Final Research Report for Grant NAG-1-995

A Design Methodology for Portable Software on Parallel Computers

Nicol and Keith W. Miller and Dan A. Chrisman *
Department of Computer Science, College of William and Mary

1 Introduction

This final report for research that was supported by grant number NAG-1-995 documents our progress in addressing two difficulties in parallel programming. The first difficulty is developing software that will execute quickly on a parallel computer. The second difficulty is transporting software between dissimilar parallel computers. In general, we expect that more hardware-specific information will be included in software designs for parallel computers than in designs for sequential computers. This inclusion is an instance of portability being sacrificed for high performance. New parallel computers are being introduced frequently. Trying to keep one's software on the current high performance hardware, a software developer almost continually faces yet another expensive software transportation. The problem of the proposed research is to create a design methodology that helps designers to more precisely control both portability and hardware-specific programming details. The proposed research emphasizes programming for scientific applications.

We completed our study of the parallelizability of a subsystem of the NASA Earth Radiation Budget Experiment (ERBE) data processing system. This work is summarized in section two. A more detailed description is provided in Appendix A ('Programming Practices to Support Eventual Parallelism').

Mr. Chrisman, a graduate student, wrote and successfully defended a Ph.D. dissertation proposal which describes our research associated with the issues of software portability and high performance. The list of research tasks are specified in the proposal. The proposal 'A Design Methodology for Portable Software on Parallel Computers' is summarized in section three and is provided in its entirety in Appendix B.

We are currently studying a proposed subsystem of the NASA Clouds and the Earth's Radiant Energy System (CERES) data processing system. This software is the proof-of-concept for the Ph.D. dissertation. We have implemented and measured the performance of a portion of this subsystem on the Intel iPSC/2 parallel computer. These results are provided in section four.

Our future work is summarized in section five, our acknowledgements are stated in section six, and references for published papers associated with NAG-1-995 are provided in section seven.

2 The Parallelism of the ERBE data processing system

In the first two years of our support from NASA grant NAG-1-995, we examined issues that will arise in the application of parallel processing to atmospheric research projects such as ERBE and CERES. Our work to date has had a dual focus: the importance of parallel architectures and algorithms for processing the atmospheric science data; and quality assurance for an atmospheric science data processing system. We have acted as an information resource concerning software engineering techniques and tools for the CERES project.

The nature of the ERBE and CERES projects has led us to important unsolved research questions concerning the use of software engineering principles in a parallel processing environment. This research area includes formal models for developing large software projects implemented on parallel architectures and the design of automated tools to aid in that development.

We extensively analyzed and rewrote sections of the ERBE data processing system as we transported it from the Control Data Corporation Cyber Series computers that used a Fortran V compiler to an Intel iPSC/2 hypercube computer and its Fortran 77 compiler. This programming effort elucidated several code characteristics that are vital in planning the CERES software development: data dependencies and extensive, well maintained documentation at all levels of the data processing system. The size of the CERES project and the amount of design, implementation, and maintenance documentation required, suggests that Computer Assisted Software Engineering (CASE) tools will be useful for CERES. Our presentation to NASA personnel involved in the CERES project gave us an opportunity to discuss our findings about the ERBE data processing system with the developers. A detailed description of the parallelization study is provided in Appendix A.

3 Design methodology proposal

Upon completion of our parallelizability analysis of the ERBE data processing, we began a detailed literature survey concerning current methodologies for designing parallel software. This survey formed the basis for Mr. Chrisman's Ph.D. dissertation proposal. He proposed a design methodology for portable software on parallel computers. This dissertation proposal was successfully defended. A brief discussion of the proposal in provided below and the dissertation proposal is provided in Appendix B.

Developing software for a parallel computer and transporting the software to a dissimilar parallel computer are two difficulties which a scientific parallel software developer faces. These two difficulties can be illustrated as developing software under two scenarios:

1. a problem's initial implementation on a parallel computer and

2. a problem's subsequent implementations on dissimilar parallel computers.

There are likely several motivations for transporting the software from a parallel computer to a dissimilar parallel computer. These motivations are based on the performance capabilities of the parallel computers. Published performance results for benchmarks executed on the dissimilar parallel computer is a likely enticement for transporting the software. The programmer considers these results when deciding to gain access to a parallel computer which will execute the software faster or execute the software for larger problems. The underlying parallel architecture and the software environment(i.e. operating system, compilers, CASE tools) are two dissimilarities between parallel computers that will both lead to the enhanced performance and for which the programmer will have to consider in detail when transporting the software.

The proposed methodology directs the programmer to construct a software design which is specific to a parallel computer. If the programmer is unsatisfied with the initial design, then the methodology prescribes actions which can improve the design. When the programmer must transport the software to a new dissimilar parallel computer, then the methodology directs the programmer to modify the software design, creating an equivalent design which is specific to the dissimilar parallel computer.

The components and attributes of the proposed parallel software design, the definition of the proposed equivalence-preserving design transformer, and the proposed method of constructing a design are described in section four of the dissertation proposal.

4 First Implementation on a Parallel Computer

We have met the first milestone concerning our parallel software research associated with the NASA CERES project (an integral part of the NASA Earth Observing System). We studied the available scientific documentation and preliminary software specifications for the compute-intensive CERES cloud retrieval algorithm, which defines one subsystem of the CERES data processing system. Then, during September 1992, the cloud retrieval algorithm group (Dr. Bryan Baum and Dr. Bruce Wielicki (project leader) at NASA Langley Research Center) provided their newest version of the still-developing research software that implements some of the cloud retrieval algorithm. We modified this software to process satellite data similar to the ERBE data processing system. Then we modified this software to execute efficiently on a parallel computer. The performance measurements of these executions are provided.

4.1 NASA Application

The proposed CERES data processing system will process the level 0 data of the CERES scanner and a satellite-specific imaging radiometer (e.g. MODerate-resolution Imaging Spectrometer (MODIS)) to produce the levels 1A, 1B, 2, and 3 data products. The long life of the CERES project (i.e. fifteen to twenty year science mission and twenty five year computational mission) nearly guarantees that the CERES software will make at least one and probably several migrations to new hardware platforms; thus, portability becomes important. A time constraint placed upon the data processing system is that twenty four hours of CERES sensor measurements must be processed within seventy two hours of the receipt of the data; thus high performance must be considered in tandem with portability. The estimated size of the CERES software is one million lines of FORTRAN, Ada, C, and UNIX. The scalability of the costs to transport the CERES software becomes crucial for a software system of this size and lingual complexity.

We are focusing upon the NASA CERES cloud retrieval algorithm [1] which computes cloud properties (e.g. short wave optical depth, window emissivity, mean droplet radius, liquid water path, cloud-top pressure, and fractional cloud cover) within a CERES scanner pixel. This algorithm comprises the cloud retrieval subsystem of the CERES data processing system. The algorithm is composed of seven methods for computing one or more cloud properties; however, the appropriate application of a method is constrained by the properties of the CERES scanner pixel being processed (e.g. geographic type, solar contribution to view, weather type).

Due to the spatial and temporal variability of clouds in the atmosphere, high resolution data that is sampled frequently leads to massive input data sets. The data will be processed in one-hour chunks and utilize the following primary data sets. The cloud-imaging MODIS instrument, which is planned on the Earth Observing System platforms, produces fifteen gigabytes of about one kilometer ground resolution measurements during twenty four hours. Several earth-gridded input data sets will be required. A earth geography map with a ten kilometer spatial resolution will provide surface-type characteristics (e.g. vegetation classification, surface elevation, and snow cover). This map will be refreshed once a day. A meteorology map with a resolution of fifty kilometers will provide macroscopic cloud information (e.g. storm systems). This map is refreshed every three hours. An atmospheric temperature and humidity three-dimensional map with one degree latitude and longitude resolution and about forty levels is desired. This description excludes many secondary data sources (e.g. the algorithmic-specific coefficient tables).

The estimated size of the cloud retrieval software that has been proposed by the CERES data management team and the cloud retrieval working group is forty thousand lines of Fortran 77, C, Ada, and UNIX.

The research code which is a precursor to the CERES cloud retrieval subsystem performs the carbon dioxide (CO_2) slicing technique. CO_2 slicing computes the ratio between the High Resolution Infrared Radiation Sounder (HIRS/2) satellite measurements of radiation and theoretically-computed radiation values. This ratio is used to compute the height or partial pressure of the clouds that fall within a pixel or a sounder footprint. The version of the software that we received is 1,483 lines of Fortran 77 (688 lines of code and 795 comments) and less than 20 lines of UNIX. Several input files were also provided. The radiosonde input data file was 2.5 kilobytes and the HIRS/2 measurements input data file was 130 kilobytes (31 scans). These measurements were taken on April 12, 1991 and coincident with the over-flight of the NOAA-11 satellite. The HIRS raw data conversion coefficients input file was 121 kilobytes. The program executes on a Sun Microsystems, Inc. Sun4c (SPARCstation 1) computer with SunOS version 4.1.1 1 and Sun Fortran compiler (V1.4, Patch Release 5, 10 Feb 1992).

4.2 Modification to the sequential research code

We made several modifications to the research code. The original version was restricted to analyzing only specific regions of the Earth, relying on data files containing time-specific and geographic-specific radiosonde and HIRS/2 sensor measurements. We have enhanced their software to analyze satellite measurements viewing any region on the Earth by replacing the radiosonde files with the European Center for Medium Range Weather Forecasting global three-dimensional atmospheric maps. The map access required incorporating 1155 lines of Fortran 77 (366 lines of code and 789 comments) and 118 lines of C (99 lines of code and 19 comments) provided by NASA. We also enhanced the software concerning the HIRS/2 sensor data. The original version was limited to analyzing fifty scan lines (approximately 5.3 minutes) of HIRS/2 data stored in ASCII format. With the enhanced version, the only constraint upon the number of scan lines that can be analyzed is the maximum file size for the computer's file system for files in binary format. Current HIRS/2 input files provided by NASA are approximately 4.5 megabytes (112.6 minutes of satellite measurements which require about an orbit and a half to measure). Enhancing the sensor data format and length of analysis required incorporating 384 lines of Fortran 77 (161 lines of code and 223 comments) and 156 lines of C (93 lines of code and 63 comments) provided by NASA.

4.3 Performance of the sequential research code

The resulting version of the software is the sequential version and, with all enhancements, is composed of 4645 lines of Fortran 77 (1368 lines of code and 3277 comments) and 274 lines of C (192 lines of code and 82 comments). For the performance data in this report, the standard test case is based on processing 128 scans of HIRS/2 data (approximately 13.6 minutes of sensor measurements). Similar to the strategy of the ERBE data processing system, the output of our software is the time-stamped latitude-longitude-located cloud height computed for each pixel of each scan in a binary file.

We copied the sequential version of the software and input data files to the Intel iPSC/2 parallel computer hard disks. The byte ordering scheme is different between the Intel 80386 processor and the Sun Microsystems Sparc processor. Swapping the bytes for each word of the binary input data

required incorporating 130 lines of C (72 lines of code and 58 comments) provided by Tom Crockett of ICASE at NASA-Langley.

We measured the performance of the software on the Intel 80386-based System Resource Manager (SRM) which is the front-end processor of the Intel iPSC/2 parallel computer. For the Intel Corporation i386 SRM computer with AT&T UNIX System 5 (release 3.2, version 2.1) and Greenhills Fortran-386 and C-386 compilers (driver 8.5), the execution time is 127.9 minutes. We also measured the execution times of the individual subroutines which comprise the software.

4.4 Parallelizing the Software

We determined that there are multiple levels of parallelism in the CO_2 slicing cloud retrieval algorithm. We completed a data dependency analysis of our version of the sequential software, a necessary step toward distributing the HIRS satellite measurements and global maps among the processors of a parallel computer. Since the software could run on a sequential computer, we measured its execution time when executed on one processor node of the sixteen-node parallel computer. For one Intel Corporation iPSC/2 80386-based CX computing node with NX/2 operating system and the node-switch compiler option for the same compilers as the SRM, the execution time is 96.7 minutes. We also measured the execution times of the individual subroutines for the one processor case. With the execution time measurements and data dependency analysis, we designed a parallel version of the software to efficiently utilize the sixteen processors of the iPSC/2 in parallel. This design was implemented and execution times for the parallel version using multiple processors of the iPSC/2 were measured. These results are provided in the next section.

4.5 Performance on the Intel iPSC/2

In the table, the number of processors is the quantity of Intel iPSC/2 processing nodes participating in the processing of the atmospheric data. The execution time is the CPU time required to process the data. The speedup is the ratio of the fastest time to process the data on one processing node divided by the the fastest time to process the data on "number of processors" processing nodes. The efficiency is the ratio of the speedup for "number of processors" divided by "number of processors".

Number of processors	Execution Time(seconds)	Speedup	Efficiency
1	5803.4	—	—
2	2951.7	1.97	0.98
4	1507.6	3.85	0.96
8	781.5	7.43	0.93
16	419.0	13.85	0.87

These results are encouraging. Because the parallelism of the problem is more than two orders of magnitude greater than the maximum number of processors used in our parallel performance studies, then a reasonably efficient execution using more parallel processors(i.e. 100 or 1000) is feasible. Our software does not execute with a speedup of 16 and an efficiency of one for sixteen processors because of the data file input and output system on the parallel computer and work imbalance between processors. Reading and writing data files is performed sequentially on the iPSC/2 and this cost is difficult to overlap with the computations of the software. Also, the execution time to process a scan is scan-specific and the resulting work imbalance requires the group of processors to wait for the processor with the most work to complete. Work imbalance

techniques that are executionally inexpensive and algorithmic simple are employed to more evenly balance the work between the processors.

