Creating Portable and Automatically Scalable Parallel Software Using the PARSA™ Programming Methodology

Vijay Murthi+, David Levine+, Jeff Marquis+ and Behrooz Shirazi+
+Department of Computer Science & Engineering
The University of Texas at Arlington
+Prism Parallel Technologies, Inc.
{vijay, levine, shirazi} @cse.uta.edu and marquis@prismPTI.com

Abstract

Creating portable and automatically scalable parallel software has been a goal for researchers and practitioners since the advent of parallel computing. In this paper we present a programming methodology that reduces parallel programming complexity, while creating portable and automatically scalable parallel software. To support this methodology two separate tools have been developed – the PARSA Software Development Environment and an accompanying thread manager.

The development environment addresses programming issues via an object-based graphical programming methodology that transforms a project automatically into a portable and scalable source code. Generated source code makes calls to the user-level thread manager, which manages the run time execution of the parallel software. Two sample applications that contain various forms of parallelism have been developed and are compiled on three different systems with diverse native threading mechanisms to demonstrate portability. Finally, the automatic scalability is demonstrated with the run time performance of the applications on multiprocessor systems.

1. Introduction

Multi-threaded software is an effective way to use the resources of shared memory computer systems. Applications can spawn threads to execute concurrently, where the system has multiple processors and the threads maybe distributed to execute on different processors. Further, applications can spawn little more threads than the system has processors without significant performance degradations because threads efficiently share resources. Data can be passed to threads in an efficient manner utilizing the shared memory system.

To deploy threading using traditional directives-based methods, programmers must hand code threading directives into their source code for calls to thread libraries [1],[2],[3]. This naturally drives up development costs and time. It also requires significant amount of programming expertise. For example, the POSIX Pthreads library has approximately 100 different directives that provide programmers with a great deal of programming flexibility, but the sheer size of the library prohibits most programmers from learning the most efficient threading mechanisms to use. Another programming approach is the compiler-directives approach, such as OpenMP [4],[5], like Pthreads, requires programmers to embed parallelization directives into their application source code. Parallel programming language approaches, such as Cilk [6] and HPF [7] require programmers to learn a new language in order develop parallel applications. These approaches are representative of many parallel programming methodologies in that they are unduly expensive in terms of software development cost, development time, and programmer expertise required.

While each of these approaches claim support for portability by adhering to their particular standard, the reality is that true portability is often mythical. A common scenario is source code is compilable for different systems, but the run time performance will vary widely between systems due to differences in the underlying threading and processor models employed [8]. For example, an application developed using Pthreads for a Linux system may be efficient on Linux systems but will have poor performance when run on a Solaris system, which uses a concurrency threading model [2],[3]. While vendors may claim that performance differences are due to their superior threading and processor models this does little to help programmers who are simply trying to support their software on multiple systems. Some methodologies, such as Cilk, are not supported on all systems, for example Microsoft Windows.
A different approach warrants consideration in today’s environment where programmers and special tools are at a premium and the ability to support software on a range of systems is a requirement. In this paper we present the PARSA programming methodology that overcomes many of the problems of other parallel programming approaches. Though this methodology supports both parallel programming for shared memory systems and distributed systems [9], this paper focuses on parallel programming on shared memory systems.

The goals of this methodology are to i.) make parallel programming little difficult to develop than sequential programming, ii.) make parallel software developed portable across a range of systems, and iii.) the resultant software performance is efficient even when ported to systems with different underlying threading models. Two tools – the PARSA Software Development Environment (SDE) and the ThreadMan™ Thread Manager – are tightly coupled to facilitate the methodology and produce the stated benefits.

Programmers are removed from the low-level details of parallel programming by i.) supporting a graphical programming methodology and ii.) using automated source code generation techniques to create the code for applications to execute in parallel.

An additional level of abstraction between programming issues and deployment issues facilitate portability and ensures that parallel software executes efficiently across a wide range of systems. This level of abstraction separates programming issues in the SDE from deployment issues in the thread manager. The interface between coding and thread management is presented in sufficient detail to demonstrate this level of abstraction works in practice.

