NASA-CR-191910

Parallel Processing for Scientific Computations

~~Final~~ Progress Report

Agreement Number NCC 2-644

June 15, 1991

Principal Investigator

Hasan S. AlKhatib, Ph.D.
Associate Professor of Computer Engineering
Santa Clara University
Santa Clara, CA 95053

(408)554-4485

Abstract

The main contribution of the effort in the last two years is the introduction of the MOPPS system. After doing extensive literature search, we introduced the system which is described next.

MOPPS employs a new solution to the problem of managing programs which solve scientific and engineering applications on a distributed processing environment. Autonomous computers cooperate efficiently in solving large scientific problems with this solution. MOPPS has the advantage of not assuming the presence of any particular network topology or configuration, computer architecture, or operating system. It imposes little overhead on network and processor resources while efficiently managing programs concurrently.

The core of MOPPS is an intelligent program manager that builds a knowledge base of the execution performance of the parallel programs it is managing under various conditions. The manager applies this knowledge to improve the performance of future runs. The program manager learns from experience.

Introduction

Multiprocessing software is desirable for two major reasons. The first use takes advantage of the multitude of multiprocessing computers being introduced by manufacturers. The second and more important attraction is to allow the large number of workstations that exist today to run applications previously reserved for super computers only. Special hardware is not needed to do multiprocessing. Given appropriate system software, installations that could not solve demanding applications due to the lack of access to super computers can use their existing workstations to do so. Doors are opened to an entire new set of computing power by allowing supercomputer-class computations to be executed on standard workstations. Computation speed may not match those of supercomputers, but the increased ability to solve problems that were not feasible before justifies the effort involved in using multiprocessing.

Every multiprocessor attempts to exploit as much parallelism as possible with the lowest possible overhead. Overhead is incurred by three basic causes: partitioning, scheduling, and synchronization (which includes interprocessor communication, (IPC)).

Constructing a workable solution requires tackling these sub-problems by 1) partitioning the given program into smaller tasks; 2) scheduling and coordinating these tasks given the availability of processing element, PE, and network resources; and 3) maintaining a

balanced work load between processors without overburdening the communication network [GAJ85], [DUB88]. Many researchers are seeking solutions to those sub-problems. What is needed is a framework that combines those existing solutions into a usable system.

The system we introduce here, called MOPPS, is that missing framework. It combines the existing approaches into a useful system by adding a bonding procedure that makes them function as a coherent unit. As the component solutions evolve and improve, so will MOPPS.

In this paper, the definition of a large program is any program whose size justifies the overhead incurred to package it into modules and to manage it over a multiprocessor in order to achieve significantly shorter computation time.

In MOPPS each node monitors its own workload to determine if it requires assistance to complete its work. If assistance is required then a heuristic calculation/decision is made which models the trade off between a program module's compute time requirements versus the likely communication costs of sending to another PE. If it finds it to be cheaper to pass a module to another node for processing, and there is a PE willing to help, then that is done.

The premises of MOPPS are: 1) Successful human managers of multinode systems begin with basic information that they expand upon while building their experience. Similarly, MOPPS program manager does not have to give perfect solutions at the onset. It too learns from experience; 2) The program manager is concerned only with those jobs which are run frequently on the current installation. Programs that appear infrequently need not be considered because the savings obtained by managing them are outweighed by program manager overhead; 3) There is no central authority. Nodes remain autonomous and help is offered voluntarily. 4) MOPPS tries to make the best use of the existing resources under the existing conditions.

Partitioning Problem

The goal of partitioning is to minimize the length of serial code because it is the serial code that determines computation completion times. If a computer has two operational speeds, the slower mode will dominate performance even if the faster mode is infinitely fast [PAT85].

A large program may be partitioned into loosely dependent modules so that several PEs can cooperate in executing it. The modules should be as independent from each other as possible to minimize communication between them. If the modules are not independent, their communication demands on the interconnection channels can outweigh the advantage of multiprocessing [BOK88].

Partitioning a program into independent modules is far from trivial. The boundaries between modules to run in parallel must be determined. The programmer may do this (called explicit parallelism) [APP89], or the programmer may insert directives to aid a compiler in partitioning, or a compiler may automatically partition the given program without programmer intervention (called implicit parallelism) [BAL89]. Graph theory plays an important role in partitioning as most partitioning techniques use a graph representation of the program.

