Preprint ANL/MCS-P792-0200, February 2000
Mathematics and Computer Science Division
Argonne National Laboratory

Jean-Pierre Goux · Jeff Linderoth[*] · Michael Yoder

Metacomputing and the Master-Worker Paradigm[**]

February 9, 2000

Abstract. The goal of our work is to create a tool that easily allows users to distribute large scientific computations in metacomputing environments. To achieve this goal, a number of difficult implementation issues must be addressed, which may explain the relative lack of *complete* tools addressing this purpose. Our tool relies on the simple master worker paradigm, and we show that this paradigm is nicely suited for performing many of the requisite tasks of our metacomputing tool. We describe an implementation and present a case study showing the paradigm's effectiveness in solving large scientific computing problems.

1. Introduction

As a result of decreasing computer hardware costs and the increasing connectivity between computers, typical users now have access to more computational resources than ever before. When large sets of resources are connected by local or wide area networks or the Internet, they can be assembled into "metacomputers" and used to solve large and complex scientific computing problems.

Before users can take advantage of metacomputers, however, there are numerous issues that must be confronted. To bring together computational resources to attack a single problem, existing algorithms must be made to work in a distributed fashion. The problem of distributing a computation has been studied and is understood in the context of traditional computing environments, but less is known about how to effectively distribute computations in a metacomputing environment. Projects such as Condor [LBRT97], Charlotte [BKKW99], Legion [GFKH99], SNIPE [FMD99], MOL [RBD$^+$97], and Globus [FK97] all provide basic software infrastructure for supporting metacomputing. For application programmers, however, using this infrastructure to build a metacomputing application can be quite difficult. The goal of our work, then, is to

---

Jean-Pierre Goux: Department of Electrical and Computer Engineering, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208, `goux@ece.nwu.edu`

Jeff Linderoth: Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439 `linderoth@mcs.anl.gov`

Michael Yoder: Computer Sciences Department, University of Wisconsin - Madison, 1210 West Dayton Street, Madison, WI 53706, `yoderme@cs.wisc.edu`

Mathematics Subject Classification (1991): ??, ??, ??

[*] Research of this author supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

[**] This work was supported in part by Grant No. CDA-9726385 from the National Science Foundation.

develop a complete, easy-to-use tool whereby users can distribute large, diverse scientific computations in a metacomputing environment. The goals of this project are quite similar to that of the EveryWare toolkit [WBK+99].

The primary purpose of this paper is to show that the implementation difficulties encountered when distributing computations on a metacomputer can easily be resolved through the use of the master-worker paradigm. Furthermore, we describe our implementation of a master-worker framework that enables users quickly and easily to build master-worker applications that will run on metacomputing platforms.

The paper is organized as follows. In Section 2, we describe the distinguishing characteristics of our target computing environment, we mention features of an ideal parallel metacomputing tool for distributing large scientific computations, and we discuss the degree to which the master-worker paradigm can achieve this ideal. Section 3.2 describes the framework from which metacomputing-based, master-worker applications can be easily built. In Section 4, we demonstrate how to exploit algorithm characteristics in order to build efficient master-worker implementations. Section 5 presents a case study showing the effectiveness of the master-worker approach in solving a complex mathematical optimization problem. We conclude by giving some future directions of research.

2. An Effective Paradigm for Metacomputing

The term *metacomputing* is generally used to denote the idea of bringing together diverse, heterogeneous, possibly geographically distributed computing environments in a seamless fashion in order to attack large-scale computing problems. A metacomputer differs from a traditional parallel computer in a number of ways, but two characteristics are important to this discussion:

- **Dynamic availability**. The quantity of resources available and the amount of computation delivered by a single resource may vary over time. Participating resources may also disappear without notice.
- **Heterogeneity**. Resources may have varying physical characteristics, such as architecture, operating system, amount of memory, and processor speed.

2.1. A Parallel Metacomputing Tool

The dynamic environment of a metacomputing system raises the need for *resource management software* that detects when processors are available, determines when they leave the computation, and matches jobs to available processors. Parallelizing a computation involves breaking the computation into elementary tasks, scheduling these tasks, and making the results of the tasks available. Therefore, to enable parallel applications to run on metacomputers, we require a tool that performs resource management, facilitates the decomposition of the problem into manageable computational subtasks, and enables the exchange of information between processors. Ideally, this parallel metacomputing tool should have the following characteristics:

(I) **Programmability**. Users should easily be able to take an existing application code and interface it with the system.

(II) **Adaptability**. The system should transparently (to the user) adapt to the dynamic and heterogeneous execution environment. Thus, new resources of varying types should be seamlessly integrated into the computation at any point.
(III) **Reliability**. The system should perform the correct computations in the presence of processors failing.
(IV) **Efficiency**. The system should be effective in the *high-throughput* sense [LR99]. That is, the resources should do useful work over long time periods.

In a heterogeneous and unreliable computing environment, building an efficient high-throughput system is a more realistic goal than aiming for traditional high-performance computing metrics such as FLOP rates. The focus of high-throughput computing is on the amount of useful work done over long time spans. Since the problems that we target to solve will require days or weeks of computing time, it makes sense to build a system that is effective on this time scale.

2.2. The Master-Worker Paradigm

In this section, we address the effectiveness of the master-worker paradigm in light of its ability to meet each of the goals of our parallel metacomputing system.