Two sample applications – matrix multiplication and merge sort – are presented to demonstrate different forms of parallelism. The run time analysis shows that the methodology and the tools produce efficient, automatically scalable and portable parallel software.

2. Programming Methodology

A unique graphical programming method allows multi-threaded (i.e., parallel) software to be developed quickly and easily. Projects developed consist of graphical objects and arcs. When a new project is being developed, graphical objects are added. Each object represents a project task, the interface the object has with other objects in the project must be defined. The interface consists of standard variable declarations that are declared in the same manner that variables are declared in standard programming languages (e.g., C, C++ and Fortran); the interface defines the “contract” a graphical object has with other graphical objects in a project, which have an INPUT interface and an OUTPUT interface [10].

Graphically, objects appear as icons. For each INPUT variable a port appears along the top edge of the icon, and similarly for each OUTPUT variable a port appears along the bottom edge of the icon. Hence, the interface properties of an object are represented graphically as ports on the icon. An example graphical object icon with 3 INPUT ports and 2 OUTPUT ports is shown in Figure 1.

![Figure 1: A task graphical object](image1)

Different types of graphical objects allow different types of parallelism. Regardless of graphical object type the interface appears the same. A task graphical object is simply a task to be performed within a project. This task is programmed by the programmer in a standard programming language. Note that this does not require programmers to program threading directives into their projects. After the project is finished the code generator is invoked to create threading directives needed for the project to execute safely in parallel.

A graphical object can be programmed to perform any task provided that i.) the code will compile and ii.) the variables referenced in the code are either declared in the graphical object's interface or declared and allocated in global memory. Semantically, each object is an independent, schedulable entity that will be spawned as a thread at run time. Multiple objects (i.e., threads) can be spawned to execute concurrently. Therefore, objects must execute without adversely affecting the execution of other objects that may also be executing.

![Figure 2: A project in PARSA.](image2)
It is intended for objects to be programmed without reference to the target system (including threading directives). That is, the task performed by an object should be generic code and not include specific system code. There are two important reasons for this: Objects are intended to be computational building blocks that can be re-used within a project and in different projects. Second, projects can be easily ported to a range of systems.

The interaction between graphical objects is shown with arcs. Arcs are lines connecting a desired OUTPUT port to a desired INPUT port. Semantically, however, arcs represent data “being passed” from a source object to a destination object.

The source code generator produces the data structures and code needed to pass data between threads. Figure 2 shows a project with arcs showing the relationship between graphical objects.

2.1. Execution Model

The execution model is: An object is eligible for execution when all of its INPUT data is available. In other words, a graphical object cannot execute until all the graphical objects that it is dependent upon have data finished executing. If it has no INPUT variables declared, then it is eligible for execution when the project begins executing. This execution model the representation of a project implicitly defines the order of execution of the objects.

The project shown in Figure 2 will execute as: The graphical object named inputGo has no INPUT ports, and therefore can execute when the project begins executing. Once inputGo has finished executing Go1 and Go2 are eligible for execution. If the system executing the project has multiple processors then Go1 and Go2 can execute concurrently in parallel on different processors, and likely will. If the system has a single processor then Go1 and Go2 will share the processor’s cycles until each completes execution. We assume the system has multiple processors and Go1 and Go2 will execute concurrently on different processors.

If Go1 executes for a relatively long time compared to Go2, then Go2 will finish executing before Go1. In this case, the OUTPUT data generated by Go2 will be available for Go3 and Go4. However, Go3 and Go4 cannot begin executing because they are dependent on data generated by Go1. Once Go1 finishes executing Go3, Go4 and Go5 are eligible for execution, and they will be spawned as threads. When Go3 and Go4 finish executing Go6 can then execute, and when Go5 finishes executing then Go7 can execute. Finally, when Go6 and Go7 finish executing outputGo can execute.

If, on the other hand, Go2 executes for a relatively long time compared to Go1, then Go1 will finish executing before Go2. In this case, the OUTPUT data generated by Go1 will be available for Go3, Go4 and Go5. Because Go5 is dependent only on data generated by Go1 it is eligible for execution when Go1 finishes executing, and Go5 will be spawned to execute while Go2 is still executing. When Go5 finishes executing then Go7 can begin and execution continues.

