March 1994

Adaptive Execution of Data Parallel Computations on Networks of Heterogeneous Workstations

Robert Prouty

Steve Otto

Jonathan Walpole

Adaptive Execution of Data Parallel Computations on Networks of Heterogeneous Workstations

Robert Prouty, Steve Otto, Jonathan Walpole
{prouty, otto, walpole}@cse.ogi.edu

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology

Abstract

Parallel environments consisting of a network of heterogeneous workstations introduce an inherently dynamic environment that differs from multicomputers. Workstations are usually considered "shared" resources while multicomputers provide dedicated processing power. The number of workstations available for use is continually changing; the parallel machine presented by the network is in effect continually reconfiguring itself. Application programs must effectively adapt to the changing number of processing nodes while maintaining computational efficiency.

This paper examines methods for adapting to this dynamic environment within the framework of explicit message passing under the data parallel programming model. We present four requirements which we feel a method must satisfy. Several potential methods are examined within the framework and evaluated according to how well they address the defined requirements.

An application-level technique called Application Data Movement (ADM) is described. Although this technique puts much of the responsibility of adaptation on the application programmer, it has the advantage of running on heterogeneous workstations. Related work, such as Dataparallel C and Piranha, is also examined and compared to ADM.

The application of the ADM methodology to a real application, a neural-network classifier based on conjugate-gradient optimization, is outlined and discussed. Preliminary results are presented and analyzed. The computation has been shown to achieve in excess of 70 MFLOPS under quiet conditions on a network of nine heterogeneous machines, two HP 9000/720s, two DEC Alphas, and five Sun SPARCstation 10s, while maintaining an efficiency of nearly 80%.

1 Introduction

High performance workstations connected by local area networks are becoming common in academic and industrial research environments. The aggregate computational power represented by these workstation networks often rivals and in some cases exceeds the power of traditional supercomputers [Bro92]. Workstation performance is rapidly increasing, and network capacity (speed and bandwidth) is also expected to increase as new technologies become widespread. Because these workstations sit idle much of the time, they are an attractive source of effectively free computational power.

Researchers have become increasingly interested in using networked workstations to perform large scientific computations. The Condor system [LLM88][LL] was developed at the University of Wisconsin in 1988 to make use of idle workstations to perform sequential computations. Condor starts a sequential job on an idle workstation and moves it to another workstation of the same architecture when the current workstation becomes busy.

More recently, support for running parallel applications on workstation networks has been researched. A large and useful subset of scientific computations can be parallelized using data parallelism. Data parallel applications are characterized by the repeated application of an algorithm or set of equations to a very large data set. This parallelism is extracted by partitioning the data and distributing each portion to a separate processor. Each processor then performs the computation using only its portion of the data. The results from each processor are continually combined so that the overall solution evolves from the separate pieces.

This paper addresses the problem of data parallel processing on networks of heterogeneous workstations (worknets), using explicit message passing systems. Traditionally, data parallel applications are run on dedicated multiprocessing systems. A worknet is not a dedicated processing resource; the component workstations are usually considered "shared" resources—often, a workstation may be utilized by a parallel job as long as the workstation owner is not using that workstation. Therefore, the number of workstations available for use is continually changing. Worknets thus introduce an inherently dynamic environment that differs from multicomputers. A parallel application running on a worknet must adapt to the dynamic nature of the environment. This adaptation is the focus of this paper.

Dataparallel C [NQ93] and Piranha [GK92][CG+] are two systems designed to adaptively utilize worknets. Dataparallel C was developed by Michael Quinn at Oregon State University as a SIMD programming language to run dataparallel programs on worknets and perform automatic load balancing. Piranha is a system from Yale University which is implemented on top of Linda [DS92][CG89]. Neither of these systems utilizes explicit message passing, and so they address a slightly different problem domain.

