Performance and Program Complexity in Contemporary Network-based Parallel Computing Systems

Technical Report No: HPPC-96-02 March 1996

Steve VanderWiel
Dafna Nathanson
David J. Lilja

UNIVERSITY OF MINNESOTA
High-Performance Parallel Computing Research Group
Department of Electrical Engineering • Department of Computer Science • Minneapolis • Minnesota • 55455 • USA

Performance and Program Complexity in Contemporary Network-based Parallel Computing Systems

Steven VanderWiel * svw@ee.umn.edu

Dafna Nathanson † nathan@mountains.ee.umn.edu

David J. Lilja * lilja@ee.umn.edu

* Department of Electrical Engineering
†Department of Computer Science
University of Minnesota
200 Union St. SE
Minneapolis, MN 55455
and
†Computing Devices, International
Bloomington, MN

ABSTRACT

It is commonly assumed that if a programmer is willing to invest the potentially significant effort required to port an application program to run on a multiprocessor system using a low-level parallel language or library, they will be able to take advantage of a larger degree of parallelism to achieve higher performance than when using a higher-level language or an automatically parallelizing compiler. However, there has been little work examining the relationship between programming complexity (or ease-of-use) and performance. As a first step towards quantifying this relationship, we use the cyclomatic program complexity metric, borrowed from the software engineering field, and the number of program source statements as measures of the relative complexity of a parallel implementation of a application program compared to it's equivalent sequential implementation. We compare several different programming paradigms across a common set of application programs executed on two workstation clusters, a shared-memory multiprocessor, an IBM SP2 and a Cray T3D. We find that message-passing languages tend to be both the most widely supported and the most complex. Other programming paradigms such as shared-memory and High-Performance Fortran produce more compact and easily understood programs than message-passing languages but are not yet well supported.

Keywords: multiprocessor performance; program complexity; HPF; automatic parallelization cyclomatic complexity; SPMD; message-passing; networks of workstations.

1. Introduction

Drawn by the promise of faster response times and increased throughput, users with large computational demands often require the performance capabilities provided by multiprocessor systems. However, once the decision is made to employ such a system, developers are faced with a daunting array of choices. Not only must potential parallel application developers choose a hardware platform, but they must also choose the best approach for porting their codes. Unfortunately, parallel language development has not kept pace with the growing demand for parallel applications and, as a result, software development costs can easily surpass that of the parallel hardware. In fact, the ease with which a programming paradigm can be used to port existing software will often dictate the choice of the parallel hardware platform.

Central to the software porting process is the choice of an appropriate parallel programming paradigm. Nearly all parallel systems offer several programming environments from which to choose, ranging from automatic parallelizing compilers to low-level machine-specific programming libraries. The appeal of the former approach is the minimal amount of reprogramming required to parallelize an application while the latter approach is generally taken to maximize performance. The specific choice of programming language or library requires the developer to weigh the performance benefits against the cost and effort for which this performance can be obtained.

Given the importance of balancing these often conflicting factors, surprisingly little empirical data exists to aid the parallel programmer in settling upon a parallel programming paradigm. For example, it is commonly believed that automatic compiler parallelization will produce parallel programs with lower performance than rewriting the application with explicit parallel constructs. Presumably, if one is willing to put forth the significant effort required to use an explicitly parallel paradigm one will be able to take advantage of a larger degree of parallelism to achieve higher performance. However, there has been essentially no work that quantifies the relationship among programming complexity and performance.

Completely evaluating all parallel languages and paradigms on all computing platforms is clearly an impossible task. Additionally, quantifying such inherently subjective factors as programming complexity is fraught with numerous definitional traps and pitfalls. Nevertheless, in this work we propose a methodology and a group of metrics for quantifying the program complexity (or ease-of-use) and compare this to the resulting performance on a variety of different parallel architectures. We examine four different programming paradigms: 1) automatic compiler parallelization, using the KAP front-end translator [15] supplied with the Silicon Graphics Challenge system; 2) message-passing, as exemplified by MPI [7] and PVM [4]; 3) shared-memory programming using both UNIX shared-memory libraries and the Cray T3D Shmem library[2]; and 4) an intrinsically parallel language, High-Performance Fortran [8].

We use the cyclomatic program complexity metric [13], borrowed from the software engineering field, and the number of source code statements [5] to quantify the relative complexity of a parallel implementation of a application program compared to it's equivalent sequential implementation. We compare performance, measured as the total execution time, on two workstation clusters, a shared-memory multiprocessor, an IBM SP2 and a Cray T3D.

In the remainder of the paper, Section 2 defines our metrics and describes our experimental methodology in more detail. Section 3 describes the experimental environment used in this study, with the corresponding results presented in Section 0. The last section summarizes our results and conclusions.

2. Metrics of Program Characteristics

2.1. Performance

Implicit in the decision to use a parallel processing system is the need for high performance. The ability of a language to produce executables which take advantage of the underlying parallel hardware is therefore a primary concern. We define a *normalized run time,* $\hat{T}_n^m$, to gauge the performance of executables compiled from the language suite :

$$\hat{T}_n^m = \frac{T_n^m}{MAX(T_n)}$$

where $\hat{T}_n^m$ is the normalized run time of test program *n* on machine *m*, T_n^m is the parallel execution time for program *n* on machine *m*, and $MAX(T_n)$ is the maximum time taken to execute program *n* across all machines. As will be shown in Section 4.1, this metric will be useful in comparing both machine performance and language performance.

In calculating T_n^m, execution time is taken to be the run time of the program kernels only. Startup and data initialization times are not included in the reported execution times. Although different parallel languages vary in the amount of time necessary to spawn processes, initialize task ids and so forth, this startup time is assumed to be small and constant relative to the overall execution time. In addition, to obviate time-sharing effects, execution times where taken during periods of dedicated machine time whenever possible. Typically, timing values were calculated by taking the average of three timing runs. If the three timing runs showed a significant variance, additional runs were added until the variance was reduced to an acceptable level.

2.2. Ease of Use

In contrast to execution time, a programming language's relative ease-of-use is difficult to quantify. Parallel programmers have an intuitive sense for this quality of a language and, after extensive use, often come to share similar opinions regarding a particular language or paradigm. However, because programmers typically have experience with only a subset of the parallel languages and are subject to personal bias, a comprehensive comparison cannot be based on an opinion poll. Rather, we borrow some techniques from the branch of software engineering known as software metrics.

Software metrics are often used to give an indication of a program's complexity. Not to be confused with algorithmic complexity, or O( ) notation, software metrics have been found to be useful in reducing software maintenance costs by giving a quantitative indication of a program module's understandability. For example, it has been found that program modules with high complexity indices have a higher frequency of failures [10]. By using software complexity metrics, software engineers isolate error-prone source code modules as those with high complexity values. These modules are then subjected to recoding or further modularization to reduce module complexity and increase overall program reliability.

Our interest in complexity metrics is motivated by the desire to quantitatively compare the relative effort of using different parallel programming languages to encode a given algorithm. As such, the effects of program modularity are ignored and all programs are treated as though they are contained in a single module. It is also important to note that we consider only user-level code when doing complexity measurements. Language library function bodies are not included since their underlying code is not specified by the application programmer and therefore does not add to the effort of coding the test programs.

