# Yale University
# Department of Computer Science

## An Exploration of Asynchronous Data-Parallelism

Michael S. Littman and Christopher D. Metcalf

YALEU/DCS/TR-684
October 1988

# An Exploration of Asynchronous Data-Parallelism

Michael S. Littman
Christopher D. Metcalf

October 10, 1988

**Abstract**

In this paper, we develop the idea of *asynchronous data-parallelism*; that is, data-parallel programs in which the processing elements may be performing different computations at any given time. We suggest that since synchronous machines are forced to sequentialize the clauses of conditional statements, programming is hindered. Programmers must waste time factoring and merging expensive expressions from conditional clauses. This is still not always sufficient to use the machines at their maximum efficiency.

We propose a method by which a synchronous machine can be used to simulate a MIMD machine language called Milan. Milan can be used as an efficient target for existing data-parallel languages such as Crystal and PARALATION LISP. By keeping Milan's instruction set small and regular, we are able to keep the simulation overhead to a minimum and achieve impressive performance on a few simple problems.

We believe that Milan could be used as a research tool for computational models which require fine-grained asynchronous machines, such as very complex neural networks, CSP or parallel symbolic processing.

# 1 Introduction

## 1.1 Common Configurations

Large-scale multiprocessor computers typically come in one of two forms. The first is called *MIMD* because there may be Multiple Instructions and Multiple Data elements active simultaneously within the machine. To utilize the power of the machine's independent processors, programmers break the control flow of their programs into chunks which execute separately (*asynchronously*). This is known as *control-parallelism.*

A second type of multiprocessor is called *SIMD* for its Single Instruction stream. All the processors of a SIMD machine execute the same instruction at the same time (*synchronously*). To use the large number of processors on such machines, the program's data must be organized into groups. The elements of each group are distributed over the machine's processors and manipulated as a unit. The SIMD machine's instruction set is tailored for acting on data objects in this way. This style of programming is called *data-parallelism.*

Both of these configurations have distinct advantages and disadvantages. An asynchronous machine supports the parallel execution of multiple expressions, subroutines, and even programs. However, control-parallel algorithms needed to run on these machines can be difficult to express and debug due to tricky synchronization issues [1].

A data-parallel program is often a clearer description of the problem and may capture a greater fraction of the potential parallelism in a problem [1,2]. Unfortunately, synchronous machines can not easily compute differing expressions in parallel. This leads to the opinion that SIMD machines are not general purpose, that is, that they are limited to a certain class of problems.

To illustrate the tradeoffs in these models, consider the problem of computing

$$\sum_{i=1}^{n} i! + (n - i + 1)!$$

.

An asynchronous control-parallel approach is to create a list of subtasks to compute the factorials and a set of subtasks which add the factorials together. Each available processor removes a task from the list, performs it, and gets a new task. Since processors will be busy as long as there are tasks to be done, wasted time is kept to a minimum. However, the termination condition for the program can be hard to specify. The program must halt when there are no more pending additions *and* there are no pending factorials or factorials in progress.

To perform this computation in a synchronous data-parallel fashion, one creates a vector with an element for each $i$ from 1 to $n$. Next it steps through the factorial function for every value in the vector at once. The same is done for every $n - i + 1$. Finally the two vectors are added elementwise and reduced to a single value using addition. This is the opposite of the control-parallel algorithm. Here, the termination condition is straight-forward and no extra work needs to be done managing subtasks. But each processor spends a significant amount of time idle, waiting for other processors in the vector to compute the factorial.

# 2 An Alternative: Asynchronous Data-Parallelism

In this paper we suggest implementing a third configuration, that of *asynchronous data-parallelism.* An asynchronous data-parallel machine is one in which operations for creating and manipulating large data objects are available but the computations used to define the individual elements of these objects execute independently. This paradigm couples the power of asynchronous computation with the clarity of the data-parallel approach.

In terms of the example above, an asynchronous data-parallel machine could have each processor compute first $i!$ and then $(n - i + 1)!$ without having to waste time resynchronizing in between.

