Abstract—One of the essential problems in parallel computing is: Can SIMD machines handle asynchronous problems? This is a difficult, unsolved problem because of the mismatch between asynchronous problems and SIMD architectures. We propose a solution to let SIMD machines handle general asynchronous problems. Our approach is to implement a runtime support system which can run MIMD-like software on SIMD hardware. The runtime support system, named P kernel, is thread-based. There are two major advantages of the thread-based model. First, for application problems with irregular and/or unpredictable features, automatic scheduling can move some threads from overloaded processors to underloaded processors. Second, and more importantly, the granularity of threads can be controlled to reduce system overhead. The P kernel is also able to handle bookkeeping and message management, as well as to make these low-level tasks transparent to users. Substantial performance has been obtained on Maspar MP-1.

Index Terms—SIMD parallel computers, portable programming environment, load balancing, thread model, scalability, irregular and dynamic applications.

I. INTRODUCTION

A. Can SIMD Machines Handle Asynchronous Problems?

The current parallel supercomputers have been developed along two major architectures: the SIMD (Single Instruction Multiple Data) architecture and the MIMD (Multiple Instruction Multiple Data) architecture. The SIMD architecture consists of a central control unit and many processing units. Only one instruction can be executed at a time and every processor executes the same instruction. Advantages of a SIMD machine include its simple architecture, which makes the machine potentially inexpensive, and its synchronous control structure, which makes programming easy [4], [34] and communication overhead low [26]. The designers of SIMD architectures have been motivated by the fact that an important, though limited, class of problems fit the SIMD architecture extremely well [35]. The MIMD architecture is based on the duplication of control units for each individual processor. Different processors can execute different instructions at the same time [15]. It is more flexible for different problem structures and can be applied to general applications. However, the complex control structure of MIMD architecture makes the machine expensive and the system overhead large.

Application problems can be classified into three categories: synchronous, loosely synchronous, and asynchronous. Table I shows a few application problems in each of the three categories [13].

- The synchronous problems have a uniform problem structure. In each time step, every processor executes the same operation over different data, resulting in a naturally balanced load.
- The loosely synchronous problems can be structured iteratively with two phases: the computation phase and the synchronization phase. In the synchronization phase, processors exchange information and synchronize with each other. The computation load can also be redistributed in this phase. In the computation phase, different processors can operate independently.
- The asynchronous problems have no synchronous structure. Processors may communicate with each other at any time. The computation structure can be very irregular and the load imbalanced.

The synchronous problems can be naturally implemented on a SIMD machine and the loosely synchronous problems on an MIMD machine. Implementation of the loosely synchronous problems on SIMD machines is not easy; computation load must be balanced and the load balance activity is essentially irregular. As an example, the simple $O(n^2)$ algorithm for N-body simulation is synchronous and easy to implement on a SIMD machine [12]. But the Bernut-Hut algorithm ($O(n \log n)$) for N-body simulation is loosely synchronous and difficult to implement on a SIMD machine [3].

Solving the asynchronous problems is more difficult. First, a direct implementation on MIMD machines is nontrivial. The user must handle the synchronization and load balance issues at the same time, which could be extremely difficult for some application problems. In general, a runtime support system, such as LINDA [2], [5], reactive kernel [27], [33], or chare kernel [30], is necessary for solving asynchronous problems. Implementation of the asynchronous problems on SIMD ma-

TABLE I

<table>
<thead>
<tr>
<th>Classification of Problem Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>Matrix algebra</td>
</tr>
<tr>
<td>Finite difference</td>
</tr>
<tr>
<td>QCD</td>
</tr>
</tbody>
</table>

The authors are with the Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260, e-mail: shu@cs.buffalo.edu; wu@cs.buffalo.edu. IEEECS Log Number D95025.
chines is even more difficult because it needs a runtime sup-
port system, and the support system itself is asynchronous. In
particular, the support system must arrange the code in such a
way that all processors execute the same instruction at the
same time. Taking the N-queen problem as an example, since
it is not known prior to execution time how many processes
will be generated and how large each computation is, a run-
time system is necessary to establish balanced computation for
efficient execution. We summarize the above discussion in
Table II.

<table>
<thead>
<tr>
<th></th>
<th>Synchronous</th>
<th>Loosely synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIMD</td>
<td>easy</td>
<td>natural</td>
<td>need runtime support</td>
</tr>
<tr>
<td>SIMD</td>
<td>natural</td>
<td>difficult</td>
<td>difficult, need runtime support</td>
</tr>
</tbody>
</table>

Various application problems require different program-
ing methodologies. Two essential programming methodolo-
gies are array-based and thread-based. The problem domain of
most synchronous applications can be naturally mapped onto an
array, resulting in the array-based programming methodol-
egy. Some other problems do not lend themselves to efficient
programming in an array-based methodology because of mis-
mash between the model and the problem structure. The solu-
tion to asynchronous problems cannot be easily organized into
aggregate operations on a data domain that is uniformly struc-
tured. It naturally demands the thread-based programming
methodology, in which threads are individually executed and
where information exchange can happen at any time. For the
loosely synchronous problems, either the array-based or the
thread-based programming methodology can be applied.