The remainder of the paper proposes a solution, analyzes it, and compares it with alternatives. Section 2 introduces the four requirements for an adaptation method. Section 3 explains the implementation platform for this work. The ADM methodology is then explained in detail at both the conceptual level and using a specific example in section 4. In section 5 alternative methods are examined. Areas for further research are presented in section 6. Work by other researchers is discussed, and the relation to this work is discussed and summarized in section 7. The contribution of this work is summarized in section 8.

2 Adaptation Requirements

Using a network of workstations to perform data parallel computations presents a number of interesting challenges not found in dedicated multiprocessing machines. First, and most important, is the concept of workstation ownership. Workstations are usually a shared resource, yet they typically have a principal user who considers that workstation as his or her own. Often, workstations owned by other users can only

be utilized for parallel processing if they are used *unobtrusively*. The obtrusiveness of a method can be measured as the time between when a workstation owner begins using the workstation and the time when the parallel computation is removed from that workstation. To be as unobtrusive as possible, the adaptation method must have the ability to quickly migrate the computation away from a workstation. Therefore, the first requirement for utilizing a network of shared workstations as a parallel computing environment is

     1. *withdrawal: the ability to quickly withdraw the computation from a machine.*

The computation should also be able to take advantage of any machines which become available during the computation. The second requirement is

     2. *expansion: the ability to expand onto newly available nodes.*

Requirements 1 and 2 will be collectively referred to as the *migration requirements*.

To accomplish expansion and withdrawal the adaptation method must be able to redistribute work. In either case, work must be redistributed among a new number of machines. The third requirement is

     3. *dynamic work distribution: the ability to redistribute work to a dynamic number of processors.*

Data parallel applications have an implicit load balancing requirement. The computation generally runs at the speed of the slowest processor. Ideally, all machines in the worknet would finish their work at the same time. If any machine lags behind, all other machines will be forced to wait for it, forming a computational bottleneck. In order to maintain efficiency, each processor should do as much work as it can without causing the other processors to wait. Since we are interested in using networks of *heterogeneous* workstations, the adaptation method must be able to distribute work to processors according to the innate computational speed of the machine. The fourth requirement we impose on the adaptation method is

     4. *static work distribution: the ability to distribute work based on static machine speed.*

Both requirements 3 and 4, the *redistribution requirements*, have an additional criteria: they must allow work to be redistributed with a precision that allows for reasonable computational efficiency. It is unacceptable if the work redistribution forces any machine to become a bottleneck.

3   The Implementation Platform

The implementation environment is based on a UNIX-based message passing system: PVM (Parallel Virtual Machine). PVM is a powerful and flexible software system which allows a worknet to be used as a single computational resource, a virtual machine. PVM runs on a per-user basis. That is, several users may have virtual machines running on overlapping sets of machines, and they will function with complete autonomy. PVM is based on UNIX sockets and provides typed data exchange between different architectures through XDR[1] encoding, thus providing limited support for heterogeneity. The PVM system consists of a runtime library and a set of daemons. It provides routines for message passing, process control, and virtual machine modification through the library which is supplied in both C and FORTRAN versions. The daemons act as message routers and perform tasks such as starting and stopping PVM processes. Each application process is a full UNIX process and is statically assigned to a processor when initiated. The system can easily be used for data parallel programming. For more information on the PVM system, see [GS92][GB+93].

As a platform, PVM provides some mechanisms which can be used to adapt the computation to a worknet, but does not actually address adaptation. PVM provides process control through the ability to start and stop processes on any workstation at any time. PVM does not provide any facility for migrating a process from one workstation to another, but it does allow all machines to exchange typed data. It provides

---

    1. *External data representation* developed by Sun Microsystems.

no method for determining the current level of activity on a machine, nor does it have any concept of static workstation speed. However, PVM does allow programs to make any system calls which non-PVM programs can make, so a PVM application could be written to determine the state of the local processor.

The local worknet on which we use PVM to run parallel applications consists of nine high-performance workstations connected by an ethernet network. The network is broken into subnets connected by a single fast router. There are two HP 9000/720s on one subnet, two DEC Alphas on a second subnet, and five Sun SPARCstation 10s on the third subnet. Each machine has at least 32 MB of physical memory and has achieved at least 8 MFLOPS when running a simple LINPACK benchmark written in C.

