A Tool Based Methodology for Development of Automatically Scalable and Reusable Parallel Code

Vijay Murthi+, David Levine+, Behrooz Shirazi+ and Jeff Marquis±
+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

Program performance may be improved by efficiently programming some key sections of the software. In this paper, we present a methodology for converting selected portions of source code into automatically scalable multithreaded routines, without forcing programmers to concentrate on parallel programming issues. These developed routines can be reused across various projects, operating systems and system architectures. To support this methodology two separate but tightly coupled tools have been developed - PARSA(TM) Software Development Environment (SDE) and the ThreadMan(TM) Thread Manager.

The SDE addresses programming issues by allowing a graphical object based approach to develop multithreaded routines that abstracts the users from parallel programming. ThreadMan manages the software developed using SDE. ThreadMan is a user-level thread manager that automatically spawns and schedules threads at runtime.

Two examples have been developed using this methodology to demonstrate that there is virtually no degradation in performance when compared to sequential code, in a single processor system and scalability is achieved as the number of processors is increased.

1. Introduction

Applications have collections of functions (or methods (routines) that they invoke to perform different tasks. Based on the time analysis an application spends on each routine, we can convert some of the key sequential routines into parallel routines and deploy them on SMP machines to take advantage of shared resources. But the bottlenecks involved in doing this are: i.) developing parallel software using traditional directives-based methods increases programmers effort and this drive up the development costs[1],[2] ii.) parallel programming approaches using compiler-directives, such as OpenMP requires programmers to embed parallelization directives into their application source code, similarly driving up costs and increasing development time[3] iii.) parallel programming language approaches, such as Cilk and HPF require programmers to learn a new language in order to develop parallel applications[4],[5].

Even if one of the above methods is chosen, the programmer still needs to concentrate on the ‘how’ of programming rather than the ‘what’ of programming. These methods are representative of most parallel programming methodologies in that they are unduly expensive in terms of software development cost, development time, and programmer expertise required.

A different approach warrants consideration in today’s environment where programming resources are at a premium and the ability to support software on a wide range of systems that are present in nearly all facilities is a requirement. In this paper we present PARSA programming methodology for developing multithreaded (i.e., parallel) routines that aids in overcoming the problems of other parallel programming approaches.

The PARSA programming methodology i.) makes parallel programming no more difficult to develop than sequential programming, ii.) ensures parallel routines are thread-safe, iii.) make parallel routines more maintainable and portable. The PARSA programming methodology is supported with two tools – the PARSA Software Development Environment (SDE) and the ThreadMan Thread Manager – that are tightly coupled to facilitate the methodology.

Two typical real life sample examples – matrix multiplication and merge sort – were selected to demonstrate a run time analysis, which shows how the
methodology and its supporting tools produce efficient and scalable parallel routines.

2. Programming Methodology

The PARS A programming methodology is a unique graphical programming method that allows multi-threaded routines to be developed quickly and easily. This methodology is based on object-based programming principals to facilitate the modular development of parallel software. The process involved are developing high performance multithreaded routines using the SDE and manage its runtime execution using ThreadMan. The routines generated using this methodology can be declared and invoked just like any other routines in standard programming languages (e.g., C and C++).

2.1. Development of Multithreaded Routines using PARS A SDE

The multithreaded routines developed in PARS A SDE consist of graphical objects and arcs. When a new project (routine) is being developed, graphical objects are added to the project. Inputs and outputs for the project are automatically converted as input and output arguments to the routine. Each graphical object represents a project task. When a graphical object is added to a project, the interface the graphical object has with other graphical objects in the project must be defined. The interface consists of standard variable declarations that are declared in the same manner as that of variables declared in standard programming languages; the interface defines the “contract” a graphical object has with other graphical objects in a project. Graphical objects have an INPUT interface and an OUTPUT interface that corresponds to the data an object is dependent on for execution and the data produced by a graphical object that is needed by other project graphical objects, respectively[6].