So that we may concentrate on *language complexity* rather than program complexity, the metrics used in the following experiments will be normalized relative to their sequential versions. That is, given a complexity metric, C, its normalized value will be defined as

$$\hat{C} = \frac{C_L}{C_s}$$

where C_s is the value of the complexity metric when applied to a sequential version of the test program and C_L is the is value of the complexity metric when applied to a version of the test program written in language L.

Several complexity metrics exist [6], each designed to measure different factors affecting program complexity. Two popular metrics that will be used in this study are non-commented source code statements (NCSS) and McCabe's cyclomatic complexity (MCC). These metrics were chosen for their reliability as complexity indicators and their appropriateness for this study.

2.2.1. Non-Commented Source Code Statements

NCSS [5] is often used to track the size of program modules in large software projects. Since lengthy modules can become difficult to comprehend, software developers will often place an upper limit on a module's NCSS value. Note that this metric is not the same as counting lines of source code (LOC). Here, NCSS includes all executable source code *statements* without regard to the placement of carriage returns or other stylistic elements. In addition, variable declarations, preprocessor directives and comments will be excluded from the calculation of NCSS so that we may focus on the more salient features of a language.

Non-commented source code statements can also be t*hough*t of as a measurement of the quantity of source code required to accomplish a given task. Given two versions of a program, the one with a lower NCSS value suggests a cleaner solution. For example, a message-passing language supporting a one-to-all broadcast library function would have a much lower NCSS value than a language lacking this feature. The latter language would require the application programmer to encode his own broadcast function, thereby increasing the total number of source code statements.

Although NCSS is a good indication of the amount of code required to accomplish a programming task, it does not offer much information concerning the content of the program. The situation is analogous to judging two books based on their relative thickness. The books may contain the same number of pages but it would be more informative to the potential reader to know that one is written by James Joyce while the other is the latest Steven King novel. Cyclomatic complexity, introduced in the following section, complements NCSS by offering additional insight into a program's "readability."

2.2.2. Cyclomatic Complexity

McCabe introduced cyclomatic complexity in 1977 as an indication of a program module's control flow complexity. Derived from a module's control graph representation, MCC has been found to be a reliable indicator of complexity in large software projects [17]. This metric is based on the assumption that a program's complexity is related to the number of control paths through the program. For example, a 20 line program consisting of 20 assignment statements is clearly less complex than a 20 line program consisting of 10 `if-then` statements. The former program would contain a single control path whereas the latter would have over a thousand possible paths.

Unfortunately, it is generally not possible to count the total number of paths through a given program since backward branches lead to a potentially infinite number of control paths. Instead, cyclomatic complexity is defined in terms of *basic paths* which, when taken in combination, can be used to generate every possible control path through a program. From basic graph theory [1], the number of basic paths within a graph, *G*, is bounded by the graph's cyclomatic number which is defined as

$$V(G) = e - n + 2p$$

where e is the number of edges in G, n is the number of graph nodes and p is the number of connected components. For our present purposes, p will always equal 1 and the interested reader is referred to [13] for a full discussion of this parameter's role in calculating $V(G)$.

As an example of how cyclomatic complexity is applied, the program source code segment given in Listing 1 is represented as a control graph in Figure 1. This program segment's complexity would then be calculated as

$$V(G) = e - n + 2 = 7 - 6 + 2 = 3.$$

A simplified version of the program segment given in Listing 1 is shown in Listing 2. Here we condense the two `for` loops into one reducing the control flow graph to that shown in Figure 2. Note that although the two versions of the program have the same value for NCSS (6), the second version has a lower cyclomatic complexity of

$$V(G) = e - n + 2 = 4 - 4 + 2 = 2.$$

Although NCSS and MCC tend to be insensitive to programming style, this last example demonstrates how algorithmic preferences can affect complexity measurements. To mitigate the effect of individual programmer style, the programs used in this study were written using standardized style guidelines and were subjected to peer review to confirm conformance to these guidelines. Algorithmic changes in different versions of a given benchmark program were allowed only when the changes were due to features of the programming language.

Together, NCSS and MCC capture most of the salient language differences in which we are interested. More will be said about how these metrics were useful for this study in Section 4.2.

```
/* reverse the string s */

len = strlen(s);

for  ( i = 0 to len - 1)
      temp[i] = s[i];

for (  j = 0 to len - 1) {

      s[j] = temp[i];
      i = i - 1;
}
```

Listing 1. An example program

```
/* reverse s in place */

j = strlen(s) - 1;

for ( i = 0 to j){

      c = s[i];
      s[i] = s[j];
      s[j] = c;

      j = j - 1;
}
```

Listing 2. A simpler version of Listing 1

3. Experimental Environment

A suite of representative programs was used to compare the various machines and languages presented in this study. Each test program was ported to several different parallel systems using languages available on each system. It is felt that these test programs represent the types of parallel program kernels one might expect to find running on a typical parallel processing systems. The systems and languages used in this study are also considered to be a representative cross-section of parallel languages and systems currently in use. The following sections describe in more detail these programs and systems.

3.1. Test Programs

The test programs can be roughly divided into three categories, each containing two programs. The *sobel* and *filter* image processing programs where chosen to represent regular, easily parallelized code. *Hough* and *warp* require larger data transfers and use less regular communications patterns than those of *sobel* and *filter* but they do not send a large number of messages. Finally, *gauss* and *trfd* represent a class of communication-intensive problems that require a large number of various sized messages. More detailed descriptions of the test programs are given below.

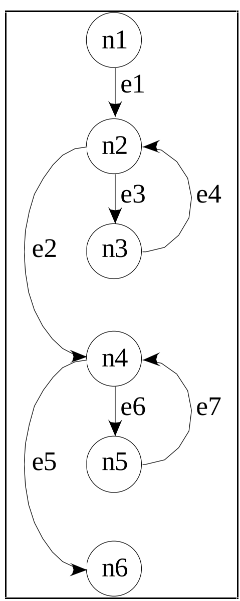

Figure 1. Control graph of Listing 1

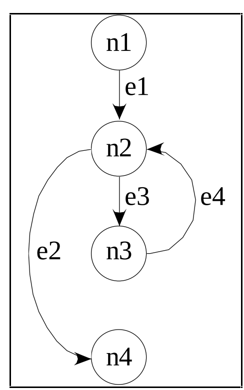

Figure 2. Control graph of Listing 2

Sobel calculates magnitude and direction gradients of an input image. An output image element value is a function of the corresponding input image value and its eight neighboring elements. Image matrices are distributed across processors by partitioning the original matrix into submatricies in a block-checkerboard fashion. Interprocessor communication results from communicating shared input data between processors holding data from adjoining input image submatricies.

Filter is an averaging (low-pass) image filtering program. This program calculates the value of an output image element as the weighted sum of up to 36 neighboring pixel elements in the input image. Data are partitioned in a fashion similar to *Sobel*.

Hough is used to detect shapes in images. As an instance of the more general *Hough* transform [3], the test program used for this project detects straight lines in the input image by finding points of intersections between lines. This algorithm therefore relies on both nearest neighbor and global communications.