4   Application Data Movement

We employ an application level approach, ADM, to meeting the four adaptation requirements explained in section 2. Adaptation consists of moving work. Withdrawal and expansion involve removing work from or giving work to a processor. Static and dynamic work distribution also involve moving work. In a data parallel application, work is directly proportional to data. Therefore, ADM moves work by moving data. Work must be moved to respond to *migration events*. A migration event is the transition of any node in the worknet from *idle* to *busy*, or vice versa. A busy machine is defined as any machine being used to perform user work other than the parallel computation. An idle machine is simply a workstation which is not busy. A mechanism is needed to detect migration events. This mechanism is encapsulated as a single function, known as the host check function, which is separate from ADM. Neither the policy nor the mechanism used by the function are part of ADM. ADM must call the host check function in order to check for migration events. Redistribution takes place, if needed, when the host check function is called. So the host check function can only be called at a time when the computation can be stopped, the data moved, and the computation restarted. ADM performs the host check at each iteration of the data parallel program. In a data parallel program, an iteration consists of each processor applying the computational algorithm to its portion of the data set.

An ADM application consists of one process on each node of the worknet. One processor is designated as the home node, and will never be withdrawn. In the worst case, all work is moved to the home node, and ADM degrades to the sequential algorithm. All other nodes on the worknet are either withdrawn or expanded. Withdrawn nodes have no data, they simply make a call to the host check function at a predefined frequency, and send a message to the home node when they need to be expanded. Expanded nodes have the data divided among them. At the end of each iteration, all expanded nodes call the host check function and send the resulting information to the home node. At any iteration at which the home node receives information that a migration event has occurred, it calculates a new data partition, broadcasts this data partition to all nodes which are either expanded or changing state, and the nodes exchange whatever data is necessary to perform the required expansion or withdrawal.

4.1   The Three Parts of ADM

The ADM model involves three steps: detecting when the data needs to be repartitioned, defining the points during computation when the data can be reshuffled, and defining a mechanism to perform the repartitioning. The first should be accomplished by the host check function which can be written once and reused. The last two are the responsibility of the programmer and may vary widely among applications.

The host check function can be written once and reused by all ADM applications. This function must be able to determine if the local processor is idle or busy, and return a Boolean value indicating a migration event has occurred or not. The host check function must decide what processor activity constitutes a busy machine, and how inactive a machine must be to be considered idle. It also determines what level of vari-

ation in processor activity is tolerated before a state change, and thus a migration event, is declared. The host check function is in effect making a policy decision about when the program will adapt, and thus needs to be reasonably intelligent. For example, it would be detrimental for the host check function to be too eager to indicate a change in the host machine's state, because responding to every small change in machine activity might cause the program to spend all its time adapting and little time computing. These policy decisions are not the focus of this work, but a possible solution is discussed in the section on future work.

The choice of when to call the host check function is very important, because it affects the ability of the program to adapt to the worknet. As described earlier, each process in an ADM calls the host check function at the end of each data parallel iteration. Thus the speed at which a computational iteration is performed determines the frequency at which migration events are checked for. The frequency at which migration events are checked impacts how quickly the program can withdraw, because the program cannot even begin to withdraw from a node until a migration event on that node is noticed. Therefore there are two factors in determining obtrusiveness. The first is the time between when a workstation owner begins using the workstation and the point at which a migration event occurs. The second is the time between the migration event and the actual removal of work from that machine. The first is the part related to the temporal granularity of the host check, as just described. The second is influenced by the redistribution mechanism, which is discussed next. Both times are also influenced by the amount of data on that node, because that amount of data determines iteration speed, and how much data must be removed.