![Figure 1. A task graphical object icon with 3 INPUT ports and 2 OUTPUT ports.](image)

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

PARSA programming methodology supports different types of graphical objects to allow different types of parallelism. Regardless of graphical object type the interface appears the same. That is, each declared INPUT variable results in an INPUT port on the icon and OUTPUT variables have similar OUTPUT port. To simplify this presentation, the graphical objects discussed are task graphical objects.

A task graphical object, as its name implies, is simply a task to be performed within a project. The task to be performed is programmed in a standard programming language. The methodology does not require programmers to put threading directives into their projects. That is, graphical objects are programmed in standard programming languages in the same manner sequential functions or subroutines are programmed. After the project is finished, the source code generator in the SDE is invoked to create all threading directives needed for the project to execute safely in parallel. Removing programmers from the low-level programming details of parallelization directives is a key of this methodology that reduces complexity.

A graphical object can be programmed to perform any desired 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 allocated and declared in global memory, which gives programmers great latitude.

Semantically, each graphical object is an independent, schedulable entity within a project that will be spawned as a thread at run time. Multiple graphical objects (i.e., threads) can be spawned to execute concurrently at any given time. Graphical objects execute without adversely affecting the execution of other objects. The phenomenon where one object adversely affects other objects executing concurrently on a multi-threaded system is known as a side effect. The PARS A programming methodology and graphical objects provide an easy-to-use framework for developing side effect free multi-threaded software.

The interaction between graphical objects is specified with arcs. Arcs connect source graphical object OUTPUT ports to destination graphical object INPUT ports. Semantically, however, arcs represent data "being passed" from a source graphical object to a destination graphical object. At run time data is passed from the thread of the source graphical object to the thread of the destination object.

Figure 2 shows a project, developed using PARS A SDE, with arcs showing the relationship between graphical objects. The source code generator produces the data structures and code needed to pass data between threads. Reentrant multithreaded code is generated.
2.2. Execution Model

The parallel routines are invoked just like any other function call with (or) without arguments (based on the how the routine has been developed by the programmer) from a typical program. Arguments may contain inputs (or) outputs (or) both based on how the programmer specified during the project development. When a routine is called with its arguments, it is ready for parallel execution.

As mentioned in the earlier section, a project developed consists of objects and arcs. 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 for data have finished executing. If it has no INPUT variables declared, then it is eligible for execution when the project begins executing. Because of the execution model the graphical representation of a project implicitly defines the order of execution of the graphical objects in the project.

The project shown in Figure 2 will execute as: The graphical object named FunctionInputs are has no INPUT ports, and therefore can execute when the project begins executing. FunctionInputs is responsible for passing the input arguments to the corresponding objects to start execution. Similarly FunctionOutputs is responsible for delivering the output variables back to the calling program. Once FunctionInputs 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, FunctionOutputs 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. Finally after Go6 and Go7 complete execution, FunctionOutputs executes and deliver the output variables back to the calling program.

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.

2.3. Source Code Generator

Once a project has been fully programmed the representation is converted into parallel multi-threaded routines 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. This code can be ported to various operating systems and system architecture supported by ThreadMan. (Currently, supported operating systems include Solaris, Linux and Windows and system architectures include Sparc and Intel based processors).
2.4. Runtime Execution of Multithreaded Routines using ThreadMan

ThreadMan is a dynamic linkable library that manages the execution of parallel software developed in PARSA SDE. It is an integral component of the PARSA programming methodology that i.) eliminates the need for programmers to generate code that controls the run time execution of their parallel software projects, ii.) provides reentrancy to the parallel routines and iii.) makes parallel source code portable across a wide range of hardware platforms and operating systems supported by ThreadMan. The parallel source code produced by the source code generator is fully compatible with functions defined in the ThreadMan Application Programming Interface (API)[7].

