A Closer Look At Scheduling Strategies for a Network of Workstations*

Shailabh Nagar *Ajit Banerjee* *Anand Sivasubramaniam* *Chita R. Das*

Technical Report CSE-98-009
October 1998 (Revised)

Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802.
Phone: (814) 865-1406
anand@cse.psu.edu

Abstract

Efficient scheduling of processes on processors of a Network of Workstations (NOW) is essential to improve system performance. However, the design of such schedulers is challenging because of the complex interaction between several system and workload parameters. Coscheduling, though desirable, is impractical for such a loosely coupled environment. Two operations, waiting for a message and arrival of a message, can be used to take remedial actions that can guide the behavior of the system towards coscheduling using local information. We present a taxonomy of three possibilities for each of these two operations, leading to a design space of 3×3 scheduling mechanisms. This paper presents an extensive implementation and evaluation exercise in studying these mechanisms.

Adhering to the philosophy that scheduling and communication are intertwined and should be studied in conjunction, a complete communication substrate for UltraSPARC workstations, connect by Myrinet and running Solaris 2.5.1, has been developed. This platform provides the entire Message Passing Interface (MPI) to readily run off-the-shelf MPI applications by employing protected low-latency user-level messaging. Several applications can concurrently use this interface. This platform has been used to uniformly design, implement, and evaluate nine scheduling strategies with a mixture of concurrent real applications with varying communication intensities. This includes four new schemes (Periodic Boost, Periodic Boost with Spin Block, Spin Yield, Periodic Boost with Spin Yield) that are presented in this paper. In addition to evaluating the pros and cons of each mechanism in terms of throughput, response time, CPU utilization and fairness, it is shown that Periodic Boost is a promising approach for scheduling processes on a NOW.

Keywords: Scheduling, Processor Management, Message Passing Interface, Network of Workstations, Application-driven Evaluation.

*This research is supported in part by an NSF Career Award MIPS-9701475, NSF grant MIP-9634197 and an NSF equipment grant CDA-9617315.

1 Introduction

Network of Workstations (NOW) has emerged as a cost-effective solution to high performance computing. Rather than relying on custom-made components, NOWs can be constructed from commercial off-the-shelf hardware. This not only decreases the cost, but also increases the availability of high performance computing platforms. However, for a NOW platform to become commercially attractive, it is essential that the delivered performance for applications approach the combined peak performance of the individual workstations. Two important issues limit the delivered performance of any parallel machine. First is the hardware and software cost of communicating between processes executing on different nodes of a machine. Second is the overhead of coordinating/scheduling these processes on the processors, and the resulting inefficiencies of CPU usage due to a non-ideal scheduling strategy.

The problem of lowering communication overhead for a NOW has drawn a lot of attention recently. On the hardware side, off-the-shelf high bandwidth networks such as Myrinet [5] and ATM [9] promise to handle the high data rates of demanding applications, with point-to-point latencies comparable to those provided by interconnection networks of custom-built parallel machines. On the software side, low-latency user-level messaging substrates (such as U-Net [26] and Fast Messages [20]) have been developed using the intelligent network interfaces provided by these networks. Approaches to translate these improvements in message latencies to the applications in the form of efficient application-level messaging layers such as MPI [18] have been undertaken [28, 16].

Optimizing communication in isolation may not necessarily translate to good delivered performance since the scheduling strategy could nullify any savings. For instance, a currently scheduled process on one node would experience a long wait for a message from a process not currently scheduled on another node regardless of the low latency for messages. Scheduling and communication are thus closely intertwined, and should not be treated as orthogonal issues. Scheduling of processes to processors on a parallel machine has always been an important and challenging area of research. It is important because the choice of a scheduling discipline can have a significant impact on the throughput and response times of the system. The research is challenging because of the numerous factors involved in implementing a scheduler. The parallel workload, presence of any sequential and/or interactive jobs, native operating system, node hardware, network interface, network and communication software are some of the factors that would influence the design and implementation of a scheduler. Most previous studies on parallel schedulers have focussed their attention on closely coupled parallel systems. Communication and synchronization costs on such machines are relatively low making it feasible to implement fancy schemes on such systems. However, many of these scheduling strategies are not very practical for a loosely-coupled NOW environment.

