Final Report on Research in Parallel Computing: Network-based Multicomputers 1990 - 1995

6 December 1995

Carnegie Mellon University
Computer Science Department

Abstract

No existing parallel architecture (including SIMD machines, MIMD machines, vector processors, machines with a common, globally-addressable memory, machines with private memories) has demonstrated its superiority for all possible applications. That is, given different applications, different parallel machines may deliver the highest absolute performance or the best price-performance.

A good solution to this problem is to link a number of different systems (ranging from workstations to parallel supercomputers) with a high-speed network to build a single, heterogeneous system. This report details our research to address this issue during the period 1990-95.

This research was sponsored by the Advanced Research Projects Agency, DoD, through ARPA Order 7330, and monitored by the Space and Naval Warfare Systems Command under contract MDA972-90-C-0035. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency or the United States Government.

Introduction

At the current time, there exist a number of different parallel architectures, such as SIMD machines, MIMD machines, vector processors, machines with a common (globally addressable) memory, machines with private memories, etc. None of these architectures has demonstrated that it is superior to all other other architectures for all possible applications. That is, given different applications, different parallel machines may deliver the highest absolute performance or the best price-performance. There is no reason to expect that in the near term a single architecture will become prevalent. In the long run, we can expect the architectures to converge as the designers begin to better understand the factors that contribute to an architecture's performance. But until such a uniform architecture exists, different applications will favor different machines.

A good solution to this problem is to link a number of different systems (ranging from workstations to parallel supercomputers) with a high-speed network to build a single, heterogeneous system. In prior research we had developed an architecture for such a heterogeneous system, as well as a parallel supercomputer (iWarp, in collaboration with Intel). In this contract period we undertook to link iWarp (and other supercomputers) via the Nectar system.

1 iWarp issues

According to the original schedule Intel was to tape out (fabricate) the first set of iWarp components in mid-1989 and to deliver the three demonstration systems to Carnegie Mellon in late 1989. Then, in the 1990, Intel was to mature the component (optimize for fabrication, remedy any problems encountered by testing, etc.) and finish the iWarp development task in 1990.

Unfortunately, work did not proceed according to plan. The first component was indeed fabricated in 1989, but the first batch of parts did not work. Due to a design error the two non-overlapping clocks overlapped and rendered this batch of chips useless. It was possible to fix this design flaw by changing the metal mask only (a saving of several months over complete redesign involving all mask levels). However, testing revealed that a major redesign was necessary to operate the components at the target speed of 20 MHz. It took Intel two more attempts to get the design to operate at speed, and even in the third iteration (C step), the component exhibits some bugs. Carnegie Mellon has helped Intel identify these problems and find solutions, not only to make the iWarp system work, but also to learn from the experiences and influence the next generation of systems. In the next two sections we describe briefly our results in the two major areas: the development of an optimizing compiler for iWarp and the design of a runtime system.

1.1 Compiler issues

The design of the earlier Warp machine made it possible to execute code generated by a high-level language compiler, but the high-level language that could be translated was severely limited. For iWarp, we did not want to work with a custom or specialized language. To make a system like iWarp acceptable to a larger user community, compilers for industry-standard languages must be provided. For this reason we decided to support compilers for **C** and Fortran. Both compilers have been implemented by Intel Corp.

The iWarp **C** compiler is the first production-quality **C** compiler that supports software pipelining. Our earlier research had shown that software pipelining produced good code when compiling W2 (a Pascal-like language) programs; our research now demonstrates that this code generation technique can be incorporated into an industrial-strength **C** compiler. Software pipelining produces a schedule so that the executions of multiple loop iterations are overlapped. Operations from multiple loop iterations are either placed into the same instruction word or are interleaved, depending on the type of instruction-level parallelism provided in the target architecture. In contrast to other scheduling techniques that rely on loop unrolling to determine the number of overlapping iterations, the degree of overlapping in software pipelining is determined by the exact resource requirements of the loop operations. Thus, accurate modelling of machine resources is especially crucial for software pipelining. Modelling the target machine is easy for idealized machines, but for a real processor, a number of challenges arise. For example:

- If there are several ways to perform a specific operation (e.g., there are multiple functional units that all implement a common set of operations, but each unit also implements some operations that cannot be performed by another unit), a specific functional unit must be assigned to an operation. A phase ordering problem exists between assigning functional units to operations and scheduling the operations.
- If the instruction set uses implicit registers, such registers impose resource constraints. A value from an implicit register must be used as soon as possible to free up the resource.
- Destructive read instructions (e.g., reading the top of a queue has the effect of dequeuing the value) impose constraints on the scheduler. If a destructively read value is used multiple times, care has to be taken that all such users obtain indeed the same value.
- If the target machine includes different instruction formats with different amounts of parallelism, there may be additional constraints on which operations can be scheduled together.

Existing code generators have often dealt with these problems by employing heuristics or by using ad-hoc methods. We found systematic methods based on abstractions and transformations that simplify the modelling of the target processor and allow efficient resource utilization during code scheduling [Adl-Tabatabai and Gross 93a].

Optimizing compilers produce dense and efficient code, but it is also known that a program that has been compiled with an optimizing compiler is hard for a symbolic debugger to understand. Most compilers allow a user to choose between optimization and debugging. We have implemented a symbolic debugger for global register allocation—an important optimization employed by the iWarp compiler. Global register allocation and assignment effectively exploit modern architectures that have large register files (iWarp, for example, has 128 registers, 96 of which can be allocated by the compiler). These optimizations are incorporated in almost all modern optimizing compilers.

Such optimizations, however, affect debugging by making variables inaccessible at a breakpoint. By attempting to pack as many variables as possible into a limited number of registers, global register allocation reassigns registers to different variables at different points in the program. Therefore, a source level variable *V* may be inaccessible at a breakpoint if the register assigned to *V* holds the value of some other variable at that point, and there is no other location that holds *V*'s value.

