Complexity and Performance in Parallel Programming Languages

Steven P. VanderWiel* svw@ee.umn.edu

Daphna Nathanson† dnathans@iastate.edu

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

* Department of Electrical Engineering, University of Minnesota, Minneapolis, MN 55455, USA

† Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50010, USA

Abstract

Several parallel programming languages, libraries and environments have been developed to ease the task of writing programs for multiprocessors. Proponents of each approach often point out various language features that are designed to provide the programmer with a simple programming interface. However, virtually no data exists that quantitatively evaluates the relative ease of use of different parallel programming languages. The following paper borrows techniques from the software engineering field to quantify the complexity of three predominate programming models: shared memory, message passing and High-Performance Fortran. It is concluded that traditional software complexity metrics are effective indicators of the relative complexity of parallel programming languages. The impact of complexity on run-time performance is also discussed in the context of message-passing versus HPF on an IBM SP2.

1. Introduction

When large computational problems outgrow single processor computer architectures, users often must consider the use of multiprocessor systems. This decision is often made with hesitation, as multiprocessors have a well-earned reputation for being difficult to program. For large codes, the cost of parallel software development can easily surpass that of the hardware on which the code is intended to run. As a result, users will often choose a particular multiprocessor platform based not only on absolute performance but also the ease with which the multiprocessor may be programmed. Parallel computer designers have responded by providing support for the more popular programming models. For example, much of the impetus behind the design of distributed-shared memory (or NUMA) [11,15] architectures stems from a desire to provide programmers with a shared-memory programming model for a distributed memory architecture. Shared memory programming models are generally considered to be simpler than other alternatives such as message-passing.

Given the importance placed on designing languages which simplify the task of programming multiprocessors, suprisingly little empirical data exists that quantifies the relative complexity of different parallel programming models. Programmers often have an intuitive sense of the complexity of a language and, after extensive use, frequently 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, the following analysis borrows some techniques from the branch of software engineering known as *software metrics*. Cyclomatic program complexity and the number of source code statements are used to quantify the relative complexity of several parallel languages by measuring the complexity of a suite of parallel applications encoded in each of the languages. The relationship between complexity and performance is also discussed in the context of message-passing versus High-Performance Fortran on an IBM SP2.

2. Complexity Metrics

Software metrics are often used to give a quantitative indication of a program's complexity. Not to be

0-8186-7882-8/97 $10.00 © 1997 IEEE

confused with algorithmic complexity, or O( ) notation, software metrics have been found to be useful in reducing software maintenance costs by assigning a numeric value to reflect the ease or difficulty with which a program module may be understood. For example, it has been found that program modules with high complexity indices have a higher frequency of failures [10]. As a result, software engineers are able to easily 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 thereby increase overall program reliability.

Complexity metrics are used in this study to quantitatively compare the relative effort of using different parallel programming languages to encode a given algorithm. The effects of program modularity are therefore ignored and all programs are treated as though they are contained in a single module. It is also important to note that only user-level code is considered when determining complexity values. 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 an application.

Several complexity metrics exist [7], each designed to measure different factors affecting program complexity. Two popular metrics that are 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. The Maintainability Assessment System (MAS) [3] tool was used to calculate NCSS and MCC values for all programs except those written in High-Performance Fortran. Because MAS is designed to work with C source code files, complexity metrics for High-Performance Fortran programs were calculated using custom tools.

2.1. Non-Commented Source Code Statements

NCSS [6] is 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. 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 thought 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.

2.2. Cyclomatic Complexity

McCabe [14] introduced cyclomatic complexity 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 more easily understood 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 [14] for a full discussion of this parameter's role in calculating $V(G)$.