5 Current Work

We met the first milestone which is to design and implement a parallel version of the software. Toward the second milestone (i.e. redesigning and re-implementing versions of the software for other parallel computers), we are currently redesigning and implementing the CO_2 slicing algorithm on the MasPar MP-2 parallel computer. The programming model for this computer is called Single-Instruction-Multiple-Data. This programming model is significantly different from the Multiple-Instruction-Multiple-Data programming model for the Intel iPSC/2. To efficiently utilize the SIMD programming model, we are currently converting our CO_2 slicing program from Fortran 77 to Fortran 90.

6 Acknowledgements

We acknowledge the contributions provided by the following people:

- Ed Howerton and Chris Harris for their ERBE Telemetry subsystem expertise during our ERBE parallelizability study,
- Carol Tolson for access to the most up-to-date CERES data processing system documentation,
- Fran McLemore for her CDC computer execution expertise during our ERBE subsystem portability study,
- Eric Schmidt and Dave Doelling for their cloud retrieval algorithm expertise during our design feasibility assessment for parallelizing the cloud retrieval subsystem,
- Bryan Baum and Jay Titlow for their algorithm expertise and software contributions during the implementation and testing of our enhanced version of the cloud retrieval software,
- Scott Nolf for the software which accesses the European Center for Medium Range Weather Forecasting global three-dimensional atmospheric maps, and
- Tom Crockett for the byte swapping software.

7 Publications

The type of research conducting under NAG-1-995 is large in scope and not of a type that publications on intermediate results are widely accepted by the computer science community. Consequently we cannot report on the acceptance of any publications that directly address the issues currently being investigated. Nevertheless, NAG-1-995 provided support for the Intel iPSC/2, a machine upon which we were able to solve and write about a number of smaller scale problems. The following list reports the papers we have published based on use of the iPSC/2, and which cite support of NAG-1-995. All papers are authored by David Nicol, other co-authors are noted (with lead authors highlighted in boldface).

1. "Rectilinear Partitioning of Irregular Data Parallel Computations", *Journal of Parallel and Distributed Computing*, to appear.
2. "Noncommital Barrier Synchronization", *Parallel Computing*, to appear.
3. "A Sweep Algorithm for Massively Parallel Simulation of Circuit-Switched Networks", with **Bruno Gaujal** and Albert Greenberg, *Journal of Parallel and Distributed Computing*, to appear.
4. "Optimistic Parallel Simulation of Markov Chains Using Uniformization", with Phil Heidelberger, *Journal of Parallel and Distributed Computing*, to appear.
5. "Parallel Simulation Today", with Richard Fujimoto, *Annals of Operations Research*, to appear.
6. "Conservative Parallel Simulation of Markov Chains Using Uniformization", with **Phil Heidelberger**, *IEEE Trans. on Parallel and Distributed Systems*, to appear.
7. "Inflated Speedups in Parallel Simulations via `malloc()`", *Int'l Journal on Simulation*, vol 2, Dec. 1992, 413-426.
8. "Parallel Simulation of Markovian Queueing Networks Using Adaptive Uniformization", with Phil Heidelberger, *1993 SIGMETRICS Conference*, Santa Clara, CA., pp. 135–145.
9. "Parallel Algorithms for Simulating Continuous Time Markov Chains", with Phil Heidelberger, *1993 Workshop on Parallel and Distributed Simulation*, San Diego, CA., pp. 11–18.
10. "Optimistic Global Synchronization for Parallel Discrete-Event Simulations", *1993 Workshop on Parallel and Distributed Simulation*, San Diego, CA., pp. 27–34.
11. "REST: A Parallelized Reliability Estimation System", with Adam Rifkin and Dan Palumbo, *1993 Reliability and Maintainability Symposium*, Atlanta, GA, pp. 436-442.
12. "MIMD Parallel Simulation of Circuit Switched Communication Networks", with Albert Greenberg, Boris Lubachevsky, *Proceedings of the 1992 Winter Simulation Conference*, 629-636.
13. "State of the Art in Parallel Simulation", with Richard Fujimoto, *Proceedings of the 1992 Winter Simulation Conference*, pp. 246-254.

A Report from ERBE Parallelizability Study

Based on our study of the parallelizability of a subsystem of the NASA ERBE data processing system, this paper provides some approaches to facilitate the transportation of sequential code to a parallel computer, particularly for a large software project like the CERES data processing system.

PROGRAMMING PRACTICES TO SUPPORT EVENTUAL PARALLELISM

Keith Miller, David Nicol, and Dan Chrisman*
Dept. of Computer Science, The College of William and Mary
Williamsburg, Virginia

January 10, 1990

1 Introduction

Most existing programs were written for sequential machines. As parallel architectures become commonplace, programmers will face the task of porting applications from sequential machines to parallel machines. For some programming languages and for some specific machine environments, sophisticated compilers will simplify this porting considerably [Padua and Wolfe]. However, even with a parallelizing compiler, the porting will probably require some handcrafting of the code. The importance of customizing is emphasized when the code to be ported is lengthy and when the resulting parallelized code must be machine efficient. Since machine efficiency is often the motivation for moving to a parallel architecture, we expect that time and space considerations will typically be important.

We are porting about 100,000 lines of source code for satellite data processing from a CDC Cyber 860 to an Intel iPSC/2 hyperCube. The complexities of this porting have led us to consider the characteristics of sequential code that aid and characteristics that complicate the task of porting sequential code to a parallel machine. In this paper we share some of the problems

*Supported by grant #NAG-1-995 from NASA Langley Research Center.

we encountered and the approaches we used to solve those problems. Then we generalize what we have learned into suggestions for writing sequential code to facilitate eventual conversion to parallel code. Although we state these suggestions in terms of new code, the same suggestions can be adapted to maintaining sequential code.

Our current project is converting FORTRAN programs, but the suggestions below should apply equally well to any imperative programming language. Different languages will provide more or less automated support to implement the suggestions; however, programmers in all languages can facilitate future parallelization. We try to avoid suggestions that require inordinate runtime penalties of time and of memory. We also do not advocate practices that will dramatically increase development time (for example, rewriting a program in a new language). We hope to offer practical suggestions for modest changes in programming practice which have a good chance of easing the transition from sequential to parallel programs.

2 Our Conversion Project

NASA's Earth Radiation Budget Experiment (ERBE) analyzes environmental data measured from three space platforms. Researchers at NASA Langley Research Center process the ERBE data from each platform by performing conversions and analyses, validating the data and results, and producing various reports and archival data analysis products for the scientific community. The programs to accomplish these services execute on a CDC Cyber 860 running under CDC NOS 2.6. The Cyber has a sequential architecture with 60 bit words.

To study potential improvements in the turnaround time on current science report products and to prepare for a 10 fold expansion of data input and output within the next two decades, the ERBE data management team want the programs to execute on a parallel architecture. We are now porting about 100,000 lines of FORTRAN-77 code from the Cyber to an Intel iPSC/2 hyperCube executing Fortran-386 from Green Hills Software.

3 Conversion Problems We Encountered

The challenges of this project have been mainly of three types:

1. *Dependency considerations:* The processing required in our application is record oriented. We chose to distribute the records one record to a processor. Given a given record R, we needed to identify all the information from other records needed to process R. These data dependencies determine the timing of messages between processors and ultimately determine the number of processors that can be used effectively in this application.

2. *Changes in word size:* Because the Cyber supports 60 bit words and the iPSC/2 supports 32 bit words, significant changes were necessary in arithmetic declarations and calculations. Similarly, bit string manipulations were problematic.

3. *Differences between the Source and Destination Systems:* Porting between two different operating systems and programming environments usually requires conversions between different system calls, machine constants, I/O conventions, and the like. These problems are not unique to porting from a sequential to a parallel processing environment, but they complicate other problems that are unique.

In the next section we discuss these problems in more detail. We also suggest programming practices that will make the problems easier to solve.

4 Programming Practices to Ease Conversions

Many of the suggestions in this section are variations on themes common in software engineering regarding design principles and coding conventions for producing easily ported code. For examples, Sommerville describes several specific problems due to word size and operating system differences [Sommerville] and Lamb advocates information hiding techniques to minimize problems when porting [Lamb].

However, the conversion from a sequential architecture and a sequential algorithm to a parallel architecture and a parallel algorithm raises new concerns and puts familiar concerns into a new light. This section as a whole presents the concerns that became most obvious to us during our conversion efforts. The first subsection on data dependencies has the least connection with traditional portability concerns.

4.1 Dependency considerations

Computations must be performed in a specified order. Sometimes the nature of an application uniquely determines the ordering of different operations in a computation; other times, different parts of a computation can be executed in any one of several orders. When a single processor is programmed to solve a problem, the ordering of different operations is more arbitrary than when multiple processor must cooperate to solve a problem. That is, some of the orderings that are equally convenient for a single processor will be either inconvenient or impossible for cooperating processes.

Data dependencies establish a partial ordering among the operations needed to complete a computation. Before a programmer can convert a sequential program to a parallel program, all these dependencies must be identified. When the dependencies are obvious to a programmer, the programmer must then determine an appropriate ordering of the operations that will facilitate independent computation by individual processors. Much of the work done by parallelizing compilers performs this task of identifying data dependencies. Unlike most parallelizing compilers, however, a human programmer may invent algorithms that violate some dependencies and still guarantee a correct conversion.

Often, an efficient parallel solution can be created which finesses certain dependencies in order to achieve a higher degree of parallelism. The programmer can use techniques such as copying data to several processors and making intelligent guesses about the outcome of a required result in order to circumvent the dependencies. However, the subtle reasoning required to devise these intricate strategies is only possible when the dependencies can be determined in great detail.

In our application, we determined two sources of dependencies: record validation and statistics gathering. Strictly speaking, a record should not

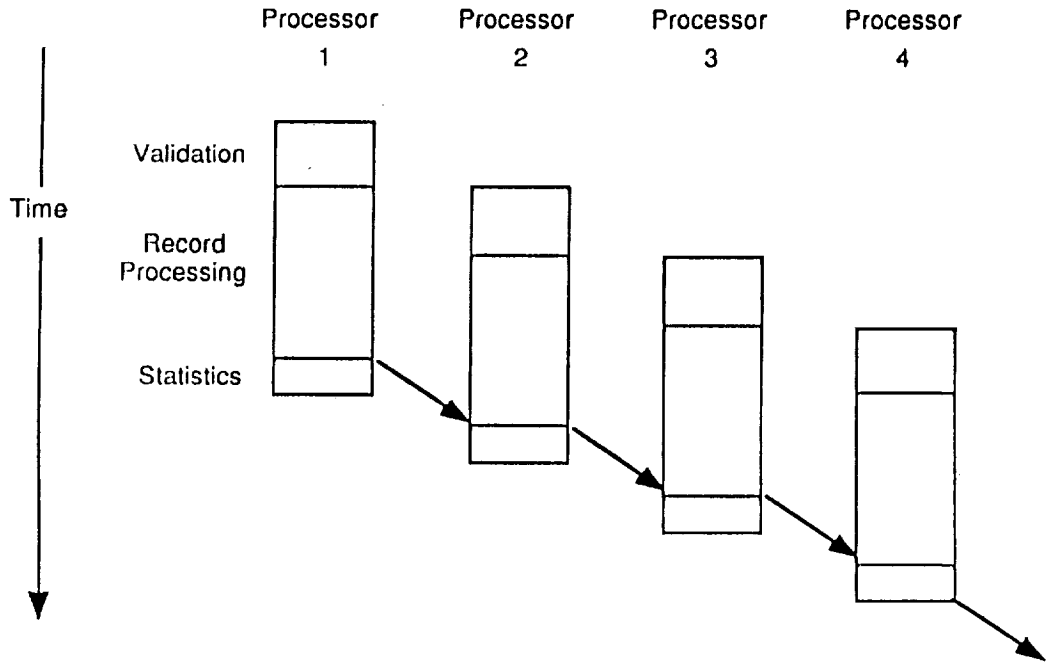

Figure 1: Parallel Record Processing

be processed until the following record had been validated. If we insisted on enforcing this dependencies, only the validation phase could have been parallelized, and much of the potential gain from multiple processors would have been lost. However, invalid records are relatively rare, and we decided to have each processor proceed under the assumption that the next record would be valid. Corrections are taken whenever an invalid record is discovered.

The statistics gathering did require sequential record to record timing, so our implementation on the iPSC/2 serializes the statistics gathering across all the records, but allows parallelization of the individual record processing. With this strategy, parallelization of the record processing proceeded according to the schematic shown in Figure 1. A message containing the accumulated statistics is passed from one processor to the next as soon as the statistics are available.

In order to implement this cooperation, we had to determine which code

did the validation task and we had to determine which data communicated that validation. The code was organized well to isolate the validation processing: there were two top level procedures that handled those tasks. However, the data dependencies were not nearly so obvious.

TIP #1: High level procedures should be organized according to purposes clearly delineated in the problem specification. This simplifies the task of isolating and, if necessary, changing the ordering of the operations involved in meeting that part of the specification.

TIP #2: Data structures should be organized according to purposes clearly delineated in the problem specification. This simplifies the task of isolating the information which must be passed among cooperating processors.

Top down design of procedures and functions has become increasingly common, and this practice encourages the type of organization suggested in TIP #1. Data structures less commonly are given this kind of attention. Often data structures are designed to minimize the length of parameter lists and data are grouped into larger and larger structures. These larger groupings can hide the true data dependencies between modules. Myers differentiates between "stamp coupling" and "data coupling": modules with stamp coupling share composite data structures; modules with data coupling share only variables that hold a single value. [Myers] For the purposes of program conversion, we do not require strict data coupling. Instead, we refine the idea slightly:

TIP #3: A routine should receive a composite data structure only if it can potentially require all the pieces in that structure. If not, a new smaller structure should be constructed, or the relevant atomic pieces sent.

The conversion effort proceeds only as quickly as the critical data dependencies can be identified. Simple declaration conventions can speed this identification.

TIP #4: Group declarations and the associated documentation according to data dependencies. Have a separate data dictionary organized alphabetically for each routine, cross-referenced into the grouped declarations.

