A Compiler Abstraction for Machine Independent Parallel Communication Generation [*]

Bradford L. Chamberlain Sung-Eun Choi Lawrence Snyder

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

Abstract. In this paper, we consider the problem of generating efficient, portable communication in compilers for parallel languages. We introduce the IRONMAN abstraction, which separates data transfer from its implementing communication paradigm. This is done by annotating the compiler-generated code with legal ranges for data transfer in the form of calls to the IRONMAN library. On each target platform, these library calls are instantiated to perform the transfer using the machine's optimal communication paradigm. We confirm arguments against generating message passing calls in the compiler based on our experiences using PVM and MPI — specifically, the observation that these interfaces do not perform well on machines that are not built with a message passing communication paradigm. The overhead for using IRONMAN, as opposed to a machine-specific back end, is demonstrated to be negligible. We give performance results for a number of benchmarks running with PVM, MPI, and machine-specific implementations of the IRONMAN abstraction, yielding performance improvements of up to 42% of communication time and 1–14% of total computation time.

[*] This research was supported by DARPA Grant N00014-92-J-1824, AFOSR Grant E30602-97-1-0152, and a grant of HPC time from the Arctic Region Supercomputing Center.

1 Introduction and Motivation

A common compilation technique for higher level languages is to translate into a general purpose intermediate source language such as C or Fortran 77 (see Figure 1). This technique both simplifies the compiler writer's task and makes the compiler machine independent. Researchers take this approach to reduce development effort, ISV's use it to achieve portability, and hardware vendors, who nominally only have a single platform to target, use it to avoid the expense of repeatedly implementing sophisticated low-level code optimizations for each language. Unlike a byte code that is customized to the role of intermediate form, the general purpose language distances the compiler writer from machine specific optimizations. For parallel language compilers this problem is perhaps most troubling in the context of communication. To be machine independent, most parallel language compilers have adopted a message passing communication abstraction implemented by general purpose libraries such as PVM [13] or MPI [20]. Though widely supported and often vendor-optimized, the message passing abstraction is a blunt instrument for producing high performance object code. Message passing has demonstrated suboptimal performance on shared address space computers like the Cray T3D [23]. Furthermore, as explained below, the marshalling, synchronization, and buffering required by the message passing abstraction are frequently unnecessary in the context of a specific machine or instance of data transfer. Compiling to a message passing library unnecessarily binds a specific communication paradigm to the compiler, whose primary concern should be *what* data is transferred and *when* it can occur, without worrying about *how*.

Compiler writers for parallel languages are therefore confronted with a dilemma: adopt the general purpose intermediate language for simplicity and machine independence, possibly sacrificing performance due to the message passing abstraction; or accept the increased implementation and maintenance efforts of writing a different back-end for each target machine in order to reap the performance benefits of platform-specific communication.

In this paper, we propose a solution to this dilemma: the Ironman machine independent communication abstraction. The Ironman abstraction solves the problem by separating the specific interprocessor *communication mechanisms*

language	*compiler organization*	*intermediate language*
HPF	Applied Parallel Research	Fortran 77
HPF	Portland Group, Inc.	Fortran 77
ZPL	University of Washington	C
pC++	University of Indiana	C++
CC++	Caltech	C++

Fig. 1. Examples of compilers that translate a parallel source language to an intermediate source language.

provided by the hardware from the *semantics of data transfer* that are the concern of the compiler. Thus:

> Data transfer in the IRONMAN abstraction is expressed (in its most basic form) as four IRONMAN calls whose semantics are based on the Modify/Use characteristics of the data values being transferred between the source and destination processor.

In this form, data transfer between processors resembles traditional assignment, thereby meeting the compiler's fundamental data transfer needs. IRONMAN calls are no more difficult to generate than message passing calls, but they allow the compiler to abstract away the specific communication paradigm used to implement data transfer, allowing it to focus on machine independent communication optimizations. IRONMAN calls are implemented using the optimal communication mechanism of the target machine, be it message passing (SP2 [3]), put- and get-based shared memory operations (T3D [15]), or cache-coherent assignment (SGI PowerChallenge [1]), and, like message passing libraries, the calls are made available as a custom library on each platform. Compilation with this library achieves data transfer specialized to the machine rather than forcing it into the *one-size-fits-all* paradigm of message passing.

