Adaptive Dynamic Process Scheduling on Distributed Memory Parallel Computers

WEI SHU

Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260

ABSTRACT

One of the challenges in programming distributed memory parallel machines is deciding how to allocate work to processors. This problem is particularly important for computations with unpredictable dynamic behaviors or irregular structures. We present a scheme for dynamic scheduling of medium-grained processes that is useful in this context. The adaptive contracting within neighborhood (ACWN) is a dynamic, distributed, load-dependent, and scalable scheme. It deals with dynamic and unpredictable creation of processes and adapts to different systems. The scheme is described and contrasted with two other schemes that have been proposed in this context, namely the randomized allocation and the gradient model. The performance of the three schemes on an Intel iPSC/2 hypercube is presented and analyzed. The experimental results show that even though the ACWN algorithm incurs somewhat larger overhead than the randomized allocation, it achieves better performance in most cases due to its adaptiveness. Its feature of quickly spreading the work helps it outperform the gradient model in performance and scalability. © 1994 by John Wiley & Sons, Inc.

1 INTRODUCTION

Large distributed memory parallel machines are becoming increasingly available. To efficiently use such large machines to solve an application problem, the computation must first be divided into parallel actions. These parallel actions are then mapped and scheduled onto processors.

Static, compile time allocation is one way to accomplish this. As a rather simple example, consider the problem of multiplying two 64×64 matrices on 16 processors. One may decide that each processor will compute a 16×16 submatrix of the result matrix by using appropriate rows and columns from the original matrix. This leads to 16 subcomputations, as desired, and either an automatic scheduler or a programmer can specify the appropriate data movement and computations.

Such static scheduling schemes cannot be used when the size of subcomputations cannot be accurately determined. In fact, in many computations, the subcomputations themselves are not known at compile time. Combinatorial search problems encountered frequently in AI provide an extreme example. Exploring a node in the search tree may lead to a large subtree search, may quickly lead to a dead end, or may lead to a solution. Even with deterministic computations, data dependencies and variable computational costs of operations lead to programs in which the detailed structure of computation cannot be predicted in

Received April 1994
Accepted May 1994

Scientific Programming, Vol. 3, pp. 341–352 (1994)
CCC 1058-9244/94/040341-12

advance. In such computations, one cannot divide the work into N equal parts, where N is the number of processing elements (PEs) in the system because the computational costs of subtasks cannot be predicted accurately. A reasonable strategy for such computations is to divide the work at run-time into many ($\gg N$) smaller granules and attempt to dynamically distribute them across the processors of the system. The grain size must be large enough to offset the overhead of parallelization. There are systems, such as the chare kernel described in the next section, that can support a grain size as small as a few milliseconds. Partitioning an application with small grain size would provide a large pool of work. Thus, even if the amount of computation within individual granules may vary unpredictably, it at least becomes possible to move these granules among processors to balance the load.

A scheduling scheme in such a context must deal with dynamic creation of work. It must cope with work generation and consumption rates that vary from processor to processor and from time to time. It cannot be a centralized scheme as it must work with a large number of processors and must scale up to a larger future system. Rather, it must be a distributed scheme, in which each processor participates in realizing the load-balancing objectives.

Obviously, a static scheduling scheme cannot be used for a computation that involves dynamic creation of work. However, a dynamic scheduling scheme can also be used for statically allocatable computations, such as the matrix multiplication problem mentioned above. In fact, a good dynamic scheduler may perform better than static schedulers even in some statically schedulable computations because it will automatically adapt to variable speeds of processors and to variable numbers of processors.

In this article we describe a dynamic and distributed scheduling scheme called adaptive contracting within neighborhood (ACWN). The next section discusses background and context in which the scheme is to operate and outlines basic issues. Section 3 describes some algorithms with similar objectives. Section 4 presents the ACWN algorithm and compares three different scheduling algorithms. Performance evaluation is given in Section 5, showing that ACWN maintains good load balance with low overhead. In Section 6, we discuss why the ACWN algorithm outperforms the other algorithms.

2 BACKGROUND

The chare kernel is a run-time support system that is designed to support machine-independent parallel programming [1–4]. The kernel is responsible for dynamically managing and scheduling parallel actions, called chares. A chare—the work stands for a small chore or task—is a process with some specific properties. Programmers use kernel primitives to create instances of chares and send messages between them, without concerning themselves with mapping these chares to processors or deciding which chare to execute next. Chares have some properties that distinguish them from processes in general. On creation and on receipt of a message, chares usually execute for a relatively short time. They may create other chares or send messages to existing ones. Having processed a message, the chare suspends, to be awakened by another message meant for it. These characteristics simplify the scheduling of chares considerably.

In this article the chare kernel concepts and terminology are used in discussing dynamic scheduling strategies. However, it should be clear that the scheduling strategies that are applicable in this context can also be used in other contexts that involve dynamic creation of small-grained tasks. For example, the REDIFLOW system [5] for applicative programming, other parallel implementations of functional languages, rewrite systems and logic languages, and actor-based languages such as Cantor [6], can all benefit from such strategies.

Many previous research efforts have been directed towards the task allocation in distributed systems [7–17]. Although some basic ideas can be shared, these strategies cannot simply be applied to multicomputer networks. A recent comparison study of dynamic load-balancing strategies on highly parallel computers is given by Willebeek-LeMair and Reeves [18]. Work with a similar assumption as mentioned in this article includes the gradient model developed by Lin and Keller [19]. Athas and Seitz also point out that random placement can be a quite simple and effective strategy [20, 21]. These strategies are discussed in the next section.