Note that high-level project analysis can be performed without getting mired in low-level programming details, which makes software maintenance easier because maintenance personnel can easily perform a high-level analysis to familiarize themselves with a project. Note that also programmers need not concern themselves with the relative execution times of the objects. The execution model enforces the data dependencies specified with the project’s arcs and ensures the objects will execute in the proper order.

2.2. Source Code Generator

Once a project has been fully programmed the representation is converted into parallel multi-threaded source code by the source code generator. The programmer-generated code is used as the basis of the code produced and is augmented with all structure declarations, data passing code, threading directives and code needed by the thread manager.

3. Thread Manager

The thread manager is an integral component of the methodology, allowing programmers to generate the code that performs the tasks of a project without concern for how the project will execute in parallel at run time.

The source code contains calls to functions embedded in the thread manager that are used to manage the execution of parallel software. As Figure 2 demonstrates, the execution of parallel software is dynamic that threads are spawned for execution based on the run time characteristics of the objects and their precedence relationships. Parallel software managed efficient that only those threads that are eligible to execute are spawned to execute. When each thread finishes executing its resources are freed and can be reused.

3.1. Parallelism

As described above, task objects can be programmed to perform any programmer-defined task within a parallel software project. As such, the granularity of task objects can vary from a simple piece of code to a complex and sophisticated routine. Hence, task objects allow
programmers to exploit irregular parallelism [11],[12] in their software projects.

Task objects can be programmed to perform asynchronous tasks (such as accessing a remote database) that respond in an asynchronous and non-deterministic manner. Task graphical objects are developed to perform asynchronous operations and are said to be asynchronously parallel.

Because many large-scale commercial and scientific applications spend a large percentage of their time executing loops, two loop intrinsic graphical objects are developed to improve the execution time of looping structures – the forall graphical object and the while graphical object which support regular parallelism [13] and repeat parallelism [12], respectively.

\[
\begin{align*}
&\text{for } (i = 0; i < \text{num\_rows\_of\_A}; i++) \\
&\text{for } (j = 0; j < \text{num\_columns\_of\_B}; j++) \\
&\text{for } (k = 0; k < \text{num\_rows\_of\_B}; k++) \\
&\quad C[i][j] += A[i][k] \times B[k][j];
\end{align*}
\]

**Figure 3: Sequential C code for matrix multiplication.**

Regular parallelism is a common form of parallelism found in many commercial and scientific applications. A classic example application that contains regular parallelism is matrix multiplication, \(C = AXB\). This can be easily programmed in the C programming language as shown in Figure 3.

Notice each iteration of the outer for loop is independent of all other iterations of the loop. That is, there are no dependencies between successive iterations of the outer for loop. As such, each iteration of the outer for loop can be calculated independently. This implies that each iteration of the loop can be calculated concurrently, and is known as regular parallelism. Figure 4 shows the semantic representation of matrix multiplication with regular parallelism.

![Figure 4: Semantic representation of matrix multiplication.](image)

Notice that \(t_0\) is a control thread that spawns threads \(t_1\), \(t_2\), and \(t_3\). Thread \(t_1\) contains regular parallelism, and it spawns threads \(t_{11}\) through \(t_{1n}\) which correspond to the \(n\) loop iterations that execute in parallel. When the \(n\) threads of \(t_1\) have finished executing control returns to \(t_0\) completing the execution of the loop structure.

The merge sort application contains irregular parallelism, regular parallelism and repeat parallelism. Figure 5 shows the semantic representation of the merge sort algorithm [14].

Notice that \(t_0\) is the control thread that spawns threads \(t_1\), \(t_2\), \(t_3\), \(t_4\), and \(t_5\). Thread \(t_1\) contains regular parallelism and it spawns threads \(t_{11}\) to \(t_{1n}\) in the same manner described above for matrix multiplication. Threads \(t_2\) and \(t_3\) exhibit irregular parallelism, and they can execute concurrently. Thread \(t_4\) contains repeat parallelism, within the looping structure of \(t_0\) threads \(t_1\) and \(t_2\) are spawned.