We investigated the problem of detecting these nonresident variables in the presence of global register allocation. Our solution is based on using data flow analysis techniques *for the debugger* to detect all points where source level values are in their assigned run-time locations. The problem of detecting resident variables is concerned only with whether a variable *V*'s register contains any source value of *V*, and not whether *V* is the value the user expects (based on the flow of the program) [Adl-Tabatabai and Gross 92] [Adl-Tabatabai and Gross 93b]. The iWarp symbolic debugger does not solve all the problems that a debugger for an optimizing compiler has to address, but it is a major step forward in this direction.

1.2 System issues

The key abstraction of the iWarp communication system is the *pathway*. A pathway is a direct connection from a cell (called the source cell) to another cell (called the destination cell). Each segment of the pathway connects the communication agent of a cell to the computation agent of the same cell or to the communication agent of another cell. Using the pathway abstraction, we can establish a "direct" connection between two cells in an array even though the two cells are not physically connected.

In the most general case, all pathways are established dynamically at runtime [Borkar, et al. 90a]. Although dynamic creation is possible, it is not always the most appropriate action. For example, since congestion in the nodes can delay establishing a pathway for an arbitrary amount of time, the nodes that want to establish a pathway have to adopt an appropriate protocol, which adds overheads. In the case of some usage models (those that reserve a set of resources exclusively for a single application) the nodes that are part of the pathway are known a priori. This information, together with knowledge about the fixed topology of the torus, can be translated at load time (or earlier, e.g. link time) into a map that is down-loaded. Then, as part of the startup operations, each node interprets this map and sets up the required pathways with minimal overheads.

1.2.1 Monitoring

A program that sends data via a pathway to another node does not have to know the route of the pathway. That is, for the sending program it does not make a difference if the destination node is adjacent (physical neighbor) or separated by a number of intermediate nodes. This feature of pathways allows us to build a novel debugging tool, since the intermediate nodes (on a pathway) can be used to gather debugging and performance information.

To understand or improve the execution behavior of a program on a parallel system, it is often necessary to monitor this interprocessor communication. If we can observe how messages travel through the parallel system, we can combine this information with information about the execution on each node to provide the user with a global picture of system performance. Consider the example where processor A must receive messages from both processors B and C before it can begin some computation. Suppose the message from C will arrive long after the message from B has arrived. If the compiled program on processor A has committed to receive C's message first, processor A will block, waiting to receive the message from C, and processor B will block, waiting for its message to be received. A single-node debugger is of little help in identifying such situations, since it cannot capture the reason for the delay in processor B by solely inspecting this processor's state. Furthermore, a single-node debugger may perturb the execution of the inspected process. Some pertinent communication issues to measure are frequency of messages, the number of words transmitted, and specific patterns in the messages.

A parallel system like iWarp offers a novel solution. Each iWarp processor contains a computation and communication engine, which are tightly coupled, providing the computation engine with a detailed view of the communication system. At the same time, the computation engine can be programmed like a general-purpose processor, providing the opportunity to use such a processor as a monitor processor. The absence of a global memory makes all communication explicit at the hardware level, so if we monitor the communication between processors, we can obtain a complete picture of the communication in the system. For the iWarp array, we have created a hybrid performance monitoring system that has many of the benefits of hardware monitoring while incurring costs close to those associated with software instrumentation [Gross and Hinrichs 92].

1.2.2 Synchronization

Barrier synchronization is a useful technique for organizing the execution of a parallel program into a sequence of loosely coordinated phases. For example, a phase of a data-parallel program running on a private memory system might consist of a communication step, where processors exchange messages among themselves, followed by a computation step, where the processors operate independently on their local data. In this example, a barrier synchronization between data-parallel phases prevents intermixing of messages from different phases.

We call a barrier that involves a group of processors a *subset barrier*, and we say a processor *participates* in a barrier synchronization if it has to wait until all other processors in the subset have reached the barrier. In practice, when programs are generated by parallel program generators (see below), an arbitrary subset of processors can participate in a barrier. Further, disjoint subsets of processors can participate in different barriers in parallel. This flexibility is an important property that a realistic barrier synchronization package must support so that a parallel program generator can allocate disjoint subsets of processors to different data-parallel tasks, and then synchronize each of these tasks independently and in parallel.

Since the mapping of processes to processors may not be known at compile time, iWarp barriers cannot rely on the identifiers (IDs) of each processor to identify different subset barriers. We implemented a flexible barrier synchronization package that allows the synchronization of sets of processors [Feldmann, et al. 92]. This package embodies two general communication models that capture important capabilities of modern private-memory systems: the *bounded buffer broadcast model*, based on a global broadcast network, and the *anonymous destination message passing model*, based on a shared, general-purpose communication network. These models are simple enough to guide the design of algorithms for subset barrier synchronization, and also provide, based on micromeasurements of a few communication operations, an accurate cost model.

Subset barrier synchronization does not have to be expensive. We found an algorithm for a (logical) ring of processors with a time complexity of $O(\sqrt{N})$, rather than the $O(N)$ running time one would initially expect. Furthermore, the algorithms presented here are reasonable: Only those processors that participate in a barrier incur any synchronization overhead. Processors that do not participate incur no overhead. The implementation on iWarp preserves this important property of the algorithm by using the appropriate hardware structures of the iWarp processors in a novel way. Even though arbitrary subsets of processors are allowed to synchronize, with only the name and number of processors involved in each barrier known at compile time, this implementation does not impose any overhead on processors that do not participate. This feature makes this design attractive for systems with large numbers of processors, since we cannot allow a small subset of synchronizing processors to impact the rest of the processors.

Efficient barrier synchronization is typically associated with dedicated global hardware resources, such as a broadcast bus. However, the ring-based barriers, which use a general-purpose communication system, consistently outperform the broadcast-based barriers, which used a global broadcast network. This leads us to the conclusion that efficient and flexible barrier synchronization can be supported by a general-purpose communication system, provided sufficient care is taken to ensure fast message transfers between processors and fast direct access to the communication system on each processor.