A chare instance goes through three phases in its life cycle: the allocating phase, the standing phase, and the active phase. It is in the allocating phase upon its creation until it enters in a pool of chares at some PE, and to be in the standing phase until it starts execution for the first time. Then the active phase begins. Opportunities for chare scheduling exist in all three phases but with different cost and effectiveness. The allocating phase strategies as well as standing phase strategies are instances of placement strategies. The active phase can also be used for scheduling. Strategies that move a chare in the active phase are called migration strategies. Because the grain size of chares is not large, migration is expensive and not necessary for load balance. Hence, this strategy is not considered in this article.

Scheduling strategies can also be classified based on the amount of load information they use. The "load" measure may include the number of messages waiting to be processed, the number of active chares, available memory, etc., possibly in a weighted combination. For the following discussion, the specific load measure is unimportant. The scheduler at a PE may periodically collect information from other PEs to calculate its own "status" information on which the scheduling decision is based. The strategies can be classified as follows:

1. Type I strategies involve using no status information.
2. Type II strategies calculate the status information by using local load information only.
3. Type III strategies calculate the status information by collecting load information from neighbors.
4. Type IV strategies calculate the status information by collecting status information from neighbors.
5. Type V strategies calculate the status information by collecting load information from all the PEs in the system.

Type I and II strategies typically have low overhead. The randomized allocation to be discussed in Section 3 is an example of a Type I strategy. It is believed that a strategy that adapts to variations in the system is necessary, and using local information alone is not sufficient to judge such variations. Type V strategies, on the other hand, are expensive in large systems and are not scalable.

The algorithm developed in this article is a Type III strategy in which the status information of a PE may be determined based on load information from itself and from its neighbors. The gradient model to be described in the next section is a Type IV strategy. The status information of a PE is determined from its neighbors' status information. Thus, the status of a PE depends on its neighbors, and theirs, in turn, depend on their neighbors. However, the time required to exchange information causes the status to be dependent on possibly outdated information.

3 RANDOMIZED ALLOCATION AND GRADIENT MODEL

Athas and Seitz [20, 21] have proposed a global randomized allocation algorithm. The randomized allocation is an allocating phase scheduling strategy and no standing phase action is involved. A randomized allocation algorithm dictates that each PE, when it generates a new chare, should send it to a randomly chosen PE. One advantage of this algorithm is simplicity of implementation. No local load information needs to be maintained nor is any load information sent to other PEs. Statistical analysis shows that the randomized allocation has a respectable performance as far as the number of chares per PE is concerned. However, a few factors may degrade the performance of the randomized allocation. First, the grain size of chares may vary. Even if each PE processes about the same number of chares, the load on each PE may still be uneven. Second, the lack of locality leads to large overhead and communication traffic. Only $1/N$ subtasks stay on the creating PE, where N is the number of PEs in the system. Thus, most messages between chares have to cross processor boundaries. The average distance traveled by messages is the same as the average internode distance of the system. This leads to a higher communication load on large systems. Because the bandwidth consumed by a long-distance message is certainly larger, the system is more likely to be communication bound compared to a system using other load-balancing strategies that encourage locality. Eager et al. [8] have modified the naive randomized allocation algorithm. They use threshold, a kind of local load information, to determine whether to process a chare locally or locate a chare randomly.

The gradient model [19] is mainly a standing phase scheduling strategy. As stated by Lin [22], instead of trying to allocate a newly generated chare to other PEs, the chare is queued at the generating PE and waits for some PE to request it. A separate, asynchronous process on each PE is responsible for balancing the load. This process periodically updates the state function and proximity on each PE. The state of a PE is decided by two parameters, the `low_water_mark` and `high_water_mark`. If the load is below the `low_water_mark`, the state is idle. If the load is above the `high_water_mark`, the state is abundant. Otherwise, it is neutral. The proximity of a PE represents an estimate of the shortest distance to an idle PE, which has a proximity of zero. For all other PEs, the proximity is one more than the smallest proximity among the nearest neighbors. If the calculated proximity is larger than the network diameter, it is in saturation and the proximity is set to be *network-diameter* + 1, to avoid unbounded increase in proximity values. If the calculated proximity is different from the old value, it is broadcast to all the neighbors. Based on the state function and the proximity, this strategy is able to balance the load between PEs. When a PE is not in saturation and its state is abundant, it sends a chare from its local queue to the neighbor with the least proximity.

The gradient model may cause load imbalance. For a tree-structured computation, this behavior could cause the upper-level nodes to cluster together near the root PE. When the results need to be collected at the root of the computation tree, the computation slows down. Furthermore, the proximity information may be inaccurate because of communication delays and the nature of the proximity update algorithm: By the time the proximity information from an idle PE propagates through the majority of PEs in a system, the state of the original PE may have changed.

4 ACWN

ACWN is a scheduling algorithm using the Type III strategy. Here, each PE calculates its own load function by combining various factors that indicate its current load. A simple measure may be the number of messages waiting to be processed. Adjacent PEs exchange their load information periodically by sending a small load message or piggybacking the load information with regular messages. Thus, each PE maintains load information on all its nearest neighbors. For PE k, its own load function is denoted by $F(k)$, and its neighbors' load functions are denoted by a set of values $F'(i)$, where $dist(k, i) = 1$. The value of $F(k)$ is calculated periodically.