4.2 Changes in Word Size

The Cyber's 60 bit integers could not be automatically converted to 64 bit integers because our compiler on the iPSC/2 does not include integers of this size. It became necessary to ascertain from documentation and from the ERBE data management team which of the integer variables would potentially use integer values that would overflow a 32 bit integer. Integer overflow is not detected by the runtime system on the iPSC/2, so no straightforward method exists for checking for this condition during execution.

TIP #5: Do not use implicit declarations. Do not use system-dependent defaults in declarations. Declare all variables explicitly, and (within the limits of the programming language) declare each variable with the tightest limits possible. When the programming language does not support sufficient declaration power to express the desired restrictions, describe the restrictions in a contiguous comment. Give these restrictive comments a distinctive syntax to simplify their automatic detection.

TIP #6: When such support is available, enable range checking. When such support is unavailable or its runtime costs are considered prohibitive, add code with explicit range checking at critical points in the computation. Mark this range checking statements with a distinctive syntax.

The Cyber word size also caused problems with bit strings. Again, it was not always clear from the documentation which bit strings had an effective length of 32 bits or fewer. Tracking down such use through levels of subroutines and function calls was tedious. This detective work was further complicated by code that used multiplication to accomplish bit shifts and code that use addition to accomplish bit sets. The detective work was simplified by named constants used in bit operations.

TIP #7: Make all operations as explicit and specific as possible. Named functions clarify the purpose of variables and the intent of operations. If explicit function and subroutine invocations are deemed too expensive, explain the specialized use of low level operations using an explicit comment with a distinctive syntax.

TIP #8: Declare each constant used in the program. Comment on the constant's purpose and derivation. For example, if a constant is a large integer, it may represent the largest integer available on the original machine, it may embody a particular value specific to the application, or it may be a value selected as a marker by the programmer. Each of these purposes may require a different action of the programmer parallelizing the code. If the same value is used for more than 1 different purpose, declare a distinct constant for each purpose.

4.3 System Differences

Some changes are expected when changing from one machine to another. Unless the machines share a standardized operating system, the conventions for job control, file management, and the like will be distinct. Syntax conventions for coding values or operations may differ. The order of parameters may differ on routines common to both systems. Careful documentation of these matters in the original code will ease the transition.

Isolation into purposeful modules has already been mentioned with respect to data dependencies. This same principle applies to the problems with system dependent features.

TIP #9: Isolate input and output operations into distinct routines. With multiple processors, communication is almost certain to change during conversion.

TIP #10: When system-dependent routines are used, attempt to group their invocation into application-specific modules. In these modules, describe the precise limitations and intended effect of each system-dependent routine. Invoke system routines with a consistent syntax throughout the program.

TIP #11: Use standard constructs and syntax whenever possible. Avoid specialized language extensions. When extensions are mandated by circumstance or decree, identify the extension in documentation and identify each use of that extension in the code. The prohibition against extensions includes "tricks," which are essentially undocumented extensions. Do not use practices that a particular compiler or operating

system allows, but are not generally available or expected in other environments.

TIP #12: Include in your documentation a bibliography of references with detailed information on the machine, operating system, programming language, and application associated with your program. Include version numbers and publication dates.

5 Conclusions

Prudence demands that we develop sequential programs in a way that will ease parallelization in the future. Current programming practice often does not lend itself to this enterprise. However, currently advocated practices, which can easily be adopted, can significantly ease the burden of such conversions. Many software engineering principles apply, but with added force. Other principles, concerning data dependencies, grow out of concerns that have not been traditionally of great concern in software engineering.

Some aspects of FORTRAN (particularly its declaration semantics) increased the difficulties of this conversion. However, no programming language forces a programmer to declare variables with tight bounds, and no compiler enforces the documentation of data dependencies. We contend that the issues raised above pertain to programming in any language. Furthermore, since parallel architectures are still in an early stage of development, the choice of languages may be limited to languages with less than ideal characteristics. In all cases, the principles should be applied, using distinctive comments when the source code does not lend itself to more explicit expression of the programmers intent.

A good parallelizing compiler may have helped us in this project. However, such compilers are not universally available, and converting to a new machine involves most if not all the problems described above. Furthermore, no compiler can be guaranteed to discern an optimal parallelizing strategy from sequential source code.

All the suggestions above can help a programmer make assumptions and intentions explicit in the source code. This added clarity facilitates a deeper understanding of the specified computation. Such understanding is valuable

not only in parallelization, but also during development and maintenance of the sequential code.

6 References

[Lamb] David Lamb. *Software Engineering: Planning for Change.* Prentice-Hall, Englewood Cliffs, New Jersey (1988), pp. 52-53.

[Myers] Glenford J. Meyers. *Reliable Software through Composite Design.* Petrocelli/Charter (1975).

[Padua and Wolfe] David A. Padua and Michael J. Wolfe. Advanced compiler optimization for supercomputers. *Comm. of the ACM.* Vol. 29, No. 12 (December 1986), pp. 1184-1201.

[Sommerville] Ian Sommerville. *Software Engineering, 2nd Ed..* Addison-Wesley, Wokingham, England (1985), pp. 132-141.

B A Dissertation Proposal

This Ph.D. dissertation proposal describes our research associated with the issues of software portability and high performance.

References

[1] CERES Data Management Team. *Clouds and the Earth's Radiant Energy System (CERES) Requirements and Functional Specifications for Process 3 -Identify Clouds: Version 0.* NASA Langley Research Center, Hampton, VA 23665-5225, September 1992.

Doctoral Dissertation Proposal

A Design Methodology for Portable Software on Parallel Computers

Dan A. Chrisman, Jr. *
Department of Computer Science, College of William and Mary

14 May 1992

Abstract

This paper proposes research which addresses two difficulties in parallel programming. The first difficulty is developing software which will execute quickly on a parallel computer. The second difficulty is transporting software between dissimilar parallel computers. In general, we expect that more hardware-specific information will be included in software designs for parallel computers than in designs for sequential computers. This inclusion, for performance reasons, complicates a software developer's attempt to keep important software on the newest high performance hardware. The problem of the proposed research is to create a design methodology that helps designers to more precisely control both portability and hardware-specific programming details.

*Supported by NASA/Langley grant and Virginia Space Grant Consortium

1 Introduction

The proposed research addresses two difficulties in parallel programming. The first difficulty is developing software which will execute quickly on a parallel computer. The second difficulty is transporting software between dissimilar parallel computers. In general, we expect that more hardware-specific information will be included in software designs for parallel computers than in designs for sequential computers. This inclusion is an instance of portability being sacrificed for high performance. New parallel computers are being introduced frequently. Trying to keep one's software on the current high performance hardware, a software developer almost continually faces yet another expensive software transportation. The problem of the proposed research is to create a design methodology that helps designers to more precisely control both portability and hardware-specific programming details. The proposed research emphasizes programming for scientific applications.

This proposal consists of five sections. Following this introductory section, the problem is discussed in section two and previously proposed solutions to the problem are discussed in section three. The proposed solution to the problem is discussed in section four. The proposed tasks are stated in section five.

2 Problem

Developing software for a parallel computer and transporting the software to a dissimilar parallel computer are two difficulties which a scientific parallel software developer faces. These two difficulties can be illustrated as developing software under two scenarios:

1. a problem's initial implementation on a parallel computer and
2. a problem's subsequent implementations on dissimilar parallel computers.

This proposal addresses the difficulties from the viewpoint of these scenarios.

Software development for a problem's initial implementation on any sequential computer is intellectually challenging and time consuming. A problem's initial implementation on a parallel computer typically requires additional development costs.

> Parallel processing is inherently more complex than sequential processing and will stay so for the simple reason that when we go to parallelism we have a domain which has many more dimensions. Sequential machines are unidimensional. When a machine becomes faster, everything becomes faster by roughly the same amount. Parallel machines are multidimensional: we have to worry about the number of processors, processor speed, communication speeds, latency, bandwidth, and so on. We have many more dimensions that affect performance of algorithms than we were accustomed to in the sequential world. (Marc Snir [47])

This additional complexity requires more sophisticated parallel programming models. Chandy contends that the models which exist for designing and programming parallel computers have heretofore been inadequate for much of the development in parallel applications:

> ...the basic sequential architecture—the von Neumann machine— has remained unchanged for decades. Programmers are typically isolated from variations in sequential computers by high-level compilers and industry-standard operating systems such as Posix. Such uniformity does not yet exist for parallel computers, which makes them less attractive platforms for software development [17].

The particular needs of scientific applications make these inadequacies quite glaring [48].

To generate an effective implementation on a parallel computer, parallel software development is generally based upon a hardware-specific parallel programming model. To transport the software to a dissimilar parallel computer, the software development must utilize another parallel programming model. Our use of the phrase *to transport parallel software* denotes the inclusion of any software development activity required to achieve an effective implementation on a dissimilar parallel computer. The cost to transport the software may range from a minimum of zero (i.e. no change) to some maximum (i.e. complete redevelopment). For Karp, the cost to transport software, even for small programs, remains ridiculously high:

> To read the trade literature, you would think that programming parallel processors is easy. In fact, the task of getting a program to run properly is often complicated—sometimes needlessly complicated. Complications arise from many sources. Many problems are due simply to running in unfamiliar programming environments. Others are

due to the preliminary nature of many of the parallel processing systems. Unfortunately, all too many are due in some way to poor planning by the designers of parallel systems. ...We take a simple program which approximates π by numerical quadrature and rewrite it to run on nine commercially available processors. It is surprising how complicated some of these programs become [41].

3 Previously Proposed Solutions

Many proposals to solve both aspects of the proposed problem have been and are currently being explored. The previously proposed solutions can be described as programming models, standardization efforts, design models, CASE tools and parallelizing compilers. For each category of the previously proposed solutions, a category description and category examples are provided.

3.1 Programming Models

Most of the previously proposed solutions relevant to the proposed problem can be described as programming models. These programming models are described in terms of Browne's parallel computation model properties (described in the next paragraph). Each property can be categorized as either the programmer's responsibility or as a programming abstraction. The portability of each programming model's implementation is considered. Finally, the formality associated with the programming model(if present) is described. The programming models to be described are:

1. MIMD-SM and MIMD-DM programming models
2. SIMD programming models
3. Gelernter's Linda,
4. Chandy's and Misra's UNITY.

Browne [12] provides a specification of the necessary but not sufficient properties of a parallel computational model. The properties include the primitive computation units, the rules for creating a computation structure which is constructed from primitive computation units, the type of address space which a computation structure can access, the way in which executing computation structures are synchronized, and the way in which information is shared between the computation structures's address spaces. Some examples of the model components are:

1. primitive units of computation: the operators and data structures used by the programmer,
2. computation structures: usually organized as streams of instructions executing in parallel, for example:
   (a) the parallel programming language construct based on the DO loop (e.g. DOALL which usually assumes iteration instance independence and DOACROSS which usually assumes dependencies between iteration instances),
   (b) the subroutine which contains work which can be partitioned and executed in parallel by many processes (e.g. PARSUB sub-name(arguments)) or which contains work which can be executed by a process and performed concurrently with other subroutines,
   (c) a physical process or a virtual process,
   (d) a micro-task or macro-task,
3. the computation structure's address space can be composed solely of shared memory(i.e. can access the address space of all instruction streams), can be composed solely of private memory(i.e. can access only its own address space), or some hierarchical combination of shared and private memories,

4. instruction stream synchronization which is explicitly-specified(e.g. semaphores) or implicitly-specified(e.g. blocking SEND which is a communication instruction),

5. information is shared between instruction streams explicitly(e.g. shared memory) or implicitly(e.g. message passing using SEND and RECEIVE instructions).

3.1.1 Multiple Instruction Multiple Data: An Introduction

One parallel computation model is the Multiple Instruction Multiple Data (MIMD) model [3]. This model assumes a parallel computer with two or more processors and a mechanism for the processors to exchange information. Each processor executes a processor-specific instruction stream and operates on a processor-specific data stream. With the required storage of an instruction stream and a data stream, the MIMD parallel computation model requires an associated memory model. Two memory models are the shared memory (SM) model and the distributed memory (DM) model. The next section describes the MIMD-SM programming model and provides several MIMD-SM implementations. Subsequent sections provide the analogous information for the MIMD-DM programming model.

3.1.2 MIMD-SM Programming Models: An Introduction

The MIMD-SM programming model provides a shared memory abstraction for the MIMD parallel computation model. Information stored in memory by any processor can be accessed by all processors.

3.1.3 MIMD-SM: The Force

Jordan's Force parallel programming language [39] provides a very broad programming model to develop portable parallel programs for MIMD shared memory multiprocessors. The Force programming language, which supports the MIMD-SM programming model, is composed of Fortran 77 with parallel language extensions. The primitive units of computation are the data types of Fortran and the associated operators. These primitive units are organized as the following computation structures: prescheduled and self-scheduled DOALL loops, prescheduled and self-scheduled parallel CASE statements, the subroutine, and the FORCESUB subroutine. The work which these constructs express is performed by Force *processes* . This process abstraction is managed by Force. To differentiate a subroutine from a FORCESUB subroutine, a subroutine call may be executed by an independent process. A FORCESUB subroutine is performed by many processes concurrently. The address space of a process is composed of memory which is either private to a process or shared with other processes. The programmer must categorize the data of a program as private or shared. The modes of synchronization of processes are either data or control synchronization and are explicitly controlled by the programmer. Data synchronization is specified by the Async, Consume, and Produce instructions. Control synchronization is specified by critical sections and the BARRIER instruction. The mode of communication for processes sharing information between their address spaces is via shared variables. The abstractions which Force provides are: independence of the number of processors executing a parallel program, suppression of process management, and non-process-specific synchronization. The Force has been implemented on the HEP, Flex/32, Encore Multimax, Sequent Balance, Alliant FX/8, and Cray-2 multiprocessors using computer-specific compilers.