B. Let SIMD Machines Handle Asynchronous Problems

To make a SIMD machine serve as a general purpose ma-
chine, we must be able to solve asynchronous problems in
addition to solving synchronous and loosely synchronous
problems. The major difficulties in executing asynchronous
applications on SIMD machines are:

- the gap between the synchronous machines and asyn-
chronous applications; and
- the gap between the array processors and thread-based
programming.

One solution, called the application-oriented approach, lets the
user fill the gap between application problems and architec-
tures. With this approach, the user must study each problem
and look for a specific method to solve it [7], [36], [37], [39].
An alternative to the application-oriented approach is the sys-
tem-oriented approach, which provides a system support to run
MIMD-like software on SIMD hardware. The system-oriented
approach is superior to the application-oriented approach for
three reasons:

- it is more efficient to develop a sophisticated solution in
system, instead of writing similar code repeatedly in the
user programs; and
- it is a general approach, and enhances the portability and
readability of application programs.

The system-oriented approach can be carried out in two
levels: instruction-level and thread-level. Both of them share
the same underlying idea: If one were to treat a program as
data, and a SIMD machine could interpret the data, just like a
machine-language instruction interpreted by the machine’s
instruction cycle, then an MIMD-like program could effi-
ciently execute on the SIMD machine [37]. The instruction-
level approach implements this idea directly. That is, the in-
structions are interpreted in parallel across all of the pro-
cessors by control signals emanating from the central control unit
[41]. The major constraint of this approach is that the central
control unit has to cycle through almost the entire instruction
set for each instruction execution because each processor may
execute different instructions. Furthermore, this approach must
insert proper synchronization to ensure correct execution se-
quence for programs with communication. The synchroniza-
tion could suspend a large number of processors. Finally, this
approach is unable to balance load between processors and
unlikely to produce good performance for general applications.

We propose a thread-based model for a runtime system
which can support loosely synchronous and asynchronous
problems on SIMD machines. The thread-level implementa-
tion offers great flexibility. Compared to the instruction-level
approach, it has at least two advantages:

- The execution order of threads can be exchanged to
avoid processor suspension. The load can be balanced for
application problems with irregular and dynamic
features.
- System overhead will not be overwhelming, since granu-
larit can be controlled with the thread-based approach.

The runtime support system is named as Process kernel or
P kernel. The P kernel is thread-based as we assign computa-
tion at the thread-level. The P kernel is able to handle the
bookkeeping, scheduling, and message management, as well as
to make these low-level tasks transparent to users.

C. Related Research

Existing work on solving loosely synchronous and asyn-
chronous problems on SIMD machines was mostly application
oriented [7], [36], [37], [39]. The region growing algorithm is
an asynchronous, irregular problem and difficult to run on
SIMD machines [39]. The merge phase, a major part of the
algorithm, performs two to three orders of magnitude worse
than its counterpart in MIMD machines. That result is due to
the communication cost and lack of a load balancing mecha-
nism. The authors concluded that “the behavior of the region
growing algorithm, like other asynchronous problems, is very
difficult to characterize and further work is needed in develop-
ing a better model.” The two other implementations, the Man-
delbrot Set algorithm [36] and the Molecular Dynamics algo-


Authorized licensed use limited to: UNIVERSITY OF NEW MEXICO. Downloaded on October 24, 2008 at 15:45 from IEEE Xplore. Restrictions apply.
inner loop for which the number of iterations varies between different iterations of the outer loop. This structure occurs in several different problems. An iteration advance method was employed to rearrange the iterations for SIMD execution, which was called SIMLAD (SIMD model with local indirect addressing) in [36], and loop flattening in [37]. Recent studies have been conducted on the simulation of logic circuits on a SIMD machine [6], [20]. The major difference between these works and our approach is that we handle general asynchronous and loosely synchronous problems instead of studying individual problems.

