EVALUATING THE POWER OF THE PARALLEL MASC MODEL USING SIMULATIONS AND REAL-TIME APPLICATIONS

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by

Mingxian Jin

August 2004

Dissertation written by

Mingxian Jin

Approved by

__________________________________, Chair, Doctoral Dissertation Committee

__________________________________, Member, Doctoral Dissertation Committee

____

____

____

Accepted by

__________________________________, Chair, Department of Computer Science

__________________________________, Dean, College of Arts and Sciences

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to deeply thank Dr. Johnnie W. Baker for his continuing enthusiasm and support for my Ph.D. study. He has been the advisor for my research work throughout these years and also the mentor for my professional career development. Without his encouragement and guidance, this dissertation would be impossible.

I would also like to thank my dissertation committee members for their willingness to serve on the committee and their valuable time.

My special thanks will go to many people who have helped me at various stages of this work – Will Meilander, Kenneth Batcher, Robert Walker, Jerry Potter, Arden Ruttan, Sue Peti, and Anne Martyniuk. In particular, Professor Will Meilander has contributed invaluable information for the real-time part of this dissertation. He has read through the part and provided many useful comments.

Finally, I would like to express my thanks to my husband Chengwei Liu and my children Xilin, Kevin, and Joyce for their support and patience so that I can make a dream come true.

Jin, Mingxian, Ph.D., August 2004 COMPUTER SCIENCE

EVALUATING THE POWER OF THE PARALLEL MASC MODEL USING SIMULATIONS AND REAL-TIME APPLICATIONS (145 pages)

Director of Dissertation: Johnnie W. Baker

The Multiple Associative Computing (MASC) model is a generalized associative-style parallel computation model that was developed at Kent State University in 1990s. It provides a complete paradigm that can be used for both special-purpose and general-purpose parallel computation. The primary focus in this research is to evaluate the power of the MASC model and to provide a better understanding of it. This work consists of three parts. The first part is to justify the timing assumptions for the basic associative operations of the MASC model in order to establish a firm theoretical foundation for the MASC model. The associative operations are used extensively in MASC algorithms. The second part creates simulations between the MASC model and well-known parallel enhanced mesh models. The simulations of the enhanced mesh models provide an efficient method for converting algorithms designed for enhanced meshes to the MASC model. The third part involves using the ASC model (i.e. MASC with one instruction stream or an associative SIMD computer) to provide an efficient polynomial time solution to the real-time Air Traffic Control problem. However, this type of real-time problem is currently considered to be unsolvable in polynomial time using a MIMD computer (i.e. a multiprocessor). Since the MIMD model of parallel computers is generally believed to be more powerful than the SIMD model due to the fact of MIMD's

asynchronous execution, the preceding anomaly indicates that this belief needs to be reconsidered.

Chapter 1

Introduction

The Multiple Associative Computing (MASC) model is a generalized associative-style parallel computation model that was developed at Kent State University. Motivated by the STARAN computer built by Goodyear Aerospace in the early of 1970s, the MASC model was formally proposed by J. Potter and J. Baker et. al. in [99] in 1994. Prior to this, an architectural version of the model with one instruction stream named ASC was introduced in [98] in the early of 1990s. But the formal proposal in [99] defines and describes the MASC model systematically, and it especially emphasizes the version of the model with multiple instruction streams. The MASC model applies the concept of associative computing to support not only data parallelism but also control parallelism. Thus it provides a complete paradigm that can be used for special-purpose and general-purpose parallel computation. It is a model that is easy to program, highly scaleable and currently buildable. The MASC model supports efficient algorithms for a wide range of problems and can be supported on a wide variety of computer platforms with a relatively simple programming language. Since MASC is a strictly synchronous model that supports SIMD computation but allows multiple instruction streams, it is also called a multiple SIMD or MSIMD model. Therefore, it is a hybrid data-parallel/control-parallel computation model. In contrast to a number of other parallel models proposed in the past few decades, the MASC model possesses certain associative properties such as constant time broadcasting, constant time global reduction operations, and constant time associative search. These associative properties have made it possible to solve some real-time problems in an especially efficient manner.

In order to further evaluate the power of the MASC model and give a better understanding of this model, we have conducted the extensive research on this model. The work presented in this dissertation consists of three parts. The first part is to justify the timings assumed for the basic associative operations of the MASC model to build a solid fundamental theoretical background for associative properties assumed for MASC. These associative properties have been widely used in MASC algorithm design and complexity analysis. Our detailed timing justification for the MASC basic operations is not only based on their hardware implementation in the STARAN, the architectural ancestor of the MASC model, but is also based on comparison with timings of basic operations for other well-established models such as PRAM and bus-based architectures.

The second part of our work is the simulations between the MASC model and enhanced meshes. Enhanced meshes are attractive parallel computational models that have received a lot of attention from researchers in the past. We create three simulation algorithms to simulate between the MASC model and two classes of enhanced meshes, namely, MMB (Meshes of Multiple Broadcasting) and BRM (Basic Reconfigurable Meshes). In particular, a constant time simulation algorithm using MASC to simulate MMB is developed. Simulations between MASC and RM (Reconfigurable Meshes), a general case of BRM, are also explored. We show that these two models are dissimilar. Neither of them can simulate the other efficiently due to their distinguishing features. Our work in this part provides an effective way to understand the power of the MASC model in terms of the comparative power with the popular enhanced mesh models. In particular, the constant time MASC simulation of MMB enables algorithms designed for this enhanced mesh model to be transferred to the MASC model with the same running time.

The third part of our work involves using MASC with one instruction stream (called ASC) to solve real-time problems. It is shown that an efficient polynomial time solution to a real-time problem for Air Traffic Control can be obtained. However, these type of real-time problems are currently considered unsolvable in polynomial time using a MIMD computer, which is generally believed to be the more powerful type of parallel computers. The existence of a feasible polynomial time solution for the ATC problem using an associative SIMD leads us to identify and analyze some of the most important features of both associative SIMDs and MIMDs with regard to real-time processing. Our work raises serious questions concerning the validity of the general belief that MIMDs are more powerful than SIMDs and can solve any problems as efficiently as SIMDs. Based on this fact, it is suggested that the importance and efficiency of SIMD computation should be reconsidered.

This dissertation is organized into seven chapters. First, Chapter 2 surveys a number of parallel computation models and the related research on simulations and comparisons among them. Chapter 3 gives an overview of the MASC model, its properties, and previous work on the model. Chapter 4 addresses justification of timings of the basic associative operations on the MASC model, based on a theoretical analysis of one possible implementation and comparison with timings of basic operations for other well-established computational models. Chapter 5 presents simulations of the enhanced meshes, e.g., MMB, BRM, and RM models, with the MASC model, and the reverse simulations, respectively. Chapter 6 discusses how to use an ASC, i.e., the MASC with one instruction stream, computer to solve a real-time problem for Air Traffic Control, in polynomial time. Chapter 7 extends our results in Chapter 6 to reevaluate the computation power of SIMD computers by comparing it with that of MIMD computers, in terms of capabilities of solving real-time problems. Chapter 8 gives the final summary and discusses some work planned for the future.

Chapter 2

Parallel Computational Models

A parallel computational model is an abstract description of parallel computers with specific type of architecture and specific properties. Such a model extracts the essential features of these computers and provides a framework for researchers to study these computers independent of any real machines. Using a model, algorithms can be developed and analyzed without involving any of unimportant implementation details related to a particular machine. A parallel model has significant impact on designing both parallel systems and parallel algorithms. Therefore, developing effective models for parallel computation at proper levels of abstraction is fundamental in parallel processing. A good model must correctly reflect essences of the parallel computers while balancing briefness, accuracy, and broad applicability.

A number of models for parallel computation have been proposed and studied in the past two decades. According to different communication schemes between processors, models are usually divided into two categories, shared-memory based and interconnection network based. In the first two sections of this chapter, we give a survey of models to provide a background for the later chapters. A few popular models are emphasized. Also, adding a certain number of buses to a particular model gives another category of parallel models, i.e., bus-based models, which we will discuss in Section 2.3. Then, in Section 2.4, we study three recently proposed parallel models. These models are based on either shared-memory or networks and have been developed for general-purpose parallel computation.

2.1. Shared-Memory Based Models

A shared memory model consists of a collection of processors communicating by reading and writing to locations in a shared memory that is equally accessible by all processors. In some cases, the processors may have local memory that can be used for local storage and computational uses, but often only shared memory is available. Its primary advantages are ease of use and portability of algorithms.

2.1.1. Parallel Random Access Machine (PRAM)

Primary among all parallel models has been the Parallel Random Access Machine (PRAM) model, in which processors execute synchronously or in *lock-step* and communicate by reading and writing to locations in the shared memory. Lock-step means all processors are synchronized in every single step. (Although asynchronous PRAM was also defined later, synchronous PRAM is usually assumed to be the default.) This is contrast with *bulk-synchronous* model in which processors are synchronized in specific step and we will discuss in later sections.

The PRAM model was first proposed as a general-purpose model for parallel computation by Fortune in 1978 [42]. A PRAM consists of p conventional sequential processors, P_1 , P_2 ,...P_p, all of which are connected to a large shared random access memory, M. Each processor has a private, or local, memory for its own computation. All communication among the processors is performed via the shared memory. In one step, each processor can access one memory location or execute a single RAM operation. The processors synchronously execute the same program through the central main control. The processors generally operate on different data but perform the same instructions. Hence, the model is sometimes called *shared-memory single instruction stream, multiple data stream* (SM SIMD) machine. A PRAM time step is defined as consisting of three phases: *read phase*, in which the processor may read from a shared memory cell; *computation phase*, in which the processor does some calculations; and *write phase*, in which the processor may write to a shared memory cell.

Based on restrictions on shared memory access, PRAMs are classified as three categories: EREW PRAM, CREW PRAM and CRCW PRAM, where E is for exclusive, C for concurrent, and R for read, W for write. ERCW PRAM is usually not considered, since a machine with enough power to support concurrent writes should be able to support concurrent reads. Here, we recall three important varieties of CRCW PRAMs, which depend on how concurrent writes are handled. The **common** CRCW PRAM allows concurrent writes only when all processors that attempt write the same value to the same location at the same time. The **arbitrary** CRCW PRAM allows an arbitrary processor to succeed. The **priority** CRCW PRAM allows only the processor with the highest (or lowest) priority to succeed. The priority CRCW PRAM is the most powerful among these variants, and the EREW PRAM is the weakest. Simulations and further relationships between different PRAM models have been explored by researchers. They can been found in [11,52,57].

The PRAM model is an idealized model and draws its power from its simplicity. The model ignores the algorithmic complexity of communication-related issues and allows the algorithm designer to focus on the fundamental computational difficulties of the problem. Until recently, there were few attractive alternatives for the PRAM. As a result, numerous efficient algorithms have been developed for this model [10,11,57]. On the other hand, it has been criticized for years for being too high level and failing to accurately model realistic parallel machines. In particular, the PRAM model has been faulted for completely failing to model memory contention, communication bandwidth limitations, and synchronization cost of parallel processing that are typical for asynchronous computation. PRAM uses synchronous computation and a simplified method for communications that often underestimates the cost and avoids the details of having to specify a particular communication network and to design appropriate routing algorithms for asynchronous network.

One way to avoid these criticisms of PRAMs is to develop methods that allow the execution of PRAM algorithms on more realistic machines. The problem is to simulate a CRCW PRAM on a realistic parallel machine, namely, one with distributed memory and an interconnection network of fixed degree. According to the summary in [52], this central problem can be generally separated into three fundamental sub-problems that can be solved individually. The three sub-problems are the concurrent access problem, the memory management problem, and the routing/interconnection problem. We denote (n, m) PRAM to be a PRAM with n processors and m memory locations. In a (n, m)-CRCW PRAM, a processor may request concurrent access to any of m memory locations. The first problem is to service these multiple concurrent requests correctly and as efficiently as possible on hardware that disallows concurrent access, namely, an EREW PRAM. The solution to the second problem of memory management, is to simulate a (n, m)-EREW PRAM on a machine with M memory modules in which the processors are fully connected, so that any processor can communicate with any other in constant time. This machine is called *Module Parallel Computer* (MPC). It allows us to focus on how to layout memory so that the amount of module contention is minimized for any given set of n requests to be serviced. The solution to the third problem is to specify a fixed-degree interconnection network (referred as a *Bounded Degree Network* (BDN)) and a routing algorithm that will allow servicing of all requests from n PRAM processors with minimum slowdown. This step is used to precisely measure the communication cost for a specific PRAM algorithm. It is assumed that memory management scheme has handled the memory requests already before the router takes control.

The authors of [52] have given the results for each of the above steps both by deterministic simulation (mapping processors deterministically) and by randomized simulation (mapping processors randomly). These results indicate the simulation cost of PRAM on currently more realistic asynchronous parallel machines in a general way.

2.1.2. Variants of PRAMs

Instead of simulating PRAMs on more realistic machines, a considerable amount of effort has been spent to propose and differentiate between variants of PRAMs in the past. In this section, we study several models developed from the original PRAM model.

- **Local-Memory PRAM (LPRAM)**

The *Local-Memory PRAM* (LPRAM) was proposed by Aggarwal in [7]. It focuses on exploring communication complexity of asynchronous PRAMs. Based on a CREW PRAM, it models the communication delay and computation time separately. At each step, the processors do one of two things as follow. In a communication step, a processor can write, or read a word from global shared memory. Or, in a computation step, a processor can perform an operation on at most two words that are present in its local memory. Usually, a communication step is viewed as taking several times longer than a computation step. This communication delay is also called *latency*. Assuming one computation step takes unit time, LPRAM defines a parameter l that specifies the length of time taken by one communication step. The total running time of an algorithm is defined as the sum of the computation and communication times.

The parameter l is the latency for shared memory access. An LPRAM with p processors can be simulated on a butterfly with the same number of processors, with each computation step being simulated in constant time and each communication step requiring O(log p) time; thus here l = O(log p). Similarly, an LPRAM can be simulated on a 2-D mesh with each communication taking time $l = O(\sqrt{p})$. The work results in the equivalence of the LPRAM model with communication cost l = log p and the butterfly model and the equivalence of the LPRAM model with $l = \sqrt{p}$ and the 2-D mesh model.

In [7], the authors present several problems solved on an LPRAM. For example, two $n \times n$ matrices can be multiplied in $O(n^3 / p)$ computation time and $O(n^2 / p^{2/3})$ communication delay using p processors (for $p \leq n^3 / (\log^{3/2} n)$). Also, sorting n words takes $\Theta((n \log n) / p)$ computation steps and $\Theta((n \log n) / (p \log (n / p)))$ communication delay.

Obviously, if a communication step takes time l, then the total running time of an algorithm is $T + l \times C$, where T is the number of computation steps and C is the number of communication steps. Thus, the total running time for the above matrix multiplication is $\Theta(n^3 / p + ln^2 / p^{2/3})$. It is further known that, when $n = \Omega(lp^{1/3})$, the algorithm is computation-bound (i.e., computation time dominates) and will be running more efficiently. Thus, for $l = \log p$, n should be $\Omega(p^{1/3} \log p)$, and for $l = \sqrt{p}$, n should be $\Omega(p^{5/6})$. The n-word sorting problem can be analyzed in a similar manner. These indicate how a problem can be executed efficiently on an LPRAM with communication delay when the input size is different; the value of l dictates how large a problem should be in order to obtain maximum efficiency.

- **Block PRAM (BPRAM)**

The *Block PRAM* was introduced by the same authors that proposed LPRAM in [6]. It is essentially another version of LPRAM since it uses the same parameter l for communication delay. Besides that, it takes into consideration that a processor may read or write a block contiguous locations from or to the shared memory in one step. Such an operation is charged $l + b$ time, where l is the startup time or latency and b is the length of the block. An access to a location of the local memory is assumed to take unit time. Any number of processors may access shared memory simultaneously, but the blocks that are being accessed need to be disjoint.

BPRAM captures spatial and temporal locality of reference for the design of more efficient algorithms. Algorithms for some problems such as matrix transposition and multiplication have been designed to take the reduced cost of block access in order to obtain a better running time.

- **PRAM(*m*)**

PRAM(*m*) is a variant of the PRAM where the shared memory size is limited to *m*. It has been pursued by many researchers in [76,63,2,3,4,125]. Unlike the unbounded shared memory size in the unrestricted PRAM, the small size *m* of the shared memory relative to the number *p* of processors restricts the number of messages that can be concurrently accessed. The reason to develop this model is that many existing parallel machines allow a relatively low level of **c**ommunication **t**hroughput (CT for short) while many parallel algorithms developed appear to need a relatively high level of CT. There appears to be a typical gap between some algorithmic needs and the capability of many computer systems concerning CT. Since increasing CT may be very expensive, it makes sense to use the PRAM(*m*) model to quantitatively analyze CT and how the efficiency of an algorithm can be affected by a limited communication channels.

In [76], a case is studied in which the size *n* of a problem is relatively big compared to the number *p* of processors. With PRAM(*m*), the lower bound of running time for concrete problems such as the list reversal, sorting, finding unique elements, convolution, or universal hashing is established as $\Omega(n/\sqrt{mp})$. Also, a nearly optimal algorithm for the list reversal problem is derived. Since such a problem can be solved in O(1) time on a PRAM with *n* processors and *n* shared memory locations, this indicates that it may be very inefficient to implement a high-CT algorithm on a low-CT parallel system. It may further imply that parallel computers with a low-CT level will be handicapped as general purpose computing systems.

The PRAM(m) model has been compared to other limited communication bandwidth models such as BSP and QSM. Both BSP and QSM will be discussed in Section 2.4 in more details and are two typical general-purpose parallel models with parameters to measure the communication limitation. BSP deals with message passing and barrier synchronization while QSM is focused on shared memory and concerns memory contention. By using an aggregate parameter m as the global communication restriction, QSM(m) is a variant of the QSM model that can only handle at most m global messages in one step [3,63]. It is shown that one step of the CRCW PRAM(m) can be simulated on the QSM(m) in time $O(p/m)$ provided $m = O(p^{1-\varepsilon})$ and $0 < \varepsilon < 1$. Even if every processor in the QSM(m) model is given the entire input in advance, the CRCW PRAM(m) is faster than the QSM(m) by a factor of $\Omega(\frac{p \log m}{m \log p})$. Consider the only difference of latency parameter L between QSM and BSP, this simulation can be transferred to BSP(m). (BSP(m) is defined in the same way as a variant of BSP.)

- **Phase PRAM**

The PRAM assumes that processors execute in synchronously, which causes difficulties in mapping PRAM algorithms onto realistic asynchronous MIMD machines. Supporting a synchronous model such as PRAM on an asynchronous machine is inherently inefficient since the ability of the machine to run asynchronously is not fully exploited and there is a (potentially large) overhead in synchronizing processors as part of each instruction. These observations have resulted in some asynchronous PRAM models being proposed [34,48,90] as variants of the PRAM. Unlike PRAM, the processors of an asynchronous PRAM run asynchronously, i.e., each processor executes its instructions independently of timing of the other processors. There is no global clock.

The *Phase PRAM* is one such model that is defined by Gibbons in [48]. It considers that a computation is a series of global, program-wide phases in which the processors run asynchronously separated by synchronization steps that involve all the processors. The processors cannot begin their next phase until the synchronization step has completed. Consider a phase that is followed by a synchronization step among a set S of processors. The cost of a synchronization step is $B(p)$, a non-increasing function of p where $p = |S|$. The running time of a program is defined to be the maximum over all processors of the completion time for its local program.

This definition leads to different time complexity for some algorithms for Phase PRAM, when compared to their PRAM versions. For example, the prefix sum computation runs on a n EREW PRAM in O(log n) time in binary tree fashion. By inserting a synchronization step after each PRAM read or write step, an EREW Phase PRAM algorithm for prefix sum rums in O(B log n) time. Generally, a PRAM algorithm running in time t using p processors can be simulated by a Phase PRAM algorithm running in time O(Bt) with p/B processors. Also, a Phase PRAM algorithm using p_0 processors and running in time $t+B(p_0)s$, where s is the number of synchronization steps, can be simulated by a Phase PRAM using $p < p_0$ processors in time $O((p_0/p)t + B(p)s)$.

- **XPRAM**

XPRAM is an extension of the PRAM family that was proposed by L. G. Valiant in [123,124]. It is essentially an early version of the BSP model that we shall discuss later. An XPRAM with p processors performs its execution in *supersteps*. In each superstep each processor executes a number of local instructions and sends or receives some messages that implement global read or write instructions. In an execution of a superstep, suppose that processor i performs a_i local operations, sends b_i messages and receives c_i messages from other processors. Assume that each local operation takes unit time, while typically each global operation takes time $g \geq 1$. Let $r_i = g^{-1} a_i + b_i + c_i$. Then if t = $\max\{r_i\}$, the runtime of this superstep is considered to be that of $(\lceil t/L \rceil)L$ standard global operations, or time $(\lceil t/L \rceil)Lg$, where L is the periodicity parameter All the processors know whether a superstep is complete or not at the end of each period of L global operations. Within each such period, however, there is no synchronization among the processors.

A primary goal for XPRAM is to capture communication costs by charging a higher cost for global reads or writes than PRAM so as to make its predicted asynchronous communication cost more realistic than PRAM does. It embodies the principle of *bulk-synchrony*, that is, the processors should be barrier synchronized at regular time intervals. A time interval is defined to be long enough for several packets to be transmitted between a pair of processors.

Simulations of EPRAMs (EREW PRAM) and CPRAMs (CRCW PRAM) with XPRAM are given in [124]. These simulations have been shown to be *optimally efficient*.

This means that there is a fixed constant such that, if an algorithm is running on the simulating model with cost c_1 and is running on the simulated model with cost c_2, the difference between c_1 and c_2 is less than this fixed constant.

- **QRQW PRAM**

The *Queue-Read Queue-Write (QRQW) PRAM* model was proposed by Gibbons et al. in [50]. This model solves the memory access contention issues in the PRAM by adding queues for each of processors as queued-read and queued-write.

There are two QRQW models defined. One is the SIMD-QRQW PRAM, running in the lock-step style on SIMD machines. The other is QRQW PRAM, running in bulk-synchronous on MIMD machines. A single step of a PRAM consists of a read operation, a compute operation, and a write operation. The *maximum contention* of the step is the maximum, over all locations *x*, of the number of processors reading *x* or the number of processors writing *x*. As a boundary case, a step with no reads or writes is defined to have maximum contention of 1. The time cost for a step with maximum contention *k* is *k*. The QRQW-PRAM model consists of a number of processors, each with its own private memory, communicating by reading and writing locations in a shared memory. The processors execute a sequence of synchronous steps, each consists of three substeps: processor *i* reads r_i shared memory locations (read); processor i performs c_i local RAM operations (compute); processor i writes w_i shared-memory locations (write). Concurrent reads and writes to the same location are permitted in a step. In the case of multiple writes to a location *x*, an arbitrary write to *x* succeeds in writing the value in *x* at the end of the step. If its maximum contention is *k*, let $m = \max\{r_i, c_i, w_i\}$, then the time costs for the step is $\max\{m, k\}$. The time of a QRQW PRAM algorithm is the sum of the time costs for its steps.

A p-processor QRQW PRAM algorithm running in time t can be simulated on a pt-processor SIMD-QRQW PRAM in time $O(t)$ [50]. In addition, simulations between other variants of the PRAMs and QRQW PRAM have been discussed. From CRCW PRAM to QRQW PRAM, there exists $\Omega(\log n)$ time cost; and from SIMD-QRQW PRAM, there exists $\Omega(\sqrt{\log n})$ time cost. Hence, relationships between these models are established as follow, EREW PRAM ≤ SIMD-QRQW PRAM ≤ QRQW PRAM ≤ CRCW PRAM where "≤ " is read as "is weaker than".

Comparison and simulations of QRQW PRAM to BSP (discussed in Section 2.4.1) have been given in [50,3]. A p-processor QRQW PRAM algorithm (or SIMD-QRQW PRAM algorithm) running in time t, where t is polynomial in p, can be simulated on a $(p/\log p)$-component BSP(g, L) model in $(t\log p)$ time with high probability, when g is a constant and L is $\Theta(\log p)$.

- **Broadcasting with Selective Reduction (BSR)**

Another shared memory model is *Broadcasting with Selective Reduction* (BSR), which is also basically an extension of the PRAM. It was first proposed by Akl in [8,9,11]. It consists of N processors numbered 1, 2, …, N, sharing M memory locations $u_1, u_2, \ldots, u_M$. The model has all of the features of CRCW PRAM. In particular, during the execution of an algorithm, several processors may read from or write into the same memory location simultaneously, such that each processor accesses at most one location. However, an additional type of memory access is permitted in BSR, namely, a

BROADCAST instruction, by means of which *all* processors may access *all* memory locations at the same time for the purpose of writing. At each memory location, a subset of the incoming broadcast data is selected and reduced to one value (using an appropriate reduction operator such as minimization) and this value is finally stored in the memory location. The selection process is carried out as follows. Along with each data item d_i, a tag t_i is also broadcast, while each memory location u_i is associated with a limit l_j. A selection rule ξ (e.g., <) is used to test the condition $t_i \ \xi \ l_j$: if the latter is true, then d_i is selected for reduction; otherwise, d_j is rejected. This is done simultaneously for all broadcast data d_i with $1 \leq i \leq N$, and all memory locations u_j with $1 \leq j \leq M$.

The BSR model is a PRAM augmented with an additional form of concurrent access to shared memory. The feature that distinguishes BSR from other models is the so-called BROADCAST instruction that is executed in three constant-time phases:

1. *Broadcast* phase, in which each processor broadcast a data item
2. *Selection* phase, in which a subset of the broadcast data is selected for each memory location if they meet a predefined selection rule
3. *Reduction* phase, in which the subset of data, selected in Phase 2, is reduced to one value and stored in that location

The BROADCAST instruction is assumed to require only O(1) time, so it is equivalent to *M* CW instructions of a PRAM with the same number of processors and memory locations, since this kind of memory access in the PRAM would require O(*M*) time. This shows that BSR is strictly more powerful than the PRAM.

Possible implementations of BSR, e.g., the particular memory access unit (MAU) design, have been discussed in [9,11]. They include using memory buses, using a mesh of tree and using a circuit for sorting and prefix computation. With this model, a variety of problems have been solved in constant-time, such as prefix sums, element uniqueness, sorting, maximal subsequence in one or two dimensions, and convex hull. However, algorithms on PRAM for these problems usually require non-constant time.

