

Task Parallelism: What a Tool Can Provide and What Should Be Left to the User

Silvia A. Crivelli and Elizabeth R. Jessup

Dept. of Comp. Science, Univ. of Colorado, Boulder, CO 80309-0430, USA

Abstract. This paper discusses some programming issues involved in the implementation of task parallelism on distributed-memory MIMD computers. In particular, we separate those issues that are application-independent and so can be part of a library from those that should be controlled by the user to maximize the performance.

1 Introduction

Task parallelism represents an approach for program implementation that concentrates on the decomposition of the computation to be performed rather than of the data domain. It is directed at solving problems presenting a complex task that can be decomposed into asynchronous subtasks to be assigned to the processors. Thus, it offers a valuable alternative to data parallelism which focus on the partition of the data structure.

Task parallelism is difficult to implement on distributed-memory MIMD computers because the number of the tasks and their execution time usually vary dynamically and unpredictably. Therefore, it requires a dynamic, asynchronous approach that lets processors work independently as much as possible and that only adds enough coordination for a more efficient use of the resources.

In this paper, we discuss the aspects to be considered in the development of a programming library to support task-parallel computation. In particular, we emphasize that, although a great deal of programming support can be provided for task parallelization, ample decision must be given to users as the only way to maximize the performance. The paper is organized as follows. In section 2, we identify the different issues involved in a task-parallel implementation. In section 3, we present an example of task-parallel problem. In sections 4 and 5, we discuss those issues that can be covered by a library and those that should be left to the users. Finally, in section 6, we provide a reference for further reading.

2 An Approach for Task Parallelism

In this section, we identify the main features of an efficient task-parallel implementation. Such an implementation begins with the identification of the tasks. Tasks are placed in a queue that can be maintained by a single processor or split into local queues maintained by all the processors. Processors select a task from the queue for computation, create more tasks along the way, and place them in

the queue. This process repeats until there are no more tasks in the queue or until a transfer of tasks becomes necessary to keep the workload distributed among the processors. Thus, load distribution is an important ingredient of achieving good performance. Observe that, to avoid unnecessary transfers of work, the load balancing mechanisms should not attempt to keep the number of tasks *evenly* distributed as the tasks sizes are mostly uneven and usually unknown.

Although an efficient implementation of task-parallel problems must keep the load distributed, it also must ensure that processors do not waste their time exploring unproductive tasks. Unnecessary work is avoided by allowing processors to share information and by assigning different priorities to the tasks.

A final ingredient is termination detection. In task parallelism, load balancing is asynchronously checked and transfer of work accordingly made from the heavily loaded processors to the lightly loaded ones. In this context, idle processors waiting for heavily loaded processors to send some work away may wait forever if not informed that all of the work has been completed. Because it is impossible for an idle processor to correctly decide whether or not to quit based on its local information, a global check for termination becomes necessary.

3 An Example: The Traveling Salesman Problem (TSP)

TSP is a discrete optimization problem in which a salesman must visit N cities in such a way as to minimize the cost of the trip. All cities, except for the starting city, must be visited only once. The TSP can be solved by using a branch-and-bound algorithm. This algorithm finds the solutions by searching through a tree of partial solutions that are created dynamically. The root of the tree corresponds to the given problem and the leaves represent the solutions.

The branch-and-bound procedure consists of two phases: a branching phase that generates new partial solutions by branching out from the current ones and a bounding phase that computes the cost associated with any partial solution. The cost of a partial solution is considered a lower bound on the cost of a complete solution deriving from it. Thus, partial solutions whose cost is larger than the least cost of the complete solutions computed so far are discarded. The cost associated with the partial solutions is also used to direct the search towards the most promising branches of the tree. Tasks with less cost are assigned higher priorities because they are more likely to produce a solution.

TSP is task-parallel because it can be associated with a tree that can be searched in parallel by partitioning it into subtrees. A task may consist of finding a new partial solution (i.e., a node of the tree) by applying the branch-and-bound procedure to a given one. An efficient parallel implementation of this problem needs to be asynchronous to exploit the fact that different subtrees can be searched independently and without involving synchronous communication. It also needs to be adaptive to dynamically redistribute the tree among the processors as new branches are created and others are pruned.