Warp is a spatial domain image restoration algorithm that aligns an input image along a given axis [18]. Such algorithms are useful in restoring satellite images which have been *warp*ed due the curvature of the earth's atmosphere, for example. This program exhibits very irregular communications patterns due to the nature of the problem. Pixel elements may need to be shifted substantially depending on their proximity to the axis in question.

Gauss is an implementation of the familiar Gaussian elimination with back-substitution algorithm used to solve large systems of equations. The rows of the input matrix are distributed in a cyclic pattern to the available processors in the multiprocessor system. This algorithm is very communication intensive, requiring several point-to-point, broadcast and reduction operations.

TRFD is a member of the Perfect Club benchmark suite which simulates a two-electron integral transformation using a fourth-order tensor equation [12]. The algorithm is implemented as a series of matrix multiplications and transpositions and therefore requires several point-to-point communication operations. The serial version of *TRFD* was altered from the original Perfect Club version which was heavily optimized to reduce it's memory requirements and therefore not comparable to the parallel versions of *TRFD*. The altered version is a more straightforward implementation of the same algorithm.

3.2. Programming Languages

The above applications were written using five different parallel programming languages The word “language” is used rather loosely here. Often, parallel programs use ordinary C or Fortran compilers and link with libraries that facilitate parallel operations, such as message-passing, synchronization or setting up shared-memory arenas. The lone exception is High-Performance Fortran (HPF) which contains special parallel constructs within the language definition. However, for the sake of convenience, we will hereafter refer to all of these parallel programming paradigms as “languages.” A description of each language follows.

PowerC and **PowerFortran** [15] are parallelizing compilers for SGI symmetric multiprocessors (SMPs) that take unadorned C and Fortran source code and automatically generate parallel executables. Such compilers require no recoding of the serial program to take advantage of parallel processing hardware.

PVM [4] is a popular message-passing language developed at Oak Ridge National Labs. In a message-passing paradigm, the programmer explicitly specifies the exchange of data and process synchronization. In addition to point-to-point messages, PVM contains broadcast and reduction group communication operations. PVM version 3.3.10 was used for all non-commercial implementations. The commercial version of PVM for the SP2, PVMe 2.1, is an implementation of PVM 3.3 with extensions. Cray Research’s PVM 3.3.4 was used on the T3D.

MPI [7] is another message-passing language that shares many common features with PVM. MPI, however, is a proposed standard for message-passing with many implementations available from several different sources. MPI also differs from PVM in the way parallel processes are spawned, in the manner in which data are buffered and in other, more subtle ways. By allowing programmers more control over how a parallel computation proceeds than other proposed approaches, message-passing languages like PVM and MPI are generally assumed to provide superior performance on distributed memory multiprocessors. Most public-domain MPI implementations used in this study were MPI ch-p4 version 1.0.11 jointly developed by Argonne National Laboratory and Mississippi State University. MPI on the T3D used the EPCC implementation, version 1.3a, which was developed by Edinburgh University in collaboration with Cray Research. The commercial version of MPI used on the SP2 is version 2.1, which is an implementation of MPI 1.0.

Explicit Shared-memory programming requires the programmer to explicitly fork parallel threads and allocate shared-memory arenas through which these threads may communicate data. Although this approach still requires the programmer to explicitly specify the parallelism in a program, SMP programming is often considered to be an simpler programming paradigm than message-passing due to the presence of a single, global name space. Indeed, much of the impetus behind the design of shared-distributed memory [11,14] and shared virtual memory [9] systems is the perceived benefits of this programming model.

The **T3D Shmem library** [2] takes advantage of the T3D's shared-distributed memory architecture by allowing the programmer to directly access a remote processor's memory through simple memory-to-memory copying. The Shmem library also includes simple group communications operations. This model of programming can be seen as a cross between the message-passing and shared-memory programming models.

High-Performance Fortran [8] is another proposed standard for parallel programming. This language represents a class of parallel languages that extend the semantics of ordinary serial programming languages to include vector and array operations. In HPF, the programmer explicitly specifies data partitioning but allows the compiler to generate any necessary data communications. This language can therefore be considered a compromise between automatic parallelization and explicit parallel programming. HPF is an emerging standard with several planned implementations for various parallel architectures. The version used here is a beta release of the IBM XL HPF compiler. As such, execution times may not reflect the performance of the release version.[1]

3.3. Multiprocessor Systems

The parallel processing systems used in this study are summarized in Table 1. These systems are described in more detail below.

The **IBM SUR Cluster** is composed of 8 IBM system 590 workstations with 66MHz Power2 processors. The workstations are connected via a 100Mb/s ATM switch. The cluster was programmed using PVM and MPI.

---

[1] Author's note : The release date of XL HPF version 1.0 is scheduled soon after the time of this writing. It is expected that version 1.0 performance figures will be available in time for the final draft of this paper.

The **SGI Challenge Cluster** consists of four SGI Challenges SMPs each containing four 200MHz MIPS R4400 processors. The four systems are connected via a 266 Mbit/s Fibre Channel network. Programs running on the Challenge Cluster were written in PVM and MPI.

The **SGI Challenge** is a 16 processor SMP configured with 1GB of main memory. Each processor is a 200 MHz R4400. PowerC, PowerFortran, PVM and explicit shared-memory programming were all used on this machine.

Although the **IBM SP2** is often viewed as a custom multiprocessor system, its internal construction is similar to a cluster of IBM RS/6000 workstations. The system used in this study is composed of 10 rack-mounted 66MHz Power^2 workstation cabinets, 8 of which participated in the actual computations. The important difference between the SP2 and a true workstation cluster lies in the 40MB/s multistage network connecting the workstation cabinets and in the custom software made to take advantage of this network. In addition to public-domain implementations of PVM and MPI, commercial versions of these languages are also used to program the SP2. A commercial version of High-Performance Fortran was also used.

The **Cray T3D** consists of 512 150MHz Alpha processors connected via a custom, 300MB/s 3D torus interconnect. T3D execution times were derived from a 64 node partition. The memory of the T3D is logically shared but physically distributed across processors. The vendor-supplied version of PVM, the EPCC version of MPI and Cray's Shmem library were used to program the T3D.

Table 1. The five parallel processing systems used in this study

Machines	Processors	Interconnect	Languages
SGI Challenge Cluster	4 SGI Challenges each w/ 4 200MHz MIPS R4400 processors	shared-memory / Fibre Channel	MPIch, PVM
IBM SUR Cluster	8 IBM System 590s each w/ 66MHz IBM Power^2 processors	ATM	MPI, PVM
SGI Challenge	16 processor SGI Challenge w/ 200MHz MIPS R4400 processors	1GB shared-memory	PowerC, PowerFortran, MPIch, PVM
IBM SP2	8 "thin" nodes w/ 66MHz IBM Power^2 processors	custom multistage	HPF, MPI, MPIch, PVM, PVMe
Cray T3D	64 150MHz DEC Alpha processors	custom 3D torus	MPI, Cray PVM

4. Results

The six test programs were ported to each of the above machine/language pairs. Each version of these programs was then tested to judge the performance and ease-of-use of the language used to code the program. These criteria are compared in the following sections using the metrics and methodology defined in Section 2.

