AN INTRODUCTION TO
THE PMESC PARALLEL PROGRAMMING
PARADIGM AND LIBRARY FOR TASK PARALLEL COMPUTATION

S. Crivelli  E.R. Jessup
Department of Computer Science
University of Colorado
Boulder, CO 80309-0430 USA
crivelli@colorado.edu
jessup@cs.colorado.edu

Abstract

Task-parallel problems are difficult to implement efficiently in parallel because they are asynchronous and unpredictable. The difficulties are compounded on distributed-memory computers where interprocessor communication can impose a substantial overhead. A few languages and libraries have been proposed that are specifically designed to support this kind of computation. However, one big challenge still remains: to make those tools understood and used by scientists, engineers, and others who want to exploit the power of parallel computers without spending much effort in mastering those tools. The PMESC programming paradigm and library presented here are designed to make programming on distributed-memory computers easy to understand and to make efficient parallel code easy to produce. The paradigm provides a methodology for structuring task-parallel problems that allows the separation of different phases in the computation. The library provides support for those phases that are application-independent allowing the users to concentrate on the application-specific ones.

1. THE PMESC PARADIGM

The PMESC programming paradigm is an abstraction for viewing all kinds of parallel algorithms on distributed-memory MIMD (Multiple Instruction, Multiple Data) parallel computers. It is used as a method of structuring parallel algorithms, allowing the separation of different phases involving different programming issues. The paradigm is called Partition-Map-Embed-Solve-Communicate (PMESC). It is composed of five phases bearing those names: the Partition phase splits the work into tasks, the Map phase assigns and reassigns those tasks to the set of processors interconnected by some convenient virtual topology, the Embed phase embeds the virtual topology into the actual machine architecture, the Solve phase performs the algorithm itself, and the Communication phase takes care of the interprocessor communication. The phases may be executed in any number and in any order.

Parallel algorithms can be classified as static, quasi-dynamic, and dynamic. Static algorithms partition and assign work to the processors only once, at the beginning of the execution. They can be efficiently applied to regular computations but do not perform well on irregular ones. Irregular problems are those for which no a priori estimates of load distribution are possible. It is only during program execution that different processors can become responsible for different amounts of work. Adaptive approaches are especially suited to these problems because they react to the changes in the system state, concentrating efforts on those areas that look more promising and making work transfer decisions to keep the processor workload balanced. Adaptive algorithms can be quasi-dynamic or dynamic. Quasi-dynamic approaches apply to those problems that are synchronous and predictable in stages and that require periodic load balancing checks to achieve good performance. Dynamic approaches apply to those computations that are asynchronous and unpredictable.
and that require continuous, instead of periodic, load balancing checks.

Although the PMESC paradigm applies to all kinds of parallel algorithms, we concentrate on the dynamic ones. Dynamic algorithms can be found, for example, in computations involving a search tree. These include parallel eigenvalue computation by the bisection procedure, parallel adaptive quadrature, discrete optimization and global optimization problems. Dynamic problems are extremely hard to program on distributed-memory computers because they can require extensive interprocessor communication for such program features as load distribution, sharing of information, and termination detection. Two systems have been implemented to address dynamic problems: Charm [4] and Express [5]. We propose a new one that combines the strengths of both.

2. THE PMESC ENVIRONMENT

The PMESC paradigm and library constitute a medium- to coarse-grain environment for managing dynamic computations on distributed-memory computers. The philosophy of the PMESC environment is simple and powerful. It provides a tool that frees the programmer from dealing with low-level details of such issues as load balancing, interprocessor communication, and program termination, while allowing her or him to concentrate on the application specific ones.

The fundamental design of PMESC is different from the design of other tools. Rather than starting from the hardware and building a communication system, PMESC began with the applications and their requirements and built up a system to fulfill them. PMESC provides the building blocks to address different programming issues and different frameworks to put these blocks together. The programmer—not the language or system—decides which of those frameworks and blocks are most suitable for the particular application and the computer architecture.

PMESC is conceptually a two-layered environment. At the lower level, it provides support for synchronous and asynchronous message passing. At the higher-level, PMESC provides the abstractions for handling more specific programming issues. These routines form the basis for a flexible model of computation in which the underlying topology of the hardware can be completely ignored. Each level of PMESC is logically distinct and practically independent of the other, with the upper level being built on top of the lower one. As a result of that design, we can port the library to a wide variety of computers by taking a vertical approach in which the low-level may need to change while the high-level, built on top of it, does not. The user-code, built on top of those levels, remains unchanged across different computers.

3. THE PMESC PROGRAMMING MODEL

PMESC is designed for task-parallel dynamic applications. The tasks and the queue in which they are stored play a key role in the implementation and performance of dynamic problems on distributed-memory computers. Tasks are important for they define the granularity of the parallel problem. The queue is also important for it originates two different approaches, centralized (maintained by a single processor) and distributed (split into local queues maintained by all the processors), that lead to different memory usage, network usage, and programming complexities.

In the distributed queue approach, every processor works on the execution of its own tasks until its local queue becomes empty. At this point, the idle processor is assigned tasks from the busy processors in order to balance their workloads. That way, the overall execution time is reduced by dynamically distributing the work so that all processors are kept busy most of the time. The choice of load balancing mechanisms is a fundamental