Finally the programmer must provide a mechanism by which the data is actually repartitioned. This is a two-step process. First a new partition must be calculated so that each process knows what data needs to be moved where. Secondly, the data must be moved. Before any process can calculate the new partition, it must know which processes are active and which are inactive. Given the distributed nature of the information returned by the host monitor, the state of all processors must be collected before a new partition can be calculated. This is easily accomplished. Each expanded process calls the host check function and sends the information to the home node. The home node collects all the state information and, if necessary, calculates a new partition and broadcasts it to all the processes which are expanded or changing state. Once each process knows the new partition, it can compare the new and old partitions to determine what data it must send to whom, and what data it will receive from whom. The required sends and receives can then be performed to actually move the data.

4.2   Satisfaction of the Four Requirements

ADM meets the withdrawal requirement in effect by removing all the data from the withdrawing process. However, the process will still be running on the workstation competing for a minimal amount of resources even after all the data has been evacuated, so strictly speaking the computation is not completely removed. This imperfection in withdrawal actually allows ADM to achieve expansion. The processes on withdrawn workstations (skeleton processes) call the host checking function once every minute, and when the local processor is again idle, they send a message to the home node, and rejoin the computation on the next iteration. Processors which are busy when the computation begins also have a skeleton started on them. Having a process on every node in the worknet allows ADM to fulfill expansion, because it can monitor all workstations, and can redistribute data to expand whenever necessary. A weakness of ADM is that it uses skeleton programs to achieve expansion, and so cannot be utterly unobtrusive. The real strength of ADM is that it achieves *heterogeneous* migration. Work can be taken from a workstation of one architecture, and given to one or more workstations of other architectures. Other methods discussed later are unable to achieve heterogeneous migration.

The redistribution requirements are met more naturally by ADM. Data parallel programs are very sen-

sitive to load imbalance. Each processor must be computing an appropriate share of the work so that all processors complete at the same time. The precision at which work can be moved around is a problem for some methods, but not for ADM. The model itself does not impose any restriction on the precision at which data may be partitioned. The application programmer is free to use whatever precision is most appropriate for the particular application. Thus the application, not the model, limits the accuracy with which the data can be allotted to the heterogeneous processors, as well as the accuracy with which data can be redistributed when the number of processors changes. This all means that the nodes will each have a near ideal amount of data, they will all complete computing together, and the computation remains efficient.

There are both costs and benefits of ADM which are not captured by the four requirements. These costs and benefits have been touched on already, but are collected here for completeness. There are two weaknesses of ADM. The first is that it is not completely unobtrusive. Skeleton processes remain running on withdrawn nodes, migration events are not checked for until the end of the current iteration, and redistribution takes place after a migration event is noticed so is itself obtrusive. The second and most serious disadvantage of ADM is the complexity added to the application programming process. The data parallel application must be structured in a certain way, and functions to accomplish redistribution must be written. The strengths of ADM are also twofold. The first is that ADM allows truly heterogeneous migration and work distribution. The second is that the precision of work redistribution is limited only by the application.

4.3   An Example Computation

The ADM method has been applied to "Opt," a neural-network classifier based on conjugate-gradient optimization [BC89]. Opt is generally employed as a speech classifier, and as such utilizes enormous (500KB to 400MB) training sets as input. A training set consists of a series of floating point vectors. These vectors, called exemplars, represent digitized speech sounds. Each sound belongs to a particular speech category, which is included in the vector as a single scalar value. Opt uses a supervised neural-network algorithm involving backpropagation and conjugate gradient descent to train the neural-network to predict the corresponding category. The predictions are compared to the correct answer, and the algorithm is run until some convergence criteria is satisfied. The system can then be used to determine categories for speech vectors that have not already been categorized. The basic algorithm is as follows. An initial neural-net, which is simply a (large) matrix of floating point numbers is established and applied to the exemplars so that a gradient is found. The gradient is also a matrix the same size as the neural-net. That gradient is then used to modify the neural-net before it is reapplied to the data to get a new gradient. This two-step process of applying the neural-net to the exemplars and then modifying the net is repeated until error values pass a threshold or a predetermined number of iterations has been performed.