4.1. Performance Results

Using the methodology described in Section 2.1, the relative performance of different machine/languages pairs was compared. These performance results are summarized in Figure 3 - 7 below. Recall that execution times are normalized relative to the longest execution time for a particular application. For example, in Figure 4, the normalized execution time of *Warp* using MPI is unity, indicating that for all machine/language pairs, the IBM SUR Cluster using MPI produced the slowest execution time for *Warp*. This normalization technique was chosen to give both an indication of performance across machines and relative language performance within a machine. For example, from Figure 3 we see that *Hough* using MPI ran approximately twice as fast as the PVM version on the Challenge Cluster, but only about a third of the speed of the T3D Shmem version.

Note that there are some exclusions in performance results denoted by an execution time of zero. These exclusions are due to programs which, although logically correct, compiled but did not run on the given machine. These failures can sometimes be attributed to memory limitations (*Warp* on the T3D, for instance) but more often were the result of instabilities in the language implementations. Although it is often possible to circumvent troublesome communications library calls by replacing them with less intuitive calls, this approach was not taken. Rather, the failed programs were allowed to exist as a comment on the stability of the language implementation.

The large number of machine/language/application combinations makes a complete analysis of these results impractical. Instead, we will concentrate on finding general trends in the data. As a start, we compare the relative execution times of executables written in the two portable message-passing languages used in this study. Four of the five systems under study support both PVM and MPI; the two cluster-based systems, the SP2 and the T3D. Note that the SP2 provides both commercial and public-domain versions of PVM and MPI. The data labels *MPI-IBM* and *PVMe* denote the commercial versions while *MPI-ch* and *PVM* represent the public-domain versions.

Across these machines, neither MPI nor PVM showed a clear performance advantage. Within a given system, only the IBM SUR cluster showed a consistent trend with the PVM programs always out performing their MPI equivalents. Similarly, performance was not correlated to specific test programs. Indeed, what is most notable about the MPI and PVM data is its unpredictability. For example, the public-domain versions of these languages on the SP2 show significantly varying results between the *warp* and *gauss* test programs. The former program ran several times faster using PVM, but this situation is reversed for *gauss*. We again emphasize that the reported times are consistent, repeatable and are not the result of measurement fluctuations.

Referring once again to Figure 5, note that the commercial versions of PVM and MPI showed comparatively consistent behavior on the SP2. Differences in execution time between the two languages tended to less pronounced and performance was consistently better than the public domain versions. The Cray T3D results shown in Figure 6 also show relatively small variations between PVM and MPI execution times. In general, language versions implemented by the machine vendor or, as in the case of MPI on the T3D, with their collaboration, tend to produce more predictable execution times.

One partial explanation for this observation may be that these language implementations are better tuned to a specific system as a result of the vendor's involvement. For example, the SP2 supports a low-level messaging protocol that runs in user space which is used in the commercial version of MPI. This allows an alternative to using the IP protocol upon which the public domain MPI implementation is built. By circumventing the higher-level protocols, programs using the user-level protocol may send and receive messages with less software overhead. Figure 8 illustrates the benefits of such an approach. In this figure, each SP2 program uses the vendor-supplied version of MPI under three different conditions: using the SP2 high-speed switch with a user-level protocol, using the same switch with an IP protocol, and using Ethernet with an IP protocol. These times have been normalized to this last case.