At the moment, there are only a few machines that might be able to support such a model directly; these include the Fluent machine [3], currently under development at Yale, the PASM (Partitionable SIMD/MIMD) machine [4], and the Ultracomputer [5], also still only on paper. Each of these is asynchronous, consists of many processors, and supports data-parallel operations. Implementation for other MIMD machines is not difficult, involving only the writing of primitives for synchronization and group operations, but not particularly attractive due to the limited number of processors typical on such machines.

To explore asynchronous data-parallelism we devised an asynchronous data-parallel machine language called Milan (Multiple Instruction LANguage) and implemented it on a commercially available synchronous machine, the Connection Machine-2 [6,7].

# 3 The Milan Virtual Machine

## 3.1 The Milan Approach

In principle, using a synchronous machine to simulate an asynchronous one is not very difficult. First, the code for the entire program is loaded into the memory for each processor. Next, some sort of instruction pointer is initialized to zero in every processor. Now the SIMD machine is used to step each processor through a standard "fetch-execute" cycle. This is done by looping through the set of possible commands, activating first all processors requesting addition, then multiplication, then logic operators, then control-flow operators, and so on through the entire instruction set.

This approach is inefficient. Implementing a typical machine language in this way would require 163 separate instructions in each cycle just to do computation./footnoteThis number comes from a rough count of the number of instructions in 68000 machine language. Adding a general communication scheme would slow things down even more.

Therefore, whereas the idea of simulating a MIMD machine using SIMD hardware is not unique to us/citeDP, doing so *practically* is new. The desired tradeoff is to win back in multiprocessing ability what is lost on simulation overhead.

Our approach to efficient MIMD simulation is similar to the approach used in RISC architectures, although for a different reason. A RISC machine minimizes the number of instructions to simplify and therefore speed up the existing ones, and to increase the amount of on-processor memory for registers. Our virtual machine shares the first goal: creating a highly-simplified set of commands makes each "virtual machine" sequential cycle through them faster, and thus makes the basic virtual MIMD cycle faster.

In this section, we describe Milan and explain the techniques we use to maximize its computational speed. The majority of speedup comes from having a very regular instruction set, collapsing similar computational instructions into single instructions (for instance, addition and relative jumps) and creating a simplified communication scheme. We show that this is sufficient to produce a surprisingly efficient interpreter.

## 3.2 Instruction Format

A Milan instruction consists of four sixteen-bit words: the opcode, two operands, and the result address. For monadic operators, the second operand is ignored[1]. Each of the sixteen bits of the opcode correspond to a *mini-instruction* (computation, addressing-mode, or communication command). For each bit in a processor's opcode that is a 1, that processor executes the corresponding mini-instruction. The regularity in the instruction format allows a high degree of SIMD parallelism to be achieved in instruction interpretation.

Table 1 contains a Milan instruction for decrementing location 5. The instruction is encoded as follows: the first bit means indirect the first operand. Since it is zero, `op1` is considered to be a constant. The second bit indirects the second operand. This causes `op2` to be replaced with the contents of location 5. The third bit negates `op1` and the fourth adds (the now indirected) `op2` to it. At the end of the cycle, `op1` is stored into **`res`**, in this case, back into location 5. The meanings of the other bits is the topic of the rest of this section (a summary of instruction bits can be found in Table 2).

[1] Except in the case of indexed indirection, as explained later.

| Opcode | op1 | op2 | res |
|---|---|---|---|
| 0111000000000000 | 1 | 5 | 5 |

Table 1: The Milan instruction for decrementing location 5.

### 3.3 Computation Mini-Instructions

In this section, we describe the ten mini-instructions which (when used in combination) are responsible for arithmetic computation, addressing-modes, boolean arithmetic, and program flow. We demonstrate the ability of these mini-instructions to express a large number of useful functions and describe simple optimizations which can be performed to speed up the instruction cycle.

All arithmetic computation in Milan is based on five simple mini-instructions: NEGATE, ADD, SIGNUM[2], MULTIPLY, and DIVIDE. Any computation which can be expressed as a subset of these commands *in this order* will execute in a single cycle. For instance: negation, addition, signum, multiplication, division, identity (no operation), subtraction (negation and addition), decrement (negation and addition), increment (addition), absolute value (signum and multiplication), comparison (negation, addition, and signum), rounding to nearest multiple (division and multiplication) all take unit time.

