Symmetrical Hopping: A Scalable Scheduling Algorithm for Irregular Problems

Min-You Wu
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260

Abstract — A runtime support is necessary for parallel computations with irregular and dynamic structures. One important component in the support system is the runtime scheduler which balances the working load in the system. We present a new algorithm, Symmetrical Hopping, for dynamic scheduling of ultra-lightweight processes. It is a dynamic, distributed, adaptive, and scalable scheduling algorithm. This algorithm is described and compared to four other algorithms that have been proposed in this context, namely the randomized allocation, the sender-initiated scheduling, the receiver-initiated scheduling, and the gradient model. The performance of these algorithms on Intel Touchstone Delta is presented. The experimental results show that the Symmetrical Hopping algorithm achieves much better performance due to its adaptiveness.

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. In many computations, the sub-computations 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 sub-tree search, may quickly lead to a dead-end, or may lead to a solution. In such computations, it is impossible to divide the work into N *equal* parts at compile-time, 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 into many ($\gg N$) small processes, 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 process kernel described in the next section, which can support a grain-size as small as a few milliseconds on Intel Touchstone Delta or IPSC/860. Partitioning an application with small grain-size would provide a large pool of work. Thus, even if the computation density within individual processes may vary unpredictably, it at least becomes possible to move these processes among processors to balance the load.

A scheduling algorithm in such a context must deal with dynamic creation of work. It must be a dynamic scheduling algorithm, since a static scheduling algorithm is not able to schedule a non-deterministic problems for a good performance. It cannot be a centralized algorithm 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 algorithm, in which each processor participates in realizing the load balancing objectives.

In this paper, we describe a dynamic and distributed scheduling algorithm called *Symmetrical Hopping*. The next section discusses programming environment in which the algorithm is to operate. Section 3 describes basic issues of scheduling algorithms. Section 4 presents the Symmetrical Hopping algorithm. Performance evaluation and a comparison to other scheduling algorithms are given in Section 5. In Section 6, we discuss the applicability of the Symmetrical Hopping algorithm.

2. Programming Model

The Symmetrical Hopping algorithm is implemented on top of the process kernel, which is a runtime support system designed to support machine independent parallel programming [1, 2]. The kernel is responsible for dynamically managing and scheduling *ultra-lightweight processes*, which are called *processes* for short in this paper. Programmers use kernel primitives to create processes and send messages between them, without concerning themselves with mapping these processes to processors, or deciding which process to execute next. A process may create other processes or send messages to existing ones. These characteristics simplify the scheduling of processes considerably.

The computation model for the process kernel is a message-driven, nonpreemptive, thread-based model. Here, a parallel computation will be viewed as a collection of *processes*, each of which in turn consists of a set of threads, called *atomic computations*. An atomic computation is a *run-to-completion* (RC) thread [3]. Once a RC thread starts execution, it will run to completion without being blocked. Processes communicate with each other via *messages*. Each atomic computation is then the result of processing a message. During its execution, it can create new processes or generate new messages [4]. A message can trigger an atomic computation, whereas an atomic computation cannot wait for messages. All atomic computations of the same process share one *common data area*. Thus, a *process* P_k consists of a set of *atomic computations* A_{k_i} and one *common data area* D_k:

$$P_k = \{D_k, A_{k_1}, A_{k_2}, ..., A_{k_n}\}, n \geq 1$$

Once a process has been scheduled to a processor, all of its atomic computations are executed on the same processor. There is no presumed sequence to indicate which atomic computation will be carried out first. Instead, it depends on the order of arrival of messages. Figure 1 shows the general organization of processes, atomic computations, and common data areas. In general, the number of processes is much larger than the number of processors so that the processes can be moved around to balance the load.

The process kernel is a runtime support system built to manipulate and schedule processes, as well as messages. A program written in the process kernel language consists mainly of a collection of process definitions and subroutine definitions. A process definition includes a process name preceded by the keyword **process**, and followed by the process body, as shown below.

process ProcName { <Common Data Area Declarations>
  entry LABEL1: (**message** msg1) <Code1>
  entry LABEL2: (**message** msg2) <Code2>
  ... }

Here, bold-face letters denote the keywords of the language. The process body, which is enclosed in braces, consists of declarations of private variables that constitute the common data area of the process, followed by a group of atomic computation definitions. Each atomic computation definition starts with the keyword **entry** and its label, followed by a declaration of the corresponding message and arbitrary user code. One of the process definitions must be the *main* process. The first entry point in the main process is the place the user program starts.