The instruction-level approach has been studied by a number of researchers [8], [9], [22], [25], [41], [40]. As mentioned above, the major restriction of this approach is that the entire instruction set must be cycled through to execute one instruction step for every processor. A common method to reduce the average number of instructions emulated in each execution cycle is to perform global ors to determine whether the instruction is needed by any processor [9], [41]. It might be necessary to insert barrier synchronizations at some points to limit the degree of divergence for certain applications. Having a barrier at the end of each WHERE statement (as well as each FORALL statement) is a good idea [8]. Other work includes an adaptive algorithm which changes the order of instructions emulated to maximize the expected number of active processors [25]. Besides this issue, load balancing and processor suspension are also unsolved problems. The applications implemented in these systems are non-communicating [41] or have a barrier at the end of the program [9]. Collins has discussed the communication issue and proposed a scheme to delay execution of communication instructions [8]. A similar technique is used in [9] for expensive operations (including communication). However, it is not clear whether the method will work. Many applications have dependences and their loads are imbalanced, inevitably leading to high communication overhead and processor suspension. In summary, this instruction-level approach has large interpreter overhead. A technique that literally transforms pure MIMD code into pure SIMD code to eliminate this overhead has been proposed in [10].

Perhaps the works that are closest to our approach are Graphinators [17], early work on combinators [21], [16], and the work on the interpretation of Prolog and FLAT GHC [19], [23], [24]. The Graphinators implementation is an MIMD simulator on SIMD machines. It was achieved by having each SIMD processor repeatedly cycle through the entire set of possible “instructions.” Our work is distinguished from their work in terms of granularity control. The Graphinator model suffered because of its fine granularity. The authors mentioned that “each communication requires roughly a millisecond, unacceptably long for our fine-grained application” [17]. In the P kernel, the user can control the gransize of the process and the gainsize can be well beyond one millisecond. Thus, the P kernel implementation can obtain acceptable performance. Besides, our model is general, instead of dedicating it merely to the functional language, as in the Graphinator model.

Our model is similar to the Large-Grain Data Flow (LGDF) model [18]. It is a model of computation that combines sequential programming with dataflow-like program activation. The LGDF model was implemented on shared memory machines. It can be implemented on MIMD distributed memory machines as well [42], [30], [11], [14]. Now we show that the model can be implemented on a SIMD distributed memory machine, too.

II. THE P KERNEL APPROACH—COMPUTATION MODEL AND LANGUAGE

The computation model for the P kernel, which originated from the Chare Kernel [30], 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. 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 [1]. 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. Fig. 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.

![Fig. 1. Process, atomic computation, and common data area.](image-url)

The P kernel is a runtime support system on a SIMD machine built to manipulate and schedule processes, as well as messages. A program written in the P 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 a 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.

The overall structure of the P kernel language differs from that of the traditional programming languages mainly in explicit declarations of

1) basic units of allocation — processes,
2) basic units of indivisible computation — atomic computations, and
3) communication media — messages.

With these fundamental structures available, computation can be carried out in parallel with the assistance of primitive functions provided by the P kernel, such as `OsCreateProc()`, `OsSendMsg()`, etc. The user can write a program in the P kernel language, deal with the creation of processes, and send messages between them. For details of the computation model and language, refer to [29].

In the following, we will illustrate how to write a program in the P kernel language using 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 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 Fig. 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` 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 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 a common data area with other atomic computations, and therefore, 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.

```
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 (ij) for (k = 1, N) { if (position (i+1,k) is marked valid) { 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() }
}
```

Fig. 2. The N-queen program.

III. DESIGN AND IMPLEMENTATION

The main loop of the P kernel system is shown in Fig. 3. It starts with a system phase, which includes placing processes, transferring data messages, and selecting atomic computations to execute. It is followed by a user program phase to execute the selected atomic computation. The iteration will continue until all the computations are completed. The P kernel software consists of three major components: computation selection, communication, and memory management.
A. Computation Selection

A fundamental difference between the MIMD and SIMD systems is the degree of synchronization required. In an MIMD system, different processors can execute different threads of code, but not in a SIMD system. When the P kernel system is implemented on an MIMD machine, which atomic computation will be executed next is the individual decision of each processor. Whereas, due to the lock-step synchronization in a SIMD machine, the same issue becomes a global decision.

Let's assume that there are K atomic computation types, corresponding to the atomic computation definitions, represented by \( a_0, a_1, ..., a_{K-1} \). During the lifetime of execution, the total number of atomic computations executed is far more than \( K \). These atomic computations are dynamically distributed among processors. At iteration \( i \), a computation selection function \( F \) is applied to select an atomic computation type \( a_k \), where \( k = F(i) \) and \( 0 \leq k < K \). In the following user program phase at the same iteration, a processor will be active if it has at least one atomic computation of the selected type \( a_k \). Let the function \( num(i, p, t) \) record the number of atomic computation with type \( a_k \) at processor \( p \) in iteration \( t \). Let the function \( act(i, t) \) count the number of active processors at iteration \( t \) if the atomic computation type \( a_k \) is selected,

\[
act(i, t) = \{ p | num(i, p, t) > 0, \ 0 \leq p < N \} \mid N \text{ is the number of processors.}
\]

We present three computation selection algorithms here. The first one, \( F_{cyc} \), is a simple algorithm.