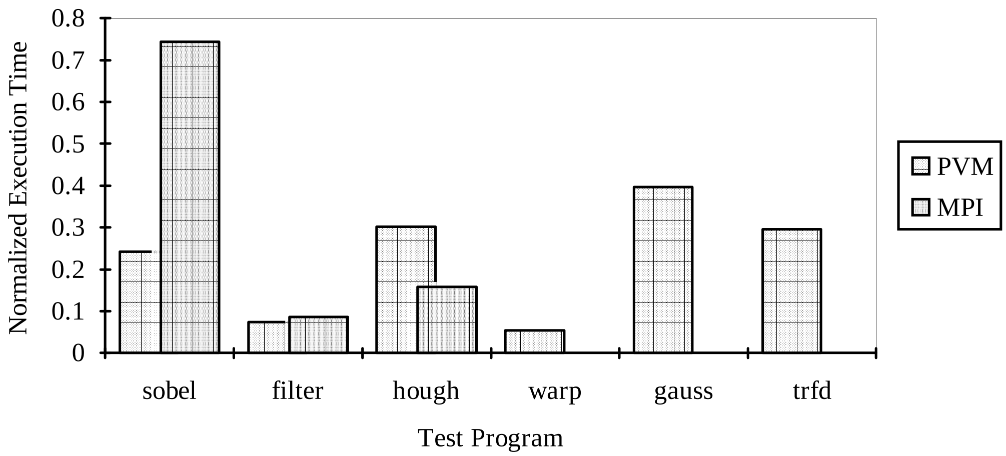

Figure 3. Language performance on SGI Challenge Cluster

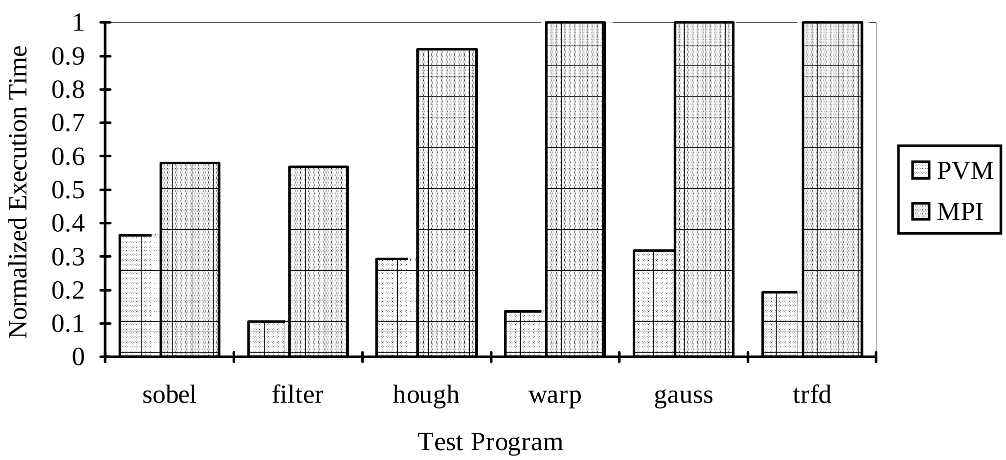

Figure 4. Language Performance on the IBM SUR Workstation Cluster

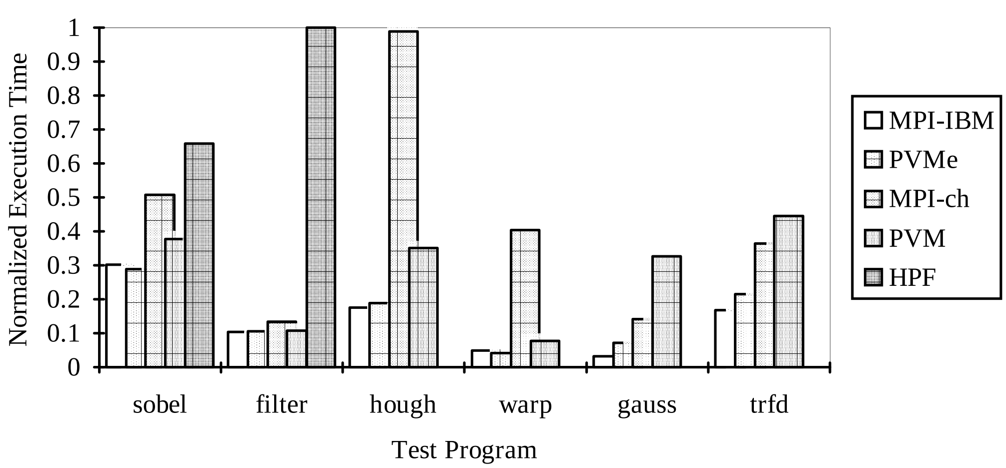

Figure 5. Language Performance on the IBM SP2

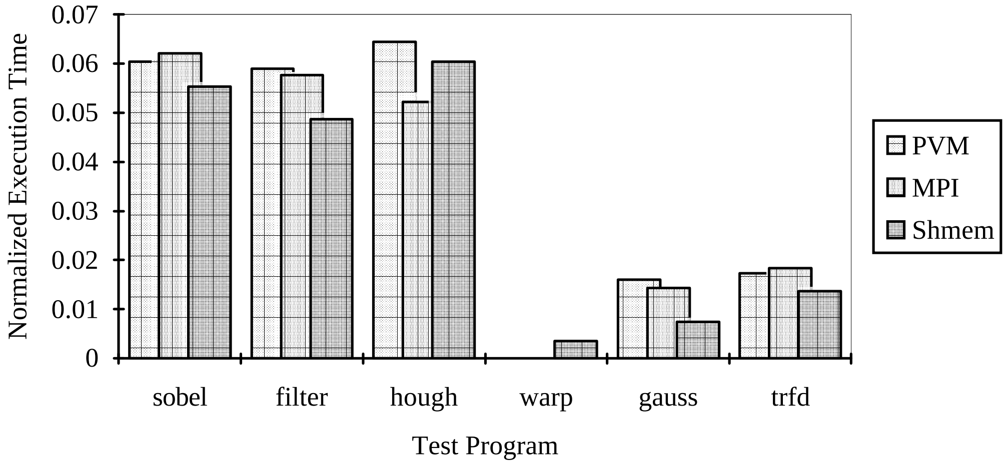

Figure 6. Language Performance on the Cray T3D

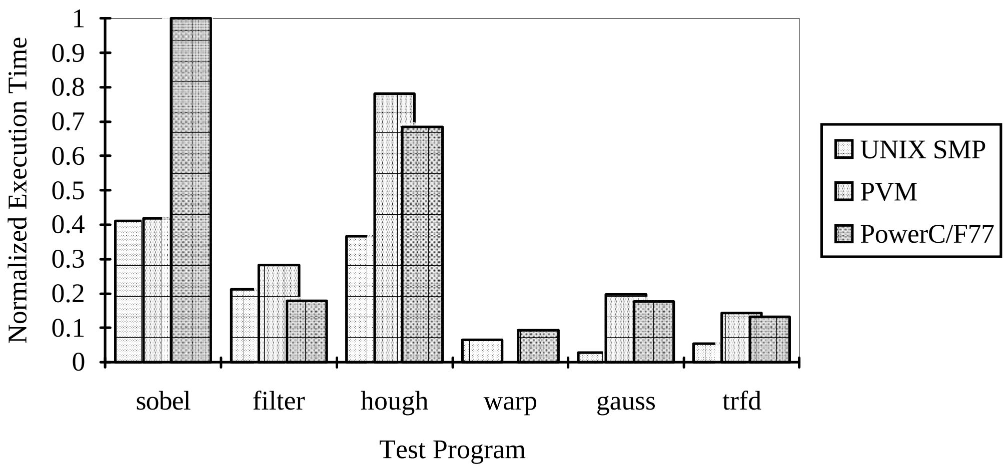

Figure 7. Language Performance on the SGI Challenge

Note that these results are application-specific. In *sobel* and *filter*, a small number (3 - 9) of medium sized messages (roughly 4-64KB each) are sent per node. The benefit of using a user-level protocol versus IP is quite slight compared to the benefit of using a faster network. This effect is even more pronounced in *warp* and *hough* where the number of messages is still relatively small but the messages sizes are large, ranging from 1 to 6MB on average. Here, the increased bandwidth of the high-speed network clearly plays the more dominate role in reducing run time. This is not the case with *gauss* and *trfd,* however. These applications both transfer a large number of messages which vary in size. Here, the software latency of sending and receiving many messages plays a more substantial role in overall program performance.

Although establishing specific factors that cause one language implementation to provide better performance than another is beyond the scope of this work, the above discussion demonstrates the potential for taking advantage of particular system features. Moreover, this process can clearly be aided by designers familiar with the target system.

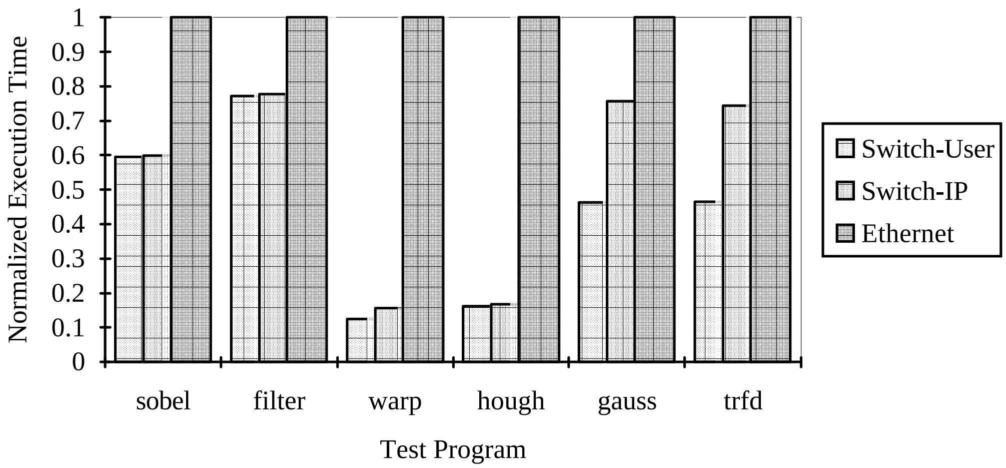

Figure 8. The Effects of Software Latency and Network bandwidth on the SP2

The remaining four languages were not available on multiple platforms as were the message-passing languages and therefore cannot be used for cross-architectural comparisons. These language results are included here, however, for intrasystem comparisons and for later reference.

The Shmem library results for the T3D shown in Figure 6 show a slight performance improvement than both MPI and PVM for all test programs but *hough*. The large reduction operations in *hough* may be more efficiently implemented by the MPI library function than its hand-coded Shmem equivalent. Note that PVM and MPI times were not available for the *warp* benchmark on the T3D due to lack of sufficient memory space. This was not the case for the Shmem library which adds comparatively little space overhead.