Partitioning also involves choosing module granularity. Very small program modules give the highest parallelism but require more synchronization overhead, IPC, and scheduling work. Large modules mean less overhead and less IPC but also less parallelism. A good solution is one that produces tasks large enough to maximize parallelism and minimize overhead. Two or more small tasks may need to be combined into a larger one to produce the desired size [KRU88], [MCC89].

After a large program is partitioned into modules the next step is to package the modules with their relevant data in preparation for trips through the network. It is important at this point to keep track of where each module is to go, where its input data is, and where to send its output. When modules are being executed remotely a contingency plan is needed in case something goes wrong at either the software or hardware level.

Note that coming up with an ideal partitioning of a program is nearly intractable [BOK88]. We have to consider not only data dependencies but also synchronization issues and the amount of intercommunication any two modules of the program will require while running. Both synchronization and interprocess communication generate traffic on the network which has to be kept minimal for shipping modules to other nodes to be attractive. The goal is to improve performance of the overall distributed system. Optimizing the utilization of the PEs at the expense of increased communication cost is not likely to be an optimal solution.

Scheduling Problem

Scheduling, or task allocation, is an important step that if not done properly can actually decrease the total throughput with an increase in the number of processors working on the same problem . The goals of scheduling are: load balancing on the different PEs, fastest execution time of a given program, and smallest overhead in network and PE usage.

Several approaches are taken by researchers at work on this problem. They range from centralized control where global knowledge of the system is maintained in one place [PAS86], [PAS87], to distributed control where all nodes have equal knowledge of the

system. Methods used vary from Baysian decision theory [STA85] to data flow graphs [CHU87-2].

An effective scheduler minimizes the overall execution time of a large program by running its partitions on a network of relatively small computers (PEs) instead of using a single large computer. The scheduler has to achieve its goal without excessive demands on the communication channels. For example, two program modules that interact heavily while executing should be run on the same PE so as not to flood the network with messages between them.

The scheduler also ensures that volunteer nodes do not waste their time waiting for it to decide which one to ask for help. In case no other nodes are available for help, the scheduler must keep subprograms executing on their local PE.

The scheduler picks the best volunteer out of several candidates and ensures that the results needed by a module are ready as close to the time they are needed as possible.

Maintaining a System Wide Balance

Communication and synchronization are overhead incurred with parallel processing. The challenge is to increase the degree of parallelism without increasing the arithmetic and communication complexity of some other aspect of the problem.

In any effective solution it is important to keep the load reasonably balanced over all nodes by fair module assignments. It is also important to keep communication channel use to acceptable levels, particularly since the channels are likely used for other functions.

There are two types of synchronization. The first type, also known as data synchronization, is the access to shared variables where two processes are not allowed to alter the shared variable at the same time. This is usually done through critical regions. The second type is conditional or control synchronization. It occurs when processes need to wait until a set of variables is in a specific state before proceeding [DIN89], [DUB88].

Synchronization produces IPC, which also increases the load on the network, impacts scheduling, and imposes restrictions on where a task can be sent. IPC refers to the exchange of data between different processes. It is most often the result of explicit directives in the program. Synchronization is a special form of communication in which the data are control information [DUB88]. Tasks that need to synchronize a great amount might be better allocated to execute on the same PE.

The Proposed Solution

Several approaches are available for partitioning, scheduling, and synchronization. However, knowledge gained by employing those approaches is confined to a particular system. A unification scheme that allows the experience gained by using a specific set of solutions on a specific machine to be shared by different systems having different architectures, hardware, and software would be very useful [BRO89], [GAU89].

MOPPS captures that experience into a knowledge base and uses it to improve the overall execution time of an important. MOPPS is not locked into a specific architecture, programming language, or operating system. It looks at a program as a collection of functions that perform specific scientific programming tasks; e.g. matrix inversion or multiplication. It stores, in its knowledge base, the execution times of the specific functions on a given machine under varying conditions of PE and network load. It then provides that information to the project manager which uses it to plan the execution path of the job at hand.