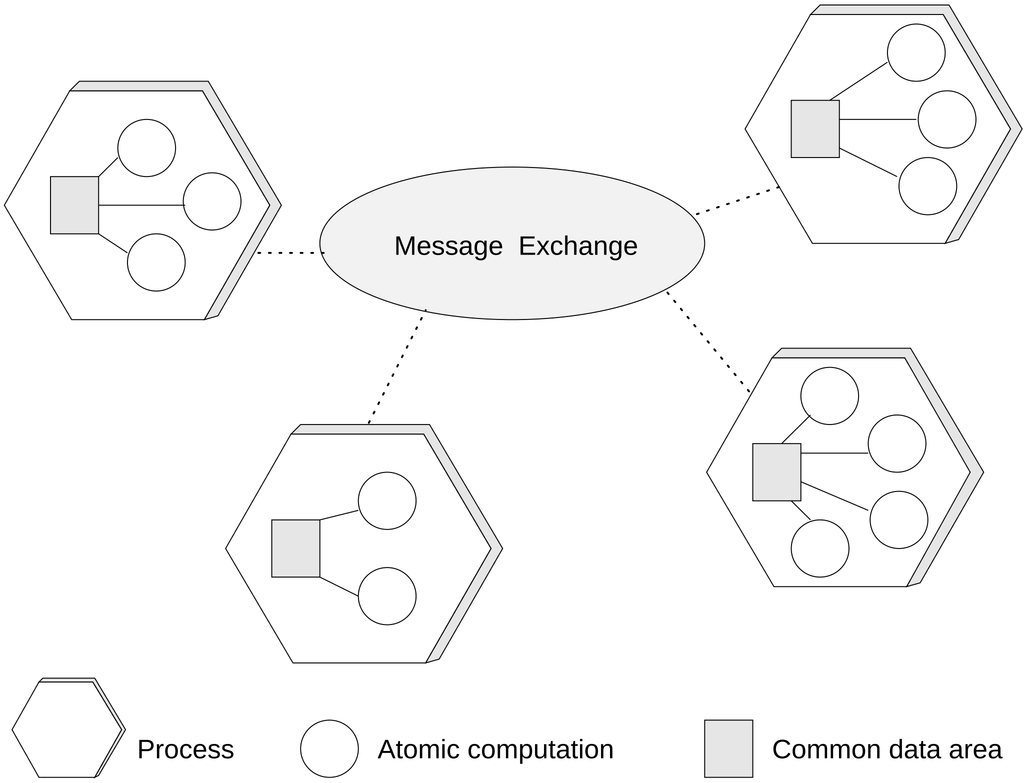

Figure 1: The structure of process, atomic computation, and common data area.

The overall structure of the process kernel language differs from that of the traditional programming languages mainly in explicit declarations of (a) basic units of allocation — processes, (b) basic units of indivisible computation — atomic computations, and (c) communication media — messages. With these fundamental structures available, computation can be carried out in parallel with the assistance of primitive functions provided by the process kernel, such as *OsCreateProc()*, *OsSendMsg()*, *etc.* The user can write a program in the process kernel language, deal with the creation of processes, and send messages between them. For details of the computation model and language, refer to [2].

In the following, we will illustrate how to write a program in the process kernel language using the exhaustive search of the N-queen problem as an example. The algorithm used here attempts to place queens on the board one row at a time if the particular position is valid. Once a queen is

```
Process MAIN
  { int solutionCount = 0; responseCount = 0;
  entry QueenInit: (message MSG1()) { int k;
    read N from input
    for (k = 1, N) {
      OsCreateProc(SUBQUEEN,ParallelQueen,MSG2(1,k,empty board));
      responseCount = N }
    }
  entry ResponseQueen: (message MSG3(m)) {
    solutionCount = solutionCount + m; responseCount−−;
    if (responseCount==0) {
      print "# of solutions =", solutionCount;
      OsKillProc() }
    }
  }

Process SUBQUEEN
  { int solutionCount = 0; responseCount = 0;
  entry ParallelQueen: (message MSG2(i,j,board)) { int k;
    invalidate row i, column j, and diagonals of (i,j)
    for (k = 1, N) {
      if (position (i+1,k) is marked valid) {
        if ((N-i) is larger than the grain-size)
            OsCreateProc(SUBQUEEN,ParallelQueen,MSG2(i+1,k,board));
        else
            OsCreateProc(SEQQUEEN,SequentialQueen,MSG2(i+1,k,board));
        responseCount++;
        }
      };
    if (responseCount==0) {
      OsSendMsg(ParentProcID(),ResponseQueen,MSG3(solutionCount));
      OsKillProc() }
    }
  entry ResponseQueen: (message MSG3(m)) {
    solutionCount = solutionCount + m; responseCount−−;
    if (responseCount==0) {
      OsSendMsg(ParentProcID(),ResponseQueen,MSG3(solutionCount));
      OsKillProc() }
    }
  }

Process SEQQUEEN {
  entry SequentialQueen: (message MSG2(i,j,board)) { int k, count;
    call sequential routine, recursively generating all valid configurations.
    OsSendMsg(ParentProcID(),ResponseQueen,MSG3(count));
    OsKillProc() }
    }
  }
```

Figure 2: The exhaustive search program (N-queen).

placed on the board, the other positions in its row, column, and diagonals, will be marked invalid for any further queen placement. The program is sketched in Figure 2. The atomic computation QueenInit in the MAIN process creates N processes of type SUBQUEEN, each with an empty board and one candidate queen in a column of the first row. There are two types of atomic computations in process SUBQUEEN: ParallelQueen and ResponseQueen. A common data area consists of solutionCount and responseCount. Each atomic computation ParallelQueen receives a message that represents the current placement of queens, and a position for the next queen to be placed. Following the invalidation processing, it creates new SUBQUEEN or SEQQUEEN processes by placing one queen in every valid position in the next row. The atomic computation ResponseQueen in processes SUBQUEEN and MAIN counts the total number of successful queen configurations. It can be triggered any number of times until there is no more response expected from its child process. The atomic computation SequentialQueen is invoked when the rest of the rows are to be manipulated sequentially. This is how granularity can be controlled. In this example, there are two process definitions besides that of process MAIN. Atomic computations that share the same common data area should be in a single process, such as ParallelQueen and ResponseQueen. The atomic computation SequentialQueen does not share common data area with other atomic computations and therefore it is in a separate process to preserve good data encapsulation and to save memory space. In general, only the atomic computations that are logically coherent and share the same common data area should be in the same process.

3. Dynamic Scheduling

We will use the process kernel concepts and terminology 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 which involve dynamic creation of medium-grained processes. For example, the *Chant* system [5] for lightweight threads, *Nexus* runtime support [6] for task-parallel programming languages, *Mentat* system [7] for object-oriented parallel processing, and Cantor for actor-based languages [8] can all benefit from such strategies.

Many previous research efforts have been directed towards the process allocation in distributed systems [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. A recent comparison study of dynamic load balancing strategies on highly parallel computers is given by Willebeek-LeMair and Reeves [20].

Eager *et al.* compared the sender-initiated algorithm and receiver-initiated algorithm [11]. Work with a similar assumption as ours includes the Gradient Model developed by Lin and Keller [21]. The randomized allocation algorithms developed by different authors are quite simple and effective [22, 23, 24, 25]. These strategies are discussed in this section.