2.2. Network-based models:

Another way for processors to communicate is via direct links connecting them, with no shared memory. Instead, the memory locations are more realistically distributed among the all processors as local memory belonging to individual processors. If any processor wishes to access the local memory of another processor, it has to route the data via a network. Depending on the structure and topology, there are different classifications of the interconnection networks. A detailed description can be found in [11]. In Table 2.1, we list them based on their two primary criteria, i.e., degree and diameter. *Degree* is the maximum number of neighbors of a processor, and *diameter* is the longest distance between all pairs of processors in a given network, where the distance of a pair of processors is the number of links on the shortest path between the pair. The network models can also be either synchronous or asynchronous. These two groups of network models are very distinct and have different characteristics. The types of algorithms designed for these groups are also very distinct and involve different issues.

Network	**Degree**	**Diameter**
Linear Array	2	O(N)
Mesh	4	O($N^{1/2}$))
Tree	3	O(log N)
Mesh of Trees	6	O(log N)
Pyramid	9	O(log N)
Shuffle-Exchange	3	O(log N)
Hypercube	O(log N)	O(log N)
Cube-Connected Cycles	3	O(log N)
De Bruijn	2d	K
Star	m-1	O(m)

Table 2.1 Degree and diameter of the network topologies

2.3 Bus-based models

Among network-based models listed in last section, the mesh model is a particularly attractive model for parallel computation. Its advantages include regular structure and simple interconnection topology, which make it particularly suitable for VLSI implementation. However, a serious disadvantage of the mesh is its large diameter. For example, a $\sqrt{n} \times \sqrt{n}$ mesh takes $\Theta(\sqrt{n}\,)$ time to route a data item in the worst case.

In order to overcome this disadvantage, researchers have considered a variety of enhancements for the mesh model. One such approach is to add buses and give the processors the ability to broadcast data over buses. Such meshes are bus-based models and we referred them to as *enhanced meshes*. At any given time, only one processor is allowed to broadcast an item on a bus. The data will be read by all of the remaining processors on that bus in O(1) time.

The two types of electronic buses used for enhanced meshes are *fixed buses* and *reconfigurable buses*. A mesh can be enhanced with a fixed bus using the single global bus model in which all processors are connected to a single bus [11,31]. Another fixed mesh model is the *mesh with multiple buses* (MMB) in which the basic mesh architecture is enhanced with row and column buses (see Figure 2.1) [27,28,32,92,100,101,112]. At each step, broadcasts can occur along one or more rows or else along one or more columns. The row and column buses cannot be used in the same step. The *reconfigurable bus* models allow buses to be created dynamically on the mesh while an algorithm is executed (Figure 2.2) [25,39,78,85,86,88,91,114,127]. The number, connection, and length of these buses are not fixed and are created by the algorithm as needed. Let each processor have four ports, referred to as N, S, E, and W. By setting local connections between pairs of such ports, several disjoint subbuses can be established. The model that allows up to two disjoint pairs of ports to be connected is referred to as the *general reconfigurable mesh* (RM).

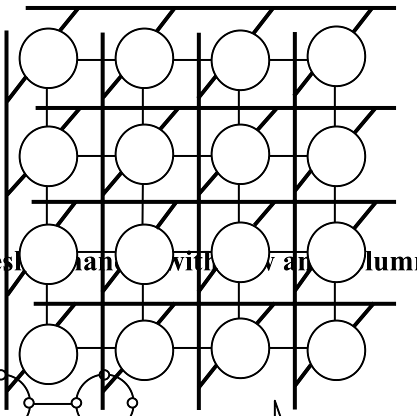
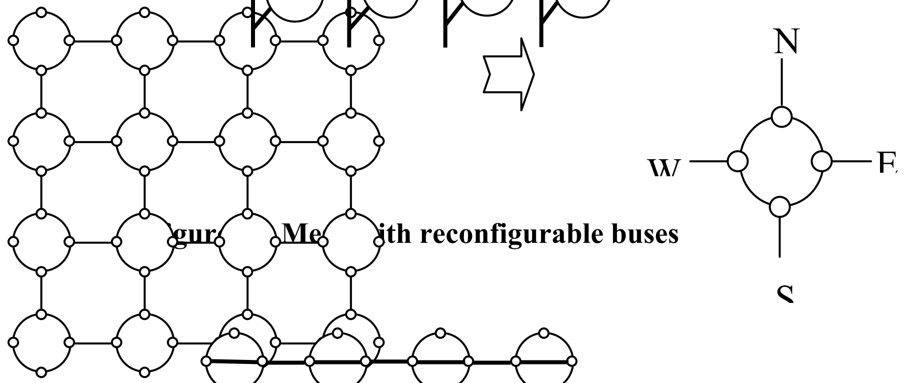


Figure 2.1 Me[illegible]an[illegible]vit[illegible]v an[illegible]lumn buses



[illegible]gur[illegible]Me[illegible]ith reconfigurable buses

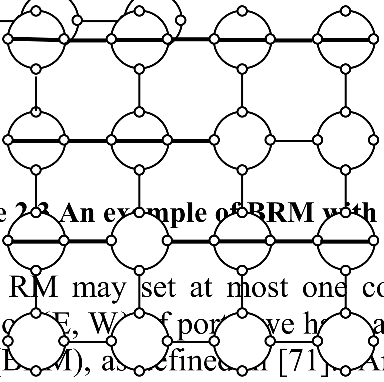


Figure 2[illegible] An ex[illegible]mple o[illegible]BRM w[illegible]h row subbuses

If every processor in a R[illegible]M may set at most one connection involving one of the reconfigurable pairs {N, S} o[illegible]E, W[illegible]f po[illegible]ve h[illegible]a restricted RM that is called the *basic reconfigurable mesh* (B[illegible]M), as[illegible]fined[illegible][71][illegible]An example of BRM is shown in Figure 2.3.

Simulations between enhanced meshes and CRCW PRAM has been given in [71]. The authors show that a $\sqrt{n} \times \sqrt{n}$ MMB can be simulated by a CREW PRAM(n, m) in O(1) time with O(n) extra memory. Similarly, if α denotes the inverse Ackermann function, they show that a $\sqrt{n} \times \sqrt{n}$ BRM can be simulated by a Common CRCW PRAM(n, m) with O($\alpha(n)$) extra time and O(n) extra memory.

Another way to augment a network-based model is using an *optical bus*. An optical bus allows multiple processors to broadcast data on the bus simultaneously and each data can have one or more destinations. These models provide a much wider bandwidth for data transmission than the electronic bus. This capability provides a completely new technique for parallel algorithm design. Since it is not related to our current research work, additional details are omitted in this dissertation and can be found in [11, 70].

2.4. Recently proposed general-purpose models

Both shared-memory models and network-based models have their advantages and disadvantages. The shared-memory model has its major drawback in the fact that there is no way to assign costs for communication between processors that enables accurate performance prediction of realistic machines. The network-based model seems to be better suited to measuring both computation and communication costs. However, it also has two drawbacks. First, it is difficult to describe and analyze algorithms for the network models, in which routing steps are asynchronous and can involve congestion, making prediction of running time difficult. Second, the network model depends heavily on the particular topology under consideration. Different topologies may require completely different algorithms to solve the same problem. This prevents algorithms being general enough and software being portable enough across a wide range of architectures. As a result, no single parallel model has yet dominated parallel computation in the way that the von Neumann model has dominated sequential computation so far.

In recent years, there have been a few models for general-purpose parallel computation proposed [35,49,123,124]. Among these, BSP [123,124] and LogP [35] have attracted most attention from researchers. We discuss them first, and then describe QSM, another general-purpose parallel model [49]. Finally, we give comparison among the three models, along with their relationships to the PRAMs.

2.4.1 BSP (Bulk-Synchronous Parallel) Model

The BSP model was first put forward by Valiant [123]. It is suggested as a candidate for a universal model for parallel computation. It arguably provides a robust model on which to base the future development of general-purpose parallel computing systems [29,47,62,104,123]. A BSP computer consists of three major components:

1. A set of processor-memory pairs, each performing processing and/or memory functions
2. A communication network that transmits messages between processor-memory pairs in the manner of point-to-point.
3. A mechanism performing efficient barrier synchronization of all or a subset of the processors at regular intervals.

A time step is defined to be the time required for a single local operation such as addition or multiplication on local data values. The performance of a BSP computer is predicted using the following three parameters:

p: the number of processors

l: the time interval for synchronization (i.e., minimal number of time steps between successive synchronization operations)

g: the ratio of the total number of local operations performed by all processors in one second to the total number of words delivered by the communication network in one second

A computation consists of a sequence of parallel *supersteps*. During a superstep, each processor is allocated a task consisting of some combination of local computation steps, message transmissions and (implicitly) message arrival from other processors. Each superstep is followed by a global check (barrier synchronization) to determine whether the superstep has been completed by all the components. If synchronized, the machine proceeds to the next superstep. Otherwise, the next period of a fixed number of time units is allocated to the unfinished superstep. In this model, the tasks of computation and communication are separated. The basic task of the communication router is to realize arbitrary *h-relations*, which means that each component in a superstep sends and receives at most h messages. Varied algorithms that have been developed for the BSP model, including broadcasting, prefix sum, and sorting, can be found in [47,123].

The BSP model has been advocated to be a bridging model between software and hardware in parallel machines, that is, between abstract models for algorithm design and realistic parallel architectures. This approach is supported in [123,124] by providing both efficient simulations (on an average) of the BSP model on hypercube networks (synchronous) and of the EREW PRAM (synchronous) model on the BSP. In particular, it is shown that the EREW PRAM can be simulated efficiently with logarithmic slowdown on the BSP model. It is also shown that the BSP can be simulated efficiently with a constant slowdown on the multiport hypercube (on an average). In the multiport hypercube with p nodes, each node can receive a message on each of its $\log p$ incoming wires and route them along the appropriate outgoing wires in constant time (with the constraint that at most one message can be sent along each outgoing wire). In order to hide the latency, synchronization, and memory granularity overheads occurring in the simulation, the choice $l = \Theta(\log p)$ and $g = \Theta(1)$ can be used in the BSP. Besides the above simulation between EREW PRAM and BSP, the most powerful model of PRAMs, CRCW PRAM, is also simulated on BSP. It is shown in [123,124] that if $v = p^{1+\varepsilon}$ for any $\varepsilon > 0$, a v-processor CRCW PRAM can be simulated on a p-processor BSP machine with $l \geq \log p$ in time $O(v/p)$ (where the constant multiplier grows as ε diminishes.) Table 2.2 gives average time costs of simulating the PRAM family using BSP.

Simulated Model	**Simulating Model**	**Simulation Slowdown**	**Ref.**
EREW PRAM	BSP(g, l)	$\geq \max(\log p, l/g)$	[123,124]
QRQW PRAM	BSP(g, l)	$\geq \max(\log p, l/g)$	[50]
CRCW PRAM	BSP(g, l)	$\geq \max(p^{1+\varepsilon}, l/g)$	[123,124]

Table 2.2 Simulations between PRAMs and BSP

2.4.2 The LogP Model

Proposed by D. Culler et al. in 1993 [35], the LogP model is a model of a distributed-memory multiprocessor in which processors communicate by point-to-point messages. It uses several parameters to describe the performance characteristic of the interconnection network, but does not specify the structure of the network. These parameters in the LogP model are:

L: an upper bound on the *latency*, or delay, incurred in transmitting a message containing a word (or small number of words) from its source module to its target module

o: the *overhead*, the length of time that a processor is engaged in sending or receiving of each message. During this time, the processor cannot perform other operations

G: the *gap*, the minimum time interval between successive message sending or successive message receiving at a processor. Thus, the reciprocal of *G* corresponds to the available per-processor communication bandwidth

P: the number of processor/memory modules

In a LogP model, at each step a processor can be either *operational* or *stalling.* When operational, a processor can execute an operation on local data, send a message, or receive a message. Conceptually, for each processor there is an output register temporarily holding a message that is to be submitted to the communication network. The preparation of a message for submission requires *o* time units. Once submitted, a message is accepted by the network, possibly with some time elapsed, eventually delivered to its destination processor. Between the submission and acceptance of a message, the sending processor is assumed to be *stalling*. When a submitted message is accepted, the submitting processor reverts to the operational state.

Although the exact delivery time of a message is unpredictable, the model guarantees that the message arrives at its destination at most *L* time steps after its acceptance. In order to capture network capacity limitations, the model requires that any time there are no more than $\lceil L/G \rceil$ messages in transit for the same destination (*capacity constraint*). Therefore, the parameter *G* corresponds to the reciprocal of the maximum message injection (hence reception) rate per processor that the communication network can sustain, while parameter *L* provides an upper bound on network latency when the system is operating within capacity.

Algorithms such as broadcasting and summation have been developed for the LogP model [35,64,62]. These algorithms are substantially different from those developed for the PRAM model and others. For example, the single item-broadcasting problem can be optimally solved in the LogP model by using a Fibonacci-type broadcasting tree. Most of the described algorithms are quite involved and require low-level handling of messages.

2.4.3 Comparison of the two models

For the BSP and LogP models both, the communication capabilities of the machine are summarized by a few parameters that broadly capture the bandwidth and latency properties. However, there are significant differences between the two models. For BSP, its fundamental features are global barrier synchronization and the routing of arbitrary message sets. For LogP, it lacks explicit synchronization but imposes a more constrained message-passing style to keep the load of the underlying communication network below a specified capacity limit. Intuitively, BSP offers a more convenient abstraction for algorithm design and programming, while LogP provides better control of machine resources.

Comparisons and evaluations between the two models have been made in [29,62,104]. In [29], a variant of LogP that disallows the so-called stalling behavior is considered. The authors create cross simulations between the two models showing their equivalence in algorithmic design when using asymptotic analysis. A non-stalling LogP(L, o, G, P) algorithm can be simulated on BSP(l, g) with slowdown $O(1+g / G + l / L)$. When $l = \Theta(L)$ and $g = \Theta(G)$, the slowdown is constant. On the other hand, a BSP superstep involving at most w local operations per processors and the routing of an h-relation can be simulated in non-stalling LogP with worst-case time $O(w + (Gh + L)S(L, G, p, h))$. Here $S(L, G, p, h)$ is a function of L, G, p, h with the value of O(1) when $h = \Omega(p^{\varepsilon} + L \log p)$ and $O(\log p)$ otherwise. This indicates that, when the bandwidth and latency parameters have the same value in both models, BSP can simulate LogP with constant slowdown and LogP can simulate BSP with at most logarithmic slowdown. Hence, within the asymptotic limits, BSP and non-stalling LogP can be viewed as closely related variants within the bandwidth-latency framework for modeling parallel computation. BSP seems somewhat preferable due to its greater simplicity and portability, and slightly greater power.

Recently, the authors of [104] improve the above results to the stalling LogP model. A non-stalling LogG algorithm can be simulated deterministically efficiently on BSP with slowdown l/L. However, for a stalling LogP, any deterministic step-by-step simulation of LogP on BSP can have arbitrary large slowdown and there is no deterministic step-by-step simulation of stalling LogP on BSP that is efficient.

It has also been shown that the two models can be implemented with similar performance on most point-to-point network [29,35,62,123]. BSP could outweigh the advantages by more convenient programming abstraction and better software portability. On the other hand, BSP assumes special hardware support to synchronize all processors at the end of superstep. This may not be available on many parallel machines, especially in the generality where not multiple arbitrary subsets of the processors can synchronize.

Instead, LogP does all synchronization by messages, although at a higher cost but more realistically.

2.4.4 QSM (Queuing Shared Memory) Model

Instead of message passing, the QSM (Queuing Shared Memory) model [49,104] is based on shared memory. It employs the QRQW strategy mentioned in Section 2.1.2 and has been further extended towards a general-purpose parallel model. It provides one more parameter, g (≥ 1), to reflect the limited communication bandwidth on most parallel machines. Here g is the communication bandwidth gap, i.e., the rate at which processors can perform local operations divided by the rate at which the processors can perform inter-processors or processor-memory communication. Instead of charging the time cost for a step as max(m, k) as a QRQW PARM does, it charges the time cost as follows. Let $m_{op} = \max\{c_i\}$, $m_{rw} = \max\{1, \max_i \{r_i, w_i\}\}$ in one step (defined as the same as QRQW PRAMs). The time cost for the step is $\max\{m_{op}, g^* m_{rw}, k\}$.

Besides dealing with memory contention as QRQW PRAM does, QSM also accounts for limited communication bandwidth while providing the shared-memory abstraction. This QSM model is intended to take advantages of shared-memory and serve as a bridging model by providing efficient simulations on both the BSP, and the LogP [49,104]. The results are included in the summary of next section.

2.4.5 Summary of relationships among the models

Table 2.3 summarizes the relationships among all of the above three models, where l and g are parameters for BSP; L and G are parameters for LogP; G is also assumed for QSM; and p is the number of processors assumed for all of the models.

Simulating Model / Simulated Model	BSP	LogP(stalling)	QSM
BSP		$O(\log^4 p + (L/G)\log^2 p)$ [104]	$O(\lceil G \log p/l \rceil)$ [49]
LogP (nonstalling)	$O(1+g/G + l/L)$ [29, 104]	1	$O(\lceil G \log p/L \rceil)$ [104]
QSM	$O((l/G) + G\log p)$ [49]	$O(\log^4 p+(L/G)\log^2 p+G \log p)$ [104]	

Table 2.3 Simulation results of BSP, LogP and QSM

Chapter 3

The MASC (Multiple Associative Computing) Model

As summarized in last chapter, a number of parallel computation models have been proposed during past decades. Even though a few of them are claimed to work for general parallel computation, none of these models have been efficiently suited to capture associative/SIMD computation well. This is because none of these models have been defined with strict synchronous execution and instruction-level associative properties. Due to broadcast, SIMD computation is very fine-grained and it usually intermingles computation and communication. This feature has not been well captured by other models either. On the other hand, associative SIMD computers have existed since 1970s. They have been used efficiently to solve many problems in various fields. In order to accurately describe associative SIMD computation performed in these computers, the MASC model was proposed by Baker and Potter at Kent State University [99].

In this chapter, we introduce a formal definition of the MASC model and its properties. The original information for this part can be found in [99]. We give them here for the purpose of completion of presenting our research work. Following the definition of the MASC model and its properties, we discuss previous related work on the model.

3.1. The MASC model

The MASC (stands for Multiple ASsociative Computing) model is a generalized associative-style computational model for parallel computation. Motivated from the associative computers STARAN and ASPRO that were successfully built by Goodyear Aerospace in the early of 1970s and 1980s, this model has been developed at Kent State University to be a practical model. The MASC model applies the concept of associative computing to support not only data parallelism but also control parallelism. Thus it provides a complete paradigm that can be used for general-purpose parallel computation. It is a model that is easy to program, highly scaleable and currently buildable. Essentially, the MASC model is a multiple SIMD (or MSIMD) type model that operates synchronously. In the MASC model, one or more instruction streams are allowed. Each of these instruction streams is sent to a unique set in a dynamic partition of processing elements (PEs). This allows a task currently in execution to be partitioned into multiple tasks using control parallelism. The associative feature of the model allows data in the local memories of processors to be located by content rather than by address.

As shown in Figure 3.1, the MASC model consists of an array of processing elements (PEs) and one or more instruction streams (ISs). A PE essentially acts as an ALU (Arithmetic and Logic Unit) in a conventional computer, as it does not store any copy of a program, neither decoding nor executing the program independently. Each PE, paired with its local memory, is called a *cell*. (In this thesis, we exchange to use PEs and cells when there is no conflict involved.) The array of MASC machine with *n* PEs and *j* ISs is written as MASC(*n*, *j*). Normally, *j* is expected to be much smaller than *n*. Each PE is capable of performing local arithmetic, logical operations, and the other usual functions of a conventional processor other than issuing instructions. There are three real or virtual networks; namely, a PE network used for communications among PEs, an IS network used for IS communications, and a broadcast/reduction network between ISs and PEs used for communication between an IS and a set of cells.

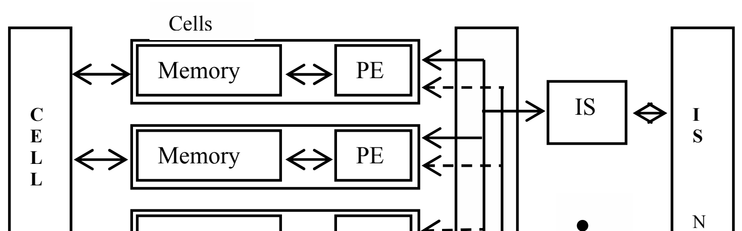


Figure 3.1 The MASC model

3.2. Properties of the MASC model

A detailed description of the MASC properties can be found in [99]. Here we briefly list them as a summary.

Cell properties

A cell is composed of a PE and its local memory. Each PE can only access its own local memory and shares no memory with other cells. PEs may obtain data stored in other cells either from the IS broadcast network or from the (synchronous) cell network.

IS properties

An IS is logically a processor with a bus connection to each cell. Each IS has a copy of the program being executed and issue an instruction to the cells that listen to by broadcasting. Each cell listens to only one IS at any given time and initially all cells listen to the same IS. A cell can switch to another IS in response to commands from the current

IS. A cell can is active, inactive, or idle. An active cell executes the program issued from the IS to which it is currently assigned while an inactive cell listens to but does not execute these instructions. An idle cell contains no data needed. All idle cells are managed by a special IS and may be reassigned to some IS later. An IS can instruct any of its inactive cells to become active again. By issuing multiple instruction streams simultaneously, the MASC model supports control parallelism in a synchronous manner.

Associative properties

Assuming the word length is considered to be a constant, the MASC model supports the global reduction operations of OR and AND of binary values and of maximum and minimum of integer or real values for each IS and its active PEs in constant time. Additionally, the MASC model supports a constant time associative search (assuming the word length is a constant) due to data in the local memories of processors to be located by content rather than by address. The cells whose data values match the search pattern are called *responders* and the unsuccessful ones are called *non-responders*. The IS can activate either the responders or the non-responders. Each IS can select (or "pick one") arbitrary responder from the set of active cells in constant time. This IS can also instruct the selected cell to broadcast a data item on the bus and all other cells listening to this IS receive this value in constant time.

Following this chapter, Chapter 4 will be dedicated to analyze why these associative operations can be justified to take constant time.

3.3. Previous work on MASC

A large amount of research work has been done for the MASC model. It has involved in multiple research projects such as developing programming techniques, designing MASC algorithms, model simulations, associative instruction implementation, etc.

A declarative associative language called ASC has been implemented for MASC(n,1) on many platforms including Goodyear's STARAN, Goodyear/Loral/Martin-Marietta's ASPRO, WaveTracer, and Thinking Machine's CM-2, and provides true portability for parallel algorithms[98,99]. In addition, an efficient ASC simulator has been implemented on both PCs and workstations running UNIX. It provides an efficient and easy way to design programs for algorithms that primarily utilize data parallelism while retaining the MASC associative properties.

A wide range of different types of algorithms and several large programs has been developed, analyzed, and implemented using ASC language. Some examples of algorithms are as the following. In [98], an optimal algorithm is given for a minimal spanning tree for an undirected graph. In [13,14], MASC versions of different convex hull algorithms have been designed and analyzed; they have optimal average costs. In [37], a MASC string matching algorithm is given that supports variable length "don't care" symbols (VLDC), using MASC with one IS and a linear network. This algorithm has the same lowest cost as established by other well-known parallel algorithms (using EREW PRAM). In addition, it has the unique feature of identifying all continuation points after each "don't care" character. More than the above algorithms, many other algorithms have also been analyzed in various application fields such as graph algorithms, image processing, database management programs, compiler optimization, and air traffic control task processing [98,99,102, 82,83,84].

Previous MASC simulations include simulation of PRAM with MASC [117,119] and self-simulation [118,119]. Let MASC(n,j) denote a MSC with n PEs and j ISs and PRAM(n, m) denote a PRAM with n processors and m shared memory locations. Without a cell network, MASC(n, j) can simulate priority CRCW PRAM(n, O(j)) in constant time with high probability. Also, MASC(n, 1) can simulate priority CRCW PRAM(n, m) in constant time when m is a constant number. Using both network algorithms and ISs to move data, MASC(n, j) can simulate priority CRCW PRAM(n,m) in O(min{n/j, m/j, route(n)}) with high probability, where route(n) is the average time for a general network routing. The self-simulation ability of MASC also allows the model to support a virtual machine with more PEs and ISs than are actually available. MASC(n, j) can simulate MASC(N,J) in O($N/n + J$) extra time and with O($N/n + J$) extra memory, where $N \geq n$ and $J \geq j$. This has established that MASC algorithms are highly scaleable.

Implementation of associative processing and associative operations at the instruction-level has also been explored in [126,129,131]. Using field-programmable logic device (FPLDs), the authors of [126] designed an initial version of an associative processor based on the earlier the STARAN computer but being updated with modern design practices. The design consists of an associative array of 8-bit RISC PEs, operating in byte-serial fashion with one IS. It can execute assembly language code produced by a machine-specific back-end compiler. Furthermore, experiments on implementing associative instructions including associative search and responder resolution have been shown in details in [131]. The authors of [129] have explored how to implement ASC in regard with scalability. The research work on the hardware implementation has provided very useful information about building the MASC machine using modern VLSI hardware design technology.

Chapter 4

Timings for Associative Operations on the MASC Model

4.1. Introduction

As mentioned in Chapter 3, a large amount of research work has been undertaken for the MASC model. However, it has been long noticed that some of the research on the MASC model has identified certain important issues regarding the cost assigned to basic associative MASC operations such as the broadcast and reduction operations. Both the accuracy of the cost assigned to these operations and its fairness with respect to the costs assigned on other parallel models are extremely important, as they determine the accuracy of the comparison between MASC and other models. Likewise, the accuracy and the fairness are essential in determining the ability of MASC to efficiently support applications in different areas, both for programming and complexity analysis of algorithms. In making fair cost assignments, it is necessary to consider the theoretical asymptotic rate of increase in the cost that occurs as the data size increases. However, it is equally important to also consider how these operations can be implemented in hardware and the running time of the implementations of these operations. The theoretical asymptotic rate of increase, together with the running times for hardware implementations can be used to produce graphs that project feasible running time for an operation on data sets of varying size. When this graph is bounded above by a small constant even when the number of processors exceeds what is considered to be feasible in the foreseeable future, we explore the option of assigning a constant running time to these basic MASC operations.

