OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

90-80-5

Implementing a Time-Driven Simulation on a MJIMD Computer Using a SIMD Language

Michael J. Quinn
Bradley K. Seevers
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3202

Philip J. Hatcher
Department of Computer Science
University of New Hampshire
Durham, NH 03824

Implementing a Time-Driven Simulation on a MIMD Computer Using a SIMD Language

Michael J. Quinn†, Bradley K. Seevers†, and Philip J. Hatcher‡

† *Department of Computer Science, Oregon State University, Corvallis, OR 97331, U.S.A.*
‡ *Department of Computer Science, University of New Hampshire, Durham, NH 03824, U.S.A.*

Abstract

It often makes sense to write a program in the SIMD style, even if the program is to execute on a MIMD computer. Simulating physical events, in which all motion takes place simultaneously, is one area in which SIMD languages fit the applications particularly well. In this paper we present the SIMD programming language Dataparallel C and describe how we compile Dataparallel C programs into C code suitable for efficient execution on shared memory multiprocessors. We outline the parallel implementation of the Wa-Tor model and benchmark the performance of the compiled Dataparallel C program on the Sequent Balance™ and Sequent Symmetry™ multiprocessors.

1. Introduction

The compute-intensive nature of many simulation problems has led to an increasing interest in the use of parallelism to reduce execution time [1]. If the simulation to be performed is stochastic in nature, and the goal is to perform long simulation runs to reduce variance, or if a particular simulation problem must be executed for a variety of parameter settings, then the simplest and probably most efficient approach is to execute independent, sequential simulation programs on different processors [2, 3]. The focus of this paper, however, is the problem of using parallelism to solve a single simulation problem.

Sequential simulation algorithms are generally event-driven. A typical event-driven simulator uses two data structures to drive the computation: state variables and and event list. The state variables describe the condition of the system. The event list contains pending events, sorted in nondecreasing order of the simulation time at which they will occur. After processing one event, the simulator moves to the first pending event on the event list and advances the simulation time to match the time of the event.

Parallel simulation algorithms may be time-driven or event-driven. Time-driven parallel simulation algorithms are characterized by a single global time, shared by all processes. Concurrency is achieved when multiple events occur at the same time. Event-driven parallel simulation algorithms are characterized by a lack of a single global time. Concurrency is achieved through the simulation of multiple events that may occur at different times.

Because earlier events can influence later events, causality errors can occur if events are simulated in the wrong order. Event-driven simulations may be divided into two

Balance and Symmetry are trademarks of Sequent Computer Systems, Inc. C* is a trademark of Thinking Machines Corporation.

categories, depending upon their approach to causality errors. Conservative methods strictly avoid causality errors. Bryant [4] and Chandy and Misra [5] have developed well known conservative algorithms for parallel discrete event simulation. Optimistic methods detect and recover from causality errors. The best known optimistic algorithm is Time Warp, developed by Jefferson [6].

Event-driven parallel simulations are more suitable when the number of events per time step is small, and when the time needed to process an event is relatively large. Conversely, time-driven parallel simulations are more suitable when the number of events happening at any time step is large, and when the grain size of the event is small.

Our goal in this paper is to illustrate that SIMD (single instruction stream, multiple data stream) languages can simplify the programming of time-driven simulations on MIMD (multiple instruction stream, multiple data stream) computers. To that end we have chosen for our case study the parallel implementation of Wa-Tor, a simple simulation problem characterized by a high degree of parallelism at every time step and fine-grained events. In the remainder of this paper we present the SIMD programming language Dataparallel C and describe how we compile Dataparallel C programs into C code suitable for efficient execution on shared memory multiprocessors. We outline the parallel implementation of the Wa-Tor model, and we conclude by presenting the speedup achieved by the compiled Dataparallel C program on the Sequent Balance and Sequent Symmetry multiprocessors.

2. The Dataparallel C Programming Language

The Dataparallel C programming language is very similar to the original C*[TM] language designed by Rose and Steele [7]. We have added the notion of *virtual topologies*, extended the specification of pointers, and made array assignment a part of the language. In this section we summarize the features of the language; a detailed description appears in [8].