Figure 7 compares the performance of three language alternatives for the SGI Challenge SMP.

In general, the explicit shared-memory programming model produced the fastest executables, with the exception of *filter* which ran slightly faster using the PowerFortran compiler. PVM programs varied in execution time relative to the explicit shared-memory programs. In general, the PowerFortran and PowerC compilers performed well compared to their hand-coded counterparts. Unlike the other languages which used only mild (-O2) compiler optimizations, the PowerFortran and PowerC compilers were allowed to use more aggressive serial optimizations. This, in part, may account for the compilers' surprisingly good performance.

(***Note to reviewers*** *: Few of the HPF run times are reported here, as shown in Figure 5. Although the HPF compiler generated executables for the test programs, run times were often too high to be considered representative. It is hoped that the final release of the XL HPF compiler will generate executables with run times comparable to the other languages used in this study).*

4.2. Complexity Results

The normalized NCSS and MCC values for each of the languages used in this study are summarized in Figure 9 and Figure 10. In these figures, all parallel complexity values have been normalized relative to their equivalent serial program complexity values. Absolute complexity values for the serial versions of the test programs are provided in Table 2 as a reference. In this table, *LOC* represents lines of source code. This table shows complexity metric values for both the entire source code file and for the computational kernals. The kernel complexity values are used to calculate the scaled complexity values shown in Figures 9 and 10.

Note that we have distinguished between PVM for the cluster systems and the version of PVM used by the Cray T3D. MPI programs were identical across all the machines and are therefore

Table 2. Serial Source Code Metrics

Test Program	Total Source Code File			Kernel Source Code		
	LOC	NCSS	MCC	LOC	NCSS	MCC
sobel	360	130	25	67	49	13
filter	368	120	15	52	22	3
hough	403	161	38	110	87	26
warp	395	150	23	53	18	7
gauss	271	120	22	67	30	10
trfd	292	98	34	247	91	33

represented by a single value. Although not explicitly shown, note that the complexity of parallel programs generated by the SGI Power compilers will always normalize to unity, since these parallelizing compilers operate directly on the sequential source code.

4.2.1. Message-passing Languages

From Figure 9 and Figure 10, it is clear that programs written in a message-passing language tend to be substantially more complex than the equivalent sequential programs. Among the message-passing languages, normalized NCSS values range from 1.2 to 16 while normalized MCC values range from 1.1 to 20.

The message-passing versions of *filter* and *warp* show the most pronounced increase in complexity. The serial versions of these programs have relatively small computational kernels which iterate over the input image. The addition of message-passing code therefore adds a significant amount of complexity since simple memory references in the serial code must be replaced with several message `send`s and `receive`s with neighboring processors. *Warp* adds the additional complication of an irregular communication pattern that must be resolved at run time. As a result, it shows a greater increase in NCSS than *filter*. *Sobel* has communication requirements similar to *filter* but its computational kernel is larger. Since this large kernel is essentially the same in both the serial and message-passing versions, the complexity added by message-passing is less pronounced.

The *hough* benchmark does not show as significant an increase in complexity as *sobel*, *filter* and *warp*. *Hough's* predominate communication operation involves finding a set of maximum arrays values and the message-passing languages provide support for this type of global reduction. Given an array distributed across several processors, PVM and MPI are able to reduce these arrays to a single maximum, minimum, sum, etc. with a single function call. Similar reduction operations are to be found in *gauss*, but *gauss* also requires several broadcast and point-to-point messages that tend to offset this advantage of message-passing. The communication patterns in *gauss* do not involve multiple sends and receives on each node, however, and therefore add only a moderate amount of programming overhead.