1.2.3 Novel usage models

General-purpose computers often cannot meet the requirements of the most demanding applications, such as radar and acoustic signal processing and high-performance image processing. These applications have high computing requirements, typically on the order of several gigaflops (billion floating-point operations per second). In addition systems for for these application are usually required to have low energy consumption and be compact in their physical sizes. Only *application-specific computing systems*, whose designs are tailored to the application, can meet these requirements. These systems can typically make use of large-scale parallelism with hundreds or thousands of powerful processors.

Prior to the design of iWarp only custom processors provided the computational power to support application-specific computing. However, the more specialized the hardware, the more difficult it is to adapt to algorithm improvements, future upgrades, wider operational dynamics, and changing system requirements. Fielded systems can be obsolete by the time they reach operational status. To overcome the inflexibility of the hardware, most systems allow some amount of programming to make possible future adjustments to changing requirements, as well as corrections to the original design. In addition, application-specific computing systems do not create a software base. If the processors are custom designed, there is no programmer community outside of the user group for that specific system, which implies that all software tools, even such basic tools as an assembler or compiler, must be newly developed. If the base processor is widely used the most basic tools can be shared, but other software components, such as a run-time system, must still be generated for each system. The iWarp component [Borkar, et al. 90b], jointly developed by Intel and Carnegie Mellon, intends to solve some of the problems mentioned above. iWarp is unique in that it integrates computation, an instruction cache, a large register file, and communication — all on a single VLSI component. By providing support for both computation and communication, iWarp is a suitable target for program development tools and forms the core of building blocks for high-performance, application-specific, computing systems [Baxter, et al. 90].

2 Program development tools

There are two aspects to programming: the development, preparation, and maintenance of programs, and the execution of programs on the target system. The former requires a *tool chain*; the latter, a suitable execution environment.

2.1 Parallel program generators

The programming of parallel systems requires more than just a compiler or assembler for each processor in the system. Each parallel program incorporates a view of the parallel computer known as a *programming model*. The programming model gives the parallel computer certain effective performance characteristics and makes it capable of computing some kinds of operations efficiently and others less efficiently or not at all. For example, in a signal processing program two programming models can be used—data partitioning and task queue. In data partitioning, each processor gets a portion of the data, and computes the output on its portion. This model can be used so long as the operation is local—for example, convolution. In the more general, task-queue model, the operation to be performed is divided into a large set of tasks, which are dealt out to processors, with each processor requesting a new task as it finishes an old one.

In a hand-written program the programmer builds up both the program and programming model. The programming model must be built from more general, basic, constructs; as a result, the programmer sacrifices efficiency (since the general constructs are less efficient than special constructs especially for the programming model), suffers long system development time, and constructs a brittle program that is difficult to change (since the programming model constructed is specific to a program).

By contrast, with application- and programming model-specific development tools, the programming model has been built by someone expert in implementing that model on the parallel computer; the programmer writes only the program. Then this program is translated into programs for each cell in the system; these cell programs contain the low-level constructs required to implement the programming model. For this reason, we refer to these tools as *parallel program generators*. This approach provides the user with a higher level abstraction, which brings numerous benefits:

1. Clarity. A programmer specifies the problem at a higher level; instead of using the implementation level of the programming model, the level of the programming tool is used.
2. Simplicity. Application-specific tools can provide special programming constructs for the applications area and programming model, making the resulting programs more succinct and easier to write. For example, a tool for image processing can contain the concept of image boundary.
3. Efficiency. Application-specific tools can exploit expert knowledge about the applications domain and parallel computer to generate better programs than would normally be written by hand.
4. Flexibility and portability. Application-specific program generators generally hide the processor configuration from the programmer. The configuration can be changed without modification of program. The processor configuration might even be changed at runtime, allowing the entire applications-specific system to be reconfigured in response to changing requirements or sensor events.
5. Integration. The program development tools can generate their programs to use a common interface, allowing separately generated programs to be integrated into a complete processing system.

The fundamental reason that application-specific program development tools can do all of this is that work on these tools can be distributed over all programs developed for their applications area, instead of having to be done again for each hand-written program. We have developed such tools for two important applications areas for iWarp, namely signal and image processing.

2.1.1 A program development tool for signal processing

As discussed in [Baxter, et al. 90] digital signal processing (DSP) includes a large class of programs that can be represented as flow graphs: Sensor data flows into one or more nodes, and then a series of operations are done, with the processed data flowing from node to node. We developed a tool, called ASSIGN, for building large DSP applications on iWarp [O'Hallaron 91a].

ASSIGN inputs a flow graph, which describes a DSP application, and a physical array graph, which describes the topology of the target iWarp array. Each vertex in a flow graph corresponds to a coarse-grained sequential task; each edge corresponds to a buffer between two tasks. Tasks execute an infinite sequence of input, compute, and output actions.

Flow graphs are built from **C** programs augmented with routines supplied by ASSIGN. Physical array graphs are supplied with the system, although they can be easily constructed with a text editor. ASSIGN maps the flow graph onto the physical array graph, with the goal of maximizing throughput, and then produces a **C** program for each iWarp cell.

The iWarp pathway mechanism allows for a clean and straightforward implementation of flow graphs on the iWarp array. For a given mapping, an edge in the flow graph whose incident vertices are on different iWarp cells is called an ‘‘exposed’’ edge. Each exposed edge in the flow graph corresponds directly to a pathway in the iWarp. Further, the flow control for the FIFO buffer corresponding to each exposed edge is automatically provided for by the iWarp DMA mechanism. Thus, each exposed edge is identified with a pathway connecting the sending and receiving cells.

2.1.2 Program development tools for image processing

Image processing operations take an image as input, and generate an image or other data structure as output. If the output is also an image, the operation can either be *local* —each output pixel depends on a small corresponding area of the input, or *global* —each output pixel depends on all or a large area of the input. If the output data structure is not an image, the operation is always global.