The load information can then be used to determine a system state. For each PE k, a function

Table 1. System States

State	
Light load	$B(k) <$ `low_mark`
Moderate load	`low_mark` $\le B(k) <$ `high_mark`
Heavy load	`high_mark` $\le B(k)$

$B(k)$ is defined as $Min_{dist(k,i)=1}\{F'(i)\}$, which represents how heavily its neighbors are loaded. Two predefined parameters, `low_mark` and `high_mark`, are used to compare with $B(k)$ to ascertain the current system state as shown in Table 1. If $B(k) <$ `low_mark`, the system is considered to be in the light-load state. If $B(k) \ge$ `high_mark`, it is in the heavy-load state. Otherwise, it is in the moderate-load state.

The ACWN scheduling consists of both allocating phase and standing phase strategies. The allocating phase strategy is called contracting and the standing phase strategy is called redistributing.

As mentioned before, a chare is in its allocating phase from the time it is created until it enters the local queue at a PE. The allocating phase strategy of the ACWN algorithm is shown in Figure 1. During this phase, a newly created chare is contracted m hops, where $0 \le m \le d$ and d is the network diameter. We set an upper limit of traveling distance d for each allocating chare to prevent unbounded message oscillation. The contracting decision is based on the system state of each PE. The number of hops traveled so far for each chare c is recorded as $c.hops$. Thus, at each PE k, for an allocating chare c, which either is created by PE k or received from other PEs, there exist the following cases: If the system is in the heavy-load state or $c.hops \ge d$, chare c will be retained locally and added to the local pool of messages, terminating its allocating phase; if the system is in the light-load state and $c.hops = 0$, PE k will contract chare c to its least-loaded neighbor no matter what its own load is. Otherwise, the chare will be contracted conditionally: If the load on the least-loaded neighbor is smaller than its own load, the chare is contracted out to that neighbor. In this way, the newly generated chare c travels along the steepest load gradient to a local minimum.

The standing phase strategy of the ACWN algorithm is shown in Figure 2. Load imbalance may appear even though the allocating phase strategy is applied. Such imbalance may appear either due to limitations of the underlying load contracting scheme, which finds only a local minimum, or due

```
Contracting(k) /* at PE k */
  For each chare c with c.state = allocating do
    /* chare c may be either newly created or newly arrived */
    if heavy-load or (c.hops >= d)
      c.state = standing
      enter chare c into the local queue
    else
      find i for all j's such that F'(i) <= F'(j), where dist(i,k)=dist(j,k)=1
      if (light-load and c.hops = 0) or (moderate-load and F(k) > F'(i))
        c.hops = c.hops + 1
        send chare c to PE i
      else
        c.state = standing
        enter chare c into the local queue
      endif
    endif
```

FIGURE 1 The allocating phase strategy for the ACWN algorithm.

to the different rates of consumption of chares. Moreover, because each PE has its own system state, it is possible that there exist PEs in the light-load state, moderate-load state, or heavy-load state at the same time in a system. During the heavy-load state, PEs accumulate chares without sending them to any other PEs. Thus, after a PE leaves the heavy-load state, it may own many more chares than other PEs. These chares need to be redistributed to other PEs as the allocation of new chares alone may not be sufficient to correct the load imbalance. Notice that the redistributing is active only when a PE is not in its heavy-load state. In the heavy-load state, because all neighbors of the PE have sufficient work to do, it is not necessary to balance the load among them.

The behavior of both contracting and redistributing scheduling strategies is affected by the system state, which is determined by the load information as well as the predefined parameters, low_mark and high_mark. Low_mark is used to switch states between the light load and moderate load. If it is too high, chares are contracted out frequently and the overhead to move chares becomes higher. If it is too low, chares are spread out slowly and load imbalance may occur. High_mark is used to decide whether the system is heavily loaded, i.e., in saturation. If this mark is too high, the scheduling algorithm keeps moving chares among PEs even when they all have sufficient work, leading to higher overhead. However, if high_mark is set too low, the heavy-load state will be reached prematurely, which may cause load imbalance. Experiments suggested that performance is not sensitive as long as these parameters are in a reasonable range. As shown in Figures 3 and 4, the low_mark could be about 2 to 5 and the high_mark about 8. These experiments used the number of messages waiting to be processed as the measure of load. In the rest of the experiments with ACWN, we chose values of low_mark and high_mark as 2 and 8, respectively.

Scheduling strategies without migration can be summarized as a general model. The model consists of three functions. In the allocating phase, whether a chare is sent out depends on an allocating phase function. If the function is true, the

```
Redistributing(k) /* at PE k */
  For each time interval and when not in the heavy-load state
      find i for all j's such that F'(i) <= F'(j), where dist(i,k)=dist(j,k)=1
      if F(k) > F'(i)
          pick up a chare c from the local queue
                  with c.state = standing and c.hops < d
          c.hops = c.hops + 1
          send chare c to PE i
      endif
```

FIGURE 2 The standing phase strategy for the ACWN algorithm.

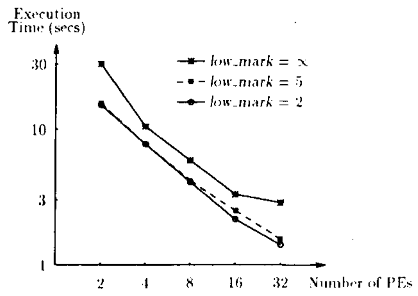

FIGURE 3 Low_mark effect on the performance for the 10-Queen problem.