Table I: The Solution Space of Dynamic Scheduling

	No information	Passive information	Active information
Non-neighborhood Strategy	Random Allocation	Sender-initiated Receiver-initiated	
Neighborhood Strategy			Gradient Model Symmetrical Hopping

The solutions of dynamic scheduling can be described in a two-dimension space. The first dimension is the information system and the second is scheduling distance, which are illustrated in Table I. First, scheduling strategies can be classified based on the underlying load information system. The *load* measure may include the number of messages waiting to be processed, the number of active processes, available memory, etc. There are three kinds of systems based on different information strategies: the system with no load information, the passive information system, and the active information system. In a passive information system, load information is only exchanged on demand. Whereas, in an active information system, load information is exchanged periodically or whenever states are changed, no matter if the information will be used or not. Since load information is not available in a passive information system, a processor that is to make a scheduling decision must request and wait for the information. The advantage of an active information system is that the partial or global information of the system is always provided so that a better decision can be made quickly. However, the overhead of information exchange may overwhelm its benefits. Insufficient information results in poor scheduling decision. On the other hand, too much information exchanged leads to large communication overhead. In addition, information may travel a long distance, resulting in information aging. A minimum, but sufficient information exchange gives the best overall performance.

Based on the amount of information exchanged, the information strategies can be classified as follows [2]:

TYPE-I strategies involve using no status information.

TYPE-II strategies calculate the status information by using local load information only.

TYPE-III strategies calculate the status information by collecting load information from neighbors.

TYPE-IV strategies calculate the status information by collecting *status* information from neighbors.

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 because no communication is involved. With no load information, the system is not able to adapt to variations in the system. Using local information alone is not sufficient to judge such variations either. Type-V strategies, on the other hand, may be too expensive in large systems. The algorithm developed in this paper 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 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.

The scheduling strategies can also be classified by scheduling distance. In a non-neighborhood strategy, a process can be scheduled to any processor in the system. Its advantage is that the load can spread fast. However, this strategy suffers from poor locality and large communication overhead. A neighborhood strategy only schedules a process to immediate neighbors. However, its neighbor can further schedule the process to the neighbor's neighbor to achieve a global load balancing. The advantages of this strategy are its locality and low communication overhead.

Next, we briefly describe four scheduling algorithms: the randomized allocation, the sender-initiated algorithm, the receiver-initiated algorithm, and the gradient model.

Athas and Seitz have proposed a global randomized allocation algorithm [22, 23]. A randomized allocation algorithm dictates that each PE, when it generates a new process, 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 processes per PE is concerned. However, a few factors may degrade the performance of the randomized allocation. First, the grain-size of processes may vary. Even if each PE processes about the same number of processes, the load on each PE may still be uneven. Second, the lack of locality leads to large overhead and communication traffic. Only $1/N$ of all processes stay on the creating PE, where N is the number of PEs in the system. Thus, most messages between processes 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. Since 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.* [10] have modified the naive randomized allocation algorithm. They use *threshold*, a kind of local load information, to determine whether to process a process locally or locate a process randomly.

In a sender-initiated algorithm, scheduling activity is initiated by an overloaded processor that attempts to send a process to an underloaded processor; whereas, in a receiver-initiated algorithm, scheduling activity is initiated by an underloaded processor that attempts to request a process from an overloaded processors [11]. Both sender-initiated algorithms and receiver-initiated algorithms are threshold algorithms. A threshold T is predefined to determine a processor being a sender or a receiver. When the load in a processor excesses T, the processor is identified as a sender. A processor identifies itself as a suitable receiver if its load is lower than T. In a sender-initiated algorithm, a sender randomly polls other processors until a receiver is found. In a receiver-initiated algorithm, a receiver randomly polls other processors until a sender is found. Each of them has its advantage. The sender-initiated strategy is best for a lightly loaded system and the receiver-initiated strategy is more efficient in a heavily loaded system [11]. A scheduling algorithm that combines both sender-initiated and sender-initiated algorithms is called a symmetrical-initiated algorithm [26].

The gradient model [21] is an information-driven strategy. As stated by Lin [27], instead of trying to allocate a newly generated process to other PEs, the process 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. An idle PE 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 process from its local queue to the neighbor with the least proximity.

4. Symmetrical Hopping

The sender-initiated and receiver-initiated algorithms request information from a randomly chosen PE in the system, resulting in heavy communication and network contention. The information exchange can be limited within a neighborhood to increase locality [20]. With this limitation, the sender-initiated and receiver-initiated strategies are called *sender-initiated hopping* and *receiver-initiated hopping*, respectively. Symmetrical Hopping is a symmetrical-initiated algorithm that combines the sender-initiated hopping and receiver-initiated hopping strategies.

Symmetrical Hopping uses the TYPE-III information strategy. Here, each PE calculates its own load function by using a simple measure: 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 $\mathcal{F}(k)$, and its neighbors' load functions are denoted by a set of values $\mathcal{F}_k(i)$, where $dist(k,i) = 1$. The value of $\mathcal{F}(k)$ is calculated periodically.

The load information can then be used to determine a system state. For each PE k, a function $\mathcal{B}(k)$ is defined as $\mathcal{M}in_{dist(k,i)=1}\,\{\mathcal{F}_k(i)\}$, which represents how heavily its neighbors are loaded. Two predefined parameters, *low_mark* and *high_mark*, are compared with $\mathcal{B}(k)$ to determine the

```
Sender-initiated-hopping(k) /* at PE k */
  For each process c do
    /* process c may be either newly created or newly arrived */
    if (light-load and c.hops = 0)
         or (moderate-load and F(k) > B(k) and c.hop < d)
      c.hops = c.hops + 1
      send process c to least-loaded neighbor
    else
      enter process c into the local queue
    endif
```

Figure 3: The sender-initiated hopping strategy of the Symmetrical Hopping algorithm.

current system state:

- light-load state: $\mathcal{B}(k) < low\_mark$
- moderate-load state: $low\_mark \leq \mathcal{B}(k) < high\_mark$
- heavy-load state: $high\_mark \leq \mathcal{B}(k)$

The *low_mark* and *high_mark* are set by experiment to 2 and 8, respectively.