This is similar to a while loop in the C programming language, but any parallelism contained within the loop body can be exploited each time the loop executes to improve the run time performance of the looping structure. In this case notice that thread \(t_6\) contains regular parallelism, and threads \(t_{6.1}\) to \(t_{6.n}\) will be spawned for each iteration of the looping structure contained within \(t_4\).

Notice that different forms of parallelism are included in a single project using a consistent methodology. Repeat
parallelism is hierarchical in that while graphical objects encapsulate other graphical objects that can contain parallelism. These features are available to programmers without the need to create the data passing or the control code required to allow these forms of parallelism to be exploited.

The source code generator produces parallel source code that execute according to the semantic representation of projects, and the generated source code is dependent on the thread manager to control the execution of parallel software. The thread management is described in the next section.

3.2. Thread Management

ThreadMan is a dynamic linkable library that manages the execution of the developed parallel software. ThreadMan is a component that i.) eliminates the need for programmers to generate code that controls the run time execution of their parallel software projects, ii.) ensures parallel software executes according to the execution model and iii.) makes the generated parallel source code portable across a wide range of hardware platforms and supported operating systems. The parallel source code produced is fully compatible with the functions defined in the Application Programming Interface (API) [15].

ThreadMan is built on top of native threading mechanisms supported by diverse parallel systems as shown in Figure 6. Specifically, the thread manager relies on the native user-level thread libraries provided with those operating systems. System-specific versions of ThreadMan have been developed that are optimized to exploit the most efficient threading mechanisms supported on each system. Figure 6 shows ThreadMan for the three systems used in this paper to demonstrate portability, but ThreadMan is available other systems as well.

While, programmers are abstracted from the vagaries of threading mechanisms, ThreadMan provides the benefit of allowing a single set of generated parallel source code to be compiled on a range of systems. This feature allows programmers to support a single set of source code on many different systems reducing source code management and maintenance costs.

ThreadMan consists of two major components: the scheduler (TMS in Figure 6) and the portable interface (TMP). The role of the scheduler is to manage the execution of application threads according to the execution model. This involves spawning threads according to the precedence relationship between objects and tracking the progress of threads as they execute. The portable interface is a lightweight interface used by the scheduler to access the native user-level thread library.

Figure 7 shows how ThreadMan works when managing matrix multiplication. Notice that thread \( t_2 \) exhibits regular parallelism and results in multiple threads being spawned at run time. The threads are spawned and scheduled by the thread manager based on the precedence relationship as shown by Figure 4.
The chosen algorithms for each application were developed in sequential C and in the PARSA version 2.0 for the C programming language. The thread manager manages the run time execution of the parallel implementation on each of the respective systems. The sequential and parallel implementations are algorithmically identical and the same in how data structures are referenced to ensure the run time comparison is fair.

We conducted the run time analysis on three different systems chosen for the differences in their underlying threading and processor models. The first system is a two-processor Sun E3000 system running Solaris 2.6; the second system is a two-processor Intel-based system running Linux 6.2; and the third system is the same two-processor Intel-system booted to run Microsoft Windows NT 4.0 Service Pack 6a.

The parallel implementation was developed in the PARSA SDE on the Solaris system, the source code generator was invoked to produce the single set of parallel source code used on all three systems; the generated source code was copied to the Linux and Windows machines for compilation without modification. The same compiler was used to compile the sequential and parallel source code on each of the target systems to eliminate disparities between different compilers.

### 4. Run Time Performance Analysis

This methodology is demonstrated on the two sample applications matrix multiplication and merge sort. We expect matrix multiplication to be very scalable since it is a very uniform application (as demonstrated by its regular parallelism). The expected performance for the merge sort algorithm is not nearly as obvious, since it is not as straightforward when implemented in parallel. Further, it is important to demonstrate that each of the applications can be ported to different systems and retain their efficiency. The following subsections provide the run time analysis performed on each of these algorithms validating that this methodology produces efficient and portable parallel software.