In this chapter, we investigate these issues based on the implementation details of Goodyear's STARAN, which is the architectural ancestor of the MASC model. Its broadcast/reduction network that was designed by K. Batcher [18,19,21,23] is the basis of our work in this chapter. We will discuss and analyze comparative fairness for the timings of certain basic operations on the MASC with respect to those on other parallel models.

This chapter is organized as follows. Section 4.2 presents motivation for the research on timings of MASC. Section 4.3 describes the hardware implementation from STARAN to support these basic operations. Section 4.4 discusses the individual operations based on the implementation facts. Section 4.5 compares the costs of these basic operations on other models and addresses the issue of what costs should be assigned to MASC when it is compared (using simulations) to the related models. The last section is a summary.

4.2. Motivation for research on timings of MASC

When we assume that the word length is considered to be a constant, the MASC model supports the following important constant time operations for an IS:

- Broadcasting an instruction stream or a data item to the set of PEs listening to the IS
- Global reduction of a binary value stored in each active PE using logic OR or AND
- Global reduction of an integer (or real) value stored in each active PE using maximum or minimum
- Associative search to find the cells whose data values match the search pattern (called responders) or do not match the search pattern (called non-responders)

- The AnyResponder operation to determine if there is any existing responder after an associative search
- The PickOne operation to select (or "pick") an arbitrary responder from its set of active PEs

These basic operations are fundamental to support the associative style of computing. With more advanced research on MASC involving in algorithm complexity analysis, especially comparative to other parallel models, the MASC timings, in particular, with those involving constant time operations have drawn more attention and been questioned. While these constant time assumptions were initially viewed as "self-evident", it is now important to provide justification for them. The correctness of the timings assigned to each of these operations depends on both possible hardware implementations and comparative fairness with respect to other parallel models. While other hardware implementations are possible, we will establish the feasibility of implementing a bit-serial broadcast/reduction network that support these operations. . Also, we will analyze the fairness of the costs assigned to these operations with respect to the costs assigned on other well-accepted models such as RAM, PRAM, and bus-based architectures. We will discuss these topics in the following sections.

4.3. Broadcast/reduction network

In this section, we take a close view on a possible hardware implementation of the broadcast/reduction network based on the STARAN computer [18,19,80,107], which is the architectural motivation for the MASC model. The STARAN was an associative SIMD computer with 512 to 4096 PEs, depending upon the size of a particular installation. The STARAN was built in the early 1970's and the ASPRO [74], its architectural descendent, was built in 1980's by Goodyear Aerospace. Currently, the ASPRO is produced by Martin-Marietta and is used by the U.S. Navy. Their hardware implementation of associative operations through the broadcast/reduction network provides a possible implementation for these basic associative operations on the MASC model.

The broadcasting/reduction network on the STARAN is constructed using a group of resolver circuits [23,18]. A N-PE resolver consists of N PEs labeled PE_0, PE_1, …, PE_{N-1} and each PE_i has a responder bit R_i that is equal to 0 or 1. The resolver is designed to be able to tell each PE whether or not any earlier PE has a responder bit equal to 1. Thus for each i, it computes $V_i = R_0 \vee R_1 \vee R_2 \vee \ldots \vee R_{i-1}$, where $\vee$ is the logic-OR operator and $0 < i \leq N$. The resolver also sends V_N to the control unit to tell it whether or not any responder bits are set to 1. A 4-PE resolver is illustrated in Figure 4.1.

In practice, by using the parallel prefix idea, a resolver for N PEs can be built as a $\log_4(N)$-level tree of 4-PE resolvers. Each leaf represents a PE and the PEs are partitioned into groups of 4 PEs, which are fed into the first level of 4-PE resolvers. A reduction operation is executed by sending the signals down the tree to the PE leaves and then back up to the root of the tree to obtain the final result, while accumulating partial results in middle. Obviously, the delay from any input to any output is at most ($2 \log_4 N$ - 1) gates. A 16-PE resolver tree is shown in Figure 4.2.

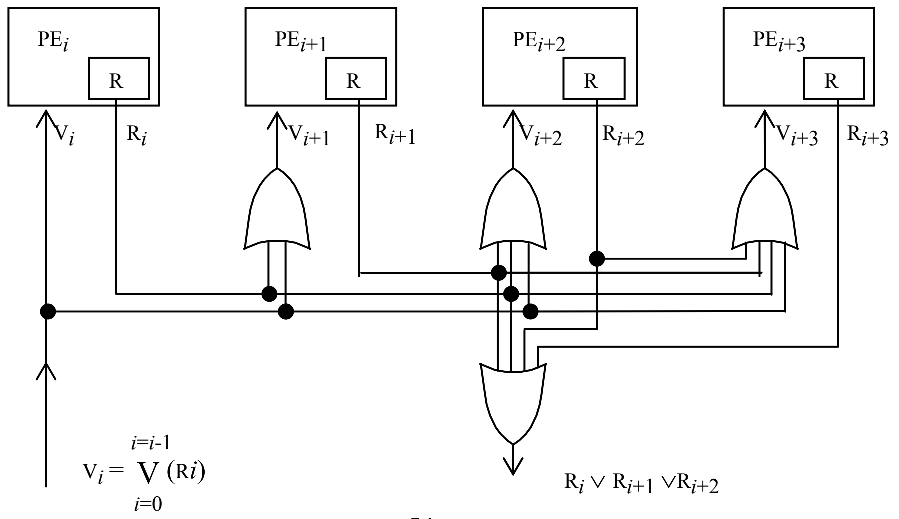


Figure 4.1 A 4-PE resolver with at most an 1-gate delay from any input to any output

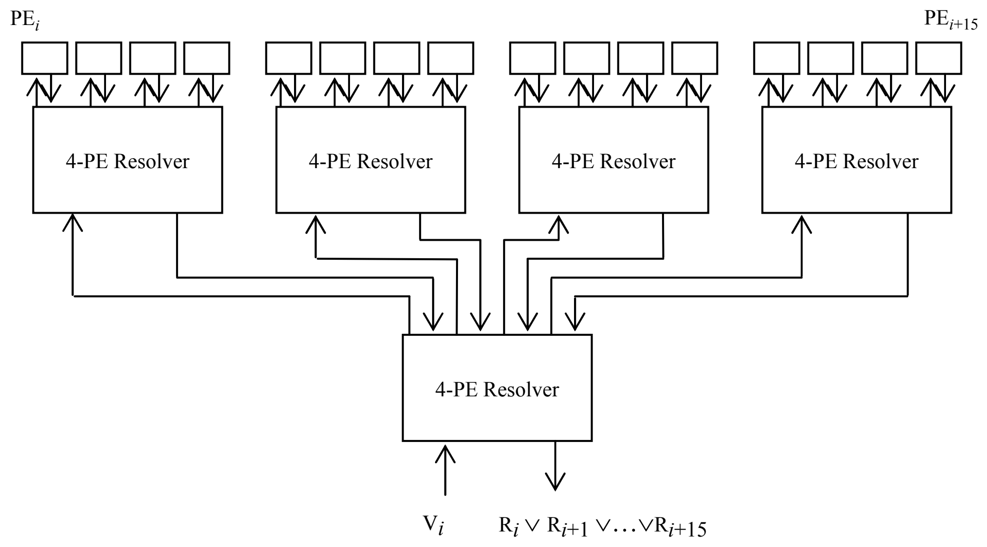


Figure 4.2 A 16-PE resolver with at most a 3-gate delay from any input to any output

4.4. Discussion on the basic operations

In this section, we discuss the timings on the individual basic MASC operations given in Section 4.2.

We consider broadcasting first. Broadcasting on bus-based architectures provides a fast way to transmit data between processors that are far apart. The assumption that an arbitrary subset of broadcasts on a bus-based architecture can occur in constant time, regardless of the number of processors or the length of buses, has been generally accepted in the literature [5,25,27,31,32,71,72,75,79,85,92,100,101,112,113]. This provides consistency among researchers in the field and is supported by experimental evidence even for very large architectures by today's standards [77,96]. For example, in today's architectures with at most tens of thousands of processors, the time to broadcast a data item over a bus is usually no more than that to perform a basic operation such as an addition by a sequential processor. Hence it can be bounded within one or a few machine cycles [25,96]. This constant time assumption for broadcasting greatly simplifies calculation of running times of algorithms and comparison of various models without loss of prediction accuracy. [1]

Based on the preceding paragraph, we assume the timing for a broadcast on the MASC model from an IS to its PEs to be constant. On the MASC model, broadcasting is used to send an instruction stream from an IS to its PEs and transmit data between these PEs. Broadcasting a bit or a word on the MASC model is performed through the broadcast network.

Practically, the broadcast network may be implemented as a separate network from the reduction network yet with the same structure (like in the STARAN), or they may share the same network. It is easy to observe that it takes $\log_4 N$ gate delay for a bit to travel from the root of the tree network to the N PE leaves at the bottom. Recall that the time required for broadcasting on a bus-based architecture increases linearly with respect to the bus length and the number of the processors. However, the time required for broadcasting using a tree-based network is logarithmic, so it increases asymptotically slower than when using a bus-based architecture.

The gate delay from any input to any output on the broadcast/reduction network for the MASC model is at most $(2\log_4 N - 1)$. This cost is given particular attention here. We argue that for practical purpose it could be bounded by a small constant. The reasoning is as follows. Typically, a gate delay takes about 1-5 nanoseconds. We may imagine even if we built an extremely large machine with size of $2^{2^{10}}$ processors, the gate delay would be at most $(2\log_4 2^{2^{10}} - 1) \times 5$ nanoseconds ≈ 5.1 microseconds. On the other hand, building a machine with $2^{2^{10}}$ processors appears to be impractical, since the number of atoms in the known universe is estimated to be less than 2^{2^9} [11]. It is likely that machines in the foreseeable future will have at most a few million processors. A machine with 100 million processors would have the gate delay less than 50 nanoseconds, which is comparable to the time for a memory access in today's systems. Since there are approximately 8 billion neurons in the human brain, it seems reasonable to conjecture that an efficient machine with that many processors would necessarily have to use a model quite different from those used today. Also, since a processor can handle much more complex computations than is considered possible for a neuron, it seems likely that if a machine were built to be able to handle the computations typical of the brain, then it would probably have far fewer than 8 billion processors.

1 Since the number of gates that a bus is able to drive is limited, perhaps a re-evaluation of the time for broadcasting at the VLSI level is needed by researchers in the field before making this assumption for millions of processors. In considering VLSI designs to maximize the fan-out of a gate and minimize the layers of gates, a possible approach might be to increase the fan-out of the *n*-ary broadcast tree used here (e.g., to a 100-ary tree).

We give brief diagrams drawn by xMaple v5.0 for the function graph of ($2\log_4 N$ -1) gate delay for this timing. Given the average 1-gate delay is 2 nanoseconds, the function graphs are shown in Figure 4.3 in regular scales and Figure 4.4 with the vertical axis in logarithmic scales, while N is between 1 to 2^{2^9} . (Notice that 2^{2^9} is approximately equal to 1.2e+154 shown in the figures. We did not give the function curves from 2^{2^9} +1 to $2^{2^{10}}$ due to the limitations of our plotting device.) The two horizontal lines in Figure 4.4 are reference lines. It is clear that this function changes very slowly and is bounded by a small constant even if N is unreasonably large.

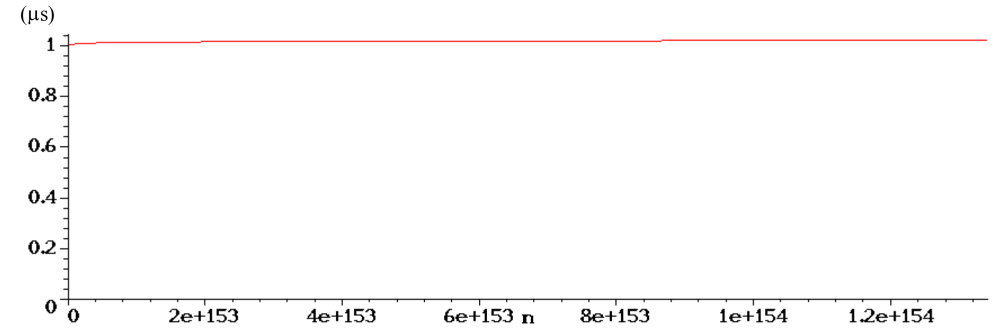


Figure 4.3 The gate delay of ($2\log_4 N$ -1) using regular scales.

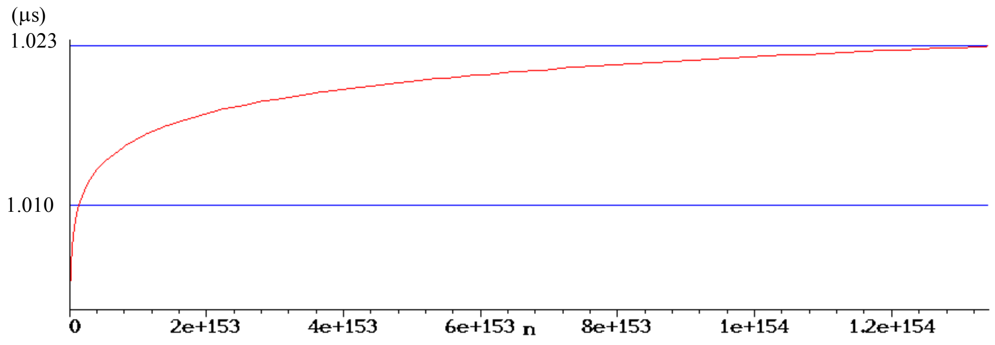


Figure 4.4 The gate delay of ($2\log_4 N$ -1) with the vertical axis using a logarithmic scale

For a very large architecture, wire length could be also an issue. According to the optimal VLSI layout, when the number N of processors is very large, the wire length can increase as large as $N^{1/3}$, based on the 3-D cube topology. However, this same problem exists in bus-based architectures. We may use the same solution (if any) that would be used in bus-based architectures to solve this problem. An option used in [25] is formulating a parameter, i.e., the cycle of a computation, as the maximal length of a single bus to capture this notion.

Let ω be the greater of the length of an instruction or a data item (i.e., word). Obviously, broadcasting an instruction or a data item one bit at a time from an IS to its PEs takes $O(\omega)$. With architectures that have already been built or those currently envisioned, the size of ω is a small constant. However, most parallel models assume that the processor's identification number can be stored in a word, thus requiring that for a machine with N processors, ω must be at least $\lceil \log N \rceil$ in size. It is traditional to assume in the bus-based architectures that the buses have bandwidth ω. In particular, for MASC we may build the separate broadcast network with the bus bandwidth ω so that it transmits ω bits simultaneously. Therefore, it is very reasonable to assume that both the word and instruction broadcasts require constant time.

Next, we discuss the logic OR global reduction operation. With each PE holding a binary value, a global OR is performed through the $\log 4\ N$-level tree of 4-PE resolver network, as shown in Figure 4.1 and Figure 4.2. Thus, the same reasoning used for the 1-bit broadcasting can be used to justify assuming that computing a global OR takes constant time, although the tree traversal is in the opposite direction or in a bottom-up manner.

A global AND operation is implemented using a global OR operation simply by adding compliment gates to the inputs. So it can be assigned the same timing as a global OR operation.

An associative search is a unique feature of an associative model like MASC. On MASC, it is executed through the broadcast/reduction network by broadcasting a predefined search pattern to the PEs from an IS. Those PEs with matching values set their responder bit and remain active. Then an AnyResponder operation can be used to check if there exists any responder. Also, using the reduction network, the IS can select (or pick) an arbitrary (usually the first) active PE to do any special processing. We refer this to as a PickOne operation.

Clearly, an AnyResponder operation is essentially a global OR operation over all the responder bits. It involves one tree traversal of the reduction network and returns true if any responder bit is set among the active PEs and false otherwise. A PickOne operation is also performed by the reduction network to pick an arbitrary but usually the first responder and to go up through the same tree traversal as a global OR. So both these two operations should have the same timing as a broadcast or a global OR, i.e., constant time. Notice that, after a PE is picked and processed, the PickOne operation clears the responder bit of the PE and the IS can proceed to pick another of the active processors. An associative search involves three steps, i.e., a broadcast of word-length pattern; a sequential comparison of two word data by each active PE; and a global AnyResponder operation. Hence, the associative search also requires constant time.

Let the word length be ω. In a bit-serial SIMD architecture such as the STARAN, operations on multiple bits are performed bit-serially. This means that, if a 1-bit local addition takes O(1), then a two-word local addition takes O(ω). If ω is assumed to be constant, then addition/subtraction is also a constant time operation, as one would expect. Some researchers argue that the word length in parallel computers with N PEs should always be (log$N$) or larger [11] so that it can store the identification number of each PE. However, since a parallel computer could have over 1.8×10^{19} PEs before log N is larger than 64, the assumption that the word length is also a constant seems to be reasonable, based on the size of current parallel computers and those in the foreseeable future.

Consider computing a global Maximum (or Minimum) of N integer (or real) values. As before, we continue to assume that the word length ω is considered a constant. Let each of the N PEs contains a value in a common word-size field. The global Maximum (or Minimum) operation can be implemented as a bit-serial operation with a global OR being computed for each bit of the field in the order of the left (most significant) bit to the right (least significant) bit. Each global OR bit operation keeps active those PEs whose bit value is 1 (or 0 for Minimum) until either only one responder is left or until all bits have been processed [98]. Additionally, if all active processors have a 0 value for one bit, then all remain active for the next round. If we assume that the time to compute a global OR is constant and ω is the word length, then the time for this operation is O(ω). If ω is also assumed to be a constant, as assumed for the addition of two words above, the time for a global Maximum (or Minimum) operation will also require only constant time.

As a related observation, we compare the implementation of the broadcast/reduction network of the MASC model that is in a bit-serial fashion and a sequential processor that is in a word-serial fashion. In fact, the hardware needed to support a sequential addition operation is not that different from the hardware needed to support the mentioned MASC basic operations on the broadcast/reduction network. It has been shown that the lower bound for a sequential processor to perform addition is Ω(log k) where k is the number of bits of the two operands [67,93]. A sequential processor performing addition using carry-look-ahead adders as building blocks is much like a MASC performing a global reduction using resolvers as building blocks. The cost of addition for a sequential processor is considered to take constant time regardless of the number of its input bits (i.e., length of the operands). With the almost same hardware circuits, it is reasonable to apply the same measurement that ignores the number of input bits (i.e., number of PEs) on the MASC reduction operations.

The prefix sum operation is not supported in hardware in either the STARAN or the ASPRO. Also, it is not yet a basic operation on the MASC model. But based on the principle we discussed earlier, it is possible to implement this operation using the reduction network for a group of 1-bit binary values with each stored in a PE. As claimed in [93], "any design for a prefix sum operation can be converted to a carry computation network by simply replacing each adder with the carry operator." This could take the same time as a sequential addition operation, as well as a MASC reduction operation. However, for a ω-bit prefix sum, it may need more expensive hardware support to take the partial sums and partial carries to the next round of addition.

It is important to keep in mind that all of the above basic operations are implemented in hardware, in contrast to other parallel computational models that execute most of them using software algorithms. These features make the MASC model a unique parallel computation model that is powerful and feasible.

4.5. Comparison of timings with other models

In order to compare the MASC model with other computation models such as PRAM or MMB, it is critical that the timings charged for the basic operations of each model be fair. For example, if two models both support a broadcast with the same hardware, it would be unreasonable to charge one model O(log N) for a broadcast and the other O(1). Also, for two operations using the same hardware in one model, we should not charge one O(1) and the other O(log N).

It is useful to compare the timing cost for the basic MASC operations to those of similar operations for RAM and PRAM memory access. We give a short summary of the computational complexity of RAM and PRAM memory access presented in [11]. The memory access on these two classic models can be implemented by a combinatorial circuit that is called the Memory Access Unit (MAU). The MAU of RAM is implemented as a binary tree of switches. Each memory access from the processor has to use the path between the root of the tree and a memory location at a leaf of the tree. When the memory size is M, the depth of the MAU is clearly Θ(log M). For PRAM, the lower bound for the depth of a MAU with M global memory locations and N (= O(M)) processors is also Ω(log M). A theoretical optimal MAU for PRAM with depth Θ(log M) is possible using a sorting circuit and a merging circuit. However, there is an efficient and more practical MAU for PRAM with depth Θ(log$M$). Since the circuit in the optimal

MAU has depth O(log M), each memory access is performed with a O(log M) switch delay. This is exactly the same situation as observed with the basic operations for the MASC model. Researchers and algorithm developers have ignored this small theoretical logarithmic factor for many years and have elected to treat memory access time as a constant, just like the time for other basic operations such as addition and comparison. This timing approach has dominated algorithm analysis on both the RAM and PRAM models in the literature and has significantly simplified and standardized complexity analysis of algorithms, both in terms of comparing algorithm performance and of comparing computational models.

We may further consider reduction operations for PRAM. Using prefix operation algorithms, all PRAM models can calculate logic OR/AND or maximum/ minimum in O(log$N$). However, using concurrent writes, the CRCW PRAM can compute every Boolean function with a small domain (e.g., binary values) in constant time [106]. But this does not hold for other variants of PRAM without concurrent writes, as they require at least Ω(log N) algorithm steps for such a computation [36]. Thus, the charges for reductions and basic operations of logic OR/AND are reduced for the CRCW PRAM due to an assumed superior architectural implementation.

Next, we compare timings on MASC with timings on MMB (Meshes with Multiple Buses), which may be considered a more practical computational model than PRAM. A MMB is a basic mesh enhanced with multiple buses, one bus for each row and one bus for each column. A processor can broadcast along its row or column bus to allow fast data transmission to all processors that are on the same row or column bus. In one time unit, only one processor is allowed to write a value on a bus. All other processors are assumed to simultaneously read the value being broadcast in constant time.

As with other bus-based architectures we discussed earlier in Section 4.4, the time to broadcast along a bus on MMB increases linearly as the number of processors increases, i.e., O(N). It has been generally accepted that this time can be considered to be constant. As for other basic operations on MMB, it has been shown in [101] that on a MMB with size of $N^{1/2} \times N^{1/2}$ and each PE holding a data item, the global reduction of finding a maximum/minimum or logic OR/AND can be computed in O($N^{1/6}$). Also, if all data items lie in the same row with only one item per processor, these reductions take O(log N). For the latter cases, it has been shown in [71] that Ω(log N) is the lower bound for these reductions. Since MMB does not contain a circuit to perform these reductions, specific algorithms have to be designed for them. It is clear that substantially different methods are used to execute these operations on the MASC model.

More recently, an augmented MMB model called the MHB (Meshes with Hybrid Buses) model was proposed in [72]. A MHB is a MMB enhanced with precharged 1-bit row and column buses. By draining the precharged high voltage to ground on the augmented bus, concurrent broadcast by several processors can be supported. This makes it possible for a global OR reduction operation to be done in constant time. This is called an ".OR" (dot OR) bus by hardware people and differs from the logic OR via gates that we discussed earlier for the MASC model. It also ignores a small but non-constant time for precharging or draining on the augmented bus (which is sometimes claimed to be logarithmic). Furthermore, it supports our simulation results in [15,16], which we will present in next chapter in details, that a MMB is less powerful than a MASC with a 2-D mesh unless it is augmented with extra hardware.

The above discussion and comparison of MASC with other models justify our constant time assumption concerning the timings of the broadcast and reduction operations on the MASC model. We argue that our assumption uses the same methodology that the other models have used, both for the purpose of theoretical research and for practical implementations.

4.6. Summary

We have given the details of our timing justification for the MASC basic operations, based on their hardware implementation on the STARAN, which is the architectural ancestor of the MASC model. We have also compared our timings with those of other models. A summary of the MASC operations is given in Table 4.1. In the second column, we list the timings when we assume that the broadcast bus has a 1-bit width. In the third column, we assume that the broadcast bus has a ω-bit width. These timings are based not only on actual implementation (e.g., STARAN and ASPRO), but on the comparable timings used by RAM, PRAM, and bus-based architectures. These models were included as part of our study in order to evaluate which MASC timing charges result in the fairest possible comparison of MASC to other models for parallel computation.

Operations	**1-bit Bus**	**ω-bit Bus**
Broadcast	O(ω)	O(1)
Addition/Subtraction	O(1)	
Logic OR	O(1)	
Logic AND	O(1)	
Associative Search	O(ω)	O(1)
AnyResponder	O(1)	
PickOne	O(1)	
Maximum/Minimum	O(ω)	O(1)

Table 4.1 Summary of the timings of the basic MASC operations

There are several issues to consider in determining the proper cost for basic operations for a model. These include implementation considerations, experimental data involving prototypes, theoretical considerations, and comparison with the costs assigned to other models with similar capabilities. Other considerations are that the costs assigned should be kept simple whenever possible, consistent with the added requirement that they must provide accurate comparison of performance of algorithms designed for the model and with the performance of this model when compared with other models. It would be useful to have more experimental data using modern technology to test the timings assigned to the basic MASC operations.

Chapter 5

Simulations of Enhanced Meshes with the MASC Model

5.1. Introduction

The MASC model provides a practical, highly scalable model that naturally supports massive parallelism and a wide range of applications. Generally, the power of a computational model is indicated both by the efficiency of algorithms it can support and by the efficiency with which it can simulate other computational models.

As we introduced in Section 2.3, the enhanced mesh models are well-established models for parallel computation. An enhanced mesh model is a mesh augmented with a certain number of fixed buses or reconfigurable buses. The basic mesh enhanced with a single global bus is referred to as *the enhanced mesh with a global bus*. The basic mesh enhanced with fixed row and column buses is called the *mesh with multiple buses* (MMB) model. The basic mesh enhanced with reconfigurable buses that allow buses to be created dynamically on the mesh while an algorithm is proceeding is called the *reconfigurable mesh* model. Each processor in a RM has four ports that can be set local connections to establish several disjoint subbuses. If up to two disjoint pairs of ports are allowed to connect, it is referred to as the *general reconfigurable mesh* (RM). If at most one connection involving one of the reconfigurable pairs is allowed, it is referred to as the *Basic reconfigurable mesh* (BRM).

In this chapter, we present efficient simulation algorithms between enhanced meshes and MASC. We will give some results based on these simulations. The simulations of the enhanced mesh models with the MASC model and the simulations in the other direction provide a better understanding of the power of the MASC model. In addition, they provide an effective method for converting a large number of algorithms designed for enhanced meshes to the MASC model.