A.1. Algorithm I: Cyclic Algorithm

Basically, it repeatedly cycles through all atomic computation types. However, if \( act(i, t) \) is equal to zero, the type \( i \) will be skipped.

\[
F_{cyc}(t) = \min \{ i | act(i \mod K, t) > 0 \},
\]

\[
F_{cyc}(t) = \min \{ i | act(i \mod K, t) > 0 \}
\]

where \( t \geq 1 \) and \( F_{cyc}(0) = -1 \). Here, it is not always necessary to carry out \( K \) reductions to compute \( act(i, t) \), since as long as the first nonzero \( act(i, t) \) is found, the value of function \( F_{cyc} \) is determined.

This algorithm is similar to the method used in the instruction-level approach in which processors repeatedly cycle through the entire set of possible instructions. The global reduction is essentially similar to the "global-or" method, which is used to reduce the number of instructions that are emananated for each execution iteration. However, in the instruction-level approach, each processor executes exactly one instruction per cycle, while in the thread-level approach, each processor may execute many threads per cycle. Also, the execution order of a program is fixed in the instruction-level approach, but the order can be exchanged in the thread-based approach.

To complete the computation in the shortest time, the number of iterations has to be minimized. Maximizing the processor utilization at each iteration is one of the possible heuristics. If \( act(k_1, t) \) is 100 and \( act(k_2, t) \) is 900, a computation selection function \( F(i) \) selecting \( k_2 \) is intuitively better, leading to an immediate good processor utilization. An auction algorithm, \( F_{au} \), is proposed based on this observation.

A.2. Algorithm II: Auction Algorithm

For each atomic computation \( i \), calculate \( act(i, t) \) at iteration \( t \). Then, the atomic computation with the maximum value of \( act(i, t) \) is chosen to execute next:

\[
F_{au}(t) = \max \{ j | act(j, t) = \max_{0 \leq k < K} act(i, t), 0 \leq j < K \}.
\]

The cyclic algorithm is nonadaptive in the sense that the selection is made almost independent of the distribution of atomic computations. In this way, it could be the case that a few processors are executing one atomic computation type while many processors are waiting for execution of the other atomic computation types. The auction algorithm is runtime adaptive. It will maximize utilization in most cases. An adaptive algorithm is more sophisticated in general. However, experimental results show that in most cases, the cyclic algorithm performs better than the auction algorithm. It has been observed that when an auction algorithm is applied, at the near end of execution, the parallelism becomes smaller and smaller, and the program takes a long time to finish. This low parallelism phenomenon degrades performance seriously, which is characterized as the tailing effect.

We propose an improved adaptive algorithm to overcome the tailing effect. To retain the advantage of the auction algorithm, we intend to maximize the processor utilization as long as there is a large pool of atomic computations available. On the other hand, when the available parallelism falls to a certain degree, we try to exploit large parallelism by assigning priorities to different atomic computation types. An atomic computation whose execution increases the parallelism gets a higher priority, and vice versa. The priority can either be assigned by the programmer or be automatically generated with dependency analysis.

A.3. Algorithm III: Priority Auction Algorithm

For simplicity, we assume that the atomic computations \( a_0, a_1, ..., a_{K-1} \) have been presorted according to their priorities, \( a_0 \),...,
with highest priority and \( a_{k,1} \) with the lowest. Use \( m = cn \) as a gauge of available parallelism, where \( c \) is a constant and \( N \) is the number of processors.

\[
\mathcal{F}_m(t) = \begin{cases} 
\min \{ \text{act}(j,t) \} & \text{if } \text{act}(j,t) > m \\
\max_{0 \leq j < K} \{ \text{act}(j,t) \} & \text{otherwise} 
\end{cases}
\]

When \( \max_{0 \leq j < K} \text{act}(j,t) \) is larger than \( m \), indicating that the degree of available parallelism is high, the auction algorithm will be applied. Otherwise, among the atomic computation types with \( \text{act}(i, t) > 0 \), the one with the highest priority will be executed next. The constant \( c \) is set to be 0.5. If more than half of processors are active, the auction algorithm is used to maximize the processor utilization. Otherwise, the priority is considered in favor of parallelism increase and tailing effect prevention.

### TABLE III
EXECUTION TIME OF THE 12-QUEEN PROBLEM

<table>
<thead>
<tr>
<th>Method</th>
<th>Cyclic</th>
<th>Auction</th>
<th>Priority Auction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10.4 Seconds</td>
<td>11.1 Seconds</td>
<td>10.1 Seconds</td>
</tr>
</tbody>
</table>

This algorithm can constantly provide better performance than that provided by the cyclic algorithm. Table III shows the performance for different computation selection algorithms with the 12-queen problem on the 1K-processor MP-1.