MOPPS is most useful for scientific applications where there is a known pool of functions run repeatedly on all the machines. For example, matrix inversion, matrix multiplication, and solutions of simultaneous linear equations are all operations done frequently in a scientific environment. If each node has a scientific applications library that provides those functions, and if we know how long it takes to run those functions for a given size of data, then we can make good estimates for run times of the problem at hand. Couple that with knowledge about the PE loads and network usage at the time of execution, and we have enough data to plan intelligently. If we give those good estimates to an intelligent project manager, it can plan the execution of the job at hand with minimum overhead on network and PE.

The target environment for MOPPS is a distributed network of heterogeneous computers. The computers do not have to be identical and may vary in speed and special capabilities.

The project is the focal point of MOPPS. A project is the executable program with all its partitions, data flow graphs, scheduling restrictions, intelligent knowledge base, and whatever else is needed to successfully execute the program.

MOPPS looks at software programs as projects, no different from any other project such as constructing a building . MOPPS is built upon the following ideas:

A) The top level control of MOPPS is an intelligent program manager that attempts to perform the project as expeditiously and efficiently as possible.

B) The control entity within any node learns from experience and does not have to be absolutely correct the first time around. The system is allowed to make mistakes during a learning period after which it is expected to be "mostly correct."

C) Nodes in a distributed system are independent of each other; there is no need for global knowledge or centralized control of the system. Nodes are considered by other nodes as resources that might be available for help.

The system is designed to mimic what humans do. Humans are seldom precise in how they apply knowledge. One begins with a set of rules, which one uses as guidelines, and then *exercises* these guides to hone one's skills. With proper supervision and feedback, a human manager can become adequately effective. Of course, the initial set of rules must have been appropriate.

Future state of the art advancements for any of the system's components will enhance system operation. This modular design is both flexible and capable of remaining current.

The manager will not be optimal at first. It will learn better over time. A feedback unit, called the Intelligent Data base (IDb), is queried by the manager as to how long each section of the code is expected to take to execute either locally or remotely including communication overhead. The IDb monitors what actually happens at execution time. If the difference is within a preset tolerance level, no action is taken. If the estimate was off by more than the tolerance level, then the IDb adjusts its estimates to be closer to the actual (i.e. experienced) run times. With time, the system will tend to converge to near optimal assignments for the most frequently invoked program modules it encounters.

Top Level System Composition

The major components of MOPPS are shown in Figure 1:

A. A partitioning entity that takes operational directives and produces partitions of a program. It could be implemented as a part of the compiler, or the programmer can create his own partitions, or some combination of both.

B. An Intelligent Data base, the feedback unit, that can be easily updated and quickly queried.

C. A network load estimator that provides IDb with information regarding the current channel traffic level. This information allows the manager to make better decisions on how long it will take for a partition to travel in the network and to execute. If the network is very busy, shipping costs might be too expensive for some of the smaller partitions and computing them locally or combining them is a better decision.

D. A manager that supervises the run time activities. The manager queries IDb, or calculates on its own if no information is available from IDb, how long each partition is likely to

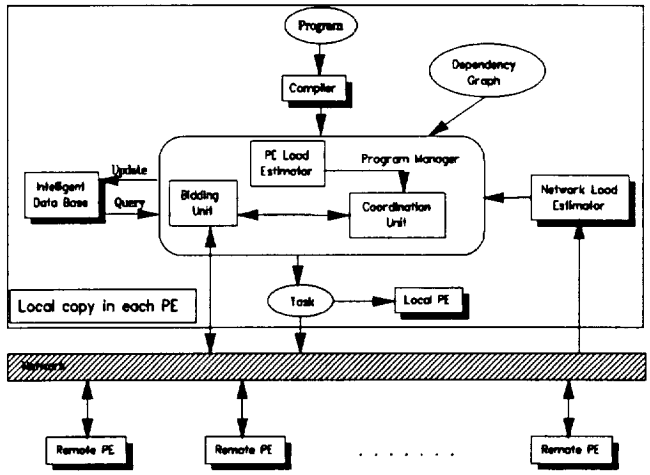

Figure 1

take to execute. The manager also checks the state of the communication channel through the network load estimator. It then utilizes a bidding process, similar to that in [PAS86], to ask for help from remote nodes. The bidding process is a protocol in which all nodes participate. It provides the nodes with a standard format for quantifying their need for help and their availability to help others, with minimum impact on communication bandwidth.