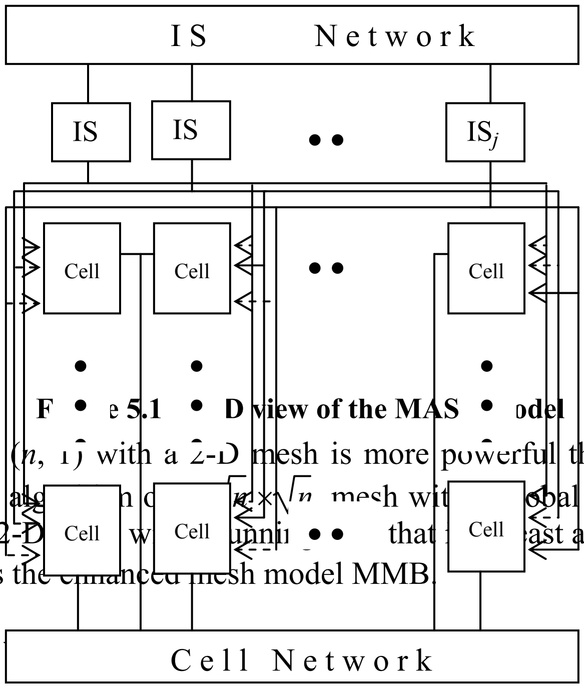
This chapter is organized into five sections. In particular, Section 5.2 gives an overview of simulations of enhanced meshes with MASC. Sections 5.3 and 5.4 present the simulations of MMB and BRM with MASC, respectively. Section 5.5 discusses the reverse simulations, i.e., simulations of MASC with MMB and BRM. Section 5.6 generalizes the simulation results from basic reconfigurable meshes to a general reconfigurable mesh. Section 5.7 gives the concluding remarks and some open problems.

5.2. The overview of simulations of enhanced meshes

Unless noted otherwise, we assume in our simulation algorithms that the mesh network for MASC(n, j) has size $\sqrt{n} \times \sqrt{n}$ and is in row–major order. The ith PE in the MASC array will be denoted PE_i ($1 \le i \le n$) and the ith IS will be denoted IS_i ($1 \le i \le \sqrt{n}$). Unless noted otherwise, the size of all enhanced meshes considered is $\sqrt{n} \times \sqrt{n}$ and is in row-major order. The processor located in row i and column j ($1 \le i, j \le \sqrt{n}$) is referred as to P(i, j).

For enhanced meshes, instructions used in algorithms consists of performing an arithmetic or boolean operation, communicating with a neighbor, broadcasting a value on a bus, or reading a value from a bus. In the MASC model, each PE is assumed to have the exactly same computation power as the PEs of an enhanced mesh processor. This assumption will simplify the simulations and require that the register and word length of both models be O(log n), the word length assumed for enhanced meshes. Since a parallel computer could have over 1.8×10^{19} PEs before log n is larger than 64, this simplifying assumption seems to be very reasonable, based on the size of current parallel computers. Additionally, in order to compare the enhanced mesh models with the MASC models fairly, we assume the MASC buses have the same power as those of the enhanced meshes. In particular, we assume that the ID number of a PE or an IS can be broadcast on a bus and stored in constant time. Since both the MASC model used here and the enhanced mesh have a $\sqrt{n}\times\sqrt{n}$ mesh network, it is natural to map the processors in MASC to the those in the same position in the mesh network in MMB. Moreover, the 1-D representation of the MASC cells (PEs) in Figure 3.1 can also be viewed as the 2-D array as shown in Figure 5.1. Since both models have identical mesh networks, the mesh operations can be simulated automatically. Therefore, the remaining work required is to simulate with MASC the operation of broadcasting over buses in an enhanced mesh.

Simulating an enhanced mesh with a global bus with MASC(n, 1) is trivial. All we need to do is to let all PEs listen to IS_1 . This means that any algorithm running on a $\sqrt{n}\times\sqrt{n}$ mesh with a global bus can be run in MASC(n,1) with mesh connection in the same time. On the other hand, the reduction operation of maximum requires only O(1) time for MASC(n, 1) but non-constant time for the mesh with a global bus [5]. Intuitively, we have the following theorem.



F[illegible]e 5.1 [illegible]D view of the MAS[illegible]odel

Theorem 5.1. MASC (n, 1) with a 2-D mesh is more powerful than a $\sqrt{n}\times\sqrt{n}$ mesh with a global bus. Any alg[illegible]n [illegible] $\sqrt{n}\times\sqrt{n}$ mesh wit[illegible]obal bus can be executed on MASC(n, 1) with a 2-D [illegible] w[illegible]unni[illegible] that [illegible]ast as fast.

We next discuss th[illegible] [illegible]h model MMB.

5.3. Simulating MMB [illegible]

In the following two sections, algorithms are presented for MASC to simulate enhanced meshes. We will discuss simulations of MMB first and then of BRM in sections 5.3 and 5.4, respectively.

Each PE$_i$ ($1 \le i \le n$) in MASC with a 2-D mesh knows its position in mesh network and two parallel variables, row\$ and column\$, are used to store its row number and column number. (A "\$" is added to the end of PE variables to avoid confusing them with IS variables.) The coordinates (row\$, column\$) are the same as that of the corresponding MMB processor. Assume the MASC-processor PE$_i$ simulates the MMB-processor P(r$_i$, c_i), where $i = (r_i - 1)\sqrt{n} + c_i$ and $1 \le c_i \le \sqrt{n}$. Since PE$_i$ can compute its coordinates points in the mesh as follows: $r_i = \lceil i / \sqrt{n} \rceil$ and $c_i = (i - 1) \bmod \sqrt{n} + 1$, the desired simulation is a sequence of steps in which every processor in MASC is responsible for simulating the corresponding MMB processor.

As mentioned earlier, we only need to deal with the simulation of broadcasting in MMB. In [15,16], we restricted the case for MASC(n, j) with $j = \sqrt{n}$. Here, the general case for MASC(n, j) with j instruction streams is considered. We assign each IS$_i$ ($1 \le i \le j$) to the following rows and columns. IS$_i$ is assigned to row (column) i, row (column) $i + j$, row (column) $i + 2j$, …, row (column) $i + \lceil \sqrt{n} / j \rceil * j$, provided they exist. In other words, PEs in row (column) i are designated to listen to their assigned IS whose ID is (i mod j) + 1. In one step, an IS can switch from instructing its rows of PEs to instructing its columns of PEs. Initially, all PEs listen to IS$_1$. Based on row broadcasting or column broadcasting, they can switch to their assigned row or column ISs in constant time. The following algorithm gives a MASC simulation of a MMB row broadcast.

Algorithm 5.1.

```
Begin
1. First all PEs switch to their assigned row IS. Each PE row should have at most one
   PE that needs to broadcast;
2. While some rows remain unprocessed
    2.1. All ISs with unprocessed PEs will activate the PEs in their next unprocessed row
         and deactivate the PEs in their other rows;
    2.2. Each active IS instructs any PE needing to broadcast a value to its row to place
         this value on its IS broadcast network.
End.
```

The broadcast operation within a row is completed in constant time. The simulation of broadcast operations along column buses is done analogously. The only difference is PEs switch to their assigned column IS. Each of the steps in the above loop takes O(1) time, and the loop has $\lfloor \sqrt{n} / j \rfloor + 1$ iterations. Thus, we have the total running time to be $O(\sqrt{n} / j)$. Each PE needs two additional local variables to store its row and column indices. The resulting increase in memory for all PEs is O(n), which is a minor cost. Another additional cost is the j instruction stream processors that provide instructions to the PEs. However, since j is usually expected to be much smaller than n, the number of PEs, it is also an insignificant cost in the simulation. Therefore, the result is summarized as the following theorem.

Theorem 5.2. MASC (n, j) with a $\sqrt{n} \times \sqrt{n}$ mesh can simulate a $\sqrt{n} \times \sqrt{n}$ MMB with a running time $O(\sqrt{n}/j)$.

Notice that a constant time simulation can be obtained when j is $\Omega(\sqrt{n})$. It indicates that the simulation with this particular MASC model establishes that an algorithm running on a $\sqrt{n} \times \sqrt{n}$ MMB can be executed on MASC(n, j) with a 2-D mesh in asymptotically the same time when j is $\Omega(\sqrt{n})$. As a result, MASC (n, j) with a 2-D mesh is at least as powerful as a $\sqrt{n} \times \sqrt{n}$ MMB, when j is $\Omega(\sqrt{n})$. In particular, many cost optimal algorithms that have been designed for MMB are also cost optimal when executed through simulation on MASC. This raises the question as to whether this particular MASC model has the same power as the MMB model or is strictly more powerful. This question is answered by the following example.

Example 5.1 There is a problem which can be solved in constant time using the MASC$(n,\sqrt{n})$ with a $\sqrt{n} \times \sqrt{n}$ mesh that requires $\Omega(n \log n)$ time for the $\sqrt{n} \times \sqrt{n}$ MMB.

Consider a $\sqrt{n} \times \sqrt{n}$ table and various partitions of these values into $\sqrt{n}$ sets, with $\sqrt{n}$ values, each of which contains exactly one value from each column and one value from each row of the table. An example of such a partition can be obtained using the wrap-around diagonals of this table. The problem is to efficiently calculate the maximum value for each set in any partition of this table. With the MASC$(n,\sqrt{n})$ model, the n numbers of this table can be stored in n PEs with one value in each PE. Then all PEs with data for the ith set listen to IS$_i$ $(1 \le i \le \sqrt{n})$ and locate the maximum value for that set in O(1) time. However, we can not solve the same problem with a $\sqrt{n} \times \sqrt{n}$ MMB in time less than $\Theta(\sqrt{n} \log n)$. This is because it requires $\Omega(\sqrt{n})$ time in the worst case to move the data so that all the data for each of the $\sqrt{n}$ sets are grouped together in a common row (or equivalently, a common column). Moreover, for $\sqrt{n}$ values stored one value per processor on a row (or a column) of a $\sqrt{n} \times \sqrt{n}$ MMB, the time lower bound to find the maximum of them is $\Omega(\log n)$ [71]. Thus, we have the following theorem.

Theorem 5.3. MASC (n, j) with a $\sqrt{n} \times \sqrt{n}$ mesh is strictly more powerful than a $\sqrt{n} \times \sqrt{n}$ MMB when $j = \Omega(\sqrt{n})$. Any algorithm for a $\sqrt{n} \times \sqrt{n}$ MMB can be executed on MASC(n, j) with $j = \Omega(\sqrt{n})$ and a $\sqrt{n} \times \sqrt{n}$ mesh with a running time at least as fast as the MMB time.

The above example can be generalized to a case when the mesh size is a rectangle both for MASC and MMB rather than a $\sqrt{n} \times \sqrt{n}$ square. A $m \times n$ table can be arbitrarily partitioned into m sets. Each set contains n values that are selected in a way of exactly one from each column and one from each row of the table. A MASC(mn, j) when j is $\Omega(\max\{m, n\})$ can calculate the maximum of all values in constant time. However, a $m \times n$ MMB requires $\Omega(\max\{m,n\}\log(\min\{m,n\}))$ time where $\Omega(\max\{m,n\})$ for data movements and $\Omega(\log \min\{m, n\})$) for finding the maximum of all values [71]. In a similar way, we can analyze a relationship between MASC(cn, 1) with a $n \times c$ mesh connection and a $n \times c$ MMB. Thus, we obtain the following corollaries.

Corollary 5.4 MASC(mn, j) with a $m\times n$ mesh connection is more powerful than a $m\times n$ MMB, when j is $\Omega(\max\{m, n\})$. Any algorithm on a $m\times n$ MMB can be executed on MASC(mn, $\Omega(\max\{m, n\})$) with a $m\times n$ mesh connection with a running time at least as fast as on the MMB.

Corollary 5.5 For any constant c, MASC (cn, 1) with a $n \times c$ mesh connection is more powerful than a $n \times c$ MMB. Any algorithm on MASC(cn, 1) with a $n \times c$ mesh connection can be executed on a $n \times c$ MMB with a running time that is at least as fast.

5.4. Simulating BRM with MASC

We now discuss the simulation between the MASC model and the BRM, a special case of general reconfigurable meshes. Consider a $\sqrt{n} \times \sqrt{n}$ BRM and MASC(n , j) with a $\sqrt{n} \times \sqrt{n}$ mesh connection. We first present the simulation algorithm of the BRM with a particular model MASC(n, j) when $j = \sqrt{n}$. Then we extend the simulation to the general case with arbitrary j.

Again, we view the desired simulation as a sequence of steps in which each PE in MASC is responsible for simulating one of the BRM processors. Since both models have a 2-D mesh connection, each processor in both models has a row and column number that correspond to their position in the mesh. The MASC PE at location (r_i , c_i) in the mesh simulates the PE in the same position in the BRM. In particular, each MASC PE$_i$ simulates the BRM-processor P(r_i , c_i), where $i = (r_i - 1)\sqrt{n} + c_i$. The value r_i and c_i are $r_i = \lceil i / \sqrt{n} \rceil$, $c_i = (i-1) \bmod \sqrt{n} + 1$ and PE$_i$ stores them in the variables *row$* and *column$*.

Besides the above parallel variables *row$* and *column$,* several other MASC variables will be needed. The parallel variable, *connection$* stores the reconfiguration status of the corresponding BRM processor P(r_i , c_i). In particular, during parallel row operations, the variable *connection$* in PE$_i$ stores a 0 when the corresponding BRM processor P(r_i , c_i) has its {E,W} ports connected and stores a 1 otherwise. The MASC parallel variable *leader$* stores the column (or row) number of the leftmost (or topmost) BRM processor which shares the same subbus with the BRM processor corresponding to PE$_i$, when connected in EW (or NS) direction. All the values in this parallel variable are initialized to 0 before each subbus broadcasts. Let p_i be a scalar IS variable for each IS$_i$ that stores the column (or row) number of a specific PE in the subbus that is currently being processed. Also, let *leader*$_i$ be a scalar variable that stores the *leader$* value of the subbus currently being processed.

All PEs on the same subbus will have the same *leader$* value with one exception. This exception occurs with horizontal subbuses when two subbuses terminate at a common PE, as illustrated by P(3,2) in Figure 5.1. In this case, the common PE will have the *leader$* value that identifies its right subbus and will be the leader of this subbus. However, since it belongs to two subbuses, it needed to have two leader$ values. An analogous situation occurs for vertical subbuses. To simplify the description of the algorithm involving BRM, we assume that there are no two horizontal (or vertical) subbuses share a common PE. However, the algorithm techniques can be extended to also work when this special case is allowed.

Now we need to deal with the simulation of broadcasting in BRM. Since there are $\sqrt{n}$ ISs assumed, let IS$_i$ be assigned to the ith row and the ith column. Initially, all PEs listen to IS$_1$. They can switch to their row or column IS in one step. Also, in one step, an IS can switch from instructing its row PEs to instructing its column PEs. Recall that in one time unit, a BRM processor can connect one opposite pair of ports, namely {N,S} or {E,W}. Since the algorithms for broadcasting on the row and column subbuses are similar, we will present only the algorithm for the row (or horizontal). This algorithm is given next and followed by a discussion of it.

Algorithm 5.2. {Broadcasts along row subbuses}

Begin

{Part 1–preprocessing when a switch is reconfigured.}

1. All ISs activate all their PEs and instruct them to listen to their row IS (e.g., PE$_i$ switches to ISr_i).

2. Each IS$_i$ has all its PEs to set *connection$* to 0 if BRM-processor P(r_i, c_i) has its EW ports connected and to 1 otherwise.
3. All ISs have their PEs do steps 4 -10 in parallel.
4. Store 0 in *leader$*.
5. Send its *connection$* value to its horizontal neighbors using mesh links.
6. If a PE does not receive a *connection$* value of 0 from one of its neighbors, it becomes inactive.
7. While there is an active PE with *connection$*=1, all ISs with an active PE with *connection$*=1 execute steps 8-9 in parallel
8. Set p_i to the maximum of the column numbers of its PEs with *connection$*=1.
9. Instruct all of its active PEs with *column$* $\geq p_i$ to store p_i in *leader$* and then become inactive.
10. For each *i*, if PE(*i*, 1) is active, IS$_i$ instructs all its active PEs to set their *leader$* to 1.

{Part 2 – simulate a broadcast.}

11. All IS$_i$ activate only those PEs that are on their subbuses.
12. All IS$_i$ instruct their PEs to set their broadcast flag if they wish to broadcast.
13. While there is an active PE, all ISs with an active PE execute steps 14-18 in parallel.
14. Set p_i to the maximum of the column numbers of its active PEs with a broadcast flag set.
15. Deactivate all PEs whose column number is greater than p_i.

16. Read *leader$* and value *v* from the PE in column p_i, and store this value of *leader$* in $leader_i$.

17. Activate only the PEs whose *leader$* value is $leader_i$ and broadcast *v* to these PEs.

18. Return to the previous set of active PEs and deactivates those PEs in whose *leader$* is $leader_i$.

End.

In part 1 of Algorithm 5.2, each PE_i initially switches and listens to its row IS. Each PE_i sets its *connection$* variable to 0 if the BRM-processor $\text{P}(r_i, c_i)$ it simulates has its connection set to {E, W} and to 1 otherwise. Next, all PEs store 0 in their *leader$* variable. Local links of the mesh connection are used in step 5 to check the connection status of their horizontal neighbors. All those PEs that do not receive a value of 0 from one of its horizontal neighbors are deactivated since they are not on a horizontal subbus. The loop in steps 7-10 first identifies the left-most PE in each horizontal subbuses. This PE is called the leader of the subbus, and then its column number is broadcast to all PEs in the subbus and is stored in the *leader$* variable in these PEs.

In step 12 in part 2 of this algorithm, the PEs that are on subbuses are activated and the rest remain inactive. These PEs are precisely the ones with a nonzero value in their *leader$* variable. Next, those PEs that wish to broadcast are instructed to set their broadcast are instructed to set their broadcast flag. In the loop in steps 15-19, the rows are processed in parallel but the broadcasts of subbuses in the same row are processed sequentially from the right to the left.

Next, consider the BRM example shown in Figure 2.3. After step 6 in Algorithm 5.2, the values in *connection$* listed in row-major order are {0, 0, 0, 0, 0, 0, 1, x, 0, 1, 0, 0, x, x, x, x} in which x indicates the corresponding PE has been set inactive. After step 11, we have *leader$* = {1, 1, 1, 1, 1, 1, 1, x, 1, 1, 2, 2, x, x, x, x}, so there are four horizontal subbuses available at this point. Any processor on a subbus may broadcast a value that can be read by all other processors on the same subbus.

In part 1 of the algorithm, there is a loop for each IS_i to assign a leader for each of its subbuses. This takes $O(\sqrt{n})$ in the worst case. The other operations of switching, activation, deactivation and finding the maximum take constant time. Local communication between neighbors in mesh connection takes constant time as well. In part 2, once the leader of a subbus has been found, a broadcast over the subbus takes constant time. However, if there are multiple subbuses in one row and each of the subbuses has a broadcast request simultaneously, the row IS must handle these requests sequentially. Since each row can have $O(\sqrt{n})$ broadcast requests, this operation requires $O(\sqrt{n})$ time in the worst case. The resulting worst case simulation time is $O(\sqrt{n})$. As for extra memory, each PE has three extra local variables to store the data needed in the simulation, so the total extra memory used is $3n$ or $O(n)$ for all PEs, which is insignificant. The other cost is that we need $\sqrt{n}$ extra processors as ISs in MASC which are not needed in BRM. However, we argue that $\sqrt{n}$ is asymptotically less than the total number n of PEs, so this cost can also be considered as an insignificant cost in the simulation. As mentioned before, in practice, the self-simulation results in [118,119] can be used reduce the $\sqrt{n}$ instruction streams to a smaller number, due to the scalability of the MASC model. Obviously, the preceding algorithm can be executed by a MASC(n, j) model when j is $\Omega(\sqrt{n})$. This yields the following theorem.

Lemma 5.6. MASC(n, j) with mesh connection where $j \in \Omega(\sqrt{n})$ can simulate a $\sqrt{n} \times \sqrt{n}$ BRM in $O(\sqrt{n})$ time with $O(n)$ extra memory.

Now it is easy to extend the result to a general case MASC(n, j) with an arbitrary j. If $j \le \sqrt{n}$, we make the same assignment to multiple rows (columns) as in Section 5.3. Each IS_i, instead of processing only one row (column), now sequentially processes multiple rows (columns), i.e., row (column) i is assigned to IS_k where $k = (i \bmod j) + 1$ for $1 \le i \le \sqrt{n}$. All j ISs execute Algorithm 5.2 while activating the PEs on the first assigned rows (columns) with the other assigned rows (columns) deactivated. Then next all j ISs execute Algorithm 5.2 on their second assigned rows (columns) with the other assigned rows (columns) deactivated. This is continued until all rows (columns) are processed after $\lceil \sqrt{n} / j \rceil$ iterations. Clearly, the total running time is $\lceil \sqrt{n} / j \rceil * O(\sqrt{n})$ or $O(n / j)$ in the worst case. Notice there is no additional memory requirement for PEs. Therefore, combing Lemma 5.6, the following theorem is given.

Theorem 5.7. MASC(n, j) with a 2-D mesh connection can simulate a $\sqrt{n} \times \sqrt{n}$ BRM with $O(n / \min\{\sqrt{n}, j\})$ time and $O(n)$ extra memory.

5.5 Simulating MASC with enhanced meshes

In this section, an algorithm is given to simulate MASC(n, j) with enhanced meshes. This provides a useful additional information for comparing the power of these two models.

The MMB model is considered first. We make the same assumption for the size of the models and mapping for processors between the two models as before. Matching processors have the same location on the $\sqrt{n} \times \sqrt{n}$ mesh. Hence, each P(r_i , c_i) in the MMB corresponds PE$_i$ in the MASC, where $r_i = \lceil i / \sqrt{n} \rceil$ and $c_i = (i - 1) \bmod \sqrt{n} + 1$. To simulate j ISs in MASC, each of the $\sqrt{n}$ MMB-processors in the first column, i.e., P(i, 1) for $1 \le i \le \sqrt{n}$, is used to simulate a MASC IS. A copy of the program is stored on each P(i, 1) for $1 \le i \le j$. The MMB simulates the execution of the instruction of the ISs sequentially. The MMB processor P(i, 1) broadcasts an IS$_i$ instruction to all the processors in the first column. Next, this instruction is broadcast along each row to the remaining MMB processors. Each MMB processor allocates register (or memory) space for the integer variable *channel* and the boolean variable *active*. The ID for the IS that the simulated MASC-PE is listening to is stored in *channel* and whether or not this MASC-PE active is stored in *active*. Each MMB processor checks these variables in order to decide whether to execute the current instruction. Initially, all PEs listen to IS$_1$, which is simulated by P(1, 1).

The MMB processor executes a local computation or memory access in one step exactly as a MASC processor does. Also, a 2-D mesh data movement instruction by a MMB processor is executed exactly as it is by a MASC processor. However, more work is required for the MMS simulation of the MASC global reduction operations of OR, AND, maximum, and minimum. When a MMB processor receives an instruction operation for one of these reduction operations from an IS, e.g. IS$_i$, it does nothing at this step provided it currently is assigned to IS$_i$ and is active. Otherwise, it determines the null value for this reduction operation and prepares to use this value in the reduction operation. The null values are as follows: 0 for OR, 1 for AND, MININT for maximum, etc. Next, the optimal MMB algorithm provided in [27] is used to compute the reduction and get the value in P(1, 1). The final step is to send this result from P(1, 1) to P(i, 1), the processor simulating IS$_i$, using the first column bus. The above discussion is summarized in the following algorithm.

Algorithm 5.3 {simulation of MASC(n, j) with a $\sqrt{n} \times \sqrt{n}$ mesh using a $\sqrt{n} \times \sqrt{n}$ MMB}
Begin

1. For $i = 1$ to j , each IS$_i$ execute its current command using steps 2-9.

2. P(i, 1) broadcasts the next IS$_i$ instruction along the first column bus.

3. All P(k, 1) ($1 \le k \le \sqrt{n}$) broadcast this IS$_i$ instruction to the kth row in parallel.

4. If the IS$_i$ instruction is a local operation, by checking the variables *channel* and *active*, those processors that are currently listening to IS$_i$ and are active execute this operation.

5. If the IS_i instruction is a data movement operation on the MASC 2-D mesh, the MMB 2-D mesh executes this operation.
6. If the IS_i instruction is a reduction, all processors do steps 7-9 in parallel.
7. If a processor is either not assigned to IS_i or is not active, then it determines the null value for this reduction to use in the next step.
8. The MMB executes the reduction operation using the algorithm in [27] and place the result in P(1,1).
9. P(1,1) uses the first column bus to send this reduction result to P(i,1).

End.

The slowest part of this algorithm is the steps of simulating a MASC reduction operation. According to [27], this reduction operation can be performed optimally on the MMB in $O(n^{1/6})$. Since this occurs inside a loop that is executed j times, the worst case time will be $O(jn^{1/6})$. For the case where $j = \sqrt{n}$, the worst case is $O(n^{2/3})$. Similarly, if $j = \sqrt[3]{n}$, then the worst case is $O(\sqrt{n})$. There are two extra variables used by each MMB processor to decide whether the processor executes the current instruction and what value should be provided to the instruction. Additionally, the MMB processors in the first column require extra memory to store a copy of the program and to execute this program, which is constant length. The total extra memory used is $O(n)$.

Observe that an alternative to using the first column of MMB-processors to simulate ISs in MASC is to use a $(\sqrt{n}+1) \times \sqrt{n}$ MMB in which the processors in the first column are used only to simulate ISs of MASC. This will eliminate the load imbalance on the processors in the first column.

Since the BRM is more powerful than the MMB, it can also execute the above simulation of MASC with a 2-D mesh. As illustrated in Example 5.1 in Section 5.3, problems can easily be chosen which do not allow effective uses of BRM subbuses. Since the primary use of BRM presently is to provide a model intermediate in power between the MMB and the reconfigurable mesh, a BRM reduction algorithm that is faster than an optimal MMB reduction algorithm is not known currently. We have established the following theorem.