Local image-processing operations can be parallelized by dividing the input and output among processors, as the APPLY parallel program generator does. . The APPLY programmer describes the desired operation more or less as it would be described in the inner core

ED— check w/ Roy

of a loop over the image pixels, except that, conceptually, the operation occurs in parallel on all image pixels at the same time. The APPLY program generator produces an efficient program for iWarp (and a variety of other parallel computers) that executes the operation on an input image. The number of iWarp processing cells and the image size can be varied arbitrarily without modification of the APPLY program.

Global image processing operations can be parallelized using a programming model known at *split and merge*. In this model the input image is divided among processors, each of which processes its portion separately, and then the independent results are combined. It can be shown that this model makes it possible to compute any image processing operation that can be computed in forward or reverse order over the image. This model is implemented in the ADAPT program, which produces code for iWarp (as well as other computer systems) [Webb 92].

2.1.3 FX - a Fortran based parallel program generator

Both ASSIGN and ADAPT simplify the programming of the iWarp system, but they are both special-purpose languages. To widen the user base, we have recently started to incorporate the key ideas into a Fortran program generator [Gross, et al. 92a], where the application-specific knowledge is encoded in directives and application-specific hints [Subhlok, et al. 93].

2.2 Execution environment

The core of the iWarp execution environment is the Runtime System (RTS), developed by Intel Corp. as part of the iWarp effort. RTS includes a resident runtime system on each cell, plus some *proxy* routines that allow each cell to invoke UNIX functions, such as opening a file, on the host. The addition of a resident runtime system is a major innovation for systolic computing. As a result, a number of services commonly associated with general purpose microprocessors are now available on iWarp as well. These services include, for example, a remote debugger and remote I/O. The remote debugger allows the local memory of an array cell be inspected by a debugger executing on a host workstation. Remote I/O allows a file located in either a workstation file system or a file system connected to a peripheral cell to be accessed by an application program executing on an array cell.

Two communications services are provided to address both the performance requirements of application programs as well as to create a usable environment for program development.

- Program communication service (PCS)—the application program manages communication explicitly and can therefore perform systolic communication [Hinrichs 91].
- RTS communication service—communication is managed by the runtime system. Services include native mode implementation of a
- reliable datagram protocol.

For example, with PCS, the application program determines the set up and sharing of pathways between programs, and if resources are shared by multiple cells, then the application program has to ensure that no cell causes starvation [Borkar, et al. 90b, Hinrichs 91]. By contrast, RTS guarantees that no cell causes starvation by making sure that no cell sends messages that exceed a specific size limit. PCS services incur less overhead (per word sent or received), however, and the comfort of the RTS services comes at a price.

2.3 Debugging array level programs

Parallel program generators simplify programming a parallel system, but they complicate the debugging of such a program if no additional tools are supplied. If, in the absence of a debugger, a user wants to debug a program translated by a parallel program generator, the user must understand and follow the mapping employed by the program generator. That is, the user has to debug the output of the program generator and has to deal with all the details of parallelism. However, the reason for using a parallel program generator is just to avoid having to deal with the parallelism.

The solution to this problem is to build an appropriate debugger that understands the mapping strategies of the parallel program generator. We have developed the framework for such a debugger and classified the optimizations commonly employed by parallel program generators so that the implementor of a parallel program generator is given guidance on which mapping strategies can be debugged and which cause problems [Cohn 92]. To the best of our knowledge, this is the first time that the issue of debugging automatically parallelized programs has been addressed from a compiler viewpoint.

2.4 Parallel compilation

Research in compiler optimization has led to better methods to translate programs for high-performance systems. Optimizing compilers can produce highly efficient code but incurs a high cost in compilation: Compilation times measured in hours are not unusual. This is not surprising considering the number of hard problems that have to be solved in the process of compiling programs for supercomputers with multiple pipelined functional units, complicated memory systems, and unusual register organizations. Furthermore, a number of important optimizations (like loop unrolling, procedure in-lining, or trace scheduling) increase the size of the program to be compiled and thereby make a bad situation even worse.

Our need to speed up compilation, combined with the desire to use a parallel system for an application outside of the domain of scientific computing, led us to investigate parallel compilation. Compilers are nontrivial programs, and mapping a compiler onto a parallel system provides a realistic test case for the problems encountered when developing realistic applications for a parallel system.

We parallelized the compilation of programs at several levels:

- *Source program units:* The structure of a parallel program, for most parallel systems, reflects the underlying machine architecture in that there can be a different program for each cell (node) of the system. Since these programs execute independently, optimizations performed within one section program are independent of those performed within another.
- *Basic block level:* An optimizing compiler often gathers information about a function (compilation unit) in such a way that the global information is obtained by combining the local information obtained for each basic block. A good example of this is global dependence information that forms the basis for common subexpression elimination, constant folding, dead code elimination, and a number of other common optimizations. The information for each basic block, however, can be obtained independently and in any order, as long as the combine operation is done correctly. This observation allows us to compute global data flow and dependence information in parallel [Zobel 90a] [Zobel 90b].
- *Global operations:* Are the hardest to parallelize, since there is no obvious partitioning of the computation. One example of a difficult optimization is global register allocation, an important optimization for compilers for modern microprocessors. It is well-known that, in general, global register allocation is NP complete. Although this fact may at first discourage parallelization, it is important to understand why this optimization is NP complete. The reason is that, *in general*, the interference graph (which indicates which variables can and which variables cannot share a register) can be an arbitrary graph. However, real interference graphs for real programs are not arbitrary, since the structure of a program influences the shape of the interference graph. That is, if the structure of the program is considered when building the interference graph, it is possible to determine for

some programs that the interference graph is a specialized graph (known as an interval graph), and it is well-known that such specialized graphs can be operated on in parallel. We have implemented this optimization in a compiler [Zobel 92].