4 What a Library Can Provide

Of the components of a task-parallel implementation discussed in section 2, some are completely independent of the application. As these components are hard to implement, they discourage many potential users from trying task parallelism. Thus, they should be supported in a library. Charm [3] and PMESC [2] offer ample support for implementing task-parallel programs using the Single Program Multiple Data (SPMD) approach. However, as both tools are still evolving and there still is a great margin for improvement, we believe that it is important to discuss these issues. The main topics that should be supported by a library are:

- *Handling the task queue structure*: This process includes retrieving tasks from and storing tasks in the queue using different queueing mechanisms such as LIFO, FIFO, and other strategies for priority queues.
- *Load balancing*: It is particularly hard to implement in task parallelism due to the asynchronicity of the target problems. Load balancing strategies can be centralized, hierarchical, distributed or hybrid. They can also be sender- or receiver-initiated. Refer to [1] for a description of these methods.
- *Sharing of Information*: The trick is how to share global information without having to synchronize the processors or to interrupt them many times. An efficient approach is the implementation of pseudo-global variables. To implement a pseudo-global variable each processor maintains its own copy and propagates its value to the other processors by periodically activating an asynchronous communication mechanism.
- *Termination Detection*: Another aspect concerning task parallelism is termination detection. Its efficient implementation is sometimes crucial to achieving overall performance. Different approaches are discussed in [1].
- *Embedding*: It is important not only to allow users to choose the virtual machine to use for implementing their codes but also to provide them with the algorithms for efficiently embedding those virtual topologies into the real machine architectures.

There are different strategies proposed in the literature to deal with all these issues. However, a strategy that works well for one problem type may not work for another. Therefore, a high degree of customization must be provided to users so they can finely tune their codes. In the next section, we discuss this subject.

5 What Should Be Left to Users

The issues and decisions that should be left to the user so as to maximize the performance are:

- *Should tasks be prioritized?*: Some problems can be split into subproblems that can be executed in any order without affecting either the correctness of the results or the overall performance. There are however, other cases (like TSP) where the assignment of priorities to the tasks may have a tremendous

impact in the performance (and even the feasibility) of the problem. Because this issue is so important to achieving good performance, the user should be given the alternative of assigning priorities to the tasks.

- *How processors share some information?*: In task-parallel computation, processors share information through pseudo-global variables. Although maintaining a pseudo-global variable involves a communication overhead, keeping it updated allows processors to make a more efficient use of the information shared. In many problems, like TSP, a frequent update of the pseudo-global variable may reduce the amount of computation to be performed. The frequency of the updatings of the pseudo-global variable then becomes a subject of trade-off. If the frequency is too low, many wasted computations may result. If the frequency is too high, the updating procedure introduces a large overhead. The user should control this frequency.
- *Which load balancing strategy is the most efficient?*: Although there are many different strategies for load balancing, there is no single one that can be efficiently applied to all the problems on all the computers. For that reason, the best approach to use in a library is to provide different strategies and let the user decide which is the most appropriate for the particular problem. The user does not need to know the strategy to use in advance, and so she or he should be provided with modules that can be easily changed.
- *When should the load balancer be invoked?*: There is a compromise between the frequency of the load balancing calls and the overhead introduced by the transfer procedure. Too frequent calls may incur high overhead, while less frequent calls may reduce the overhead but also increase idle processors. Thus, the frequency of these calls should be controlled by the users.

6 Final Comments

The conclusions discussed in this paper are based on our experience designing and testing the PMESC library. Our analyses showed us that even naive users could improve the performance of their implementations by changing parameters and strategies. Refer to [1] for a thorough discussion of this topic.

References

1. Crivelli S.A.:
A Programming Paradigm and Library for Distributed-Memory Computers.
Ph.D. Thesis, Tech. Report CU-CS-787-95, Dept. of Comp. Science,
Univ. of Colorado, Boulder, (1995).
2. Crivelli S.A. and Jessup E.R.:
A User's Manual for the PMESC Library.
Dept. of Comp. Science, Univ. of Colorado, Boulder, (1995).
3. Gursoy A. and Kale L.:
Charm 4.3 Programming Language Manual.
Dept. of Comp. Science, Univ. of Illinois, Urbana-Champaign (1994).