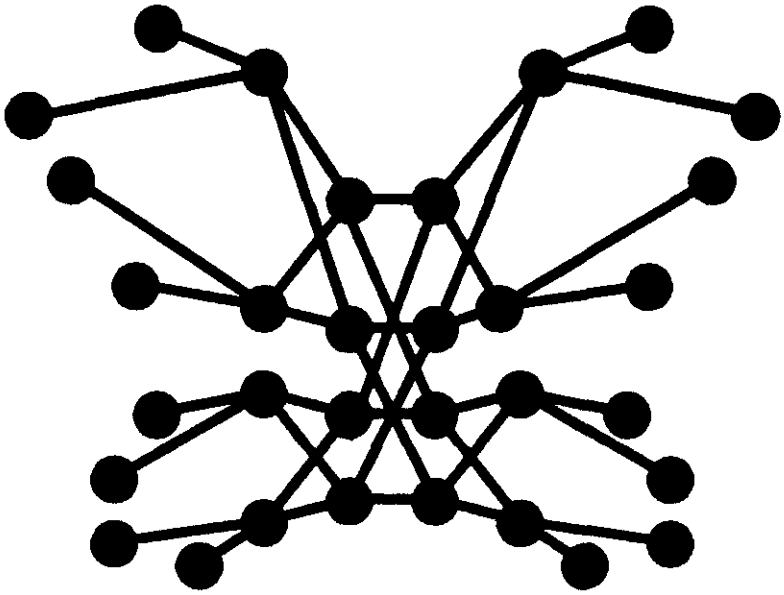

Butterfly Project Report
22

Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor

T. J. LeBlanc, M.L. Scott and C.M. Brown

September 1988

Computer Science Department
University of Rochester
Rochester, NY 14627

Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor

Thomas J. LeBlanc, Michael L. Scott, and Christopher M. Brown

Department of Computer Science
University of Rochester
Rochester, NY 14627

ABSTRACT

For three years, members of the Computer Science Department at the University of Rochester have used a collection of BBN Butterfly™ Parallel Processors to conduct research in parallel systems and applications. For most of that time, Rochester's 128-node machine has had the distinction of being the largest shared-memory multiprocessor in the world. In the course of our work with the Butterfly we have ported three compilers, developed five major and several minor library packages, built two different operating systems, and implemented dozens of applications. Our experience clearly demonstrates the practicality of large-scale shared-memory multiprocessors, with non-uniform memory access times. It also demonstrates that the problems inherent in programming such machines are far from adequately solved. Both locality and Amdahl's law become increasingly important with a very large number of nodes. The availability of multiple programming models is also a concern; truly general-purpose parallel computing will require the development of environments that allow programs written under different models to coexist and interact. Most important, there is a continuing need for high-quality programming tools; widespread acceptance of parallel machines will require the development of programming environments comparable to those available on sequential computers.

1. Introduction

In September 1984, the Department of Computer Science at the University of Rochester acquired a 3-node BBN Butterfly™ Parallel Processor [5]. In May 1985, with funding from an NSF CER grant, the department acquired a 128-node Butterfly, the largest configuration in existence. Over the last 3.5 years, the Butterfly has been used by various members of the department to develop numerous software packages and applications. This paper traces the history of

This work was supported in part by NSF CER grant number DCR-8320136, NSF grant number CCR-8704492, DARPA/ETL contract number DACA76-85-C-0001, DARPA/ONR contract number N00014-82-K-0193, an ONR Young Investigator Award, contract number N00014-87-K-0548, and an IBM Faculty Development Award.

This paper was presented at the ACM SIGPLAN PPEALS Conference (Parallel Programming: Experience with Applications, Languages, and Systems), New Haven, CT, 19-21 July 1988.

our software development for the Butterfly and describes our collective experience using the world's largest shared-memory multiprocessor.

For us the Butterfly and its systems software represented two unique scientific opportunities. First, it was flexible enough to support the implementation of our new ideas; second, it incorporated several interesting solutions to problems that themselves represented research issues. Our findings are based on a computer that is now a generation removed from the current product line. In any case, this is not a product review. We wish first to document ideas and concerns that are shaping the evolution of parallel computing. Second, and more important, we believe that our experiences are still intellectually relevant and that they will be useful in future contexts when similar or related issues arise.

Our experiences fall into several categories. In section 2 we describe our experience with the hardware and software provided by BBN. Section 3 describes the history of the Butterfly software development effort at the University of Rochester, including both systems software and applications. Section 4 discusses lessons we have learned along the way and section 5 provides a summary of our experience and an assessment of the future of Butterfly-like machines.

2. BBN Butterfly Hardware and Software

The original Butterfly Parallel Processor (the "Butterfly-1") was developed by BBN Laboratories in the late 1970's as part of a research project funded by the Defense Advanced Research Projects Agency. It eventually evolved into a commercial product, marketed for the past two years by BBN Advanced Computers, Inc. It has recently been succeeded by a second generation of hardware and software, the Butterfly 1000 series, announced in October of 1987. The experiences reported in this paper were obtained in the course of work with the original version of the Butterfly. Much of our current research has moved to the new machine.

2.1. Butterfly Hardware

The Butterfly Parallel Processor (figure 1) consists of up to 256 processing nodes connected by a high-speed switching network. Each node in the switching network is a 4-input, 4-output switch with a bandwidth of 32 Mbits/sec. In the Butterfly-1, each processing node is an 8 MHz MC68000 with 24 bit virtual addresses and up to 1 Mbyte of local memory (4 Mbytes with additional memory boards). A 2901-based bit-slice co-processor called the processor node controller (PNC) interprets every memory reference issued by the 68000 and is used to communicate with other nodes across the switching network. All of the memory in the system resides on individual nodes, but any processor can address any memory through the switch. The Butterfly is therefore a NUMA[1] machine; remote memory references (reads) take about 4 μs, roughly five

[1] Non-Uniform Memory Access time. Unlike UMA multiprocessors (with uniform access times), NUMA machines have the potential to scale to very large numbers of nodes.

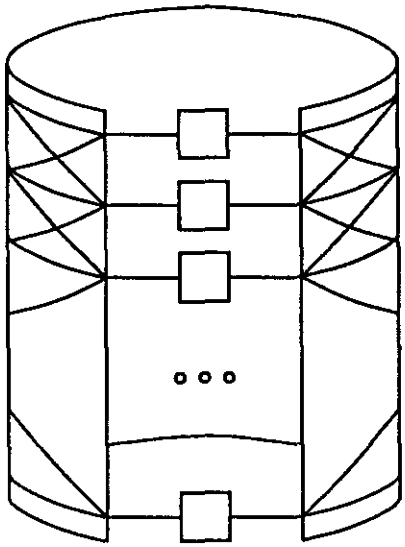

Figure 1: Butterfly Parallel Processor

times as long as a local reference.

The PNC on the Butterfly-1 implements a segmented virtual memory. Each virtual address contains an 8 bit segment number and a 16 bit offset within the segment. A process can have at most 256 segments in its address space, each of which can be up to 64 Kbytes in size. Each segment is represented by a SAR (Segment Attribute Register), which defines the base, extent, and protection of the associated memory. A process's virtual address space is represented by an ASAR (Address Space Attribute Register), which defines the address and extent of a group of SARs corresponding to the segments addressable by the process. There are 512 32-bit SARs and 1 16-bit ASAR per processor. Chrysalis allocates the available SARs in blocks of 8, which are arranged in a buddy system. Three bits in the ASAR are used to specify the size of the SAR block, which must be one of 8, 16, 32, 64, 128, or 256.