These arithmetic functions can be combined with memory indirection to create a host of addressing modes. We chose to have four possible indirections in each cycle:

1. Indirect the first operand.
2. Indirect the second operand.
3. Indirect the first operand again.
4. Indirect the result pointer.

This allows a possibility of 16 different addressing modes for dyadic operations in a single cycle. For monadic operations, there are an additional 48 addressing modes. This is a result of the fact that the third indirection above occurs immediately following the ADD in the cycle. This allows indexed operand indirection (for instance, referring to locations relative to the stack pointer) in a single cycle. Similarly, we add an INDEX mini-instruction following the fourth indirection which adds the second operand to the result pointer. This allows information to be stored relative to the contents of a location (like the stack pointer) in one cycle.

Using one and zero for true and false, we can express the fundamental boolean functions as follows:

[2] The SIGNUM function takes a number and returns -1, 0, or 1 depending on whether the number if negative, zero, or positive.

- **normalize.** (create a boolean from an arbitrary number). Signum and then absolute value (2 cycles).
- **logical not.** Subtract the value from one (1 cycle).
- **does not equal.** Compare and then absolute value (2 cycles).
- **less than or equal to.** Compare, then add one and signum (2 cycles).
- **greater than or equal to.** Compare, then subtract from one and signum (2 cycles).
- **equal to, greater than, less than.** Perform does not equal, less than or equal to, or greater than or equal to as appropriate, and subtract the result from 1 (3 cycles).
- **logical and.** Multiply (1 cycle).
- **logical or.** Addition and signum (1 cycle).
- **logical exclusive or.** Subtraction followed by absolute value (2 cycles).

Since the instruction pointer is represented as a machine location, control flow is easily expressed in terms of these primitives. Subroutines and unconditional jumps amount to adding values to the instruction pointer and perhaps manipulating the stack. A conditional jump is slightly more tricky in that some boolean value must be multiplied by an offset before it is added to the instruction pointer.

Thus, the simplicity and regularity of the instruction set allows a single ADD mini-instruction to be active simultaneously in add, subtract, compare, boolean and jump instructions.

Computational functions that cannot be expressed using the primitives discussed so far are handled by triggering an "exception". An exception is a mini-instruction which causes the interpreter to suspend the cycle so as to execute a command stored in the excep location of the processor that signaled the exception. Multiple exceptions occurring in the same cycle are serialized. Input, output, and expensive mathematical operations such as logarithms are supported through the EXCEP mini-instruction (a summary of available exceptions can be found in Table 3).

## 3.4 Computation Optimizations

There are a few simple techniques which can be applied to make the virtual MIMD cycle even faster.

On an architecture in which indirection is slow, the flexibility achieved in our four-indirection model can be accomplished with only two indirections per cycle: one for either the first or second operand and one for either the first operand or the result pointer.

Multiplication and division operations, around ten times slower than add or subtract, are not always needed in their full generality. Our Milan implementation allows division to be replaced with a bit shift command (allowing division and multiplication by powers of 2) and multiplication to be replaced with multiplication only by 1, 0, and -1 (permitting the use of multiplication for absolute value and conditional jumps).

In programs which do require generalized multiplication or division, a test can be done called a *global-or* to quickly determine if any processors need to do multiplication or division in the cycle. If the operations are not needed, they are not performed. Similarly, other time-consuming instructions are checked with a global-or before execution to determine whether any processors need to perform them. In fact, as the Milan program is running via interpreter, the interpreter can analyze the program code at load-time and never attempt to execute an unused mini-instruction.

We have also considered a variety of schemes for speeding up exceptions. For instance, by counting the number of logarithm exceptions, the interpreter can choose to use the front end to handle each of these serially, or use the Connection Machine's own logarithm instruction to serve all of the calls in parallel. Each operation has its own threshold which determines when it is more efficient to perform in parallel.

In practice, we found that eliminating generalized multiplication and division wherever possible had the most significant effect on the runtime of our implementation. The other optimizations did not give improvement in the programs we tested. This could be a function of the small problems we used in testing.

