The Virtual-Time Data-Parallel Machine

Shioupyn Shen and Leonard Kleinrock

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024-1596

Abstract

We propose the "Virtual-Time Data-Parallel Machine" to execute SIMD (Single Instruction Multiple Data) programs *asynchronously*. We first illustrate how asynchronous execution is more efficient than synchronous execution. For a simple model, we show that asynchronous execution outperforms synchronous execution roughly by a factor of $(\ln N)$, where N is the number of processors in the system. We then explore how to execute SIMD programs asynchronously without violating the SIMD semantics. We design a *FIFO priority cache*, one for each processing element, to record the recent history of all variables. The cache, which is stacked between the processor and the memory, supports asynchronous execution in hardware efficiently *and* preserves the SIMD semantics of the software transparently. Analysis and simulation results indicate that the Virtual-Time Data-Parallel Machine can achieve *linear speed-up* for computation intensive data-parallel programs when the number of processors is large.

1 Introduction

For the past twenty years, solid state technology has been much more successful in reducing the cost of VLSI chips than in increasing the peak speed of ECL circuits. As a direct result, the architectural superiority of supercomputers is vanishing because we can easily implement most of the advanced features of supercomputers on a single chip[1]. The performance gap between the fastest processor (in terms of MIPS) and the most cost-effective processor (in terms of MIPS/$) is diminishing rapidly. In the future, the key to supercomputing will not be the high speed of a single processor; instead, it will be the high degree of parallelism.

The difficulties of parallel processing are two-fold. The first problem is that the computational model is hard to use (for asynchronous execution) and the second problem is that the hardware efficiency is poor (for synchronous execution). We propose the "Virtual-Time Data-Parallel Machine" to solve *both* problems at once. The concept of this machine is derived from "The Connection Machine" [4] and "Virtual Time" [6].

The Connection Machine introduced the data-parallel computational model [5]. The SIMD semantics of the *data-parallel* model make it easy to develop parallel programs and make it capable of expressing fine-grain parallelism. Though the Connection Machine achieves significant speed-up for a large number of processors, hardware efficiency may be poor because of its requirement for synchronous execution.

Virtual Time introduced the "Time Warp" synchronization mechanism for parallel discrete event simulation [7]. The *optimistic* approach of Time Warp eliminates unnecessary blocking, and therefore makes better use of the hardware. However, it is hard to generalize Virtual Time to other paradigms of parallel processing.

We suggest the use of Time Warp to execute data-parallel programs asynchronously in hopes of exploiting more parallelism and obtaining better efficiency. By performance modeling, we show that the efficiency (i.e., the sustained speed over the raw speed) of the system is asymptotically 40% for a large number of processors; this is a significant improvement over the traditional approach.

The organization of this paper is as follows: Section 2 provides the motivation, Section 3 explores the key concept, Section 4 addresses performance modeling, Section 5 describes the hardware support, Section 6 discusses the extensions, and the last section concludes the paper.

2 Motivation – The Interconnection Network Bottleneck

The data-parallel approach has been very successful in solving or avoiding many of the technical difficulties of parallel processing.

Data-parallel computers would be the obvious champion in the parallel processing arena if the interconnection network were not the bottleneck.

The performance of the data-parallel approach is more sensitive to the interconnection network than that of the other approaches because of its SIMD semantics. Even though all processors start executing the same instruction simultaneously, they seldom finish this instruction together for many reasons. For

[1] There are approximately three million transistors in an Intel 80586 microprocessor but only two million transistors in a CRAY-1 supercomputer.

0-8186-2772-7/92 $3.00 © 1992 IEEE

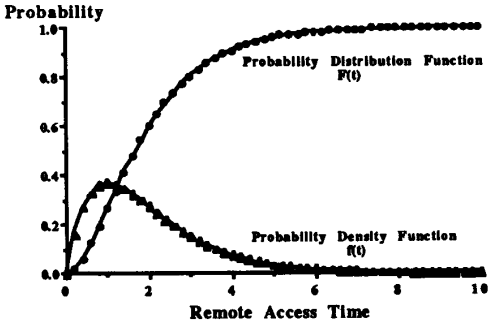

Figure 1: A hypothetical example of remote access time distribution.