2.1. Virtual Processors

The conceptual model presented to the Dataparallel C programmer is that of a SIMD computer: a front-end uniprocessor attached to an adaptable back-end parallel processor. The sequential portion of the Dataparallel C program (consisting of conventional C code) is executed on the front end. The parallel portion of the Dataparallel C program (delimited by constructs not found in C) is executed on the back end.

The back end is adaptable in that the programmer selects the number of processors to be activated. This number is independent of the number of physical processors that may be available on the hardware executing the Dataparallel C program. For this reason the Dataparallel C program is said to activate *virtual* processors when a parallel construct is entered.

Virtual processors are allocated in groups. Each virtual processor in the group has an identical memory layout. The Dataparallel C programmer specifies a virtual processor's memory layout using syntax similar to the C `struct`. A new keyword `domain` is used

```
domain cell { struct key_info key, *nbr[DIRECTIONS];
              int             seed;
              unsigned char   direction, prior_kind;
              void            look_for(), swim_and_breed();
            } ;
```

Fig. 1. Declaring the Wa-Tor domain.

```
domain cell ocean[CELLS][CELLS];
```

Fig. 2. Declaring virtual processors. The world of Wa-Tor is a CELLS×CELLS torus.

```
[domain cell].{
   seed = ID;
   nbr[0] = &north->key;
   nbr[1] = &east->key;
   nbr[2] = &south->key;
   nbr[3] = &west->key;
}
```

Fig. 3. Activating virtual processors through a domain select statement.

to indicate that this is a parallel data declaration. Figure 1 contains the domain declaration for the Wa-Tor simulation. As in C structures, the names declared within the domain are referred to as *members*.

Instances of a domain are declared using the C array constructor. Each domain instance becomes the memory for one virtual processor. The array dimension, therefore, indicates the size of the virtual back-end parallel processor that is to be allocated. Figure 2 contains a domain array declaration. Note that domain arrays can be multidimensional, in which case the number of virtual processors allocated is the product of the array dimensions.

Figure 3 illustrates the Dataparallel C *domain select* statement. The body of the domain select is executed by every virtual processor allocated for the particular domain type selected. The virtual processors execute the body synchronously. The domain members `seed` and `nbr` are included within the scope of the body of the domain select. These names refer to the values local to a particular virtual processor.

2.2. Mono and Poly Data

Data located in Dataparallel C's front-end processor is termed *mono* data. Data located in a back-end processor is termed *poly* data.

The code executing in a virtual processor of a Dataparallel C program can reference a variable in the front-end processor by referring to the variable by name. A variable that is visible in the immediately enclosing block of a domain select statement is visible within the domain select.

Similarly, the members of a domain instance are accessible everywhere in a program. The members of one domain can be read and written from within a domain select statement for a different domain. Poly data can also be read and written from the sequential portion of the program. The syntax employed is to provide a full domain array reference followed by a member reference.

Dataparallel C, like C++, has a keyword `this`. In Dataparallel C `this` is a pointer to the domain instance currently being operated on by a virtual processor. Pointer arithmetic on `this` can be performed to access other virtual processors' members. For example, the Wa-Tor program has seven macros

```
#define ID    (this-&ocean[0][0])
#define ROW   (ID/CELLS)
#define COL   (ID%CELLS)
#define north (ROW ? (this-CELLS) : (this+CELLS*(CELLS-1)))
#define south ((ROW==(CELLS-1)) ? (this-CELLS*(CELLS-1)) : (this+CELLS))
#define east  ((COL==(CELLS-1)) ? (this-(CELLS-1)) : (this+1))
#define west  (COL ? (this-1) : (this+(CELLS-1)))
```

that allow each virtual processor to access the memories of its neighbors to the north, east, south, and west.

Dataparallel C supports general pointer-based communication. A parallel pointer dereference operation can produce a nearly arbitrary memory access pattern. Each virtual processor's memory access is determined solely by the contents of its local pointer value.