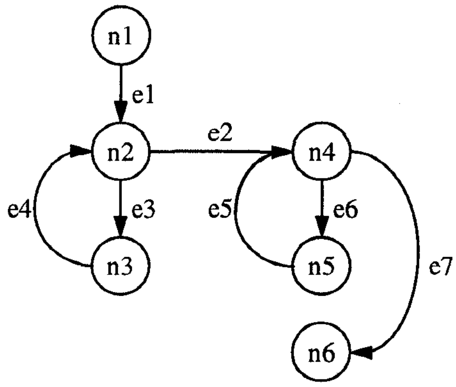
As an example of how cyclomatic complexity is applied, a sample program source code segment and its related control graph are given in Figure 1(a). This program segment's complexity would then be calcu-

```
/* reverse the string s */

len = strlen(s);

for( i = 0; i < len; i++)
     temp[i] = s[i];

for( j = 0; j < len; j++){
     i--;
     s[j] = temp[i];
}
```



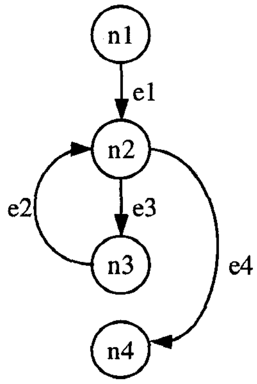
a) Program segment and related control-flow graph.

```
/* reverse s in place */

j = strlen(s);

for ( i = 0; i < j; i++){

     j--;
     c = s[i];
     s[i] = s[j];
     s[j] = c;
}
```



b) Simplified version of the above program segment.

Figure 1. Two versions of a string reversal program and their related control-flow graphs.

lated as $V(G) = e - n + 2 = 7 - 6 + 2 = 3$. A simplified version of the program segment is given in Figure 1(b). Here we condense the two `for` loops into one, thereby reducing the size of the control flow graph. 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 $V(G) = e - n + 2 = 4 - 4 + 2 = 2$.

In the context of parallel programs, control flow graphs can become extremely large if the control paths of all possible parallel threads of execution are charted. However, basing MCC measurements on such dynamic control graphs would be misleading since the separate paths followed by each parallel thread originate from a single source code file. Note also that most parallel programs are written to execute on a variable number of processors and defining the control flow graph in this way would produce MCC values that increased with the number processors used at run-time. As our interest lies in program complexity rather than execution complexity, our MCC measurements are derived from the parallel source code file in the same manner as sequential programs.

Although NCSS and MCC generally tend to be insensitive to programming style, the example given in Figure 1 demonstrates how algorithmic preferences can affect complexity measurements. To mitigate the effect of individual programmer style, algorithmic changes in different versions of a given program were allowed only when the changes were due to features of the programming language.

3. Experimental Environment

A suite of representative programs was used to compare the various languages presented in this study. Each test program was ported to several different parallel systems using languages available on each system. Although not an exhaustive collection, these test programs are felt to be representative of the types

Program	Description
sobel	Calculates magnitude and direction gradients of an input image. Image matrices are distributed in block-checkerboard fashion and communications result from the need to share boundary data among processors holding data from adjacent input image submatrices.
filter	Averaging (low-pass) image filtering program which calculates the value of an output image element as the weighted sum of up to 36 neighboring pixel elements in the input image. Data partitioning and communications are similar to those of *sobel*.
hough	As an instance of the more general Hough transform [4], this program 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	Spatial domain image restoration algorithm that aligns an input image along a given axis [18]. This program exhibits very irregular communications because picture elements may need to be shifted substantially depending on their proximity to the axis in question.
gauss	Gaussian elimination with back substitution. The rows of the input matrix are distributed in a cyclic pattern. This algorithm is very communication intensive, requiring several point-to-point, broadcast and reduction operations.
TRFD	A member of the Perfect Club benchmark suite which simulates a two-electron integral transformation using a fourth-order tensor equation [13]. The algorithm is a series of matrix multiplications and transpositions requiring several point-to-point communication operations.

Figure 2. Test program descriptions.

of parallel program kernels one might expect to find running on typical parallel processing 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 in Figure 2.