3.1.4 MIMD-SM: ARguably Fortran

Saltz's programming model provides a global name space for solving sparse and unstructured scientific problems and is implemented as the programming language ARF (ARguably Fortran) [60] for MIMD distributed memory computers. Rosing's, Schnabel's, and Weaver's DINO environment [53] (described in a subsequent section) is designed to solve regular problems by distributing the work and data based on the data reference patterns which can be predicted at compile time. The irregular nature of Saltz's chosen problems makes data reference patterns unpredictable at compile time. Runtime analysis of data reference patterns are required to achieve reasonable processor load balancing. The primitive computation units are the data types and operators of Fortran 77. One computation structure is the DISTRIBUTE DO parallel construct. The iterations within the loop are to be distributed to the processors of the parallel computer. Another computation structure is the process, whose management is abstracted from the programmer except for the inclusion of the processor work-assignment instruction ON CLAUSE of Kali Fortran 1 [44]. The address space is abstracted as virtual shared memory although the target hardware for ARF is distributed memory parallel computers. The programmer must organize the address space by providing data distribution information to the compiler (e.g. *S1 DISTRIBUTED REGULAR USING BLOCK REAL K(SIZE)*). The synchronization required for correct execution of the work within a DISTRIBUTE DO is managed by the compiler and runtime support. No explicit synchronization statements are provided in ARF. Information is shared between processes via a virtual shared memory. The virtual shared memory on a distributed memory parallel computer is provided by the compiler and runtime support using a message-passing mechanism. This support manages the dynamic demands for interprocess communication. The programmer is alleviated from the responsibility for any message-passing specification by using a communication library Parallel Automated Runtime Toolkit at ICASE (PARTI) . The programmer provides a Fortran 77 program and adds data distribution information. The compiler adds the data communication code necessary to execute the program on a MIMD-DM computer. The runtime analysis of the code (performed during the inspector loop), inspects the potential work and schedules the necessary efficient communication to satisfy any off-processor data dependencies, either as input or output data. The runtime system then executes (during the executor loop) the input communication phase, the computation phase, and the output communication phase. The data arrays are distributed among the processors. The data does not necessarily reside on the processor on which it is updated. Each processor has a hash-table data cache which holds temporary data. The hashing is used to remove duplicate data items. Due to the sparse nature of the computation, the data, although distributed, may be stored in an irregular pattern on the processors. The location for each datum is stored in a distributed translation table.

3.1.5 MIMD-SM: CEDAR Fortran

CEDAR Fortran [34] provides a programming model for expressing parallelism on the CEDAR shared memory multiprocessor. The programming model provided by CEDAR Fortran is tightly coupled to the CEDAR hierarchical shared memory multiprocessor. Although providing a general parallel programming model is not a goal of the CEDAR Fortran development, the language's semantics are an interesting attempt to exploit different grains of parallelism on a hierarchical shared memory multiprocessor. The processors of the CEDAR multiprocessor are configured as four eight-processor clusters. For the Cedar-1 multiprocessor, a cluster is an Alliant FX/8 multiprocessor.

The shared memory of the CEDAR multiprocessor is hierarchical. The processors of a cluster share cluster memory. In addition, all processors may access global memory which is distinct from the cluster memories. CEDAR Fortran is composed of Fortran 77 with parallel language extensions. The primitive units of computation are the data types of Fortran 77 and associated operators and the array data type and array operators. Some computation structures are the array and vector extensions to Fortran 77 (i.e. triplet notation, FORALL vector loop statement and the WHERE vector conditional statement). Another computation structure is the microtasking provided by the concurrent loop statements DOALL and DOACROSS. Both of these statements have variants which are specific to a level of the hierarchical memory hierarchy. The processors which perform the iterations of a CDOALL or CDOACROSS statement must reside within a common cluster. The iterations of a SDOALL or SDOACROSS statement may be executed by processors across cluster boundaries. The XDOALL and XDOACROSS statements utilize all of the processors of the CEDAR multiprocessor. Another set of computation structures utilize the macrotasking library which allows the programmer to manage a larger grain of parallelism than the microtasking concurrent loops. A program's macrotask performs the work described in a subroutine using the processors allocated to it during the macrotask's invocation. The address space of a process may access either cluster shared memory or global memory. Each datum of a program may be declared as CLUSTER or GLOBAL, defining a datum's initial location. For macrotask execution, the Fortran 77 COMMON block is attributed as either a plain COMMON(a unique common is created for each macrotask in the program) or a PROCESS(only one COMMON block is created for a program). The modes of synchronization for parallel execution include synchronization at several levels of parallelism. These modes are determined by the use of either micro- or macrotasking. Synchronization for DOACROSS loops is specified by the ADVANCE and AWAIT statements. The medium grain lock and event synchronization primitives are used by processes owned by separate macrotasks. For processes within a macrotask, the WITH statement provides a critical section synchronization mechanism. The modes of communication which allow information to be shared between processes' address spaces are based on direct access to the cluster shared memory and global shared memory.

3.1.6 MIMD-DM Programming Models: An Introduction

The MIMD-DM programming model provides a distributed memory abstraction for the MIMD parallel computation model. Information stored in memory by a processor is private and cannot be accessed by another processor. Unlike the MIMD-SM programming model, explicit communication instructions are required to move data from the private memory of one processor to the memory of another processor in the MIMD-DM programming model. One instantiation of the MIMD-DM parallel programming model is Hoare's Communicating Sequential Processes (CSP) [36]. Hoare's CSP model incorporates Dijkstra's nondeterministic guarded command, Dijkstra's PARBEGIN command for concurrent execution of sequential commands, and process-blocking interprocess communication commands. One instantiation of Hoare's CSP on a parallel computer is the Occam 2 language which executes on a mesh of INMOS Transputers. The European computer science community is currently discussing the adoption of CSP as a general purpose parallel programming model for all parallel computers. Zenith contends that this adoption is inappropriate for efficiency reasons, particularly for MIMD-SM parallel computers [61].

3.1.7 MIMD-DM: Intel Hypercube Message-Passing Fortran

For the Intel iPSC/2 hypercube parallel computer, the MIMD-DM programming model is supported by an extended procedural language [24]. For the extended Fortran 77 language [23], the primitive computation units are the data types and operations of Fortran 77. The computation units are the processes which are instances of program executions. These instances may be organized in a Single Program Multiple Data (SPMD) configuration or as concurrent executions. The address space of a program is its private memory. There is no shared memory. The modes for synchronization may explicitly synchronize processes using the BARRIER-like WAITALL() and WAITONE() system calls. Implicit synchronization is incorporated into the one of the communication modes. Message passing between processes provides the only way to share data between processes. The synchronous communication instructions CSEND() and CRECV() cause the process to block until the message has been sent or received respectively. The asynchronous communication instructions ISEND() and IRECV() do not block and contain no implicit synchronization. A third communication mode allows message passing calls HSEND() and HRECV() to be used as interrupt handlers, defining a function to be executed when the message has been successfully sent or received respectively. Each send instruction instance is parameterized by a programmer-defined message type, a pointer to the location of the message to be sent, the message size (in bytes), the destination processor id, and destination process id. Each receive instruction instance is parameterized by a programmer-defined message type, a pointer to a location for the incoming message, and the message size (in bytes). The programmer is responsible for insuring that deadlock will not occur. The support for these communication modes is provided by the Intel NX operating system which manages the message buffers. The monitoring and flushing of these buffers is under the programmer's control. At a higher level of abstraction, global operation instructions are provided which utilize distributed data and provide each participating process with the result. This programming model is supported by Fortran and C extensions which are specific to the iPSC/2 hypercube computer.

3.1.8 Single Instruction Multiple Data: An Introduction

The Single Instruction Multiple Data (SIMD) parallel computation model [3] is almost identical to the MIMD parallel computation model. Unlike the MIMD parallel computation model, each SIMD processor executes an instruction stream which is identical to the other SIMD processors. Like the MIMD computation, each SIMD processor operates on a processor-specific data stream. The next sections provide several SIMD programming model implementations.

3.1.9 SIMD: DAP Fortran

The vendor-specific SIMD programming model used to program the Active Memory Technology (AMT) DAP parallel computer [1, 37] is incorporated in DAP Fortran-Plus [15]. This extended version of Fortran 77 provides a programming model which allows the programmer to manipulate vectors and matrices on the DAP. The DAP is a 32 x 32 array of single bit processors. The data object manipulations are specified using vector and matrix notations. Intrinsic manipulation functions (e.g. vector maximum) also operate on these data objects. The primitive units of computation are vectors, matrices, and their associated operators. The rules for turning the primitive units of computation into computation structures is based on the SIMD execution method(i.e. lock step). The modes of synchronization for parallel execution are implicit in the SIMD lockstep execution

scheme. The modes of communication between computation structures and their address space are limited to accessing a processor's four nearest processors with wraparound at the edges of the DAP.

3.1.10 SIMD: Paragon

Reeves's Paragon programming environment addresses the problem of developing efficient and portable scientific software for a variety of multicomputers [52]. Reeve's previous research concerned a parallel version of Pascal [51] which was developed for the NASA MPP (a SIMD computer which predates the Connection Machine) [3]. The Paragon environment includes a compiler and a runtime system. The language semantics are designed to support scientific processing on a wide range of multicomputers. The targeted range of computers is shared and distributed memory multicomputers and algorithmically appropriate vector and SIMD array computers. Reeves's approach is to focus on task management. The programmer must provide some hardware-independent task information and the run-time system will manage the tasks. The Paragon language is based on the SIMD programming model. The first data structure which the author considers is the array. The Paragon language syntax is C with extensions. Intrinsic functions for array manipulation are provided. These functions are similar to those in Parallel Pascal [51]. Reeves separates the declaration of the type of the data structure with the declaration of the shape of the structure. This information is used to distribute the data on the parallel computer and is used by the task manager. Reeves also introduces communication structures. These are data structures which serve as pointers used to specify how elements of a data structure are mapped to another data structure. Data locking is an alternative to communication structures when only several elements of a data structure are required for a computation. The Paragon language has been implemented for any uniprocessor which has a C++ compiler and implemented on two homogeneous multicomputers: an Intel iPSC/2 hypercube with a Unix operating system and a network of transputers with Trollius operating system [20]. The automatic task allocation required by the distributed memory computers uniformly distributes the array data structure as contiguous blocks. The runtime system will dynamically redistribute the data array based on load balancing concerns.

3.1.11 Linda and Tuple Space

Gelernter developed a parallel programming model composed of a sequential base programming model (e.g. C or Fortran) and the Linda memory model [14]. The Linda memory is called tuple space. One extended sequential language which supports the Linda parallel programming model is Linda-Fortran . The base language is Fortran 77 and the extensions are tuple space operations. The primitive units of computation are data types and operators of the base language and the tuple types and tuple operators. There are two types of tuples: data tuples and process tuples. A data tuple is a passive object and a process tuple is under active evaluation. Tuples are manipulated with four fundamental tuple operations: OUT(t1), IN(t2), RD(t2), and EVAL(t1) where t1 is a tuple and t2 is a tuple template. OUT(t) evaluates a tuple and places the resulting tuple in tuple space. IN(t) finds a tuple in tuple space which matches the tuple template t, assigns formal arguments to actuals, and withdraws the matching tuple from tuple space. RD(t) finds a tuple in tuple space which matches the tuple template t, assigns formal arguments to actuals, and leaves the matching tuple in tuple space. EVAL(t) is the reverse of OUT(t) in that the tuple is placed in tuple space where the tuple becomes a process tuple and is evaluated. The result of the evaluation causes the process tuple to become a data tuple which contains the result. With the process tuple, the Linda

model provides for process creation and coordination. INP(t) and RDP(t) are predicate versions of IN(t) and RD(t) respectively. If a tuple match is not immediately found for the INP(t) or RDP(t) tuple operator, then a tuple-match failure indicator is returned. The computation units are the control structures of the base language which utilize the tuple operators. The address space in which the complex computation structures executes is tuple space. The modes of synchronization for parallel execution are based on the execution blocking semantics of the IN(t) and RD(t) tuple operators. Execution of either operator requires that a tuple be found in tuple space which matches the tuple template t. Both operators will block until a tuple in tuple space is found. The modes of communication between computation structures and their address space are via the RD(), IN(), and OUT() tuple operators.

There are concerns about the effectiveness of Linda programs [54] and concerns about the insidious demands placed upon the programmer by the Linda compiler optimization techniques [61].

3.1.12 UNITY

Misra's and Chandy's UNITY programming model is expressed in four parts. The two formal parts are the abstract program and a program proof. The two informal parts are the mapping and the architecture descriptions [18].

A UNITY program is a computer-independent specification which is expressed with a notation composed of four constructs: a DECLARE section, an ALWAYS section, an INITIALLY section, and an ASSIGN section [16]. A DECLARE section contains a variable's name and type declaration. The ALWAYS section contains the functional relationship between variables. The INITIALLY section allows the assignment of an initial value to a variable. The ASSIGN section contains a set of assignment statements.

The program proof system is based on Hoare axiomatic system[35]. Two UNITY extensions are *guarantee* and *goal* which are conditional terms which facilitate the program proof by compartmentalizing the information [16].

Due to the computer-independence of the UNITY program, a mapping of the program to a parallel computer is required. Thus the simplicity of specifying a UNITY program due to its computer-independence is balanced by the cost of specifying a mapping. Unlike the formal specification of a UNITY program, the description of a mapping of a UNITY program to a parallel computer and the description of the parallel computer remain informal. The programmer describes the mapping as a distribution of program statements, variables, and control flow for each processor.

Chandy indicates that compilers are being written for SIMD and MIMD computers.

3.2 Standardization Efforts

The focus of several language standardization efforts has been to provide programmers of Fortran 77 [10] with access to the high performance of parallel computers. Karp proposed the need for parallel language standardization as a response to the questionable health of the software engineering aspects of parallel programming:

> The state of the art of parallel programming and what a sorry state it is in [40].