Functional Description

The partitioning entity

There are data parallel and control parallel algorithms. Data parallel algorithms are those that partition the data but apply the same algorithm to the different parts. Control parallel algorithms are those that partition the code itself and may or may not partition the data.

Many researchers are working on this problem [APP89], [BAL89], [BOK88], [CHU80], [CHU87-1], [CHU87-2]. Most solutions use a graph representation of the program at hand where nodes represent functions that execute independently and arcs between them represent precedence relations and/or inter process communication.

Although partitioning is a significant problem, MOPPS is flexible enough not to care who does it or how. This is not to suggest we are ignoring it here. We decided to create our own partitioned application instead of using other researchers' results. The application was chosen to be both of practical and theoretical importance. It also had to be non-trivial. It is described in [ARA89] and has produced a significant contribution employing Epsilon Decomposition (ED) [SEZ86] to solve large sets of simultaneous linear equations. ED is applicable to any linear system having a large array that can be partitioned. We believe that such systems account for the largest portion of scientific problems.

It is possible to expand the partitioning unit by using run time data provided by the IDb. Partitions that are too small to justify shipping individually can be grouped together (maintaining logical and data dependencies) for transmission to another node for processing. Such partitions then become candidates for economical shipping [MCC89], [KRU88].

The goal of partitioning in MOPPS is to produce well defined pieces of code that execute well defined scientific applications. The advantages are as follows: 1) It is possible to build a library of such functions that are available on all nodes. 2) It is easy to compute how long it takes to run such a function on a certain size of data. MOPPS learns these execution times and applies that knowledge to future runs in order to improve the total execution time of a program. 3) IPC is minimized since each function needs only its input data at the beginning of execution and is only heard from at the end of execution when it produces the output data. Those restrictions are not unusual because they follow the nature of scientific applications where a function has one line of input, does one task, and produces one line of output. 4) It is possible to implement partitioning in either the compiler or the program. By making the compiler aware of the library functions that can run independently, the compiler can automatically package these functions for independent execution pending the availability of their input data. Alternatively, to avoid modifying the compiler, the programmer can wrap the calls to the prepackaged library routines by directives to the PM to allow for parallel execution of those functions.

Data partitioning can use a similar approach. Data partitioning is either simple as in breaking the data into n equal parts, or complicated, as in ED [ARA89], where the break down criteria is not as straight forward. In the simple case, partitioning can either be done by the programmer, when the compiler is not MOPPS-aware, in which case the partitioned data is passed to multiple calls of the library function. Or, in the case where the compiler is MOPPS-aware, it can determine, or be told, how many partitions of data are needed. Knowing the type of function, the compiler determines how to partition data and into how big a piece, given existing memory sizes and whatever other limitations are relevant. Alternatively, the compiler can be told how to do it.

If MOPPS is implemented as an application, as opposed to a part of the operating system, then some changes have to be made. Partitioning becomes a part of the program itself which has to make the correct calls to the correct library functions. If the compiler is MOPPS-aware, then it can be taught where to find the functions called for by a program.

Intelligent Database (IDb)

IDb is the entity which captures system experience. It builds and provides run time estimates for each module worthy of monitoring.

There are solutions that make use of knowledge bases to store experience with a given system [TER89]. Most of them, however, memorize static solutions to the problem. For example, [YAN89] solves a problem on the same set of processors and improves the solution quickly. MOPPS does not assume that any given set of processors is available. It learns categories of solutions rather than specific ones. If the problem is, for example, matrix multiplication, then IDb learns how long it takes to execute the available matrix multiplication function given the size of matrices involved, the type and speed of PE, the load on the PE and, the load on the network at the time of execution.

The IDb is designed to categorize different *types* of modules and parameterize their computational and communication time requirements. A *type*, as defined here, describes the higher level operation that the module implements. For instance, if the module implements a matrix inversion routine, then its *type* could be matrix-inversion, and its operational characteristics might include matrix size, element type (eg, integer, real, complex), and number of multiplications.