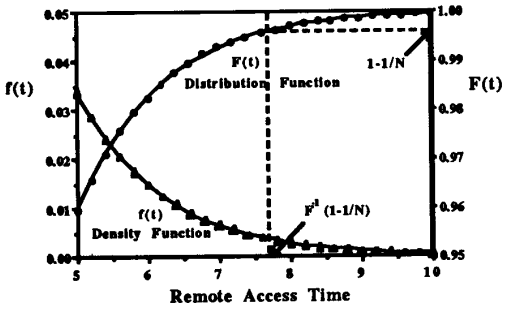

Figure 2: The longest remote access time for N processors is $F^{-1}(1-\frac{1}{N})$, which is mainly determined by the long tiny tail of the probability density function $f(t)$.

example, the access of remote operands (i.e., variables on other processing elements) may take vastly different amounts of time due to network contention and blocking. As a result, the execution time of an instruction varies among the processors.

Figure 1 shows a hypothetical example of the remote access time distribution, where $f(t)$ and $F(t)$ are the probability density function and probability distribution function of the remote access time, respectively. Though the remote access time distributions for various interconnection networks are different, they have several characteristics in common – they have large mean and variance, and more importantly, their probability density functions have long, tiny tails. The long tiny tail has little influence on the mean remote access time because it is so tiny. However, it is the long (though tiny) tail that drives the performance down.

The synchronous execution of SIMD programs forces a processor which finishes the instruction early wait until all processors finish this instruction. Therefore, what really counts is the longest execution time (in other words, the worst case in remote access time) across all processors. The maximum value of N independent samples is approximately $F^{-1}(1-\frac{1}{N})$, where $F^{-1}(t)$ is the inverse function of $F(t)$. For large N, the above term is determined by the long, tiny tail of $f(t)$ as shown in Fig. 2.

In our experience and that of the others, the critical bottleneck of data-parallel computing is the interconnection network. Even though the bandwidth of the interconnection network is large, we cannot justify the cost of providing sufficient bandwidth to reduce the maximum remote access time for random communication patterns. We would like to know how good the performance will be if we smooth out the variation of the remote access time. If the performance is very good, we also would like to figure out how to do it at acceptable cost.

First, we show that the system can achieve linear speed-up (constant efficiency) for a large number of processors. Second, we propose a very cost-effective solution (asynchronous execution) to reduce the susceptibility to the variation of remote access time without modifying the interconnection network. Our approach is to attach minimal hardware support to every processing element to "neutralize" the network hazards, instead of resorting to an expensive "upgrade" of the interconnection network. We can achieve roughly the same performance as if the remote access time is constant but with *twice* the mean. This approach trades the variance for a larger mean. The above trade-off is favorable because the major bottleneck is in the variance instead of the mean, especially for systems with a large number of processors.

In addition to the remote operand fetch, the actual computation of the instruction sometimes introduces large variations into the instruction execution time as well. Conditional enabling/disabling is a common practice in data-parallel programming. Even though it takes constant amount of time for the enabled processors to execute the instruction, collectively speaking, the instruction execution time varies because it takes no time for the disabled processors to skip the instruction. If the disabling probability is high[2], then the execution time varies a lot. In this paper, we use the generic term "instruction execution time", which may refer to either the remote access time or the computation time or both.

[2]For example, a tree-reduction operation [5] of size N takes $(\log_2 N)$ iterations for a total of $(N-1)$ operations. The disabling probability is as high as $(1-\frac{1}{\log_2 N})$.

3 The Key Concept – Asynchronous SIMD

The Connection Machine is a typical example of the traditional data-parallel machine, which has the following characteristics – (i) SIMD, (ii) distributed memory, (iii) massive parallelism, and (iv) programmable interconnection. We are perfectly happy with these properties except the first one – SIMD, or more precisely, synchronous execution of SIMD programs. The inefficiency of synchronous execution comes from unnecessary blocking. Processors that finish the current instruction early are blocked until all processors finish this instruction, even though the operands needed to execute the next instruction may be available. As we know, synchronous execution is a direct way to enforce the "SIMD semantics", but what really matters is the SIMD semantics itself, rather than the synchronous execution.