Our experiments show that large grain parallelism is a successful step towards parallelizing compilers. Parallel languages for parallel target architectures are a rewarding source for large-grain parallelism.

3 iWarp experience

During the contract period, Carnegie Mellon has operated three iWarp systems. These systems have been used to retarget our earlier program generators and to support novel application development. We highlight here three examples of this usage:

3.1 Parallel LAPACK

Implementation of linear algebra algorithms on high performance computers has attracted considerable attention in recent years. Parallel computers offer new challenges for implementing linear algebra algorithms. Parallelism, efficient, local computation, and communication overhead are the important factors that need to be taken into account for getting good performance. One of the important decisions that an algorithm designer has to make is how to distribute data among the processors on a distributed-memory, parallel machine. Data distribution is an important factor that influences load balancing and communication overhead. For machines with memory hierarchy and true parallel processing, it is desirable to partition matrices into blocks and perform computations by matrix-matrix operations on blocks.

We developed a general method to implement blocked linear algebra algorithms in LAPACK on iWarp parallel computer. Matrices in the problem are divided into two dimensional blocks and each block is distributed uniformly on the two dimensional processor array. BLAS3 routines are generally implemented as systolic routines over the processor array, a strategy well supported by the iWarp communication architecture. Relatively less effort is needed to develop parallel LAPACK routines from the sequential ones, once the BLAS routines are in place and the procedure can be automated [Kung and Subhlok 91].

3.2 Sorting

Sorting is an important application and communication patterns similar to those encountered by sorting occur in a wide range of applications. We implemented a sorting package based on the observation that a binary hypercube can be mapped onto lower dimensional iWarp meshes. This virtual hypercube can then execute any well-known hypercube-based algorithm. We analyzed this method in a model derived from the architecture of a modern mesh machine, the iWarp, and found that this approach provides excellent performance.

The measured sorting rate of more than 2×10^6 keys/s on an iWarp torus with just 64 processors shows the excellent *absolute performance* of our approach [Stricker 92]. These performance results compare well with much larger parallel computers.

3.3 Finite element methods

In collaboration with other researchers at Carnegie Mellon and elsewhere, we recently started exploring automated systems for partitioning, placing, and routing data that is necessary for the efficient parallel solution of large problems in scientific computing, specifically the numerical solution of partial differential equations [Schwabe, et al. 92]. Our key insight was to use geometric information from the original problem to find graph separators that allow us to partition the problem into subproblems with good load balancing properties and a relatively small amount of communication between subproblems.

4 iWarp/Nectar integration

The integration of Nectar with iWarp is detailed in the next section.

I. Bibliography

[Adl-Tabatabai 92]
Adl-Tabatabai, A.
Nonresident and Endangered Variables: The Effects of Code Generation Optimizations on Symbolic Debugging.
Technical Report CMU-CS-92-221, School of Computer Science, Carnegie Mellon University, December, 1992.

[Adl-Tabatabai and Gross 92]
Adl-Tabatabai, A., and T. Gross.
The Effects of Register Allocation and Instruction Scheduling on Symbolic Debugging.
In *Proc. Supercomputer Debugging Workshop '92*, pages 115-126. Los Alamos National Laboratory, Dallas, October, 1992.

[Adl-Tabatabai and Gross 93a]
Adl-Tabatabai, A., T. Gross, G. Lueh, and J. Reinders.
Modelling Instruction-Level Parallelism for Software Pipelining.
In *Proceedings of IFIP WG 10.3 (Concurrent Sytems) Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 321-330. IFIP WG 10.3, North Holland, Orlando, FL., January, 1993.

[Adl-Tabatabai and Gross 93b]
Adl-Tabatabai, A., and T. Gross.
Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging.
In *Proc. 20th POPL Conf.*, pages 371-383. ACM, January, 1993.

[Adl-Tabatabai and Gross 93c]
Adl-Tabatabai, A., and T. Gross.
Detection and Recovery of Endangered Variables Caused by Instruction Scheduling.
In *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pages 13-25. ACM, June, 1993.

[Adl-Tabatabai and Gross 94]
Adl-Tabatabai, A., and T. Gross.
Symbolic Debugging of Globally Optimized Code: Data Value Problems and Their Solutions.
Technical Report CMU-CS-94-105, School of Computer Science, Carnegie Mellon University, January, 1994.

[Adl-Tabatabai and Gross 95]
Adl-Tabatabai, A., and T. Gross.
Engineering a Global Optimizer and Code Generator for Reuse.
Proc. Fifth Workshop on Compilers for Parallel Languages, Malaga, Spain (Tech Report UMA-DAC-95/09), pages 395-407.
1995

[Agarwal 91] Agarwal, A.
Limits on Interconnection Network Performance.
IEEE TPDS 2(4):398-412, October, 1991.

[Agrawal, et al. 93]
Agrawal, G., A. Sussman, and J. Saltz.
An Integrated Runtime and Compile-Time Approach for Parallelizing Structure and Block Structured Applications.
Technical Report 93-77, ICASE, 1993.

[Aiken and Nicolau 90]
Aiken, A., and A. Nicolau.
A Realistic Resource-Constrained Software Pipelining Algorithm.
In Gelernter, D., T. Gross, A. Nicolau, and D. Padua (editors), *Research Monographs in Parallel and Distributed Computing*. Volume : *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 274-290. Pitman/MIT Press, Irvine, CA, 1990.

[Amarasinghe and Lam 93]
Amarasinghe, S., and M. Lam.
Communication Optimization and Code Generation for Distributed Memory Machines.
In *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation*, pages 100-111. Albuquerque, NM, June, 1993.

[Amarasinghe, et al. 93]
Amarasinghe, S., J. Anderson, M. Lam, and A. Lim.
An Overview of a Compiler for Scalable Parallel Machines.
In *Lecture Notes in Computer Science*. Volume 768: *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 253-272. Springer Verlag, Portland, OR, 1993.