![Figure 8: Merge Sort managed by ThreadMan.](image)

Figure 8 shows the relationship between ThreadMan and the threads of the merge sort application. Notice that the thread manager is invoked multiple times to manage the execution of the hierarchical levels of parallelism contained in the project. Specifically, a new invocation of ThreadMan has been initiated to manage the thread $t_2$ which exhibits repeat parallelism. A new invocation is required to ensure the non-blocking execution of other threads when thread $t_2$ is executing. There can be multiple invocations of the thread manager within the same project depending on the hierarchy contained within the project.

#### 4.1. Matrix Multiplication

The matrix multiplication application multiplies two randomly generated square matrices of integers. The input matrices are randomly generated before the matrix multiplication is performed. Also, the dimensions of both input matrices are $n$ by $n$, so the result matrix is also of size $n$ by $n$. Multiple tests were conducted with $n$ varying from 400 to 1500 to demonstrate application scalability.

Figures 9a, 9b, and 9c show the run time performance comparison between the sequential and parallel implementation of matrix multiplication on the three test systems. Note the scale in Figures 9a, 9b, and 9c are logarithmic. Figure 9a, 9b, and 9c shows that the parallel implementation runs nearly twice as fast as the sequential implementation on all three test systems.
c. Microsoft Windows NT system.

d. Normalized performance comparison.

Figure 9: Run time analysis between sequential and parallel versions of matrix multiplication.

Figure 10 shows a normalized performance comparison between the systems in terms of speedup obtained by the parallel implementation of matrix multiplication. The speedup data points for the lines of Figure 9d were calculated by dividing the sequential execution times by the parallel execution times to normalize the performance on each system. Figure 9d shows matrix multiplication executes at very nearly the maximum theoretical speedup of 2 on the Solaris and Linux systems with slightly lower performance on the Windows system.

4.2. Merge Sort

The merge sort application sorts a randomly generated list of integers. The algorithm chosen requires the list size to be an even power of 2. Multiple tests were conducted with the size of the list to be sorted ranging from 16k to 1,024k.

Figure 10 shows the run time performance comparison between the sequential and parallel implementation of merge on the three test systems. Again, the scale in the figure 10a, 10b and 10c are logarithmic. As Figures 10a, 10b, and 10c show the parallel implementation runs nearly twice as fast as the sequential implementation on all three test systems.

Figure 10d shows a normalized performance comparison between the systems in terms of speedup obtained by the parallel implementation of merge sort on the three systems. The speedup data points for the lines of Figure 10d were calculated similar to Figure 9d. As Figure 10d shows merge sort executes slightly above the twice theoretical speedup (due to operating system artifacts) on the Linux system with slightly below 2 times the performance on the Solaris and Windows systems, demonstrating that this methodology produces portable source code that attains near linear scalable performance regardless of the target system that executes the software.

4.3. Summary and Conclusions

In this paper the PARSA programming methodology was presented. The paper began with a brief description of problems associated with traditional parallel programming
approaches. Our methodology was then shown to obviate some of those problems by i.) abstracting programmers from low-level parallel programming issues and ii.) abstracting programming issues from deployment issues.

Two tools were presented in detail. The Software Development Environment was shown to support an object-based, graphical programming methodology that allows programmers to develop parallel software without concern for how parallel software executes at run time. The source code generator was shown to produce all source code required to allocate data passing structures, to pass data between threads, and to manage the parallel execution of software projects.

The second tool the thread manager was shown to i.) eliminates the need for programmers to generate code that controls the run time execution of their parallel software projects, ii.) ensures parallel software executes according to the execution model and iii.) makes generated parallel source code portable across a range of hardware platforms and operating systems.

A run time performance analysis of two sample applications was presented that shows the methodology removes many of the problems with traditional parallel programming methodologies while producing significant performance gains.

In conclusion our methodology may be applied to more programming projects, that can benefit from improved run time performance because i.) parallel software can be developed with little more effort and of programming expertise as required for sequential programming, ii.) parallel software can be easily ported to a wide range of systems, and iii.) the performance of parallel software will be significantly faster.

References