Theorem 5.8. MASC(n, j) with a $\sqrt{n} \times \sqrt{n}$ mesh connection can be simulated by a $\sqrt{n} \times \sqrt{n}$ MMB or BRM with O($jn^{1/6}$) time and O(n) extra memory.

5.6. Simulations between MASC and RM

A reconfigurable mesh (RM) is a very powerful parallel computational model [25,69,85,86]. A 2-D reconfigurable mesh was shown to be at least as powerful as the CRCW PRAM model in [127]. Later, it was proved to be strictly more powerful than the CRCW model in [91]. These indicate that simulation of the reconfigurable mesh with MASC can be potentially a significant tool to evaluate the power of the MASC model.

RM is a general case of BRM. RM allows up to two disjoint pairs of ports to be connected. We first consider simulation of RM using MASC.

Any simulation of a $\sqrt{n} \times \sqrt{n}$ RM with MASC(n, j) with a $\sqrt{n} \times \sqrt{n}$ mesh must also simulate the BRM since RM extends the capabilities of the BRM. Therefore, a natural way to simulate RM using MASC is to extend earlier MASC simulation of BRM. To simulate a $\sqrt{n} \times \sqrt{n}$ RM with MASC(n, j) with a $\sqrt{n} \times \sqrt{n}$ mesh, we go through the steps as in the simulation of a broadcast on BRM with MASC, i.e., processing multiple rows (or columns) in parallel while processing PEs within a row (or column) in sequential. Assume the RM has configured k buses in order to execute simultaneous broadcasts. (A RM bus is conceptually equal to a BRM subbus that we used in earlier discussion.) To simulate these broadcasts, MASC creates k subsets of PEs corresponding to the processors in each RM bus. The k subsets are assigned to ISs in order and each IS has $\lceil k/j \rceil$ at most subsets of PEs. Each IS performs a broadcast for these subsets sequentially. All ISs execute in parallel. Since a $\sqrt{n} \times \sqrt{n}$ RM can configure as many as O(n) buses (e.g., each of the buses consists of constant number of processors), the worst case time to simulate a broadcast of the RM with the MASC is O(n/j). Trivially the best case for simulating RM buses takes constant time and occurs when the RM has configured at most j buses.

The reverse direction of simulating MASC with a RM that has the same number of processors is not as easy as might be expected when the power of RM is considered. This is because that MASC has extremely high flexibility of letting all PEs be partitioned to disjoint sets in an arbitrary way. An IS can instruct a set of PEs to execute commands in parallel. This is especially useful when we solve problems with irregular data sets that cannot be efficiently constructed to buses by RM. We next pose two open problems. The claims made in these problems appear to be true and are left for future work. The focus of both problems is to establish that there are broadcasts that MASC can handle in constant time but which RM can not handle efficiently.

Problem 5.1

Given a $\sqrt{n} \times \sqrt{n}$ mesh of processors labeled with $P_1, P_2, \ldots, P_n$ and positioned in the row-major order. Let k be an integer with $\sqrt{n}/2 < k < \sqrt{n}$. We may construct k subsets in the following way. From P_1, we pick up all processors in every k processors to make the first subset; From P_2, we pick up all processors in every k processors to make the second subset; and so on. Then we have all processors that are partitioned into k subsets. Within each of the subsets, every processor is separated from the next one by k

positions. We consider the problem of performing a broadcast within each of the subsets. This problem is easy for MASC(n, k). We assign all PEs with the ID number P_1 , P_{1+k} ,…, $P_{1+\lfloor n/k \rfloor k}$ to IS_1 ; assign all PEs with ID number P_2 , P_{2+k} , …, $P_{2+\lfloor n/k \rfloor k}$ to IS_2 ;…; and assign P_k , P_{2k} …, $P_{\lfloor n/k \rfloor (k+1)}$ to IS_k . MASC(n, k) can then broadcast to all k subsets in parallel in constant time. For MASC(n, j) with an arbitrary number j of instruction streams, the time is $O(k/j)$ or $O(\sqrt{n}/j)$ in the worst case. The following observations appear to be true. Without overlapping, there is no effective way to configure a $\sqrt{n} \times \sqrt{n}$ RM with k buses such that each bus consists of processors of a subset as described above. A RM bus allows only one data value to be broadcast at one time unit on each bus. If there are multiple simultaneous broadcasts with each on one of the overlapped buses, each of the overlapped buses must perform a broadcast sequentially in order to use the overlapped paths alternatively. When P_k is in the middle of $P_{\sqrt{n}/2}$ and $P_{\sqrt{n}}$, or $P_{3\sqrt{n}/4}$, the RM requires $O(\sqrt{n}/4)$ or $O(\sqrt{n})$ time to complete a broadcast for all subsets.

Problem 5.2

Given an $n \times (1+\log n)$ mesh of processors, we may construct subsets of this mesh based on the butterfly topology. Each subset consists of all processors on a path of the butterfly connection. This results in n subsets with each containing $1+\log n$ processors. The problem is to perform a broadcast within each of the subsets. Clearly, MASC($n(1+\log n)$, j) can perform this task in $O(n/j)$. When $j=n$, this is a constant time task. However, our following observation also appears to be true. Since a same size RM cannot configure such buses to perform broadcasts without overlapping, it cannot complete broadcasts for all subsets in constant time. An 8×4 array of processors connected with the butterfly topology is shown in Figure 5.2.

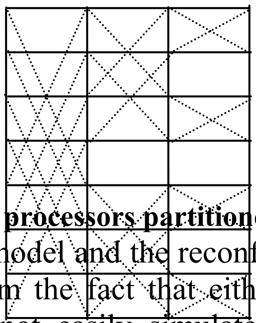


Figure 5.2 An 8 × 4 mesh of processors partitioned in the butterfly topology

We observe that the MASC model and the reconfigurable mesh cannot efficiently simulate each other. This stems from the fact that either of the two models possesses some properties that the other cannot easily simulate. The MASC model has great flexibility to partition all PEs in an arbitrary manner. One IS can control the parallel execution of the PEs in each set of this partition. In the RM model, configuration of buses is restricted by the fact that at most two buses can pass through each processor. As a result, the RM cannot use buses to efficiently support communication between the processors in each of the partition sets for many nonstandard partitions. This results in algorithms which execute the processors in a partition set in parallel but execute some of the tasks performed by the partition sets sequentially, which degrades the parallelism of the RM model.

On the other hand, the maximum number of different groups of PEs that MASC(n, j) can execute simultaneously and use there is for data movements is j. In contrast, the RM can easily support O($\sqrt{n}$) buses (e.g., columns) and support as many as O(n) buses (e.g., each consisting of a constant number of processors). While all of these buses are executing the same algorithm steps simultaneously, each can support both simultaneous execution and data movement for their set of processors. Therefore, in practice, we expect the number of "restricted" instruction streams in RM to be much larger than the number of instruction streams in MASC.

5.7 Summary

Three simulation algorithms to simulate between two classes of enhanced meshes, MMB and BRM, and the MASC model have been presented in this chapter. Simulations between MASC and RM, a general case of BRM, have also been discussed.

We have shown that MASC(n, j) with a 2-D mesh can simulate a $\sqrt{n} \times \sqrt{n}$ MMB in O($\sqrt{n} / j$) time with no extra memory. In particular, the simulation takes constant time when $j = \Omega(\sqrt{n})$. Also we show that MASC(n, j) with a 2-D mesh can simulate a $\sqrt{n} \times \sqrt{n}$ BRM in O(n / j) time with O(n) extra memory. Simulation of MASC(n, j) with a $\sqrt{n} \times \sqrt{n}$ MMB or BRM takes O($jn^{1/6}$) time with O(n) extra memory. Also, it is shown that MASC(n, 1) with a 2-D mesh is more powerful than a $\sqrt{n} \times \sqrt{n}$ mesh with a global bus, and that MASC(n, j) with a 2-D mesh is also more powerful than a $\sqrt{n} \times \sqrt{n}$ MMB when $j = \Omega(\sqrt{n})$. These results provide effective ways in understanding the power of the

MASC model by comparing it with enhanced meshes that are currently very popular parallel computation models in research literature. Also, the constant time simulations enable the algorithms designed for enhanced meshes to be transferred to the MASC model with the same running time.

Discussion on simulations between MASC and RM has revealed the dissimilarity of the two models. Neither of them can simulate the other efficiently. It has been shown that they have distinct features that might be good at solving different problems, but not both.

There are several interesting open problems related to this work, including proof of optimality of the simulation algorithms given in this chapter. One important open problem is to provide an example of a problem that is easy for MASC to solve but difficult for RM. Such a problem would establish a lower bound for a simulation of MASC by a RM. An example of this type could be provided by establishing either of the conjectures posed as problems in Section 5.6. Other open problems involve finding additional simulations between MASC and RM. For example, one such problem is to generalize all of our simulations between MASC and RM models to general meshes of size $m \times k$. Another such problem is to investigate simulations of MASC and RM models of different sizes.

Chapter 6

Associative Computing in Real-time Processing

In the following two chapters, we discuss how to apply the MASC model with one instruction stream (also called the ASC model for Associative Computing) in the area of real-time processing. In particular, we take the Air Traffic Control (ATC) system as a typical real-time problem. For over 40 years, extensive effort has been devoted to finding an efficient MIMD solution to the ATC problem. It is currently widely accepted that large real-time problems such as ATC do not have a polynomial time MIMD solution. However, using ASC, an efficient polynomial time solution to the ATC problem can be obtained. In addition, associative computers (also called associative SIMD computers) have been built which support the associative model and can support this polynomial time solution to the ATC and meet the required deadlines. These observations raise serious questions about the validity of the common belief that the MIMD model is more powerful than the SIMD model. We are motivated to investigate this situation further and reevaluate the power of SIMD computation compared to that of MIMD computation. This chapter primarily discusses the ASC solution to the ATC problem, a large portion of which has been presented in [82,83,84]. Some real-time definitions are given in Section 6.2. Section 6.3 describes the ATC problem including assumptions, constraints of the system and examples of certain ATC tasks. Section 6.4 presents a polynomial time ASC solution to the ATC. Section 6.5 reviews some complexity results and current practice regarding the ATC problem. Section 6.6 gives a summary of the results of this chapter.

6.1 Introduction

Flynn's taxonomy for parallel computers based on the numbers of instruction streams and data streams has been widely used in the research literature of parallel processing since it appeared in 1972 [41]. SIMDs (Single Instruction-stream Multiple Data-stream) and MIMDs (Multiple Instruction-stream Multiple Data-stream) are the most important categories of computer systems for parallel computation. Most early parallel computers had a SIMD-style design. A SIMD computer has a central controller that broadcasts commands to multiple processors. The commands are executed synchronously on the set of processors. On the other hand, with MIMD architectures, each of processors has its own program and executes at its own pace. Processors exchange information by shared memory or messages passing. Due to their flexibility in taking advantages of high-speed off-the-shelf microprocessors, MIMD computers are generally believed to be more powerful and more performance/cost effective compared with SIMD computers. Therefore, current research in various application areas including real-time systems has put a great deal of emphasis on use of MIMDs.

Here we will follow the common practice in real-time computing [111] and complexity results for parallel computation [46] of using the term multiprocessors (or MP) for asynchronous computers. Therefore, the terms of multiprocessors and MIMDs are interchangeable in this work. It should be observed that this usage of multiprocessors is in conflict with the usage of some other books and papers [103], where this term is referred to MIMD computers with shared memory.

The Air Traffic Control (ATC) system is a typical real-time system. Tasks in an ATC system are time critical for obvious reasons. Use of the multiprocessor approach for ATC has been widely explored for over 40 years. The FAA (Federal Aircraft Administration) has spent a great deal of effort on finding a predictable and reliable system to help the country to achieve free flight. This would allow pilots to choose the best (e.g. shortest) path rather than following pre-selected flight corridors. However, all results so far have been unsatisfactory. To a large degree, this is due to the fact that the software used by multiprocessors in solving real-time problems involves solutions to many difficult problems such as dynamic scheduling. Numerous complexity results have established that almost all real-time task scheduling problems on multiprocessors are NP-hard [46,108,110,111]. This means no tractable (e.g., polynomial time) multiprocessor solution to a real-time problem is expected if this solution has to handle scheduling of real-time tasks. Even though many researchers have put extensive efforts into producing a reasonably good (e.g., using heuristic algorithms) solution to the ATC problem, these efforts have been disappointing. The nation's ATC system using multiprocessors has repeatedly failed to meet the requirements of the ATC automation system. One such project was canceled after ten years [81,82,102]. It is frequently been reported that the current ATC system periodically loses radar data without any warnings and misses tracking of aircraft in the sky. This problem has even occurred with Air Force One [120].

On the other hand, our solution provides a completely different approach to this real-time processing problem. The ASC model is an associative SIMD model and a computer that supports this model is called an Associative Processor (AP) or associative SIMD computer. Instead of using a multiprocessor approach that has been shown to be theoretically and practically difficult, we provide a simple solution using an associative SIMD computer. The efficiency and predictability of this approach have been established in previous simulation demonstrations for this problem.

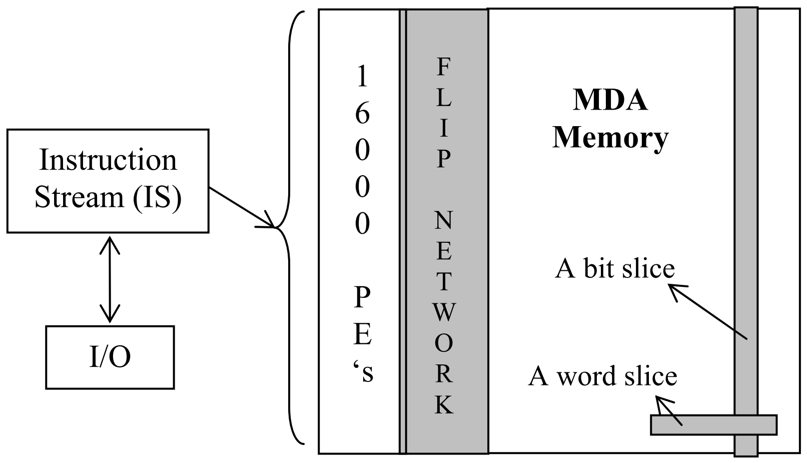


Figure 6.1 A possible AP architecture for ATC

Figure 6.1 indicates the AP architecture that is assumed for the ATC solution. The AP provides a new dimension of memory access. Instead of accessing 32 bits in one memory cycle, an AP with 16,000 PEs (or ALUs as we sometimes call them) can access 16,000 bits at one time. Traditional SIMDs cannot handle the ATC requirements because of I/O limitations. This I/O limitation is overcome by the MDA (Multidimensional Access) memory [18,20,107] of Figure 6.1. The flip network is a corner turning network that provides access to a bit slice of data in the normal SIMD mode, or a word slice of data for I/O [18,21,98]. The word direction of data is compatible with the concept of entries of a relational database record. These features of the AP architecture exist in both the STARAN and ASPRO machines [18, 74].

6.2 Real-time processing

A real-time system is distinguished from a traditional computing system in that its computations must meet specified timing requirements. A real-time system executes tasks to ensure not only their logical correctness but also their temporal correctness. Otherwise, the system fails no matter how accurate a computation result is.

As conventionally defined [111], a real-time *task* is "an executable entity of work that, at minimum, is characterized by a worst-case computation time and a time constraint or deadline". A *job* is "an instance of a task or an invocation of a task". When a task is invoked, a job is generated to execute this task with the given conditions. The *release time* of a task is the time that the task (or a job of the task) is ready to be executed. A real-time task can be *periodic,* which is activated (released) in a regular interval (period); a*periodic*, which is activated irregularly in some unknown and possibly unbounded interval; or *sporadic*, which is activated irregularly in some known bounded interval. Typically, real-time scheduling can be static or dynamic. *Static scheduling* refers to the case that the scheduling algorithm has complete knowledge a priori about all incoming tasks and their constraints such as deadlines, computation times, shared resource access, and future release times. In contrast, in *dynamic scheduling*, the system has knowledge about the currently active tasks but it does not have knowledge about new task arrivals. Thus the scheduling algorithm has to be designed to change its task schedule over time.

We introduce a new term that is called *jobset* for the AP. Observe that an AP is a set processor, as each of its instructions operates on a set of data. As a result, the AP can execute a set of jobs involving the same basic operations on different data simultaneously. This set of jobs is called a jobset. Compared with the common understanding of a job as an instance of a task in the MP, in the AP, a task is considered to be a sequence of jobsets.

For a specific real-time system like ATC, some parameters are defined. A *system period P* is the time during which the set of all tasks must be completed. The *system deadline D* is the constraint time for a system period. A *task deadline d* is the time constraint for an individual task. Deadlines can be hard, firm, or soft. A *hard deadline* is a time constraint for a task that failure to meet it will result in disastrous consequences. A *firm deadline* is a time constraint for a task that failure to meet it will produce useless results but cause no severe consequences. A *soft deadline* is the time constraint for a task that the result produced after this deadline will be degraded but may be still useful in a limited time period. Hard deadline tasks are critical while soft deadline tasks are non-critical. For the ATC system it is obvious that missing deadlines is not tolerable. Therefore we consider only hard deadlines in our work. According to [111, static scheduling], a static schedule can be used if the summation of task times for the system period is less that the system deadline D, otherwise a static schedule is infeasible.

A real-time system can be built on a uniprocessor or on a multiprocessor. Given a set of real-time tasks, a primary concern for a real-time system is how to schedule these tasks to ensure they meet various constraints including deadlines. Depending on different criterion such as minimizing the sum of all task computation times and minimizing the maximum lateness (the time elapse between a task release and its execution), different scheduling algorithms can be designed. An *optimal* scheduling algorithm is one that may fail to meet a deadline only if no other scheduling algorithm can [111]. With a uniprocessor system or a sequential processor, it has been shown that there exist optimal scheduling algorithms. For example, the *Earlier Deadline First* (EDF) algorithm and *Least Laxity First* (LLF) are optimal for scheduling a set of independent real-time tasks on a uniprocessor [110]. (Here *laxity* is deadline minus computation time minus current time.) In certain circumstances, the *Rate-Monotonic* (RM) scheduling algorithm has also been shown to be optimal [65, 73]. However, as real-time systems become larger and tasks become more sophisticated, real-time processing has become much more dependent on parallel systems. In other words, using more than one processor to execute tasks simultaneously seems to be mandatory. Unfortunately, this is not a straightforward change, because of the long-established theory that almost all real-time scheduling problems using multiprocessors are NP-complete [46,108,110]. With the multiprocessor system, there is no optimal scheduling algorithm has been discovered for almost all cases. Researchers have been driven to using heuristics to design scheduling algorithms on a multiprocessor. However, these heuristics are usually expensive in that they consume a lot of computation time, and sometimes additional hardware support such as a scheduling chip is needed. Moreover, use of heuristics result in systems that in are inherently unpredictable.

6.3. Air Traffic Control (ATC)

An ATC system is a real-time system that continuously monitors, examines and manages space conditions for thousands of flights by processing a large amount of data that is dynamically changing due to reports by sensors, pilots, and controllers. The ATC software consists of multiple real-time tasks that must be completed in time to meet the individual deadlines. By performing various tasks, the ATC system keeps timely correct information concerning positions, velocities, contiguous space conditions, etc., for all flights under control. Since any lateness or failure of task completion could cause disastrous consequences, time constraints for tasks are critical.

Data inputs come mainly from sensor reports and are stored in a real-time database. In our working prototype of the ATC system, a set of sensor reports or track reports arrive every 0.5 seconds. Multiple sensors may return redundant data on some aircraft. All tasks must be completed before new sensor data arrive for the next cycle.

In order to present our work more clearly, we briefly describe our analytical prototype of the ATC system and ATC tasks in this section. System constraints and ATC task characteristics are listed first, then ATC tasks and task examples are explained. A worst case ATC environment that is based on real world facts is also given. Some details have been discussed in [82, 83, 84].

6.3.1 System constraints

Besides time constraints, a real system may have other constraints. These are resource constraints, precedence constraints, fault tolerance, and communication requirements. They specify limitations of the system in different aspects. For the ATC system, we consider the following features.

- Resource constraints – an ATC task may require access to certain resources other than the processing system, such as I/O buffers which retain information for non-conflicting use by scheduled tasks at each task release time.
- Precedence constraints – tasks are strictly scheduled in precedence order (This is also EDF order) so that no more precedence concerns are involved in task releasing and execution.
- Concurrency constraints – tasks cannot obtain concurrent access to resources if there is a single instruction stream to control the tasks.
- Communication requirements – a system that has distributed elements requiring bi-directional communication paths is not considered.
- Fault tolerance – when multiple instances of a task are executed for fault-tolerance, the different instances are executed simultaneously on different processor systems.

6.3.2 Task characteristics

Besides time constraints, there are other assumptions for ATC tasks in our ATC system. Before specific ATC tasks are described in Section 6.3.4, we give general characteristics of an ATC task.

- All tasks are periodic. Although each task has its own deadline, all of them must be completed by the system deadline *D*, or the system period *P*.
- All deadlines are known at the task release time.
- Aperiodic jobs are handled by a special task in a particular time slot in every cycle.
- Tasks are independent in that there is no synchronization between them nor shared resources. The static schedule fixes release times and deadlines for each task.

- Overhead costs for interrupt handling are included in each task cost.
- Task execution is non-preemptive.
- Task deadlines are all hard and critical. None can be preempted or deleted without possible adverse effects.

6.3.3 The worst-case environment of ATC

The US nation air traffic control system is divided into 20 airspace regions called ATC enroute centers. Each center is divided into sectors. A controller may control one or more sectors in a center. In the ATC center we consider, there are 600 controllers. Also, our worst case ATC environment has 12,000 aircraft tracks. There are 4,000 controlled IFR (Instrument Flight Rules) flights in the current controller center. The remaining 8,000 are uncontrolled VFR (Visual Flight Rules) flights and adjacent center controlled IFR. Because of the multiplicity of sensors, we assume there are 12,000 sensor reports coming into the system per second.

Corresponding to the number of flight tracks, the ASC is assumed to have 16,000 PEs that can be used to process data simultaneously. Each of PEs is randomly designated to a particular track. Since ASC has wide memory access bandwidth, it can read from or write to any or all tracks as local memory access. This provides highly parallel processing of data.

6.3.4 ATC Tasks

An ATC system has to process a large amount of data coming from radar sensors and provides accurate flight information in highly constrained time. Various processing procedures can be grouped to following eight major tasks [82].

- Report Correlation & Tracking
- Cockpit Display

- Controller Display Update
- Aperiodic Requests
- Automatic Voice Advisory
- Terrain Avoidance
- Conflict Detection & Resolution
- Final Approach (100 runways)

Each of these tasks is executed periodically. A task has an individual number of jobsets in each period *P* and each jobset has its derived execution time. The running time for a task is the execution time of the worst case jobsets. A task has a deadline determined by its release time plus its running time. All deadlines of individual tasks must conform to the system major deadline *D* or the major period *P*.

6.3.5 Task examples

To better understand the ATC tasks, we give some examples of the ATC tasks in this section. They were also presented in more detail in [83]. Since the ASC is used to execute these tasks, we are able to take advantages of the ASC machine and execute the tasks in jobsets. Intuitively these tasks are more efficiently executed on an associative processor than on a multiprocessor.

Example 6.1 – Report correlation & tracking

Report correlation and tracking is a task that correlates aircraft data about position as reported by radar and the predicted data of established tracks for the aircraft in the system. A track is the best estimate of position and velocity of each aircraft under observation. This task is present in many command and control real-time systems and a major limitation in ATC performance. In our work, it is assumed to execute every half second period.

Since all these data are stored in a shared relational database, we use two database relations to record aircraft data from radar reports (Relation R) and the predicted positional data of established tracks for the aircraft (Relation T). The correlated radar reports are used to smooth the position and velocity of the tracks to obtain the next estimate of position and velocity. Altitude information would be used to obtain a better correlation, but for simplicity we do not include that operation in our work. Given an unordered set of tracks, each report record must be evaluated with every track record in the system to assure a match (correlation) is not missed. Multiple matches are treated different from unique matches, and any report records that do not match a track record are entered as new tentative tracks.

The correlation process proceeds as follows. First, a box is developed around each track to accommodate track uncertainties. A box is also developed around each report to accommodate report uncertainties. The two database relations R and T contain information about the x and y coordinates of points in 3-D space. The objective of the task is to determine the join of the two relations as the intersection of two boxes, which is a many-to-many join. Each box from R is evaluated for intersection with every box in T until all boxes from R have been compared with all boxes in T.

As presented in [82,83], let T have columns x, y, j, x1, y1, and q. Let R have columns x, y, r, and k. Here (x, y) is the position of aircraft on the records of T or R; j, r are used to give the sizes of boxes developed for aircrafts in T and R, respectively. (x1, y1) in T is the reported position for the track record that is based on the current correlated radar report. Both q and k are flags set during the correlation procedure. As shown in Fig. 6.1, a box is developed around each point (x, y) with the four corner points (x±j, y±j) in T. A similar box is developed about each point (x, y) in R with the four corner points (x±r, y±r). j is based on the uncertainties, as shown by track quality, of each track in T and r in R is based on uncertainties in the radar report in R. Each box of the radar report in R is compared with each box of the track record in T. If an intersect is found between one report box and one track box, for example, r2 and t2 in Figure 6.1, then the report data is entered into x1 and y1 of the correlated record in T and a correlation flag is set in column k. This record will be excluded from further testing with other reports in this period. If two or more matches are found, an ambiguity flag is set into the correlated k values to indicate matches are ambiguous, and associated tracks are not updated in this period. If no match is found, a not match flag is set into column q of the correlated record in R.