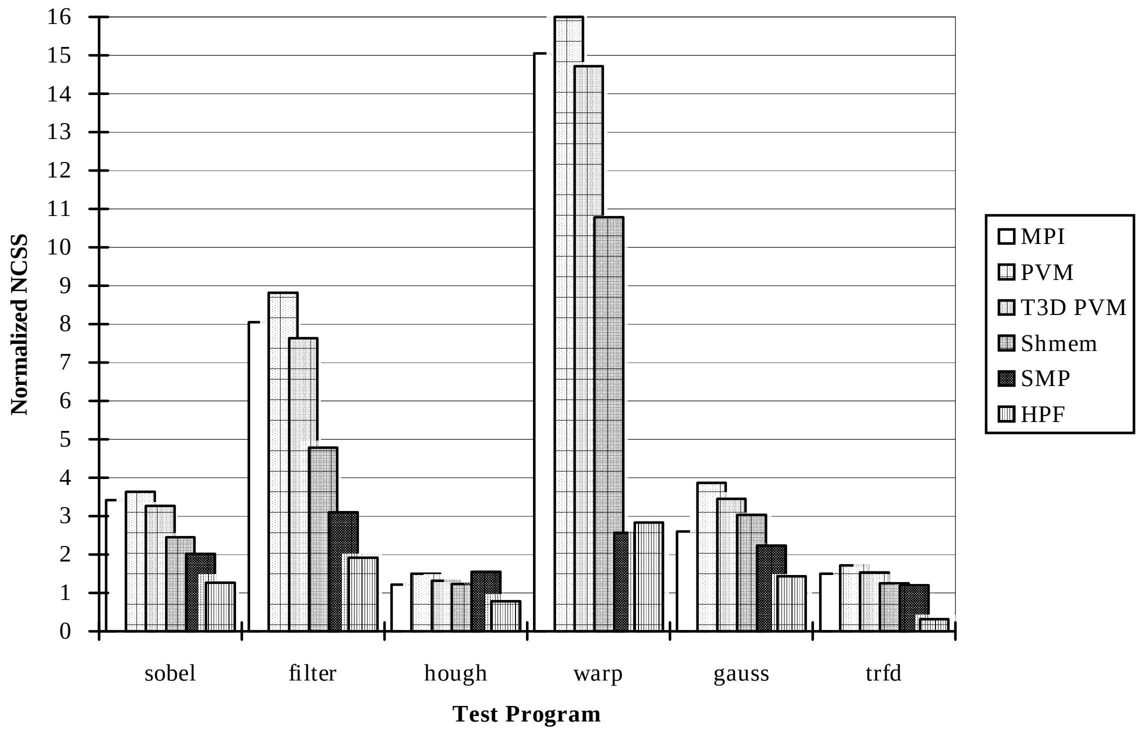

Figure 9. NCSS measurements for the different parallel languages

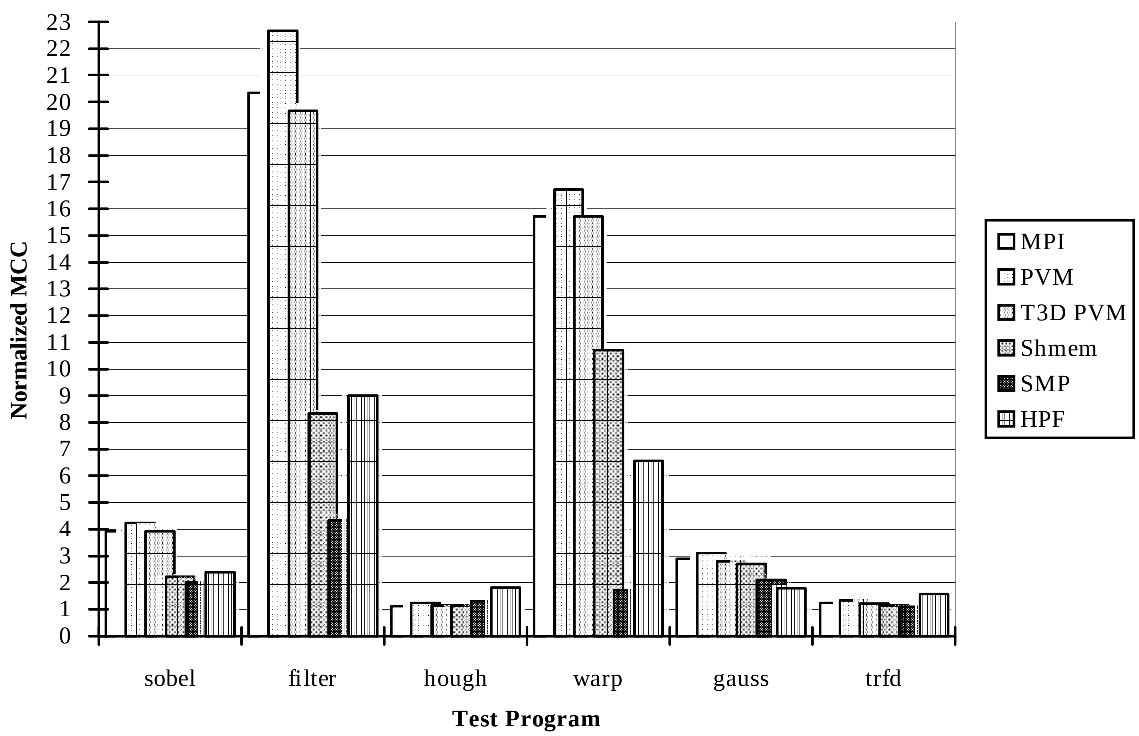

Figure 10. MCC measurements for five different parallel languages

Trfd shows a comparatively small increase in complexity despite being one of the more communication-intensive programs. There are two explanations for this apparent discrepancy. First, like *sobel*, the computational kernel of *trfd* is comparatively large. Secondly although *trfd* generates many messages at run time, these messages originate from only a few message-passing function calls which are called repeatedly within looping constructs.

Note that PVM programs generally show slightly higher complexity values than those of MPI programs. The source of this increased complexity is a result of several subtle differences in the languages. For example, the PVM programs required more data packing/unpacking operations and PVM provides less powerful reduction operations than MPI. The main difference in complexity between standard PVM programs and their Cray PVM equivalents is due to the former's method of spawning processes and assigning task identifiers. Both Cray PVM and MPI handle this these startup operations somewhat more elegantly than standard PVM.

4.2.2. Shared-memory Languages

The shared-memory programs tend to produce noticeably lower complexity values compared to their message-passing equivalents. Much of this difference can be attributed to the absence of data packing, unpacking and buffering operations which are an intrinsic part of message-passing languages but are not needed in the shared-memory languages. Instead, shared objects are referenced directly without the need to first copy them into and then out of buffer space.

Note that the programs written using the Cray Shmem library typically showed higher complexity values than the respective SMP programs. This difference is a consequence of the Shmem library's distributed view of memory. Although objects on different processors may share the same name, remote references to these objects must be made through Shmem library calls prefaced by a PE number in a fashion similar to message-passing. For example, if a one hundred element array, A, is distributed across processors P0 and P1, and P0 wishes to write into P1's portion of this array, P0's program would be similar the following code segment:

```
int A[50]; /* Half of the total array size */
int B[10]; /* Data to be written to A */

main( ){

      /* Write 10 ints from B into A on processor 1 */
      Shmem_put( B, A, 10, 1);
}
```

In fact, this code shares much in common with a `send` message-passing operation although no matching `receive` is required. The Shmem library's hybrid design is reflected in it's complexity values which typically fall between those of the message-passing languages and true shared-memory code.

The exception to this observation occurs in *hough* which actually produced slightly higher complexity ratings for the SMP version than the Shmem and the message-passing versions. This increase in the SMP complexity is a result of the reduction operations in *hough* for which message-passing languages have built-in support but is not part of the SMP library and therefore had to be hand-coded.

4.2.3. High-Performance Fortran

HPF programs do not explicitly specify process spawning, communication or synchronization. Consequently, they tend to require fewer source code statements to encode a given program than the other languages. In fact, *hough* and *trfd* written in HPF actually required fewer statements than the original serial versions. HPF produces low statement counts for these programs because they are easily described as sequences of matrix operations. A major strength of HPF is its support of high-level vector and matrix operations which are not found in ordinary serial programs. For example, a sequential element-wise addition of two matrices would be written in standard Fortran as

```
      do 20 i = 1, x
            do 10 j = 1, y
                  C(i,j) = A(i,j) + B(i,j)
10          continue
20    continue
```

However, the equivalent operation in HPF would require only a single assignment statement :

```
C(1:x, 1:y) = A(1:x, 1:y) + B(1:x, 1:y)
```

Turning to the MCC complexity results, we see that HPF does not fare as well as with the NCSS comparisons. The MCC ratings values for HPF tend to be high compared to the NCSS values for the same program because of HPF's "shorthand" `forall` statements notation for array operations. A single `forall` statement typically contains several predicates. For example, the following code fragment was taken from the HPF version of *warp*:

```
forall( i = 1:16; j = 1:16, k = 1:4)
      warp_coeff (k, j, i) = q(k,j) * q(1,i)
```

Note that this code fragment would produce an NCSS value of 2 (one `forall` statement plus one assignment statement). An equivalent sequential program would require three nested `do` loops and one assignment statement resulting in an NCSS value of 4. However, both versions produce MCC values of 3. Determining meaningful complexity values for a parallel language such as High-Performance Fortran is an open question. For this study, we have chosen to treat `forall` statements, such as that given above, as a single statements that produces an MCC value equal to its sequential equivalent.

4.3. Complexity Summary

With the exceptions noted above, porting a sequential program to a parallel language tends to add, often significantly, to the program's complexity. The amount of added complexity is language dependent. Message-passing languages tend to add the most programming overhead. Shared-memory languages and HPF add less complexity than message-passing. HPF produces very concise programs as evidenced by its low NCSS ratings, but may actually add more control complexity than SMP programming as indicated by its MCC values. The added source code complexity of parallel languages can largely be attributed to five main factors, to varying degrees depending of the language:

1. **Separate control paths for different "classes" of parallel threads -** Often, the code executed by a particular parallel thread depends on what role that thread is currently playing in the overall parallel computation. For example, recall that when doing Gaussian elimination using partial pivoting, the pivot row (or column) requires special processing. If the input matrix is partitioned among processors, the processor holding the pivot row will need to execute along a different control path than the other processors to handle the special processing of the pivot row. This type of code dichotomy is common in message-passing and shared-memory programs but has no analogue in HPF programs.