SIMD semantics is in fact a kind of causality constraint, which is explained as follows. The execution of the i-th instruction (an event "scheduled" at "simulation" time i) depends on the execution of the (i-1)-th instruction (an event scheduled at simulation time i-1). The sequence count of the instruction stream is analogous to the simulation time, which specifies when an event *should* happen, in contrast to "real-time" when the event *does* happen. The data-dependency constraint of the SIMD semantics is thus equivalent to the causality constraint of parallel discrete event simulation (PDES).

The synchronous execution of SIMD programs is essentially the time-stepped execution of PDES, which is considered an inefficient implementation of PDES. On the other hand, the optimistic approach of Virtual Time employs periodic state-saving so that processors have more freedom to go ahead instead of being blocked unnecessarily. The Virtual-Time Data-Parallel Machine takes a similar (but not identical) approach – the execution of the next instruction can proceed independently of the progress on other processors as long as its own data dependencies are satisfied and its current state is properly saved.

Figure 3 illustrates how asynchronous execution of SIMD programs is far more efficient than the conventional synchronous execution. In a task graph, nodes correspond to tasks (instructions) and links correspond to causality constraints (data dependencies). Figure 3.a shows the intrinsic data dependencies of an example program, which ignores all artifacts due to the execution model. When synchronous execution is enforced, it is equivalent to adding more links to the task graph such that every task depends on all the tasks one row above it. Figure 3.b shows the large number of additional data dependencies of the program caused by the requirements of the synchronous execution model.

We know that adding/removing links to a task graph decreases/increases the parallelism of the task graph, respectively. The Virtual-Time Data-Parallel Machine promotes asynchronous execution by removing those extra links associated with synchronous execution (i.e., to achieve better performance) while preserving the original data dependencies (i.e., without sacrificing the semantics).

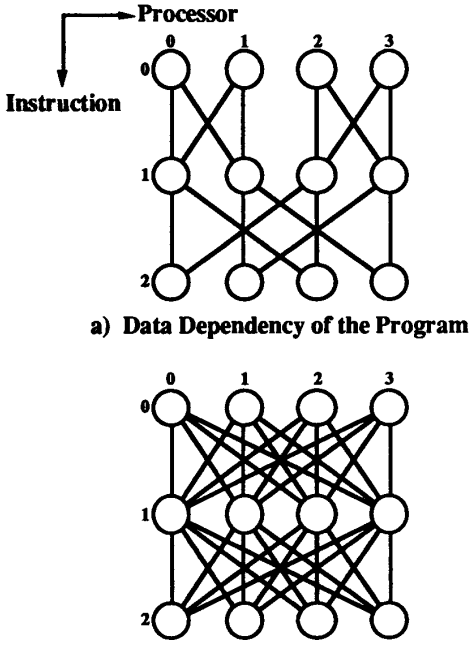

Figure 3: The task graph representation of data dependencies.

4 Performance Modeling – Partial Synchronization

A simple model of the Virtual-Time Data-Parallel Machine is the "partial synchronization" model [1], in contrast to "total synchronization" model (i.e., barrier synchronization). The model is as follows. The SIMD machine consists of N homogeneous processors, i.e., every processor has the same processing power and executes the same instruction stream such that the behavior of every processor is *statistically* equivalent. The task graph (Fig. 4) is an ∞ by N matrix, one column for each processor. Each task depends on a set of tasks one row above it. Let the size of this set be A, which is the number of immediate ancestors of this task. If $A = N$ for all tasks, then it is total synchronization; otherwise, it is partial synchronization. We are interested in the case where A is a small number[3].

[3]If the tasks correspond to instructions, then the value of A is analogous to the number of operands of an assembly instruction, which is usually small.