IDb contents are tailored to the installation where it resides. The database is initiated with the types of applications being run, and its contents are updated as new applications are introduced. For instance, if parallel numerical algorithms were the primary focus for the system, then data such as timing requirements, operational characteristics, and precedence requirements specific to such operations would be embedded within the database [CHU87-2 and SAM77]. A similar idea, although in a slightly different context, has been proposed by Pasquale [PAS86 and PAS87].

Upon each task completion, the IDb is updated to reflect the most current experience. When no data is available about an application, we can run it the first time with gross overestimates, or underestimates, and let IDb learn about it. Simulation may also be used to obtain initial estimates.

It is essential that IDb be queried and updated quickly as it would be a system bottleneck and might slow down the entire system if not properly designed. Ultimately IDb can be implemented in hardware as a content addressable memory, or even as a neural network which could save interpolation time if the exact data is not available.

Network Load Estimator (NLE)

Since we assume that MOPPS runs on systems connected with a common network, it is important not to overload the network with messages from MOPPS. Another consideration is that if the network is busy, transmission times will be altered greatly and as a consequence total execution times will also be impacted. Therefore, we introduced the NLE to MOPPS to give the program manager an up-to-date reading of the network traffic. Smaller partitions that take a relatively short time to execute can become too expensive to ship if transmission times become too severe. In that case, it might be better to keep them on the local node, defer shipping them, or combine two or more into larger partitions.

It is the responsibility of this unit to provide the Program Manager with real time channel traffic information. The NLE can be as simple as a bus monitor which continually updates a register (interpreted as an integer) signifying channel utilization levels of high, medium, or low.

Program Manager (PM)

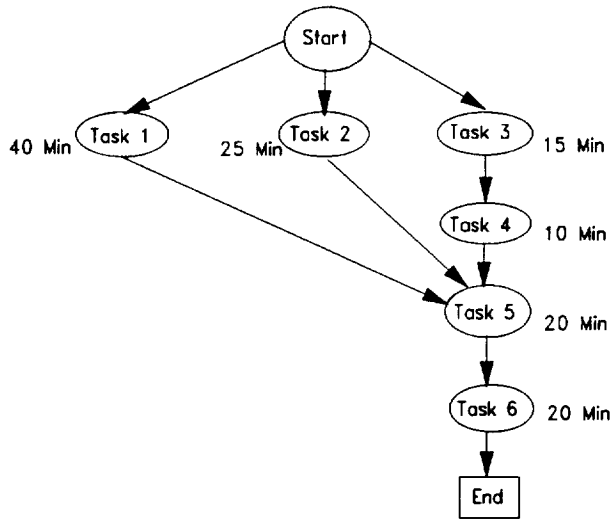

Figure 2

Central to the entire concept proposed is the Program Manager. It is the PM that is responsible for the overall execution and control of a program's modules. The PM can be likened to a project management software like those used in construction. It takes as input the number of tasks and their dependence graph, provided by the partitioning entity, and their estimated execution time, provided by the IDb. What it lacks is the list of resources, PEs and communication channel, and their availability. NLE provides information about

the channel, and the bidding process, which is introduced later, provides information about the PEs.

The PM takes all that information and schedules tasks for execution so that a task does not start much before or after its input is available. This minimizes the buffering of intermediate data and the wait a PE suffers when it tries to run a task whose input is not yet available. Intelligent scheduling is done by using the dependency graph of the program and the estimated (best case) execution times of tasks. By going backwards in the graph, PM decides when each module should start execution.

Figure 2 shows an example of a program graph with six tasks and their estimated execution times. The last task, T6, requires 20 minutes to execute and depends on the results of tasks T1, T2, T4, and T5. T5 depends on T4 and T4 depends on T3. Given those times, PM schedules T1 to start at time t_0, T2 and T3 at time t_0+15. When T3 is finished at t_0+30, T4 starts. By the time T4 is finished at t_0+40, so should T2 and T1. At this point T5 begins, and when it is finished T6 is executed. From this example, it is clear that, for example, T3 should start no later than t_0+15. This means that PM has 15 minutes to ask for bids for T3 and ship it. This is how PM decides when to start asking for help with a given task and when it has to decide computing it locally.