We have used the IRONMAN abstraction in the implementation of our ZPL compiler and runtime system [8], although the principle is general and applicable to any compiler for a parallel machine. In addition to describing the IRONMAN concepts, this paper reports on our experience using IRONMAN and presents performance measurements based on ZPL programs with various instantiations of the IRONMAN primitives. This paper makes the following contributions:

- Confirms claims from Stricker *et al.* [23] regarding the weaknesses of message passing, including the problems of marshalling, synchronization and buffering.
- Identifies practical problems with compilers using the MPI message passing library.
- Introduces the IRONMAN concept as a compiler-oriented abstraction that reduces data transfer to its most basic constituents and separates transfer from the implementing communication paradigm.
- Illustrates instantiations of the IRONMAN calls using a number of common communication paradigms, including put- and get-based operations and asynchronous message passing.
- Assesses the performance overhead of the IRONMAN calls.
- Presents experimental results demonstrating improved performance of programs using platform-specific implementations of the IRONMAN abstraction over that of message passing with PVM and MPI. Experiments treat five benchmarks on the Cray T3D and Intel Paragon.

This paper is organized as follows. In Section 2, we detail the IRONMAN abstraction and argue that it is an appropriate and effective alternative to abstractions based on a particular communication paradigm. In Section 3, we quantify

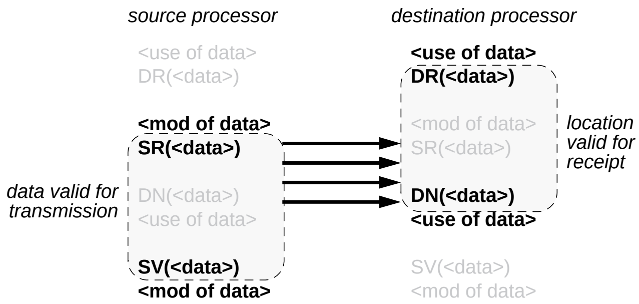

Fig. 2. Ironman calls as viewed from source and destination processor. The four calls collectively define the region in which data transfer must occur. Notice that we have illustrated an SPMD program, where a source and destination processor have the same code but execute different code segments. Therefore each processor only executes its darkened code.

the overhead of Ironman and show that it improves performance over PVM and MPI for five benchmark programs. In Section 4, we discuss related research in the area of communication abstraction. Finally, we conclude in Section 5.

2 The Ironman Interface

While application programmers may want to use standardized message passing libraries for portability, compiler writers have the resources to be much more flexible with the goal of generating portable and highly tuned code across a wide range of machines and applications. Ironman is a communication abstraction designed to allow compilers to be independent of the communication facilities of parallel machines, while avoiding the *one-size-fits-all* approach of popular communication paradigms such as message passing. Specifically, Ironman is an abstraction of data state, rather than a communication paradigm. In this section, we present a detailed description of the Ironman abstraction and an analysis of its advantages over an interface based on a particular communication paradigm.

2.1 The Ironman Abstraction

Unlike traditional interfaces, the Ironman abstraction formulates data transfer based on data state rather than a particular communication paradigm. Collectively, a set of Ironman calls demarcate a region within a program where data transfer between a source and destination processor may be required. The calls indicate the statement range in which the data is valid for transmission and receipt, as well as the source or destination location of the data on each processor. *Beyond this, the calls have no specific semantics and can be instantiated on a given machine to perform the communication as optimally as possible.*

The compiler annotates the generated code with this state information in the form of calls to the Ironman library. The basic point-to-point calls are as follows.

At the destination processor, the following two calls are relevant:

DR (*Destination Ready*). The locations at the data destination will not be used again until the transfer has completed. The destination processor is now ready to accept data from the source processor.

DN (*Destination Needed*). The values at the data destination are about to be read. Execution cannot continue until the data from the source processor has been received.

At the source processor, the following two calls are relevant:

SR (*Source Ready*). The values at the source processor will not be written again prior to transfer. The source processor is now ready to begin the data transfer.

SV (*Source Volatile*). The data at the source processor is about to be overwritten. Execution cannot continue until the transfer is completed.

As Figure 2 illustrates, the compiler uses the Modify/Use characteristics of the data being transferred to interleave the four IRONMAN calls as follows: DR, SR, DN, SV. (Note that although this figure illustrates IRONMAN in an SPMD context, the abstraction is generally applicable to any programming model). Since many data parallel programs use communication patterns in which a processor is transferring data to one processor while receiving data from another, a single processor often serves as both a source and a destination. For clarity, we will keep the two roles distinct in our discussion.

It is also important to note that some parallel machines provide more than one communication mechanism. For example, the NX libraries on the Intel Paragon [10] provide a blocking send and receive, csend and crecv, as well as the asynchronous analog, isend and irecv. IRONMAN is able to bind to either of these, allowing an implementor to provide more than one set of libraries per machine, or to switch between the mechanisms dynamically based on the characteristics of the communication. In the next section, we present example mappings for particular platforms.

2.2 Example Ironman Implementations

The following are example instantiations of the IRONMAN interface for the communication mechanisms available on the Intel Paragon and the Cray T3D.