The calculation of the gradient dominates the computation time in the sequential program. Since it involves the repeated application of the neural-net algorithm to a very large date set, it can be written in data parallel style. The inherently sequential portion of the algorithm, the modification of the neural network, is not parallelized. The ADM version of opt is written in a master-slave arrangement with the master process performing the network modification, then communicating the resulting neural-net to the slaves. The neural-net is replicated in each slave process, each of which applies the network to its portion of the data, accumulates a partial gradient, and sends that back to the master. The master combines the partial gradients into the gradient which is used to modify the network. Figure 1 illustrates the parallel algorithm.

ADMopt also detects migration events and performs data movement. This is the functionality which allows ADMopt to adapt to the worknet. These functions must be incorporated into the data parallel algorithm at both the master and slave level. The slaves employ the majority of the code to accomplish ADM.
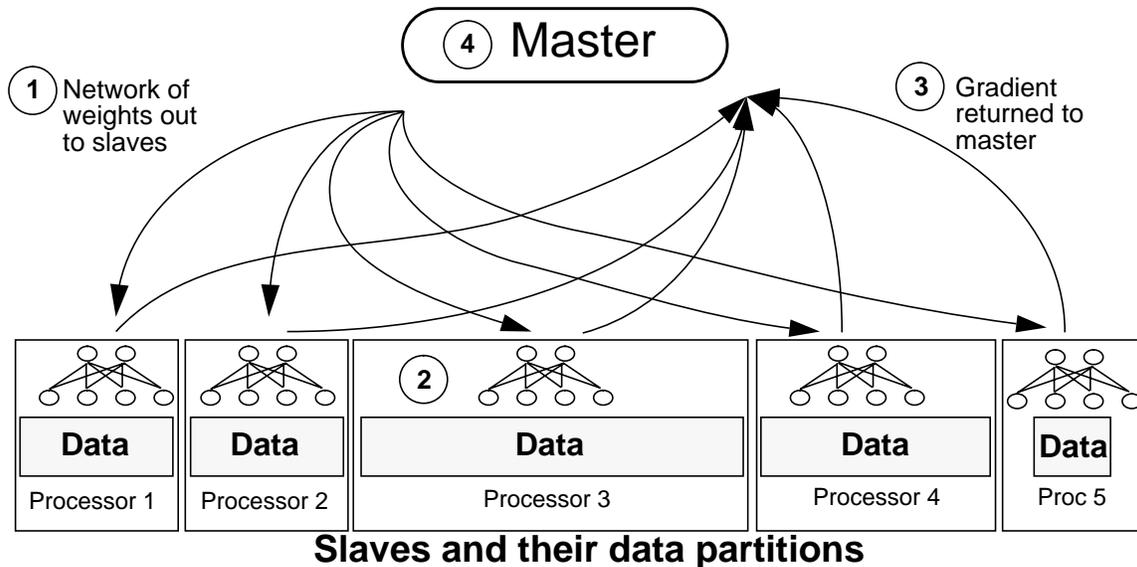
Figure 1: *Data-parallel opt written in a master-slave style. Each slave has its own portion of the training set. 1. The master broadcasts network to slaves. 2. Each slave calculates a partial gradient. 3. Partial gradients sent to master. 4. Master accumulates partial gradients into complete gradient, performs conjugant-gradient analysis to form a new network. The cycle continues until the network is trained.*

The master simply serves as an overall coordinator, informing all slaves of migration events and calculating new data partitions.