## 3.5 Communication Mini-Instructions

In order to limit the amount of serialization necessary in communication, we allow only one publically readable and writable location on each processor. We call this the *result port* as it is where a processor stores the results it computes for other processors to read. Since this value may have different meanings to different processors, an additional *tag value* serves as a description; it is up to the Milan programmer to choose a consistent tagging protocol.

A processor attempting to read this value has the option of having the read *block* (fail to terminate) until the tags set in the remote processor match a value set locally.

Parallelism in the Milan model comes from processor allocation. The `ALLOC` mini-instruction provides for the dynamic allocation of subordinate processors. An efficient (only 6 global operations) allocation algorithm is used to allow an arbitrary number of processors to allocate an arbitrary number of free processors quickly.

In Milan, allocation serves the function of creating new distributed objects. Each allocated processor computes an element of the object. Allocations can be nested, permitting arbitrarily complex structures to be created.

A processor is allocated by having its `status` location set to 1. The allocation algorithm uses the status bit to enumerate the free processors for allocation. Like multiplication and division, allocation is very time consuming. The allocation procedure is never called unless there are processors wishing to allocate.

Almost as important to the overall performance of the system is processor deallocation. The decision as to when a processor should deallocate is crucial in Milan since asynchrony results in reclamation nondeterminism. In the current model, we deallocate as part of the reduction operation (that is, after finding the sum or maximum of values in a group of processors). In the future, high-level scope analysis can be used to help determine when a processor can be reclaimed.[3]

Another important set of communication operations are the scans (also known as parallel-prefix operators). A *scan* is a function defined over the members of a particular group. For instance, an "add scan" returns the sum of the first $i$ elements of the group to element $i$. Scans for addition, replication, and maximization are accessible as calls directly to the hardware. These calls are special in that they are defined only from within an allocated group (otherwise it makes no sense to speak of the the $i$th element of a group). Generalized user-defined scans can easily be created from the Milan primitives. These, however, run much more slowly.

To use the hardware of the Connection Machine most efficiently, our Milan implementation introduces extra synchrony not required by the abstract model. In our implementation, a scan exception will not complete in any processor in a group until all processors in that group have signaled the scan. The synchronization is achieved by having each group count the number of processors which have reached the synchronization point. When all have, a segmented scan is initiated.

We have also made the observation that a copy-scan (that is, a scan to replicate values across a group) can be expressed in terms of either an add-scan or a max-scan. Hence, it is rarely necessary to do more than one scan in a cycle, since all requested add-scans in the machine (or likewise all max-scans) can be performed in a single hardware operation.

## 3.6 Communication Optimizations

Since communication is slow but any number of processors can communicate at the same time with no added overhead, a *subway optimization* can be used. The

[3] Note that the problem of determining exactly when a value is no longer needed is incomputable. Otherwise, a program could be written which asks "Will I use this value again?" and does the opposite.

subway optimization is to make processors wait before actually communicating. Then, every 5 or 10 cycles, all waiting processors communicate. We have not found this to significantly effect runtime in the simple problems we tried. Programs with complex patterns of communication would probably show greater improvement.

A problem with the current model is that programming using only one port for interprocessor communication can be cumbersome. If supported efficiently enough by the machine, the model can easily be adapted to allow access to remote memory locations. This change necessitates the addition of some sort of status tag for each memory location. This tag would be used to determine whether or not a piece of data is ready to be used by remote processors.

Another potential difficulty with the model is that allocation can possibly take linear time with respect to the number of processors. This may occur when there has been a large amount of allocation and deallocation resulting in excessive fragmentation. The *rooming* scheme (similar to the *paging* scheme used to control fragmentation in memory allocation) is a matter of making fixed sized chunks of $d$ processors which are always allocated together. Allocating a group of size $n$ amounts to allocating $\frac{n}{d}$ chunks from anywhere within the machine. The overhead for this optimization seems to exceed the gain in a system of less than ten thousand processors.

## 3.7 Milan Summary

The Milan cycle is summarized in Table 2. Table 3 contains a list of the available exception operations.