**B. Communication**

There are two kinds of messages to be transferred. One is the data message, which is specifically addressed to the existing process. The other kind is the process message, which represents the newly generated process. Where these process messages are to be transferred depends on the scheduling strategy used. The two kinds of messages are handled separately.

#### B.1. Transfer of Data Messages

Assume each processor initially holds \( d_{a}(p) \) data messages to be sent out at the end of the computation phase. Because of the SIMD characteristics, only one message can be transferred each time. Thus, the message transfer step must be repeated at least \( D_0 \) times, where

\[
D_0 = \max_{0 \leq p < N} d_{a}(p).
\]

The real situation is even more complicated. During each time of the message transfer, a collision may occur when two or more messages from different processors have the same destination processor. Therefore, we need to prevent the message loss due to the collision. Let \( \text{dest}(p) \) be the destination of a message from processor \( p \) and \( \text{src}(q) \) be the source from which processor \( q \) is going to receive a message. There is a collision if two processors \( p_1 \) and \( p_2 \) are sending messages to the same processor \( q \), so that \( \text{dest}(p_1) = \text{dest}(p_2) = q \). The processor \( q \) can receive only one of them, say from \( p_1 \), by assigning \( \text{src}(q) = p_1 \). Thus, only processor \( p_1 \) can successfully deliver its message to the destination. In general, we perform a parallel assignment operation for all processors.

If any collision happens, the send-with-overwrite semantics are applied. When this collision prevention scheme is applied, the processors \( p_i \) with \( p_i \neq \text{src}(\text{dest}(p_i)) \) must wait for the next time to compete again. After the first transfer of data messages, there may still be some unsent messages. Some processor \( p \) has more than one message to send (\( d_{d}(p) > 1 \)), and not every processor is able to send out its message during the first transfer due to collisions. Hence,

\[
d_k(p) = \begin{cases} 
\text{act}(q,p) - 1 & \text{if } p = \text{src}(\text{dest}(p)) \\
\text{act}(q,p) & \text{otherwise} 
\end{cases}
\]

\[
D_k = \max_{0 \leq p < N} d_k(p), k \geq 1.
\]

As long as \( D_k > 0 \), we need to continue on with \( D_{k+1}, D_{k+2}, \ldots, D_n \), where \( D_n = 0 \). The message transfer process can be illustrated in Fig. 4. Many optimizations could be applied to reduce the number of time steps to send messages. One of them is to let \( \text{src}(q) = p_1 \) when \( \text{act}(p_1) \geq \text{act}(p_2) \) and \( \text{dest}(p_1) = \text{dest}(p_2) = q \).

Notice that for later transfers, it is most likely that only a few processors are actively sending messages, resulting in a low utilization in the SIMD system. To avoid such a case, we do not require all messages to be transferred. Instead, we attempt to send out only a majority of data messages. The residual messages are buffered and will be sent again during the next iteration. Because of the atomic execution model, the processors that fail to send messages will not be stalled. Instead, a processor can continue execution as long as there are some messages in its queue.

We use \( \Theta_k \) to measure the percentage of data messages that is left after \( k \) times of data transfers,

\[
\Theta_k = \frac{\left\{ p \mid d_{d}(p) > 0, 0 \leq p < N \right\}}{\left\{ p \mid d_{d}(p) > 0, 0 \leq p < N \right\}}.
\]

Thus, a constant \( \theta \) can be used to control the number of transfers by limiting \( \Theta_k \geq \theta \). The algorithm for the data message transfer is summarized in Fig. 5. By experiment, the value of \( \theta \) has been determined to be around 0.2. Fig. 6 shows an example for the 12-queen problem on MP-1 in which the minimal execution time can be reached when \( \theta = 0.2 \).

#### B.2. Process Placement

The handling of process messages is almost the same as that of data messages, except that we need to assign a destination processor ID to each process message. The assignment is called process placement. The Random Placement algorithm has been implemented in the P kernel.
$k = 0, \Theta_k = 1, \theta = 0.2$
while $\Theta_k \geq \theta$
  for each processor $p$ with $d_k(p) > 0$
    assign $\text{dest}(p)$ and $\text{src}(p)$
    if $p = \text{src}(\text{dest}(p))$
      send data messages
      $d_{k+1}(p) = d_k(p) - 1$
    else
      $d_{k+1}(p) = d_k(p)$
    $k = k + 1$
  assign $\Theta_k$
end while
for any processor with $d_k(p) > 0$
buffer the residual data messages

Fig. 5. Data message transfer procedure.

![Execution time graph](image_url)

Fig. 6. The execution time for different values of $\theta$ for 12-queen.