2.3. Flow of Control

The sequential portion of a Dataparallel C program is just C code and executes according to the normal C semantics. Conceptually, the parallel sections of a Dataparallel C program execute synchronously under the control of a *master program counter* (MPC). A virtual processor's local program counter is either active, executing in step with the MPC, or inactive, waiting for the MPC to reach it.

For example, the MPC steps through an `if-then-else` statement by first evaluating the control expression, then executing the `then` clause, and finally executing the `else` clause. A local program counter would also proceed first to the control expression. However, if the expression evaluated to zero (*false* in C), then the local program counter would proceed to the `else` clause and wait for the MPC to reach it. If the expression evaluated to nonzero (*true* in C), then the local program counter would wait at the `then` clause for the MPC.

As well as being synchronous at the statement level, Dataparallel C is also synchronous at the expression level. No operator executes within a virtual processor unless all active virtual processors have evaluated their operands for the operator. Once the operands have been evaluated, the operator is executed as if in some serial order by all active virtual processors. This seemingly odd use of a serial ordering to define parallel execution is required to make sense of concurrent writes to the same memory location.

Because the execution of Dataparallel C programs is synchronous at the expression level, it is actually easier to implement algorithms requiring simultaneous motion in Dataparallel C than in C. For example, consider the problem of performing a cyclic rotation of an array. Because C manipulates only a single value at a time, a temporary variable must be introduced. In contrast, the entire assignment can be performed in a single step in Dataparallel C.

Dataparallel C allows member functions to be defined for domain types. A Dataparallel C member function is similar to a C++ member function in that the names of the other members of the domain are visible within the body of the member function. Unlike C++, however, a Dataparallel C member function is a parallel control structure. The body of the member function is executed synchronously in parallel for each active virtual processor of the corresponding domain type.

Member functions may be called from either sequential code or from within a domain select. If a member function is called from sequential code, then every virtual processor executes the function. If the member function is called from within a domain select statement, then it is executed by only those virtual processors that are active at the time the function is invoked. Figure 4 contains a member function, as well as a code segment that invokes the function.

3. Compiling Dataparallel C Programs for a Shared Memory Multiprocessor

Our Dataparallel C compiler generates C code as output. The compiler parses Dataparallel C input, transforms Dataparallel C syntax trees into C syntax trees, and then unparses C code suitable for compilation and execution on the Sequent Balance and Sequent Symmetry multiprocessors. In this section we provide a brief overview of the compiler. More detailed descriptions appear in [8, 9].

The abstract model of parallel computation imagined by the Dataparallel C programmer has virtual processors and synchronous execution. The role of the compiler is to bridge the gap between the abstract machine and the actual architecture in as efficient a way as possible.

The compiler translates synchronous data-parallel programs into SPMD (single program, multiple data) [10] programs suitable for execution on a shared memory multiprocessor. The key to the efficient implementation of a data-parallel language on an asynchronous MIMD computer is the realization that physical processors need only synchronize when virtual processors interact. Between these points every physical processor can emulate its virtual processors at full speed. The compiler inserts barrier synchronizations at those points in the emitted C code where synchronization is necessary. To a great extent the performance of the generated code relative to a hand-coded parallel C program is determined by the number of barriers the compiler inserts.

All variables, both mono and poly, are stored in shared memory, where they are accessible by all processors. Sequential code is executed by only a single process. During the execution of the C program generated by the compiler, the state of the system switches from single process execution to multiple process execution every time a domain select or domain member

```
main (argc,argv)
   int argc; char *argv[];
{

   ...
   while (its-- > 0) {
      [domain cell].{
          prior_kind = key.kind;
          key.moved = FALSE;
          if (key.kind == FISH) {
             key.age++;
             look_for(WATER);
          }
      }
   ...
}


void cell::look_for(desired_kind)
   int desired_kind;
{
   int found_it, i;

   found_it = FALSE;
   direction = (int) 4 * random(&seed);
   for (i = 0; (i < DIRECTIONS) & (found_it == FALSE); i++) {
      direction = (direction + 1) % DIRECTIONS;
      if (nbr[direction]->kind == desired_kind) found_it = TRUE;
   }
   if (found_it) nbr[direction]->id = ID;
   else direction = DIRECTIONS;
}
```

Fig. 4. Member function `look_for` from the Wa-Tor simulation, along with a code segment invoking the function. Only those virtual processors whose value of `key.kind` equals `FISH` execute the member function.

function is entered, and the state reverts to single process execution when the locus of control returns to sequential code.