3.2. Programming Languages

The above applications were written using the five different parallel programming languages listed in Figure 3. Note that the word "language" is used rather loosely here. Often, parallel programs use ordinary C or Fortran compilers which then link with libraries that facilitate parallel operations, such as message-passing, synchronization or establishing shared-

Language	Description
PVM	A popular message-passing language developed at Oak Ridge National Labs [5]. Except where noted, PVM version 3.3 was used for all measurements.
MPI	A proposed standard for message-passing with many implementations [8]. Results are based on version 1.1 of the MPI standard.
SMP	Symmetric shared-memory multiprocessors explicitly fork parallel threads and allocate shared-memory arenas through which these threads may communicate data.
Shmem	The Cray T3D's Shmem library [2] allows the programmer to directly access a remote processor's memory through simple memory-to-memory copying (i.e. `get()` and `put()`). The library also includes simple group communications operations.
HPF	High-Performance Fortran [9] extends the semantics of ordinary serial programming languages to include vector and array operations. The programmer explicitly specifies data partitioning but allows the compiler to generate any necessary data communications.

Figure 3. Programming languages used in this study.

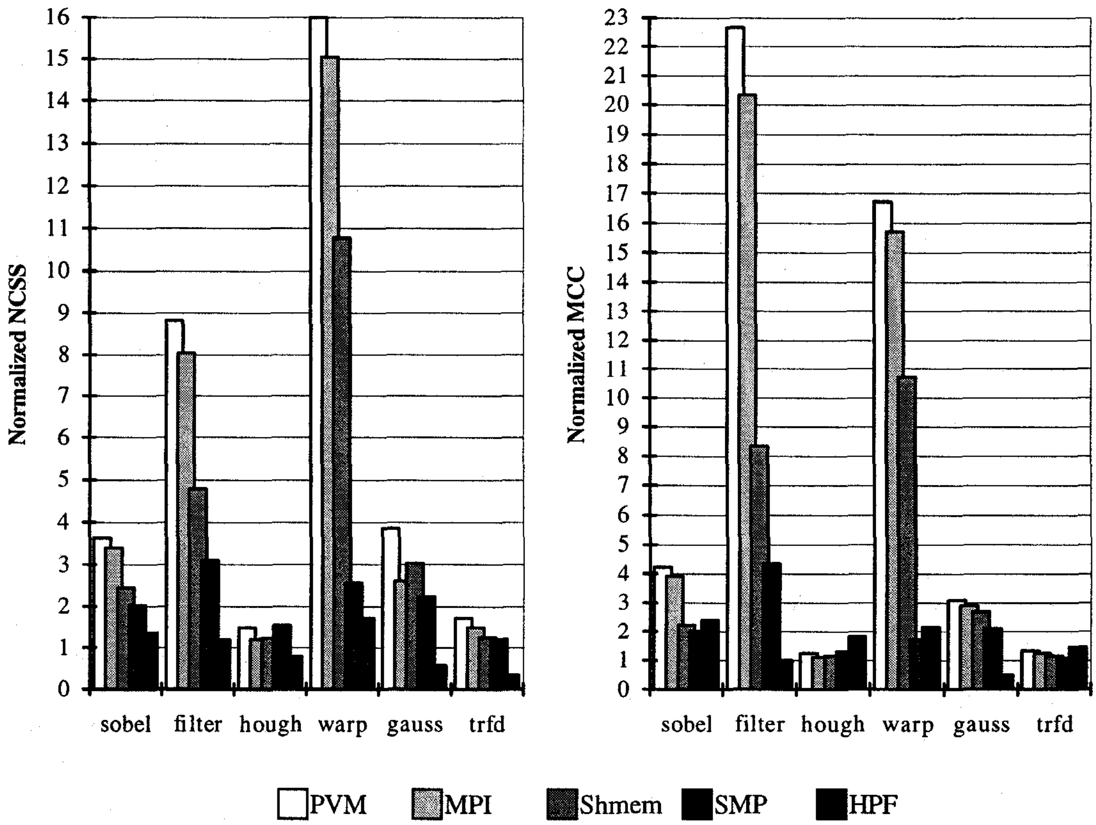


Figure 4. NCSS and MCC values for the test programs normalized to the equivalent sequential program.

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."

