplar. The SE algorithm and the ML algorithm perform similarly because, in this case, the SE algorithm had little chance to improve the ML algorithm by readjusting the load distribution. Algorithms LA, EA, and GA performed the best among them all because they adjusted scheduling granularity more aggressively. Combining these results with the experimental results of SOR, we conclude that for load balanced applications, aggressively adjusting scheduling granularity is an efficient method to reduce scheduling and synchronization overhead, thus to improve performance well. These results also show that the overhead of collecting state information is not significant comparing with the benefit gained from adaptively adjusting scheduling granularity.

Again, we tested the scheduling algorithm and its variations for the transitive closure kernel with a skewed input graph of 640 nodes containing a clique of 320 nodes, and no other edges. In this case, load imbalance is significant in the computation across iterations and the total load of the parallel loop increases from one execution to the next. The execution times for each scheduling algorithm are presented in Fig. 4a and Fig. 4b. Although the authors of [9] claim that the SE algorithm assumes that the execution time of any particular iteration does not vary widely from one execution of the loop to another, our results show that the SE algorithm can still improve the ML algorithm in our case studies. Because our adaptive algorithm capture the variance in load more precisely than the SE algorithm, LA, EA, GA, and HA, performed better than the SE algorithm. The CA variation performed similarly to the SE algorithm.

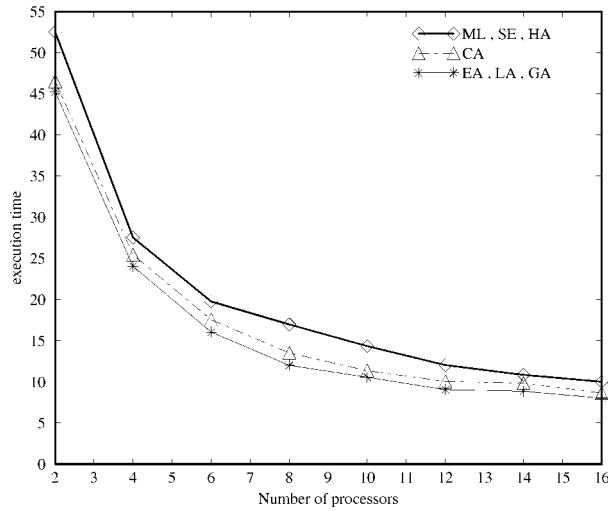


(a)

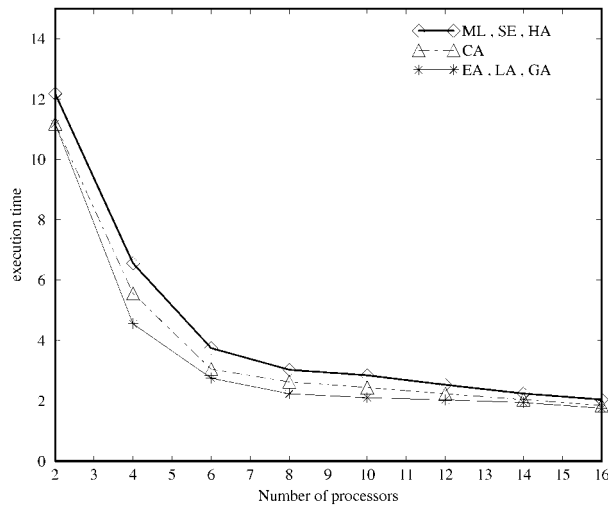


(b)

Fig. 5. Performance of MM on the KSR-1 (a) and the EXEMPLAR (b).



(a)



(b)

Fig. 6. Performance of AC on the KSR-1 (a) and the EXEMPLAR (b).

These experimental results show that adaptively adjusting scheduling granularity is an efficient way to handle the load imbalance in unpredictable loop applications.

If the parallel loop is not embedded in a sequential loop (we call it a nonaffinity loop), both the SE algorithm and our heuristic variation HA have no chance to improve the ML affinity algorithm. because they adjust the initial partition or adjust the chunk size near the end of one execution of the parallel loop, and hope that the new partition or the new chunk size can play a role in the next execution of the parallel loop. Now we want to see if other adaptive variations can perform better than the ML algorithm for the nonaffinity loop.