B.3. Random Placement Algorithm

It is a simple, moderate performance algorithm:

\[
\text{dest}(p) = \text{random}(\text{mod } N)
\]

where $N$ is the number of processors. Once $\text{dest}(p)$ has been
assigned, we can follow the same procedure as in the transfer
of data messages. However, the two kinds of messages are
different from each other in that the $\text{dest}(p)$ of a data message
is fixed, whereas that of a process message can be varied. In
taking advantage of this, we are able to reschedule the process
message if the destination processor could not accept the
newly generated process because of some resource constraint
or collision. Here, rescheduling is simply a task that assigns
another random number as the destination processor ID. We
can repeat this rescheduling until all process messages are as-
signed the destination processor IDs. However, it is a better
choice that we only offer one or two chances for rescheduling,
instead of repeating it until satisfaction. Thus, the unsuccessful-
fully scheduled process messages are buffered similar to the
residual data messages, and then wait for the next communi-
cation phase.

A load balancing strategy called Runtime Incremental Parallel
Scheduling has been implemented for the MIMD version of
the P kernel [31]. In this scheduling strategy, the system
scheduling activity alternates with the underlying computation
work. Its implementation on a SIMD machine is expected to
deliver a better performance than random placement.

C. Memory Management

Most SIMD machines are massively parallel processors. In
such a system, any one of the thousands of processors could
easily run out of memory, resulting in a system failure. It is
certainly an undesired situation. Ideally, a system failure can
be avoided unless the memory space on all processors is ex-
hausted. Memory management provides features to improve
the system robustness. When the available memory space on a
specific processor becomes tight, we should restrict the new
resource consumption, or release memory space by moving out
some unprocessed processes to other processors.

In the P kernel system implemented on SIMD machines, we
use two marks, $m_1$ and $m_2$, to identify the state of memory
space usage. The function $\eta(p)$ is used to measure the current
usage of memory space at processor $p$,

\[
\eta(p) = \frac{\text{the allocated memory space at processor } p}{\text{the total memory space at processor } p}
\]

A processor $p$ is said to be in its normal state when $0 \leq \eta(p) < m_1$, in its nearly-full state when $m_1 \leq \eta(p) < m_2$, in its full
state when $m_2 \leq \eta(p) < 1$, and in its emergency state when
running out of memory, i.e. $\eta(p) = 1$.

C.1. The Nearly-Full State

We need to limit the new resource consumption since the
available memory space is getting tight. It is accomplished by
preventing newly generated process messages from being
scheduled. Thus, when a process message is scheduled to
processor $p$ with $m_1 \leq \eta(p)$, the rescheduling has to be per-
formed to find out another destination processor.

C.2. The Full State

In addition to the action taken in the nearly-full state, a
more restricted scheme is applied such that any data messages
addressed to the processor $p$ with $m_2 \leq \eta(p)$ are deferred. They
are buffered at the original processor and wait for change of
the destination processor's state. Although these data messages
are eventually sent to their destination processor, the delay in
sending can help the processor relax its memory tightness.
Note that these deferred data messages will be buffered sepa-
rately and not be counted when calculating $\Theta_k$.

C.3. The Emergency State

If processor $p$ runs out of memory, several actions can be
taken before we declare its failure. One is to clear up all resid-
ual data messages and process messages, if there are any. An-
other is to redistribute the unprocessed process messages that
have been previously placed in this processor. If any memory
space can be released at this time, we will rescue the processor
from its emergency state, and let the system continue on.

IV. PERFORMANCE

The P kernel is currently written in MPL, running on a 16K-
processor Maspar MP-1 with 32K bytes memory per processor.
MPL is a C-based data parallel programming language.
We have tested the P kernel system using two sample pro-
grams: the N-queen problem and the GROMOS Molecular
Dynamics program [38], [37]. GROMOS is a loosely synchronous problem. The test data is the bovine superoxide dismutase molecule (SOD), which has 6,968 atoms [28]. The cutoff radius is predefined to 8 Å, 12 Å, and 16 Å.

The total execution time of a P kernel program consists of two parts, the time to execute the system program, \( T_{sys} \), and the time to execute the user program, \( T_{usr} \). The system efficiency is defined as follows:

\[
\mu_{sys} = \frac{T_{usr}}{T_{usr} + T_{sys}},
\]

where \( T \) is the total execution time. The system efficiency of the example shown in Fig. 7 is:

\[
\mu_{sys} = \frac{0.4 \times 3}{0.4 \times 3 + 0.1 \times 3} = 80\%.
\]

The system efficiency depends on the ratio of the system overhead to the grainsize of atomic computations. Table IV shows the system efficiency on the 1K-processor MP-1. The high efficiency results from the high ratio of granularity to system overhead. The P kernel executes in a loosely synchronous fashion and can be divided into iterations. Each iteration of execution consists of a system program phase and a user program phase. The execution time of a system program varies between 8 and 20 milliseconds on MP-1. The average grainsizes of a user phase in the test programs are between 55 and 330 milliseconds.