Scheduling is usually done in two steps. The first step is assigning a process to a processor, and the second is scheduling the processes assigned to a processor. There is a considerable body of literature [22, 17, 21, 29, 25, 10, 8, 7] related to the first step on closely coupled multiprocessor systems. Some of these studies [25, 29, 17] exploit the relatively low communication and synchronization overheads of these machines, particularly those with shared memory capabilities, to dynamically move processes across processors based on CPU utilization. Other studies [10, 8, 7] have, however, proposed static processor allocation strategies, under the assumption that process migration is expensive, and they spatially partition the set of processors to minimize communication and synchronization overheads during job execution. On a network of workstations environment, communication and synchronization costs are relatively high. Further, processes, as implemented by the native operating system at each workstation, are heavyweight, making it expensive to migrate them (of the order of a few minutes as pointed out in [2]). Hence, it may not be a good idea to migrate processes on these environments unless the jobs themselves are long [1]. In this study, we assume that processes are statically assigned to the workstations and do not migrate during execution.

The second scheduling step, which is perhaps more important for a NOW environment, is the scheduling of assigned processes at each workstation. There are a spectrum of choices that range from basing the scheduling strategy purely on

local knowledge at a workstation to using global knowledge across workstations for making more intelligent decisions. Local scheduling, which does not require any global knowledge, is relatively simple to implement. In fact, one could leave the processes to be scheduled by the native operating system of the workstation. The drawback is that the lack of global knowledge can result in lower CPU utilization and higher communication or context switching overheads. At the other end of the spectrum is coscheduling (also called gang scheduling) [19, 13, 14], which schedules processes of a job simultaneously across all processors, giving each job the impression that it is running on a dedicated system. While coscheduling has been shown to be essential for the efficient performance of fine-grained parallel applications, it would be exceedingly expensive to implement this scheme on a loosely coupled NOW environment. At the expiration of each time quantum, all nodes should synchronize and decide on the job to execute for the next quantum.

A few recent studies [24, 2, 11, 3] have explored scheduling strategies for a NOW, but they have been evaluated with simulation and/or limited workloads. Only two previous scheduling strategies [4, 23] to dynamically approach coscheduling have been proposed, implemented and evaluated on an actual NOW environment. These two strategies, called implicit coscheduling [4] and dynamic coscheduling (DCS) [23, 6], use information available locally to estimate what is scheduled on the other nodes without requiring any explicit messages for obtaining this information. Two actions, namely, waiting for a message and receipt of a message, are used to implement these schemes. Implicit coscheduling is based on the heuristic that a process waiting for a message should receive it in a reasonable time (as determined by the message latency and other factors) if the sender is also scheduled currently. Dynamic coscheduling, on the other hand, uses message arrival to schedule processes with the presumption that an incoming message indicates that the process of the same job (the sender) is scheduled on the remote node. The former [4] has been implemented and evaluated on Active Messages [27], which offers a closer coupling between the sender and receiver processes than MPI on Fast Messages [20], which has been used in evaluating DCS [23]. Further, the version of Fast Messages used in [23] can handle only one parallel application per node, and as a result the evaluation is rather limited.

There are still several unanswered questions to be addressed for developing a NOW environment that can efficiently handle coexisting jobs. First, what is the design spectrum for developing scheduling mechanisms on a NOW? In particular, what are the pros and cons of different approaches to waiting for a message and the different approaches to the handling of an incoming message? Second, how can we implement these techniques within the context of the current user-level messaging platforms? Third, how do these schemes compare with each other, and how much do they deviate from ideal behavior? Next, how do the schemes fare with respect to mixed workloads in terms of throughput and response time? Finally, in addition to throughput and response times, how do the schemes compare in terms of fairness? Answers to these questions require an experimental testbed to design and implement various scheduling strategies and a detailed evaluation to understand the intricate interaction between several factors. To our knowledge, no previous study has extensively evaluated these issues on a unified framework.