The two most serious problems with the original hardware proved to be its software floating point and its primitive memory management. Hardware floating point was provided by BBN in 1986, using a daughter board containing an MC68020 processor and MC68881 floating point co-processor,[2] but the shortcomings of the memory architecture remained. Even though the physical address space of the machine is 1 Gbyte, the virtual address space of a process could include

[2] We have upgraded 8 spare nodes and 8 nodes from our large Butterfly to provide the Department with a 16-node floating point machine.

at most 16 Mbytes of memory (256 segments, each containing 64 Kbytes), and then only if there were at most two processes per processor. This limitation forced the programmer to modify the address space of processes dynamically (at a cost of over 1 *ms* per segment added or deleted) and to avoid using shared memory whenever possible. It also limited severely the number of processes that could be allocated to a processor. In our experience the need to manage SARs explicitly has been a recurring source of irritation and problems.

Another problem with the memory architecture is that remote references steal memory cycles from the local processor. If many processors busy-wait on a shared location (a common synchronization technique), the impact on the processor containing the memory can be substantial.

Most of the problems just described have been addressed in the design of the Butterfly Plus, the hardware base for the Butterfly 1000 series [3]. Each node in the Butterfly Plus has an MC68020 processor, MC68881 floating point co-processor, and MC68851 memory management unit. The MC68020 and MC68851 on the Butterfly Plus enable demand paging, but remote references still steal memory cycles from the local processor. The presense of a modern memory management system makes the new machine an extremely attractive vehicle for research in operating systems.

2.2. Chrysalis Operating System

First-generation Butterfly machines use BBN's own Chrysalis operating system [4]. Chrysalis was originally developed for real-time packet-switching applications. Its facilities constitute a protected subroutine library for C programs. They include operations for process management, memory management, and interprocess communication. Many of the most common operations, including much of the process scheduler, are implemented in microcode in the PNC.

A Chrysalis process is a conventional heavyweight entity with its own address space. Processes are scheduled by the kernel. They do not migrate. The memory space of a process consists of a collection of *memory objects*, each of which can range in size from zero to 64 Kbytes.[3] Each process, when created, is allocated a static portion of the SARs on its node. One SAR is consumed by each memory object in the process's address space. SAR contents can be changed dynamically; explicit operations permit the current process to change its address space by mapping and unmapping arbitrary memory objects.

---

[3] Actually, segments can only be allocated in 16 standard sizes. An odd-sized memory object is rounded up to the next standard size, with an inaccessible fragment at the end.

Atomic memory operations can be used to implement spin locks. In addition, Chrysalis provides highly efficient mechanisms for scheduler-based synchronization. *Events* resemble binary semaphores on which only one process (the *owner*) can wait. The process that posts an event can also provide a 32-bit datum that will be returned to the owner by the wait operation. *Dual queues* are a generalization of events that can hold the data from multiple posts and can supply that data to multiple waiters. Microcode implementation of events and dual queues allows all of the basic synchronization primitives to complete in only tens of microseconds.

All of the basic Chrysalis abstractions (processes, memory objects, events, and dual queues) are subsumed by a single *object model.* Among other things, this model supports a uniform *ownership* hierarchy with reference counts that allows the operating system to reclaim the resources used by subsidiary objects when a parent is deleted. Unfortunately, a facility for transferring ownership to "the system" makes it easy to produce objects that are never reclaimed. Chrysalis tends to leak storage.

Chrysalis incorporates an exception-handling mechanism patterned after the MacLISP *catch* and *throw* [39]. Exception handlers are implemented with C macros that save information for non-local gotos. In the event of an error, whether detected by hardware (in a trap handler) or software (in a kernel call or user program), Chrysalis unwinds the stack to the nearest exception handler and optionally suspends the process for examination by a debugger. At first glance the catch/throw mechanism appears to be an extremely attractive way of managing errors. Unfortunately, it suffers from several limitations. First of all, it is highly language-specific. To program in Modula-2, one must insert an extra subroutine (written in C) around every system call in order to catch and handle throws. Even in C, the programmer must be aware that register and non-register variables will behave differently in the event of a throw, and that gotos, breaks, or continues in or out of catch blocks will leave the process in an unpredictable state. Entering and leaving a protected block of code is expensive enough (about 70 μs [17]) that a highly-tuned program must have every possible catch block removed from its critical path of execution.

The interface provided by Chrysalis is too low-level for convenient use by application programmers. We have found, however, that its primitive operations constitute a very general framework upon which to build efficient higher-level communication protocols and programming environments. The following section describes a number of the software packages we have built on top of Chrysalis. Their success has depended on the fact that Chrysalis allows the user to explicitly manage processes, memory, and address spaces, and provides highly efficient low-level mechanisms for synchronization and communication.

Largely as a result of its research-environment origins, Chrysalis leaves much to be desired as a general-purpose operating system. It has no support for virtual memory or paging. It lacks a file system; file system operations are implemented over the Ethernet by a daemon process on a host machine (a VAX, Sun, or Symbolics workstation). Its user interface is built around a primitive ASCII terminal window manager (though X-window support is available when running the

network software). Interaction with the command interpreter requires intimate knowledge of the hardware and the operating system. This need for expertise means that Chrysalis is an inappropriate programming environment for all but the most sophisticated users. It is also not a development environment; programs are written, compiled, and linked on the host and downloaded for execution on the Butterfly.

Extensive use of a global referencing environment makes Chrysalis essentially a single-user system. Users can partition the machine into multiple virtual machines, but there is no support for multiple users within a partition. Moreover, protection loopholes in both the hardware and in Chrysalis allow processes (with a little effort) to inflict almost unlimited damage on each other and on the operating system. Chrysalis allows a process to map in any memory object it can name, and names are easy to guess. More fundamentally, the PNC microcode is designed in such a way that a process can enqueue and dequeue information on any dual queue it can name, regardless of any precautions the operating system might take.

The Butterfly GP-1000, now in Beta test, will run the Unix-compatible Mach operating system [1]. The availability of Mach should guarantee a convenient development environment on all of the newer machines. It is most unlikely, however, to provide users with the efficiency or the degree of control over low-level resources available with Chrysalis. BBN has announced plans to provide a hybrid approach on future machines (the RT-1000), with applications running on top of a real-time kernel in dedicated subsets of the machine, under the overall control of Mach.

2.3. Uniform System

The BBN Uniform System (US) library package [6] implements lightweight tasks that execute within a single global address space. The US interface consists of calls to create a globally-shared memory, scatter data throughout the shared memory, and create tasks that operate on the shared memory. During initialization, US creates a manager process for each processor, which is responsible for executing tasks. A task is a procedure to be applied to shared data, and is usually represented by a function name and a pointer into shared memory. A global work queue (accessed via microcode operations) is used to allocate tasks efficiently to processors. Since each task inherits the globally-shared memory upon creation, US supports a very small task granularity.

The Uniform System is the programming environment of choice for most applications, primarily because it is easy to use. All communication is based on shared memory, and the mapping of tasks to processors is accomplished automatically. Moreover, the light weight of tasks provides a very cheap form of parallelism. Nevertheless, there are significant disadvantages to using US. The work queue model of task dispatching has led to an implementation in which tasks must run to completion. Spin locks must be used for synchronization. Waiting processors accomplish no useful work, implementation-dependent deadlock becomes a serious possibility, and programs

can be highly sensitive to the amount of time spent between attempts to set a lock [55]. Spin locks also steal remote cycles, exacerbating the problem of memory contention.