Synchronizations are needed when one virtual processor defines a variable that is later read by another virtual processor, or vice versa. A data flow analysis of the Dataparallel C program yields these def-use and use-def dependencies. The compiler examines the entire set of constraints and attempts to derive a minimal set of barriers with the property that every use-def and def-use dependency spans at least one barrier. In addition, when it has the opportunity, the compiler places the barrier at the least disruptive point. For example, everything else being equal, it inserts a barrier between two statements, rather than at some

point in the evaluation of an expression, which would require the introduction of a temporary variable.

Every physical processor must participate in every barrier synchronization. If a control structure contains a barrier synchronization, then the structure must be rewritten to pull the barrier to the outermost level.

After the compiler has added synchronization points to the Dataparallel C program and transformed the control structures to pull the barriers to the outermost level, the program consists of sequential code and domain select statements separated by barriers. At this point the compiler replaces all Dataparallel C domain select statements with C `for` loops that emulate virtual processors. Given p physical processors and interleaved emulation, every processor i emulates every pth virtual processor, beginning with virtual processor i. The contiguous virtual processor emulation strategy partitions the virtual processors into p contiguous regions, with the largest region having no more than one virtual processor more than the smallest region.

4. Wa-Tor

In the December 1984 issue of *Scientific American*, Dewdney describes a computer program to model the ecology of the mythical planet Wa-Tor [11]. The planet Wa-Tor is shaped like a torus, is covered completely by water, and is inhabited by two higher life forms, called sharks and fish. The computer program models the interactions of the predators (sharks) and their prey (fish). We have chosen Wa-Tor as a case study, because even though the model is simple, it shares attributes with more sophisticated physical models. In particular, we extend Dewdney's specification to require that each class of animal move simultaneously. Modeling the simultaneous movement of multiple objects is error-prone, when only a single object can be examined at a time. For this reason the Dataparallel C program is actually simpler than the corresponding sequential program written in C.

The planet surface is modeled by a two-dimensional array. Each cell may be occupied by a fish or a shark, or it may be empty. The simulation is divided into time steps. Every time step has two phases. In the first phase the fish move; in the second phase the sharks move. Fish move by looking for an adjacent empty cell in one of four directions: north, south, east, or west. If there are empty cells in more than one direction, the fish has an equal probability of choosing each of the available cells as the cell to which it would like to swim. After the fish have moved, it is the sharks' turn. First the sharks examine neighboring cells to see if any of them contain fish. If one or more of the adjacent cells contains a fish, the shark randomly chooses one of the cells. When a shark enters a cell containing a fish, it devours the fish. During the second part of the sharks' phase those sharks unable to eat a fish try to locate an empty cell, and if such a cell is found, swim in that direction. When seeking and swimming toward empty cells, the behavior of the sharks is identical to that of the fish.

Recall that all fish simultaneously attempt to move to an adjacent empty cell during their phase. Likewise, all sharks simultaneously attempt to eat fish, and those sharks unable to eat a fish attempt to move to adjacent empty cells at the same time. One of the principal