Karp discussed the different types of parallel computers and the hardware-specific tools for expressing parallelism. He focused upon scientific programming on MIMD processors using Fortran

dialects. He concluded that the parallel programming community needed a set of appropriate extensions to Fortran. These extensions would allow programmers to express the parallelism of their problems at a medium parallelism grain. These programs would attain the speed desired by the programmers. Several efforts to standardize these extensions for Fortran have begun. Fortran 90 [38] incorporates a concise array syntax and semantics which aids vectorization for supercomputers [45]. A DO loop parallel programming feature was removed from the Fortran 90 standard just prior to the standard's release [33]. Two different standardizations of parallel Fortran dialects were begun by the Parallel Computing Forum [55] and by the High Performance Fortran Forum.

3.2.1 X3H5

The effort to standardize a parallel Fortran dialect, which was begun by the Parallel Computing Forum, became an effort of the American National Standards Institute (ANSI)-accredited technical committee X3H5 committee. The ANSI X3II5 model document [13] describes a language-independent programming model with the following purpose:

> our intent is to standardize current practice through the definition of parallel constructs which are portable and language independent. They are intended to be implemented for procedural, imperative languages such as Fortran 77, Fortran 90, Pascal and C.

The primitive units of computation are the data types and operators of a procedural, imperative language like Fortran 77 or C. The computation structures are the parallel constructs. The process is a computation structure, but is managed implicitly during the execution of a parallel construct. A program begins with one process and proceeds serially until a parallel construct is executed. Within a parallel construct, all processes of the team execute the same instructions until a work sharing construct is encountered, then each process of the team is assigned the work specified in the work sharing construct. A barrier is formed at the end of work sharing construct and at the end of the parallel construct. A process's data may be explicitly categorized as private to a process or shared among more than one process. Synchronization is provided to communicate values of shared objects, control the access to shared objects, and provide control synchronization. All shared variables modified during a construct must be updated before the constructs barrier is crossed(i.e. implicit synchronization). Explicit synchronization is available in the form of locks, events, and sequences. The information which is to be shared between processes relies upon accessing shared memory. No message passing between processes is allowed for this model. The process management is abstracted. The model is based on a shared memory computer but may be bound to languages which target distributed memory computers. This standardization effort is incomplete and this programming model has not been implemented on any parallel computer. One restriction is that all programs for this model must be executable on a single processor.

The X3H5 Fortran 77 language binding document [22] defines a binding to the aforementioned model document [13]. This binding is an extended version of Fortran 77 whose design was originated by the Parallel Computing Forum [55]. The parallelism which this language exploits is thread-based unlike the array-based extended Fortran 77 dialects (e.g. DAP Fortran-Plus [15], High Performance Fortran (HPF), the Connection Machine's CM Fortran [58]). The language is being designed to

> provide simple, intuitive constructs which allowed users to introduce parallelism into an existing application with minimal changes and also to code new parallel programs.

As the design of the programming model and the binding of the extended Fortran 77 language have evolved, the committee opined that this language may also be a target language for a compiler in addition to a high level programming language for a human programmer.

The primitive units of computation are the data types and operators of Fortran 77. The computation structures are the PARALLEL DO and PARALLEL SECTION constructs.

> The PARALLEL DO construct is used to specify parallelism among the iterations of a block of code. The PARALLEL SECTIONS construct is used to specify parallelism among sections of code.

The address space for the computation structures is a virtual shared memory, thus this language's implementation targets are shared memory and distributed memory parallel computers. The model provides implicit and explicit modes of synchronization. The implicit synchronization is provided in the PARALLEL DO and PARALLEL SECTIONS constructs. Specifically, synchronization of the processes is implicitly established at the end of the parallel constructs and the end of the work sharing constructs. The explicit synchronization is built upon the synchronizer type declarations: GATE, EVENT, and SEQUENCE. Variables of the GATE type are used in the unstructured synchronization statements LOCK and UNLOCK. Variables of this type are also used in the structured synchronization statement CRITICAL SECTION. Variables of the EVENT type are used with the event synchronization statements: CLEAR, POST, POSTED, and WAIT. Variables of the SEQUENCE type are used with the sequence synchronization statements: SET, POST, POSTED, and WAIT. The modes of communication between computation structures and their address space are via direct assignment of shared variables. Message passing is not a feature of this language. This extended Fortran 77 language binding has not been implemented on any parallel computer.

3.2.2 High Performance Fortran Forum

The first meeting of the High Performance Fortran Forum (HPFF) was January 27-28, 1992. The formation of HPFF is in negative response to the emerging PCF standard. The High Performance Fortran Forum is a working group convened by Ken Kennedy and Geoffrey Fox to create an informal standard for Fortran extensions aimed at data parallel computation. HPF will be strongly influenced by Fortran D [31], Vienna Fortran [19], Kali Fortran 1 [44], DEC's HPF proposal [42], and the Cray Research MPP Fortran programming model [49]. Agreement has been reached that an array-based programming model will be more effective in a timely manner than the thread-based PCF programming model. Fortran D is the basis for the standard being discussed and is described below.

The Fortran D language [31] provides a programming model for writing data parallel programs which are portable across parallel computers. Fortran D is a data-parallel version of Fortran 77. The programmer specifies a problem mapping using the DECOMPOSE and ALIGN statements and a computer mapping using the DISTRIBUTE statement. Irregular distributions are specified with a ALIGN MAP WITH statement for an array of pointers. A FORALL statement is used for specifying a parallel DO loop where loop instances are independent. Some reduction operations on arrays are included. For the assignment of specific work to a processor, the ON clause of Kali is included. This language is aimed toward the scientific data parallel problems. These commands require the expression of the problem in terms of a one or more arrays. The target computers for Fortran D are shared and distributed memory parallel computers. The primitive units of

computation are the data types and operators of Fortran 77. The rules for turning the primitive units of computation into computation structures are specified by the definitions of the Fortran 77 extensions (i.e. FORALL statement) and managed by the compiler. The rules for constructing address spaces in which the complex computation structures execute are implicit in the definition of the DECOMPOSE, ALLIGN and DISTRIBUTE statements. The modes of synchronization for parallel execution are implicit as specified by the definitions of the FORALL instruction. The modes of communication between computation structures and their address space are via the shared memory abstraction.

3.3 Domain-Specific Libraries

Dongarra et al. [4] have provided a domain-specific programming model via a portable library Linear Algebra Package (LAPACK). This package is an extension to the EISPACK and LINPACK libraries. All of these computer-optimized libraries are based on the Basic Linear Algebra Subroutines (BLAS). The purpose of LAPACK is to

> provide routines for solving systems of simultaneous linear equations, finding least-squares solutions of overdetermined systems of equations, and solving eigenvalue problems.

LAPACK utilizes the BLAS which are categorized as Levels 1, 2, and 3 BLAS. Currently, computer-specific optimizations are limited to the BLAS. Level 1 BLAS are vector-vector operations. Dongarra contends that

> the Level 1 BLAS permit efficient implementation on scalar machines, but the granularity is too low for effective use on most vector or parallel machines.

Level 2 BLAS are matrix-vector operations and Level 3 BLAS are matrix-matrix operations. These upper levels offer larger granularity than Level 1 BLAS and allow the programmer to design algorithms which reduce memory traffic on the memory hierarchies of the newer sequential and parallel computers. In particular, a programmer exploits the Level 3 BLAS by expressing algorithms in terms of submatrices operations (versus vector or scalar operations). However, research is continuing to define the optimal or near optimal submatrix sizes for differing computers. The target computers for LAPACK are fast scalar, vector, and large-scale, general purpose MIMD-SM computers. Current portability is provided for CRAY-2, Cray X-MP, Cray Y-MP, Fujitsu VP, IBM 3090/VF, NEC SX, Hitachi S-820, Alliant FX/80, Convex C-1, Convex C-2, Stardent, Sequent Symmetry, Encore Multimax, and BBN Butterfly. One of the proposed extensions to LAPACK [5] is a distributed-memory version. Current research for the MIMD-DM implementations suggests that introducing parallelism at the top-level algorithmic level will be more beneficial than at the BLAS level. Current implementations of LAPACK are expressed in Fortran 77 with limited computer-specific extensions. A list of bugs in a preliminary version of LAPACK and some timing data for LAPACK after the BLAS routines were partially performance-tuned are also provided [6].

3.4 Design Models

Design models have been previously proposed as solutions to developing and transporting software for parallel computers. These design models are hampered by the lack of parallel programming

models which cover dissimilar parallel computers and which lead to efficiently executing parallel software. The current design models must deal with computer-specific details which are not abstracted by the parallel programming model. If these details are not considered, then performance is likely sacrificed.

For the design models discussed below, the transformational design methodology of Gelernter's design model, Darlington's parallel-execution-efficiency-enhancing transformation methodology, and the proof system associated with Chandy's design methodology have heavily influenced the proposed solution. Huang's research is an example of a parallel detailed design methodology which is closely linked a parallel programming model to achieve the desired performance.

3.4.1 Gelernter's Design Methodology

Gelernter's design methodology [14] focuses upon frameworks of parallelism and techniques used to translate a solution from one framework to another framework. Such a translation is usually motivated by a desire for increased efficiency. Gelernter categorizes parallelism using three frameworks and associates a programming method with each framework. For each pair of frameworks, techniques are presented which allow conversion between frameworks.

The parallelism which is inherent in a problem can be viewed from three perspectives, or within three frameworks: result, agenda, and specialist parallelism. The three frameworks are described in terms of a problem which exhibits parallelism and many workers to perform the necessary work. For result parallelism, the result is partitioned into quanta of work. This partitioning is not based on functional lines, instead it is a brute force partitioning. Each worker produces a piece of the result and the workers are working in parallel. For agenda parallelism, the result is described as a sequential agenda of activities which must be completed. All of the workers, viewed as generalists, cooperatively tackle and complete a given agenda item before beginning the next agenda item. For specialist parallelism, the result is defined along functional lines. All of the workers are specialists and each worker applies his specialty as needed.

> Result parallelism focuses on the shape of the finished product; specialist parallelism focuses on the makeup of the work crew; and agenda parallelism focuses on the list of tasks to be performed.

The three programming methods for translating these concepts into working programs are live data structures, distributed data structures, and message passing.

The live data structures are associated with result parallelism. The designer views the result as a data structure. The designer, exploiting result parallelism, partitions the result into pieces (i.e. the data structure, which represents the result, is partitioned into data elements). A process is implicitly associated with each data element, thus each element is considered *live*. When a live data element computes its value, it communicates its value to other live data elements by becoming that data object. Processes implicitly communicate by referring to other data structure elements.

The message-passing approach is associated with the specialist parallelism. A process is explicitly created by the programmer and explicitly associated with a specialist. A specialist explicitly communicates its value as often as necessary to other specialists.

While the message-passing and live data structures have data structures distributed among processes, the distributed-data-structure approach is composed of a group of processes and a group of data structures. The distributed data structures approach is associated with agenda parallelism. Process coordination and communication is via shared data structures.

The techniques of abstraction/specialization, explicit/implicit and clumping/declumping are used to translate a solution from one framework to another.

Abstraction is used to translate from a result or specialist parallelism framework to an agenda parallelism framework. Abstraction is applied to a live data structure (result parallelism) or a group of specialists (specialist parallelism). Their processes are cleaved from their data, and the data is centralized into a shared distributed-data-structure (agenda parallelism). The opposite of abstraction is specialization, where the distributed data structure is partitioned and distributed to the processes of a live data structure or the specialists.

Explicit communication, and optionally *clumping*, are used to translate from a result parallelism framework to a specialist parallelism framework. The designer can translate from a live data structure (result parallelism) to the message passing of the specialist (specialist parallelism) by making process communication *explicit*. Optionally, the processes of the live data structure may also be *clumped* (i.e. composing the processes) for a coarser parallelism grain. Alternatively, *implicit* communication and optionally, *declumping* are used to translate from a specialist parallelism framework to a result parallelism framework. The designer can translate from the message passing of the specialist (specialist parallelism) to a live data structure (result parallelism) by making process communication *implicit*. Optionally, the processes of the message passing specialists may be *declumped* (i.e. decomposing to many processes) for a finer parallelism grain.

3.4.2 UNITY Design Methodology

The four components of Chandy's and Misra's UNITY program development methodology are: a specification notation, a programming notation, a proof system, and design heuristics. The design specification notation, which is a computer-independent programming notation, is discussed in section *Previously Proposed Solutions*, subsection *UNITY*. UNITY encourages the design strategy of rapid prototyping followed by stepwise refinement to provide adequate performance efficiency. A majority of the computer-specific details are considered during the later stepwise refinement steps where the mapping is developed. Program efficiency is considered mostly during the mapping development. The programmer is encouraged to develop small programs, combine the small programs for increased functionality, and apply heuristics to refine the program to meet performance requirements. The respecification is proved correct for each refinement.

3.4.3 Darlington's Design Methodology

The Hope+ declarative language is part of a transformation system used to develop computer-specific programs [27]. Darlington et al. present a development philosophy for parallel architectures and declarative languages. The sound mathematical basis and amenability to formal manipulation of declarative languages makes them attractive for being transformed from a higher-level specification using meaning-preserving rewrites to an efficient program. *Program fragments*, written in higher-level specification language, are replaced with fragments having the same denotational semantics but more efficient operational semantics. The major operational inefficiency is attributed to the load-sharing imbalance among processors. The responsibility for load sharing is taken away from the programmer and hardware and given to the program transformation techniques. Simple performance prediction is considered during the application of transformations to the original program. Execution statistics are used to validate the performance prediction equations.