US limits the amount of memory that can be shared on the original Butterfly. Like any Chrysalis process, a US manager can have at most 256 segments in its virtual address space. Since all managers have identical memory maps, only 16 Mbytes (out of a possible 1 Gbyte of physical memory) can actually be used by a computation under the Uniform System. Similarly, US limits how the data is structured. One of the main advantages of a segmented address space is that memory segments can be allocated to logical quantities regardless of their size, since each segment is of arbitrary size. This is not a reasonable approach under the Uniform System (at least on the Butterfly-1) because the number of available SARs, and hence memory segments, is severely limited. In order to be able to access large amounts of memory, each segment must be large. Data must be structured on the basis of this architectural limit, rather than logical relationships. Large amounts of data irregular in structure must be allocated in regular patterns to economize on SARs. Even on the new hardware, where SARs are not a problem, the single globally-accessible data space will tend to discourage the development of modular program structure.

Finally, the Uniform System model does not encourage the programmer to exploit locality. US creates the illusion of a global shared memory, where all data is accessed using the same mechanisms. The illusion is not supported by the hardware, however, since frequent access to individual words of remote memory is undesirable. Thus, in many applications, each task must copy data into local memory, where it is processed and then returned to the shared memory.

Our conclusion is that the Uniform System provides an outstanding environment for certain kinds of applications. It is best for programs in which (1) the available parallelism displays a high degree of regularity (as in many data-parallel symbolic and numerical applications), (2) the task-size granularity is on the order of a single subroutine call, and (3) almost all of the dependencies and interactions between tasks are statically defined. For other sorts of applications there are other useful models. Several of these are described in the following section.

3. Rochester Software Development

Butterfly software development at Rochester has always been driven by applications. Our work in computer vision, connectionist networks, and computational geometry motivated both our purchase of the Butterfly and subsequent system development. Applications programmers struggled hard at first to learn the details of the new architecture and operating system. Their effort was hampered by unreliable software, poor diagnostics, a lack of good tools, an absence of documentation, and the need to use low-level system calls for most important operations. Over time, BBN improved both software reliability and documentation, and developed the Uniform System library package, while the Rochester systems group has worked to ease the programming task by developing a large number of additional packages and tools. The Butterfly has also

formed the hardware base for implementations of two different student operating systems, and a major research effort in parallel operating systems is now underway on the Butterfly Plus.

3.1. Applications

The first significant application developed for the Butterfly at Rochester was the Connectionist Simulator [21], now in use (in its uniprocessor incarnation) at over 100 sites. The simulator supports a neural-like model of massively-parallel computing. Rochester's AI group is using it to investigate algorithms that might be used by a computer resembling the brain [22]. The Butterfly version of the simulator runs directly on top of Chrysalis. It was our first concrete example of the Butterfly's processing power. With 120 Mbytes of physical memory we were able to build networks that had led to hopeless thrashing on a VAX. With 120-way parallelism, we were able to simulate in minutes networks that had previously taken hours.

Several other early applications were drawn from work in computer vision [7,9]. The vision group at Rochester uses the University of British Colombia's IFF (Image File Format) as an internal standard. IFF includes a library of vision utilities that can be used as filters, reading an image from an input pipe and writing another to an output pipe. Complex image operations can be implemented by composing simpler filters. An early goal of the software development effort at Rochester was to extend the IFF model into the realm of parallel processing with an implementation on the Butterfly. The BIFF (Butterfly IFF) package [40], completed in the summer of 1986, contains Uniform System-based parallel versions of the standard IFF filters. A researcher at a workstation can download an image into the Butterfly, apply a complex sequence of operations, and upload the result in a tiny fraction of the time required to perform the same operations locally.

Perhaps the best-studied early application on the Butterfly was the diagonalization of matrices by Gaussian elimination. Bob Thomas of BBN conducted extensive experiments with a Uniform System-based implementation [16,55]. In an attempt to capitalize on previous experience with distributed programming, we implemented and analyzed a message-based solution to the same problem, comparing it to the Uniform System version [28,29]. The results of this comparison suggested that neither shared memory nor message passing was inherently superior, and that either might be preferred for individual applications, both from a conceptual point of view and from the point of view of maximizing performance.

In a single three-week period in the summer of 1986, seven different benchmarks were developed as part of a DARPA-sponsored investigation into parallel architectures for computer vision [8,10,11,41]. These benchmarks included edge finding and zero-crossing detection, connected component labeling, Hough transformation, geometric constructions (convex hull, Voronoi diagram, minimal spanning tree), visibility calculations, graph matching (subgraph isomorphism), and minimum-cost path in a graph. Four different programming environments were used: bare C with Chrysalis calls, the Uniform System, the Structured Message Passing package

(section 3.2), and the Lynx distributed programming language (also section 3.2). Experience with these applications and environments reinforced our conviction that different models of parallel programming can be appropriate for different applications.

Several pedagogical applications have been constructed by students for class projects, including graph transitive closure, 8-queens, and the game of pentominoes. In addition, we have running a large checkers-playing program (written in Lynx), that uses a parallel version of alpha-beta search [23]. As part of our research in debugging parallel programs (section 3.3), we have studied a non-deterministic version of the knight's tour problem and have performed extensive analysis of a Butterfly implementation of Batcher's bitonic merge sort. As part of our research in parallel file systems (section 3.4), we have developed I/O intensive algorithms for copying, transforming, merging, and sorting large external files. Ph. D. dissertations are currently in preparation in the areas of parallel compilation [25], parallelizing compilers [44], and parallel programming language design [14].

3.2. Programming Environments

NET [26] was the first systems package developed for the Butterfly at Rochester. NET facilitates the construction of regular rectangular meshes (including lines, cylinders, and tori), where each element in the mesh is connected to its neighbors by byte streams. Where Chrysalis required over 100 lines of code to create a single process, NET could create a mesh of processes, including communication connections, in half a page of code. Our experience with NET showed how valuable even a very simple systems software package could be.

Another early decision in our work with the Butterfly was that experimentation with multiple models of parallel programming would be facilitated by the availability of languages other than C. Source was available for a Modula-2 compiler developed at DEC's Western Research Center. The construction of a 68000 code generator and Butterfly run-time library provided us with our second Butterfly language [42]. In addition to addressing well-known weaknesses in C (in the areas of modularity and error-checking, for example), Modula-2 has allowed us to construct packages such as Ant Farm (see below), in which the fine-grain pseudo-parallelism of coroutines plays a central role.

Both BIFF and NET showed the value of message passing, even in a shared-memory multiprocessor. BIFF applications based on the Uniform System would copy data into and out of the shared memory using essentially a message-passing style. NET byte streams implemented untyped messages. Together with the experiments in Gaussian elimination, this early experience suggested the need to provide general-purpose support for message passing on the Butterfly. Projects were therefore launched to provide that support both at the library package level and in the form of a high-level programming language.

The SMP (Structured Message Passing) package [30, 31] was designed to provide a level of functionality comparable to that of the BBN Uniform System. It supports the dynamic

construction of *process families*, hierarchical collections of heavyweight processes that communicate through asynchronous messages (figure 2). In a generalization of the NET interconnection facility, process families can be connected together according to arbitrary static topologies. Each process can communicate with its parent, its children, and a subset of its siblings, as specified by the family topology. An SMP library is available for both C and Modula-2. For C programs it eliminates most of the cumbersome and error-prone details of interacting with Chrysalis. For Modula-2 programs it also provides a model of true parallelism with heavyweight processes and messages that nicely complements the built-in model of pseudo-parallelism with coroutines and shared memory. In order to economize on SARs, an SMP process with many communication channels must map its buffers in and out dynamically. To soften the roughly 1 ms overhead of map operations, SMP incorporates an optional *SAR cache* that delays unmap operations as long as possible, in hopes of avoiding a subsequent map.