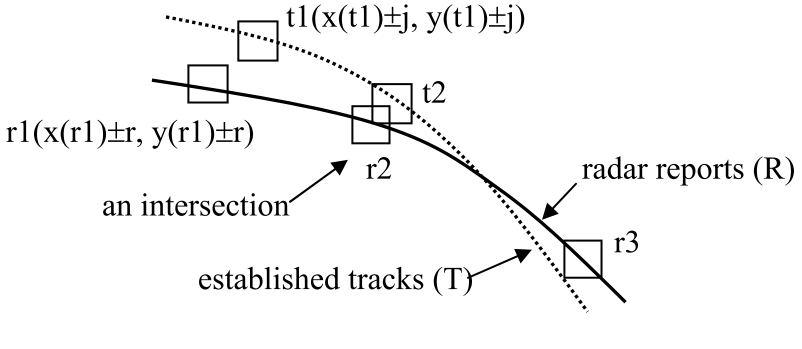


Figure 6.2 Track/Report correlation

When all report boxes have been compared with all track boxes, we repeat the above process using an increased size of boxes for those tracks that have not been correlated. This is to consider the uncertainties that may have been caused by track acceleration, turns, or by greater noise in the report. The value of j is doubled to enlarge the box for all tracks that do not have a correlation flag or an ambiguity flag in column k of T. The process is repeated for all unmatched radar reports, which have the "not match flag" of column q set in R. When no intersections are found, the process is repeated with $j= 3*j$. If there still remain unmatched radar reports, new tentative tracks are started for the reports in order to detect arrival of any new flights. If reports are due to noise, they usually will be dropped in a few periods (several seconds) based on further evaluation of other radar reports.

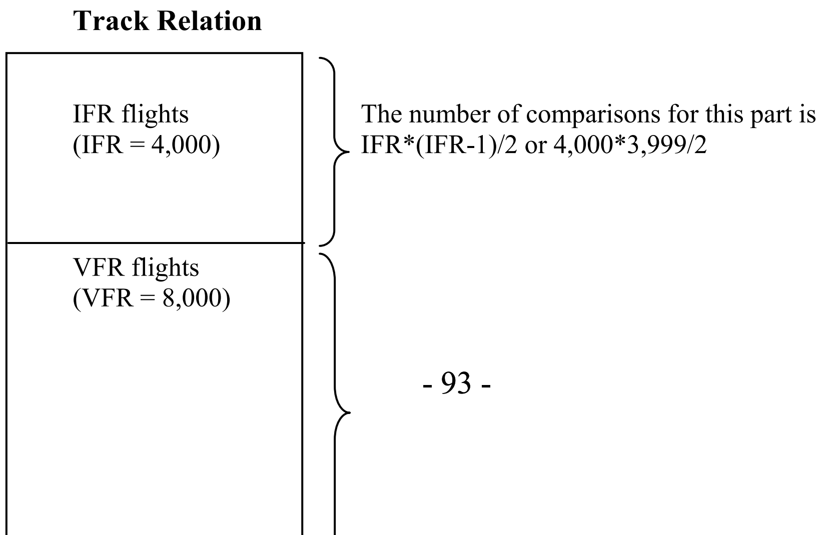
It is noted that, in the AP solution, each report record is tested with every track record in one set operation set operation that requires constant time. That is, the overall AP time is $O(n)$ where n is the number of reports in this period. In the MP on the other hand it is seen that the MP process is $O(n^2)$ because it has $|R| \times |T|$ processes, assuming O(1) processors can access and update needed data from the dynamic data base and work on this problem simultaneously. In the worst case situation we anticipate 12,000 reports per second against 12,000 tracks. For the AP this is 12,000 operations which is $O(n)$, whereas in the MP it is $n(n\text{-}1)/2$ or 7.2×10^7 which is $O(n^2)$.

Example 6.2 – Flight conflict detection

In the second example, we look at the task of detecting conflicts between aircraft paths in a dense environment. We consider the worst case ATC environment, as given in Section 6.3.3. The objective of this task is to determine the possibility of a future conflict between any pairs of aircraft within a twenty-minute period. To assure timely evaluation, the detection cycle is set to be 8 seconds. Any near approach within 3 miles or 2,000 feet in altitude will be considered a conflict.

Specifically, all IFR flights are evaluated for their future relative space positions between each other; and each of these IFR flights is also evaluated against all VFR flights for their future space positions. This means that every 8 seconds we must evaluate each of the 4,000 IFR flights with the other 11,999 IFR and VFR flights in the worst environment. The process is essentially a recursive join on the track relation where the best estimate of each flights future position is projected as an envelope into future time. The envelope has some uncertainty in its future position that is a function of the track state. It is modified by adding 1.5 miles to each x, y edge of the future position (to provide a 3 mile minimum miss distance) and 1,000 feet in the altitude (to provide a 2,000 feet minimum miss height). For each future space envelope, its data is broadcast to all remaining PEs first. Then an intersection of this envelope with every other space envelope in the environment can be checked simultaneously in constant time. It takes 4,000 iterations or O(n) time to complete all jobs. However, it is readily seen that this process requires $\Omega(n^2)$ steps in a MP. As each flight is tested, it is removed from the list. The total number of operations is *IFR*(IFR-1)/2+IFR*VFR* or 4,000*(3,999)/2 + 4,000*8,000 which yields 39.998*10^6 operations (as shown in Figure 6.2).

It is noted that this number does not include many other MP overhead costs that are usually necessary for concurrency management of multiple jobs on a MP, such as data distribution, processor assignment, mutual exclusion of data access, etc.


Track Relation
IFR flights
(IFR = 4,000)
The number of comparisons for this part is
IFR*(IFR-1)/2 or 4,000*3,999/2
VFR flights
(VFR = 8,000)

The number of comparisons for this part is
IFR * VFR or 4,000 * 8,000

Figure 6.3 The worst-case number of operations for conflict detection in MP

However, an AP is much more efficient in executing this task because of its set operations. In the AP this same process takes O(n) steps and requires only 4,000 operations at most. The reason for this is as follows. First, all future flight envelopes are generated simultaneously which takes O(1) steps. Then, the first “trial envelope” of the 4,000 IFR future flight envelopes is compared for possible conflict with all the other 11,999 flight envelopes for a look-ahead period of 20 minutes. Thus the equivalence of maximum 11,999 jobs is completed simultaneously in this jobset. This occurs because each of the 12,000 records in the track table is simultaneously available to each of the 12,000 PEs that are active in the AP. The next operation selects the second trial envelope and repeats the conflict tests against the remaining 11,998 tracks. When a trial envelope has been tested it is marked “done”, and future trial envelopes will exclude all prior trial tracks. When the last of the 4,000 trial envelopes is tested the AP will have completed 39.998 $\times 10^6$ jobs just as the MP did. But due to the simultaneous execution of all jobs in each jobset, the AP will complete the entire set of jobs in at most 4,000 steps.

6.4 ASC solution to the ATC problem

In the ATC system, all the other tasks we list earlier are essentially similar to the task examples presented in last section, although they have different deadlines and release times. Handling the execution of the same kind of k set tasks is the major clue of our solution to the ATC problem. Static scheduling will be used to ensure tasks can meet their deadlines in the worst case. Aperiodic tasks, such as controller requests or weather update, are handled by a special task executed once per second.

6.4.1 Static schedule solution

We apply static scheduling in our AP solution to the ATC problem, which is fundamentally different from the heuristic scheduling normally used in MP solutions. Like most other real-time systems, the ATC problem cannot be efficiently scheduled using multiprocessor. One of the main reasons is that an MP approach uses dynamic scheduling to schedule all the tasks. The data keeps changing, so the MP system cannot determine, a priori, how many records will actually participate in the computation of a task or how much actual computation time a task in a cycle will require. In the AP, since tasks are executed as jobsets, there is no need to differentiate between a jobset with one record and a jobset with thousands of records, since both will require the same amount of time. Thus, the number of records in a jobset is simply a "don't care" parameter. All set operation times are based on the worst case assumption. This makes it possible to schedule ATC tasks statically in the AP solution.

We assume that there is a different PE for each flight and that the data for each of the n flights is stored in its PE. Due to the obvious data-intensive nature of these computations, an AP can efficiently allocate and manipulate data in terms of set operations. Each record of a relation resides in the memory of one PE. Since the same operation is often performed on all the involved records, we can process all the records simultaneously. When using an AP for this example, all of the *n* jobs in a jobset are processed simultaneously where *n* does not exceed the number of PEs. If each jobset cost is c_i (for task T_i), this cost can be calculated at the instruction level. The following condition establishes overall system feasibility, where D is the deadline:

$$\sum_{i=1}^{k} c_i \leq D \qquad (1)$$

If Equation (1) is satisfied, the jobs can be processed in order of precedence constraints and all deadlines will be met.

Using this theory, a static schedule for the ATC tasks can be designed. Table 6.1 quoted from [84] provides realistic data of all eight major ATC tasks with their periods, deadlines and computation costs based on work that was previously done on two associative SIMD or ASC architectures, the STARAN and

Tasks	p	j	c	d	Proc time
1. Report Correlation & Tracking	.5	15	.09	.10	1.44
2. Cockpit Display 750 /sec)	1.0	120	.09	.20	.72
3. Controller Display Update (7500/sec)	1.0	12	.09	.30	.72
4. Aperiodic Requests (200 /sec)	1.0	250	.05	.36	.4
5. Automatic Voice Advisory (600 /sec)	4.0	75	.18	.78	.36
6. Terrain Avoidance	8.0	40	.32	2.93	.32
7. Conflict Detection & Resolution	8.0	60	.36	3.97	.36
8. Final Approach (100 runways)	8.0	33	.2	6.81	.2
Summation of Tasks in a period P					**4.52**

The system period P (in which all tasks must be completed) is 8 seconds
p the task period time, in seconds, is used to determine the next release time $r_{i+1} = r_i + p$,
j is the execution time, in microseconds, for each jobset of a task,
c is the cost for each task for the worst-case set of jobsets,
d the deadline time for each task $r_i + c + .01$ (includes 10 ms interrupt processing per task)

Table 6.1 Statically Scheduled Solution Time

ASPRO. The STARAN was used in the Dulles airport in Washington, DC for a complete ATC demo at the International Air Show in 1972. The STARAN-Y was used in a demo of the ATC system prototype for FAA at a Knoxville terminal in 1971. Also, about 140 ASPROs have been used in the US Navy E2C airborne early warning system for an environment of 4000 sensor reports starting in 1979. The ASC model used to provide the performance shown in Table 6.1 had 32 times as many PEs as the 1972 STARAN. The memory was assumed to be 40 nanoseconds, and the PEs were upgraded similar to the MPP ALU which also had 16,384 PEs [22]. By examining the figures in Table 6.1, it is easily seen that, with the system major period of 8 seconds, statically scheduling the tasks with the total processing time of 4.52 seconds to have them meet the deadlines is possible. If the memory access time were reduced to 20 nanoseconds using today's technology, the total time would only be 2.26 seconds.

The resulting fixed schedule of tasks is illustrated in Figure 6.4 from [84]. Time slots for individual tasks within a system major period (8 seconds) are carefully tailored. Each of them provides sufficient time for the worst-case execution of the complete set of system tasks in the ATC system. The periodic tasks are run at their release times and each is completed by its deadline. For a particular task, its required jobsets need to be identified at first. Then each of these worst case jobsets is evaluated to determine its computation time. Depending on its function and repetition, this computation time multiplied by the worst case number of the jobsets is assigned to be the worst case computation cost *c* of the task.

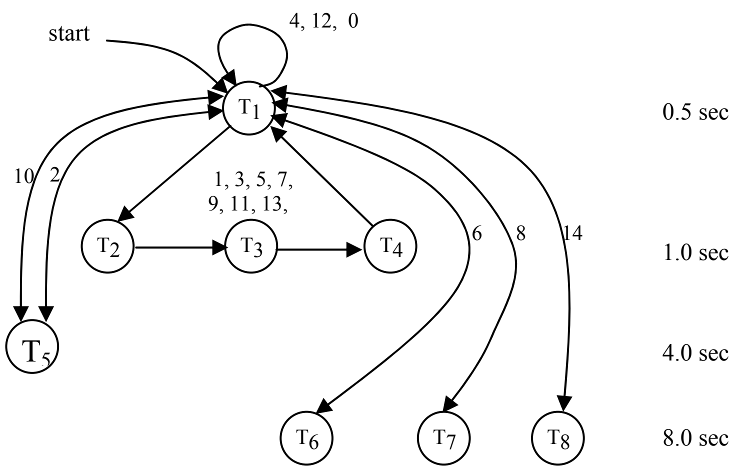


Figure 6.4. The AP ATC schedule
(The number is the execution order in a period)

One task is specially designed to handle aperiodic and sporadic jobsets that may have arrived within the last one second period. These jobsets do not have a regular period. A one second period as shown in Task 4 in Table 6.1 is designated to process all of these jobsets. We assume there are no more than 200 such jobsets per second and each requires less than 250 microseconds, which is reasonable based on real world facts.

Each jobset within a task executes the same operation on multiple data items simultaneously. As a result, we view these as set operations and consider the AP to be a set processor. Since we assumed the worst case situation, it is suggested that the maximum set of operands may be of the order of 12,000. An operation over the entire set of records is executed simultaneously (i.e., in lock step) over all data using one instruction stream that is executed only once.

6.4.2 Analysis of the AP solution

As given in last section, the static scheduling of all the ATC tasks is feasible. When the tasks $T_1 \ldots T_8$ are executed using the STARAN design, updated for today's technology, they can be completed in 4.52 seconds, which is 56% of the time available in each 8 second cycle. It is expected that the total time could be cut in half using 20 nanoseconds memory technology.

The worst case running time for each task can be determined in advance. Because we design a static schedule based on the fixed worst-case computation time for each of the tasks, there is sufficient time for execution of all the jobsets. This schedule ensures that all tasks will meet their deadlines. Therefore, this schedule runs in constant time. In the likely case when the processing load is lower there will be more waiting time

By taking the advantages of set operations, the wider data access bandwidth (200 to 500 times increase), and the ability to predict the worst case execution time for the AP, this approach provides a much simpler solution to the ATC problem. The inclusion of this schedule into a table-driven scheduler [111] has greatly simplified the design procedure. In addition, the AP also virtually eliminates other problems that the ATC real-time processing usually requires in a multiprocessor environment. These include costs of running a dynamic scheduling algorithm, resource sharing due to concurrency, processor assignment, data distribution, concurrency control, mutual exclusion of data access, indexing and sorting, etc. These costs not only complicate ATC software and algorithm design, but also dramatically increase the size, complexity, and processing time of the system. On the other hand, static scheduling assures the predictability of system performance. A static schedule is both reliable and adaptable and can easily be modified to incorporate system changes as needed.

6.5 Complexity theory and current practice for ATC

Seeking an efficient solution for a real-time problem involving scheduling tasks on a multiprocessor has been explored extensively over the past 40 years. In fact, complexity results for the problem of scheduling a set of real-time tasks on multiprocessors were established very early. In theorem [SS8] of [46, page 238] by Garey and Johnson, it is shown that, given a set of real-time tasks each having varied computation time, there is no known polynomial time algorithm using a multiprocessor to schedule this set of tasks. In particular, let T be a set of tasks, $m \in Z^+$ (the set of positive integers), length $l(t) \in Z^+$ for each $t \in T$, and a deadline $D \in Z^+$. The problem of whether there is an *m*-processor schedule for *T* that meets the overall deadline *D* is NP-complete for $m \geq 2$, assuming not all tasks have the same length. (Here, *m*-processor refers to a MP with *m* processors.) This theorem applies to the problem of scheduling ATC tasks on a MP.

The conditions presented in the AP solution are made more difficult for a MP than would be the case if all the tasks were independent and did not share resources. Since all tasks share the same data source (a common database), the scheduling problem for this set of tasks with the added requirements that they be performed on a MP is NP-hard as observed in [46,108,109]. On the other hand, a polynomial time AP algorithm for the ATC problem (including scheduling) has been described in this chapter. While the MP could conceivably also solve the problem using static scheduling by simulating the AP solution, communication time makes such a solution highly unlikely. This AP solution is not the one normally chosen by professionals working in this area. One explanation for this is that AP (and SIMD, in general) algorithms are very fine grained, but coarse grain algorithms that emphasize "parallel slackness" and faster average running time are considered to be more efficient for asynchronous computation. Other reasons for this include the inability of the MP to efficiently simulate an AP algorithm, which will be discussed in the next chapter. An inefficient simulation of the AP algorithm is likely to result in the MP being unable to meet the required deadlines.

As a result of the preceding discussion, it is apparent that the AP can solve this real-time problem in polynomial time, but that an MP cannot normally solve the same problem in polynomial time. In view of all the attention this problem has received, and expenditure of thousands of person years, it is highly unlikely that a polynomial time MP solution will be found.

However, given the current dominance of MIMDs, current research on command and control real-time systems like ATC remains focused almost exclusively on MIMD architectures. A lot of work is devoted to using heuristic techniques to speed up the MP solutions, but the resulting software is still unsatisfactory.

MP solutions of the ATC problem have repeatedly failed to meet the Federal Aviation Administration air traffic control requirements [81,82,102]. One notable example is the ten-year effort to develop the Automated Air Traffic Control System (AAS). The AAS program was canceled in June 1994 after expenditure of several billion dollars [102]. A USA Today editorial [120] sees the situation this way: “This time, the FAA has only itself to blame. Back in 1995, Congress freed the FAA from cumbersome procurement rules that the FAA claimed were the main cause for the unrelenting delays and cost overruns plaguing their 17-year, $41 billion modernization effort.” Another example is a recent special broadcast on ABC news about current ATC system difficulties [132]. They stated that the cost of the development of the current solution to the ATC problem through 2005 has been projected to be 44 billion dollars. All attempts at providing a multiprocessor solution to the ATC problem dating back to 1963 have failed to meet their performance requirements (e.g., CCC that started in 1963, DABS/IPC in 1974, AAS in 1983, STARS in 1994.). The STARS project is now over budget and nearly four years behind schedule [102].

On the other hand, the AP solution to ATC presented here indicates that a simple, low order polynomial time solution is possible. In fact, a demonstration of a complete hardware-software ATC system using an early version of the STARAN, an ASC computer, was given in the Dulles airport in Washington, DC at the International Air Show in 1972 by Goodyear Aerospace [81,84], while the engineering development team was headed by Will Meilander. The ATC software consisted of the following processing tasks: automatic tracking, conflict detection, conflict resolution, terrain avoidance, and automatic voice advisory for pilots in an environment of 2000/4000 tracks/reports. The 1972 AP demo provided capabilities that are still not possible with current ATC systems. Given the above fact, we feel that resources should be provided to develop appropriate AP hardware to test the ATC system prototype and expand this restricted AP solution to a full-scale solution to the current nation's ATC problem. Additionally, more attention needs to be given to possible ASC and MASC solutions to other real-time problems and to computational intensive application areas involving large real-time databases such as bioinformatics and military command and control systems.

6.6 Summary

In this chapter, a feasible and predictable ASC solution to the ATC problem is presented. The ATC problem is a typical real-time problem that has been extensively explored using multiprocessors by numerous researchers and software developers. However, a satisfactory solution has not been discovered. In complete contrast to the conventional solutions using multiprocessors, the ASC solution employs static scheduling and takes advantage of the huge processor to memory bandwidth and the unique associative features of the AP architecture. Using ASC, each task is allocated sufficient execution time based on its worst case computation time and executed in jobsets using set operations. An AP can allow considerable amounts of real-time data to be accessed in constant time. No data indexing/sorting is needed, as data can be located easily using associative searches. Similarly, no message passing is needed as there is only one instruction stream and this IS controls all execution activities. In the prototype of the worst case ATC environment, the total computation time of all eight periodic ATC tasks is 4.52 seconds, about 56% of the system major cycle period of 8 seconds, and this time could be cut by a half using today's memory technology. It is seen that a feasible and predictable schedule of all the ATC tasks can be designed. The ASC solution runs in polynomial time, in contrast with the conventional MP solution that involves dynamic scheduling that has been proven to be an NP-complete problem. Repeated failure of the national ATC system design for over forty years demands an efficient ATC solution using a completely different system paradigm. The ASC solution provides this new approach.

The reasons that the ASC model can solve the ATC problem efficiently are its constant time associative operations, SIMD execution style, and extremely wide memory bandwidth. These have eliminated a number of difficulties that have to be handled in other MIMD architectures. We give more detailed discussion in the next chapter.

Chapter 7

Power of SIMDs vs. MIMDs in Real-time Processing

7.1. SIMD vs. MIMD

We have briefly introduced the Flynn's classification of computer systems in last chapter. As we mentioned in Section 6.1, the SIMD and MIMD models are the most important categories for parallel computation. With SIMD, all processors are ALUs controlled by a central control unit and operate in lockstep or synchronously. The SIMD model has advantages of being easily programmed, since it is programmed even more simply than a sequential processor is programmed by virtue of the fact that it never needs to manage iteration over a set of data. The SIMD model is also cost-effective, highly scaleable, and especially good for massive fine-grain parallelism. On the other hand, with MIMD, each of processors has its own program and executes at its own pace or asynchronously. Thus, the MIMD model has advantages of high flexibility in exploiting various forms of parallelism, availability of using current high-speed off-the-shelf microprocessors, and being good for coarse-grain parallelism. As hardware prices have dramatically dropped during the past decades, MIMD computers (particularly clusters) have become cheaper to build and to update architecturally than SIMD computers. They have shown a better performance/cost ratio on average and drawn more researchers' attention in general. It is commonly believed that the MIMD model is more powerful than the SIMD model. One example of this is the statement "MIMD machines can solve any problems as efficiently as any other types of computers" [124]. As a result, MIMD computers dominate today's market and SIMD computers have all but vanished. Many people in the field are pessimistic about the future of SIMD computers [10,30,95,97].

However, for some real-time problems, in particular, the Air Traffic Control problem in the last chapter raises questions concerning the validity of the claims that the MIMD model is more powerful than the SIMD model. For the ATC problem, an efficient SIMD solution can be obtained that runs in low-order polynomial time, while it appears to be very difficult and perhaps impossible for a MIMD to provide a polynomial time (i.e., tractable) solution that meets the required deadlines. This suggests that SIMDs may not be outdated and may have an important role to play in meeting current computational needs. MIMDs have some intrinsic weaknesses such as synchronization costs, communication limitations, mutual exclusion for accessing shared resources. The impact of all these weaknesses was not seriously recognized until MIMD solutions to the ATC real-time problem with hard deadlines was compared to a AP solution of this problem. While an AP has additional capabilities not possessed by a traditional SIMD computer, most SIMD computers can simulate an AP efficiently and in low order polynomial time.

In this chapter, comparisons of SIMDs and MIMDs in terms of power in real-time processing are further discussed. We will analyze the limited power of MIMDs based on discriminating simulations between a SIMD model (ASC) and a MIMD model (BSP). We will also discuss theoretical and practical difficulties of finding an efficient MIMD solution for real-time processing. While a typical MIMD solution to a real-time problem involves dynamic task scheduling which is an NP-hard problem, an associative SIMD solution to ATC can avoid dynamic task scheduling so that an efficient polynomial time ASC solution can be developed. Based on this fact, it is strongly urged that the research community reconsider the power of SIMD computations.

This chapter is organized as follows. Section 7.2 lists advantages of associative SIMD or ASC when used to solve real-time problems. Section 7.3 presents possible MIMD approaches to solve ATC problem and shows why they cannot work efficiently. Section 7.4 discusses difficulties of efficient MIMD solutions to a real-time problem. Section 7.5 analyzes simulations between ASC and BSP in the common sense and explains why they cannot be used to transfer a SIMD solution to a

MIMD model for real-time processing. Section 7.6 provides the summary.

7.2. Advantages of associative SIMDs or ASC

The ASC model is an associative SIMD model. It has one instruction stream (IS) and all PEs execute synchronously. Its associative properties include locating data by content rather than by address and other constant time associative operations. It stores, retrieves, and processes data in a different way compared with a conventional parallel model. It naturally supports massive data-parallelism. The following detailed features make it especially efficient in solving real-time problems such as ATC. In fact, using these features, associative SIMDs are expected to be useful in providing extremely efficient solutions to other command and control type problems and, more generally to other real-time problems with deadlines.

- Eliminating expensive synchronization and communication costs

Since a MIMD has the flexibility to allow every processor to run at its full speed on independent computations without waiting for the other processors, it is generally believed that such an asynchronous model is faster than a synchronous SIMD model. This belief is sometimes interpreted to mean that if a problem can be solved efficiently in a synchronous model, then it should be possible to execute the same algorithm in an otherwise identical asynchronous model at least as efficiently. However, it has been shown that synchronization of p processors in an asynchronous PRAM model takes $\Omega(\log p)$ by gathering responses from all the processors [48]. Moreover, the actual synchronization costs are very high for most realistic models and actual computer systems. Even with a low complexity bound for communications, in practice the value of the constant for these complexity bounds can be quite large. Also with the communications capabilities available in most models, the complexity of the operation of gathering responses from all processors will be much greater than $\Theta(\log p)$. In practice, it is difficult and usually impossible to provide guaranteed efficient bounds on message transmission times in an asynchronous system with heavy communication traffic. Coarse-grained computation is normally used with asynchronous systems to maximize the amount of computation done between synchronizations and to minimize the number of synchronization points. However, these problems are all eliminated in the ASC model since all the PEs in ASC execute synchronously. An associative computer has a single IS. This IS, all PEs (which are really only ALU's), and the interconnection network operate using a single clock, so that there is no concern about synchronization costs. The communication algorithm steps are executed synchronously so that communication times can be predicted.

- Locating data by content rather than by address

In ASC, data is accessed by content rather than by address. Sorting and indexing are normally avoided in ASC algorithms since the usual reasons for sorting or indexing is to allow items to be located quickly. Normally, MIMD real-time applications maintain multiple sorted lists in order to more quickly locate different data items based on different search parameters. This increases both the size of the MIMD software package and the amount of computation required. In a highly dynamic real-time system, the cost of updating a large set of indexes and searching for data items in a large distributed data base can be very high. This additional time can make it much more difficulty for a MIMD system to meet real-time deadlines. Since ASC does not require these tasks, it simplifies the design of real-time algorithms and reduces their running times significantly.

- Massive parallelism for intensive data-parallel computations

Due to its ability to support a very large number of processors, a SIMD usually assigns one or, at most, a small number of records of each type to a PE in contrast to a MIMD. As a result, a SIMD supports data-parallelism much more efficiently. The broadcast capability of the ASC model allows data to be broadcast to all PEs and minimizes (and often eliminates) any needed data movement on the interconnection network between the PEs. Any parallel data flow that occurs in SIMDs using the interconnection network between the PEs will be programmed to move synchronously, avoiding the problems associated with procedures such as store-and-forward, pipelines, etc. However, with a MIMD model, the data movement is handled using either a network between the processors or shared memory. It is difficult to predict the time required for data movements in a congested MIMD network or on a jammed memory bus. Additionally, since the PEs in the ASC model execute all operations simultaneously, a jobset for a large number of objects can be executed simultaneously and the exact worst case time can be calculated and be allocated to the jobsets. However, a MIMD model executes operations of a real-time task asynchronously and each processor normally has multiple processes running at the same time. Since all of the processes on each MIMD processor will need to access and update a distributed database, it is usually impossible to calculate the exact running time of a single task on one processor. It is even more difficult to calculate a reasonably tight upper time for all the processors to execute all the similar tasks that a SIMD can handle in one jobset.