The Symmetrical Hopping scheduling consists of sender-initiated hopping and receiver-initiated hopping strategies. The sender-initiated hopping strategy of the Symmetrical Hopping algorithm is similar to that in the Adaptive Contracting Within Neighborhood (ACWN) algorithm [28], which is shown in Figure 3. Here, a newly created process moves m hops, where $0 \leq m \leq d$ and d is the network diameter. We set an upper limit of traveling distance d for each process to prevent unbounded message oscillation. This decision is based on the system state of each PE. The number of hops traveled so far for each process c is recorded as $c.hops$. Thus, at each PE k, for a process 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 $c.hops = d$, process c will be retained locally and added to the local pool of messages; if the system is in the *light-load* state and $c.hops = 0$, PE k will send process c to its least-loaded neighbor no matter what its own load is. Processes can distributed quickly in a lightly loaded system with this policy. If the system is in the *moderate-load* state, the process will be sent conditionally: if the load on the least-loaded neighbor is smaller than its

```
Receiver-initiated-hopping(k) /* at PE k */
  if (F(K) < low_mark) and (not(wait))
    initiate a request to most-loaded neighbor
    set wait = true
  for each request
    if (heavy-load)
      send a process to the PE that initiated the request
    else
      if (r.hop < d)
        r.hop = r.hop + 1
        send the request to most-loaded neighbor
      else
        send a reject signal to the PE that initiated the request
      endif
    endif
  upon receiving a process or a reject signal, set wait = false
```

Figure 4: The receiver-initiated hopping strategy of the Symmetrical Hopping algorithm.

own load, the process is sent to that neighbor. In this way, the newly generated process c travels along the steepest load gradient to a local minimum.

Load imbalances may appear when only the sender-initiated hopping strategy is applied. Such imbalances may appear either due to limitations of the underlying load balancing scheme which finds only a local minimum, or due to the different rates of consumption of processes. Moreover, since 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 processes without sending them to any other PEs. The receiver-initiated hopping strategy is applied to solve this problem, which is shown in Figure 4. If a PE is in the *light-load* state, it initiates a request to its most-loaded neighbor. A PE, upon receiving a request, will send a process to the PE that initiated the request if it is in the *heavy-load* state. Otherwise, it sends the request to its most-loaded neighbor if $r.hops < d$, where $r.hops$ records the number of hops the request traveled so far. If the processor is not in the *heavy-load* state and $r.hops = d$, it sends a *reject* signal to the PE that initiated the request. The processor, upon receiving a reject signal, may initiate another request if it is still in the *light-load* state.

Symmetrical Hopping uses a moderate amount of load information for scheduling decision. A more accurate decision can be made compared to the randomized allocation, the sender-initiated and receiver-initiated algorithms. It uses less information than the gradient model so that the overhead of information exchange is reduced. Because the load information is only exchanged between neighbors, it is much more accurate than the proximity information, which travels a long distance to reach the processor that uses the information. Symmetrical Hopping also combines the advantages of the sender-initiated and receiver-initiated strategies and adapts to lightly loaded and heavily loaded systems. Actually, the sender-initiated hopping strategy itself delivers fairly good performance. But sometimes the load may gather at one point, forming a "hot spot." The receiver-initiated hopping strategy can "steal" processes from hot spots to reach an evenly distributed load.

5. Performance Evaluation

We have tested a number of sample programs on 512-node Intel Touchstone Delta to study the effectiveness of dynamic scheduling algorithms on multicomputers. Five algorithms, the randomized allocation, the sender-initiated algorithm, the receiver-initiated algorithm, the gradient model, and Symmetrical Hopping, were implemented. They shared most subroutines except the scheduling module. For each program the best sequential program written in C was also tested without changing the algorithm.

The execution time can be broken into three parts: computation time, overhead, and idle time. *Computation time* is spent on problem solving and the sum of computation time of all PEs should be equal to the sequential execution time. This time is invariant with different scheduling strategies and different numbers of PEs. *Overhead* is the time spent on works other than computation which includes the work of bookkeeping, communication, and load balancing. *Idle time* is the time in which PEs have no work to do.

The overhead can be further divided into three parts: the bookkeeping overhead, the communication overhead, and the scheduling overhead. *Bookkeeping overhead* depends on the number of processes and the number of messages. For each individual process, the system maintains a

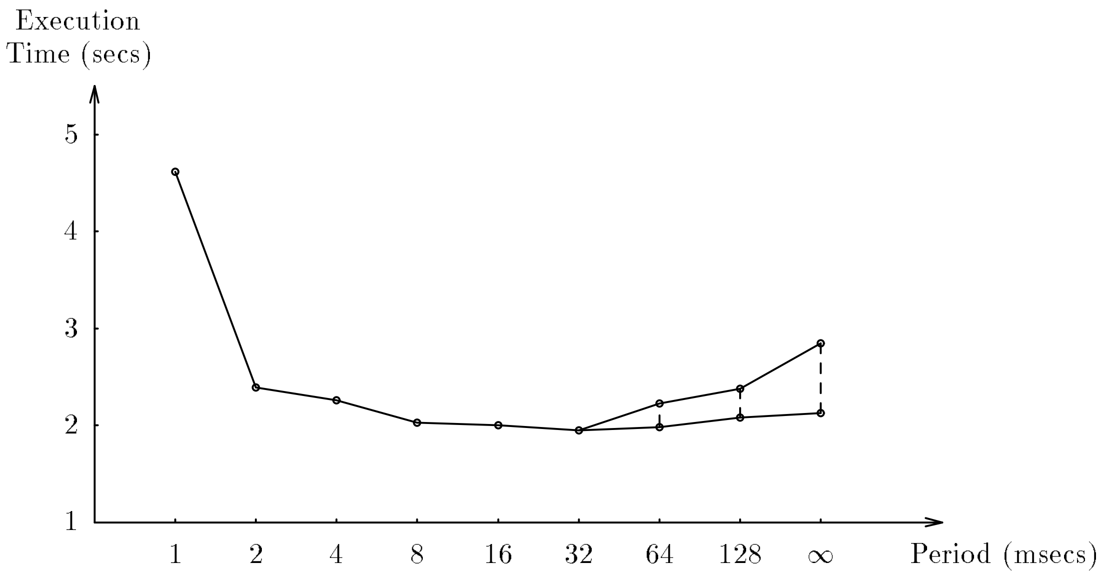

Figure 5: Comparison of different load information exechange periods.