Paragon: csend/crecv. Perhaps the most straightforward IRONMAN binding is to a message passing library that supports copy sends and receives such as the NX library's csend and crecv. In this instance, we bind SR to csend, since it is the earliest location where the source data is ready; we bind DN to crecv since it is the point at which data must be received at the destination. DR and SV are not needed to specify communication in this binding and therefore are no-ops. Note that although DR is a legal point for receiving data and SV a legal point for sending, moving the NX calls to these locations could sequentialize the communication or cause deadlock in cases where processors act as both source and destination.

Ironman *interface*		*Intel Paragon*		*Cray T3D*	*Standard Message Passing*	
program state	*call*	*NX*		*SHMEM*	*MPI*	*PVM*
destination ready	DR	–	irecv	synch	MPI_Irecv	–
source ready	SR	csend	isend	shmem_put	MPI_Isend	pvm_send
destination needed	DN	crecv	msgwait	synch	MPI_Wait	pvm_recv
source volatile	SV	–	msgwait	–	MPI_Wait	–

Fig. 3. Ironman bindings for the NX library routines on the Paragon, the SHMEM library routines on the T3D, and the standard message passing libraries.

Paragon: isend/irecv. A slightly more interesting instantiation of Ironman is demonstrated with non-blocking sends and receives such as the NX isend and irecv routines. In this case, DR can be used to post the non-blocking receive (irecv), while SR posts the non-blocking send (isend). DN is then implemented as a wait for the receive to complete (msgwait), and SV as a wait for the send to complete (msgwait).

T3D: shmem_put. The last example describes the Ironman binding for a deposit-based interface using a non-blocking *put* operation such as the T3D's shmem_put. In DR the destination *put*s a flag on the source, indicating that it is ready to receive data. In SR the source checks that the flag is set and then *put*s the data to the destination processor, followed by a flag indicating that the transfer is done. In DN the destination waits for this flag to be set and then continues execution. SV is not needed in this binding.

These examples are summarized in Figure 3, along with mappings for PVM (equivalent to NX's csend/crecv) and MPI (equivalent to NX's isend/irecv).

Note that although we've only defined Ironman in the context of point-to-point data transfers, the same principles are applicable to other communication patterns such as broadcasts, reductions, and parallel prefix operations.

2.3 Problems with Compiling to Message Passing

Message passing is a communication paradigm that requires data marshalling, synchronization, and buffering. Depending on the application and target machine, these overheads may potentially be eliminated to improve performance.

- Marshalling the data is required to bring disparate items together to form a message. In cases where data is adjacent in memory, marshalling is not necessary. When it is disjoint, marshalling is required to form a message even though some machines (T3D) are capable of directly transmitting scattered data without linearizing it.
- Synchronization is needed to preserve message passing semantics, and to ensure that the transfer has occurred, either in principle or in fact. Stricker *et al.* demonstrate that message passing's synchronization causes a performance degradation on the T3D [23].

– Buffering is often used by message passing libraries on the source and/or destination processors. This prevents programs from blocking during calls to the message passing library and allows data to arrive before its corresponding receive is posted. Buffering plays an important role in performance since it involves copying the message in order to relax the tight synchronization between processors that would otherwise be required.

Depending on the specific interface, message passing on a parallel machine can involve all of these issues. By binding these characteristics to the interface, message passing disables the compiler's opportunities to optimize away operations that are unnecessary for a particular data transfer or machine. Message passing forces the user to accept these facilities en masse, whereas the IRONMAN abstraction imposes minimal requirements by simply specifying where the data is located and where the bounding states of the transfer are located.

Applications programmers choose to use standardized message passing libraries because they aid in the effort of quickly writing portable parallel programs in Fortran or C. Compiler writers have made the same choice for the same reasons, even though message passing doesn't suit their needs as effectively. Whereas users need a tool that is intuitive and easy to use, compiler writers need optimal abstractions for the customized code that they produce to implement a language's high level semantics. Based on our experience with MPI, we make the following observations which demonstrate that performance and portability are often at odds in message passing libraries.

Standard interfaces such as MPI provide a wide variety of message passing models without regard for an efficient implementation of the solutions. For example, MPI 1.1 provides eight different send operations, including buffered, synchronous, asynchronous, and *ready* mode (in which the corresponding receive must be posted before the send); MPI 2.0 provides additional point-to-point communication models such as one-way communication. This variety is offered in the hope that at least one of them will closely match the machine's native communication mechanism, yielding good performance. Note however that the optimal model will vary from machine to machine, and furthermore that each machine will have a number of models that perform sub-optimally. Therefore, portable data transfer performance is undermined since the optimal communication routines now depend on the target machine in addition to the particular application.