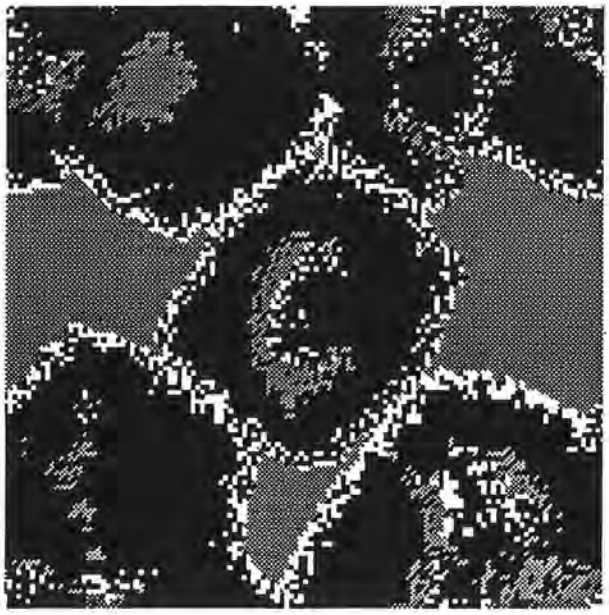

Fig. 5. Typical state of the Wa-Tor world. Fish are grey and sharks are white. This scene depicts a 128–by-128 torus.

complicating factors resulting as a consequence is that more than one swimmer may attempt to enter the same cell. When such collisions occur, the rule is that only one creature occupies the cell, and the other contenders must retreat to their former locations. For this reason movement is divided into two phases. During the first phase fish or sharks determine the direction in which they would like to travel and place their identifying number in that cell they want to occupy. If more than one creature places its number in the same cell, only one number will remain. After this phase has been completed, fish can check their desired destination to see if they have "won the right" to swim to that cell.

Whenever a shark or a fish successfully moves to a new cell, if the age of the creature is greater than or equal to the age at which it can bear offspring, a baby is left behind at the cell just vacated. The age of the child is 0. The age of the parent is reset to 0, to prevent an animal from bearing an offspring every iteration after it reaching maturity.

Figure 5 illustrates a typical state of the Wa-Tor world. In this figure fish are gray and sharks are white. Readers interested in learning more about the rules of Wa-Tor and the implementation of the model on MIMD computers can find more information in [11, 12, 13].