ThreadMan is built on top of native threading mechanisms supported on a wide range of diverse parallel systems as shown in Figure 3. Specifically, ThreadMan relies on the native user-level thread libraries provided with numerous operating systems. As stated above, underlying threading mechanisms can vary dramatically on different operating systems[2],[8],[9]. Therefore, system-specific versions of ThreadMan have been developed and are highly optimized to exploit the most efficient threading mechanisms supported on each system. Figure 3 shows ThreadMan for the three systems used in this paper to demonstrate portability, but is available on a wider range of systems.

ThreadMan provides the important benefit of allowing a single set of PARSA-generated parallel source code to be compiled on a wide range of systems in a portable manner. This 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 3) and the portable interface (TMPI in Figure 3). The role of the scheduler is to automatically manage the execution of application threads. This involves spawning threads according to the precedence relationship between graphical objects specified graphically and tracking the progress of threads (i.e., graphical objects) as they execute. The portable interface is a lightweight interface used by the scheduler to access the native user-level thread library.

3. Performance Analysis

To demonstrate, this programming methodology produces efficient run time performance an analysis was performed on two sample routines: matrix multiplication and merge sort. Specifically, these applications contain irregular, regular and repeat parallelisms[10],[11],[12]. This performance analysis demonstrates scalability.

3.1. Single Processor Performance

To show that multithreaded routines have low overhead, the merge sort algorithm and matrix multiplication have been implemented as sequential code and in PARSA (multithreaded). The algorithms are identical, but are implemented differently. The same compiler was used to compile each version of the application on the test system to ensure the results were not skewed.

Figure 4 shows the performance difference (in time) between a sequential version of the merge sort algorithm and a parallel version of the algorithm. There is only a marginal performance difference seen for smaller sized lists, but the sequential and parallel versions execute in virtually the same amount of time when larger lists are sorted, which shows that overhead is negligible.
3.2. Scalable Performance

Additional tests were conducted with a second processor added to the system to demonstrate that parallel software developed and managed scales.

Figure 6 shows the performance difference (in time) between the sequential version of the merge sort algorithm and the parallel version of the algorithm developed for various sized lists on a dual processor system. Notice that the performance of the parallel version is nearly twice as fast as sequential for all list sizes. Figure 7 shows the performance difference (in time) between the sequential version of the merge sort algorithm and the parallel version. Each iteration size produces scalable performance, where the parallel version is nearly twice as fast as the sequential for all list sizes.

3.3. Speed Up

Multithreaded routines developed using this methodology are scalable, proportionate to the number of processors. Based on the results from the sequential and parallel version of merge sort and matrix multiplications described above, speedup graphs are drawn.

Figure 8 shows a normalized performance comparison of the speedup obtained by the parallel implementation of merge sort on a single processor and dual processor machine. The parallel code performance is almost equal to the sequential performance on a single processor machine and as the number of processors increases, proportionate speed up (in this case twice) is achieved.
4. Summary and Conclusions

In this paper the development and management of multithreaded routines using a software tool was presented. The paper began with a brief description of problems associated with programming approaches and the use of the tool was then shown to obviate many of those problems by i.) abstracting programmers from low-level parallel programming issues and ii.) abstracting programming issues from deployment issues.

Development and management of parallel routines using tools was presented and shown to support an object-based, graphical programming methodology that allows programmers to develop parallel routines without concern for how parallel software executes at run time. All source code that are required to allocate data passing structures, to pass data between threads, and to manage the parallel execution of software projects was automatically generated.

A second tool – the ThreadMan Thread Manager – was shown to be an integral component of the methodology that i.) eliminates the need for programmers to generate code that controls the run time execution of their parallel software projects, ii.) ensures parallel routines are reentrant and iii.) makes parallel source code portable across a wide range of hardware platforms and operating systems supported.

Finally, a comprehensive run time performance analysis of two sample examples was presented that shows the many of the problems with traditional parallel programming methodologies are reduced while producing nearly linear performance gains.

References