# 4 Data-Parallel Languages

Milan could not be useful without some high-level language interface. In this section, we review seven common data-parallel languages and classify them as to whether or not they could be used asynchronously.

An important observation to make in the comparison of synchronous and asynchronous models of computation is that they only differ in their interpretation of *conditionals*.[4] A program with no conditional statements will run the same in synchronous and asynchronous modes. Therefore the ability of a data-parallel language to execute asynchronously depends on its semantics for the conditional. As a test-bed for conditionals, consider dividing even numbers in a vector by 2 and multiplying odd numbers by 3 and adding one.

[4] The term "conditional" is meant to be interpreted broadly as any branch in control-flow.

| Instruction bit | Action |
|---|---|
| unconditional | The instruction, res, op1, and op2 are loaded from the location indicated by ip. |
| IND1 | op1 gets the value stored in the location indicated by op1. |
| IND2 | op2 gets the value stored in the location indicated by op2. |
| NEGATE | op1 gets minus op1. |
| ADD | op1 gets op1 plus op2. |
| REMOTEREAD | No action. |
| IND1B | If REMOTEREAD is set, op1 gets the value on the read port of processor op2, blocking if BLOCK is set. Otherwise, op1 gets the value stored in the location indicated by op1. |
| BLOCK | No action. |
| SIGNUM | op1 gets the signum of op1. |
| DIVIDE | op1 gets the integer quotient of op1 and op2. |
| MULTIPLY | op1 gets op1 times op2. |
| ALLOCATE | op1 gets a pointer to the lead processor of a group of size op2 in which each of those processors begin execution at op1. |
| EXCEP | op1 gets the value of executing exception excep on arguments op1 and op2. |
| INDRES | res gets the value stored in the location indicated by res. |
| INDEX | res gets res plus op2. |
| REMOTEWRITE | No action. |
| READY | Unused. |
| unconditional | op1 is stored into the memory location indicated by res. If REMOTEWRITE is set, the destination processor is op2. ip is incremented. |

Table 2: A summary of the Milan interpreter cycle.

| Math | Logical | Send with... | Scans | Other |
|---|---|---|---|---|
| logarithm | logical and | add | add | input |
| logarithm base 10 | logical or | logical and | copy | output |
| exponential | logical xor | logical or | maximum | set random |
| square root | logical equivalence | logical xor | | random |
| power | | maximum | | start clock |
| sine | | minimum | | split clock |
| cosine | | overwrite | | break |
| tangent | | unsigned maximum | | |
| | | unsigned minimum | | |

Table 3: A list of available exception operations in Milan.

Neither APL [8], nor Paris [9] (PARallel Instruction Set—the machine language of the Connection Machine) have any notion of a parallel conditional. In APL, our example function can be implemented by computing both the case for odd and the case for even and merging the two results together (using addition, for example).

Paris has a slightly more elegant method in that processors in the odd case can be turned off while processors in the even case are computing and vice versa. Nonetheless, both languages throw away the tiny amount of asynchronousness possible.

The Connection Machine's high-level languages (*Lisp [10], C* [11], and CM-Lisp [12]) support a variety of parallel conditionals. In these languages, our example function can be expressed straight-forwardly as a standard conditional expression. However, this syntax is really only an illusion of asynchronousness. Due to each of these languages' dependence on side-effects, their designers chose to define the semantics of conditionals to be the same as if the clauses were computed sequentially. In other words, these languages express no more in the way of asynchronous behavior than APL or Paris.

Paralation Lisp [13], and Crystal [14] do not allow side-effects in parallel conditionals; Crystal because it is a functional language and Paralation Lisp to avoid the need for resynchronization. In Crystal, computing our example expression on all elements of $A$ can be written (and executed) as:

```
[ x in A | << isodd(x) -> 3*x+1, else -> x/2, >> ],
```

This means that both languages allow for the execution of conditional branches in parallel, that is, they support an asynchronous interpretation. Thus, there are at least two existing data-parallel languages that could serve as a high-level language for Milan.