Figure 2 shows the algorithm employed by the slave. Each slave first goes through an initialization phase. In this phase, each slave checks the load on the local processor and determines if it should be active in the computation or inactive. The state is saved in the *Status* variable, which is sent to the master. Each then receives the initial data partition, which the master calculated using the status information from all the slaves. Each slave then receives its initial portion of the training set. At this point each slave checks its *Status* variable, and if it is inactive, it calls the function *inactive()* (explained later). All active slaves proceed to the main computational loop, where they set the variable *Iteration* to 1, and wait for the first message with tag equal to *Iteration*[1]. The *Iteration* message will tell the slaves what to do next, either redistribute data, or proceed with the computation. *Iteration* is incremented as soon as the corresponding message is received. If the *Iteration* message indicates to proceed with the computation, each slave receives a copy of the neural-net from the master, applies the net to its exemplars to calculate a partial gradient, checks the load on the local processor and sets *Status*, sends *Status* and the partial gradient to the master, and finally returns to the beginning of the computational loop by waiting for the next *Iteration* message. Note that *all* slaves proceed to wait for the next message, even slaves whose *Status* indicates they should be withdrawn. The need to redistribute data will be globalized by the next *Iteration* message.

When the *Iteration* message indicates that the slaves should redistribute data, they all call the function *redistData()*. For historical reasons, *redistData()* is simplistic and makes no attempt to minimize the amount of data that will be communicated. This is a known area for improvement in the application. For these reasons the details of *redistData()* are not explained, only the general functionality. First each slave receives a copy of the entire new data partition form the master. Each slave then calculates if any exem-

---

1. In PVM all messages are tagged with an integer tag and can be uniquely identified. All messages in a broadcast have the same tag.
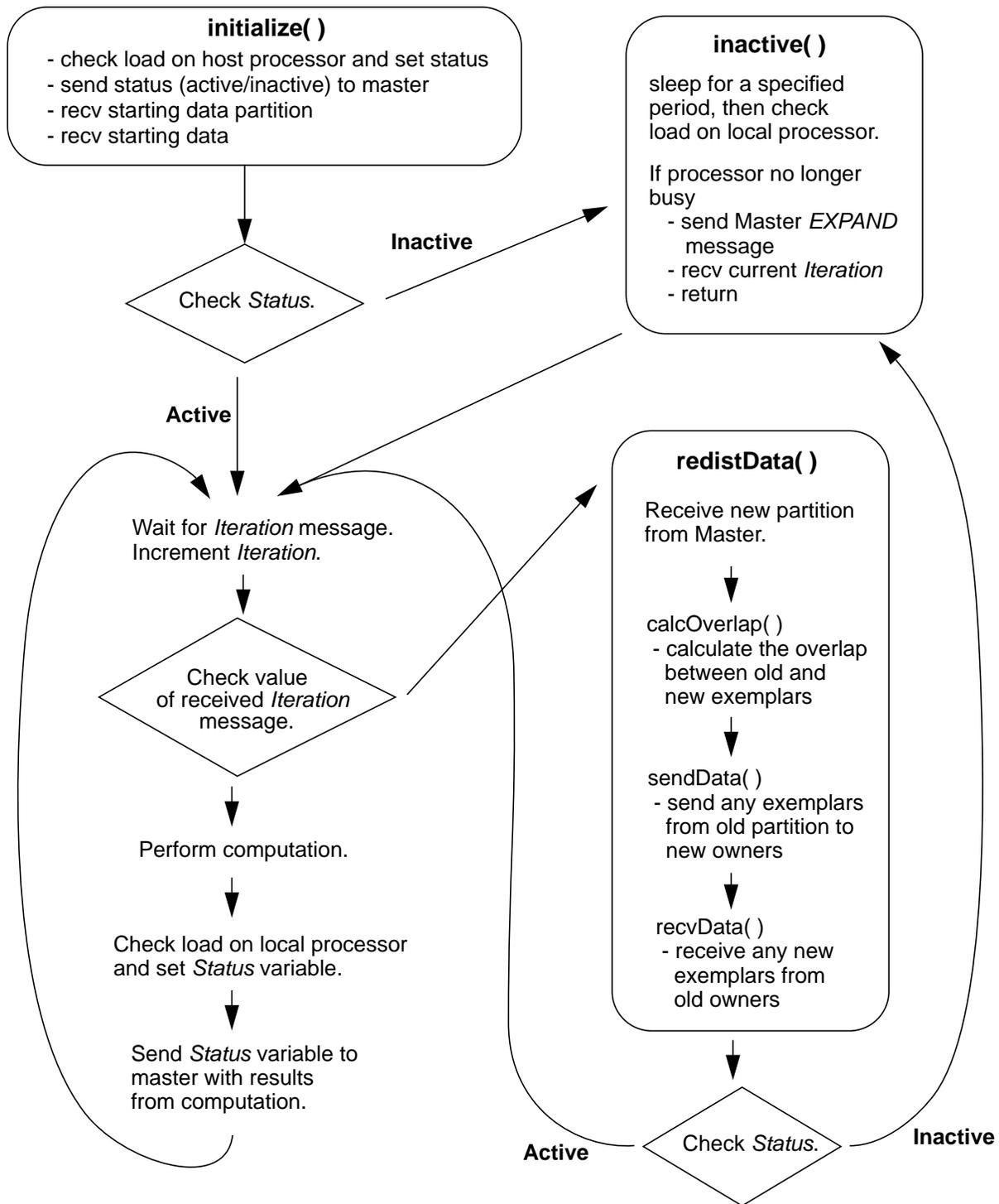
initialize( )