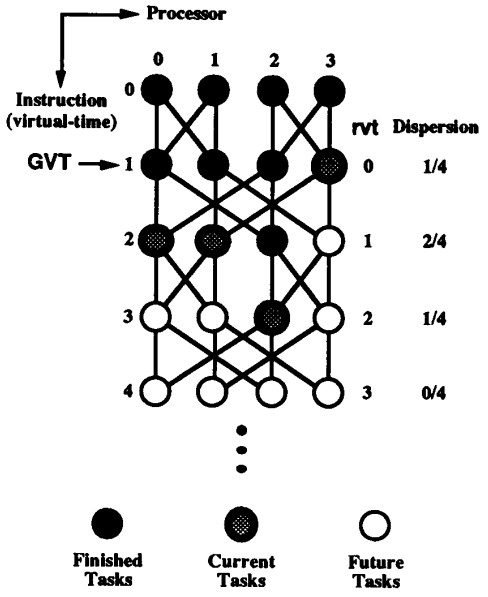

Figure 4: A snapshot of a task graph in execution.

For partial synchronization, the instructions in execution spread out in virtual-time[4]. However, this dispersion is confined due to the data dependency constraints. In order to describe the dynamic behavior of the system, we introduce the following terms (refer to Fig. 4). The *global virtual-time* (GVT) of the system is defined to be the minimum virtual-time of all processors, i.e.,

$$\texttt{GVT} \triangleq \min_{\forall proc.} \{\text{virtual-time}\} \tag{1}$$

The *relative virtual-time* (rvt) of an instruction (or a processor) is defined to be the difference between its virtual-time and the GVT, i.e.,

$$\texttt{rvt} \triangleq \text{virtual-time} - \texttt{GVT} \tag{2}$$

The dispersion function describes how processors scatter in virtual-time, i.e.,

$$\texttt{dispersion}(i) \triangleq \text{Prob}[\texttt{rvt} = i] \tag{3}$$

which can be interpreted as the probability that the rvt of a (tagged) processor is i (averaged over a long period of real-time), or as the distribution of processors in virtual-time (at some real-time instant). Figure 4 shows a snapshot of the system, and Figure 5 illustrates its dynamic behavior.

[4]In this paper, "virtual-time" and "simulation-time" are used interchangeably, as are "instruction" and "task".

We make the following assumptions for the purpose of performance modeling:

1. The execution time of tasks is exponentially distributed.
2. The number of immediate ancestors (A) is exactly two.
3. One ancestor is the preceding task on the same processor, and the other ancestor is uniformly distributed among the tasks in the preceding row.

Though these assumptions are not particularly realistic, they are simple enough to capture some fundamental insight as to how asynchronous execution outperforms synchronous execution.

We are interested in the following performance measures:

Speed-Up:

$$\text{Speed-Up} \triangleq \frac{\text{execution time with a } \textit{single} \text{ processor}}{\text{execution time with } N \text{ processors}} \tag{4}$$

Efficiency:

$$\text{Efficiency} = \frac{\text{Speed-Up}}{N} \tag{5}$$

Efficiency-Gain:

$$\text{Efficiency Gain} = \frac{\text{efficiency of } \textit{asynchronous} \text{ execution}}{\text{efficiency of } \textit{synchronous} \text{ execution}} \tag{6}$$

Figure 6 shows the speed-up and efficiency of *synchronous* execution from analysis [2]. The efficiency of synchronous execution drops (to zero) as the number of processors increases. Figure 7 shows the speed-up and efficiency of *asynchronous* execution as obtained from simulation[5]. Analytical results [8] show that the asymptotic efficiency for a large number of processors is approximately 40%. Figure 8 shows that the efficiency gain is proportional to the logarithm of the number of processors. From this figure we see the motivation for considering asynchronous execution.

We now address the *scalability* of the Virtual-Time Data-Parallel Machine. The traditional definition of scalability is with respect to the speed-up of running the *same* program on an increasing number of processors. This definition is not directly applicable to data-parallel machines where the number of processors is on the same order as the intrinsic parallelism of the program. When the number of processors increases, the problem size must increase *proportionally* so that the intrinsic parallelism increases proportionally as well. If a system scales up well, the execution time is relatively constant.

[5]Analyses are also available [8] for an upper-bound, a lower bound, and an approximation to the efficiency of asynchronous execution.