3.4.4 Huang's Design Methodology

Huang and Fencl provide a methodology for designing parallel versions of sequential algorithms [29]. These designs are sensitive to the processor interconnection network. Huang and Fencl focus upon loop sequential programs. A sequential algorithm is characterized by a set of operations and their sequential execution order. Based on the data dependence of the operations, a relaxed operation execution order is expressed with semantic relations. A parallel algorithm is completely specified as an operation set (set of work quanta), a parallel scheduling function (when a quanta of work is performed), an operation distribution function (where a quanta of work is performed), a data distribution function (where data resides as a source), and a data movement function (how data is communicated between processors). The designer of the parallel scheduling function must maintain consistency with the operation's semantic relations. Huang and Fencl focus upon the parallelization of a doubly nested-loop computation as it would be expressed in a sequential, imperative language (i.e specifically an iterative sequential algorithm). Huang and Fencl acknowledge the need to provide a method of deriving the specification of the operation set and accompanying four functions. This is indeed a weakness of their approach. Huang and Fencl provide a methodology which synthesizes an executable parallel program from their parallel algorithm design [30]. An extended Pascal program is synthesized from the design. The main extension is FORALL p WHERE c DO s where p is the set of processors, c is a condition evaluated for each work quanta and s is the set of work quanta. This command activates a set of processors p that satisfy condition c. These processors simultaneously begin performing their respective quanta of s. The other extensions are a nonblocking destination-specific SEND command and blocking source-specific RECV command. Huang and Fencl explore the synthesis of computer-specific programs from processor-interconnection-network-specific designs. A methodology for the synthesis of programs for synchronous execution model architectures with static or dynamic data distribution is provided. A similar methodology is provided for the synthesis of programs for asynchronous execution model architectures which require synchronization via a *variable switch* for shared memory (a variable lock) and *message passing* for distributed memory.

3.5 CASE Tools

The maturity of the CASE tools are limited by the maturity of the underlying software development models which they support. Due to the immaturity of the parallel software development models, CASE tools tend to be specific to one parallel computer, or to one parallel language on a parallel computer. Chang and Smith [21] classify programming tools which develop software for parallel computers. The variables for classifying the tools are: tool type, language type, computational model, language features, environmental features, hardware platform, operating system(s), language support, graphics interface, design philosophy, application, developing status, and author. A paragraph evaluating each of the 19 tools is provided, along with a 47-entry bibliography. Their report is a stepping stone to many tools.

One trio of software development tools(BUILD, SCHEDULE, and HeNCE) was sponsored by the Department of Energy's Argonne National Laboratory. BUILD [11] is a graphical tool to construct execution dependency graphs where each node is a problem expressed as Fortran subroutine. The library SCHEDULE [28] contains Fortran subroutines used to 1) specify the parallelism of the problem and 2) incorporate a queued task execution model. The SCHEDULE programs are executed on a variety of sequential computers and shared memory parallel computers (e.g. VAX

11/780, Alliant FX/8, Sequent Balance 21000, Encore Multi-Max, Cray-2, Cray-XMP, Cray-YMP, IBM 3090, and Flex 32). HeNCE [9] extends the SCHEDULE/BUILD capabilities to dynamic scheduling of the tasks in a network-based heterogeneous computing environment.

Several of the report's parallelizing compilers are discussed in the next section(i.e. PED, PTOOL, PFC, PTRAN, and Parafrase).

3.6 Parallelizing Compilers

The goal of a parallelizing compiler is to automatically restructure sequential programs to execute efficiently on a parallel computer. Many compilers rely upon program execution profile information in order to choose the appropriate restructuring technique. The complexity of some sequential software requires human interaction. The research in this area remains computer-specific due to the immaturity of parallel programming models.

3.6.1 PED, PTOOL, and PFC

Kennedy et al. developed the Parascope EDitor (PED) [7] to interactively parallelize sequential software. PED is the combination of several automatic parallelizing tools. The Parallel Fortran Converter (PFC) began as an automatic Fortran 77 vectorizer which performs data dependence analysis, interprocedural side effect analysis, and interprocedural constant propagation. The dependence analysis categorizes the relationships between statement execution instances as true, anti-, or output dependence and loop-independent or loop-carried dependencies. Loop-carried dependencies constrain the order of execution of the iterations of a DO loop. PFC was subsequently reconfigured as a conservative parallelization tool. The output of PFC is

> a statement dependence graph that specifies a partial ordering on the statements which must be maintained to ensure the sequential semantics of the program do not change due to parallelization.

PFC's main weakness was indicating spurious race conditions in large, complex loops, thus thwarting parallelization. An interactive browser PTOOL was developed to debug programs which had been parallelized by PFC. PTOOL used PFC's statement dependence graph, but applied a dependence filtering mechanism to evaluate and automatically remove many of the spurious race conditions. PTOOL incorporated improved data dependence analysis. To further facilitate parallelization,PTOOL applied another technique to determine whether a variable is private to a loop iteration (i.e. a private variable cannot inhibit parallelization) or shared between the loop body and either the pre-loop code or post-loop code. PED extended the capabilities of PTOOL by allowing the programmer to interact during the program's parallelization. Operationally, PED performs data dependency analysis at the statement level and then queries the user interactively concerning the desired code transformation. The code transformations which the user can interactively apply are similar to those described by Padua and Wolfe [46]. PED provides a programmer with a low level profitability estimate for a potential transformation. PED generates parallelized code for shared memory parallel computers. Currently, PED generates IBM parallel Fortran and parallel Fortran for the Sequent Symmetry.

3.6.2 PTRAN

Parallel TRANslator (PTRAN) [2] is a research compiler which searches for parallelism in a sequential Fortran program. This tool relies upon studying the control dependence of a sequential program to determine its parallelism [25]. The control dependence information is stored in a control flow graph, where a node can be a Fortran statement or a Fortran condition (i.e. IF condition). PTRAN inserts additional nodes into the control flow graph to identify the loop structures within the sequential code. The new nodes of the resulting augmented control flow graph contain information from data dependence analysis, constant propagation analysis, and alias analysis. To simplify the partitioning analysis of the augmented control flow graph, the forward control flow graph, a subgraph of the augmented control flow graph, is formed by making the augmented control flow graph acyclic. To generate IBM parallel Fortran constructs (e.g. PARALLEL LOOP and PARALLEL CASE), the forward control flow graph is traversed in depth-first order. Profiling information is associated with each augmented node of the augmented control flow graph and is considered during parallel task partitioning decisions.

3.6.3 Parafrase

Polychronopoulos et al. [50] contend that functions traditionally performed by the operating system may be more effectively performed by the compiler for a parallel computer. Polychronopoulos et al. are developing Parafrase-2, a source-to-source restructuring compiler. A preprocessor converts a C or Fortran program to P-2 intermediate code and this code is manipulated to execute on a parallel computer. The data dependence analysis is based on determining the IN and OUT sets (i.e data used and data modified, respectively). Scalar and indexed statements are considered and the order of execution between two statements is determined. The statement dependencies are categorized as flow, anti-, and output dependence. The data dependence graph is constructed from the evaluation of the data dependence distance (i.e. nesting levels) and the true distance (i.e. the number of iterations between a variable instance's access). Graph transformations are performed to extract the parallelism of the problem (e.g. loop-parallelization, loop-vectorization, loop-distribution, loop-interchanging). When the classical dependence tests (e.g. gcd, Bannerjee's bounds test and exact tests) fail to decipher the dependencies, then the relatively expensive symbolic tests are performed. Another facet of the Parafrase-2 compiler is the Static Performance Analyzer (SPA) which estimates a program's execution time using 3 performance models. These estimates are used by the compiler to choose the appropriate performance-enhancing program transformations (e.g. vectorization or parallelization). Interprocedural analysis is performed to gather general information concerning data object reference, aliasing, and execution contexts (i.e when data values can define data dependence relationships). While insufficient data dependence information may lead a lesser compiler to abandon an optimization approach or apply a conservative optimization, Parafrase-2 will generate multiple versions of the code where a runtime decision will guide the choice of versions. The final feature of the compiler is auto-scheduling which partitions and dynamically schedules tasks, normally an operating system responsibility.

3.6.4 Browne's ESP

Browne et al. [56] developed an interactive programmer-assisted sequential Fortran program parallelizer. It is based on the research of Kennedy (PTOOL), Allen (PTRAN), and Polychronopoulos

et al. (Parafrase). It is also based on vendor-supplied parallelizing compilers which, for limited cases, find DO loop parallelism in sequential code and insert vendor-specific parallel DO loops into the software. Browne's distinctive thesis is that

> successful attainment of large scale parallelism will require both macro-level parallelism created by human analysts and micro-level parallelism recognized and implemented by automated analysis.

The parallel structure of the code is represented in a hierarchical dependence graph. The level of graph resolution becomes as fine as the statement level. Complete dependence information, determined from data dependence and intraprocedural analysis, at all levels of the hierarchy, is stored in a database. Profiling data from the execution of the sequential code is also stored in the data base. This data base is a resource for the parallel computer simulator which can guide the programmer's parallelization choice of several parallel constructs for a block of sequential code. Developed by Scientific and Engineering Software, Inc. under a contract from Concurrent Computer Corporation, the tool was eventually named ESP. Although the original intent of the tool was to simulate several parallel computers, only the simulation of the Concurrent Computer Corporation 3200 MPS shared memory multiprocessor was completed [59]. The environment allows the programmer to specify the execution measurements to gather performance data for the parallel code.

4 Our Proposed Solution

The proposed solution provides a design methodology which addresses two difficulties in parallel programming. The first difficulty is designing software which will execute quickly on a parallel computer. The second difficulty is transporting software between dissimilar parallel computers. The proposed methodology directs the programmer to construct a software design which is specific to a parallel computer. If the programmer is unsatisfied with the initial design, then the methodology prescribes actions which can improve the design. When the programmer must transport the software to a new dissimilar parallel computer, then the methodology directs the programmer to modify the software design, creating an equivalent design which is specific to the dissimilar parallel computer.

The components and attributes of the proposed parallel software design, the definition of the proposed equivalence-preserving design transformer, and the proposed method of constructing a design are described below.

4.1 Introduction

The software development process begins with a programmer's need to solve a problem on a parallel computer. For many scientific programming problems, a programmer turns to parallel programming to solve a problem faster or to solve a larger problem than achievable on a sequential computer.

> Nobody wants parallelism. What we want is performance. It is the fact that going to parallelism is the only way to continue to enhance performance that makes parallelism a necessity (Boeing Computer Services [47]).

In the proposed methodology, the programmer uses the design construction method to guide the specification of a software design for a problem's initial implementation on a parallel computer. Since the design is specific to a parallel computer, the software based on the design should execute quickly relative to its potential performance on the parallel computer. When the performance of a new, but dissimilar, parallel computer tantalizes the programmer, then a description of this parallel computer is substituted into the old design. Since most of the design is specific to the old parallel computer, the programmer uses the design construction method to modify the design so that the potential performance on the new parallel computer will be achieved. The methodology depends on specifying three fundamental components, which are described in the next section.

4.2 Design Components

The software design's primary components are the algorithmic specification, the mapping of the algorithm to a parallel computer, and the hardware characteristics of the parallel computer. Each primary component is composed of one or more secondary components which are unique to the primary component. A description of the design's primary and secondary components follows.

4.2.1 Algorithmic Specification

The algorithmic specification is a hardware-independent description of the problem . This description is composed of the data of the problem, the data dependencies, and the functions to be applied to the data. The algorithmic specification component must be able to express a broad range of algorithms. The language used to express the algorithmic specification must allow the description of the data at different levels of abstraction.

4.2.2 Hardware Characteristics

The hardware characteristics component is composed of several secondary design components: number of processors, the profile of processor speed to communication cost, ...etc. The number of processors is a hardware-dependent component and is quantified at least as good as or better than an order of magnitude estimate. A processor may be simple (e.g. the bit processors of the Connection Machine or the MasPar MP-1) or may be relatively complex (e.g. an Intel 80486, Intel iWarp chip, Motorola 68040). However, a processor is not envisioned to be a stand-alone computer networked with heterogeneous computers (e.g. Mentat system [32]). The processor speed/communication cost profile is also a hardware-dependent component. This profile describes a processor's cost to access data in memory relative to the its cost to perform computations using the data. Hierarchical memory and nonuniform memory access may require hierarchy-specific costs, thus the use of a profile instead of a ratio.

4.2.3 Mapping to a Parallel Computer

The specification of the mapping can be expressed at many levels of abstraction. At the highest level of abstraction, the mapping can be described by the type of parallelism to be exploited (e.g. Gelernter's result, agenda, or specialist parallelism). At the lower levels of abstraction, the programmer may describe the type of parallelism by specifying the data distribution and the work distribution in varying detail. The data distribution is a relation which maps a datum to a processor. The work distribution is a relation which maps a function instance to a processor. The data and the work can be described at several abstraction levels. The mapping must be expressed in terms of the algorithmic specification as its domain and the hardware characteristics as its range.

4.2.4 Originality of the Design Components

The definitions of the proposed components are similar to those in the DINO programming language. The DIstributed Numerically Oriented (DINO) parallel programming language generates software for distributed-memory multiprocessors [53]. DINO's parallel programming model supports specifying a program based on three components:

1. one or more virtual computers where the virtual computer is parameterized by the number of processors and either a vector or matrix communication topology (via the ENVIRONMENT statement),

2. a mapping specification of a globally-referenced distributed data structure for the virtual computer, and

3. work distribution specification based on the Single Programming Multiple Data (SPMD) programming model (via the COMPOSITE procedure).

This programming model abstracts the details of message passing, process management, and synchronization, all of which are handled by the DINO compiler.

DINO's virtual computer is more abstract than the proposed hardware component; however, the DINO programming model is specific to MIMD-DM parallel computers. DINO's mapping specification of the globally-referenced distributed data structure is similar to the data distribution component of the proposed mapping. DINO's work distribution specification is more restrictive (i.e.

SPMD) than the work distribution component of the proposed mapping. DINO's work distribution specification is composed of a description of the work to be performed and a description of how the work is mapped to the virtual computer. The proposed design has separated these two descriptions into the algorithmic specification and the mapping component respectively.

Other influential research included Dahlstrand's characterization of the information within a software application based on its portability [26]. Dahlstrand's characterization is an informal portability description and not a formal notation. An application is composed of more than just the source code. An application also includes the command control code (e.g. JCL), code of program-accessible libraries, and data files. There are three types of information in an application: algorithmic, optimizing, and environmental information. Algorithmic information can be made to be 100 percent portable. Optimizing information is provided by the programmer to help the compiler optimize the application's execution. This information is inherently less portable due to its computer dependence. Environmental information is inherently nonportable (e.g. user id and password).