Although MPI's interface is strictly defined, the functionality provided by each routine is often underspecified, forcing users to make overly conservative assumptions to ensure portability. Again, to enable the MPI implementor to do the best job possible for a given platform, the exact details of the MPI routines are not specified. For example, MPI's basic send and receive operations, MPI_Send and MPI_Recv, do not specify whether or not the transferred data is buffered internally. An immediate consequence of this is that MPI_Send may not return before its corresponding receive operation has completed. As this example illustrates, it is difficult to generate communication that is portable and efficient without explicit knowledge of the MPI routines' characteristics.

machine	*communication library*	*node characteristics*		
		processor	*operating system*	*timer granularity*
Intel Paragon	NX (native) MPICH MPI (message passing)	Intel i860 XP 50 MHz	OSF/1 1.0.4	∼100 ns
Cray T3D	SHMEM (native) CRI/EPCC MPI (message passing) Cray PVM (message passing)	DECAlpha 21064 150 MHz	MAX 1.3.0.2	∼150 ns

Fig. 4. Machine parameters and communication libraries for the Paragon and the T3D.

Though we have been targeting MPI in our examples, we believe that any interface based on a specific communication paradigm will lead to the performance and portability tensions described above. The fundamental problem with providing an abstraction based on a specific paradigm is that it is too far removed from the problem being solved, data consistency. In particular, communication is just one mechanism for maintaining data consistency. The mechanisms vary from machine to machine. For example, on cache-coherent multiprocessors, the hardware is responsible for maintaining consistency. As a general principle, consistency models should be implemented at the level of the data, not the mechanism. The Ironman abstraction adheres to this principle by specifying data state, rather than a communication mechanism, to implement consistency.

3 Experimental Results

In this section, we compare Ironman instantiations for a machine's native communication routines with those written using MPI and PVM. First, we describe our implementation and the characteristics of the target machines. Then, we measure the overhead of using Ironman. Next, we measure the impact of using various instantiations of the Ironman libraries for a purely communication-oriented micro benchmark. Finally, we evaluate five benchmark programs that use the Ironman interface with the goal of evaluating the benefits of Ironman in applications.

3.1 Methodology

Experiments were run on two platforms: the Intel Paragon [10] and the Cray T3D [15] (see Figure 4). On the Paragon, we use the MPICH [14] implementation of MPI and the native NX communication library routines. On the T3D we use a vendor-optimized version of PVM [16], CRI/EPCC MPI [6], and the native SHMEM [4] library routines. All benchmark programs were run on dedicated partitions. Measured deviations were always below 1% and therefore will not be reported. All timings were taken using the machines' native timers.

In order to compare our platform-specific Ironman implementations with message passing libraries, we implemented versions of the Ironman libraries using MPI and PVM to avoid the task of writing separate back ends for the compiler. In the next section, we demonstrate that this approach introduces negligible overhead. A summary of our IRONMAN bindings is given in Figure 3.

Our benchmarks are written in ZPL, a portable data parallel array language developed at the University of Washington. ZPL is useful for solving regular problems similar to those suitable for Fortran 90 and has been used for scientific and engineering applications [12, 19, 22, 21] as well as to implement many standard parallel benchmarks. Our ZPL compiler generates ANSI C code annotated with IRONMAN calls to indicate the required data transfers. Point-to-point communications are optimized by the compiler using message vectorization, the removal of redundant communications, and the overlapping of computation and communication [9]. Note that IRONMAN calls are no harder to insert and optimize than asynchronous message passing calls, since both rely on similar analysis of Modify/Use information. The resulting C code is compiled on each platform using its native C compiler and linked to each of the IRONMAN bindings to create the executables.

3.2 Ironman Overhead

In this section we measure the cost of calling message passing libraries via the IRONMAN interface to determine the overhead compared with a compiler that directly generates MPI or PVM calls[1]. It is shown that the overhead incurred by IRONMAN is negligible.

The parameters to the IRONMAN routines are minimal. They describe the layout of source and destination data in memory using pointers and stride information, as well as a communication ID that is used to tag a cooperating set of IRONMAN calls. Within each IRONMAN routine, the computations performed are the same as those that would be required for direct calls to MPI in an SPMD program — the processor is classified as a sender or receiver and it marshals and unmarshals data as necessary. Thus, we expect IRONMAN to incur minimal overhead as compared to the direct calls.

To verify our hypothesis experimentally, we wrote two programs to perform one million point-to-point communications. The first contains IRONMAN calls implemented using MPI, while the second calls the MPI routines directly. In both programs, the MPI routines themselves are stubbed out, allowing us to measure the overhead of calling down to the MPI interface without actually performing any communication. Thus, the difference between the execution times gives an indication of IRONMAN's overhead. We consider MPI to be a conservative upper bound for PVM since it uses all four IRONMAN calls rather than PVM's two. Figure 5 summarizes our results. Four different timings are given to indicate the measured overhead for a processor acting as a sender, a receiver, both, or neither.