We have claimed that many physical models are more amenable to solution via SIMD languages than MIMD languages. To buttress our claim, we present the Dataparallel C implementation of the Wa-Tor simulation in its entirety. The entire code is less than 150 lines long, excluding I/O and comments. In contrast, the MIMD-style implementation of Wa-Tor appearing in Angus *et al.* [13] requires more than 600 lines of C code, excluding I/O and comments.

```
/*
 *   Wa-Tor in Dataparallel C
 */

#include <stdio.h>

#define DIRECTIONS  4       /* North, East, South, West */
#define WATER       4       /* Color of water cell */
#define FISH        2       /* Color of fish cell */
#define SHARK       0       /* Color of shark cell */
#define CELLS       128     /* Ocean size */
#define ITERATIONS  100     /* Simulation length */
#define FISHINIT    1000    /* Initial number of fish */
#define FISHBREED   3       /* Fish breeding age */
#define SHARKINIT   100     /* Initial number of sharks */
#define SHARKBREED  3       /* Shark breeding age */
#define SHARKSTARVE 4       /* Shark starving time */
#define FALSE       0
#define TRUE        1

struct pixelinfo { unsigned char x, y, grayvalue, dummy; }
   buffer[CELLS*CELLS+FISHINIT+SHARKINIT];

#define ID    (this-&ocean[0][0])
#define ROW   (ID/CELLS)
#define COL   (ID%CELLS)
#define north (ROW ? (this-CELLS) : (this + CELLS*(CELLS-1)))
#define south ((ROW==(CELLS-1)) ? (this-CELLS*(CELLS-1)) : (this+CELLS))
#define east  ((COL==(CELLS-1)) ? (this-(CELLS-1)) : (this+1))
#define west  (COL ? (this-1) : (this+(CELLS-1)))

struct key_info { unsigned char age, kind, moved, starved;
                  int id; } ;


domain cell {
   struct key_info key;
   int             seed;
   unsigned char   prior_kind;
   unsigned char   direction;
   void            look_for(), swim_and_breed();
   struct key_info *nbr[DIRECTIONS];
          } ocean[CELLS][CELLS];


int buffersize;

float random();

main (argc,argv)
   int argc; char *argv[];
{
```

```
    void initialize_ocean();

    int  its = ITERATIONS;
    int  x, y;

    initialize_ocean();

    [domain cell].{       /* Each cell stores pointers to neighbors */
       seed = ID;
       nbr[0] = &north->key;
       nbr[1] = &east->key;
       nbr[2] = &south->key;
       nbr[3] = &west->key;
    }

    while (its-- > 0) {
       [domain cell].{
           prior_kind = key.kind;
           key.moved = FALSE;
           if (key.kind == FISH) {
              key.age++;
              look_for(WATER);              /* Fish look for water */
              swim_and_breed(FISHBREED)     /* Fish swim and breed */
           }
           if (key.kind == SHARK) {
              key.age++;
              key.starved++;
              look_for(FISH);               /* Sharks look for fish */
              swim_and_breed (SHARKBREED); /* Sharks swim and breed */
           }
           if (key.kind == SHARK) {
              if (!key.moved) {
                 look_for(WATER);     /* Unfed sharks look for water */
                 if (key.starved >= SHARKSTARVE) {
                    key.kind = WATER;
                 }
                 else swim_and_breed (SHARKBREED);  /* Unfed sharks swim */
              }
           }
      }
      buffersize = 0;
      for (x = 0; x < CELLS; x++)
         for (y = 0; y < CELLS; y++)
            if (ocean[x][y].key.kind != ocean[x][y].prior_kind) {
               buffer[buffersize].x = x;
               buffer[buffersize].y = y;
               buffer[buffersize++].grayvalue = ocean[x][y].key.kind;
            }
      write (1, buffer, buffersize*sizeof(struct pixelinfo));
   }
}
```

```
void cell::look_for(desired_kind)
   int desired_kind;
{
   int found_it, i;

   found_it = FALSE;
   direction = (int) 4 * random(&seed);
   for (i = 0; (i < DIRECTIONS) & (found_it == FALSE); i++) {
      direction = (direction + 1) % DIRECTIONS;
      if (nbr[direction]->kind == desired_kind) found_it = TRUE;
   }
   if (found_it) nbr[direction]->id = ID; /* Claim neighboring cell */
   else direction = DIRECTIONS;           /* No suitable neighbors */
}

void cell::swim_and_breed (breed)
   int breed;
{
   struct key_info newkey;

   if (direction < DIRECTIONS) {
      if (nbr[direction]->id == ID) {   /* If swimmer wins cell... */
         newkey.moved = TRUE;
         newkey.kind = key.kind;
         if (key.kind == SHARK) {
            if (nbr[direction]->kind == FISH) key.starved = 0;
            newkey.starved = key.starved;
         }
         if (key.age >= breed) {
            newkey.age = key.age = 0;   /* Leave offspring behind */
            key.moved = TRUE;
         } else {
            key.kind = WATER;           /* Leave water behind */
            newkey.age = key.age;
         }
         *nbr[direction] = newkey;
      }
   }
}

void initialize_ocean()
{
   int i, x, y;
   int seed;

    /* initialize the ocean */
   [domain cell].{ key.kind = WATER; }

    /* initialize our share of fish and sharks */
    seed = 2;

    for (i = 0; i < FISHINIT; i++) {
```

```
        x = CELLS * random (&seed);
        y = CELLS * random (&seed);
        ocean[x][y].key.kind = FISH;
        ocean[x][y].key.age = i%FISHBREED;
    }
    for (i = 0; i < SHARKINIT; i++) {
        x = CELLS * random (&seed);
        y = CELLS * random (&seed);
        ocean[x][y].key.kind = SHARK;
        ocean[x][y].key.age = i % SHARKBREED;
        ocean[x][y].key.starved = 0;
    }
    buffersize = 0;
    for (x = 0; x < CELLS; x++)
        for (y = 0; y < CELLS; y++) {
            buffer[buffersize].x = x;
            buffer[buffersize].y = y;
            buffer[buffersize++].grayvalue = ocean[x][y].key.kind;
        }
    write (1, buffer, buffersize*sizeof(struct pixelinfo));
}
```

5. Benchmark Results and Conclusions

We have compiled the Dataparallel C Wa-Tor program and executed it on a Sequent Balance and a Sequent Symmetry, both shared memory multiprocessors. We recorded the time needed for the parallel program to execute 100 iterations, excluding time spent performing I/O.