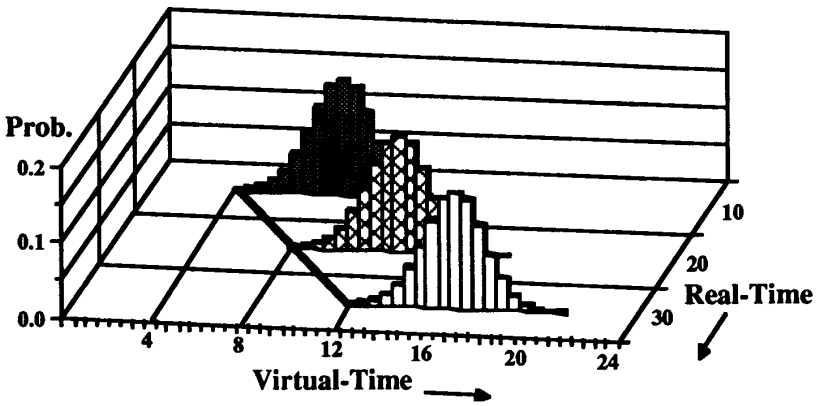

Figure 5: The distribution of processors in virtual-time at several real-time instants. $GVT(t)$, global virtual-time as a function of real-time, summarizes the progress of program execution at real-time t. In this figure, $GVT(10) = 4$, $GVT(20) = 8$, and $GVT(30) = 12$.

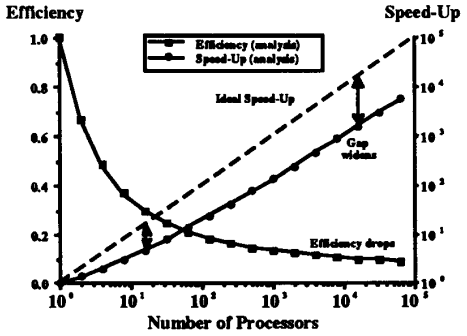

Figure 6: Speed-up and efficiency of synchronous execution.

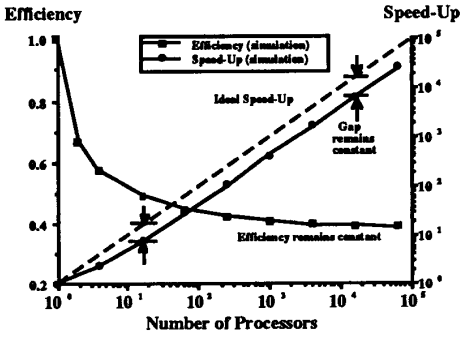

Figure 7: Speed-up and efficiency of asynchronous execution.

An architecture simulator has been developed to run "real" data-parallel programs on the Virtual-Time Data-Parallel Machine to illustrate its superb scalability. The main assumption[6] made in the simulator is the randomness in instruction execution time (in fact, the remote access time). Figure 9 shows the time required to solve a Laplace's equation asynchronously versus synchronously. This diagram reveals that asynchronous execution scales up well as shown by the almost constant execution time, while the execution time for synchronous execution increases. The above argument does *not* mean that asynchronous execution is more favorable than synchronous execution in solving partial differential equations since, had the instruction execution time been constant, as if, with such equations, then the synchronous and asynchronous approaches would have behaved similarly.. However, it indicates that asynchronous execution is capable of smoothing out the variations of instruction execution time.

[6] We also make other assumptions on the number of cycles for integer and floating-point operations. As long as the number of cycles for computation is less than that for communication, the simulation results are not sensitive to these assumptions.

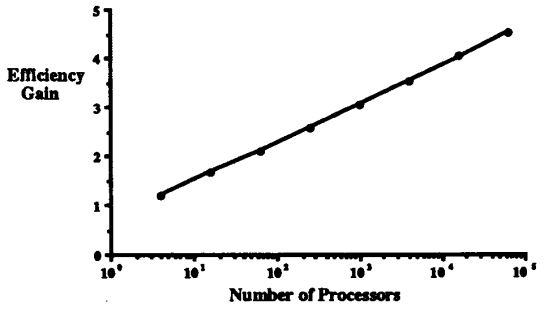

Figure 8: The efficiency gain – asynchronous over synchronous.