[1] Although IRONMAN is intended as a replacement for direct compilation to MPI and PVM, either library can be used to implement the IRONMAN calls. This allows for a quick port to a platform supporting MPI or PVM until a machine-tailored IRONMAN library is implemented.

machine		*overhead of* Ironman *calls to MPI*	*overhead of direct calls to MPI*	Ironman *overhead*
Intel Paragon	*send*	18.93 μsec	15.60 μsec	3.33 μsec
	recv	19.28 μsec	15.59 μsec	3.69 μsec
	both	30.58 μsec	25.52 μsec	5.06 μsec
	neither	7.62 μsec	6.00 μsec	1.62 μsec
Cray T3D	*send*	19.85 μsec	16.99 μsec	2.86 μsec
	recv	21.52 μsec	18.57 μsec	2.95 μsec
	both	33.04 μsec	30.20 μsec	2.84 μsec
	neither	3.89 μsec	2.94 μsec	0.95 μsec

Fig. 5. Observed worst case overhead of using MPI in the Ironman framework. The first column indicates the time required to call down to the MPI interface using Ironman routines. The second column shows the time required to make the same MPI calls directly, without using Ironman. The last column reports the difference between these timings, indicating the worst-case overhead of Ironman. For each machine, times are given to indicate the overhead for a processor acting as a sender, a receiver, both, or neither.

Dynamic communications were counted and categorized for each benchmark in the following sections, to determine the expected overhead for using MPI and PVM versions of Ironman rather than calling into the message passing libraries directly. In all cases, the estimated overhead was less than 1% of the total running time, allowing us to conclude that Ironman does not significantly penalize message passing. We therefore use MPI and PVM implementations of Ironman for the remainder of our experiments.

3.3 Micro Benchmark

To measure the potential performance benefits of instantiating Ironman using a machine's native communication routines, we created a micro benchmark that performs eight-way nearest-neighbor communications within a tight loop. The micro benchmark is written in ZPL and linked with each Ironman implementation on both machines. The resulting executables were run on 16 processors, using a variety of processor configurations (see Figure 6). On the Paragon, the Ironman instantiations improve overall running times by 1–12%. The absolute differences are fairly small, which is expected since MPI maps well to the native message passing routines without a paradigm shift.

On the T3D, implementing Ironman using shmem_put improves overall running times by 50–65%. Ironman significantly outperforms MPI and PVM due to its ability to directly exploit the machine's preferred communication interface. The MPI and PVM versions, though highly optimized, incur the costs of marshalling, synchronization, and buffering that any implementation of message passing would require on the T3D.

Note that as the processor configurations are skewed away from a square, faster execution times result. This is partially due to the fact that in these configurations, fewer processors have all eight neighbors, resulting in less overall communication. Additionally, for configurations with more processor rows than

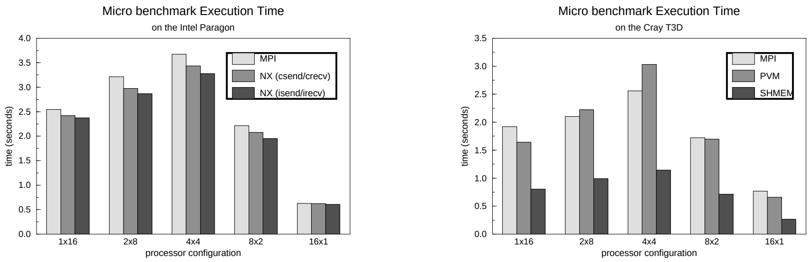

Fig. 6. Performance of a micro benchmark written in ZPL. The Ironman libraries are implemented using MPI, PVM, and the native communication libraries (NX or SHMEM). Each program performs eight-way nearest-neighbor communication in a tight loop that iterates one thousand times. The transmitted data volume is held constant for each pair of neighboring processors.

columns, the majority of communicated data is laid out sequentially in memory resulting in better spatial locality and no need for marshalling.

3.4 Benchmark Suite

Though the micro benchmark demonstrates the potential advantage of using a native Ironman implementation over MPI or PVM, it does not provide insight into how the abstraction affects overall application performance. To do this, we performed experiments using a set of benchmark programs that use kernel computations and communication patterns that are commonly found in large-scale scientific programs (see Figure 7).

Cannon's Algorithm for Matrix Multiplication (Cannon). Cannon's algorithm [7] is a systolic approach to matrix multiplication. As an initialization step, it uses cyclic shifts to skew the operand matrices. The result matrix is then computed by repeatedly multiplying elements in a pointwise manner and performing cyclic shifts, causing corresponding elements to flow past one another.

Jacobi Iterations (Jacobi). The Jacobi iteration method is a stencil computation used to model the steady state of physical systems. Every iteration, each array element is replaced by the average of its four nearest neighbors. Convergence is detected when the maximum difference between the new values and the old values is less than some constant.