Table 1 contains the execution times and speedups of the Dataparallel C program executing on the Balance and the Symmetry. We define speedup to be the time required by our best sequential algorithm divided by the time required by the parallel algorithm. Given the same input parameters, our best sequential C Wa-Tor program executed in 622 seconds on the Sequent Balance and 249 seconds on the Sequent Symmetry. The execution times on the two machines are plotted in Figure 6. The irregularities in the curves are caused by imbalances in the work performed by each processor. The compiler statically assigns virtual processors to physical processors, and there is no dynamic load balancing mechanism in the current system.

We have contended that it sometimes makes sense to program MIMD computers in SIMD languages, and the Wa-Tor problem is a case in point. Our Dataparallel C implementation of Wa-Tor is less than 25% the length of the MIMD-style Wa-Tor program published by Angus et al. [13]. Our Dataparallel C compiler is able to take this program and generate code that achieves reasonable speedup on a MIMD computer.

Acknowledgments

This work was supported by National Science Foundation grant DCR-8906622. We appreciate the generosity of the Advanced Computing Research Facility, Mathematics and

Processors	Sequent Balance		Sequent Symmetry	
	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
1	691.8	0.9	289.7	0.9
2	396.6	1.6	154.8	1.6
3	284.8	2.2	98.4	2.5
4	215.7	2.9	74.2	3.4
5	163.6	3.8	61.5	4.0
6	137.0	4.5	52.1	4.8
7	113.5	5.5	42.3	5.9
8	90.6	6.9	41.8	6.0
9	80.4	7.7	35.3	7.1
10	77.7	8.0	33.3	7.5
11	70.6	8.8	28.5	8.7
12	67.7	9.2	28.6	8.7
13	63.2	9.8	24.2	10.3
14	60.5	10.3	24.3	10.2
15	61.0	10.2	22.3	11.2
16	59.5	10.5	27.9	8.9
17	58.4	10.7	20.5	12.1
18	60.2	10.3	19.8	12.6
19	59.8	10.4	19.4	12.8
20	64.0	9.7	19.3	12.9

Table 1 Execution time and speedup of compiled Dataparallel C Wa-Tor program on the Sequent Balance and Sequent Symmetry multiprocessors.

Computer Science Division, Argonne National Laboratory, and the Department of Computer Sciences at the University of Wisconsin-Madison, who gave us access to their Sequent Symmetry computers.