The PM is responsible for assigning modules, one by one from its local queue, to the selected processing elements including its own local PE. This local PE may be assigned a new computationally intensive software module while it is already doing a large amount of work. In this case, it is the responsibility of the PM to determine whether it is more efficient to ship the module to another PE or to keep it in the queue of the local PE. The PM is essentially another process that manages sub processes.

The program manager does not assume that nodes which helped it in the past will help it in the future. Therefore, it does not memorize a fixed solution for a problem (configuration) that would be repeated each time. Instead, through IDb it learns how long it takes to execute each module given system-wide conditions that include network traffic and PE loads (relevant PEs). Every time the PM encounters that module, it repeats the process of asking for help. During subsequent runs, it is armed with better estimates of how the rest of the network will behave under the existing set of conditions. In this manner, the most appropriate PE available for help is used rather than relying on the same PEs every time.

Definition of PE load

In MOPPS the PE load is defined to mean not just the number of processes but also their estimated length of execution. Merely counting the number of processes in the PE queue

and taking that as an estimate of the load on that PE is insufficient. Jobs vary in their execution time. For example, one PE might have a single job that takes hours to do, while another could have several jobs that take a minute to finish. A better way of measuring the load on a single PE is to sum the estimates of how long each job takes. Through comparing these load estimates a reasonably fair balance, not necessarily the fairest, will be maintained.

If MOPPS is not implemented as a part of the operating system, then PE load reflects only those jobs that the PM is working on.

Good channel usage relies on an intelligent bidding algorithm. This process is described in the following section.

The Bidding Process

An important consideration in the design of any system that manages parallel processing applications is the capability to communicate its need for help to other nodes. It has to do so without sending too many messages over the network. The message sent has to include enough information to allow remote nodes to decide whether they can help or not. The bidding algorithm described here is an efficient way for achieving this goal. It sends the minimum number of messages, wastes very little time of the nodes offering to help, and does not require a central information keeper to know who is doing what. For the bidding algorithm to work effectively we must have an efficient measure of the load on nodes. We defined a new yardstick for measuring PE loads in the previous section.

The local PM weighs execution time versus shipping and management time for each resident module. If execution time is greater than shipping and management time and the load on the local PE is higher than a predefined threshold, the PM broadcasts a global message through the interconnection network asking for help. This "help wanted" message includes enough information about the module to be sent enabling other nodes to determine if they can offer their help. The information includes the estimated module execution time, memory and disk requirements, and any other information that is useful in making the decision.

Those PEs which can potentially bid to accept the module for processing will examine this information and determine whether it is feasible to bid. If a node is capable of assisting it will return a message stating its availability, and will commit to this bid for a period long enough for the asking node to receive the return message and act on it. Through this process, nodes that bid for help and are not accepted will waste little time before considering later "help wanted" messages.

Each node will monitor the network before sending its reply to determine if any other node(s) have responded to the bid and will not send its reply if any node did respond. It is assumed that the first node to reply will get the job, and there is no need for others to do so. The PM sends the module to the first node that replies to the request.

There are a number of possible variations on this scheme; simulation will show which are more effective and economical.

It is important that the system be able to estimate the load on the PEs. Each node has a preset limit of load that its PE is allowed to be committed to at any one time. A possible implementation would be to use a load gauge. The gauge is incremented every time a large job joins the PE queue by the estimated execution time of that job. When the job is finished, the execution time is subtracted from the gauge. The maximum load of a PE occurs when a PE is committed for the next, say, five minutes. Thus, only those nodes currently below their maximum load will respond to requests. Notice there is no central control in this scheme, instead, distributed cooperation is enforced.

Two rules make it unnecessary for the PM to decide which PE to ask for help. First, each PE observes the preset busy limit and does not take any bids if it is busy longer than the limit. Second, when a PE accepts a bid, it accepts it regardless of its length. This way, tasks that take longer than the preset limit are not blocked out. Given those two rules, when a PM asks for help and a PE accepts it, the PM knows that the maximum time the task will take is the sum of its execution time and the preset busy limit plus a factor of uncertainty (to account for things like interactive use of the remote PE).

The PM repeats the help wanted message for a given task until either it receives a response or the task is at the point where it has to be executed in order not to delay the rest of the tasks.