Fig. 5a and Fig. 5b present the performance of the scheduling algorithms for the matrix multiplication (MM) with N = 512. Algorithms ML, SE, and HA performed similarly. Algorithms EA, LA, and GA dynamically detect the workload distribution conditions and rapidly increase the chunk size, so that the processors take all the remaining iterations to execute after only a few accesses to the local work queue. The CA variation also increases the chunk size to a limit ($2/P$ of the remaining iterations); therefore, it involves less synchronization and loop allocation overhead than ML but presents more overhead than GA, LA, and EA. Compared with the experimental results on kernel SOR, adaptive variations do not improve the ML algorithm on MM significantly because the parallel loop only executes for one time.

Fig. 6a and Fig. 6b present the performance of the scheduling algorithms for kernel Adjoint Convolution with N = 128. SE and HA could not improve the ML algorithm, since the parallel loop is not embedded within a sequential loop. Load imbalance across iterations was significant since the first iteration took time proportional to $O(N^2)$, while the last iteration took time proportional to $O(1)$. As expected, ML, SE, and HA performed similarly, while EA, LA, and GA performed the best among them all. The CA variation's performance was in between.

4.2 Determine the Cost-Efficient Value

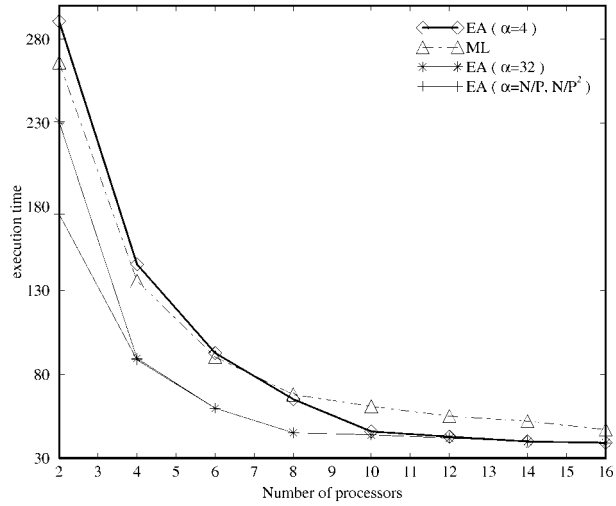
In the previous section, we used N/P^2 as the value of α in our adaptive scheduling algorithm and its variations,

where N is the number of iterations in the parallel loop and P is the number of processors we used to execute the parallel loop. Here, we tested several values of α in trying to give an optimal value.

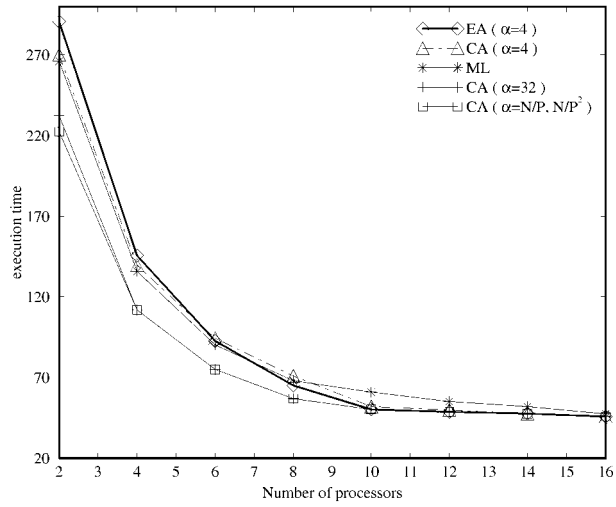
We evaluated our adaptive scheduling algorithm and its variations with different α values for the five benchmark applications on both the KSR-1 and the Exemplar. The α values we selected to evaluate are N/P, N/P^2, 32, and 4, respectively. Due to the space limitation, we only present part of the results here for two of our adaptive scheduling variations EA and CA with respect to one kernel application. The remaining results that we do not specify in what follows also support the conclusions we are going to present.