The languages used in this study can be grouped into three categories: message passing (PVM and MPI), shared memory (SMP and Shmem) and implicit parallel languages (HPF). With the exception of the Shmem libraries, each of these languages run on multiple hardware platforms. The Shmem library runs exclusively on the Cray Research T3D and T3E but is included here to represent possible alternatives to true shared memory programming models that may be found in shared virtual memory systems [12] or distributed shared memory multiprocessors.

4. Complexity Results

The normalized NCSS and MCC values for each of the languages used in this study are summarized in Figure 4. In these figures, all parallel complexity values have been normalized to their equivalent serial program values. In all cases, complexity metrics were applied only to the application kernels without regard for I/O or data initialization code.

4.1. Message-passing Languages

From Figure 4, it is clear that programs written in a message-passing language tend to be substantially more complex than the equivalent sequential programs, although complexity values varied widely across different test programs. 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 that 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 `sends` and `receives` with neighboring processors. *Warp* adds the additional complication of an irregular communication pattern that must be resolved at run time. *Sobel* has communication requirements similar to *filter* but its computational kernel is larger relative to its communication requirements. 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 operations are array reductions for which message-passing languages provide integrated support. That is, 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 required in *gauss,* but *gauss* also requires several broadcast and point-to-point messages that tend to offset this advantage of the message-passing languages.

The message passing versions of *TRFD* shows a comparatively small increase in complexity compared to the other programs 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 that are called repeatedly within looping constructs.

Note that PVM programs consistently produce slightly higher complexity values than those of MPI programs. 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. Also, MPI handles task spawning and the assignment of task IDs somewhat more elegantly than PVM.

4.2. Shared-memory Languages

The shared-memory programs generally 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 send and receive operations which are an intrinsic feature of message-passing languages but are not needed in the shared-memory languages. Instead, shared objects are referenced directly without the need to explicitly pass them between processes.

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];
int B[10];

main( ){
    ...
    if ( myid == 0)
        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 its 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 anomaly is a result of the many reduction operations in *hough* for which the Shmem library has built-in support but which must to be hand-coded when using an SMP programming model.