[Anderson and Lam 93]
Anderson, J., and M. Lam.
Global Optimization for Parallelism and Locality on Scalable Parallel Machines.
In *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation*, pages 112-125. Albuquerque, NM, June, 1993.

[Annaratone, et al. 91]
Annaratone, M., E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J. Webb.
The Warp Machine: Architecture, Implementation and Performance.
In Lilja, D. (editor), *Architectural Alternatives for Exploiting Parallelism*, pages 338-353. IEEE Computer Society, 1991.
First appeared in the IEEE Trans. on Computers, Dec. 1987, pages 1523-1538.

[Baheti, et al. 90] Baheti, R., D. O'Hallaron, and H. Itzkowitz.
Mapping Extended Kalman Filters onto Linear Arrays.
IEEE Transactions on Automatic Control 35(12):1310-1319, December, 1990.

[Baxter, et al. 90] Baxter, B., G. Cox, T. Gross, H.T. Kung, D. O'Hallaron, C. Peterson, and J. Webb.
Building Blocks for a New Generation of Application-Specific Computing Systems.
In *Proceedings of IEEE Application Specific Array Processor Conference*, pages 190-201. IEEE, Princeton, New Jersey, September, 1990.

[Beguelin and Bruegge 94]
Beguelin, A., and B. Bruegge .
A configurable monitoring system for parallel programming.
In *Proceedings of 2nd International Workshop on Configurable Distributed Systems*, pages 206. IEEE Computer Society Press, Pittsburgh, March, 1994.

[Beguelin, et al. 91a]
Beguelin, A., J. Dongarra, G. Geist, R. Manchek, and V. Sunderam,
Heterogeneous Network Computing.
In *Proc. 2nd Intl. Seminar on the Design and Application of Parallel Digital Processors*. IEE, April, 1991.

[Beguelin, et al. 91b]
Beguelin, A., et. al.
A User's Guide to PVM
Oak Ridge National Laboratories, Oak Ridge, TN, 1991.

[Beguelin, et al. 93]
Beguelin, A., J. Dongarra, A. Geist, and V. Sunderam.
Visualization and Debugging in a Heterogeneous Environment.
IEEE Computer 26(6):88-95, June, 1993.

[Bielak, et al. 92] Bielak, J., and others.
Earthquake Ground Motion Modelling in Large Basins.
Proposal to the National Science Foundation.
April, 1992

[Bielak, et al. 93] Bielak, J., and others.
Earthquake Ground Motion Modelling in Large Basins.
Proposal to the National Science Foundation.
June, 1993

[Blelloch 92] Blelloch, G.
NESL: A Nested Data-Parallel Language.
Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, 1992.

[Blelloch and Sabot 90]
Blelloch, G., and G. Sabot.
Compiling Collection-Oriented Languages onto Massively Parallel Computers.
Journal of Parallel and Distributed Computing 8(2):119-134, February, 1990.

[Blelloch and Sipelstein 91]
Blelloch, G., and J. Sipelstein.
Collection-Oriented Languages.
Proc. IEEE 79(4):504-523, April, 1991.

[Borkar, et al. 90a]
Borkar, S., R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb.
Supporting Systolic and Memory Communication in iWarp.
In *Proc. 17th Intl. Symposium on Computer Architecture*, pages 70-81. ACM, May, 1990.
A revised version appeared as technical report CMU-CS-90-197.

[Borkar, et al. 90b]
Borkar, S., R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb.
Supporting Systolic and Memory Communication in iWarp.
Technical Report CMU-CS-90-197, School of Computer Science, Carnegie Mellon, December, 1990.
First published in Proc. 17th Intl. Symposium on Computer Architecture, pp. 70-81.

[Bornstein and Steenkiste 94]
Bornstein, C., and P. Steenkiste.
Data Reshuffling in Support of Fast I/O for Distributed-Memory Machines.
In *Proceedings of the Third International Symposium on High-Performance Distributed Computing*, pages 227-235. IEEE, San Fransisco, August, 1994.

[Bruegge and Steenkiste 91]
Bruegge, B., and P. Steenkiste.
Supporting the Development of Network Programs.
In *Eleventh International Conference on Distributed Computing Systems*. 1991.

[Bruegge, et al. 93]
Bruegge, B., T. Gottschalk, and B. Luo.
A framework for dynamic program analyzers.
In *8th Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1993)*, pages 65-82. ACM, Washington, September, 1993.

[Cate 92]
Cate, V.
Alex - a Global Filesystem.
In *Proceedings of the Usenix File Systems Workshop*, pages 1-11. Ann Arbor, MI, May, 1992.

[Chen and Bershad 93]
Chen, J., and B. Bershad.
The Impact of Operating System Structure on Memory System Performance.
In *Proc. 14th Symp. on Operating Systems Principles*. ACM, December, 1993.

[Cohn 91] Cohn, R.
Source Level Debugging of Automatically Parallelized Code.
In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 132-143. Santa Cruz, California, May, 1991.

[Cohn 92] Cohn, R.
Source-level Debugging of Automatically Parallelized Programs.
PhD thesis, School of Computer Science, Carnegie Mellon, October, 1992.

[Cooper, et al. 90] Cooper, E., P. Steenkiste, R. Sansom, and B. Zill.
Protocol Implementation on the Nectar Communication Processor.
In *SIGCOMM90*, pages 135-143. ACM, Philadelphia, September, 1990.

[Dinda and O'Hallaron 95]
Dinda, P., and D. O'Hallaron.
The Impact of Address Relation Caching on the Performance of Deposit Model Communication.
In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. 1995.
To appear.

[Dinda, et al. 94] Dinda, P., T. Gross, D. O'Hallaron, E. Segall, J. Stichnoth, J. Subhlok, J. Webb, and B. Yang.
The CMU Task Parallel Program Suite.
Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, March, 1994.