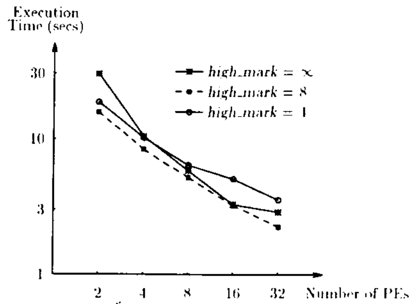

FIGURE 4 High_mark effect on the performance for the 10-Queen problem.

chare is sent out; otherwise it is kept local. In the standing phase, whether chares are moved depends on a standing phase function. If the function is true, the chares are redistributed between PEs. The third function is a destination function that determines which PE will receive a chare when the chare is to be allocated or redistributed. Different scheduling strategies can set different values for each of the three functions. If a scheduling strategy sets the allocating phase function always false, it is considered to be inactive during the allocating phase. Similarly, if a scheduling strategy sets the standing phase function always false, it is said to be inactive in the standing phase. To compare the randomized allocation, gradient model, and ACWN algorithms under this general model, we list three functions for each of them in Table 2. For the gradient model, $P(k)$ represents the proximity at PE k and d is the network diameter. For the randomized allocation, whenever PE k generates a new chare, a random number m is obtained to determine its allocating phase function as well as its destination function, where $0 \leq m < N$ and N is the number of PEs in the system. If m is equal to k, the allocating phase function is false. Otherwise, the chare will be sent to PE m as its destination.

The gradient model has virtually no allocating phase action. When a chare is generated, it is put in the local queue. This leads to slow spreading of the load. On the other hand, the randomized allocation does not have standing phase action. It usually generates good distribution of the load. However, when the sizes of chares vary in a wide range, this strategy is unable to redistribute the load even if some PEs are busy and others are idle. ACWN conducts the actions in both phases, resulting in a more reliable performance.

5 PERFORMANCE STUDIES

We have tested several examples on an Intel iPSC/2 hypercube to study the effectiveness of dynamic scheduling schemes on multicomputers. The machine used has a 32-node configuration

Table 2. The Allocating Phase, Standing Phase, and Destination Functions of PE k

	Randomized Allocation	Gradient Model	ACWN
Allocating phase function	True	False	It is true if (light load and $c.hops = 0$) or ((moderate load or (light load and $c.hops > 0$)) and $F(k) > min(F'(i))$), where $dist(i, k) = 1$ otherwise, it is false
Standing phase function	False	It is true if $P(k) \leq d$ and abundant and $P(k) > min(P(i))$, where $dist(i, k) = 1$ otherwise, it is false	It is true if not heavy load and $F(k) > min(F'(i))$, where $dist(i, k) = 1$; otherwise, it is false
Destination function	Random	j, where $P(j) = min(P(i))$, and $dist(i, k) = dist(j, k) = 1$	j, where $F'(j) = min(F'(i))$, and $dist(i, k) = dist(j, k) = 1$

with 4 megabytes of memory at each node. Three algorithms, randomized allocation, gradient model, and ACWN, were implemented. They shared most subroutines except for the allocating phase function, the standing phase function, and the destination function. Notice that the programs are chosen not because they are good parallel algorithms for the problems they solve, but for the suitability of illustrating different computation patterns handled by the dynamic scheduling. For each program the best sequential program written in C was also tested without changing the algorithm.

In general, the sum of execution times of all PEs can be broken into three parts: computation time, overhead, and idle time. Computation time is spent on problem solving and should be equal to the sequential execution time. This time is invariant with different scheduling strategies, different numbers of PEs, and different grain sizes. Overhead includes the work of bookkeeping, communication, and load balancing. Idle time is the time in which PEs have no work to do. The overhead and idle time depend on granularity of partitioning as well as scheduling strategy. Experiments for different grain sizes of the 10-Queen problem were conducted for analysis of the factors of granularity. Figure 5 shows the efficiency of this problem for different numbers of PEs with different grain sizes. The performance of the largest grain size slumps as the number of PEs increases because the pool of available chares is not large enough to keep all the PEs busy. The poor performance of the curve with the smallest grain size is due to overhead. For example, Figure 6 shows the components of the execution time for different grain sizes with 16 PEs. Here, a small grain size imposes a large amount of overhead. On the other

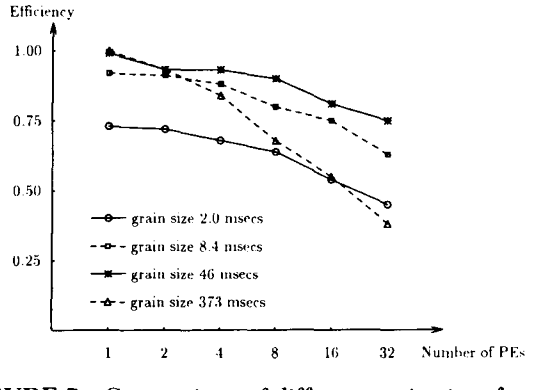

FIGURE 5 Comparison of different grain sizes for the 10-Queen problem.