In the system phase, every processor participates in the global actions. On the other hand, in the user program phase, not all processors are involved in the execution of the selected atomic computation. In each iteration \( i \), the ratio of the number of participating processors \( N_{active}(i) \) to the total number of processors \( N \) is defined as utilization \( u(i) \):

\[
u(i) = \frac{N_{active}(i)}{N}.
\]

For example, the utilization of iteration 1 in Fig. 7 is 75%, since three out of four processors are active.

The utilization efficiency is defined as follows:

\[
\mu_{util} = \frac{\sum_{i=1}^{m} u(i)T(i)}{\sum_{i=1}^{m} T(i)} = \frac{\sum_{i=1}^{m} N_{active}(i)}{N \sum_{i=1}^{m} T(i)}
\]

where \( T(i) \) is the execution time of \( i \)th iteration and \( \Sigma_{i=1}^{m} T(i) = T_{usr} \). The utilization efficiency depends on the computation selection strategy and the load balancing scheme. Table V shows the utilization efficiencies for different problem sizes with the priority auction algorithm and random placement load balancing.

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Utilization Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>12-queen</td>
<td>80%</td>
</tr>
<tr>
<td>13-queen</td>
<td>87.3%</td>
</tr>
<tr>
<td>14-queen</td>
<td>89.6%</td>
</tr>
<tr>
<td>8 Å</td>
<td>90.3%</td>
</tr>
<tr>
<td>12 Å</td>
<td>98.0%</td>
</tr>
<tr>
<td>16 Å</td>
<td>98.6%</td>
</tr>
</tbody>
</table>

In irregular problems, the grainsizes of atomic computations may vary substantially. The grainsize variation heavily depends on how irregular the problem is and how the program is partitioned. At each iteration, the execution time of the user phase depends on the largest atomic computation among all active processors. The other processors that execute the atomic computation of smaller grainsizes will not fully occupy that time period. We use an index, called fullness, to measure the grainsize variation of atomic computation for each iteration:

\[
f(i) = \frac{\sum_{k=1}^{N_{active}(i)} T_k(i)}{T(i)}
\]

where \( T_k(i) \) is the user computation time of the \( k \)th participated processor at iteration \( i \). For example, the fullness of iteration 1 in Fig. 7 is

\[
f(1) = \frac{0.4 + 0.2 + 0.3}{0.4 \times 3} = 75\%.
\]

The fullness efficiency is defined as:

\[
\mu_{full} = \frac{\sum_{i=1}^{m} f(i)T(i)N_{active}(i)}{\sum_{i=1}^{m} T(i)N_{active}(i)} = \frac{\sum_{i=1}^{m} \sum_{k=1}^{N_{active}(i)} T_k(i)}{\sum_{i=1}^{m} T(i)N_{active}(i)}
\]

Table VI shows the fullness efficiencies for different problem sizes. The N-queen is an extremely irregular problem and has a substantial grainsize variation and a low fullness efficiency. The GROMOS program is loosely synchronous and the grainsize variation is small.
Now we define the overall efficiency as follows:

$$\eta = \eta_{sys} \times \eta_{alg} \times \eta_{util}$$

The system efficiency can be increased by reducing system overhead or increasing the grainsize. It is not realistic to expect a very low system overhead because the system overhead is dominated by communication overhead. The major technique of increasing system efficiency is the grainsize. It is not realistic to increase the grainsize. The major technique of increasing system efficiency is the grainsize control. That is, the average grainsize of atomic computations should be much larger than the system overhead. The utilization efficiency depends on the computation selection and load balancing algorithms. The algorithms discussed in this paper deliver satisfactory performance, as long as the number of processes is much larger than the number of processors. The most difficult task is to reduce the grainsize variation which determines the fullness efficiency. The grainsize variation depends on the characteristics of the application program. However, it is still possible to reduce the grainsize variation. The first method is to select a proper algorithm. The second method is to select a good program partition. For a given application, there may be different algorithms to solve it and different partitioning patterns. Some of them may have small grainsize variation and some may not. A carefully selected algorithm and partitioning pattern can result in a higher fullness efficiency.

The overall efficiencies of the N-queen problem and the GROMOS program on the 1K-processor MP-1 are shown in Table VII. Compared to the N-queen problem, GROMOS has a much higher efficiency mainly because of its small grainsize variation.

Tables VIII and IX show the execution times and speedups of the N-queen problem and the GROMOS program, respectively. A high speedup has been obtained from the GROMOS program. The N-queen is a difficult problem to solve, but a fairly good performance has been achieved.