Simple Hydrodynamics (Simple). The Simple code [11] is a basic hydrodynamics simulation. Our implementation uses 8-way nearest neighbor communication, global reductions, and a solver.

Shallow Water Model (SWM). The shallow water mathematical model is a finite-differencing method used in many atmospheric and fluid computations. Our version (based on the SPEC benchmark) requires communication to perform six cyclic shifts of the matrix.

benchmark program	*description*	*characteristics*	*base problem size*
Cannon	Cannon's Algorithm for matrix multiplication	1-way cyclic shift	128x128
Jacobi	Jacobi iterations	4-way nearest neighbor, reduction	256x256
Simple	Hydrodynamics simulation from Livermore Labs	8-way nearest neighbor, reduction	32x32
SWM	Weather prediction from the SPEC benchmark suite	3-way cyclic shift	64x64
Tomcatv	Thompson solver and grid generation from the SPEC benchmark suite	8-way nearest neighbor,	16x16

Fig. 7. Ironman evaluation benchmark suite. These benchmark programs demonstrate kernel computations generally found in scientific applications. All benchmark programs are written in ZPL. The global problem size is scaled so that every processor computes on the base problem size.

Thompson solver (Tomcatv). Tomcatv is a SPEC benchmark program that performs eight-way nearest neighbor communication to solve a system of linear equations.

For each of the benchmarks, the problem size is scaled proportionally to the number of processors in order to maintain a constant volume of data *per processor* for all runs. This prevents starving a processor of work and keeps data transfers at a fixed size for each processor across all runs of a benchmark. In addition, we ensure that the number of iterations remains constant (sometimes terminating prior to convergence), thereby keeping the amount of work done per processor fixed when possible. The computations in Cannon, Simple and Tomcatv are dependent on the problem size and therefore the number of iterations increases with the problem size. Figures 8 and 9 show the performance of our benchmark programs on the Paragon and the T3D. On the Paragon, the Ironman implementations using the NX libraries reduce time spent in communication by up to 13%, yielding overall improvements of 0–4%. On the T3D, the Ironman implementation using the SHMEM libraries reduces the time spent in communication by up to 42%, yielding overall improvements of 1–14%.

3.5 Interpretation of Results

The results confirm that the Ironman abstraction is a flexible mechanism for implementing data transfer to achieve maximum performance.

In the case of the Paragon where message passing is the available hardware mechanism, Ironman should be expected to offer performance comparable to the message passing libraries. However, as the micro benchmark indicates, Ironman offers an advantage over MPI even for message passing machines. As seen in Figure 6, where Ironman is instantiated with different message passing paradigms, not all message passing protocols are equal. There is a slight advantage for isend and irecv. In compilers that use the Ironman abstraction, users can benefit from

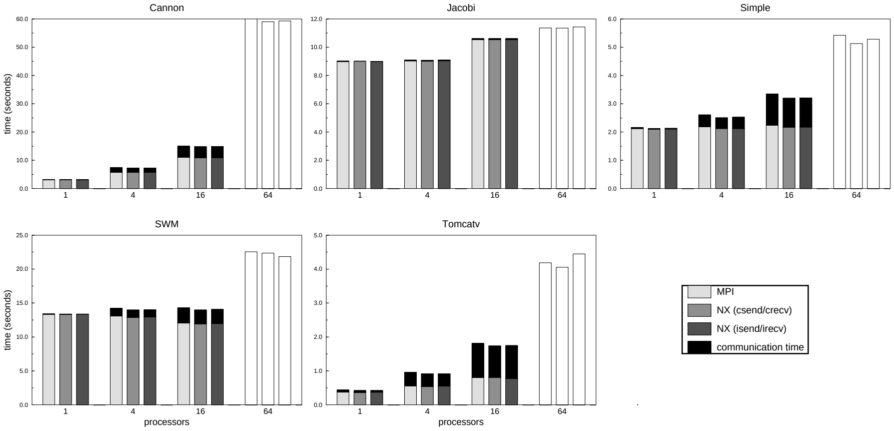

Fig. 8. Performance of benchmark programs on the Intel Paragon. The Ironman libraries are instantiated using MPI or NX. The base problem size is scaled with the number of processors to maintain a constant volume of data per processor for all runs of that benchmark. Note that our 64-node Paragon was retired before we were able to measure the time spent in communication; as a result, these numbers only show the total execution time.

a late, application specific binding of their message passing implementation. The benchmarks indicate that this advantage extends to larger applications, though the effect is diminished somewhat as communication becomes a smaller part of the overall time.