At a higher level of abstraction, message passing is also supported by the Lynx distributed programming language [46,48]. Like SMP with Modula-2, Lynx supports a collection of heavyweight processes containing lightweight threads. Unlike SMP, it incorporates a remote procedure call model for communication between threads, relying on a message dispatcher and thread scheduler in the run-time support package to provide the performance of asynchronous message passing between heavyweight processes. Connections (links) between processes can be created, destroyed, and moved dynamically, providing the programmer with complete run-time control

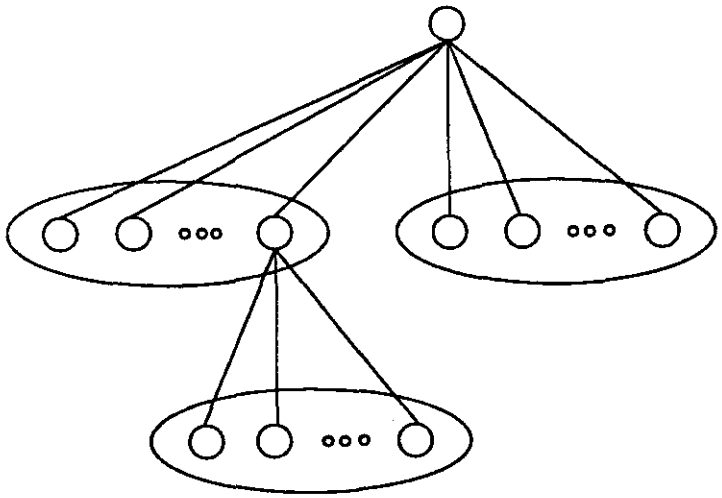

Figure 2: Hierarchy of SMP Process Families

over the communication topology (figure 3). On the Butterfly, a standard Lynx library also permits processes to share memory, though message-passing (or spin locks) must still be used for synchronization.

Because it is a language instead of a library package, Lynx offers the advantages of syntactic cleanliness, secure type checking for messages, high-level naming (with scatter/gather of message parameters), Ada-like exception handling, and automatic management of context for interleaved conversations. Unlike most parallel and distributed languages, Lynx provides these advantages without requiring compile-time knowledge of communication partners and without sacrificing protection from potential errors in those partners. Experience with Lynx has yielded important information on the inherent costs of message passing [49] and the semantics of the parallel language/operating system interface [47].

Applications experience, particularly with graph algorithms and computational geometry, has convinced us of the need for a programming environment that supports very large numbers of lightweight blockable processes. Parallel graph algorithms, for example, often call for one process per node of the graph. At the time of the DARPA benchmark, none of the programming environments available on the Butterfly supported algorithms of this type. Lightweight tasks form the core of the Uniform System, but have no facilities for blocking or synchronization other than spin locks. Lightweight threads are also available in Lynx and in Modula-2 (e.g. with SMP), but the mechanisms for interaction between threads in the same process are radically different from the mechanisms for interaction between threads in different processes. We have recently developed a library package called Ant Farm [50] that encapsulates the microcoded communication primitives of Chrysalis with a Lynx-like coroutine scheduler. Originally designed for use in Modula-2, Ant Farm is currently being modified to work with a C-based coroutine package provided in recent releases of Chrysalis. In either language, invocation of a blocking operation by a

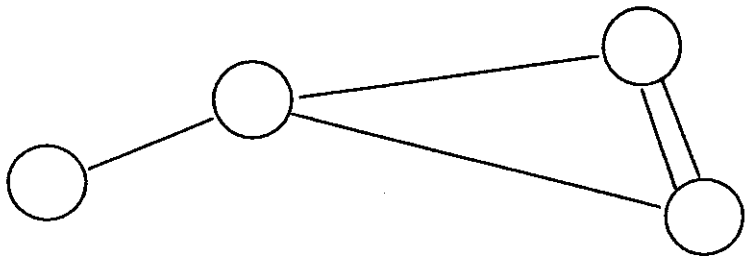

Figure 3: Processes and Links in Lynx

lightweight Ant Farm thread causes an implicit context switch to another runnable thread in the same Chrysalis process. In the event that no other thread is runnable, the coroutine scheduler blocks the process until a Chrysalis event is received. When combined with a global heap and facilities for starting remote coroutines, the resulting system allows lightweight threads to communicate with each other without regard to location.

The Uniform System, SMP, Lynx, and Ant Farm are all significantly safer and more convenient to use than the Chrysalis primitives on which they are implemented. They are also significantly less flexible, and bear little resemblance to the abstractions of the operating system. A more compatible improvement to the Chrysalis environment was provided by Chrysalis++ [12], an encapsulation of Chrysalis abstractions in C++ class definitions. To implement Chrysalis++ we first modified the standard AT&T implementation of C++ to generate code for the Butterfly, then recast the explicit object management of Chrysalis into the implicit object management of C++. Implicit object management reduces the amount of code necessary to create and manage processes, memory objects, events, dual queues, and atomic variables. The strong type checking of C++ also reduces the frequency of run-time errors. Problems encountered in the implementation of Chrysalis++ led to general observations about the difficulty of reconciling the object-management needs of languages and operating systems [13, 15].

3.3. Programming Tools

Numerous small projects undertaken in the course of our work with the Butterfly can be categorized loosely as systems software tools. Modifications to the Uniform System (e.g. for faster initialization) have been incorporated into the standard BBN release. A general-purpose package called Crowd Control allows similar tree-based techniques to be used in other programs [32], spreading work over multiple nodes. The Crowd Control package can be used to parallelize almost any function whose serial component is due to contention for read-only data. Other packages have been developed for highly-parallel concurrent data structures [19, 35] and memory allocation [20].

A local facility for software partitioning (to subdivide a Butterfly into smaller virtual machines) was brought up prior to the release of the BBN version. Local enhancements to the host-based remote file server allow us to access NFS files from Butterfly programs. The only full set of published benchmarks for PNC and Chrysalis functions is a Rochester technical report [17]. Experiments with eight different implementations of remote procedure call explored the ramifications of these benchmarks for interprocess communication [34].

Despite the improvement in the programming environment achieved by software packages and tools, the parallel program debugging cycle continued to be frustrating, particularly for nondeterministic applications. It was the realization that cyclic debugging of nondeterministic behavior was impractical, coupled with the observation that the standard approach to debugging parallel programs based on message logs would quickly fill all memory, that led to the

development of Instant Replay [33]. Instant Replay allows us to reproduce the execution behavior of parallel programs by saving the relative order of significant events as they occur, and then forcing the same relative order to occur while re-running the program for debugging. Instant Replay requires less time and space than other methods because the actual information communicated between processes is not saved. It is also general enough to use in all of our software packages because it assumes a communication model based on shared objects, which are used to implement both shared memory and message passing. No central bottlenecks are introduced by execution monitoring and there is no need for synchronized clocks or a globally-consistent logical time.

Our experiments indicate that the overhead of monitoring can be kept to within a few percent of execution time for typical programs, making it practical to run non-deterministic applications under Instant Replay all the time. We are in the process of building a toolkit based on Instant Replay that allows a full range of debugging and performance analysis tools to be integrated with a graphical user interface [24]. The graphics package, known as Moviola, makes it possible to examine the partial order of events in a parallel program at arbitrary levels of detail. It has been used to discover performance bottlenecks and message-ordering bugs, and to derive analytical predictions of running times.