# 5 Arguments Against Synchronous Data-Parallelism

There are supercomputers and established languages for doing synchronous data-parallel computation. Asynchronous data-parallelism requires writing programs in less common languages to run on non-existent machines or bearing the costs of substantial interpreter overhead. In this light, it is reasonable to ask what is wrong with the synchronous model.

We claim that programs written for synchronous machines can be slow and unreadable. Programmers must do their best to factor and merge time-consuming code from conditionals. This leads to semi-optimized programs which are difficult to read and understand.

## 5.1 Factoring

To limit the time spent sequentializing conditional branches, complex expressions must be carefully "factored out" of the conditional much as an optimizing compiler removes expensive operations from an inner loop.

Consider, for instance, the Crystal function $w(n)$ defined as[5].

```
w(n) = << n = 1     -> 0,
         isodd(n) -> 1 + w( 3*n + 1 ),
         else     -> 1 + w( n/2 ) >>,
```

If this function were translated in the obvious way to run on the Connection Machine, it would be very inefficient. The recursive calls are in the conditional branches and therefore serialized. Stack space is quickly depleted and the function runs very slowly.

Removing the recursive call from the conditional, like this:

```
w(n) = << n = 1 -> 0,
         else  -> 1 + w ( << isodd(n) -> 3*n + 1,
                             else     -> n/2) >>, >>,
```

results in a program which can be made to execute with no stack space and time proportional to whichever $n$ takes the longest to reach 1. The reason for this profound difference is that in the first case, the slow recursive call is made twice at

[5] It is an open problem in number theory whether or not this function terminates for all $n$. Quickly evaluating this function for many $n$ might help theorists detect patterns that would not have been obvious in a few examples. Hence, parallel execution might be of great assistance.

each level of recursion (once in each branch of the conditional) while in the second case, the call is made once at each level.

The benefits of performing this factorization are astounding. Factoring potentially reduces the parallel runtime of the program from $O(2^x)$ to $O(x)$ (where $x$ is the maximum value of $w(n)$ being computed in parallel). In terms of real computing time, a factored version of $w$ (written in Paris) running on the numbers 1 to 4000 was over 400 times faster than the corresponding unfactored program. Unfortunately, it is not always obvious to the compiler—or the programmer—how to perform such a factorization.

## 5.2 Merging

A second commonly-used tool for decreasing the amount of time spent in conditional branches is merging. In this method, similar operations occurring in each conditional branch are reduced to a single operation. This is the primary technique with which Milan is built.

Consider an expression to compute either the quadratic determinant or the area of trapezoid depending on the value of t:

```
<< t = 1 -> b^2-4*a*c, else -> 1/2*h*(b1+b2), >>,
```

This expression can be rewritten `c1*w*x+c2*y*z` by insuring that $c1$, $w$, $x$, $c2$, $y$, and $z$ took on the values 1, $b$, $b$, $-4$, $a$, $c$ respectively in the case when $t = 1$ and $\frac{1}{2}$, $h$, $b1$, $\frac{1}{2}$, $h$, $b2$ otherwise. This may be desirable to speed up a program's execution, but it is not desirable to read.

There are times when merging is even more difficult and more crucial. In the case of Milan, it took us several months to developed a merged instruction set. This is not the kind of task that a compiler should be asked to perform.

## 5.3 Intractable

Some conditionals are so complex that no amount of factoring or merging can prevent costly sequentialization. No compiler or human should have to produce an efficient version of a program which computes (for example) fibonacci, factorial, ackermann, greatest common divisor, a test for primality and the original $w$ function all in different branches of a conditional. Asynchronous execution is the most straight-forward way of reducing the amount of time for such a conditional from the sum of the branches to the maximum of the branches.

# 6 Benchmarks

Although the importance of asynchronous computation is clear, it remains to demonstrate its practicality. To evaluate the efficiency of our interpreter, we wrote Paris, *Lisp and Milan code to compute the standard matrix multiplication algorithm and the $w$ function. We found that the Milan interpreter, with the power of the asynchronous model, performs only marginally slower than Paris on a typical synchronous problem and actually outperforms *Lisp on an asynchronous one.