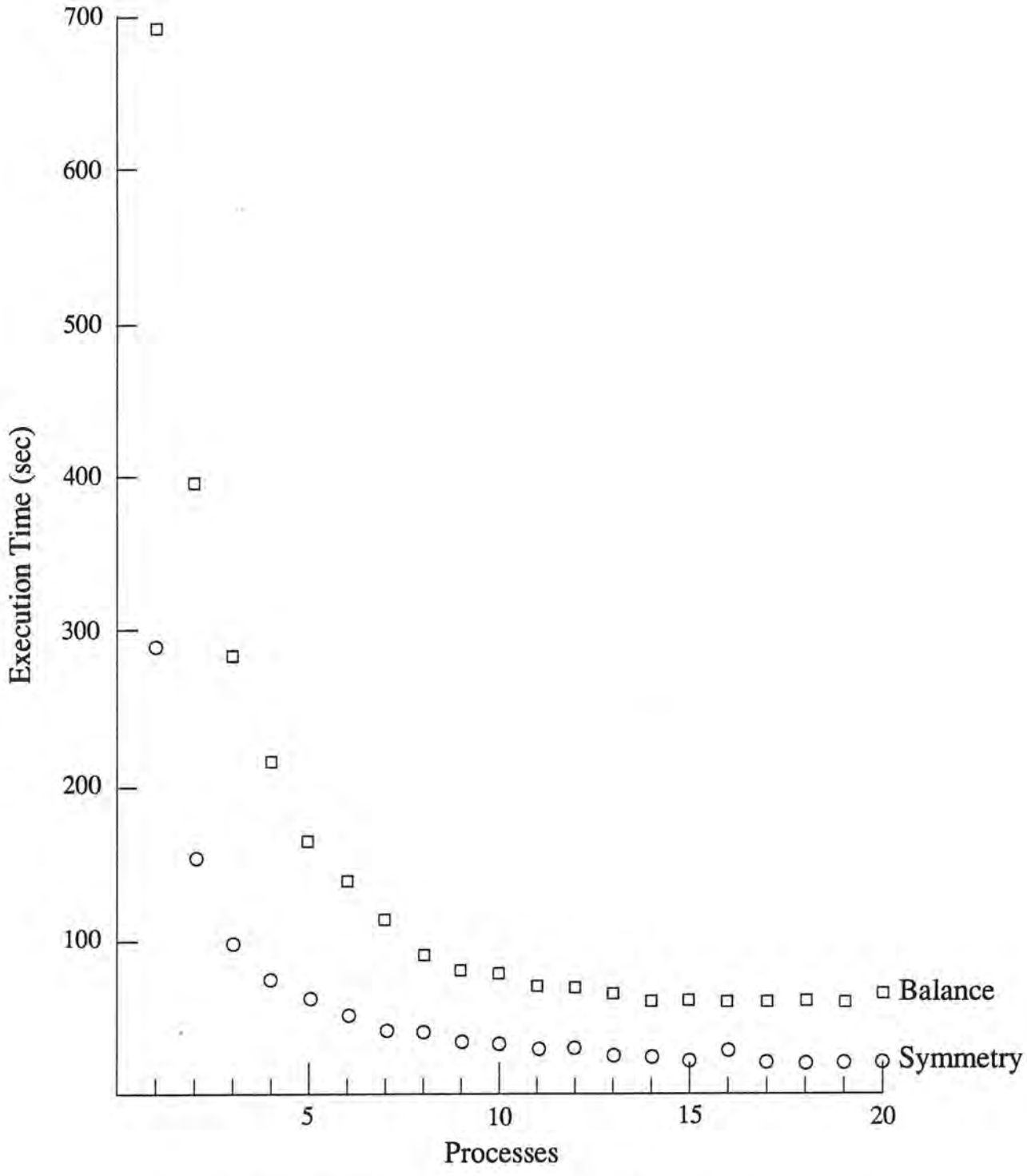

Fig. 6. Time needed to perform 100 iterations of 128 × 128-cell Wa-Tor program on Sequent Balance and Sequent Symmetry.

References

[1] R. M. Fujimoto, "Parallel discrete event simulation," in *1989 Winter Simulation Conference Proceedings*, pp. 19–28, 1989.

[2] W. E. Biles, C. M. Daniels, and T. J. O'Donnell, "Statistical considerations in simulation on a network of microcomputers," in *1985 Winter Simulation Conference Proceedings*, pp. 388–393, 1985.

[3] P. Heidelberger, "Statistical analysis of parallel simulations," in *1986 Winter Simulation Conference Proceedings*, pp. 290–295, 1986.

[4] R. E. Bryant, "Simulation of packet communication architecture computer systems," Tech. Rep. MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[5] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 440–452, Sept. 1979.

[6] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[7] J. R. Rose and G. L. Steele, Jr., "C*: An extended C language for data parallel programming," Tech. Rep. PL 87–5, Thinking Machines Corporation, 1987.

[8] P. J. Hatcher and M. J. Quinn, *Data-Parallel Programming*. MIT Press, to appear.

[9] M. J. Quinn, P. J. Hatcher, and B. K. Seevers, "Implementing a data parallel language on a tightly coupled multiprocessor," in *Third Workshop on Programming Languages and Compilers for Parallel Computers*, MIT Press, 1990.

[10] A. Karp, "Programming for parallelism," *IEEE Computer*, pp. 43–57, May 1987.

[11] A. K. Dewdney, "Computer recreations," *Scientific American*, pp. 14–22, Dec. 1984.

[12] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*, vol. 1. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[13] I. Angus, G. Fox, J. Kim, and D. Walker, *Solving Problems on Concurrent Processors*, vol. 2. Englewood Cliffs, NJ: Prentice-Hall, 1990.