- Wide memory bandwidth providing a large amount of data simultaneously

Conventional memory access suffers from a very limited bandwidth that usually allows 32, 64, or 128 bits (or a word) per memory access. While it can be argued that a MIMD can access a memory item during the same general time period, a problem with a MIMD is that each memory access by a processor only provides one processor with the item retrieved from the memory. With a distributed memory MIMD, an item retrieved from memory by one processor has to be sent to another processor that needs that item using message passing. With a shared memory MIMD, all processors that need a particular item must obtain it from shared memory, which is a much slower and much more complex process than a processor obtaining data from its local memory. The process of storing an item from a MIMD processor to distributed or shared memory is similar. All of the continual data movement between the distributed memory and the various distributed processors contribute substantially to the demands on the network, which must also handle data movements required by the algorithm being executed. A similar situation occurs with the shared memory MIMD system. Also, since the number of processor in a MIMD is normally small in comparison to the number of PEs in a SIMD, each processor in a MIMD must have much larger number of memory accesses and memory stores than each PE in a SIMD.

In contrast, with a SIMD, all processors can simultaneously retrieve a bit (alternately byte or word) from their local memory. For example, the STARAN had a memory access capability of 44,096 bits per access. Since records for a particular object are all stored in the local memory of the same processors, there is frequently no need to send an item from these records to other processors. When a data item is needed by multiple processors, that item is normally broadcast in one step to all of the processors using the fast broadcast bus.

The ASC model assumed here for the ATC application has been enhanced with a multidimensional access memory (MDA) scheme based on the computers that motivated the ASC model, namely the STARAN and ASPRO [18,74]. This MDA feature was implemented by a flip network between the memory and the PEs. (Actually, the ASPRO had a series of flip networks, but that is not important to the current discussion.) The flip network allowed the PEs to access in one step either a bit slice containing a bit from each PE or a block of memory (or a long-word slice) in one PE containing a record. In addition, the MDA scheme is designed to convert the accessing data format from bitslices to word-slices in constant time, which is called *corner turning* [18, 98]. A bit-slice is used for wider bandwidth to move data between PEs and memory, while a word-slice is used for faster input/output to transfer data between memory and I/O devices. The MDA feature enables the ASC to overcome the bit by bit memory bottleneck problem existing in nearly all other SIMD computers. It also provides a SIMD with a fast I/O capability.

7.3. Possible MIMD approaches to solve the ATC problem

Most of the work of developing ATC software during the last 40 years has focused on software for various types of MIMD computers. As noted earlier, extensive efforts have been made at various points in time to develop software for a multiprocessor system and these have involved enormous costs and thousands of person years of effort. Here, we will consider two possible MIMD solutions to the ATC problem. The first approach considered is a standard MIMD-type solution using dynamic scheduling for ATC tasks. This approach has been the principal one followed during the past 40 years and has been explored by many researchers and software developers who work in this field. The second possible solution considered is a MIMD simulation of the AP solution presented in last chapter, which uses static scheduling. Since MIMDs are assumed to be able to simulate SIMDs and to execute SIMD algorithms as efficiently as SIMDs, it is particularly important to consider this approach carefully. This section provides detailed discussion about both of these possible approaches and analyzes problems existing in each approaches.

In the first (or the standard MIMD) approach, all processors execute their own program asynchronously. The focus is on obtaining the best average case running time (as opposed to the fastest worst case time used in the earlier AP solution) in order to obtain the best overall performance. Typically, software for MIMD systems emphasizes coarse grain computation (which maximizes the amount of computation the processors perform between synchronizations) and parallel slackness (which expects multiple jobs to be assigned to each processor in order to maximize the amount of time that, on average, the processors stays busy doing useful computations). The execution times for most ATC tasks are unpredictable for many reasons, including the fact that each processor is running multiple jobs and the time to obtain needed data from the distributed or shared memory system is unpredictable. As a result, the time taken by different (or the same) processors to execute identical jobs for the same task can differ widely. Therefore, tasks have to be scheduled dynamically. Since a large amount of real-time data is also being maintained and accessed by tasks in a real-time database, the database activities need to be scheduled dynamically as well. However, almost all versions of the problem of dynamic scheduling of a set of real-time tasks on MIMD are NP-hard [46, 108, 110]. As a result, such a MIMD solution of the ATC includes the solution to one or more NP-hard problems. The difficulty of finding solutions to real-time problems that does not involve the continuous solution of one or more NP-hard problems has led to the general belief that these real-time problems are themselves NP-hard. However, the ATC is widely accepted as a typical real-time problem with hard deadlines and our earlier polynomial time AP solution to the ATC problem showed that the ATC problem is not NP-hard.

This MIMD approach has been expanded to include various heuristic algorithms or methods to provide a low quality but acceptable (e.g., approximate) solution to computationally intensive problems including dynamic scheduling. There are various types of heuristic algorithms. The most common version is to provide an "approximately correct" answer, although sometimes it is not feasible (and sometimes impossible) to check on the closeness of the approximation. Other heuristic algorithms can produce the correct answer most of the time but either not produce an answer or produce a wrong answer occasionally. For example, a heuristic algorithm might produce a feasible schedule for executing real-time tasks that was not the "best possible". However, these heuristics produce a lower quality answer in some sense and adding additional software to an already unwieldy system can provide additional possibilities for software errors. Some heuristics require additional hardware support, such as a scheduling chip. Generally, a heuristic algorithm will increase the inherent unpredictability of the running time and the size of the software.

As we mentioned in Section 6.5, using a MIMD approach to solve the US version of the ATC problem has been explored by a large number of experts for a long time. No ATC software has met the

FAA specifications since 1963. It seems that if a polynomial time MIMD solution is possible, it would have been discovered. In fact, it is generally accepted that a polynomial time solution for real-time problems with deadlines such as the ATC is unlikely and probably impossible.

We next discuss the second multiprocessor approach mentioned earlier. It is generally believed that it is possible to efficiently simulate SIMD computers with MIMD computers. With the rapid advances in MIMD architectures, some are also inclined to believe that an appropriately designed MIMD computer could execute a SIMD solution to a problem like ATC faster than would be possible for any SIMD built for that problem using today's technology. A MIMD simulation of the ASC solution to the ATC problem given in Chapter 6 would raise the following problems:

- A typical MIMD usually has far fewer processors than the 12,000 PEs assumed for the associative processor executing the ASC solution. A MIMD simulation of this solution would assign each of its processor to simulate a large number of the PEs for the ASC solution. As the number of SIMD PEs assigned to a single MIMD processor increases, there will be degradation in the MIMD execution. Even if the speed of a MIMD can be significantly faster than that of a SIMD processor, the overhead of this multiple assignment must be considered. On the other hand, if a MIMD is built that is large enough to have only a few SIMD PEs assigned to each of its processors, then the cost of communication between the MIMD processors, the cost of accessing and maintaining a distributed database, and the cost of each synchronization will increase significantly.
- The ASC solution for the ATC is saturated with the use of the associative operations. A possible hardware implementation of these in constant time on an associative processor is described in Chapter 4. A significant slowdown in the MIMD simulation of these associative operations is almost certain to result in missed deadlines in the ATC static schedule, as these associative operations are likely to be substantially slower on a MIMD than on an associative computer. As indicated by the experiments reported in [51], the fast implementation of broadcast and reduction operations on a cluster of workstation has latency of several hundred nanoseconds. This slower execution of the broadcasts and reductions becomes significantly worse as the data size and the number of processors increases. Since the implementation of the other associative operations (e.g., associative search) can be implemented

using the broadcast and reduction commands or in a similar fashion to them, it is unlikely that a MIMD will do a better job of implementing these other commands than with the broadcast and reduction commands. A software implementation of these associative requires a communication latency of at least $\Omega(\log n)$, where n is the number of ASC PEs. On the other hand, the associative processor can execute these in constant time, using a hardware implementation. It appears unlikely that a MIMD execution of the associative operations will be sufficiently fast to meet the required deadlines for the ASC solution to the ATC problem.

- The computing style used in the ASC solution for the ATC is not usually good for MIMDs. In particular, this solution involves fine grain parallelism, and not the course grain parallelism usually recommended for MIMD computation. As a result of this fine grain computation, the number of synchronizations is likely to be much higher than would be the case for a coarser grain solution designed for a MIMD computer.

In view of the reasons given above, a significant slowdown in the MIMD simulation of the ASC solution to the ATC is almost certain to occur and result in missed deadlines in the ATC static schedule. Since multiprocessor solutions to the ATC problem have been studied for over 40 years, it would seem that a simple solution like this would have already been considered and tried if there was hope that it would work. On the other hand, it is hoped that the existence of a tractable ASC solution to the ATC problem motivates those working on MIMD solutions to this problem to design appropriate MIMD architectures that will support a tractable MIMD solution. This will be a major breakthrough for many real-time applications like ATC and will have significant implications regarding MIMD computation in general.

7.4. Difficulties of an efficient MIMD solution

A standard MIMD solution to a real-time problem such as ATC usually has to deal with dynamic real-time task scheduling. Given a relatively small environment with a small data size, real-time scheduling is solvable on a uniprocessor. Some optimal scheduling algorithms such as the EDF algorithm have been developed. Whether a set of real-time tasks is schedulable or not can be determined in polynomial time using a uniprocessor. However, the real-time scheduling problem may become NP-hard when one of the following conditions is added: multiprocessors, shared resources, non-preemption, varied release times, precedence constraints, etc. [46,110].

We now analyze why these conditions may be difficult to avoid whenever a MIMD system is used.

- Multiprocessors

When multiprocessors are used, tasks must be partitioned and assigned to individual processors. The algorithm used can assign the tasks to processors using some pre-assignment scheme or some heuristic procedure can be used to dynamically assign the tasks to processors during execution. However, an optimal assignment of tasks to processors is desirable and an optimal assignment involves the PARTITION problem. Since the PARTITION problem of a finite set is a basic NP-complete problem, so is the problem of optimally assigning a set of real-time tasks with deadlines to a multiprocessor (the detailed proof can be found in [46]).

- Shared resources

Shared resources also cause a real-time scheduling problem to be NP-hard [46,110]. In some real-time systems, a real-time database is used as a common resource accessible by all tasks. Generally, resource sharing can be realized by mutual exclusion constraints. Unfortunately, finding an optimal schedule for a set of real-time tasks with mutual exclusion constraints and different computational times is an NP-hard problem. This is because mutually exclusive scheduling blocks have different computational times that cause NP-hardness in the same way as discussed earlier using the partition problem.

MIMDs either with shared memory or with distributed memory cannot efficiently handle mutual exclusion for shared resources. Besides, a shared memory MIMD suffers from having limited bus bandwidth to transmit data between the shared memory and individual processors. To overcome this problem, it has to handle cache coherence to maintain data consistency. A distributed memory MIMD also needs to take into account the significant costs for data transmission.

The ASC model eliminates the need for mutual exclusive access of shared resources. All data records for a given object are stored together in the local memory of the same processor. Most computations involving that object will be computed by the PE containing its object data. All processors do the computation for each task using jobsets, so the data needed by all PEs can be retrieved and broadcast. For example, in flight conflict detection task, the needed flight data for an aircraft is broadcast to all PEs and then each PE use this data to compute whether or not this aircraft is on a collision course with its aircraft (i.e., the aircraft whose data it stores). In particular, a single instruction stream does not require sharing of resources.

- Non-preemption

Although non-preemptive scheduling is easier to implement and incurs a lower overhead, an efficient non-preemptive scheduling algorithm on MIMD systems is almost impossible. Most scheduling algorithms on multiprocessors uses preemption [105,110,111]. However, when preemption on MIMDs is allowed, one must consider the difficulty of predicting costs involved in overhead, asynchronous execution of tasks distributed on different processors, synchronization of task execution after preemption, etc. Preemption may also worsen task migration and load balance among processors that exist in MIMD systems. All of these increase the complexity of a scheduling problem, causing it to be intractable.

- Varied release times

If all tasks have the same release time, a scheduling algorithm can be run in polynomial time. If tasks have varied release times, the scheduling problem becomes NP-hard [46]. In the MIMD environment, there is no global clock to provide an accurate time stamp and to measure elapsed time. If each processor operates independently at its own speed, it is impossible to have all tasks proceed synchronously at each step. In addition, unpredictable communication delays can also change task time stamps. Therefore, the asynchrony of MIMDs is unlikely to accurately maintain the same timetable for a set of real-time tasks even though they have the same release time. In contrast, in ASC, all processors run synchronously. They have absolute time stamps by sharing a global clock. If release times that are given are the same, they will not be changed.

- Precedence constraints

The existence of precedence constraints is another reason that causes NP-hardness in real-time scheduling problems. Compared to SIMDs, MIMDs have a small number of processors. Precedence constraints may require waiting for higher priority tasks or preemption of lower priority tasks. When tasks have to satisfy a certain order of execution, MIMDs have to handle problems such as idle processor time, expensive overhead, task migration, load balancing, communication delays, and so on. The scheduling problem that assumes precedence constraints is intractable. Unlike MIMDs, the static execution schedule of the ATC real-time tasks is determined at the time the program is created. The schedule is hard-coded into the program. The schedule does not change during execution. That is the reason for calling the schedule static.

The above difficulties seem to be intrinsic weaknesses of a MIMD system when it is used in real-time processing. When they cannot be avoided in a MIMD system, it is impossible that we can use the full power of MIMDs in solving the real-time problem. Consequently, it is not likely that a feasible polynomial time MIMD solution to a real-time problem such as ATC will be found.

7.5. Discriminating simulation of ASC with BSP

We have introduced various parallel models earlier. The abstract BSP model is a popular model for general-purpose computation, but essentially models MIMD-style computation. It appears to be an excellent model for most current MIMD architectures including SMPs and clusters, which are the currently the two most popular MIMD architectures. As stated in [123], BSP is expected to have universal efficiency over a special-purpose model. Then, “special-purpose machines have no major advantage since general-purpose machines can perform the same functions as fast” and use “high-level programming languages and transportable software”. Moreover, "theoretically, any parallel algorithms can be executed efficiently on the MIMD model” [123,124]. Based on the preceding claims, when an efficient optimal simulation is obtained, any algorithms can be transferred to the general-purpose model, with optimal efficiency in costs.

In this section, we consider the simulation between the BSP and MASC models to analyze the power of MIMDs. Theoretically, not only is simulation between parallel models an efficient way to transfer algorithms from one model to another model, but also a good method to examine the comparative power of the models related. If one model can simulate another model efficiently, any algorithm on the simulated model can be transferred to an algorithm on the simulating model through the simulation. The running time of a transferred algorithm is bounded by the cost of the algorithm on the simulated model multiplied by the simulation cost. Intuitively, BSP is more powerful that ASC. Simulating BSP using ASC requires a lot of extra cost. However, our investigation of the simulations between BSP and ASC shows that the power of MIMDs is usually established on “average-case” does not hold in “the worst-case”. It makes the common belief that MIMDs are more powerful than SIMDs seems to be limited and not applicable in all cases, especially in the field of real-time processing where a worst case is usually critical in term of meeting a deadline.

We next create a simulation of MASC by BSP by giving a series of simulations from BSP to PRAM, and then from PRAM to MASC. We use *M*(*p*) to denote a model *M* with *p* processors.

- BSP → PRAM.

It has been shown in [124] that there exist expected efficient optimal simulations between BSP and PRAM either with exclusive memory access (EREW PRAM or EPRAM) or with concurrent memory access (CRCW PRAM or CPRAM). When a simulation has *optimal efficiency*, executing an algorithm on both the simulating model and the simulated model will result in asymptotically the same running costs (or asymptotically the same amount of work). If the simulation is randomized, the efficiency is said to be *expected efficiency*. Giving a good randomization function, the author has proven that BSP(p) can simulate a EPRAM(v) with expected optimal efficiency if $v \geq p\log p$. A BSP(p) can simulate a CPRAM(v) with expected optimal efficiency if $v \geq p^{1+\varepsilon}$ (ε is a constant and $\varepsilon > 0$). Clearly, any polynomial algorithm running on PRAM can be transferred to BSP in an expected polynomial time.

- Asynchronous PRAM → synchronous PRAM

  The PRAM model referred to in [124] is assumed to be an asynchronous PRAM, in which each processor executes its own program. It strongly favors the power of asynchronous MIMD machines. In fact, as variants of synchronous PRAM, asynchronous PRAMs have been studied by many researchers in the past [11,48,34]. The best known is Gibbons' asynchronous PRAM [48]. The author has shown that a time t algorithm on EPRAM(p) can be simulated by an asynchronous PRAM(p/B) running in time O(Bt) where B is the time to synchronize all the processors used in the algorithm and it is a function of the number of the processors. Usually $B(p) \leq p$ and mostly $B(p)$ = O(log p). This simulation is a work-preserve simulation, which means that the amount of work through the simulation remains asymptotically the same.

- EPRAM → CPRAM

  It has been long known that an EPRAM(p) can simulate any variant of a CRCW PRAM(p) or CPRAM(p) in O(log p) [52]. The simulation can be either randomized or deterministic.

- CPRAM → MASC

  A CRCW PRAM(p) or CPRAM(p) can deterministically simulate MASC(p) in O(log$p$). This is because a processor of a synchronous PRAM can simulate each of PEs of the MASC. One of the PRAM processor takes the role of the only IS. Broadcast and reduction operations take O(log p). Therefore, a polynomial algorithm on MASC can be transferred to a polynomial algorithm on a synchronous PRAM.

Combining the above simulations, we have obtained a series of simulations from BSP to MASC. All of these have provided a chain of polynomial transformations for an algorithm from one model to the other with good expectation in a given simulation. An algorithm running in polynomial time on MASC can be transformed to BSP also running in polynomial time. Theoretically, this should also be applicable to a real-time scheduling algorithm. Considering the power of the BSP model that has been claimed for years, this seems to be an unarguable "theorem".

However, based on our analysis in Section 7.3, there are problems with this claim. We observe that a randomized simulation does not accurately encompass the power of the models. An average-case simulation cannot be applied to an algorithm that requires worst-case analysis. For a real-time problem like ATC, the power of MIMDs based on the average case or probabilistic expectation seems to be less meaningful. In this sense, the power of SIMDs has been underestimated.

It appears that when simulating and comparing a synchronous model and an asynchronous model (or a SIMD model and a MIMD model), there might be some factors underestimated or ignored. First, some costs on communication and synchronization are underestimated; especially for the situation when multiple processors try to access a shared resource simultaneously. Second, the existing simulation algorithms (e.g. between BSP and PRAM, between PRAM and MASC) are based on randomization and use of hash functions. They completely ignore the worst case. However, a real-time processing system requires that task execution be predictable to meet the deadline in the worst case as well as in a randomized case or an average case. One of the reasons a solution to the NP-hard multiprocessor dynamic scheduling problem is needed in real-time processing is due to having to meet the deadlines in the worst case and not just in the average case.

The NP-hardness of the multiprocessor scheduling problem is also caused by the consideration of the worst case, as an MIMD algorithm has to be designed for the case as worst as possible to guarantee its correctness that might have accumulated an exponential increasing running time.

7.6 Summary

In this chapter, we have given detailed discussion of the power of SIMDs vs MIMDs in real-time processing. We have also presented advantages of an associative SIMD and discuss difficulties MIMDs in finding an efficient solution to real-time problems. These difficulties were not fully recognized until MIMDs were applied to solve real-time problems such as ATC. As a popular MIMD model, the BSP model was intended to model most parallel systems, both SIMDs and MIMDs. However, as we have observed, BSP ineffectively models SIMDs and can result in SIMD algorithms run very inefficiently in the worst-case environment. Our research has shown that SIMDs are not outdated as MIMD researchers have pessimistically stated. They are more efficient and powerful enough to solve some real-time problems involving massive data-parallel computation. These problems are considered intractable for MIMD systems. Moreover, considering the SIMD's advantages of simple programming styles and hardware implementations, we suggest that it deserves more attention and more utilization, especially in real-time applications. The importance and efficiency of SIMD computation need to be reconsidered.

Chapter 8

Overall Summary and Future Work

We have presented our research work on evaluation of the power of the Multiple Associative Computing (MASC) model in this dissertation. The description of this work has been broken into three parts.

First, a detailed justification for timings assumed for the MASC basic (i.e., associative) operations has been given. This justification is based on the hardware implementation of the STARAN, the architectural ancestor of the MASC model, and on a comparison of these timings with the basic operation timings of other well-established models such as PRAM and MMB. This part of this research has established that assigning a constant time for the execution of the basic associative operations in the MASC model is justified. These timings are needed in order to analyze the complexity of algorithms designed for this model and to compare the running time of these algorithms to those of algorithms designed for other models.

In the second part of our work, simulation algorithms between two classes of enhanced meshes (i.e., MMB and BRM) and the MASC model are given. Simulation and comparison of MASC and RM have also been explored. It is shown that MASC(n, j) with a $\sqrt{n} \times \sqrt{n}$ mesh can simulate a $\sqrt{n} \times \sqrt{n}$ MMB in O($\sqrt{n}$ / j) time with no extra memory. In particular, the simulation takes constant time when j = $\Omega(\sqrt{n})$. Also, MASC(n, j) with a $\sqrt{n} \times \sqrt{n}$ mesh can simulate a $\sqrt{n} \times \sqrt{n}$ BRM in O(n / j) time with

O(n) extra memory. Simulation of MASC(n, j) with a $\sqrt{n} \times \sqrt{n}$ MMB or BRM takes O($jn^{1/6}$) time with O(n) extra memory. These simulations have concluded that MASC(n, 1) with a $\sqrt{n} \times \sqrt{n}$ mesh is more powerful than a $\sqrt{n} \times \sqrt{n}$ mesh with a global bus, and that MASC(n, j) with a $\sqrt{n} \times \sqrt{n}$ mesh is also more powerful than a $\sqrt{n} \times \sqrt{n}$ MMB when $j = \Omega(\sqrt{n})$. These simulations between MASC and the currently popular enhanced mesh models provide an effective way of understanding the power of the MASC model, particularly in view of the extensive research literature concerning these models. Also, the constant time simulations enable the algorithms designed for these enhanced meshes to be transferred to the MASC model with the same running time. Exploration of simulating and comparing MASC and RM (General Reconfigurable Meshes) suggests that the two models are very dissimilar. Neither of them can simulate the other efficiently due to their distinguishing features.

In the third part of this work, a feasible polynomial time solution is described for the Air Traffic Control system for the ASC model, which is the MASC model restricted to one instruction stream. The importance of this application establishes the importance of the ASC model. While it is not addressed in this dissertation, it is easy to see that the MASC model can be used to provide a faster execution of the ASC solution to the ATC problem. A consequence of this observation is that the ASC solution to the ATC problem also indicates the importance of the MASC for more than one IS. On the other hand, after over 40 years of intensive study and thousands of person years of effort, no polynomial time multiprocessor solution has been found for the ATC problem and none is expected. The above observations led to a general comparison between the power of the associative SIMD (or ASC) model and the MIMD model. It is difficult for a MIMD to provide a polynomial time solution to a real-time problem like ATC because the MIMD solutions normally use dynamic task scheduling and essentially all dynamic task scheduling problems have been shown to be NP-complete. The existence of a feasible polynomial time solution for the ATC problem for an associative SIMD leads us to identify and analyze some of the most important features of both associative SIMDs and MIMDs with regard to real-time processing. This analysis indicates that associative SIMDs are a better choice than MIMDs for applications involving real-time problems such as ATC. The results of this third part of our work raise serious questions concerning the validity of the general belief that MIMDs are more powerful than SIMDs and can solve any problems as efficiently as SIMDs. Based on this fact, we highly suggest that the importance and efficiency of SIMD computation should be reconsidered and reevaluated.

Some open problems are given in the following.

1. Timings of the basic MASC associative operations were justified based on the past machines: the STARAN and ASPRO. It would be useful to justify these conclusions using current implementation data and modern hardware design technology. There is currently a research group working on this. We expected to receive updated data from their work.

2. Prefix sums and the more restrictive operation of global summation are frequently used in parallel computation. The current MASC model does not support these operations. Although we briefly discussed a possible implementation of prefix sums in an earlier chapter, it is worth exploring implementing these operations in hardware in more detail. Giving the current assumption of the MASC model, it is possible to implement the prefix sum operation in constant time.

3. Simulations between MASC and the enhanced mesh models have been established. It would be interesting to prove the optimality of particularly the MASC simulation of MMB.

4. Simulations between MASC and RM should be further explored. It would be interesting to provide an example of a problem that is easy for MASC to solve but difficult for RM and establish a lower bound for a simulation of MASC by RM. Another open question for this simulation is the possibility to use different sizes of the two models to establish a simulation between them.
5. With regard to the associative processor solution of the ATC problem, it would be useful to be able to obtain experimental data using a modern hardware implementation or simulation of ASC.
6. A document that gives a much more detailed account of the ASC solution for the ATC is needed. This more detailed document would address additional algorithms and issues that are not covered in this dissertation.

APPENDIX A

Glossary

ALU: an Arithmetic and Logic Unit

AP: an Associative Processor or an ASC computer

ASC: the Associative Computing Model, or MASC with one instruction stream

ASPRO: A parallel associative processor using the STARAN architecture, produced by Martin-Marietta in 1980s and used by the U.S. Navy

ATC: Air Traffic Control

BDN: Bounded Degree Network

BPRAM: Block PRAM

BRM: Basic Reconfigurable Meshes

BSP: the Bulk Synchronous Parallel model

CPRAM: CRCW PRAM

CRCW: Concurrent Read Concurrent Write

CT: Communication Throughput

EDF: Earlier Deadline First, a scheduling algorithm for real-time tasks

EPRAM: EREW PRAM

EREW: Exclusive Read Exclusive Write

FAA: the Federal Aircraft Administration

FPLD: Field-Programmable Logic Device

IFR: Instrument Flight Rules

IS: Instruction Stream

LLF: Least Laxity First, a scheduling algorithm for real-time tasks

LPRAM: Local-Memory PRAM

MASC: Multiple Associative Computing model

MASC(n, j): a MASC model with n PEs and j ISs.