2. **Explicit exchange of data in using communications library calls -** Partitioning data among several processors usually necessitates communicating data between processors. The bulk of the message-passing libraries consists of functions which implement various types of interprocessor communications operations. The T3D Shmem library does not require both a sender and receiver and therefore adds less complexity in this regard. In addition to the communication call itself, several message-passing source code statements may be required to either pack data to be sent or to unpack received data. Using HPF, interprocessor communications operations are generated by the compiler and therefore hidden by the compiler.
3. **The need to spawn several parallel threads -** Process spawning is normally done explicitly in PVM and SMP programming, whereas MPI, Shmem and HPF programs spawn the appropriate number of processes automatically based on command line options or the run time environment.
4. **Data partitioning -** Data partitioning is done explicitly in HPF when data objects are declared. With MPI, PVM and Shmem, data are implicitly distributed. The global data objects are then programmer abstractions constructed by combining these distributed objects. Shared-memory does not require explicit data partitioning.
5. **Explicit synchronization library calls -** Although the passing of messages between processors often serves as a form of process synchronization, it is sometimes necessary to insert explicit synchronization calls into message-passing programs. Shared-memory languages do not have the inherent synchronization implied by the sending and receiving of data and therefore generally require more such operations than message-passing languages. HPF requires no explicit processes synchronization.

5. Conclusions

Clearly, message-passing languages dominate the current network-based parallel programming terrain. Standard message-passing languages, as exemplified by PVM and MPI, run on a variety of parallel platforms ranging from workstation clusters to large MPPs. Our survey suggests that neither language presents a clear performance advantage over the other. Of more importance than the choice between PVM and MPI is choosing an efficient implementation of either message-passing language for the given system. These implementations can vary widely in performance and stability across different platforms and across different implementations within a single platform.

Language implementations based on a thorough understanding of the underlying system are essential to good run time performance.

Compared to other possible alternatives, message-passing languages tend to produce more complex programs as measured by the NCSS and MCC software metrics. Although MPI programs typically produce less complex programs than PVM, the differences are small compared to the shared-memory or HPF programming styles. It is not yet clear that these alternatives can achieve the same level of performance as message-passing on distributed memory multiprocessors, however.

Distributed-shared architectures that support the shared-memory programming model are not yet available. Although virtual shared-memory software packages exist, these programming environments are usually built atop a message-passing language and, therefore, are not likely to offer any performance benefits. The Shmem library on the Cray T3D presents an interesting compromise between message-passing and shared-memory programming. This hybrid programming paradigm produced lower complexity ratings than PVM and MPI while producing shorter execution times. It is not clear, however, if such a language can be used outside of a shared-distributed architecture.

While the run times of programs generated with the beta HPF compiler used in this study were often disappointing, HPF compilers are only now becoming widely available. Consequently, final judgment should wait for the regular release versions of the compiler. Our complexity measurements show that HPF programs tended to be much more compact than other programming paradigms. However, the MCC complexity values suggest that shared-memory programming may actually produce programs with simpler control flow.

We conclude that the performance benefits and wide availability of PVM and MPI on network-based multiprocessors will continue to fuel their usage in the near-term despite the relatively high complexity of the programs they produce. Shared memory programming, using both an SMP model and the model adopted by the Shmem library, tend to produce less complex programs than message-passing but are likely to require architectural support to be made feasible for distributed systems. High-Performance Fortran is less architecturally dependent than the shared-memory languages and produces less complex programs than message-passing but currently lacks a performance benefit over message-passing. Thus, the current predominance of message-passing languages on network-based multiprocessors supports the conclusion that performance and complexity share an inverse relationship. Emerging trends in both parallel language design and multiprocessor architectures show promise in remediating this situation, however.

6. Acknowledgments

Support for this project was provided in part by Computing Devices International, Inc., National Science Foundation grant no. MIP-9221900, and a University of Minnesota McKnight Land-Grant Professorship. Steve VanderWiel was supported in part by an IBM Graduate Fellowship. Access to the machines used in this work was provided by Pittsburgh Supercomputer Center grant no. ASC950001P, the University of Minnesota-IBM Shared Research Project, National Science Foundation equipment grant no. CDA-9414015, and by the Army Research Office contract no. DAALO3-89-C-0038 with the University of Minnesota Army High Performance Computing Research Center (AHPCRC) and the DoD Shared Resource Center at the AHPCRC.

The authors would also like to acknowledge the programming efforts of Joshua Simer, Dawn Sweno and C.J. Chen.

7. References

1. Berge, C., *Graphs and Hypergraphs*, Amsterdam, The Netherlands:North-Holland, 1973.
2. Cray Research, Inc, *SHMEM Technical Note*, technical report SN-2516 2.3, Eagan, MN, 1994.
3. Ferretti, M. "The Generalized Hough Transform on Mesh-Connected Computers," *J. Parallel and Distributed Computing*, v19, 1993, pp. 51-57.
4. Geist, A., et al., "PVM 3 User's Guide and Reference Manual," Oak Ridge National Labs Tech. Report ORNL/TM-12187, 1993.
5. Grady, R. "Successfully Applying Software Metrics," *Computer*, v27, n9, 1994, pp. 18-26.
6. Grady, R., *Practical Software Metrics For Project Management*, Prentice Hall, Englewood Cliffs, NJ, 1992.
7. Gropp, W, E. Lusk and A. Skjellum, *Using MPI*, MIT Press, Cambridge, MA, 1994.
8. High Performance Fortran Forum, *High Performance Fortran Language Specification version 1.1*, technical report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, November, 1994.
9. Kai, L "Shared Virtual Memory on Loosely Coupled Multiprocessors," Ph.D. Thesis, Yale University, September 1986.
10. Lanning, D.L. and T.M. Khoshgoftaar, "Modeling the Relationship Between Source Code Complexity and Maintenance Difficulty," *Computer*, v27, n9, 1994, pp. 35-41.
11. Lenoski, D., et al. "The Standford DASH Multiprocessor," IEEE Computer, v23 n3, March 1992, pp. 63-79.
12. Lilja, D.J. and J. Schmitt, "A Data Parallel Implementation of the *TRFD* Program from the Perfect Benchmarks," *EUROSIM International Conference on Massively Parallel Processing Applications and Development*, Delft, The Netherlands, 1994, pp. 355-362.
13. McCabe, T.J, "A Complexity Measure," *Trans. on Software Eng.*, v13, n10, 1977, pp. 308-320.
14. Nitzberg, B. and V. Lo "Distributed Shared Memory : A Survey of Issues and Algorithms," *IEEE Computer*, v24 n8, August 1991, pp. 52-60.
15. Silicon Graphics Inc. *Fortran 77 Programmers Guide*. Mountain View CA, 1994.
16. Thinking Machines Corporation, *The Connection Machine CM-5 Technical Summary*, Cambridge MA, 1991.
17. Ward, W. "Software Defect Prevention Using McCabe's Complexity Metric," *Hewlett Packard J.*, v40, n2, 1989, pp. 64-69.
18. Wolberg, G. and T. Boult "Separable Image Warping with Spatial Lookup Tables," *Computer Graphics*, v.23 n.3, July 1989.