3.4. Operating Systems

The Butterfly-1 has been used at Rochester as a hardware base for two different pedagogical operating systems. The Osiris kernel was an early prototype of low-level routines for the Psyche operating system (see below). It was preceded by Elmwood [36], a fully-functional RPC-based multiprocessor operating system constructed as a class project in only a semester and a half. Though it was not originally conceived as a research project, experience with Elmwood led to a considerably deeper understanding of the Butterfly architecture. It also provided us with useful experience in the management of multi-person projects, and provided some ten different graduate students with first-hand experience writing low-level software on a parallel machine.

Our accumulated experience with both applications and systems software has convinced us that no one model of process state or style of communication will prove appropriate for all applications. The beauty of large-scale NUMA machines like the Butterfly is that their hardware supports efficient implementations of a wide variety of models. Truly general-purpose parallel computing demands an operating system that supports these models as well, and that allows program fragments written under different models to coexist and interact. These observations have led to the development of a parallel operating system we call Psyche [51]. Psyche facilitates dynamic sharing between threads of control by providing a user interface based on passive data abstractions in a uniform virtual address space. It ensures that users pay for protection only when necessary by permitting the lazy evaluation of privileges, using a system of keys and access lists. The data abstractions are known as realms. Their access protocols define conventions for sharing the

uniform address space. An explicit tradeoff between protection and performance determines the degree to which those conventions are enforced. In the absence of protection boundaries, access to a shared realm (figure 4) can be as efficient as a procedure call or a pointer dereference. A Psyche implementation is currently under construction on the Butterfly Plus.

In the gray area between operating systems and programming tools, we are investigating issues in the design of highly-parallel file systems that can be used to increase the performance of I/O bound applications. From the point of view of parallel processing, any performance limit on the path between secondary storage and application program must be considered an I/O bottleneck. Faster storage devices cannot solve the I/O bottleneck problem for large multiprocessor systems if data passes through a file system on a single processor. Implementing the file system as a parallel program can significantly improve performance. Selectively revealing this parallel structure to utility programs can produce additional improvements, particularly on machines in which interprocessor communication is slow compared to aggregate I/O bandwidth. The Bridge parallel file system [18] distributes each file across multiple storage devices and processors. The approach is based on the notion of an *interleaved file*, in which consecutive logical blocks are assigned to different physical nodes. Naive programs are able to access files just as they would with a conventional file system, while more sophisticated programs may export pieces of their code to the processors managing the data, for optimum performance. Analytical and experimental studies indicate that Bridge will provide linear speedup on several dozen disks for a wide variety of file-based operations, including copying, sorting, searching, and comparing.

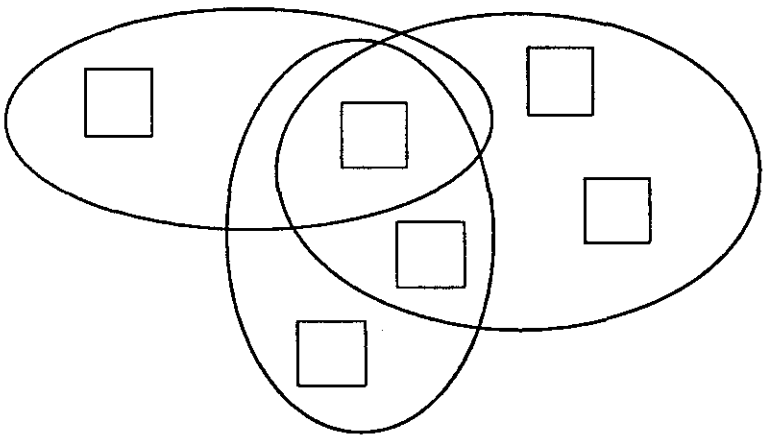

Figure 4: Overlapping Protection Domains in Psyche

4. Lessons

The following summarizes the lessons we have learned in developing both system software and applications for a large-scale multiprocessor over a three year period. Our work has emphasized architectural implications and programming environment issues; our lessons reflect this emphasis. Although our particular experience is with the Butterfly-1, we believe these lessons generalize to other multiprocessors as well.

4.1. Architectural Implications

Large-scale shared-memory multiprocessors are practical. We have achieved significant speedups (often almost linear) using over 100 processors on a range of applications including connectionist network simulation, game-playing, Gaussian elimination, parallel data structure management, and numerous computer vision and graph algorithms. In the course of developing these applications, we have also discovered that many interesting effects become obvious only when large numbers of processors are in use. In the Gaussian elimination experiments, for example, our SMP implementation outperformed the Uniform System implementation whenever fewer than 64 processors were used, despite the fact that communication in SMP is significantly more expensive than direct access to shared memory. Beyond 64 processors the timings for the Uniform System remained constant (no additional improvements), while the SMP timings actually *increased* (figure 5). This anomaly is due to the amount of communication used in each implementation. The number of messages sent in the SMP implementation is P*N, where P is the number of processors and N is the size of the matrix. In other words, doubling the amount of parallelism also doubles the amount of communication. Beyond 64 processors, the increased amount of communication caused by each additional processor is not justified by the incremental gain in parallelism. The number of communication operations in the Uniform System implementation is $(N^2\text{-}N)+P(N\text{-}1)$; doubling the amount of parallelism does not significantly increase the amount of communication. The point at which the increase in communication dominates additional parallelism in the Uniform System implementation is not even visible with 128 processors. Without a large number of processors, we might not have discovered the anomaly.

Locality of reference is important, even with shared memory. Although each processor can access the memory of others, remote references on the Butterfly-1 are five times slower than local references. This disparity is not so great as that found in local-area networks, where two or three orders of magnitude are common, but it cannot be ignored without paying a substantial performance penalty. Any measurable difference between local and remote access time requires the programmer to treat the two differently; caching of frequently accessed data is essential. A standard technique used in Uniform System programs is to copy blocks of data from the (logically) global shared memory into local memory for processing; results are then copied back to the global shared memory. In the Hough transform application, this technique improved performance by 42% when 64 processors were used [41]. Local lookup tables for transcendental functions