The master-worker paradigm is very easy to program. All algorithm control is done by one processor—the master. The user need not be burdened with the difficult issue of how to distribute algorithm control information to the various processors. Moreover, the typical parallel programming hurdles of load balancing and termination detection are circumvented. Having a central point of control facilitates the collection of a job's statistics. Furthermore, a surprising number of sequential approaches to large-scale problems can be mapped naturally to the master-worker paradigm. Tree search algorithms [KRR88], genetic algorithms [CP98], parameter analysis for engineering design [ASGH95], and Monte Carlo simulations [BRL99] are just a few examples of natural master-worker computations. All these features increase a system's ease of use, accomplishing goal (I) of our parallel metacomputing system.

Programs with centralized control are easily able to adapt to a dynamic and heterogeneous computing environment. If additional processors become available during the course of the computation, they simply become workers and are given portions of the computation to perform. Having centralized control also eases the burden of adapting to a heterogeneous environment, since only the master need be concerned with the matchmaking process of assigning tasks to resources making the best use of the resource characteristics. Thus, we are able to accomplish goal (II) for our parallel metacomputing tool.

Using the master-worker paradigm, we can easily achieve goal (III) of our metacomputing tool and ensure that the computation is fault-tolerant. If a worker fails and is executing a portion of the computation, the master simply reschedules that portion of the computation. A small difficulty is that the basic master-worker paradigm is not robust in the presence of failure of the master. To overcome this liability, the state of the computation can be occasionally checkpointed. This is a simple matter, since all state information is located in the master process.

Attaining efficiency of our parallel metacomputing tool (goal (IV)) is not completely straightforward, but the master-worker paradigm *can* be used to build efficient implementations in the high-throughput sense. In this context, there are two main roadblocks to efficiency: scalability and task dependence. The master-worker paradigm is not scalable, since as the number of workers increases, there may be a bottleneck at the master, because it attempts to deal with the many requests from the workers. By task dependence, we mean the degree to which the start of one task computation depends on the completion of other task computations. A high degree of task dependence can greatly decrease efficiency in metacomputing environments, since processing and communication times of tasks and results are highly variable. The combination of this variability and task dependence can result in a large number of idle processors that are waiting on the completion of a small number of tasks. To overcome the potential problems with the master-worker paradigm in a metacomputing environment, it may be necessary to exploit certain characteristics of the parallel algorithm. These characteristics may be inherent in the algorithm, or the algorithm may be modified in order to highlight these characteristics.

One algorithm characteristic that can be exploited is *dynamic grain size*, which refers to the ability to break the computation into portions of work of variable size. We can use a dynamic grain size in order to increase a program's scalability. For example, in a tree search, the master can assign larger portions of the tree to each processor, reducing the rate at which the processors make work requests. However, this contention reduction technique comes at the expense of a loss in algorithm control, since workers now process for longer amounts of time without any feedback on whether their search is useful. Section 4.2 gives a case study showing the importance of this algorithm characteristic.

A second algorithm characteristic that can be exploited is an *incremental data requirement*, referring to the situation in which a large amount of data is required to initialize a worker process, while each individual task can be defined by a relatively small amount of additional data. Scalability problems at the master can stem from the large amount of network traffic to the workers. Only passing the incremental data required for the task reduces the necessary bandwidth and increases the scalability.

A final algorithm characteristic that can be exploited is a *weak synchronization requirement*, or a low task dependence, meaning that the ability to execute a task does not depend on the completion of a large number of other tasks. In a cutting plane algorithm for stochastic programming [KW94], each iteration consists of a number of tasks. However, the correctness of the algorithm does not depend on all of the tasks completing before proceeding to the next iteration. The synchronization requirement can be reduced, and the efficiency increased, by starting the next iteration after only a certain number of the previous iteration's tasks have completed. The significance of exploiting this algorithm characteristic will be demonstrated in Section 4.1.

Note that increasing the efficiency of the algorithm by increasing the grain size or by reducing the synchronization requirement may actually worsen the basic algorithm; more tree nodes might be explored or more iterations required. This algorithm deterioration is made in the hope of obtaining a higher parallel efficiency and lower overall computation time. Users of metacomputers and the master-worker paradigm should be keenly aware of this tradeoff, however.

One final point can be made in the case where the task dependencies of a computation can be grouped into "work cycles", where the whole algorithm is blocked until a certain set of tasks is completed. As noted before, ideally the parallel algorithm would not require this much task dependence. If work cycles are unavoidable, and tasks in a work cycle are not of equal size, or the processors performing the tasks are not of equal speed, then many processors may be left idle. An advantage of using the master-worker paradigm in this case is that the master can attempt to balance the work cycles by varying the size of the tasks sent to various processors. Pruyne and Livny [PL96] have made a similar observation.

3. A Software Framework

We have developed a software framework that meets the goals of programmability, adaptability, reliability, and efficiency outlined in Section 2.1. Based on the master-worker paradigm, it is called MW. In this section, we briefly discuss the resource management system used by MW (Section 3.1) and then describe the abstract base classes of MW (Section 3.2).

3.1. Resource Scheduling and Message Passing

MW uses Condor [LBRT97] as its resource management system. Condor differs from many other resource management systems in that it harnesses nondedicated computing resources that would otherwise go wasted. The Condor system matches user-submitted jobs with idle machines in its "pool". Specifically, MW uses Condor's support for the popular PVM software package [PL96]. PVM provides the necessary message-passing interface between the master and workers. Condor's resource management decisions are passed to the master process through special PVM messages. The master is held on a dedicated processor, and the workers are nondedicated resources from the Condor pool. These relationship are depicted in Figure 1.