For the Cray T3D the advantages are more significant, since the hardware does not impose message passing on the user. The micro benchmark shows a substantial advantage to using Ironman instantiated with shmem_put, rather than MPI or PVM. As before, SHMEM's advantage extends to the larger applications in proportion to the amount of communication in the overall computation. The 1–14% advantage over the two messaging passing libraries is substantial considering that no communication is removed, just expressed in a form that can be more efficiently mapped to the hardware.

In summary, the Ironman abstraction allows users the option of late binding of the communication paradigms. The results show that the late binding has advantages over message passing libraries that can be substantial for modern machines like the T3D. Even message passing machines can benefit from alternative Ironman instantiations.

4 Related Work

The desire for portable high-performance data transfer has motivated a variety of communication interfaces and paradigms. Message-passing interfaces such as PVM [13] and MPI [20] were designed to provide an intuitive and portable means

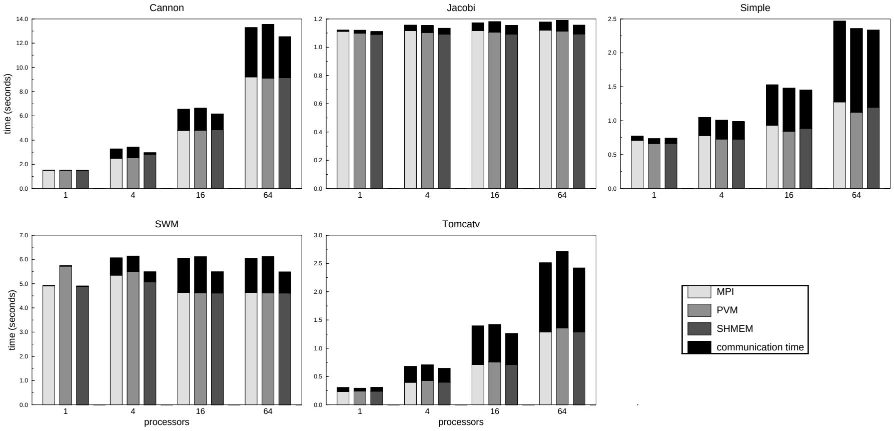

Fig. 9. Performance of benchmark programs on the Cray T3D. The IRONMAN libraries are instantiated using MPI, PVM or SHMEM. The base problem size is scaled with the number of processors to maintain a constant volume of data per processor for all runs of that benchmark.

of specifying data transfer. However, by fixing a paradigm of communication in their definitions, these libraries do the compiler-writer a disservice. When implemented on platforms whose built-in data transfer paradigm is fundamentally different — such as the T3D's SHMEM interface — these libraries necessarily add overhead to the communication time in order to make the required paradigm shift. Furthermore, the interfaces themselves can cause data to be marshalled unnecessarily on machines whose built-in communication primitives require no marshalling [23].

Some effort has been made to implement highly-tuned versions of the MPI and PVM libraries for platforms that do not inherently support send/receive-based message-passing, such as the Cray T3D [16, 6]. However, such tuning does not remove the existence of the paradigm skew and results in performance that falls short of optimal for that platform, as we demonstrate in our experiments. Another attempt to reduce the cost of message passing on the T3D has been undertaken by the Illinois Fast Messages project [17]. Their approach has been to implement a message passing library by bypassing the T3D's SHMEM library and using specific hardware characteristics. Although this hardware dependency causes their technique not to be general across platforms, it is this type of highly-tuned library that a machine-independent compiler writer wants to be able to utilize effortlessly. IRONMAN's paradigm-neutral approach would enable this.

Stricker *et al.* measured the costs involved in performing data transfer on the T3D using a variety of communication paradigms [23]. Their results quantify the effects of using a standard message-passing library like PVM, and indicate that the deposit-based paradigm outperforms others due to its reduced synchronization and buffering requirements. However, it is unlikely that a platform without built-in deposit-style communication mechanisms (such as the Intel Paragon) would be able to efficiently implement the paradigm. This is further evidence that a single data transfer paradigm will be unlikely to provide portable performance across all platforms.

Another important data transfer paradigm is the Active Messages interface designed by von Eicken *et al.* [24]. Consider this to be a highly optimized communication interface that is becoming increasingly widespread. As such, it represents technology that compiler writers would like to use when available, but might hesitate to rely upon since portability is constrained by availability. To this end, Ironman's paradigm-neutral approach allows the compiler writer to use Active Messages when it is available without relying on its presence on every platform.

Consistency models have been used to express how a shared memory changes state [2, 5, 18]. The consistency model is the mechanism by which the programmer and the compiler agree on when memory updates take place, and as such is a source language rather than a compiler concept like Ironman. A compiler supporting a memory consistency model could manage its own memory coherency using Ironman calls, with the advantage that the programs would port directly to noncoherent global address space machines such as the Cray T3D.