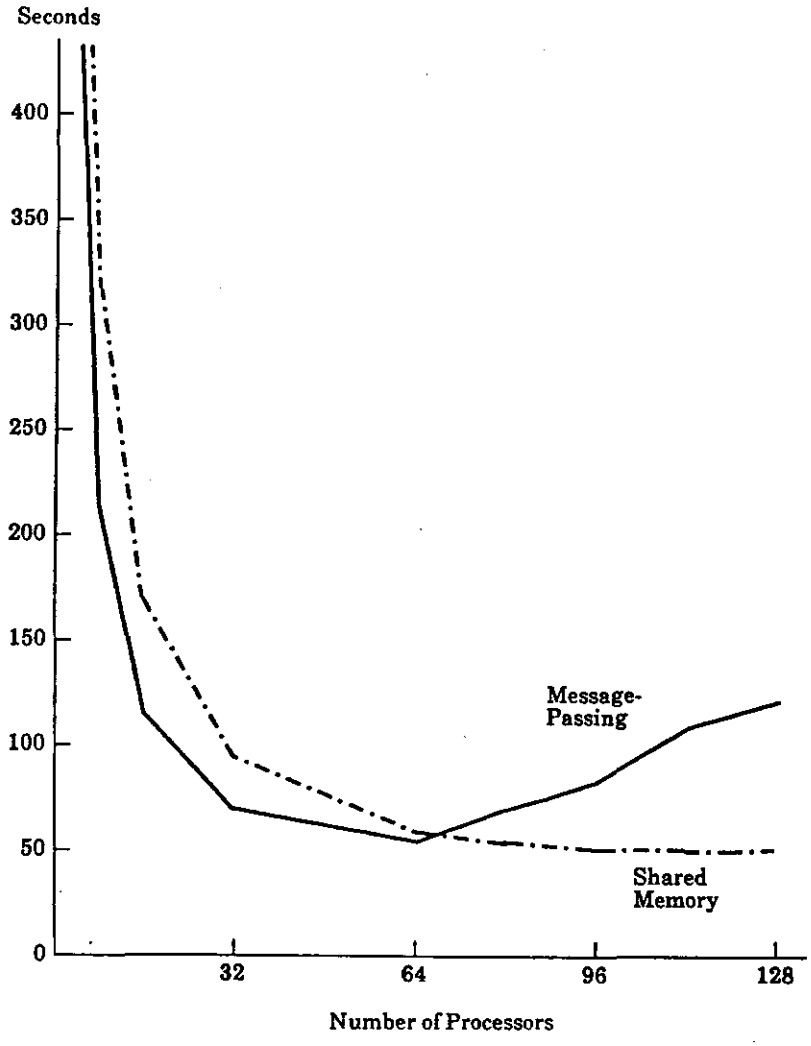

Figure 5: Gaussian Elimination Performance; Shared Memory versus Message Passing

improved performance by an additional 22%. The issue of locality will be even more important in the Butterfly Plus, since local references have improved by a factor of four, while remote references have improved by only a factor of two.

Contention has the potential to seriously impact performance. Remote references on the Butterfly can encounter both memory and switch contention. The potential for switch contention was clearly anticipated in the design of the Butterfly hardware, and has been rendered almost negligible [45]. On the other hand, the potential for memory contention appears to have been underestimated, since remote references steal memory cycles from the processor containing the memory. Only one processor can issue local references to a given memory, but over a hundred

processors can issue simultaneous remote references, leading to performance degradation far beyond the nominal factor of five delay. The careful programmer must organize data not only to maximize locality, but also to minimize memory contention. For example, the Gaussian elimination program (on 64 processors or fewer) displays a performance improvement of over 30% when data is spread over all 128 memories [29]. The greatest effect occurs when roughly 1/4 to 1/2 of the total number of processors are in use. When a larger fraction of processors are performing computation, most of the memory is already in use. Not enough is left to reduce contention noticeably. When too few processors are used, the resulting memory traffic is not heavy enough to cause significant contention.

Amdahl's law is extremely important in large-scale multiprocessors. Serial program components that have little impact on performance when a few processors are in use can lead to serious bottlenecks when 100 processors are in use. Massive problem sizes are sometimes required to justify the high costs of serial startup. Serialization in system software is especially difficult to discover and avoid. For example, the Crowd Control package was created to parallelize process creation, but serial access to system resources (such as process templates in Chrysalis) ultimately limits our ability to exploit large-scale parallelism during process creation. Serial memory allocation in the Uniform System was a dominant factor in many programs until a parallel memory allocator was introduced into the implementation [20]. Serial access to a large file is especially unacceptable when 100 processes are available to process the data; the Bridge file system is designed to address this particular bottleneck. None of these parallel solutions is particularly simple, and the elimination of similar bottlenecks can be expected to pose a serious problem for any highly parallel application.

Architectural variety inhibits the development of portable systems software. A myriad of different multiprocessor architectures are now commercially available, including bus-based multiprocessors like the Sequent Balance and Encore Multimax, switch-based multiprocessors like the BBN Butterfly, cosmic cube variants like the NCUBE and Intel hypercube, and the Connection Machine from Thinking Machines, Inc. Despite the architectural variety, few general principles of parallel programming have emerged on which programming environments could be based. Some notable attempts have been made to provide general parallel programming environments [43, 53, 54], but substantial investments in software development are still required for every new machine. In many cases it may even be difficult to develop a production-quality operating system fast enough to make truly effective use of a machine before it becomes obsolete. The problem is less severe in the sequential computer world, since uniprocessors tend to resemble one another more than multiprocessors do. While an operating system such as Unix can make effective use of a variety of conventional sequential computers, simply porting Unix to a multiprocessor would not provide fine-grain parallelism, cope effectively with non-uniform memory access times (the so-called "NUMA problem"), or address a host of other issues. The emergence of Mach may improve matters significantly, but its effectiveness for NUMA architectures has yet

to be demonstrated.

4.2. Programming Environment

The programming environment must support multiple programming models. We have implemented many different applications using an assortment of operating systems, library packages, and languages. Empirical measurements demonstrate that NUMA machines like the Butterfly can support many different programming models efficiently. For example, efficient communication based on shared memory has been implemented in the Uniform System and Ant Farm. Higher-level communication based on message passing and remote procedure call has been implemented in SMP, Lynx, and Elmwood. Extensive analysis of the communication costs in these systems suggests that, for the semantics provided, the costs are very reasonable [36,47,49]. A comparison with the costs of the basic primitives provided by Chrysalis shows that any *general* scheme for communication on the Butterfly will have comparable costs.

Even though each model can be implemented efficiently on the Butterfly, no single model can provide optimal performance for all applications. Moreover, subjective experience indicates that conceptual clarity and ease of programming are maximized by different models for different kinds of applications. In the course of the DARPA benchmark experiments, seven different problems were implemented using four different programming models. One of the basic conclusions of the study was that none of the models then available was appropriate for certain graph problems; this experience led to the development of Ant Farm. Some large applications may even require different programming models for different components; therefore it is also important that mechanisms be in place for communication across programming models. These concerns form the motivation behind the Psyche operating system.

It is difficult to exercise low-level control over parallelism without accepting explicit control of other resources as well. Programmers use a multiprocessor for performance gains, and therefore must maximize the (true) parallelism in an application program. Since it is impossible to anticipate the needs of every application, a parallel programming environment will usually provide low-level mechanisms for mapping processes to processors. Unfortunately, in allowing the programmer to control parallelism (and the corresponding processes), the environment will often force the programmer to manage other resources as well. For example, the programmer may be required to manage address spaces explicitly in order to co-locate a process and its data. All of the parallel programming environments on the Butterfly couple the ability (or inability) to manage parallelism with the ability (or inability) to manage memory. Chrysalis allows the programmer to create a process on any Butterfly node, but it also requires the programmer to manage shared memory explicitly. Even very simple sharing requires several system calls, each with several parameters. The Uniform System attempts to make processor boundaries transparent; each task may execute on any available processor. There is no attempt, however, to co-locate a task and the data it manipulates. To achieve acceptable performance, the programmer must cache

data explicitly. SMP does not require the user to manage the address space of a process explicitly; however, it allocates processes to processors using a fixed allocation algorithm, which can lead to an imbalance in processor load. A better balance between flexibility and ease of use must be found.

An efficient implementation of a shared name space is valuable even in the absence of uniform access time. The primary advantage of shared memory is that it provides the programmer with a familiar computational model. Programmers do not have to deal with multiple address spaces; programs can pass pointers and data structures containing pointers without explicit translation. The attractiveness of a single address space cannot be overstated; it is the primary reason why most programmers choose to use the Uniform System as their programming environment. Even when non-uniform access times warp the single address space model by forcing the programmer to deal explicitly with local caching of data, shared memory continues to provide a form of global name space that appeals to programmers. Data items, including pointers, can be copied from one local memory to another through the global name space. In effect, the shared memory is used to implement an efficient Linda tuple space [2]. The Linda **in**, **read**, and **out** operations correspond roughly to the operations used to cache data in the Uniform System.