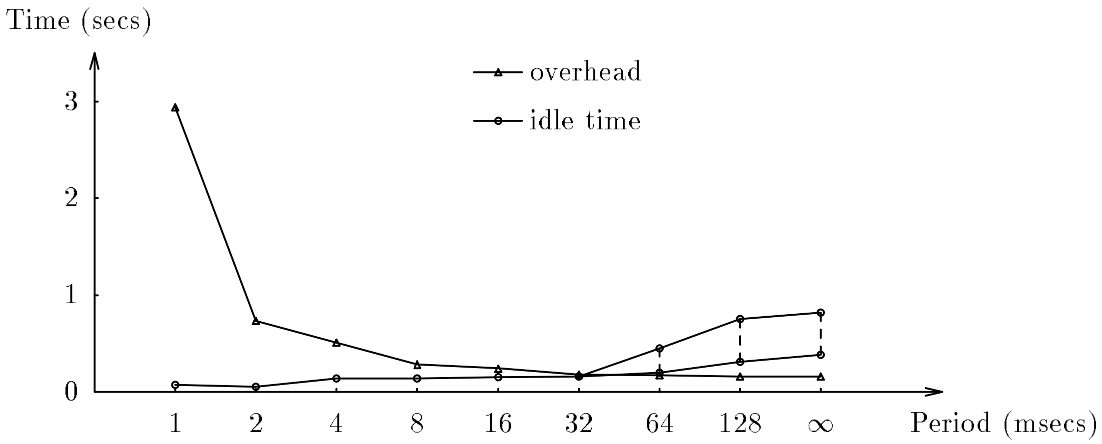

Figure 6: Overhead and idle time.

process block, and for each message there is a message header including its source and destination process information. The overhead of bookkeeping is about 90 microseconds whenever a new process is created or a message is sent. The *communication overhead* consists of the time spent by the processor that deals with the sending and receiving of messages. The actual transmission time is overlapped with computation and does not need to be considered. The communication overhead depends on many factors: the size of messages, the number of hops a message travels, and network contention. The average overhead for a communication is about 2 milliseconds. Not all the messages between processes introduce communication overhead. Only those going to PEs other than the source PE have that 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 processes. *Scheduling overhead* includes two parts: updating load information and process placement. Time spent on process placement is proportional to the number of processes and is determined by granularity. System load information can be exchanged periodically. A typical curve of execution time for different information exchange periods is shown in Figure 5. This curve is for the exhaustive search (15-Queen) on 256 PEs. With a long exchanging period, the system acts unstably. We give both worst and best time from many repetitions of experiments for periods of more than 64 milliseconds. Figure 6 shows the average overhead and idle time. Too short a period makes the frequently updated load information unnecessary and increases overhead. Too long a period leads to inaccurate load information due to sluggish updates, leading to an unbalanced load. From the curves, it can be seen that the best period is between 16 and 64 milliseconds. In the rest of the experiments, the period of load information exchanging is set to be 32 milliseconds for Symmetrical Hopping and the best value of exchanging interval is also selected for the gradient model.

We now discuss the influence of scheduling strategies. A good scheduling algorithm must be able to balance load for different application problems. At the same time, it has to keep scheduling overhead low. Furthermore, it must keep good locality so that as many processes as possible can be executed locally to reduce communication overhead. The only scheduling overhead for the randomized allocation is to generate random numbers whenever a process is created. Therefore, its scheduling overhead is negligible. However, communication overhead is high since the randomized allocation has a few local processes (about $1/N$) and most processes from other

PEs, irrespective of whether the system is heavily or lightly loaded. The sender-initiated algorithm involves large communication overhead on polling idle processors. The load is not well balanced since a limited number of processors are polled. Similarly, the receiver-initiated algorithm spends time on polling overloaded processors. Sometimes, it may be idle for a while before a process can be received. 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 processes 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 Symmetrical Hopping exhibits better locality than all other scheduling algorithms except the gradient model. Therefore, it has less communication overhead. A well-balanced load is obtained by combining the sender-initiated hopping and receiver-initiated hopping strategies. Its scheduling overhead is also small, due to a moderate amount of load information exchange.

The system has been tested with three application problems. The first one, the exhaustive search of the N-queen problem, is of irregular and dynamic structure. In this problem, the grain-size is not even, since whenever a new queen is placed, the search either successfully continues to the next row or fails. The number of processes generated is unpredictable. The second one, iterative deepening A* (IDA*) search, is a good example of parallel search techniques [29]. The sample problem is the 15-puzzle. The grain-size may vary substantially, since it dynamically depends on the current estimated cost. Also, synchronization at each iteration reduces the effective parallelism. Performance of this problem is therefore not as good as others. The third one, a molecular dynamics program GROMOS, is a real application problem [30, 31]. The test data for GROMOS is the bovine superoxide dismutase molecule (*SOD*), which has 6968 atoms [32]. The cutoff radius is predefined to 16 $\mathring{A}$. GROMOS has a more predictable structure. The number of processes is known with the given input data, but the computation density in each process varies. Thus a load balancing mechanism is necessary.

Table II shows the exhaustive search (16-queen) performance of the Symmetrical Hopping algorithm. This problem has 22150 processes and 44300 messages (atomic computations). The average grain-size of each atomic computation is 63.8 milliseconds. A speedup of 441 is obtained on 512 nodes.

Table II: Symmetrical Hopping on Intel Touchstone Delta

	number of PEs						
	seq.	1	4	16	64	256	512
Execution time (*secs*)	2826	2830	712	179	46.3	12.1	6.41
Speedup		0.999	3.97	15.8	61.0	234	441
Efficiency		99.9%	99.3%	98.7%	95.4%	91.2%	86.1%

Table III: Performance Comparison on Intel Touchstone Delta

Execution Time (*secs*)

		number of PEs						
		seq.	1	4	16	64	256	512
Exhaustive search	Random	411	413	109	28.3	7.70	2.38	1.43
	Sender-initiated	411	413	161	43.2	11.3	3.03	1.58
	Receiver-initiated	411	413	116	41.9	13.6	5.78	3.43
	Gradient	411	414	111	53.0	54.1	46.2	47.6
	Symmetrical Hopping	411	414	107	27.8	6.97	1.95	1.08
IDA* search	Random	905	906	265	72.1	21.7	8.51	6.12
	Sender-initiated	905	906	336	101	30.2	10.1	7.19
	Receiver-initiated	905	906	291	92.9	30.3	14.7	10.5
	Gradient	905	906	279	112	109	107	107
	Symmetrical Hopping	905	906	247	64.6	18.7	7.10	4.95
GROMOS	Random	533	534	153	40.0	11.9	4.32	2.33
	Sender-initiated	533	535	312	82.2	21.0	5.22	2.89
	Receiver-initiated	533	535	153	45.4	15.8	5.60	3.82
	Gradient	533	535	144	41.5	21.4	19.7	19.9
	Symmetrical Hopping	533	535	141	37.1	9.70	2.86	1.72