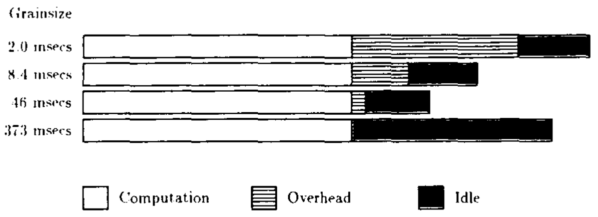

FIGURE 6 Overhead and idle time as grain size varies.

hand, a large grain size reduces overhead but may result in longer processor idle time because of load imbalance.

Bookkeeping overhead depends on the number of chares and the number of messages only. For each individual chare, the system maintains a chare block, and for each message there is a message header including its source and destination chare information. The overhead of bookkeeping is about 250–400 microseconds whenever a new chare is created or a message is sent. The communication overhead consists of the time spent by the processor that deals with sending and receiving messages. The actual transmission time is overlapped with computation and does not need to be considered. The overhead for each communication is about 450 microseconds. The granularity also affects communication overhead because the number of messages exchanged between PEs tends to increase when the grain size becomes smaller. Not all the messages between chares introduce communication overhead. Only those going to PEs other than the source PE have the result. Thus, the load-balancing strategies also influence the communication overhead, as different strategies have different effects on what fraction of the messages will be between local chares. Scheduling overhead can be divided into two parts: updating load information and chare placement. Time spent on chare placement is proportional to the number of chares and is determined by granularity. System load information can be exchanged periodically. As shown in Figure 7 for the 10-Queen problem on 16 PEs, too short a period increases communication overhead and too long a period leads to inaccurate load information due to sluggish updates. With a long exchanging period, the system acts unstably. We give both worst and best time from many repetitions of experiments for periods after 256 milliseconds. In Figure 7, two curves are shown, with and without piggybacking for different exchanging periods. Piggybacking load information on the regu-

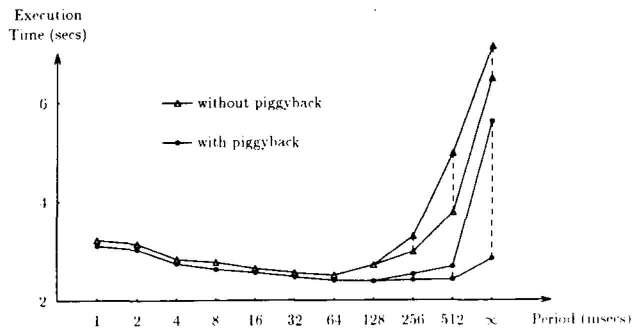

FIGURE 7 Comparison of different periods to exchange load information.

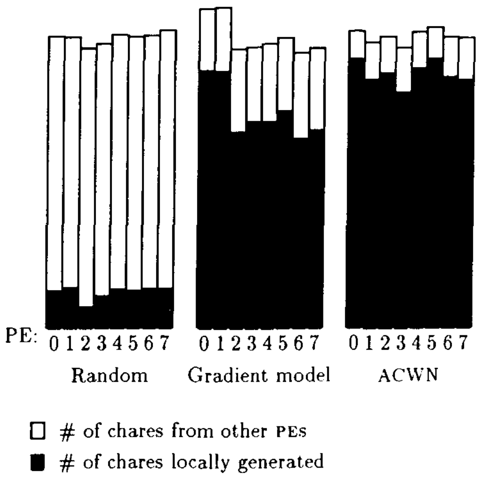

FIGURE 9 Distribution of chares over PEs

lar outgoing messages can reduce the number of load information messages exchanged. One with piggybacking behaves better than one without piggybacking because with every message we update load information with a negligible additional cost. Figure 8 selects one instance with piggybacking to show the sum of overhead and sum of idle time at all 16 PEs. A short exchanging period makes the frequently updated load information unnecessary. However, if the period is too long, load is highly unbalanced with long idle time. From the curves, it can be seen that the best period is between 50 and 150 milliseconds. In the rest of the experiments, piggybacking is applied to both the ACWN and the gradient model algorithms. The period of load information exchanging is set to be 100 milliseconds for ACWN and the best value of exchanging interval is also selected for the gradient model.

The influence of scheduling strategies will now be discussed. A good scheduling algorithm must be able to balance load for different application problems. At the same time, it has to keep scheduling overhead small. Furthermore, it must keep good locality so that most chares can be executed locally to reduce communication overhead. Three scheduling algorithms are compared, randomized allocation, gradient model, and ACWN. Figure 9 lists chare distribution at each PE with different scheduling algorithms for Fibonacci 32 on 8 PEs. Each chare processed at PE k is either generated by PE k itself or from other PEs. The ACWN has the most locally generated chares and a few from other PEs. At the other extreme, the randomized allocation has a few local chares (about $1/N$) and most chares from other PEs.

The only scheduling overhead for the randomized allocation is to generate random numbers whenever a chare is created. However, communication overhead is high because most chares are sent to other PEs irrespective of whether the system is heavily or lightly loaded. For the same problem shown in Figure 9, Figure 10 illustrates percentage of computation time, overhead, and idle time. To compare the algorithms, the overhead time can be subdivided further into three subcategories: the bookkeeping overhead (T_B), communication overhead (T_C), and load-balancing overhead (T_L). Figure 11 extracts the overhead parts from Figure 10 and illustrates each kind of overhead for different algorithms. The random-

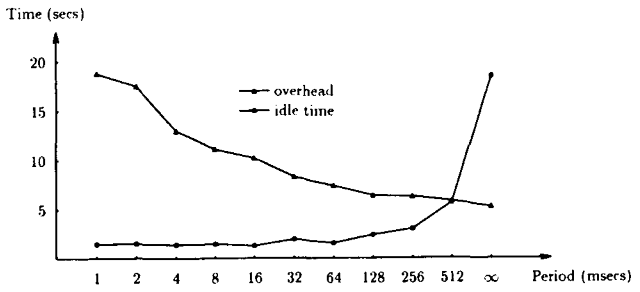

FIGURE 8 Total overhead and idle time at all 16 PEs.