Fig. 7a presents the performance of the SOR kernel on the KSR-1 using the EA adaptive variation with the different α values. We also present the performance of the ML algorithm running the SOR kernel for a comparison. EA with $\alpha = N/P$ and $\alpha = N/P^2$ showed the best performance. While EA with $\alpha = 4$ presented the worst performance. Since SOR is a well-balanced application, all the processors should have a normal workload. A very large value of α like N/P guarantees that the workload state of each processor is always "normal" so that the processor can increase its chunk size and reduce its execution time. Although SOR is well-balanced, sometimes events such as cache misses, page faults, and interprocessor communication delays can bring some execution time variance among iterations. If we use a very small value for α, such as $\alpha = 4$, in the presence of interference from such kinds of events, some processors take their workload states as "heavy" and, therefore, decrease their chunk size by a factor of two. Since we do not give a limit for the chunk size for the EA variation, this decrease of chunk size at an exponential rate may cause some processors to take a very small chunk so that the processor may take only one iteration for each access to the local work queue (similar to *self-scheduling*). This is why the EA (with $\alpha = 4$) spent much more time than the ML when the number of processors is two or four. When the number of processors increases to six and eight, $\alpha = 4$ becomes close to N/P^2. The same reason holds with $\alpha = 32$ for EA. When $P = 2$, the processors cannot determine their workload states correctly due to the system interference and the small value of α. As the number of processors increases, the $\alpha = 32$ gets close to the value of N/P^2. Therefore, we had the good performance of the algorithms with number of processors four, six, and eight.

Fig. 7b presents the performance of the SOR application on the KSR-1 using the CA adaptive variation with different α values. We also show the curve for the ML algorithm and the curve for the EA variation with $\alpha = 4$ in order to compare them. Fig. 7b shows that CA with $\alpha = N/P$ and $\alpha = N/P^2$ performed the best among them all. CA with $\alpha = 4$ shows that too small a value of α (in comparison with the value of N/P^2) may cause a negative effect on the performance of our adaptive algorithm due to system interference. CA with $\alpha = 4$ performed the worst among the other CA curves. But we also notice that this curve is much lower than that of the EA with $\alpha = 4$. The reason is that we limit the range of chunk size for the CA variation within



(a)



(b)

Fig. 7. Performance of SOR on the KSR-1: using EA with different α values (a); using CA with different α values (b).

[P/2, 2P]. It guarantees that the processor takes at least $\frac{1}{2P}$ of the remaining iterations to execute each access to the local work queue.

5 Conclusion

By adaptively adjusting the loop allocation granularity according to the workload and execution speed of each processor, our loop scheduling algorithm demonstrates better performance than the affinity scheduling algorithm proposed by Markatos and Leblanc in [6] and the dynamic partitioned affinity scheduling algorithm proposed by Subramaniam and Eager in [9]. The authors had shown that the two algorithms presented the best performance among all the loop scheduling algorithms. Our adaptive scheduling algorithm is suitable for a wider range of application programs. They can reduce the execution time not only for well load-balanced parallel loops, but also for those load unbalanced parallel loops. Our experiments show that the overhead caused by collecting state information is not sig-

nificant comparing with the benefit gained. One important conclusion from this research is that efficiently using runtime information can significantly improve the efficiency of loop scheduling algorithms.

Among the variations of the adaptive scheduling algorithm, the EA, LA, and GA variations always demonstrate better performance than the CA and HA variations. Although EA, LA, and GA have higher risk than CA in terms of causing the load-imbalance, and in terms of being much more sensitive to the system interference, we have not observed the worst performance phenomena in our case studies, such as Ping Pong effect where the state of a processor is often switched between the lightly loaded and the heavily loaded, to cause overwhelmed scheduling overhead. In addition, the negative effect of the EA, LA, and GA variations can be significantly reduced by selecting the appropriate workload control constant α as N/P^2. Currently, we are developing an analytical model to determine optimal values of α.

Machine architecture may be another important factor that affects the performance of loop scheduling algorithms. So far, we are only able to test our adaptive algorithm and its variations on the KSR-1 and the Exemplar. Our experimental results indicate that the algorithm's performance is quite independent to shared-memory architectures. However, the effectiveness of the adaptive algorithm is significantly affected by the system size. When the system size scales very large, the cost to collect runtime information increases so that the advantages of the adaptive algorithm is nullified by the increased overhead. So, the adaptive algorithm is very suitable for scheduling the parallel loops over a small number of processors.