5 Conclusions

We have argued that implementors of parallel languages should avoid binding their compiler to a particular communication paradigm in order to achieve maximum portability without sacrificing performance. As a solution, we have presented the Ironman abstraction which circumvents the problem by providing the minimal set of information required to perform a data transfer — where the data is located in memory, and when the transfer can occur during the program's execution. This abstraction is then realized by implementing the Ironman calls on each machine so that they perform the transfer in accordance with the machine's underlying communication model. This effectively nullifies the paradigm skew that can occur when using a specific communication paradigm on a machine for which it is not well suited.

We have used the Ironman abstraction in our ZPL compiler and runtime libraries. The experiments show that Ironman's late communication binding yields improved performance, which can be substantial for machines like the T3D where the hardware provides data transfer paradigms that are less constrained than message passing. The opportunities to benefit from late binding are expected to improve as architectures continue to move away from message passing designs.

Acknowledgments. We'd like to thank the ZPL compiler group for their support of the ZPL compiler and runtime system, and the San Diego Supercomputer Center and the Arctic Region Supercomputing Center for providing us access to their parallel hardware, on which these ideas were developed and tested.

References

1. The Power Challenge. Technical report, Silicon Graphics, Inc., 1995.
2. Sarita V. Adve and Kouroush Gharachorloo. Shared memory consistency models: A tutorial. Technical Report 95/7, Digital Western Research Laboratory, 1995.
3. T. Agerwala, J. L. Martin, J.H. Mirza, D.C. Sadler, D.M Dias, and M. Snir. SP2 system architecture. *IBM System Journal*, 34(2):152–184, 1995.
4. Ray Barriuso and Allan Knies. SHMEM user's guide for C. Technical report, Cray Research Inc., June 1994.
5. Brian N. Bershad, Matt J. Zekausaka, and Wayne A. Sawdon. The Midway distributed shared memory system. In *CompCon Conference*, February 1993.
6. Kenneth Cameron, Lyndon J. Clarke, and A. Gordon Smith. CRI/EPCC MPI for CRAY T3D. In *1st European Cray T3D Workshop*, September 1995.
7. L. F. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, 1969.
8. Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. Factor-Join: A unique approach to compiling array languages for parallel machines. In *Workshop on Languages and Compilers for Parallel Computing*, August 1996.
9. Sung-Eun Choi and Lawrence Snyder. Quantifying the effect of communication optimizations. *to appear in International Conference on Parallel Processing*, August 1997.
10. Intel Corporation. *Paragon User's Guide.* 1993.
11. W. Crowley, C. P. Hendrickson, and T. I. Luby. The Simple code. Technical Report UCID-17715, Lawrence Livermore Laboratory, 1978.
12. Marios D. Dikaiakos, Calvin Lin, Daphne Manoussaki, and Diana E. Woodward. The portable parallel implementation of two novel mathematical biology algorithms in ZPL. In *9^{th} International Conference on Supercomputing*, 1995.
13. A. Belguelin et al. A user's guide to PVM. Technical report, Oak Ridge National Laboratories, 1991.
14. William Gropp and Ewing Lusk. User's guide for mpich, a portable implementation of MPI. Technical report, Argonne National Laboratory, 1996.
15. Cray Research Inc. *Cray T3D System Architecture Overview Manual.* Mendota Heights, MN, 1993.
16. Cray Research Inc. *PVM and HeNCE Programmer's Manual.* Mendota Heights, MN, 1994. SR-2501 5.0.
17. Vijay Karamcheti and Andrew A. Chien. Optimizing memory system performance for communication in parallel computers. In *Proceedings of the International Symposium on Computer Architecture*, June 1995.
18. Peter Keleher, Alan L. Cox, and Willie Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the International Symposium on Computer Architecture*, May 1992.
19. E Christopher Lewis, Calvin Lin, Lawrence Snyder, and George Turkiyyah. A portable parallel n-body solver. In D. Bailey, P. Bjorstad, J. Gilbert, M. Mascagni, R. Schreiber, H. Simon, V. Torczon, and L. Watson, editors, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 331–336. SIAM, 1995.
20. Message Passing Interface Forum. *MPI: A Message Passing Interface Standard.* June 1995.

21. Wilkey Richardson, Mary Bailey, and William H. Sanders. Using ZPL to develop a parallel Chaos router simulator. In *1996 Winter Simulation Conference*, December 1996.
22. Prasenjit Saha, Joachim Stadel, and Scott Tremaine. A parallel integration method for solar system dynamics. *to appear in Astronomical Journal*, June 1997.
23. T. Stricker, J. Subhlok, D. O'Hallaron, S. Hinrichsand, and T. Gross. Decoupling synchronization and data transfer in message passsing systems of parallel computers. In *9^{th} International Conference on Supercomputing*, July 1995.
24. Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.