[Feldmann, et al. 92]
Feldmann, A., T. Gross, D. O'Hallaron, and T. Stricker.
Subset Barrier Synchronization on Private-Memory Machines.
In *Proc. SPAA 92*, pages 209-218. ACM, San Diego, June, 1992.

[Feldmann, et al. 93]
Feldmann, A., T. Stricker, and T. Warfel.
Supporting Sets of Arbitrary Connections on iWarp through Communication Context Switches.
In *Proc. SPAA*, pages 203-212. ACM, June, 1993.

[Fisher and Gross 91]
Fisher, A.L., and T. Gross.
Teaching the Programming of Parallel Computers.
In *Proc. of 1991 SIGCSE Technical Symposium*, pages 102-107. ACM Special Interest Group on Computer Science Education (SGICSE), San Antonio, TX, March, 1991.

[Fisher and Gross 92]
Fisher, A.L., and T. Gross.
Teaching Empirical Performance Evaluation of Parallel Programs.
In *Proc. of 1992 SIGCSE Technical Symposium*, pages 309-313. ACM Special Interest Group on Computer Science Education (SGICSE), Kansas City, MO, March, 1992.

[Gross 93a] Gross, T.
An Overview of Programming the iWarp System.
Intl. J. of High Speed Computing 5(3):379-401, 1993.

[Gross 93b] Gross, T.
The Evolution of Systolic Computers: From Special-Purpose Processors to Parallel Computing Systems.
In Fahlman, S. (editor), *Computer Science Annual Review*, pages . School of Computer Science, Pittsburgh, PA, 1993.

[Gross and Hinrichs 92]
Gross, T., and S. Hinrichs.
Debugging a Parallel Program: Capturing Inter-Processor Communication in an iWarp Torus.
In *Proc. Supercomputer Debugging Workshop '92*, pages 239-276. Dallas, October, Oct. 1992.

[Gross and Steenkiste 90]
Gross, T., and P. Steenkiste.
Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compiler.
Software: Practice & Experience 20(2):133-155, February, 1990.

[Gross and Steenkiste 94]
Gross, T., and P. Steenkiste.
Architecture Implications of High-Speed I/O for Distributed-Memory Computers.
In *Proc. Intl. Supercomputing Conference ISC94*, pages 176-185. ACM, Manchester, England, July, 1994.

[Gross, et al. 92a] Gross, T., S. Hinrichs, G. Lueh, D. O'Hallaron, J. Stichnoth, and J. Subhlok.
Compiling Task and Data Parallel Programs for iWarp.
In *Proc. of the Second Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, pages 32-35. SIGPLAN, Boulder, CO, September, 1992.

[Gross, et al. 92b] Gross, T., A. Hasegawa, S. Hinrichs, D. O'Hallaron, and T. Stricker.
The Impact of Communication Style on Machine Resource Usage for the iWarp Parallel Processor.
Technical Report CMU-CS-92-215, School of Computer Science, Carnegie Mellon University, November, 1992.

[Gross, et al. 94a] Gross, T., A. Hasegawa, S. Hinrichs, D. O'Hallaron, and T. Stricker.
The Impact of Communication Style on Machine Resource Usage for the iWarp Parallel Processor.
IEEE Computer 27(12), December, 1994.

[Gross, et al. 94b] Gross, T., S. Hinrichs, and J. Subhlok.
Construction and Delivery of Messages for Modular Parallel Programs.
In Arabnia, H. (editor), *Transputer Research and Applications 7*, pages 176-185. IOS Press, Amsterdam, 1994.
Transputer and Occam Engineering Series.

[Gross, et al. 94c] Gross, T., D. O'Hallaron, and J. Subhlok.
Task Parallelism in a High Performance Fortran Framework.
IEEE Parallel and Distributed Technology 2(3):16-26, Fall, 1994.

[Gupta and Schonberg 93]
Gupta, M., and E. Schonberg.
A Framework for Exploiting Data Availability to Optimize Communication.
In *Lecture Notes in Computer Science*. Volume 768: *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 216-233. Springer Verlag, Portland, OR, 1993.

[Hinrichs 91] Hinrichs, S.
Programmed Communcation Service Tool Chain User's Guide
Release 2.8 edition, Carnegie Mellon University, 1991.
Now part of Intel RTS 3.0.

[Hinrichs and Gross 92]
Hinrichs, S., and T. Gross.
Utilizing New Communication Features in Compilation for Private-Memory Machines.
In *Lecture Notes in Computer Science*. Volume 757: *Proc. 5th Intl. Workshop, Languages and Compilers for Parallel Computing*, chapter 35, pages 563-576. Springer, 1992.

[Hinrichs, et al. 94a]
Hinrichs, S., C. Kosak, D. O'Hallaron, T. Stricker, and R. Take.
An Architecture for Optimal All-to-All Personalized Communication.
Technical Report CMU-CS-94-140, Carnegie Mellon University, School of Computer Science, 1994.

[Hinrichs, et al. 94b]
Hinrichs, S., C. Kosak, D. O'Hallaron, T. Stricker, and R. Take.
An Architecture for Optimal All-to-All Personalized Communication.
In *ACM Symposium on Parallel Algorithms and Architectures*, pages 310-319. Cape May, New Jersey, June, 1994.
A revised version is available as Tech. Report CMU-CS-94-140.

[Kung 92] Kung, H.T.
Gigabit Local Area Networks: A Systems Perspective.
IEEE Communications Magazine 30(4):79-89, April, 1992.

[Kung and Subhlok 91]
Kung, H.T., and J. Subhlok.
A New Approach for Automatic Parallelization of Blocked Linear Algebra Computations.
In *Proceedings of Supercomputing '91*, pages 122-129. Albuquerque, NM, November, 1991.

[Kung, et al. 91] Kung, H.T., P. Steenkiste, M. Gubitoso, and M. Khaira.
Parallelizing a New Class of Large Applications over High-Speed Networks.
In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. , ACM SIGPLAN, Williamsburg, April, 1991.