In Table III and Figures 7–9, we compare the performance of the randomized allocation, the sender-initiated algorithm, the receiver-initiated algorithm, the gradient model, and the Symmetrical Hopping algorithm. The test problems are exhaustive search (15-queen), IDA* search (15-puzzle), and GROMOS with cutoff radius of 16 Å. Characteristic features for these problems are shown in Table IV. The granularity is between 4 and 30 milliseconds, resulting from the medium-grained partitioning. Coarse granularity causes serious load imbalance and fine granu-

Table IV: Characteristics of Test Problems

Problems	number of processes	execution time per process ($msecs$)	number of messages	execution time per msg ($msecs$)
Exhaustive search	15941	25.8	31882	12.9
IDA* search	95052	9.5	190104	4.8
GROMOS	11220	47.5	22440	23.8

larity leads to large overhead. The randomized allocation performs fairly well, especially for these medium-grained processes. A similar conclusion is also made by Shu and Grunwald [2, 33]. The sender-initiated algorithm is not good for a small system, because the system is heavily loaded for a fixed size of problem. The receiver-initiated algorithm does not perform well in a large system which is lightly loaded. Both sender-initiated algorithm and receiver-initiated algorithm suffer from their frequent polling. The gradient model performs well in a small system, but becomes worse when the number of PE increases. This is because the gradient model is not able to spread load fast, in addition to its heavy information exchange. As we can see, Symmetrical Hopping is better than other scheduling algorithms in all the cases due to its adaptive behavior.

Table V shows the speedup of these dynamic scheduling algorithms over an initial static allocation. In this allocation, initial processes are allocated before the program starts execution. Each processor gets about the same number of processes. When a process generates other processes, the newly generated processes will stay in the same processor without being scheduled. When the processors start to execute the program, one processor may finish the execution before others because of uneven process granularity and different number of newly generated processes. The data in Table V shows that most dynamic scheduling algorithms outperform the static one. However, sometimes a dynamic scheduling algorithm may perform poorer than the static algorithm. That is because some dynamic scheduling algorithms do not distribute the load evenly. As an example, the gradient model spreads load very slowly in a large system, and the load is even less equally distributed as a static scheduling algorithm does.

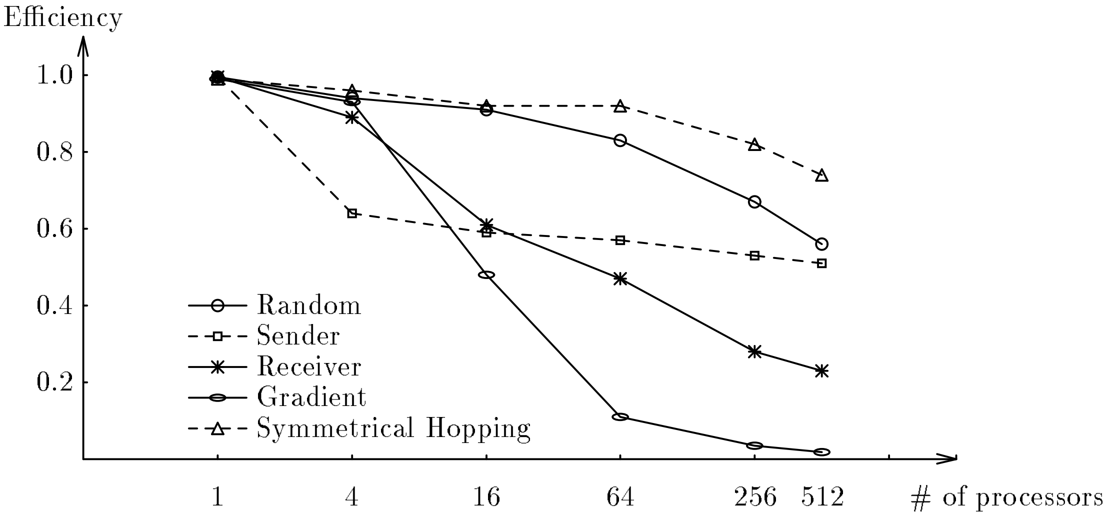

Figure 7: Exhaustive search efficiency.

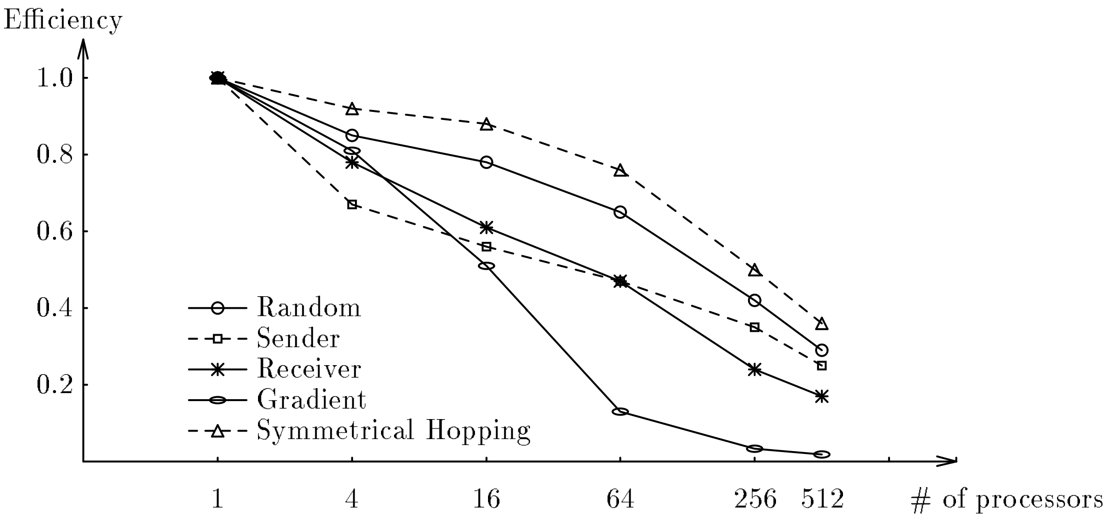

Figure 8: IDA* search efficiency.