The matrix multiplication programs generate two sample matrices and then form the product by creating a new matrix whose $(i, j)$ elements are equal to the inner product of row $i$ of the first matrix and column $j$ of the second matrix.

The *Lisp and Paris programs have one major difference from Milan—since no facility exists for creating nested objects in these languages, the programs are written with matrices represented as linear structures. We give statistics for two versions of these programs. The *multiread* version has many communication collisions when reading each value to compute the inner products. The *copy-scan* version reads each value once and replicates the values using the Connection Machine's `copy-scan` primitive[6].

It is important to note here that Milan's ability to easily represent the two-dimensional structure of the matrix comes from dynamic allocation, a function present in PARALATION LISP, Crystal and Milan.

Table 4 gives the running time (in seconds) for each program on $n \times n$ matrices. The *Lisp programs were timed running interpreted on a Vax front-end to be consistent with the execution of Milan.

The *Lisp program to compute the values of $w(n)$ for $n$ from 1 to 445 would not run as stack space was depleted too quickly. Therefore, we extrapolated based on comparisons with the identically coded Paris versions. Table 5 gives timings (in seconds) for the 5 different versions. In the table, data marked with a dagger (†) had to be estimated due to machine limitations.

We have shown that a SIMD problem (such as matrix multiplication) runs approximately 16 times more slowly in Milan than in the native machine language and about 4 times more slowly than a "high-level" language.

For a MIMD problem (like $w$) the ratio is worse when comparing the factored synchronous versions to Milan. This is due to the overhead of handling individual control flow and serialized indirection. This difference will diminish somewhat

[6] In the next software release, this distinction will not be necessary as read-time collisions will be handled efficiently. The result of this change is that the Milan and *multiread* versions will be faster. We estimate that the runtime for Milan on a 15 × 15 matrix will be about 1.77 seconds—only 16 times slower than the native Paris machine language.

| Language | Version | n | Time |
|---|---|---|---|
| Paris | multiread | 5 | 0.03 |
| | | 10 | 0.07 |
| | | 15 | 0.19 |
| | copy-scan | 5 | 0.03 |
| | | 10 | 0.10 |
| | | 15 | 0.11 |
| *Lisp | multiread | 5 | 0.69 |
| | | 10 | 0.69 |
| | | 15 | 0.69 |
| | copy-scan | 5 | 0.79 |
| | | 10 | 0.79 |
| | | 15 | 0.79 |
| Milan | | 5 | 0.87 |
| | | 10 | 1.32 |
| | | 15 | 3.06 |

Table 4: A comparison of 5 programs to compute matrix multiplication.

| Language | Version | Time (secs) |
|---|---|---|
| Paris | unfactored | 12.00 |
| | factored | .25 |
| *Lisp | unfactored | 145.00† |
| | factored | 2.53† |
| Milan | | 36.00 |

Daggers signify estimated values.

Table 5: A comparison of 5 programs to compute the value of $w(n)$ for $n$ from 1 to 445 in parallel.

when indirection is supported through the system software of the Connection Machine (although the hardware is currently in place, the software has not been fully upgraded to take advantage of it).

Most interesting of all is the *Lisp unfactored version of $w$ which is estimated to be four times *slower* than Milan due to the amount of serialization occurring in conditional branches. Milan actually outperformed code which was running very close to the machine.

So our simple benchmarks indicate that the cleaner asynchronous model of data-parallelism can be utilized with an almost acceptable amount of overhead on an existing machine. This overhead will only decrease in coming years with improved hardware and system software.

# 7 Extensions and Directions

Simulation of a fine-grained asynchronous machine can be accomplished with a reasonable amount of efficiency on any synchronous machine that possesses generalized communication and memory indirection. In this section, we discuss fairly straightforward techniques for keeping the virtual cycle time down. There is much more work that can be done to make MIMD simulation even more efficient.

At the hardware and system software level, there are many ways in which the virtual cycle can be accelerated. Direct microcode support for all of the mini-instructions (including allocation) would be beneficial. Further, software support for indirection and message combination and streamlining of the Vax front-end interface (expected in the next release of the Connection Machine's software) would speed things up considerably. The results we have achieved do not depend on the memory indirection of the machine. This is why the runtime of the Milan version of the $w$ function is relatively poor. Even though it should be faster (as it is more of a MIMD problem), memory indirection slows it down considerably.