MDA: Multi-Dimensional Access

MHB: Meshes with Hybrid Buses

MIMD: Multiple Instruction-stream Multiple Date-stream

MMB: Meshes with Multiple Buses

MP: Multiprocessor, usually referred to a shared-memory MIMD system. In our work, it is referred to as a general MIMD system (including a distributed memory multi-computer system)

MPC: Module Parallel Computer

MSIMD: Multiple-SIMD

PE: Processing Element

PPRAM: Phase PRAM

PRAM: the Parallel Random Access Machine

PRAM(*m*): A PRAM with *m* shared-memory locations

PRAM(*n*, *m*): A PRAM with *n* processors and *m* shared-memory locations

RAM: the Random Access Machine

RM: Rate-Monotonic, a scheduling algorithm for real-time tasks

QRQW: Queue Read Queue Write

QSM: the Queuing Shared-Memory model

RM: Reconfigurable Meshes

SIMD: Single Instruction-stream Multiple Data-stream

STARAN: A parallel associative processor using multi-dimensional access memories and flip networks, built by Goodyear Aerospace Corporation in 1970s

VFR: Visual Flight Rules

XPRAM: A PRAM that performs its execution in supersteps, an early version of BSP

Bibliography

[1] N. Abu-Ghazaleh, P. Wilsey, J. Potter, R. Walker, J. Baker, Flexible Parallel Processing in Memory: Architecture + Programming Model, *Proceedings of the 3rd Petaflow Workshop*, February 1999, 7pgs, http://vlsi.mcs.kent.edu/~parallel/papers/tpf3.ps

[2] M. Adler, J.W. Byers, Parallel Sorting with Limited Bandwidth, *SPAA'95*, Santa Barbara, CA, USA, 1995, pp. 129-136

[3] M. Adler, P.B. Gibbons, Y. Matias and V. Ramachandran, Modeling Parallel Bandwidth: Local vs. Global Restrictions, *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures,* June 1997

[4] A. Agbaria, Y. Ben-Asher and I. Newman, Communication-Processor Tradeoffs in Limited Resources PRAM, *Proc. of SPAA'99,* Saint Malo, France, pp.74-82

[5] A. Aggarwal, Optimal Bounds for Finding Maximum on Array of Processors with *k* Global Buses, *IEEE Trans. on Computers*, Vol. 35, 1986, pp.62-64

[6] A. Aggarwal, A.K. Chandra and M. Snir, On Communication Latency in PRAM Computations, *Proc. 1st ACM Symp. On Parallel Algorithms and Architectures*, pp.11-21, June 1989

[7] A. Aggarwal, A.K. Chandra and M. Snir, Communication Complexity of PRAMs, *Theoretical Computer Science*, 71(1): 3-28, 1990

[8] S. G. Akl and G. Guenther, Broadcasting with Selective Reduction, *Proceedings of the 11the IFIP Congress*, San Francisco, 1989, pp.515-520

[9] S. G. Akl and I. Stojmenovic, Broadcasting with Selective Reduction: A Powerful Model of Parallel Computation, Ed. A. Zomaya, *Parallel and Distributed Computing Handbook*, McGraw-Hill, 1996, pp.192-222

[10] S. G. Akl, *Design and Analysis of Parallel Algorithms*, Prentice Hall, New York, 1989

[11] S. G. Akl, *Parallel Computing: Models and Methods*. Prentice Hall, New York, 1997

[12] S. G. Akl, The design of efficient parallel algorithms, in *Handbook on Parallel and Distributed Processing*, Blazewicz, J., Ecker, K., Plateau, B., and Trystram, D., Eds., Springer Verlag, Berlin, 2000, pp. 13 -91

[13] M. Atwah and J. W. Baker, An Associative Dynamic Convex Hull Algorithm, *Proc. of the 10th IASTED International Conference on Parallel and Distributed Computing Systems*, 1998, pp. 250-254

[14] M. Atwah, J. W. Baker and S. G. Akl, An Associative Implementation of Classical Convex Hull Algorithms, *Proceedings of the 8th IASTED International Conference on Parallel and Distributed Computing Systems*, 1996, pp. 435-438

[15] J. W. Baker and M. Jin, Simulation Between Enhanced Meshes and the Multiple Associative Computing (MASC) Model, *Proc. of the 1999 Midwest Workshop on Parallel Processing*, Aug. 1999

[16] J. W. Baker and M. Jin, Simulation of Enhanced Meshes with MASC, a MSIMD Model, *Proc. of the 11th International Conference on Parallel and Distributed Computing Systems*, Nov., 1999, pp. 511-516

[17] A. Bar-Noy and D. Peleg, Square Meshes Are Not Always Optimal, *Proc. of 1st SPAA*, June 1989, pp.138-147

[18] K. Batcher, STARAN Parallel Processor System Hardware, *Proc. Of the 1974 National Computer Conference* (1974), pp. 405-410

[19] K. Batcher, STARAN Series E, *1977 International Conference on Parallel Processing*, pp. 140-143

[20] K. Batcher, The Multidimensional Access Memory in STARAN, *IEEE Trans. on Computers*, Feb. 1977, pp. 174-177

[21] K. Batcher, The Flip Network in STARAN, *1976 International Conference on Parallel Processing*, pp. 65-71

[22] K. Batcher, Design of a Massively Parallel Processor, *IEEE Trans. On Computers*, C-29, 1980, pp. 836

[23] K. Batcher, The Resolver Network, seminar notes, October, 1999

[24] Y. Ben-Asher, D. Gordon and A. Schuster, Efficient Self-Simulation Algorithms for Reconfigurable Arrays, *J. of Parallel and Distributed Computing*, Vol. 30, 1995, pp.1-22

[25] Y. Ben-Asher, D. Peleg, R. Ramaswami and A. Schuster, The Power of Reconfiguration, *J. of Parallel and Distributed Computing*, 13, 1991, pp. 139-153

[26] D. Bhagavathi, H. Gurlar, S. Olariu and J. L. Schwing, L. Wilson and J. Zhang, Time- and VLSI-optimal Sorting on Enhanced Meshes, *IEEE Trans. on Parallel and Distributed Systems*, to appear 1998

[27] D. Bhagavathi, S. Olariu, W. Shen, L. Wilson, A Unifying Look at Semigroup Computations on Meshes with Multiple Broadcasting, *Parallel Processing Letters*, Vol. 4, 1994, pp. 73-82

[28] D. Bhagavathi, S. Olariu, W. Shen, and L. Wilson, A Time-Optimal Multiple Search Algorithm on Enhanced Meshes, with Applications, *Journal of Parallel and Distributed Computing*, 22, (1994), pp.113-120

[29] G. Bilard, K. Herley, A. Pietracaprina, G. Pucci and P. Spirakis, BSP versus LogP, *Algorithmica* (1999) 24, pp.405-422

[30] T. Blank, J. Nickolls, A Grimm Collection of MIMD Fairy Tales, *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*, pp. 448-457, October, 1992

[31] S. Bokhari, Finding Maximum on an Array Processor with a Global Bus, *IEEE Trans. On Computers,* C-33, 2 (1984), pp.133-139

[32] V. Bokka, H. Gurla, S. Olariu, and J. Schwing, Time and VLSI-Optimal Convex Hull Computation on Meshes with Multiple Broadcasting, *Information Processing Letters*, 56, (1995), 273-280

[33] K. Chung, Prefix Computations on a Generalized Mesh-Connected Computer with Multiple Buses, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6, No. 2, Feb. 1995, pp. 196-199

[34] R. Cole and O. Zajicek, The APRAM: Incorporating Asynchrony into the PRAM Models, *Proc. of 1st ACM Symp. of Parallel Algorithms and Architectures*, June 1989, pp. 169-178

[35] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T.Eicken, LogP: Towards a Realistic Model of Parallel Computation, *Proc. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming,* May 1993, pp. 1-12

[36] M. Dietzfelbinger, M. Mkutylowski, and R. Reischuk, Feasible Time-optimal Algorithms for Boolean Functions on Exclusive-write Parallel Random-access Machines, *SIAM J. Computing*, Vol. 25, No. 6, Dec. 1996, pp.1196-1230

[37] M. Esenwein, J. W. Baker, VLCD String Matching for Associative Computing and Multiple Broadcast Mesh, *Proceedings of the 9th IASTED International Conference on Parallel and Distributed Computing Systems*, 1997, pp. 69-74

[38] J. A. Fernandez-Zepeda, R. Vaidyanathan, and J. Trahan, Scaling Simulation of the Fusing-Restricted Reconfigurable Mesh, *IEEE Trans. On Parallel and Distributed Systems*, Vol. 9, No. 9, Sep. 1998, pp. 861-871

[39] J. A. Fernandez-Zepeda, R. Vaidyanathan, and J. Trahan, Improved Scaling Simulation of the General Reconfigurable Mesh, *Proc. of 1999 Symposium of Parallel and Distributed Processing, IPPS/SPDP Workshops*, San Juan, Puerto Rico, USA, April, 1999, pp. 616-633

[40] J. Fix and R. E. Ladner, Sorting by Parallel Insertion on a One-dimensional Subbus Array, *IEEE Trans. on Computers,* Vol. 47, No. 11, Nov., 1998, pp. 1267-1281

[41] M. Flynn, Some computer organizations and their effectiveness, *IEEE Transactions on Computers*, pp. 948-960, Sep., 1972

[42] S. Fortune, Parallelism in Random Access Machines, *Proc. Of the 10th ACM Symp. On Theory of Computing,* May, 1978

[43] G. C. Fox, What Have We Learned from Using Real Parallel Machines to Solve Real Problems? Technical Report C3P-522, Cal Tech, December 1989

[44] P. Fragopoulou, On the Comparative Powers of the 2D-PARBUS and the CRCW-PRAM Models, *Parallel Processing Letters*, Vol. 3, 1993, pp. 301-304

[45] M.R.Garey, R.L.Graham and D. S. Johnson, Performance Guarantees for Scheduling Algorithms, *Operations Research,* Vol. 26, No. 1, Jan,-Feb. 1978 pp 3-21

[46] M. R. Garey and D. S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-completeness*, W.H. Freeman, New York, 1979

[47] A.V. Gerbessiotis and L. G. Valiant, Direct Bulk-Synchronous Parallel Algorithms, *J. of Parallel and Distributed Computing*, Vol. 22, 1994, pp.251-267

[48] P.B. Gibbons, A More Practical PRAM Model, *Proc. of 1st ACM Symp. on Parallel Algorithms and Architectures*, pp.158-168, June 1989

[49] P.B. Gibbons, Y. Matias, and V. Ramachandran, Can a Shared-memory Model Serve as a Bridging Model for Parallel Computation? *Proc. of 9th ACM Symp. on Parallel Algorithms and Architectures*, June 1997, pp.72-83

[50] P. B. Gibbons, Y. Matias, and V. Ramachandran, The Queue-Read Queue-Write PRAM Model: Accounting For Contention in Parallel Algorithms, *SIAM J. on Computing,* Vol. 28, No. 2, 1998, pp. 733-769

[51] R. Gupta, P. Balaji, D.K. Panda and J. Nieplocha, Efficient Collective Operations using Remote Memory Operations on VIA-Based Clusters, *Proc. of the 16th International Parallel and Distributed Processing Symposium*, April 2003

[52] T. Harris, A Survey of PRAM Simulation Techniques, *ACM Computing Surveys*, Vol. 26, No. 2, June 1994, pp. 187-206

[53] M. Herbortdt and C. Weems, Associative, Multiassociative, and Hybrid Processing, *Reading in Associative Processing and Processors*, A. Kriskelis, ed., IEEE Computer Society Press

[54] J. L. Hennessey and D. A. Patterson, *Computer Architecture: A Quantitative Approach,* 2nd Edition, Morgan Kaufmann Publishers, 1996

[55] W. D. Hillis and G. L. Steele, Jr., Data Parallel Algorithms, *Communication of ACM*, Vol. 29, No. 12, Dec. 1986, pp. 1170-1182

[56] R. M. Hord, *Parallel Super-computing in SIMD Architectures*, CRC Press, Inc., Boca Raton, Florida, 1990

[57] J. Jaja, *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing, 1992

[58] K. Jeffay, Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints, *Proc. of the 10th IEEE Real-Time Systems Symposium*, Santa Monica, CA, December 1989, pp. 295-305

[59] K. Jeffay, D. F. Stanat, and C. U. Martel, On Non-Preemptive Scheduling of Periodic and Sporadic Tasks, *Proc. of the 12th IEEE Real-Time Systems Symposium*, San Antonio, TX, December 1991, pp. 129-139

[60] M. Jin, J. Baker, and K. Batcher, Timings of Associative Operations on the MASC model, *Proc. of the 15th International Parallel and Distributed Processing Symposium* (*Workshop in Massively Parallel Processing*), April 2001

[61] M. Jin, J. Baker, and W. Meilander, The Power of SIMD vs. MIMD in Real-time Processing, *Proc. of the 16th International Parallel and Distributed Processing Symposium* (*Workshop in Massively Parallel Processing*), April 2002

[62] B. H. Juurlink and H. A. Wijshoff, A Quantitative Comparison of Parallel Computation Models, *ACM Trans. on Computer Systems*, Vol. 16, No. 3, Aug., 1998, pp. 271-318

[63] R. M. Karp and V. Ramachandran, Parallel Algorithms for Shared Memory Machines, J. van Leeuwen ed., *Handbook of Theoretical Computer Science*, Vol. A, The MIT Press/Elsevier Science, New York, 1990

[64] R. M. Karp, A Sahay, E. E. Santo, and K. E. Schauser, Optimal Broadcast and Summation in the LogP Model, *Proceedings of the 5th Symposium on Parallel Algorithms and Architectures*, 1993, pp.142-153

[65] M.H. Klein, J.P. Lehoczky, and R. Rakumar, "Rate-Monotonic Analysis for Real-Time Industrial Computing", IEEE Computer, pp. 24-33, 1994

[66] C. M. Krishna and Kang G. Shin, *Real-time Systems*, McGraw-Hill, 1997

[67] I. Koren, *Computer Arithmetic Algorithms*, Prentice Hall, New Jersey, 1993

[68] R. Lea and I. Jalowiecki, Associative Massively Parallel Computers, *Proceedings of the IEEE*, Vol. 79, No. 4, April 1991

[69] H. Li and Q. Stout, Reconfigurable SIMD Massively Parallel Computers, *Proceedings of the IEEE*, Vol. 79, No. 4, April 1991

[70] K. Li, Y. Pan, and S.-Q. Zheng, eds., *Parallel Computing Using Optical Interconnections* , Kluwer Academic Publishers, Boston, USA, Hardbound, ISBN 0-7923-8296-X, October 1998

[71] R. Lin, S. Olariu, Simulating Enhanced Meshes with Applications, *Parallel Processing Letters*, 3(1), pp. 59-70, 1993

[72] R. Lin, S. Olariu, J. Schwing, The Mesh with Hybrid Buses: an Efficient Parallel Architecture for Digital Geometry, *IEEE Trans. On Parallel and Distributed Systems*, Vol. 10, No. 3, March 1999, pp.226-279

[73] C. L. Liu and J.W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *Journal of the ACM,* 20(10), 1973

[74] ASPRO-VME Hardware/Architecture ER3418-5 LORAL Defense Systems Akron, OH June 1992

[75] T. Maeba, S. Tatsumi, and M. Sugaya, Algorithms for Finding Maximum and Selecting Median on a Processor Array with Separable Global Buses, *IEICE Trans. A*, J72-A(6):950-958, 1989

[76] Y. Mansour, N. Nisan and U. Vishikin, Trade-offs between Communication Throughout and Parallel Time, *Proc. 26th Annual ACM Symp. on Theory of Computing,* 1994, pp.372-381

[77] M. Maresca, Polymorphic Processor Arrays, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No.5, May, 1993, pp. 490-506

[78] Y. Matias and A. Schuster, Fast, Efficient Mutual and Self Simulation for Shared Memory and Reconfigurable Mesh, *Parallel Algorithms And Applications*, 8, 1996, pp.195-221

[79] S. Matsumae and N. Tokura, Simulating a Mesh with Separable Buses by a Mesh with Partitioned Buses, *Proc. of the 4th International Symposium on Parallel Architectures, Algorithms, and Networks*, Fremantle, Australia, June 1998

[80] W. C. Meilander, STARAN an associative approach to multiprocessing, Multiprocessor Systems, *Infotech State of the Art Reports*, Infotech International 1976, pp 347-372

[81] W. C. Meilander, J. L. Potter, K. J. Liszka and J. W. Baker, Real-Time Scheduling in Command and Control, *Proc. of the 1999 Midwest Workshop on Parallel Processing*, Aug. 1999

[82] W.C. Meilander, J.W. Baker, M. Jin, Predictable Real-Time Scheduling for Air Traffic Control, *Fifteenth International Conference on Systems Engineering*, August 2002, pp 533-539 (Unofficial version:http://vlsi.mcs.kent.edu/~parallel/papers)

[83] W.C. Meilander, J.W. Baker, M. Jin, Importance of SIMD Computation Reconsidered, I*PDPS 2003, MPP workshop,* IEEE Digital Library (Unofficial version: http://vlsi.mcs.kent.edu/ ~parallel/papers)

[84] W.C. Meilander, M. Jin, J.W. Baker, Tractable Real-Time Control Automation, *Proc. of the 14th IASTED Inte'l Conf. on Parallel and Distributed Systems (PDCS 2002)*, pp. 483-488 (Unofficial version:http://vlsi.mcs.kent.edu/~parallel/papers)

[85] R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout, Meshes with Reconfigurable Buses, *Proc. of the 5th MIT Conference on Advanced Research in VLSI*, Boston, 1988, pp. 163-178

[86] R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout, Parallel Computations on Reconfigurable Meshes, *IEEE Trans. on Computers*, Vol. 42, No. 6, June 1993, pp. 678-691

[87] C. Murthy and G. Manimaran, *Resource Management in Real-Time Systems and Networks*, The MIT Press, Cambridge, Massachusetts, 2001

[88] K. Nakano, A Bibliography of Published Papers on Dynamically Reconfigurable Architectures, *Parallel Processing Letters 5*, 1 (1995), pp. 111-124

[89] M. Nigam and S. Sahni, Sorting n^2 Numbers on $n\times n$ Meshes, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6, No. 12, Dec., 1995, pp. 1221-1225

[90] N. Nishimura, Asynchronous Shared Memory Parallel Computation, SPAA'90, pp.76-84

[91] S. Olariu, J. Schwing and J. Zhang, On the Power of Two-dimensional Processor Arrays with Reconfigurable Bus Systems, *Parallel Processing Letters*, vol. 1, No. 1 (1991), pp. 29-34

[92] S. Olariu and I. Stojmenovic, A Time-Optimal All-Nearest Neighbor Algorithm on Meshes with Multiple Broadcasting, *Proc. Of International Conference on Parallel Processing Systems*, 1994

[93] B. Parhami, *Computer Arithmetic Algorithms and Hardware Design*, Oxford University Press, New York, 2000

[94] B. Parhami, Search and Data Selection Algorithms for Associative Processors, *Associative Processing and Processors*, edited by A. Krikelis and C. C. Weems, IEEE Computer Society; July 1997

[95] B. Parhami, SIMD Machines: Do They Have a Significant Future? *Report on a Panel Discussion at The 5th Symposium on the Frontier of Massively Parallel Computation,* McLean, LA, Feb., 1995

[96] D. Parkinson, D. Hunt, and K. MacQueen, The AMT DAP 500, *Proc. of the 33rd IEEE Computer Society International Conference*, 1988, pp. 196-199

[97] F. Pfister, *In Search of Clusters*, Prentice Hall, New Jersey, 1995

[98] J. Potter, Associative Computing: A Programming Paradigm for Massively Parallel Computers, New York; Plenum Press, 1992

[99] J. Potter, J. Baker, S. Scott, A. Bansal, C. Leangsuksun, and C. Asthagiri, ASC: An Associative-Computing Paradigm, Computer, 27(11), 1994, 19-25

[100] V.K. Prasanna-Kumar and C.S. Raghavendra, Array Processor with Multiple Broadcasting, *J. of Parallel Distributed Computing,* 4:173-190, 1987

[101] V.K. Prasanna-Kumar and D. I. Reisis, Image Computations on Meshes with Multiple Broadcast, *IEEE Trans. on Pattern Analysis and Machine Intelligence,* Vol. 11, No. 11, 1989, pp1194-1201

[102] L. Qian, *Complexity Analysis of an Air Traffic Control System Using an Associative Processor*, Master's Thesis, Kent State University, 1997

[103] Michael J. Quinn, *Parallel Computing: Theory and Practice***,** Second Edition, McGraw-Hill, New York, 1994, ISBN 0-07-051294-9

[104] V. Ramachandran, B. Grayson, and M. Dahlin, Emulation Between QSM, BSP and LogP: A framework for General-Purpose Parallel Algorithm Design, UTCS Technical Report TR98-22, Nov. 1998

[105] K. Ramamritham, J. A. Stankovic, and W. Zhao, Distributed Scheduling of Tasks with Deadlines and Resource Requiremnets, *IEEE Trans. On Computers*, Vol. 38, No. 8, August 1989, pp.1110-1123

[106] R. Reischuk, Simultaneous WRITES of Parallel Random Access Machine Do Not Help Compute Simple Arithmetic Functions, *Journal of ACM*, Vol. 34, No. 1, January 1987, pp. 163-178

[107] J. A. Rudolph, A Production Implementation of an Associative Array Processor – STARAN, *The Fall Joint Computer Conference (FJCC)*, December 5-8, 1972, Los Angeles, California, USA

[108] J. A. Stankovic, Misconceptions about Real-time Computing, *IEEE Computer*, Vol. 21, No. 10, October 1988, pp. 17-25

[109] J.A. Stankovic, Son and Hansson, Misconceptions About Real-Time Databases, *Computer*, June 1999

[110] J. A. Stankovic, M. Spuri, M. Di Natale, G. C. Buttazzo, Implications of Classical Scheduling Results for Read-Time Systems, *Computer*, June 1995, pp. 16-25

[111] J. A. Stankovic, M. Spuri, K. Ramamritham and G. C. Buttazzo, *Deadline Scheduling for Real-time Systems*, Kluwer Academic Publishers, 1998

[112] Q. Stout, Mesh-connected Computers with Broadcasting, *IEEE Trans. On Computers*, vol. C-32, No.9, Sept. 1983, pp. 826-830

[113] Q. Stout, Meshes with Multiple Buses, *Proc. of the 27th Annual Symposium of Foundations of Computer Science,* Oct, 1986, pp. 264-273

[114] Q. Stout, Ultrafast Parallel Algorithms and Reconfigurable Meshes, *Proc. of DARPA Software Technology Conference 1992*, pp. 184-188

[115] J. L. Trahan, R. Vaidyanathan, and R. K. Thiruchelvan, On the Power of segmenting and Fusing Buses, *J. of Parallel and Distributed Computing,* 34, (1996), pp. 82-94

[116] M. Torngren, Fundamentals of Implementing Real-time Control Applications in Distributed Computer Systems, *Real-time Systems*, 14, 1998, pp. 219-250

[117] D. Ulm, J. Baker, Simulating PRAM with a MSIMD Model (ASC), *Proceedings of the International Conference on Parallel Processing*, 1998, pp.3-10

[118] D. Ulm, J. Baker, Virtual Parallelism by Self-simulation of the Multiple Instruction Stream Associative Model, Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 1421-1430, 1996

[119] D. Ulm, Establishing the Power of Associative Model (ASC) Through Simulations with PRAM and Virtual Parallelism, Ph.D. Dissertation, Department of Math and Computer Science, Kent State University, Aug. 1998

[120] April 19 *USA Today editorial*

[121] R. Vaidyanathan and J. L. Trahan, Optimal Simulation of Multidimensional Reconfigurable Meshes By Two-dimensional Reconfigurable Meshes, *Information Processing Letters 47*, (1993) pp. 267-273

[122] L. G., Valiant, A Scheme for Fast Parallel Communication, *SIAM J. Computing*, 11(1982), pp.350-361

[123] L. G. Valiant, A Bridging Model for Parallel Computation, *Comm. Of ACM*, Vol. 33, No. 8, 1990, pp. 103-111

[124] L. G. Valiant, General Purpose Parallel Architectures, J. van Leeuwen ed., *Handbook of Theoretical Computer Science*, Vol. A, The MIT Press/Elsevier, New York, 1990, pp. 943-971

[125] U. Vishkin and A. Wigderson, Trade-offs Between Depth and Width in Parallel Computation, *SIAM J. Computing*, 14(2): 303-314, 1985

[126] R. A. Walker, J. Potter, Y. Wang, and M. Wu, Implementing Associative Processing: Rethinking Earlier Architectural Decisions, *Proc. of the 15th International Parallel and Distributed Processing Symposium* (*Workshop in Massively Parallel Processing*), April 2001

[127] B. Wang, G. Chen, Two-dimensional Processor Array with Reconfigurable Bus System Is At Least As Powerful As CRCW Model, *Information Processing Letters,* 36 (1990), pp. 31-36

[128] F. Wang, K. Ramamritham, and J. A. Stankovic, Bounds on the Performance of Heuristic Algorithms for Multiprocessor Scheduling of Hard Real-Time Tasks, *Proceedings of Real-time Systems Symposium*, 1992, pp.136-145

[129] H. Wang, and R. A. Walker, Implementing a Scalable ASC Processor, *Proc. of the 17th International Parallel and Distributed Processing Symposium* (*Workshop in Massively Parallel Processing*), April 2003

[130] C. Weems and A. Krikelis, Associative Processing and Processors, (Guest Editor's Introduction to a special issue), *IEEE Computer*, Vol. 27, No. 11, November 1994, pp.12-17

[131] M. Wu, R. A. Walker, and J. Potter, Implementing Associative Search and Responder Resolution, *Proc. of the 16th International Parallel and Distributed Processing Symposium* (*Workshop in Massively Parallel Processing*), April 2002

[132] J. Yang, $2 Billion Control System Not Ready for Use, *ABC News*, June 5, 2002 http://abcnews.go.com/sections/ wnt/DailyNews/yang_faa020605.html

[133] S. Yau and H. Fung, Associative Processor Architecture -- a Survey, *Computing Surveys*, Vol. 9, No. 1, March 1977, pp. 3-27