Based on the three types of information, the proposed algorithmic specification is 100 percent portable, the proposed mapping component, categorized as optimizing information, is inherently less portable, and the proposed hardware components, categorized as environmental information, is inherently nonportable.

4.3 Design Attributes

A design attribute is a property or characteristic of one or more designs. There are two types of design attributes: intra-design and inter-design attributes. An attribute which characterizes only one design is called an intra-design attribute (e.g. portability of the design between parallel computers or the potential performance of a design's implementation on a parallel computer). An attribute which characterizes a relationship between more than one design is called an inter-design attribute (e.g. the equivalence of two designs).

4.3.1 Design Utility

Design utility is that which makes the design useful to the programmer. *Design utility* is an intra-design attribute based on one or more intra-design attributes. A programmer must choose one or more design attributes to be emphasized in the design. A design is most useful to a programmer when the design components have been specified to maximize or minimize these chosen design attributes. For example, the programmer might choose the design's portability among many parallel computers as a design attribute. One method to achieve this *design utility* is to design for an abstract computer which can be mapped to most parallel computers. For the proposed research, the *design utility* is based primarily on the performance of the implemented software. The *design utility* is achieved by specifying a mapping between the algorithmic specification and the hardware characteristics which leads to an implementation which executes quickly on a parallel computer.

4.3.2 Design Equivalence

Design equivalence is an inter-design attribute based on one or more intra-design attributes. Two or more designs are *equivalent* for an attribute when their intra-design attribute's values are equivalent. For example, if two designs are shown to be correct with respect to a requirement specification,

then their designs are equivalent for the design attribute correctness. The design equivalence for the correctness of two or more designs might be shown by comparing their implementation's input/output behavior. Another method would be to show their *design equivalence* for correctness axiomatically. Two or more designs might be equivalent with respect to their implementation's performance on a parallel computer.

4.4 Design Transformer

A design transformer modifies a design to produce a new design. A design transformer can be represented as a function. The design components serve as the function's domain and range. The transformer maps one or more components of an old design to one or more components of a new design.

The key property of a design transformer is that it creates a new design which is equivalent to the old design for one or more design attributes. The design attribute for the proposed work will likely be the input/output behavior of the design. If a design transformer for this design attribute creates a new design, then the old and new designs would be equivalent for their input/output behavior. Note that each design transformer will not necessarily enhance the *design utility* of the old design. Applying an inappropriate design transformer may decrease the utility of the design, or may have no effect on the design utility. The determination of the appropriate transformer is addressed in a subsequent section.

4.5 Design Construction Method

The purpose of the design construction method is to create a design with a high *design utility* for a specific parallel computer. The design construction method is composed of three steps: design assembly, design evaluation, and design prescription. A brief description of the design construction method for creating an application's initial implementation on a parallel computer is provided. A similar description for creating a design for an application's subsequent implementations on dissimilar parallel computers is also provided. Then a detailed description of the design construction method follows.

4.5.1 A Quick Tour

For an application's initial implementation on a parallel computer, the programmer begins by assembling a design. Typically, the programmer provides design information and then the *design utility* is evaluated. If the *design utility* meets the programmer's expectations, then the design construction method is terminated. If the *design utility* fails to meet the programmer's expectations, then a design prescription is created. This prescription will guide the programmer's attempt to enhance the *design utility* as the design is reassembled during the subsequent design assembly steps. This prescription may suggest that the programmer apply a design transformer to create a new, but equivalent, design or may suggest that the programmer respecify the design components using a respecification strategy. The programmer continues to use the design construction method until either the desired *design utility* is achieved, or the programmer's expectations are decreased.

For an application's subsequent implementations on dissimilar parallel computers, the programmer begins with the design assembly step as before. However, the design from the previous

implementation supplies the default values for the algorithmic specification and mapping component of the new design. The programmer replaces the hardware characteristics component of the old design with a description of the new parallel computer. Then the design utility is evaluated and the design construction method continues as in the initial implementation.

The design assembly step will be discussed as two cases (i.e. initial design assembly and subsequent design assembly) as illustrated in the following pseudocode: InitialDesignAssembly()
WHILE (DesignEvaluation()==UnsatisfactoryDesignUtility)
DO
  DesignPrescription()
  SubsequentDesignAssembly()
ENDWHILEDO
In the temporary place of a data flow diagram as a LaTeXfigure, the above psuedocode is expanded as:

> The programmer specifies the intra-design attributes which define the *design utility*, specifies the inter-design attributes which define the *design equivalence*, and specifies the problem in the programmer's native tongue. These definitions and specification remain unchanged for the duration of the design construction. When these constants are established, then the programmer proceeds with the design construction method as illustrated below:

```
design=InitialDesignAssembly()
designutility=DesignEvaluation( design )
WHILE ( designutility==UnsatisfactoryDesignUtility )
DO
  prescription=DesignPrescription( design, designutility )
  newdesign=SubsequentDesignAssembly( design, prescription )
  design=newdesign
  designutility=DesignEvaluation( design )
ENDWHILEDO
```

4.5.2 Initial Design Assembly

The only time during the design construction method when the design can be altered is during a design assembly step. In general, the programmer may assemble a design using two methods: specifying a design component or applying a design transformer to an old design. The programmer is encouraged but not restricted to employ only one instance of either component specification or transformer application between design evaluations.

The construction of a design begins with the Initial Design Assembly. Specifying the design components must initiate this step. At this early point in the design construction method, the selection of a design transformer must be based on the programmer's intuition since a design prescription is unavailable.

4.5.3 Design Component Specification during Initial Design Assembly

The design component specification step occurs in two parts. First, the programmer has the opportunity to modify the descriptions of the problem and the parallel computer. Since the programmer

is executing this step for this application's first time, then the programmer will be providing much design information. The algorithmic specification's secondary design components (i.e. data, data dependencies, and the functions applied to these data) are specified. The hardware characteristics are also specified.

The programmer has the opportunity to modify the mapping of the problem to the parallel computer. Mapping decisions for the proposed design methodology are based on the primary components algorithmic specification and hardware characteristics. In terms of the proposed methodology, many strategies found in the literature for mapping a problem to a parallel computer focus upon one aspect of the algorithmic specification, either data or work. A mapping may also be characterized as static or dynamic. Since the domain for the proposed mapping component is defined in terms of the algorithmic specification, then the abstraction level of the data, data dependencies graph, and functions partially defines a context for the description of the mapping. The range of the mapping component is defined in terms of the hardware characteristic.

4.5.4 Design Evaluation Method

The purpose of the design evaluation method is to determine the *design utility*. Based on the programmer's choice of intra-design attributes to define the *design utility*, the components of the design are evaluated and the result is a quantifiable value for the *design utility*. This value is not necessarily one number, but might be represented as a vector of data. Possible techniques for assessing the utility of the design include simulation, closed-form solutions, expert systems and neural nets.

The factors which are considered during the evaluation of the design may be specific to the design attributes which define the *design utility*. For our envisioned proof-of-concept, performance is the intra-design attribute which will define the *design utility*. Performance of a parallel application is influenced by the degree of load balance between the processors and by a processor's idle time as it waits for a data dependency to be satisfied. Incorporating a reasonable mapping component(which maps the algorithmic specification to the hardware characteristics), the implementation of the design should lead to a parallel application with good performance. This reasonable mapping component should be quantifiable as a mapping utility. Thus, when the design utility is defined for performance, then the evaluation of the mapping utility is one way to evaluate the design utility. The evaluation of the mapping utility is discussed below.

4.5.5 Mapping Utility Evaluation during Design Evaluation

The mapping utility is one way to measure *design utility* for the performance design attribute. A design with a high value for mapping utility should lead to software for a parallel computer which executes quickly relative to the potential performance of the computer. Mapping utility evaluation is currently composed of searching for violations of some rule-of-thumb generalizations. These rules focus upon maximizing the number of busy processors and minimizing the time required to satisfy their data dependencies in the algorithmic specification. Gelernter also focuses upon the amount of work and the number of processors during the evaluation of the programmer's choice of parallelism grain [14]. Some indications of an unsatisfactory mapping utility can be:

- when the number of processors grossly mismatches the parallelism of the data (a data-parallel perspective).

- when the number of processors grossly mismatches the parallelism of the work (a specialist parallelism perspective)
- when the potential data locality is ignored and the hardware characteristic communication-cost-to-computation-cost is large
- when the mapping introduces unnecessary control synchronization

4.5.6 Design Prescription Method

Using the design and the value of its utility, the design prescription step creates a design prescription. The prescription suggests the design assembly actions to enhance the *design utility*. To create this prescription, the set of design respecification strategies and the set of design transformers are considered with respect to their potential enhancement of the design utility. Then the prescription is composed of the design assembly actions with the greatest potential to enhance the *design utility*. A prescription may suggest that the design be respecified and provide a respecification strategy. A prescription may suggest that the design be transformed and suggest a transformer to apply to the design. The suggestion is to be based on a measurement of the potential enhancement of the design; however, this potential enhancement measurement process is not as thorough as the *design utility* measurement process performed during the design evaluation step.

There are several possible ways for the potential *design utility* enhancement to be measured. One way is to simulate or model an attempt to enhance the design using each respecification strategy or design transformer. The *design utility* of the resulting simulated design may be measured several ways. A gross measurement of the simulated design's utility might be made. Another way to measure the *design utility* of the resulting simulated design may be to only measure the altered components of the simulated design. The value of the potential is the difference in *design utility* measurements between the original and simulated designs. Kennedy and Fox are using neural networks to evaluate a Fortran D program's data distribution specifications upon its potential performance.

Another way for the potential *design utility* enhancement to be measured would be by diagnosing the reason for the unsatisfactory *design utility* and, knowing the strengths and weaknesses of the respecification strategies and design transformers, prescribe the best design assembly action to cure the ailing design.

The following text describes several respecification strategies and several design transformers.

4.5.7 Respecification Strategies

Respecification of the design components may lead to enhanced *design utility*. One respecification strategy focuses upon the abstraction level of the algorithmic specification. A second respecification strategy addresses the mapping component where the respecification can not be expressed as a design transformer. Note that a respecification strategy is suggested as a specific action, but, unlike the design transformer application, it is the programmer's responsibility to maintain *design equivalence* between the original design and the respecified design.

One of the strategies for respecification during the design assembly step is to alter the abstraction level of the algorithmic specification. The algorithmic specification can be described in more detail (less abstractly) by decomposing the data, the data dependencies graph, or the functions applied

to the data. Or this specification can be described in less detail by abstracting the aforementioned components. The new abstraction level of the problem provides the programmer with a new vocabulary to express a new mapping of the problem to a parallel computer. The purpose of the new description of the problem is to exploit a grain of parallelism which is more similar to the parallelism offered by the parallel computer. The respecification of the algorithmic specification will likely require re-expressing the mapping in terms of its new domain.

A less dramatic respecification strategy is to respecify the mapping of the design without altering the abstraction level of the algorithmic specification. While there may exist design transformers to create many new mapping components (similar to the graph decomposition research of McCreary and Gill [43]), it is likely that the programmer will have to generate some new mappings. This mapping opportunity may have always existed (i.e. the multiple grains of parallelism were always available to the programmer), but the abstraction level which leads to the highest *design utility* was not chosen during a previous design assembly step. If the mapping is viewed as a partitioning of the data dependencies graph, then the mapping can be respecified by increasing or decreasing the number of partitions, or by changing the shape of the partitions.

4.5.8 Prescribed Design Transformers

Applying a design transformer to a design may enhance the *design utility*. The current set of design transformers focus upon partitioning data or work of the algorithmic specification component among the processors of the hardware characteristics component. Two hypothetical examples of applying design transformers follow.

Hypothetically, the programmer has specified the design components, in particular, the algorithmic specification at an abstraction level, the number of processors for the hardware component and a mapping of the problem to the processors. Subsequently, the design evaluation step indicates an unsatisfactory *design utility*. If there exists a gross mismatch between the number of partitions in the mapping and the number of processors, then the design prescription step might choose a design transformer which alters the number of partitions. A subsequent design evaluation step might indicate that the unsatisfactory *design utility* remains. The resulting design prescription might suggest that the programmer needs to respecify the abstraction level of the algorithmic specification. However, as fate would have it, the desired level of abstraction had been previously defined. Then the transformer would revisit the abstraction level with the most potential to enhance the *design utility*.

Hypothetically, the programmer has specified the design components, in particular, the work distribution and the data distribution and communication cost/computation cost profile for the Intel iPSC/2 parallel computer (a hypercube communication topology where each node is an Intel 80386 processor). We assume that a satisfactory *design utility* is achieved and software development proceeds using the design. But that is not the end of the story. Two years later, the programmer gains access to an Intel iPSC/860 parallel computer (the same hypercube communication topology where each node is a much faster Intel 80860 processor). The programmer substitutes the communication cost/computation cost profile for the new parallel computer into the design. The design evaluation step indicates an unsatisfactory *design utility* and the communication cost/computation cost profile is the only difference between the designs. The design prescription might suggest a design transformer which respecifies the work distribution or data distribution, emphasizing data locality for the computations.

4.5.9 Subsequent Design Assembly

During the initial design assembly step, the programmer is encouraged to assemble the design by component specification but discouraged from assembling the design by design transformation. However, during the subsequent design assembly steps, the programmer is provided with a design prescription which suggests the design assembly actions to perform. This prescription may suggest that the programmer respecify the design components based on a respecification strategy or apply a design transformer to the design. These design assembly actions are discussed below.