4.3. 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 parallel languages. In fact, *hough, gauss,* and *TRFD* written in HPF actually required fewer statements than the original serial versions. HPF produces low NCSS values 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 = A + B
```

Turning to the MCC complexity results, we see that HPF does not fare as well as with the NCSS comparisons. The MCC values for HPF tend to be high compared to the NCSS values for the same program because of HPF's "shorthand" `forall` 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.

Despite the simplicity of the HPF programming model, overhead is introduced by explicit data distribution directives and additional code used to reduce interprocessor communication. For example, HPF programs often make several local copies of remote data and additional user code is required to manage these copies. This additional programming effort tends to increase HPF programs' complexity ratings.

5. Performance Results

In an earlier study [16], we found that language complexity and program performance generally cannot be correlated. Rather than stemming from features inherent to a particular language, performance was often found to depend on application characteristics and language implementation issues. Some general observations concerning performance can be made, however. For the sake of brevity, we present performance data for a single machine, an 8-node IBM SP2. These results are representative of more general trends although they are constrained to the message passing languages and HPF since the shared memory style of programming is not supported on the SP2. The interested reader is referred to [16] for more complete performance data.

The performance results for the IBM SP2 are given in Figure 5. In this figure, performance has been normalized to the longest running version of a particular application. That is, each test program was encoded in all five languages and the language that produced the longest execution time was used to normalize the remaining four versions of the program. Program timings were taken during periods of dedicated machine use to avoid multitasking and network congestion effects. Several runs of each version of the application programs were measured and averaged to insure that the reported run times are not the result of

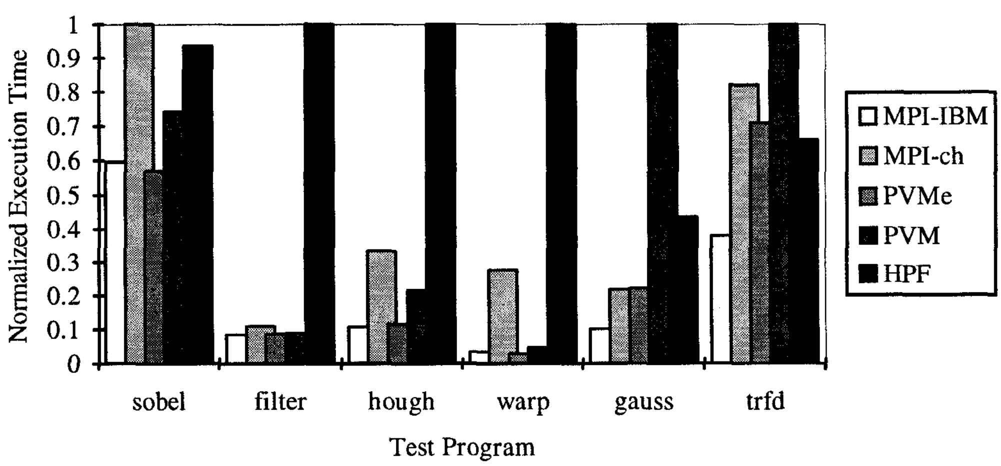


Figure 5. SP2 performance normalized to the slowest version of each test program.

temporary fluctuations in performance. In Figure 5, *MPI-IBM* and *PVMe* represent the vendor-supplied versions of the respective message-passing libraries while *MPI-ch* and *PVM* represent the public domain implementations.

Concentrating first on the public domain versions of the message passing languages, it is clear that neither PVM nor MPI offers a clear performance advantage. The PVM version of *warp*, for example, ran considerably faster than the MPI version, but this situation is reversed for *gauss*. In general, what is most notable about the relative performance of public-domain versions of PVM and MPI programs is the unpredictability of the resulting run times. This is not the case for the commercial implementations, however. As Figure 5 shows, the relative performance differences between PVM and MPI are comparatively slight for the commercial implementations. Moreover, the performance of the commercial implementations tends to be better than that of the corresponding public domain implementations.

HPF program performance varied considerably across the six application programs. The HPF versions of *sobel*, *gauss* and *TRFD* yielded run times comparable to those of their message-passing equivalents while *filter*, *hough* and *warp* ran several times slower than the fastest message-passing version. HPF tends to perform best when an application can be easily decomposed into a series of vector and array operations and *gauss* and *TRFD* conform to this style of programming. *Sobel*, although less regular than *gauss* and *TRFD*, is easily parallelized due to the comparatively simple operation it applies to its input matrices. The remaining programs require more irregular data communications which are not as easily handled by the HPF compiler and therefore produce slower executables.

6. Conclusions

Because complexity metrics are designed to reflect a programmer's intuitive sense of a program's relative clarity, the confirmation of complexity results can be somewhat subjective. In large software development efforts where program maintenance is tracked, these metrics have been substantiated by demonstrating their correlation with bug reports and maintenance costs. In the absence of such maintenance data for the test programs used here, it has been assumed that these metrics can be extended to parallel programs. Although the final word regarding the validity of this assumption will require further work, the complexity results given in Figure 4 match the authors' experience using these languages and can be understood in terms of the features of each language.

Overall, message-passing languages tend to produce the highest complexity values, with PVM showing slightly higher values than MPI. Shared-memory programs show lower complexity values than message-passing languages and within this category, SMP programs usually produced lower NCSS and MCC values than those written using the Shmem library. HPF programs are often very concise as evidenced by their low NCSS ratings, but showed no clear advantage over SMP programs based on MCC ratings. In general, the complexity added to programs by using a parallel language can be largely attributed to five main factors :