[O'Hallaron 91a] O'Hallaron, D.
The ASSIGN Parallel Program Generator.
In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 178-185. Portland, OR, April, 1991.

[O'Hallaron 91b] O'Hallaron, D.
Uniform Approach for Solving Some Classical Problems on a Linear Array.
IEEE Transactions on Parallel and Distributed Systems 2(2):236-241, April, 1991.

[Schwabe, et al. 92]
Schwabe, E.J., G.E. Blelloch, A. Feldmann, O. Ghattas, J.R. Gilbert, G.L. Miller, D. O'Hallaron, J.R. Shewchuk, and S. Teng.
A Separator-Based Framework for Automated Partitioning and Mapping of Parallel Algorithms for Numerical Solution of PDEs.
In *Proceedings of the 1992 DAGS/PC Symposium*, pages 48-62. June, 1992.
Revised version accepted for Comm. ACM.

[Steenkiste 92] Steenkiste, P.
Analyzing Communication Latency using the Nectar Communication Processor.
In *SIGCOMM92*, pages 199-209. ACM, Baltimore, August, 1992.

[Steenkiste, et al. 92]
Steenkiste, P., B. Zill, H.T. Kung, S. Schlick, J. Hughes, B. Kowalski, and J. Mullaney.
A Host Interface Architecture for High-Speed Networks.
In *Proceedings of the 4th IFIP Conference on High Performance Networks*, pages A3 1-16. IFIP, Elsevier, Liege, Belgium, December, 1992.

[Steenkiste, et al. 94]
Steenkiste, P., M. Hemy, T. Mummert, and B. Zill.
Architecture and Evaluation of a High-Speed Networking Subsystem for Distributed-Memory Systems.
In *Proc. of the 21st Intl. Symp. on Computer Architecture*, pages 154-165. ACM/IEEE, Chicago, 1994.

[Stichnoth 93] Stichnoth, J.M.
Efficient Compilation of Array Statements for Private Memory Multicomputers.
Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February, 1993.

[Stichnoth and Gross 95]
Stichnoth, J., and T. Gross.
A Communication Backend for Parallel Language Compilers.
Proc. Fifth Workshop on Compilers for Parallel Languages, Malaga, Spain (Tech Report UMA-DAC-95/09), pages 65-77.
1995

[Stichnoth, et al. 93]
Stichnoth, J., D. O'Hallaron, and T. Gross.
Generating Communication for Array Statements: Design, Implementation, and Evaluation.
In *Conf. Record of the 6th Workshop on Languages and Compilers for Parallel Computing*. Portland, OR, August, 1993.

[Stichnoth, et al. 94]
Stichnoth, J., D. O'Hallaron, and T. Gross.
Generating Communication for Array Statements: Design, Implementation, and Evaluation.
Journal of Parallel and Distributed Computing 21(1):150-159, 1994.

[Stricker 91]
Stricker, T.
Message Routing on Irregular 2D-Meshes and Tori.
In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 170-177. Portland, OR, April, 1991.
Also appeared as Technical Report CMU-CS-91-109, Carnegie Mellon School of Computer Science.

[Stricker 92]
Stricker, T.
Supporting the hypercube programming model on meshes; (A fast parallel sorter for iwarp).
In *Proc. SPAA 92*, pages 148-157. ACM, San Diego, June, 1992.

[Stricker and Gross 95]
Stricker, T., and T. Gross.
Optimizing Memory System Performance for Communication in Parallel Computers.
In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 308-319. ACM/IEEE, Portofino, Italy, June, 1995.

[Stricker, et al. 94]
Stricker, T., J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross.
Decoupling Communication Services for Compiled Parallel Programs.
Technical Report CMU-CS-94-139, Carnegie Mellon University, School of Computer Science, 1994.

[Stricker, et al. 95]
Stricker, T., J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross.
Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers.
In *Proc. Intl. Conf. on Supercomputing*, pages 1-10. ACM, Barcelona, July, 1995.

[Subhlok 93] Subhlok, J.
Automatic Mapping of Task and Data Parallel Programs for Efficient Execution on Multicomputers.
Technical Report CMU-CS-93-212, School of Computer Science, Carnegie Mellon University, November, 1993.

[Subhlok, et al. 93]
Subhlok, J., J. Stichnoth, D. O'Hallaron, and T. Gross.
Programming Task and Data Parallelism on a Multicomputer.
In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13-22. May, 1993.

[Subhlok, et al. 94]
Subhlok, J., D. O'Hallaron, T. Gross, P. Dinda, and J. Webb.
Communication and Memory Requirements as the Basis for Mapping Task and Data Parallel Programs .
In *Supercomputing '94*, pages 330-339. Washington, DC, November, 1994.

[Suzuoka, et al. 95]
Suzuoka, T., J. Subhlok, and T. Gross.
Performance Debugging Based on Scalability Analysis.
In *Proceeding of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 406-413. McLean, VA, February, 1995.

[Webb 92] Webb, J.
Steps Toward Architecture-Independent Image Processing.
IEEE Computer Magazine 25(2):21-31, February, 1992.

[Yang, et al. 93] Yang, B., J. Webb, J. Stichnoth, D. O'Hallaron, and T. Gross.
Do&Merge: Integrating Parallel Loops and Reductions.
In *Lecture Notes in Computer Science*. Volume 768: *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 169-183. Springer Verlag, Portland, OR, 1993.

[Zobel 90a] Zobel, A.
Parallel Interval Analysis of Data Flow Equations.
In *Proc. International Conference on Parallel Processing*. 1990.

[Zobel 90b] Zobel, A.
Parallel Compiler Optimization.
In *Proceedings of the First Workshop on Parallel Compilation*. May, 1990.

[Zobel 92] Zobel, A.
Program Structure as Basis for Parallelizing Global Register Allocation.
In *Proc. International Conference on Computer Languages*. April, 1992.

Table of Contents