ACKNOWLEDGMENTS

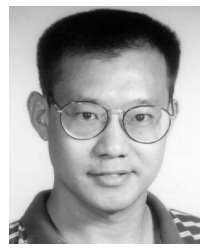
We appreciate Neal Wagner's and Samir Das' careful reading of the manuscript and constructive comments. We wish to thank the anonymous referees for their helpful comments and suggestions. This work is supported in part by the U.S. National Science Foundation under grants CCR-9102854 and CCR-9400719, by the U.S. Air Force under research agreement FD-204092-64157, and by the Air Force Office of Scientific Research under grant AFOSR-95-1-0215.

REFERENCES

[1] *CONVEX Exemplar Architecture*. CONVEX Computer Corp., second edition, document no. 710-004730-001, Nov. 1994.
[2] S.E. Hummel, E. Schonberg, and L.E. Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops," *Comm. ACM*, vol. 35, no. 8, pp. 90-101, 1992.
[3] *KSR-1 Technology Background*. Kendall Square Research, 1992
[4] S. Lucco, "A Dynamic Scheduling Method for Irregular Parallel Programs," *Proc. ACM SIGPLAN '92 Conf. Programming Language Design and Implementation*, pp. 200-211, 1992.
[5] J. Liu, V.A. Saletore, and T.G. Lewis, "Safe Self-Scheduling: A Parallel Loop Scheduling Scheme for Shared-Memory Multiprocessors," *Int'l J. Parallel Programming*, vol. 22, no. 6, pp. 589-616, 1994.
[6] E.P. Markatos and T.J. Leblanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 4, pp. 379-400, Apr. 1994.
[7] L.M. Ni and C.E. Wu, "Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor System," *IEEE Trans. Software Eng.*, vol. 15, no. 3, pp. 327-334, Mar. 1989.
[8] C. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Self-Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. Computers*, vol. 36, no. 12, pp. 1,425-1,439, Dec. 1987.
[9] S. Subramaniam and D.L. Eager, "Affinity Scheduling of Unbalanced Workloads," *Proc. Supercomputing '94*, pp. 214-226, 1994.
[10] P. Tang and P.C. Yew, "Processor Self-Scheduling for Multiple Nested Parallel Loops," *Proc. 1986 Int'l Conf. Parallel Processing*, pp. 528-535, 1986.
[11] T.H. Tzen and L.M. Ni, "Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers", *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 1, pp. 87-98, Jan. 1993.

Yong Yan is a PhD candidate in computer science at the University of Texas at San Antonio. He received the BS and MS degrees in computer science from Huazhong University of Science and Technology, Wuhan, China, in 1984 and 1987, respectively. He has been a faculty member there since 1987. He was a visiting scholar in the High Performance Computing and Software Laboratory at UTSA from 1993 to 1995. Since 1987, he has extensively published in the areas of parallel and distributed computing, performance evaluation, operating systems, and algorithm analysis. He is a member of the IEEE Computer Society and the Association for Computing Machinery.

Canming Jin received BS and MS degrees in computer science from Huazhong University of Science and Technology, Wuhan, China, in 1984 and 1987, respectively, and a second MS degree in computer science from the University of Texas at San Antonio in 1995. Before she came to the United States, she lectured in the Department of Computer Science in Huazhong University of Science and Technology, China, for about six years. She is currently working at InterVoice Inc., in Dallas, developing reliable distributed database systems. Her interests include high-performance software development, object-oriented programming methodology, and parallel computing.



Xiaodong Zhang received his BS degree in electrical engineering from Beijing Polytechnic University in 1982, and his MS and PhD degrees in computer science from the University of Colorado at Boulder in 1985 and 1989, respectively. He is an associate professor of computer science at the University of Texas at San Antonio, where he is directing the High Performance and Computing and Software Laboratory. His research interests are parallel and distributed computation, computer system performance evaluation, and scientific computing. Dr. Zhang is the current chair of the IEEE Technical Committee on Supercomputing Applications.