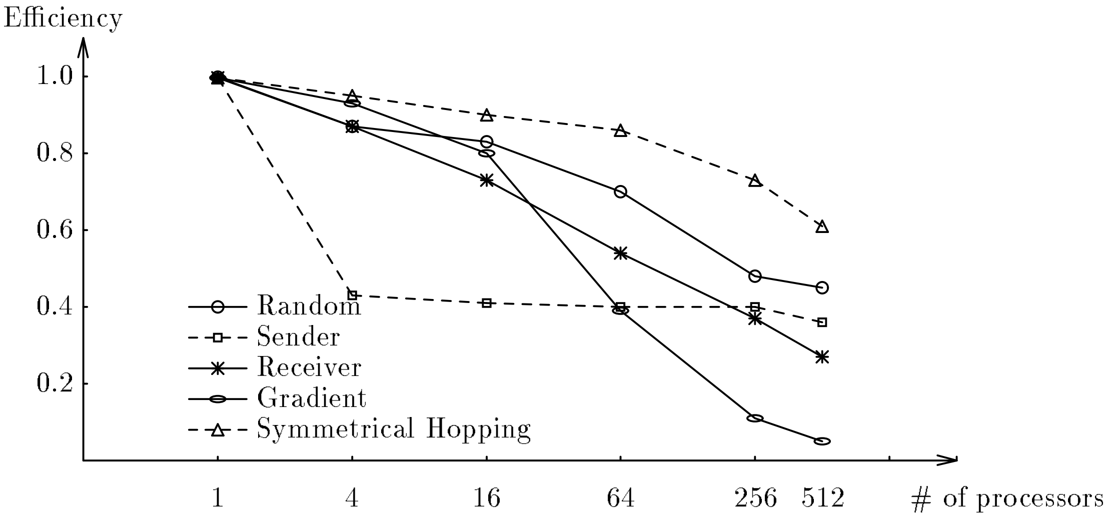

Figure 9: GROMOS efficiency.

Table V: Speedup of Dynamic Scheduling Over Static Scheduling on 256-node Touchstone Delta

	Random	Sender	Receiver	Gradient	Symmetrical
Exhaustive search	1.56	1.22	0.64	0.08	1.90
IDA* search	1.89	1.61	1.10	0.11	2.34
GROMOS	1.45	1.18	1.10	0.31	2.15

6. Discussion

The basic assumption for dynamic scheduling of a single application is that the application satisfies the following two conditions: (1) the application can be partitioned into many processes so that the number of processes p is much larger than the number of processors N; and (2) the average grain-size of processes g is much larger than the average overhead T_o associated with a process. The second condition is not always satisfied because sometimes the overhead increases with the grain-size for the applications with heavy dependency. In many other cases, the overhead is nearly a constant. In the process kernel, the average overhead T_o is about 1 millisecond. Assume $p > 30N$ and $g > 30T_o$, then the problem size $pg > 900NT_o$ milliseconds, that is, 900 milliseconds per processor. The applications running on a parallel computer are usually larger than this and a satisfactory performance can be expected with a reasonable partitioning algorithm and a good scheduling algorithm.

The Symmetrical Hopping can adapt to different process sizes. Assume that at the same time both PE i and PE j have m messages waiting for processing. 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, the idle PE j is able to request processes from PE i. In contrast, the randomized allocation cannot adapt to such a case. The Symmetrical Hopping algorithm outperforms other scheduling algorithms, partly due to its complementary sender-initiated and receiver-initiated hopping strategies, and partly due to its adaptive locality. Its good locality reduces communication overhead, whereas the randomized allocation, the sender-initiated algorithm, and the receiver-initiated algorithms do not. The Symmetrical Hopping allows load to spread out faster than the gradient model. For a small number of PEs, the gradient model may make better load balance than the randomized allocation. However, since the gradient model was designed based on good locality to reduce communication overhead, it does not spread the load fast. For a large number of PEs, the gradient model leads to more load imbalance than other scheduling strategies do. As shown in Figure 10 for the exhaustive search (15-Queen), the idle time of the gradient model at a large number of PEs is much longer than the randomized allocation. Symmetrical Hopping reaches the most even load distribution among the five scheduling algorithms.

Does overhead of a complicated scheduling algorithm always overwhelm the benefits 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 Symmetrical Hopping pays more scheduling overhead compared to the randomized allocation, it still can achieve better performance in most cases.

Overhead can be reduced by using co-processors. A co-processor can be attached to the main processor in each PE, which handles all bookkeeping, load balancing, and communication activities. In the Touchstone Delta, each PE has a communication co-processor which shares part