Future Direction

Most significant scientific applications require the use of very large address space. Moreover, there is some sharing of memory in those applications. Therefore, the next step of our work will focus on extending the virtual address space of a task from a single node to several nodes. The address space will be a concatenation of all address spaces of all the nodes on the network. This address space may contain holes of non-existing address ranges due mainly to nodes being down or being un-involved in the task being run. Each node knows which parts of the address space are local and which are remote. Of the remote parts, it is known where each contiguous region resides, i.e. which node's physical memory it represents. When a task with a large address space requirement is run, each thread in it runs on

a different node's physical memory. When a thread accesses a variable that is not local, a remote-access fault is generated and the operating system takes over and fetches that variable from the node it resides on. The operating system knows where to look for that variable because it has a map of physical address regions to nodes.

Most of the design of MOPPS will remain the same. Bidding will still be implemented, so that nodes that are busy will not be overrun by extra work. The scheduler will also remain about what it is right now.

A new set of problems will have to be addressed. The most important of which are task migration and sharing memory across a slow network.

Memory sharing requires first recognizing that a particular variable is not local and then fetching it across the network efficiently. A variable's location is known by its address. As described earlier, different address regions reside on different nodes. Fetching a variable across the network requires the cooperation of both operating systems. First, we need a way to identify the particular application to the remote system because the remote system may be running other applications that may have the same address space. Then we can ask for a particular address within that application's address space.

The simplest case is when the variable is in active memory and its application is running. In this case we merely need to send its value back. A second case is when the variable is swapped to disk while its application is still running. This case generates a paging fault to get the page from swapping storage. Address resolution is easier here because all the tables are loaded with the desired applications base addresses. The third case is when the application itself maybe swapped to disk altogether. Here we need to reload the information necessary to locate a particular address in the address space.

Task migration can be classified in three main categories. The first is where both data and program reside on the remote node and all we have to do is activate the program to operate on the data. This is equivalent to pure RPC. This is the simplest case where there is no shared data and all we need to worry about is how to send the output to the right place.

The other extreme is where we need to migrate an entire thread and its data. This is a difficult problem especially if we decide to migrate a thread after it started execution. The main problem here is maintaining the execution environment of the thread. A thread runs under the umbrella of a task. The task contains everything a thread needs to execute such as system resources, shared variables, and others. In its simplest scenarios, this case can turn into the third category discussed later where we merely ship the code and then share the data.

There is a large body of research in the area of stand-alone thread migration. Threads in this body of research are self contained entities similar to our tasks and unlike our threads which may share data. We expect to apply some of the conclusions reached in that area to our problem. [ART89], [ZAY87], and [SMI??].

The last category of task migration is a middle ground of the previous two. It can be thought of as RPC with shared memory. In this case, the program resides on the remote node, maybe in the form of a canned library of scientific routines. Data is then shared between remote nodes.

The performance of the final system will depend heavily on the efficiency of task migration and memory sharing. Every time a program accesses shared data, a significant performance loss is expected. We expect to substantially reduce this performance loss by three main ways. One, by having smart algorithms for partitioning, scheduling and memory sharing. Two, by creating local caches of some of the shared data that reside on remote nodes. Three, by using faster networks such as FDDI and TurboLan.

Factor one, above, is under our control and will be a part of our solution. Factors two and three are outside the scope of this work. However, they are heavily researched by others and once resolved they will enhance the performance of multi-node computing in the future including our own.

Conclusion

In the past two years, we have developed a flexible system for the management of parallel scientific programs running on a distributed computer network. It learns and adapts to improve its system management decisions as it monitors the execution of applications.

The system will be expanded so that it looks at the entire network of workstations as a single address space. New issues to deal with are process migration and memory sharing across a slow network.

References

APP89 Appelbe, B., Smith, K. and McDowell, C. "Start/Pat: A parallel programming toolkit." IEEE Software, July 1989.

ARA89 Arafeh, Hassan and AlKhatib, Hasan, "Using the Epsilon Decomposition Method to Solve Sets of Simultaneous Linear Equations," Technical Report 143, EECS Department, Santa Clara University, 1989.