In this paper, we use an experimental NOW platform to implement and evaluate nine different scheduling mechanisms. We have developed a testbed on a network of SUN UltraSPARC server machines running Solaris 2.5.1, connected by Myrinet [5]. Using a protected, user-level communication substrate (U-Net), we have implemented the entire MPI messaging layer so that several off-the-shelf applications can be readily used for evaluations. This is, perhaps, just one of two implementations of MPI on Solaris that uses efficient user-level messaging while still supporting protected access (letting multiple applications concurrently use the network). On this platform, we describe the implementation of a range of scheduling disciplines (including the ones presented in prior research), and conduct an indepth performance study. The entire exercise has involved the implementation of software for the Myrinet interface card, user-level libraries, and kernel drivers, without requiring any modifications to the Solaris kernel.

We present a unified taxonomy for classifying different approaches to waiting for a message and handling message arrival, leading to a design space of nine scheduling mechanisms. This includes five new mechanisms, called *Periodic*

Boost (PB), *Periodic Boost with Spin Block* (PB-SB), *Spin Yield* (SY), *Periodic Boost with Spin Yield* (PB-SY), and *Dynamic Coscheduling with Spin Yield* (DCS-SY), in addition to the already existing schemes, namely Spin Block (SB), Dynamic Coscheduling (DCS) and Dynamic Coscheduling with Spin Block (DCS-SB). We conduct an exhaustive comparison of the nine mechanisms with a mixture of multiple processes having varying communication granularities at each workstation to understand their implications with real MPI workloads.

As expected, with workloads having low communication, there is little difference between the scheduling schemes. As communication increases, there is clearly a need for a scheme which uses some heuristic to guide the system towards coscheduling. Of the schemes considered, it is observed that PB outperforms most other mechanisms over a range of different workloads in terms of the overall system throughput (total completion time divided by the number of jobs serviced), and response time. PB is also reasonably fair when we consider workloads with similar communication intensities. However, PB can unfairly favor higher communication jobs when we consider mixed workloads. This is analogous to the traditional multi-level priority-based UNIX System V scheduler, which can unfairly favor I/O bound jobs in a mixture of CPU and I/O bound jobs. DCS-SB, DCS-SY and DCS are also good candidates to provide improved performance. In addition, we show that SY can be used as an alternative to SB in augmenting certain scheduling strategies.

The rest of this paper is organized as follows. Section 2 gives details on the design and implementation of the scheduling schemes. A description of the evaluation methodology, performance results comparing the scheduling disciplines, and implication of these results is presented in Section 3. Finally, Section 4 concludes with a summary of results and identifies directions for future research.

2 Scheduling Strategies

We have implemented a protected user-level messaging layer that provides the complete MPI [18] functionality which is based on the MPICH distribution [15]. Details of its implementation and performance are not included here due to space limitations. We refer to this platform as the baseline, which serves as a uniform framework for implementing and evaluating different scheduling strategies. Essentially there are three software components that are important to understand the rest of this discussion. The first is the LANai control program executing on the LANai processor of the Myrinet interface. This program performs the data transfer between the host memory and the network. Though this can raise an interrupt for the host processor, this feature is not used in the baseline implementation since message transfer is implemented by polling at the user level. The second is the set of user-level libraries, which includes U-Net [26] from Cornell, together with umlib and MPI Unet which we have developed to provide an efficient MPI interface. The implementation incorporates several optimizations to eliminate multiple levels of copying. These routines manage the send and receive queues mapped in directly to the user address space. There is no kernel invocation for data transfers. The third component is a kernel device driver, which in the baseline implementation is used only at the initialization stage to set up endpoints. An endpoint is a virtual network interface that provides a process a handle into the communication mechanism. The device driver also offers the potential for performing some actions in kernel mode (via an ioctl call), used in implementing certain scheduling schemes. A schematic showing the different components in the baseline implementation and potential additions for implementing the different scheduling strategies is shown in Figure 1. The baseline platform delivers one way latency of 32μs and a peak bandwidth of 26.2 MBytes/sec. It should be noted that the the different modules in Figure 1, except U-Net, have been developed in-house. This represents a substantial development and integration effort.

In the following discussion, we present a brief description of each scheduling strategy considered in this study, its implementation on our platform and the potential pros and cons. The reader should note that the implementation of a strategy may require the modification of one or more of the following: the LANai Control Program (LCP), the device driver that interacts with the network interface and allows certain actions to be performed in kernel mode, and the umlib