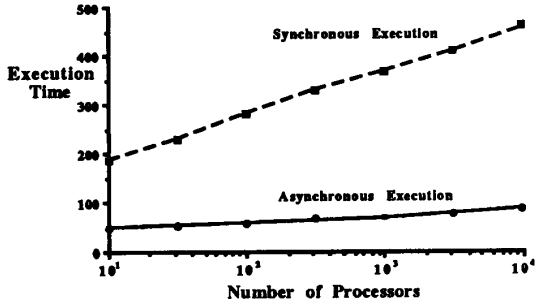

Figure 9: The execution time of a data-parallel program when the number of processors increases with the problem size.

5 Hardware Support – The FIFO Priority Cache

During asynchronous execution, processors are allowed to spread out at different virtual-times. Consider the case in which one processor at a smaller virtual-time (say, i) sends a memory request[7] to another processor at a larger virtual-time (j, where $j > i$). The time (virtual-time) of the request is *current* to the former processor but *previous* to the latter processor. The requested value has actually been generated in a previous instruction (before i) on the latter processor, and is subject to being overwritten by instructions between i and (j-1). In order to prevent overwriting useful data, every processor must save all previous values of its variables (i.e., the memory "history") back to GVT(since no processor has a virtual-time earlier than GVT, no values earlier than GVTwill be requested in the future). For practical reasons, there is a physical limit on the size of the memory history, say K. If a processor goes so fast that it is K instructions ahead of GVT, it must be temporarily suspended because it has used up its memory history. Analysis [8] shows that if K is greater than ($\ln N$), the above situation rarely occurs and the performance hardly degrades due to the limited size of memory history.[8]

Memory history is so important and so frequently used that it deserves special hardware support. We have designed a "FIFO priority cache" (which implements the *incremental backup* algorithm) for the memory history (one cache per processor) with the following characteristics.

FIFO Queueing: Write requests are not executed immediately; instead, they are pushed into a FIFO queue (of size K). If the queue is full, the oldest pending write request (i.e., the one with the earliest virtual-time) is popped out of the FIFO queue and then sent to the main memory (i.e., to update the main memory).

Associative Search: For every read request whose virtual-time is less than or equal to the program counter of the processor, we conduct an *associative* search (like a cache memory) in the queue for hits, where a hit is any pending write request in the queue with matching address and earlier virtual-time.

Priority Arbitration: If there is more than one hit, we choose the latest hit, i.e., the one with the largest virtual-time. Priority arbitration (e.g., choosing the latest hit) can be implemented by a priority encoder/decoder pair. If there is no hit (i.e., a cache miss), then we consult the main memory because the requested value is stored in the main memory.

Related research on the space-time memory can be found in [3] and [8].

Synchronous SIMD machines can hardly benefit from the cache memory because a cache miss for one processor is aggravated to a cache miss for the whole system. Asynchronous execution allows the system to take full advantage of the cache memory technology to resolve the speed discrepancy between the CPU and the main memory. A small FIFO priority cache not only supports the memory history, but also accelerates memory references.

6 Extensions – Two-Phase Write

Even though we can smooth out the variations of the remote access time, the interconnection network may

[7] Memory requests are time-stamped to unambiguously specify the requested values.

[8] What a coincidence! ($\ln N$) happens to be the performance gain as well.

still be the performance bottleneck (however, the bottleneck is less severe than before). The problem resides in the mismatch of *local* access time and *remote* access time. The technology is such that the speed of the processor (and the memory) has improved quickly but the speed (in terms of latency instead of throughput) of the interconnection network has improved at a much slower rate. As a result, the remote access time is tens to one hundred times larger than the local access time, and the mismatch will become even worse in the future. The processors spend most of their time waiting for remote operands because the actual computation or the local references can be done in a flash. If we could pipeline the instruction execution, the *throughput* of the system would improve dramatically. However, the variation of remote access time makes pipelining difficult if not impossible.

Traditional pipelining is "synchronous" pipelining in the sense that timing information is known in advance so that a reservation table synchronizes the resource allocation. Data-flow is "asynchronous" pipelining in the sense that only the data-dependency counts and timing information is either unknown or irrelevant. Since this paper is promoting the *asynchronity*, whenever the "synchronous" approach fails, try its "asynchronous" counterpart.