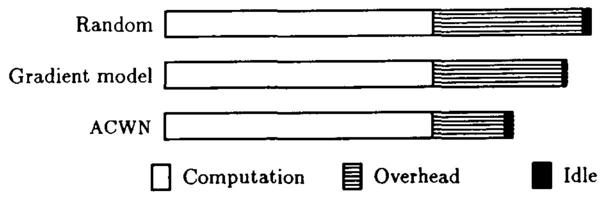

FIGURE 10 Timing analysis for different scheduling algorithms.

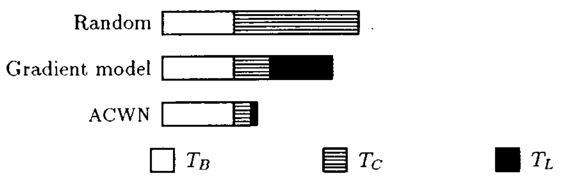

FIGURE 11 Three parts of overhead.

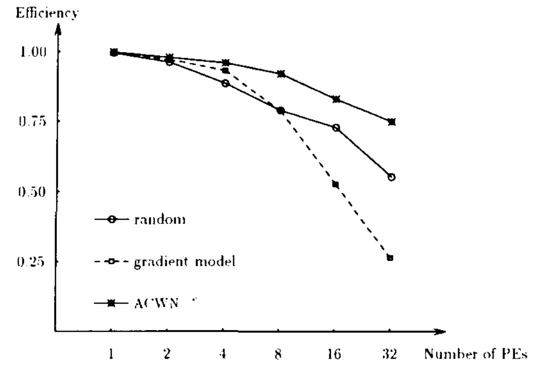

FIGURE 12 Comparison for the 10-Queen problem.

ized allocation has large overhead spent on communication although its scheduling overhead is negligible. The gradient model utilizes the system status information to make loads balanced among PEs so that the idle time is reduced. More importantly, the gradient model sends chares away only when necessary. Due to this locality property, the gradient model does not incur high communication overhead compared to the randomized allocation case. However, the gradient model must exchange load information more frequently to balance the load, resulting in large load balancing overhead. The ACWN exhibits better locality than the gradient model. Therefore, it has less communication overhead. Its scheduling overhead is also small due to a low frequency of load information exchange.

In Table 3 and Figures 12–15, we give the performance comparison of the randomized allocation, the gradient model, and the ACWN algorithms. Here, one instance of each program has been chosen for execution, i.e., 10 Queens, Fibonacci 32, one configuration of 15-puzzle, and the Romberg integration with 14 integrations. Characteristic features for different problems are shown in Table 4. The granularity is between 1 to 100 milliseconds, resulting from the medium-grained partitioning. Coarse granularity causes serious load imbalance and fine granularity leads to large overhead. The Fibonacci problem is a regular tree-structured computation. The grain sizes of leaf chares are roughly the same. In the Queen problem, the grain size is not even because whenever a new queen is placed, the search either successfully continues to the next row or fails. The 15-puzzle is a good example of an AI search problem. Here the iterative deepening A* algorithm is used [23]. The grain size may vary substantially because it dynamically depends on the current estimate cost. Also, synchronization at each itera-

Table 3. Performance on Intel iPSC/2 Hypercube: Execution Time (seconds)

	Number of PEs						
	Seq.	1	2	4	8	16	32
Queen 10							
Random	29.5	29.9	15.5	8.41	4.72	2.56	1.69
Gradient	29.5	29.9	15.4	7.98	4.74	3.53	3.54
ACWN	29.5	29.9	15.2	7.74	4.07	2.24	1.24
Fibonacci 32							
Random	30.0	36.2	21.2	11.4	5.99	3.21	1.73
Gradient	30.0	36.4	18.6	9.68	5.65	3.51	1.99
ACWN	30.0	36.4	18.4	9.51	4.89	2.52	1.36
15-puzzle/IDA*							
Random	50.2	50.9	29.3	16.2	10.8	8.17	5.17
Gradient	50.2	51.0	26.6	15.0	9.94	8.50	8.48
ACWN	50.2	50.9	27.2	14.9	8.55	5.52	4.11
Romberg integration 14							
Random	24.8	26.1	15.2	8.50	4.77	2.99	1.97
Gradient	24.8	26.1	14.5	8.46	5.76	4.55	4.51
ACWN	24.8	26.1	14.0	7.30	4.02	2.41	1.64

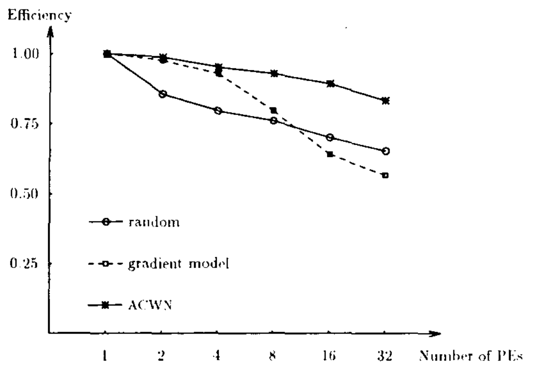

FIGURE 13 Comparison for the Fibonacci problem.