ART89 Artsy, Yeshayahu and Finkel, Raphael. "Designing a Process Migration Facility The Charlotte Experience." IEEE Computer, September 1989. pp47-56.

SMI?? Smith, Jonathan M. "A Survey of Process Migration Mechanisms."

BAL89 Baldwin, D. "Consul: A parallel constraint language." IEEE Software, July 1989.

BOK88 Bokhari, Shahid H., "Partitioning Problems in Parallel, Pipelined and Distributed Computing," IEEE Transactions on Computers, Vol 37 #1, January 1988 pp. 48-57.

BRO89 Browne, J. C., Azam, M. and Sobek, S. "CODE: A unified approach to parallel programming." IEEE Software, July 1989.

CHU80 Chu, Wesley W. et. al., "Task Allocation in Distributed Data Processing," IEEE Computer, November 1980, pp57-69.

CHU87-1 Chu, Wesley W. and Leung, Kin K., "Module Replication and Assignment for Real-Time Distributed Processing Systems," Proceedings of the IEEE, Vol. 75, No. 5, May 1987, pp.547-562.

CHU87-2 Chu, Wesley W. and Lan, Lance M-T, "Task Allocation and Precedence Relations for Distributed Real-Time Systems," IEEE Transactions on Computers, Vol. C-36, No. 6, June 1987, pp. 667-679.

DIE84 Dietel, Harvey M., "An Introduction to Operating Systems," Addison-Wesley Publishing Company, 1984.

DIN89 Dinning, Anne, "A Survey of Synchronization Methods for Parallel Computers," IEEE Computer, July 1989, pp. 66-77.

DUB88 Dubois, M., Scheurich, C., and Briggs, F. A. "Synchronization, Coherence, and event ordering in multiprocessors." IEEE Computer, February 1988.

GAJ85 Gajski, Daniel D., and Peir, Jih-Kwon, "Essential Issues in Multiprocessor Systems," IEEE Computer, June 1985, pp 9-27.

GAU89 Guarna, Vincent A. Jr; Gannon, Dennis; Jablonowski, David; Malony, Allen D.; and Yogesh Gaur, "Faust: An Integrated Environment for Parallel Programming," IEEE Software, July 1989, pp. 20-29.

KRU88 Kruatrachue, B. and Lewis, T. "Grain size determination for parallel processing." IEEE Software, January 1988.

MCC89 McCreary, C., and Gill, H. "Automatic determination of grain size for efficient parallel processing." Communications of the ACM, September 1989, Volume 32 Number 9.

PAS86 Pasquale, Joseph, "Knowledge-Based Distributed Systems Management," Report No. UCB/CSD 86/295, UC Berkeley, Computer Science Division, June 1986.

PAS87 Pasquale, Joseph, "Using Expert Systems to Manage Distributed Computer Systems," Report No. UCB/CSD 87/334, UC Berkeley, Computer Science Division, January 1987.

PAT85 Patton, P. "Multiprocessors: Architecture and Applications." IEEE Computer, June 1985.

SAM77 Sameh, Ahmed H., "Numerical Parallel Algorithms -- A Survey," in High Speed Computer and Algorithm Organization, Academic Press, 1977, pp. 207-228.

SEZ86 Sezer, M. E. and Siljak, D.D., "Nested Epsilon-Decompositions and Clustering of Complex Systems," Automatica, Vol 22, No. 3, pp. 321-331, 1986.

STA85 Stankovic., J. A. "An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling." IEEE Transactions on computers, Vol. c-34, NO. 2, February 1985.

TER89 Terrano, A. E., Dunn, S. M., and Peters, J. E. "Using an architectural knowledge base to generate code for parallel computers." Communications of the ACM, September 1989 Volume 32 Number 9.

YAN89 Yan, J. C., and Lundstrom, S. F. "The Post-Game Analysis Framework-- Developing Resource Management Strategies for Concurrent Systems." Submitted to: Transactions on Data and Knowledge Engineering, 1989.

ZAY87 Zayas, Edward R. "Attacking the Process Migration Bottleneck." ACM 1987.

ZOR89 Zorn, B., Ho, K., Larus, J., Semenzzato, L., and Hilfinger, P. "Multiprocessing extensions in Spur Lisp." IEEE Software, July 1989.