A more esoteric research direction would be to study ways of compiling asynchronous data-parallel programs for synchronous machines. It would be interesting to explore issues such as getting a compiler to decide between MIMD simulation and direct SIMD execution for a given problem or to select exactly which set of mini-instructions to include in the main cycle. Further, a model for switching between SIMD and MIMD computation could potentially result in computers faster than any which exist today.[7]

Nonetheless, the model offers immediate uses. The language as currently written supports an arbitrary number of programs running simultaneously on the Connec-

[7] The PASM machine [4] is an attempt to do this at the hardware level.

tion Machine (up to the number of available processors). A future version of the interpreter could function as a "daemon", managing resources on the Connection Machine and accepting new programs to run as resources became available. This would allow a dynamic form of multitasking that might make the Connection Machine easier to share.

Other applications lend themselves to immediate implementation. Milan could support higher-level neural net programming with many thousands of highly sophisticated neurons. This naturally leads into an implementation of actors [15] or of interacting agents for natural language and knowledge representation problems. Finally, the machine provides an easy way to implement the various higher-level asynchronous languages that have been proposed, such as CSP [16].

# 8 Conclusions

The common configurations for multiprocessor computers fail to provide the combination of efficiency and expressibility. Asynchronous data-parallelism provides the best features of asynchronous execution and data-parallel programming, and it is easier to use and more efficient than it might at first seem. In the absence of any machines tailored to support this model, we have created a method using existing SIMD hardware to simulate an asynchronous computer. Using such a system could result in programs that do not require confusing source-level optimizations to run efficiently.

In the future, we suggest that more work be done at the hardware level to support this model. This support would be invaluable in creating a system capable of pushing the speed limitations of current parallel machines.

# Acknowledgments

The authors wish to thank their advisors, friends from TMC, Bellcore and the Crystal Group, roommates, and significant others for support and helpful discussions. The authors acknowledge the support of The Office of Naval Research under Contract Number N00014-86-K-0310.

# References

[1] W. Daniel Hillis and Guy L. Steele Jr. "Data Parallel Algorithms," *IEEE Computers*, 29(12):1170-1183, 1986.

[2] Douglas Baldwin, University of Rochester Technical Report, TR 224, August 1987

[3] A.G. Ranade, S.N. Bhatt and S.L. Johnsson, "The Fluent Abstract Machine," Proceedings of the 5th MIT Conference on Advanced Research in VLSI, March 1988.

[4] A Partitionable SIMD/MIMD machine.

[5] The NYU Ultracomputer Project.

[6] W. D. Hillis. *The Connection Machine,* MIT Press, 1985.

[7] "Connection Machine Model CM-2 Technical Summary," Thinking Machines Technical Report HA87-4, April 1987.

[8] Kenneth E. Iverson. *A Programming Language*, John Wiley and Sons, New York, 1962.

[9] "Connection Machine Parallel Instruction Set (Paris), The C Interface, Version 4.0", Thinking Machines Corporation, 1987.

[10] "The Essential ∗Lisp Manual. Release 1, Revision 7." Technical Report, Thinking Machines Corporation, July 1986.

[11] John R. Rose and Guy L. Steele Jr. "C∗: An extended C Language for Data Parallel Programming," Technical Report PL87-5, Thinking Machines Corporation, April 1987.

[12] Guy L. Steele Jr. and W. Daniel Hillis. "Connection Machine Lisp: fine-grained parallel symbolic processing," In *Proceedings of the 1986 Symposium on Lisp and Functional Programming,* pages 279-297, 1986.

[13] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*, MIT Press, Cambridge, Massachusetts, 1988.

[14] M. C. Chen. "Very-high-level parallel programming in Crystal," In *The Proceedings of the Hypercube Microprocessors Conf. Knoxville, TN,* September 1986.

[15] Gul Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, London, England, 1986.

[16] C. A. R. Hoare. "Communicating sequential processes," *Communication of ACM,* 21(8):666-677, 1978.