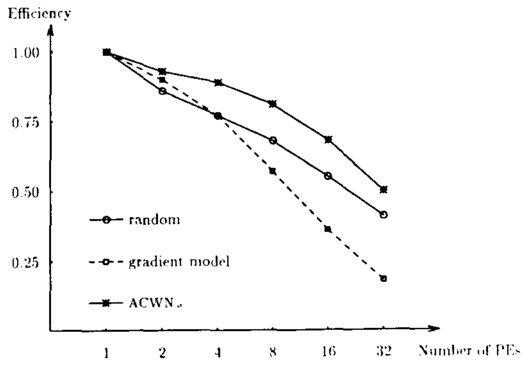

FIGURE 15 Comparison for Romberg Integration.

tion reduces the effective parallelism. Performance of this problem is therefore not as good as others. In the Romberg integration, the evaluation of function points at each iteration is performed in parallel. As we can see, ACWN is better than both the randomized allocation and the gradient model in all the cases.

6 DISCUSSION

The ACWN algorithm outperforms the randomized allocation and the gradient model partly due to its two-phase scheduling strategy and partly due to its adaptive locality. Its good locality reduces communication overhead whereas the randomized allocation does not. Besides the standing phase strategy, the allocating phase strategy of ACWN allows load to spread out faster than the gradient model. The ACWN can adapt to different chare sizes too. Assume at a time both PE i and PE j have m messages waiting for processing, respectively. It happens that PE i gets a message with a large amount of computation. After a while, PE i still holds $m - 1$ messages and PE j may have no messages left. At this time, ACWN is able to schedule messages from PE i to PE j to balance the load. In contrast, the randomized allocation cannot adapt to such a case.

For a small number of PEs, the gradient model can make better load balance than the randomized allocation. However, because the gradient model was designed based on good locality to reduce communication overhead, it does not spread the load very fast. For a large number of PEs, the gradient model leads to more load imbalance than the randomized allocation does. As shown in Figure 16 for the 10-Queen problem, the idle time of the gradient model at 16 and 32 PEs is longer than the randomized allocation. A similar conclusion is also made by Grunwald [24]. ACWN reaches the most even load distribution among the three scheduling algorithms.

From experiments, the overhead for a local chare that does not involve communication over-

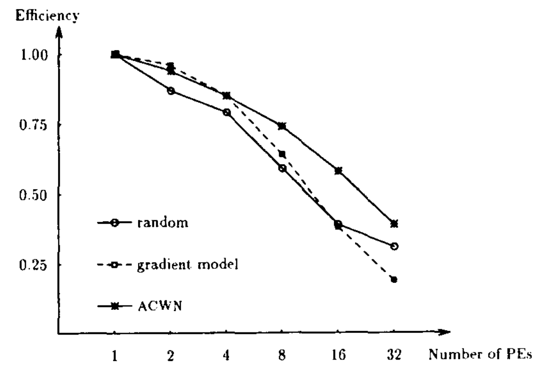

FIGURE 14 Comparison for 15-puzzle/IDA* problem.

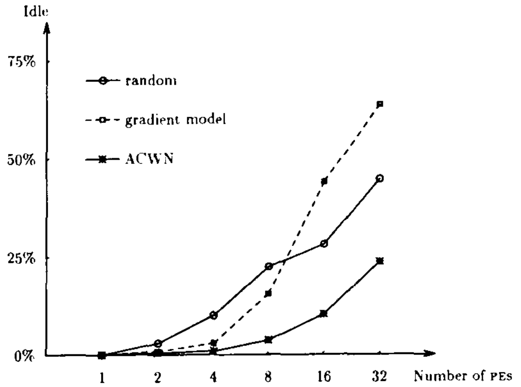

FIGURE 16 Comparison of PE idle time for different scheduling algorithms (10-Queen).

Table 4. Characteristics of Different Problems

Problems	Number of Chares	Execution Time per Chare (milliseconds)	Number of Messages	Execution Time per msg (milliseconds)
Queen 10	593	49.7	1186	24.9
Fibonacci 32	8361	3.59	16722	1.79
15-puzzle/IDA*	1172	42.8	2344	21.4
Romberg Integration 14	2026	12.2	4052	6.1

head is about 0.3 to 0.4 milliseconds and for a remote chare that involves communication overhead it is about 1.2 to 1.3 milliseconds. Thus, performance may not suffer much from bookkeeping and communication overhead if the grain size of a chare is much larger than that. A few 10-milliseconds can therefore be counted as a reasonable grain size. Due to a large number of remote chares, the communication overhead for the randomized allocation is large, which in turn implies a large grain size.

Does overhead of a complicated scheduling algorithm always overwhelm the benefit it achieves? Certainly, a complex algorithm (as an extreme example, one that looks for the least loaded processor across the entire system at every scheduling decision) loses its uniform distribution advantage to its high overhead. The randomized allocation algorithm bears negligible overhead for load-balancing decisions but the communication overhead is high and the suspension is large. We have shown that a good load balance can be obtained by a simple algorithm with low scheduling overhead. Even though ACWN pays more scheduling overhead compared to the randomized allocation, it still can achieve better performance in most cases.