**TABLE VI**

<table>
<thead>
<tr>
<th>N-queen</th>
<th>GROMOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>12-queen</td>
<td>13-queen</td>
</tr>
<tr>
<td>19.2%</td>
<td>18.0%</td>
</tr>
</tbody>
</table>

**TABLE VII**

<table>
<thead>
<tr>
<th>N-queen</th>
<th>GROMOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>12-queen</td>
<td>13-queen</td>
</tr>
<tr>
<td>13.8%</td>
<td>15.2%</td>
</tr>
<tr>
<td>74.4%</td>
<td>82.7%</td>
</tr>
</tbody>
</table>

**TABLE VIII**

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>12-queen</th>
<th>13-queen</th>
<th>14-queen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec.)</td>
<td>Speedup</td>
<td>Time (sec.)</td>
<td>Speedup</td>
</tr>
<tr>
<td>1K</td>
<td>10.1</td>
<td>197</td>
<td>142</td>
</tr>
<tr>
<td>2K</td>
<td>6.82</td>
<td>210</td>
<td>31.5</td>
</tr>
<tr>
<td>4K</td>
<td>4.61</td>
<td>311</td>
<td>18.7</td>
</tr>
<tr>
<td>8K</td>
<td>3.19</td>
<td>458</td>
<td>11.2</td>
</tr>
<tr>
<td>16K</td>
<td>2.37</td>
<td>604</td>
<td>6.78</td>
</tr>
</tbody>
</table>

**TABLE IX**

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>8 A</th>
<th>12 A</th>
<th>16 A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec.)</td>
<td>Speedup</td>
<td>Time (sec.)</td>
<td>Speedup</td>
</tr>
<tr>
<td>1K</td>
<td>18.3</td>
<td>761</td>
<td>24.3</td>
</tr>
<tr>
<td>2K</td>
<td>13.1</td>
<td>1397</td>
<td>15.2</td>
</tr>
<tr>
<td>4K</td>
<td>4.20</td>
<td>2410</td>
<td>18.4</td>
</tr>
<tr>
<td>8K</td>
<td>3.53</td>
<td>900</td>
<td>7.50</td>
</tr>
<tr>
<td>16K</td>
<td>1.00</td>
<td>6326</td>
<td>2.01</td>
</tr>
</tbody>
</table>

**V. CONCLUDING REMARKS**

The motivation of this research is twofold: to prove whether a SIMD machine can handle asynchronous application problems, serving as a general-purpose machine, and to study the feasibility of providing a truly portable parallel programming environment between SIMD and MIMD machines.

The first version of the P kernel was written in CM Fortran, running on a 4K-processor TMC CM-2 in 1991. The performance was reported in [32]. Fortran is not the best language for system programs. We used the multiple dimension array with indirect addressing to implement queues. Hence, not only was the indirect addressing extremely slow, but accessing different addresses in CM-2 costed much more [9]. In 1993, the P kernel was rewritten in MPL and ported to Maspar MP-1. The MPL version reduces the system overhead and improves the performance substantially. Experimental results have shown that the P kernel is able to balance load fairly well on SIMD machines for nonuniform applications. System overhead can be reduced to a minimum with granularity control.

**ACKNOWLEDGMENTS**

This research was partially supported by National Science Foundation grants CCR-9109114 and CCR-8809615.

We are very grateful to Reinhard Hanxleden for providing the GROMOS program, and Terry Clark for providing the SOD data. We also thank Alan Karp, Guy Steele, Hank Dietz, and Jerry Roth for their comments. We would like to thank the anonymous reviewers for their constructive comments. The performance data was gathered on the MP-1 at NPAC, Syracuse University.

**REFERENCES**


Wei Shu received the PhD degree from the University of Illinois at Urbana-Champaign in 1990, the MS degree from Santa Clara University, and the BS degree from Hefei Polytechnic College, China. From 1989 to 1990, she worked at Yale University as an associate research scientist. Since then, she has been an assistant professor in the Department of Computer Science at the State University of New York at Buffalo. Her current interests include dynamic scheduling, runtime support systems for parallel processing, and parallel operating systems.

Min-You Wu received the MS degree from the Graduate School of Academia Sinica, Beijing, China, and the PhD degree from Santa Clara University, Santa Clara, California. Before he joined the Department of Computer Science, State University of New York at Buffalo, where he is currently an assistant professor, he held various positions at the University of Illinois at Urbana-Champaign, University of California at Irvine, Yale University, and Syracuse University. His research interests include parallel operating systems, compilers for parallel computers, programming tools, application of parallel systems, and VLSI design. He has published over 40 journal and conference papers in the above areas. He served as a guest editor for the Journal of Supercomputing and Scientific Programming. He is a member of IEEE.