Better monitoring and debugging tools are essential. The lack of such tools contributes dramatically to program development time, and is probably the most frequently cited cause of frustration with parallel programming environments. Performance is paramount in multiprocessors, yet few general tools exist for measuring performance. Bottlenecks such as memory or switch contention are difficult to discover and must usually be measured indirectly. Single process debuggers cannot capture parallel behavior, and performance monitoring and debugging tools for distributed systems [27, 37, 38] are not particularly well-suited to multiprocessors. The problem is especially acute in NUMA machines, since they lack a shared communication medium that could facilitate monitoring.

Significant progress has been made recently in monitoring and debugging tools for shared-memory multiprocessors [24, 52]. In particular, we have begun construction of an extensible, integrated toolkit for parallel program debugging and performance analysis, as mentioned in section 3.3 [24]. Ultimately, the toolkit will include an interactive debugger, a graphical execution browser, performance analysis packages, and a programmable interface for user queries. We hide the complexity of how an algorithm is implemented by emphasizing a graphical representation of execution. (Figure 6, produced by the toolkit, is a graphical view of deadlock in an odd-even merge sort program.) Top-down analysis at all levels of abstraction is possible because the graphical representation is integrated with access to the low-level details of an execution. The analysis process converges because all executions are repeatable. The toolkit is programmable, hence extensible. It allows programmers to analyze the behavior of parallel programs interactively, much as interactive debuggers and profilers are used to analyze the behavior of sequential programs. Our experience to date confirms the utility of the toolkit; the debugging and analysis

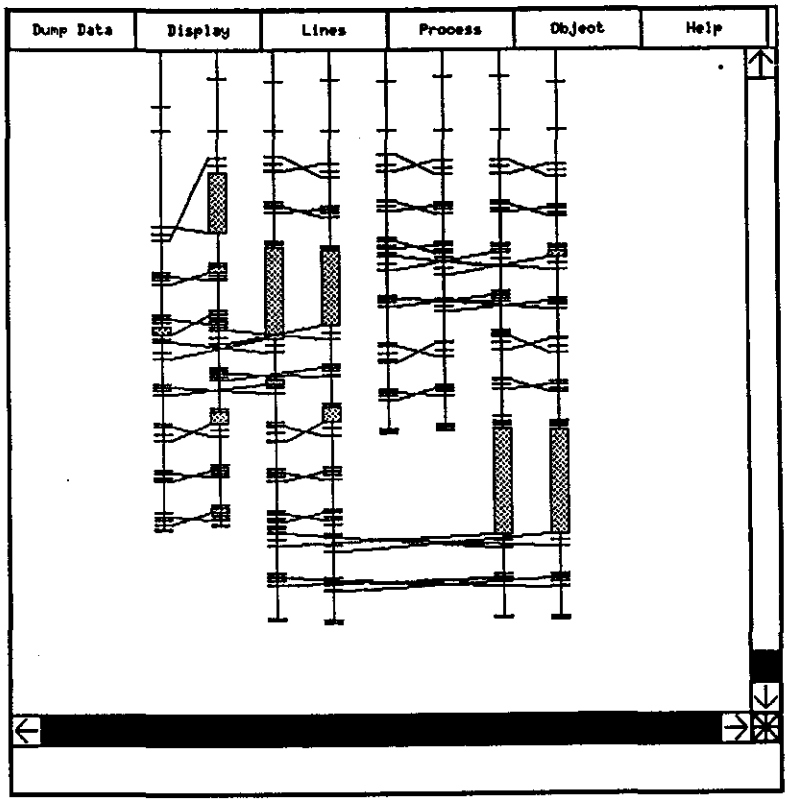

Figure 6: Graphical View of Odd-Even Merge Sort

cycle has decreased from several days to a few hours.

Programming environments are often more important than processing speed. Many application programmers in our department who could exploit the parallelism offered by the Butterfly continue to use Sun workstations and VAXen. These programmers have weighed the potential speedup of the Butterfly against the programming environment of their workstation and found the Butterfly wanting. New processors, switching networks, or memory organizations will not change this fact, although the introduction of Mach on the Butterfly is clearly a step in the right direction. The most important task ahead for the parallel programming community is not the development of newer and bigger multiprocessors, but rather the development of programming environments comparable to those available on sequential computers.

5. Conclusions

The existence of a large-scale multiprocessor at the University of Rochester has dramatically affected how we think about parallel programming. Special-purpose techniques do not tend to extrapolate well to 120 processors; we have learned to avoid taking advantage of a specific number of processors.

We are generally satisfied with the Butterfly. We have had access to all of the system details necessary to implement system software; we have invested the effort to become experts. However, despite the level of local expertise, to this day only intrepid programmers use the Butterfly to solve real problems. It remains to be seen whether the newer Mach-based Butterfly software will change this situation appreciably.

Butterfly-family machines remain the largest shared-memory multiprocessors commercially available. They are vastly more flexible than the competing message-based multicomputers (e.g. hypercubes), and are not subject to the bandwidth limitations of bus-based shared-memory machines. The problems presented by the architecture, especially the NUMA problem, will be with us for some time, and solutions will be required in any future large-scale parallel machine. Perhaps most important from our point of view, parallel processors have helped bring applications programmers and system developers together in a spirit of cooperation. This cooperation will be crucial to the development of the parallel programming environments of the future.

Acknowledgments

The authors would like to express their thanks to the research and support staff of BBN Laboratories and BBN Advanced Computers Incorporated, and to the many students, staff, and faculty members whose willingness to immerse themselves in an experimental and often frustrating environment has made this research possible. Special thanks are due to Liud Bukys, our tireless lab manager and all-around Butterfly guru.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986, pp. 93-112.

[2] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *Computer 19*:8 (August 1986), pp. 26-34.

[3] BBN Advanced Computers Incorporated, "Inside the Butterfly Plus," Cambridge, MA, 16 October 1987.

[4] BBN Advanced Computers Incorporated, "Chrysalis® Programmers Manual, Version 3.0," Cambridge, MA, 28 April 1987.

[5] BBN Laboratories, "Butterfly® Parallel Processor Overview," BBN Report #6148, Version 1, Cambridge, MA, 6 March 1986.

[6] BBN Laboratories, "The Uniform System Approach to Programming the Butterfly® Parallel Processor," BBN Report #6149, Version 2, Cambridge, MA, 16 June 1986.

[7] C. M. Brown, "Parallel Vision on the Butterfly Computer," *Supercomputer Design: Hardware and Software*, Volume 3 of the *Proceedings of the Third International Conference on Supercomputing*, May 1988, pp. 54-68.

[8] C. M. Brown, R. J. Fowler, T. J. LeBlanc, M. L. Scott, M. Srinivas, and others, "DARPA Parallel Architecture Benchmark Study," BPR 13, Computer Science Department, University of Rochester, October 1986.

[9] C. M. Brown, T. Olson, and L. Bukys, "Low-level Image Analysis on a MIMD Architecture," *Proceedings of the First IEEE International Conference on Computer Vision*, June 1987, pp. 468-475.

[10] L. Bukys, "Connected Component Labeling and Border Following on the BBN Butterfly Parallel Processor," BPR 11, Computer Science Department, University of Rochester, October 1986.

[11] J. Costanzo, L. Crowl, L. Sanchis, and M. Srinivas, "Subgraph Isomorphism on the BBN Butterfly Multiprocessor," BPR 14, Computer Science Department, University of Rochester, October 1986.

[12] L. Crowl, "Chrysalis++," BPR 15, Computer Science Department, University of Rochester, December 1986.

[13] L. A. Crowl, "An Interface Between Object-Oriented Systems," TR 211, Department of Computer Science, University of Rochester, Apr 1987.