Overhead can be reduced by using coprocessors. A coprocessor can be attached to the main processor in each PE, which handles all bookkeeping, load-balancing, and communication activities. In the iPCS/2 hypercube, each PE has a communication coprocessor that shares part of the communication overhead. Because we are not able to program coprocessors, overhead of bookkeeping, load balancing, and part of communication must be handled by the main processor. If the ACWN scheduling can be applied to a system with coprocessors, the frequency of load information exchange can be increased and more communication activities may take place to improve load balance, as long as the load of the coprocessor does not exceed the load of the main processor. The randomized allocation and the gradient model may benefit more from the coprocessor than ACWN does because the randomized allocation has more communication overhead and the gradient model has more scheduling overhead.

7 CONCLUSION

We described a scheme for dynamic scheduling of medium-grained processes on multicomputers. The scheme, called ACWN employs two substrategies: an allocating phase strategy and a standing phase strategy. The allocating phase strategy moves a new piece of work along the steepest load gradient to a local minimum within a neighborhood. It estimates the system state and ensures that pieces of work are moved only when the system requires it. The standing phase strategy corrects load imbalance by redistributing pieces of work that were initially allocated by the allocating phase strategy. Every processor maintains load information about their neighbors only, and such information is often exchanged by piggybacking it on regular messages. Thus, the scheme incurs low load balancing overhead. As it manages to retain many pieces of work on the processor that produced them, it has low communication overhead.

ACWN was compared with two other schemes, the randomized allocation and the gradient model. The randomized allocation incurs negligible load balancing overhead and achieves reasonably uniform distribution of work. However, it incurs much communication overhead. The gradient model, on the other hand, enforces locality at the expense of agility in spreading work out quickly to processors. All these schemes were implemented in a system called the chare kernel running on the Intel iPSC/2 hypercube. The experimental results demonstrate that ACWN performs better than the other two algorithms for many computation patterns.

ACKNOWLEDGMENT

The research was partially supported by NSF grant CCR-9109114.

REFERENCES

[1] W. Shu and L. V. Kale, "Chare Kernel—a runtime support system for parallel computations," *J. Parallel Distrib. Comput.*, vol. 11, pp. 198–211, 1991.

[2] W. Shu, "Chare Kernel and its Implementation on Multicomputers," PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, January 1990.

[3] W. Shu and L. V. Kale, *Supercomputing '89.* 1989, pp. 389–398. ACM Press.

[4] L. V. Kale and W. Shu, *International Conference on Parallel Processing.* 1989, pp. 118–121. CRC Press.

[5] R. Keller and F. C. H. Lin, "Simulated performance of a reduction based multiprocessor," *IEEE Comput.*, vol. 17, pp. 74–82, 1984.

[6] W. C. Athas and C. L. Seitz, "Cantor user report," Technical Report, Department of Computer Science, California Institute of Technology, January 1987.

[7] N. G. Shivaratri, P. Krieger, and M. Singhal, "Load distributing for locally distributed systems," *IEEE Comput.*, vol. 25, pp. 33–44, 1992.

[8] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 662–674, 1986.

[9] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *Performance Eval.*, vol. 6, pp. 53–68, 1986.

[10] J. A. Stankovic, "Simulations of three adaptive, decentralized controlled, job scheduling algorithms," *Comput. Networks*, vol. 8, pp. 199–217, 1984.

[11] T. L. Casavant and J. G. Kuhl, *International Conference on Distributed Computing System.* 1986, pp. 232–239. IEEE Computer Society Press.

[12] T. L. Casavant and J. G. Kuhl, *International Conference on Distributed Computing System.* 1987, pp. 185–192. IEEE Computer Society Press.

[13] A. Hac and X. Jin, *International Conference on Distributed Computing System.* 1987, pp. 170–177. IEEE Computer Society Press.

[14] V. Singh and M. R. Genesereth, *9th International Joint Conference on Artificial Intelligence.* 1985, pp. 39–45.

[15] Y.-T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Trans. Comput.*, vol. C-34, pp. 204–217, 1985.

[16] A. Barak and A. Shiloh, "A distributed load-balancing policy for a multicomputer," *Software Practice Exp.*, vol. 15, pp. 901–913, 1985.

[17] Z. Lin, "A distributed fair polling scheme applied to parallel logic programming," *Int. J. Parallel Programming*, vol. 20, 1991.

[18] M. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *J. Parallel Distrib. Comput.*, vol. 9, pp. 979–993, 1993.

[19] F. C. H. Lin and R. M. Keller, "The gradient model load balancing method," *IEEE Trans. Software Eng.*, vol. 13, pp. 32–38, 1987.

[20] W. C. Athas, "Fine grain concurrent computations," PhD thesis, Department of Computer Science, California Institute of Technology, May 1987.

[21] W. C. Athas and C. L. Seitz, "Multicomputers: Message-passing concurrent computers," *IEEE Comput.*, vol. 21, pp. 9–24, 1988.

[22] F. C. H. Lin, "Load balancing and fault tolerance in applicative systems," PhD thesis, Department of Computer Science, University of Utah, August 1985.

[23] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, pp. 97–109, 1985.

[24] D. C. Grunwald, "Curcuit switched multicomputers and heuristic load placement," PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, UIUCDCS-R-89-1514, September 1989.