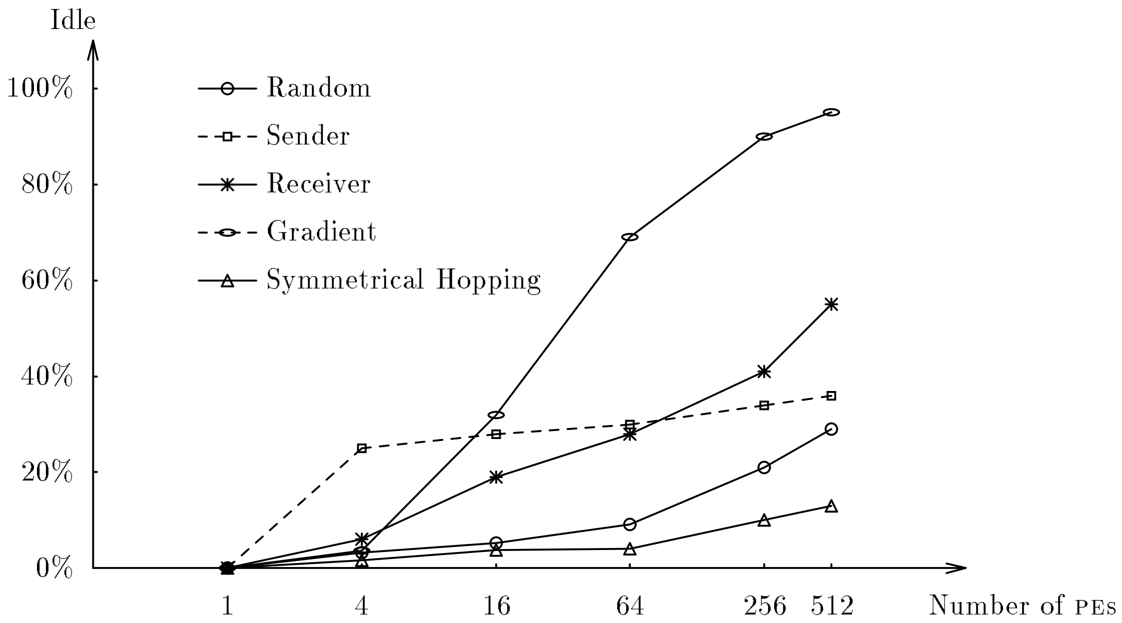

Figure 10: Comparison of PE idle time for different scheduling algorithms (exhaustive search).

of the communication overhead. Since we are not able to program co-processors, overhead of bookkeeping, load balancing, and part of communication must be handled by the main processor. If the Symmetrical Hopping scheduling can be applied to a system with co-processors, 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 co-processor does not exceed the load of the main processor. The randomized allocation and the gradient model may benefit more from the co-processor than Symmetrical Hopping does, since the randomized allocation has more communication overhead and the gradient model has more scheduling overhead.

7. Conclusion

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

Symmetrical Hopping was compared with four other algorithms: the randomized allocation, the sender-initiated algorithm, the receiver-initiated algorithm, and the gradient model. All these algorithms were implemented in a system called the process kernel running on the Intel Touchstone Delta. The experimental results demonstrate that Symmetrical Hopping performs better than the other algorithms for many computation patterns.

References

[1] W. Shu and L. V. Kale, "Chare Kernel — a runtime support system for parallel computations," *Journal of Parallel and Distributed Computing*, vol. 11, pp. 198–211, Mar. 1991.

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

[3] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing.* Benjamin/Cummings Publishing Company, Inc., 1994.

[4] G. A. Agha, *A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[5] M. Haines, D. Cronk, and P. Mehrotra, "On the design of Chant: A talking threads package," in *Supercomputing 94'*, Nov. 1994.

[6] I. Foster, C. Kesselman, R. Olson, and S. Tuecke, "Nexus: An interoperability layer for parallel and distributed computer systems," Tech. Rep. ANL/MCS-TM-189, Argonne National Lab, 1994.

[7] A. S. Grimshaw, "Easy-to-use object-oriented parallel processing with Mentat," *IEEE Computer*, vol. 26, pp. 39–51, May 1993.

[8] W. C. Athas and C. L. Seitz, "Cantor user report," tech. rep., Dept. of Computer Science, California Institute of Technology, Jan. 1987.

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

[10] 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, May 1986.

[11] 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, Mar. 1986.

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

[13] T. L. Casavant and J. G. Kuhl, "A formal model of distributed decision-making and its application to distributed load balancing," in *Int'l Conf. on Distributed Computing System*, pp. 232–239, May 1986.

[14] T. L. Casavant and J. G. Kuhl, "Analysis of three dynamic distributed load-balancing strategies with varying global information requirements," in *Int'l Conf. on Distributed Computing System*, pp. 185–192, May 1987.

[15] A. Hac and X. Jin, "Dynamic load balancing in a distributed system using a decentralized algorithm," in *Int'l Conf. on Distributed Computing System*, pp. 170–177, May 1987.

[16] V. Singh and M. R. Genesereth, "A variable supply model for distributing deductions," in *9th Intel. Joint Conf. Artificial Intelligence*, pp. 39–45, Aug. 1985.

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

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

[19] Z. Lin, "A distributed fair polling scheme applied to parallel logic programming," *International Journal of Parallel Programming*, vol. 20, Aug. 1991.

[20] M. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Trans. Parallel and Distributed System*, vol. 9, pp. 979–993, Sept. 1993.

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

[22] W. C. Athas, *Fine Grain Concurrent Computations.* PhD thesis, Dept. of Computer Science, California Institute of Technology, May 1987.

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

[24] R. M. Karp and Y. Zhang, "A randomized parallel branch-and-bound procedure," *Journal of ACM*, vol. 40, pp. 765–789, 1993.

[25] S. Chakrabarti, A. Ranade, and K. Yelick, "Randomized load balancing for tree structured computation," in *IEEE Scalable High Performance Computing Conference*, pp. 666–673, 1994.

[26] P. Krueger and M. Livny, "The diverse objectives of distributed scheduling policies," in *Proc. of the 7th International Conf. on Distributed Computing Systems*, pp. 242–249, 1987.

[27] F. C. H. Lin, *Load Balancing and Fault Tolerance in Applicative Systems.* PhD thesis, Dept. of Computer Science, University of Utah, Aug. 1985.

[28] W. Shu, "Adaptive dynamic process scheduling on distributed memory parallel computers," *Scientific Programming*, vol. 3, pp. 341–352, 1994.

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

[30] W. F. van Gunsteren and H. J. C. Berendsen, "GROMOS: GROningen MOlecular Simulation software," tech. rep., Laboratory of Physical Chemistry, University of Groningen, Nijenborgh, The Netherlands, 1988.

[31] R. v. Hanxleden and K. Kennedy, "Relaxing SIMD control flow constraints using loop transformations," Tech. Rep. CRPC-TR92207, Center for Research on Parallel Computation, Rice University, Apr. 1992.

[32] J. Shen and J. A. McCammon, "Molecular dynamics simulation of superoxide interacting with superoxide dismutase," *Chemical Physics*, vol. 158, pp. 191–198, 1991.

[33] D. C. Grunwald, *Circuit Switched Multicomputers and Heuristic Load Placement.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, UIUCDCS-R-89-1514, Sept. 1989.