[14] L. A. Crowl, "A Model for Parallel Programming," pp. 71-84 in *Proceedings of the 1988 Open House*, ed. C. A. Quiroz, TR 209, Department of Computer Science, University of Rochester, May 1988.

[15] L. A. Crowl, "Shared Memory Multiprocessors and Sequential Programming Languages: A Case Study," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, Jan 1988.

[16] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar, "Performance Measurements on a 128-Node Butterfly Parallel Processor," *Proceedings of the 1985 International Conference on Parallel Processing*, 20-23 August 1985, pp. 531-540.

[17] P. Dibble, "Benchmark Results for Chrysalis Functions," BPR 18, Computer Science Department, University of Rochester, December 1986.

[18] P. C. Dibble, M. L. Scott, and C. S. Ellis, "Bridge: A High-Performance File System for Parallel Processors," *Proceedings of the Eighth International Conference on Distributed Computing Systems*, 13-17 June 1988, pp. 154-161.

[19] C. S. Ellis, "Extendible Hashing for Concurrent Operations and Distributed Data," TR 110, Computer Science Department, University of Rochester, October 1982.

[20] C. S. Ellis and T. J. Olson, "Parallel First Fit Memory Allocation," *Proceedings of the 1987 International Conference on Parallel Processing*, 17-21 August 1987, pp. 502-511.

[21] M. Fanty, "A Connectionist Simulator for the BBN Butterfly Multiprocessor," TR 164, BPR 2, Computer Science Department, University of Rochester, January 1986.

[22] J. A. Feldman, M. A. Fanty, N. H. Goddard, and K. J. Lynne, "Computing with Structured Connectionist Networks," *CACM 31*:2 (February 1988), pp. 170-187.

[23] J. P. Fishburn and R. A. Finkel, "Parallel Alpha-Beta Search on Arachne," Computer Sciences Technical Report #394, University of Wisconsin – Madison, July 1980.

[24] R. J. Fowler, T. J. LeBlanc, and J. M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors," *Proceedings, ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[25] N. M. Gafter, "Algorithms and Data Structures for Parallel Incremental Parsing," *Proceedings of the 1987 International Conference on Parallel Processing*, 17-21 August 1987, pp. 577-584.

[26] E. Hinkelman, "NET: A Utility for Building Regular Process Networks on the BBN Butterfly Parallel Processor," BPR 5, Computer Science Department, University of Rochester, February 1986.

[27] J. Joyce, G. Lomow, K. Slind, and B. Unger, "Monitoring Distributed Systems," *ACM TOCS 5*:2 (May 1987), pp. 121-150.

[28] T. J. LeBlanc, "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study," *Proceedings of the 1986 International Conference on Parallel Processing*, 19-22 August 1986, pp. 463-466. Expanded version available as BPR 3, Computer Science Department, University of Rochester, January 1986.

[29] T. J. LeBlanc, "Problem Decomposition and Communication Tradeoffs in a Shared-Memory Multiprocessor," in *Numerical Algorithms for Modern Parallel Computer Architectures*, IMA Volumes in Mathematics and its Applications #16, Springer-Verlag, 1988.

[30] T. J. LeBlanc, "Structured Message Passing on a Shared-Memory Multiprocessor," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988, pp. 188-194.

[31] T. J. LeBlanc, N. M. Gafter, and T. Ohkami, "SMP: A Message-Based Programming Environment for the BBN Butterfly," BPR 8, Computer Science Department, University of Rochester, July 1986.

[32] T. J. LeBlanc and S. Jain, "Crowd Control: Coordinating Processes in Parallel," *Proceedings of the 1987 International Conference on Parallel Processing*, 17-21 August 1987, pp. 81-84.

[33] T.J. LeBlanc and J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers C-36*:4 (April 1987), pp. 471-482.

[34] J. Low, "Experiments with Remote Procedure Call on the Butterfly," BPR 16, Computer Science Department, University of Rochester, December 1986.

[35] J. M. Mellor-Crummey, "Concurrent Queues: Practical Fetch-and-Phi Algorithms," TR 229, Computer Science Department, University of Rochester, Nov 1987.

[36] J. M. Mellor-Crummey, T. J. LeBlanc, L. A. Crowl, N. M. Gafter, and P. C. Dibble, "Elmwood — An Object-Oriented Multiprocessor Operating System," *Software — Practice and Experience*, to appear. Also published in the *University of Rochester 1987-88 Computer Science and Computer Engineering Research Review*, and available as BPR 20.

[37] B. P. Miller, C. Macrander, and S. Sechrest, "A Distributed Programs Monitor for Berkeley Unix," *Software — Practice and Experience 16*:2 (February 1986), pp. 183-200.

[38] B. P. Miller and C.-Q. Yang, "IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs," *Proceedings of the Seventh International Conference on Distributed Computing Systems*, 21-25 September 1987, pp. 482-489.

[39] D. Moon, *MacLISP Reference Manual, Revision 0*, Project MAC, Laboratory for Computer Science, MIT, Cambridge, MA, April 1974.

[40] T. J. Olson, "An Image Processing Package for the BBN Butterfly Parallel Processor," BPR 9, Computer Science Department, University of Rochester, September 1986.

[41] T. J. Olson, "Finding Lines with the Hough Transform on the BBN Butterfly Parallel Processor," BPR 10, Computer Science Department, University of Rochester, September 1986.

[42] T. J. Olson, "Modula-2 on the BBN Butterfly Parallel Processor," BPR 4, Computer Science Department, University of Rochester, January 1986.

[43] T. Pratt, "PISCES: An Environment for Parallel Scientific Computation," *IEEE Software*, July 1985, pp. 7-20.

[44] C. A. Quiroz, "Compilation for MIMD Architectures," Thesis Proposal, Department of Computer Science, University of Rochester, May 1986.

[45] R. Rettberg and R. Thomas, "Contention is No Obstacle to Shared-Memory Multiprocessing," *CACM 29*:12 (December 1986), pp. 1202-1212.

[46] M. L. Scott, "LYNX Reference Manual," BPR 7, Computer Science Department, University of Rochester, August 1986 (revised).

[47] M. L. Scott, "The Interface Between Distributed Operating System and High-Level Programming Language," *Proceedings of the 1986 International Conference on Parallel Processing*, 19-22 August 1986, pp. 242-249.

[48] M. L. Scott, "Language Support for Loosely-Coupled Distributed Programs," *IEEE Transactions on Software Engineering SE-13*:1 (January 1987), pp. 88-103.

[49] M. L. Scott and A. L. Cox, "An Empirical Study of Message-Passing Overhead," *Proceedings of the Seventh International Conference on Distributed Computing Systems*, 21-25 September 1987, pp. 536-543.

[50] M. L. Scott and K. R. Jones, "Ant Farm: A Lightweight Process Programming Environment," BPR 21, Computer Science Department, University of Rochester, August 1988.

[51] M. L. Scott, T. J. LeBlanc, and B. D. Marsh, "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System," *Proceedings of the 1988 International Conference on Parallel Processing*, 15-19 August 1988, pp. 255-262.

[52] Z. Segall and L. Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software* 2:6 (November 1985), pp. 22-37.

[53] L. Snyder and D. Socha, "Poker on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environment," *Proceedings of the 1986 International Conference on Parallel Processing*, 19-22 August 1986, pp. 628-635.

[54] W. K. Su, R. Faucette, and C. Seitz, "C Programmer's Guide to the Cosmic Cube," TR 5203:85, Computer Science Department, California Institute of Technology, Sept 1985.

[55] R. Thomas, "Using the Butterfly to Solve Simultaneous Linear Equations," Butterfly Working Group Note 4, BBN Laboratories, March 1985.