The sequential semantics of SIMD programs adds an implicit ancestor to every instruction on every processor – the preceding instruction on each of these processors. However, we observe that the result of an instruction may not be used immediately by the next instruction. If the current instruction is blocked (e.g., waiting for a remote operand), the execution of the next instruction can proceed without waiting for the current instruction to finish. Before the next instruction starts executing, the processor must schedule the execution of the current instruction and invalidate the variables that may be modified by the current instruction. Such a problem was addressed many years ago by the Tomasulo algorithm [9]. This algorithm, used by the floating point unit of the IBM 360/91, converts sequential computation into data-flow computation within a small sliding window of instructions.

The main idea is to separate a `write` operation of a variable into two phases – the *logical* write and the *physical* write. The logical write is executed first before the content of the write operation is available; it invalidates the variable and assigns a unique identifier to the content of the write operation. From then on, all read requests to that variable (before the variable is overwritten) are transformed to waiting for that identifier. The next instruction can proceed after the logical write without waiting for the physical write. The physical write is executed when the content of the variable becomes available; it is sent to every processor waiting for the corresponding identifier. Once we adopt the two-phase write, then head-of-the-line blocking, which enforces sequential execution, is eliminated; at the same time, the sequential semantics are preserved. Thus we see that the techniques used to compensate for the long pipeline stages of floating point arithmetic units in sequential machine may now be used to compensate for the long (network) delays due to remote access in parallel processing systems.

The two-phase write can be easily implemented in the memory history by adding an extra *busy* bit to every outstanding update. A logical write sets the busy bit to one, representing the fact that an update is taking place, and the content is not available. A physical write resets the busy bit to zero, representing the fact that the data in this update is available. References to a busy update receive the time-stamped address of the update (which serves as the unique identifier), and then get blocked. When a physical write is executed, all references to the matching identifier are unblocked.

With the two-phase write, the Virtual-Time Data-Parallel Machine converts the SIMD computation from control-flow to data-flow (within a small sliding window of neighboring instructions). Data-flow execution *recovers* more threads of execution than control-flow, which increases the concurrency and improves the efficiency of the Virtual-Time Data-Parallel Machine.

7 Conclusions

Long and unpredictable remote access latency is often the performance bottleneck of massively parallel computing. Asynchronous execution of the Virtual-Time Data-Parallel Machine provides one way to relieve this bottleneck. We have proposed some minimal modifications to the architecture of the traditional data-parallel machine (e.g., the CM-2), which converts the way it executes SIMD programs from synchronous to asynchronous. We have provided a basic foundation for the understanding of both *why* and *how* to improve the efficiency of SIMD programs by allowing asynchronous execution.

Asynchronous execution makes the machine more MIMD (Multiple Instruction Multiple Data)-like. It is nevertheless a SIMD machine from the programmer's point of view! A more detailed discussion of these ideas and a comprehensive quantitative analysis of this machine can be found in [8].

Acknowledgments

This research has been supported by the Advanced Research Projects Agency of the Department of Defense under contract MDA 903-87-C0663, Parallel Systems Laboratory.

References

[1] C. S. Chang and R. Nelson, "Bounds on the Speedup and Efficiency of Partial Synchronization in Parallel Processing Systems," Research Report RC 16474, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, January 1991.

[2] Robert E. Felderman and Leonard Kleinrock, "An Upper Bound on the Improvement of Asynchronous versus Synchronous Distributed Processing," *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 22, No. 1, pp. 131–136, January 1990.

[3] Richard M. Fujimoto, "The Virtual Time Machine," *International Symposium on Parallel Algorithms and Architectures*, pp. 199–208, June 1989.

[4] W. Daniel Hillis, *The Connection Machine*, The MIT Press, Cambridge, Massachusetts, 1985.

[5] W. Daniel Hillis and Guy L. Steele, Jr., "Data Parallel Algorithms," *Communications of the ACM*, Vol. 29, No. 12, pp. 1170–1183, December 1986.

[6] David R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404–425, July 1985.

[7] Jayadev Misra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, Vol. 18, No. 1, pp. 39–65, March 1986.

[8] Shioupyn Shen, *The Virtual-Time Data-Parallel Machine*, Ph. D. Dissertation, Computer Science Department, UCLA, 1991.

[9] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal*, pp. 25–33, January 1967.