4.5.10 Component Respecification during the Subsequent Design Assembly

During the initial component specification step, the programmer is encouraged to specify as much design component information as possible. During the subsequent component specification step, the programmer is guided to respecify the components prescribed by the design prescription. The design prescription provides this guidance by suggesting a respecification strategy . If the algorithmic specification or the hardware characteristics components have been changed during the current component specification step, then the mapping component must be examined (and altered if necessary) to insure that the mapping is expressed in terms of the current algorithmic specification and the hardware characteristics components.

4.5.11 Design Transformation during the Subsequent Design Assembly

Since a design prescription does not exist during the initial design assembly step, the programmer must rely on intuition to select a transformer. However, during the subsequent design assembly steps, the design prescription guides the programmer by suggesting a design transformer to apply to the design.

5 Proposed Tasks

Two tasks are proposed.

1. Provide a methodology for designing software for parallel computers. This task is composed of the following six subtasks.
   (a) State the design's representations for the problem, the parallel computer, and the mapping of the problem to the parallel computer.
   (b) Describe the steps which the programmer performs to compose a software design for a parallel computer.
   (c) Define the term *design equivalence* and find a technique to measure the *design equivalence* between two designs.
   (d) Define the term *design utility* and find a technique to measure the utility of a design.
   (e) Find a set of design transformers where each design transformer creates a new design from an old design, and maintains the *design equivalence* between the two designs.
   (f) Find respecification strategies where each respecification strategy creates a new design from an old design.
2. Apply the design methodology to a realistic problem as a proof-of-concept. This task is composed of three subtasks.
   (a) Choose a realistic problem.
   (b) Create computer-specific designs for several parallel computers using the design methodology.
   (c) Evaluate the designs and the methodology.

5.1 Design Methodology Task

The first task is to provide a methodology for designing software for parallel computers. This task is composed of six subtasks.

The first subtask is to state the design's representations for the problem, the parallel computer, and the mapping of the problem to the parallel computer. For each representation, the syntax and semantics of the representation's language must be formally described.

The second subtask is to describe the steps which the programmer performs to compose a software design for a parallel computer. A significant portion of this subtask has been provided in a previous section of this proposal.

The third subtask is to define the term *design equivalence* and find a technique to measure the *design equivalence* between two designs. The *design equivalence* for the proposed work will likely be based on the input/output relationship design attribute.

The fourth subtask is to define the term *design utility* and find a technique to measure the utility of a design. The *design utility* for the proposed work will likely be based on the performance design attribute.

The fifth subtask is to find a set of design transformers where each design transformer creates a new design from an old design, and maintains the *design equivalence* between the two designs. The properties of the set should be considered.

The sixth subtask is to find respecification strategies which create new designs which cannot be created by a design transformer. The degree of formality of the respecification strategies and their relationship with the design equivalence should be considered.

5.2 Proof-of-Concept Task

The second task is to apply the design methodology to a realistic problem as a proof-of-concept. The problem which has been chosen is the National Aeronautics and Space Administration (NASA) Clouds and the Earth's Radiant Energy System (CERES) cloud retrieval algorithm [8, 57]. The NASA-Langley members of the cloud retrieval algorithm group (Dr. Eric Schmidt, Dr. Bryan Baum, and Dr. Bruce Wielicki (project leader)) have provided preliminary algorithmic information which suggests potential implementations at several grains of parallelism. Several designs using this problem will be created. The computer-specific designs will be evaluated on several fronts. One front will focus upon a formal evaluation of the designs. Another front will focus upon software which is based on the created designs.

References

[1] Active Memory Technologies, 65 Suttons Park Avenue, Reading, UK. *DAP Series AMT Technical Overview*, October 1989.

[2] Fran Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis for multiprocessing. In E. N. Houstis, T. S. Papatheodorou, and C. D. Polychronopoulos, editors, *Supercomputing: First International Conference Athens, Greece Proceedings*, Lecture Notes in Computer Science, pages 194–211. Springer-Verlag, June 1987.

[3] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* Computer Science and Engineering. Benjamin/Cummings, Redwood City, CA, first edition, 1989.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. LAPACK Working Note 20 CS-90-105, Computer Science Department, University of Tennessee, Knoxville, TN 37996, May 1990.

[5] E. Anderson, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, S. Hammarling, and W. Kahan. Prospectus for an extension to LAPACK: A portable linear algebra library for high-performance computers. LAPACK Working Note 20 CS-90-118, Computer Science Department, University of Tennessee, Knoxville, TN 37996, November 1990.

[6] Edward Anderson and Jack Dongarra. Results from the initial release of LAPACK. LAPACK Working Note 16 CS-89-89, Computer Science Department, University of Tennessee, Knoxville, TN 37996, November 1989.

[7] Vasanth Balasundaran, Ulrich Kremer, Ken Kennedy, Kathryn McKinley, and Jaspal Subhlok. The ParaScope editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, pages 540–550. IEEE Computer Society and ACM SIGARCH, ACM Press, November 1989.

[8] Bruce R. Barkstrom. *CERES Data Analysis Algorithm Plan (Addendum 3, Version 1.0).* Atmospheric Sciences Division, NASA Langley Research Center, Hampton, VA 23665-5225, August 1990.

[9] Adam Beguelin, Jack J. Dongarra, G. A. Geist, Robert Manchek, and V. S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing '91*, pages 435–444. ACM and IEEE, IEEE Computer Society Press, November 18-22, 1991.

[10] Walt Brainerd. FORTRAN 77. *Communications of the ACM*, 21(10):806–820, October 1978.

[11] Orlie Brewer, Jack Dongarra, and Danny Sorensen. A graphics tool to aid in the generation of parallel FORTRAN programs. In *COMPSAC 89: Proceedings of the Thirteenth Annual International Computer Software and Applications Conference*, pages 89–93. IEEE Computer Society Press, September 20-22, 1989.

[12] James C. Browne. Understanding execution behavior of software systems. *I.E.E.E. Computer*, 17(7):83–87, July 1984.

[13] Curt Burmeister. ParallelFortran:X3H5/91-0023-d. Internal Committee Document(07-23-91).

[14] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–358, September 1989.

[15] Centre for Parallel Computing, QMW University of London. *DAP Fortran-Plus*, 1990.

[16] K. Mani Chandy. Programming parallel computers. In Howard E. Sturgis, editor, *Proceedings of the 1988 International Conference on Parallel Processing, Volume II: Software*, pages 314–321, University Park and London, August 15-19, 1988. The Pennsylvania State University, The Pennsylvania State University Press.

[17] K. Mani Chandy and Carl Kesselman. Parallel programming in 2001. *I.E.E.E. Software*, 8(6):11–20, November 1991.

[18] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[19] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Vienna Fortran–a Fortran language extension for distributed memory multiprocessors. ICASE Report 91-72, Institute For Computer Applications In Science and Engineering, NASA Langley Research Center, September 1991.

[20] Alex L. Cheung and Anthony P. Reeves. The Paragon multicomputer environment: a first implementation. Technical Report EE-CEG-89-9, School of Electrical Engineering, Cornell University, July 1989.

[21] Longchyr Chang and Brian T. Smith. Classification and evaluation of parallel programming tools. Technical Report CS90-22, Department of Computer Science, University of New Mexico, November 1990.

[22] X3H5 Technical Committee. Parallel extensions for Fortran 77, X3H5 language binding. [X3H5/91-0040-C](07-24-91).

[23] Intel Scientific Computers. *iPSC/2 and iPSC/860 Programmer's Reference Manual.* Intel Corporation, Beaverton, Oregon, revision 003 edition, June 1990.

[24] Intel Scientific Computers. *iPSC/2 and iPSC/860 User's Guide.* Intel Corporation, Beaverton, Oregon, revision 006 edition, June 1990.

[25] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Experiences using control dependence in PTRAN. In David Gelernter, Alexander Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 186–212. MIT Press, 1990.

[26] Ingemar Dahlstrand. *Software Portability and Standards*, volume 27 of *Computers and their Applications.* Ellis Horwood Limited, Chichester, West Sussex, England, first edition, 1984.

[27] John Darlington, Mike Reeve, and Sue Wright. Declarative languages and program transformation for programming parallel systems: A case study. *Concurrency: Practice and Experience*, 2(3):149–169, September 1990.

[28] Jack Dongarra, Danny Sorensen, and Orlie Brewer. Tools and methodology for programming parallel processors. In M. H. Wright, editor, *Proceedings of the IFIP WG 2.5 Working Conference on Aspects of Computation on Asynchronous Parallel Processors*, pages 125–137. North-Holland, 1989.

[29] H. Allan Fencl and Chua-Huang Huang. On the design of parallel algorithms for non-linear space-time representations. In *Proceedings of International Computer Symposium, Volume 2*, pages 457–464, December 17-19, 1990.

[30] H. Allan Fencl and Chua-Huang Huang. On the synthesis of programs for various parallel architectures. Technical Report OSU-CISRC-10/90-TR-32, Department of Computer and Information Science, Ohio State University, October 1990.

[31] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. FORTRAN D language specification. Rice COMP TR90-141, Department of Computer Science, Rice University, December 1990, revised February, 1991.

[32] Andrew S. Grimshaw. A software environment for high-performance parallel computing. In *Proceedings of the 1991 Minnowbrook Workshop on Software Engineering for Parallel Computing*, July 16-19, 1991.

[33] Language Working Group. A review and analysis of Fortran 8x. Technical Report 87-40, Mathematics and Computer Sciences Division, Argonne National Laboratory, October 1987.

[34] Mark D. Guzzi, David A. Padua, Jay P. Hoeflinger, and Duncan H. Laurie. Cedar Fortran and other vector and parallel Fortran dialects. In *Proceedings of Supercomputing '88*, pages 114–121. IEEE Computer Society and ACM SIGARCH, IEEE Computer Society Press, November 1988.

[35] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[36] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[37] D.J. Hunt. *AMT DAP-a procesor array in a workstation environment.* Active Memory Technologies, 65 Suttons Park Avenue, Reading, UK, vol 4 no 2 edition, April 1989.

[38] American National Standards Institute. *American National Standard for Information Systems Programming Languages Fortran.* June 1989.

[39] Harry F. Jordan, Muhammad S. Benten, Gita Alaghband, and Ruediger Jakob. The Force: A highly portable parallel programming language. In *COMPSAC 89: Proceedings of the Thirteenth Annual International Computer Software and Applications Conference*, pages II–112 – II–117. IEEE Computer Society Press, September 20-22, 1989.

[40] Alan H. Karp. Programming for parallelism. *I.E.E.E. Computer*, 20(5):43–57, May 1987.

[41] Alan H. Karp and Robert G. Babb II. A comparison of 12 parallel Fortran dialects. *I.E.E.E. Software*, 5(2):190–222, September 1988.

[42] David B. Loveman. Digital Equipment Corporation high performance Fortran proposal. Presented at the Rice University High Performance Fortran Forum, January 27, 1992.

[43] C. McCreary and H. Gill. Automatic determination of grain size for efficient parallel processing. *Communications of the ACM*, 32(9):1073–1078, September 1989.

[44] Piyush Mehrotra and John Van Rosendale. Programming distributed memory architectures using Kali. ICASE Report 90-69, Institute For Computer Applications In Science and Engineering, NASA Langley Research Center, October 1990.

[45] Michael Metcalf and John Reid. *Fortran 90 Explained.* Oxford Science Publications, 1990.

[46] David A. Padua and Micheal J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[47] Cherri M. Pancake. Software support for parallel computing: Where are we headed? *Communications of the ACM*, 34(11):53–64, November 1991.

[48] Cherri M. Pancake and Donna Bergmark. Do parallel languages respond to the needs of scientific programmers? *I.E.E.E. Computer*, 23(12):13–23, December 1990.

[49] Douglas M. Pase. Cray Research Inc. MPP Fortran programming model, draft 1.2. Presented at the Rice University High Performance Fortran Forum, January 2, 1992.

[50] Constantine D. Polychronopoulos, Milind B. Girkar, Mohammad R. Haghighat, Chia L. Lee, Bruce P. Leung, and Dale A. Schouten. The structure of Parafrase-2: an advanced parallelizing compiler for C and Fortran. In David Gelernter, Alexander Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 423–453. MIT Press, 1990.

[51] Anthony P. Reeves. Parallel Pascal: An extended Pascal for parallel computers. *Journal of Parallel and Distributed Computing*, 1:64–80, August 1984.

[52] Anthony P. Reeves. Paragon: a programming paradigm for multicomputer systems. Technical Report EE-CEG-89-3, School of Electrical Engineering, Cornell University, January 1989.

[53] Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver. The DINO programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September 1991.

[54] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–311, 1990.

[55] Brian T. Smith. Parallel computing forum(PCF) Fortran. In M. H. Wright, editor, *Proceedings of the IFIP WG 2.5 Working Conference on Aspects of Computation on Asynchronous Parallel Processors*, page 235. North-Holland, 1989. abstract only.

[56] K. Sridharan, M. McShea, C. Denton, B. Eventoff, J. C. Browne, P. Newton, M. Ellis, D. Grossbard, T. Wise, and D. Clemmer. An environment for parallel structuring of FORTRAN programs. In Emily C. Plachy and Peter M. Kogge, editors, *Proceedings of the 1989 International*

Conference on Parallel Processing, Volume II: Software, pages II–98 – II–106, University Park and London, August 8-12, 1989. The Pennsylvania State University, The Pennsylvania State University Press.

[57] CERES Data Management Team. *Clouds and the Earth's Radiant Energy System (CERES) Data Management Plan.* NASA Langley Research Center, Hampton, VA 23665-5225, June 1990.

[58] Thinking Machines Corporation, Cambridge, Massachusetts. *Getting Started in CM Fortran*, version 5.2-0.6 edition, November 1989.

[59] Tim Wise. private communication, April 1992. Scientific and Engineering Software, Inc.

[60] Janet Wu, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. ICASE Report 91-13, Institute For Computer Applications In Science and Engineering, NASA Langley Research Center, January 1991.

[61] Steven Ericsson Zenith. A rationale of programming with Ease. lecture note from a NASA ICASE lecture, May 1, 1991.