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, 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.** 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 an explicit send and receive 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 communication operations are generated by the compiler and are therefore hidden from the programmer.

3. **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. In addition, all of these

languages often need to create local copies of remote data to minimize data communications overhead. In contrast, true shared-memory programming does not require data partitioning nor does it require the programmer to differentiate between local and remote memory.

4. **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.

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 process synchronization.

Although programming complexity is a major concern for application developers, performance is ultimately the reason for employing a multiprocessor. If a language compiler does not produce executables which take advantage of the underlying architecture, little is gained by moving to a parallel hardware platform. For the message-passing languages MPI and PVM, the results of Section 5 and earlier studies [16] suggest that neither language offers 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. In general, language implementations which are based on a thorough understanding of the underlying system are essential to good performance. Vendor-supplied implementations of these languages, such as those found on the SP2, tend to perform better for this reason than public domain implementations.

HPF programs on the SP2 varied dramatically in performance compared to the message-passing languages. Although the HPF programs never produced faster executables than the fastest message passing language for a given program, *sobel*, *gauss* and *TRFD* demonstrated comparable performance. The remaining test programs ran several times slower when compiled from HPF sources. It is assumed that some portion of this performance gap can be attributed to the relative maturity of the message-passing libraries when compared to the newer HPF compiler. However, given adequate implementations of both types of languages, it seems likely that programmers willing to code at a level close to the underlying parallel hardware will continue to produce faster executables than a compiler translating high-level code.

When restricted to message-passing languages and HPF, parallel language complexity and performance currently seem to share an inverse relationship much like that which exists between assembly language programming and higher-level sequential programming languages. However, while the performance difference between hand-coded assembly and that generated by, for example, a C compiler is generally considered small enough to warrant the use of C, this is not yet the case for parallel languages. Although some of the SP2 results give cause for optimism, it is not clear that the relative simplicity and hardware independence of HPF programs will justify the corresponding loss in performance.

Shared memory languages and systems may offer an attractive compromise. These languages produce complexity values that typically fall between message-passing and HPF and yet are not far removed from the hardware model on which they run. While the use of this programming paradigm has traditionally been restricted to SMPs with a limited number of processors, the advent of commercial distributed-shared multiprocessors [15] allows a shared memory style of programming to be used on larger systems.

7. 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 and National Science Foundation equipment grant no. CDA-9414015. The authors would like to acknowledge the programming efforts of Dawn Sweno, C.J. Chen and Joshua Simer and the HPF assistance provided by Jonathan Harris and Shin Lee of Digital Equipment Corporation.

8. 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. Day, D., P. Felzien, D. Sims, M. Vieselmeyer, and R. Wynn, *Maintainability Assessment System User Manual*, Department of Computer Science, University of Idaho, 1991.

4. Ferretti, M. "The Generalized Hough Transform on Mesh-Connected Computers," *J. Parallel and Distributed Computing*, v19, 1993, pp. 51-57.

5. Geist, A., et al., "PVM 3 User's Guide and Reference Manual," Oak Ridge National Labs Tech. Report ORNL/TM-12187, 1993.

6. Grady, R. "Successfully Applying Software Metrics," *Computer*, v27, n9, 1994, pp. 18-26.

7. Grady, R., *Practical Software Metrics For Project Management*, Prentice Hall, Englewood Cliffs, NJ, 1992.

8. Gropp, W, E. Lusk and A. Skjellum, *Using MPI*, MIT Press, Cambridge, MA, 1994.

9. 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.

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 Stanford DASH Multiprocessor," IEEE Computer, v23 n3, March 1992, pp. 63-79.

12. Li, K. "IVY: A Shared Virtual Memory System for Parallel Computing," *Proc. International Conference on Parallel Processing*, University Park, PA, 1988, v. 2, pp. 94-101.

13. 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.

14. McCabe, T.J, "A Complexity Measure," *Trans. on Software Eng.*, v13, n10, 1977, pp. 308-320.

15. Silicon Graphics, Inc., "Origin Servers: Technology of the Origin Family," *http://www.sgi.com Products/hardware/servers/technology/overview.html*

16. VanderWiel, S, D. Nathanson and D. Lilja, "Searching for the Holy Grail: Performance, Complexity and Portability of Single-Program, Multiple Data Parallel Programs," University of Minnesota High-Performance Parallel Computing Research Group Technical Report HPPC-96-02, March 1996.

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.