- check load on host processor and set status
- send status (active/inactive) to master
- recv starting data partition
- recv starting data

inactive( )

sleep for a specified period, then check load on local processor.

If processor no longer busy
- send Master *EXPAND* message
- recv current *Iteration*
- return

Inactive

Check *Status*.

Active

Wait for *Iteration* message.
Increment *Iteration*.

Check value of received *Iteration* message.

Perform computation.

Check load on local processor and set *Status* variable.

Send *Status* variable to master with results from computation.

redistData( )

Receive new partition from Master.

calcOverlap( )
- calculate the overlap between old and new exemplars

sendData( )
- send any exemplars from old partition to new owners

recvData( )
- receive any new exemplars from old owners

Check *Status*.

Active

Inactive

Figure 2: *Algorithm for ADMopt Slave*

plars in its old portion are also in its new portion. These "overlap" exemplars are neither sent to anyone, nor overwritten with received exemplars. Each slave then calculates which exemplars it must send to whom. After performing the required sends, each slave then calculates and performs the required receives. Finally, each slave checks *Status*. Active slaves return to the main computational loop. Inactive slaves, which are the ones being withdrawn, call *inactive()*.

The *inactive()* function is very simple. It simply loops over a host check followed by a sleep statement until the host is idle. Then it sends a special *Expand* message to the master, receives the current value of *Iteration* from the master, and returns. The duration of the sleep is currently a compile time constant, and is usually set at one minute. The use of inactive() allows withdrawn slaves to be as unobtrusive as possible.

Figure 3 shows the algorithm used by the master, which is considerably simpler than the slave algorithm. The master also goes through an initialization phase. First it starts slave processes on all machines. It then receives and stores the initial slave status information, which it uses to calculate the initial partition. Next the master sends the partition to each slave along with its portion of the data. It then sets *Iteration* to 1 and sends all active slaves the first *Iteration* message, which always tells the slaves to continue with the computation. Then master then moves on to its main computational loop. It formulates an neural net and sends it out to all active slaves. It then waits for the partial gradients and new *Status* information from each active slave. Once the master has heard from all active slaves, it then checks for any *Expand* messages from previously inactive slaves, which are added to the list of active slaves. If any *Expand* messages were received, or the *Status* from any active slave indicates it needs to be withdrawn, the master calls the function *calcNewPartition()*. Otherwise it increments *Iteration*, sends out the *Iteration* message telling the slaves to continue computing, and continues with the main loop.

When *calcNewPartition()* is called the master sends an *Iteration* message to all active and withdrawing slaves, then increments *Iteration*. A new partition is then calculated and sent to the same slaves. The withdrawing slaves are now removed from the active list.

Withdrawal events must be handled as quickly as possible if the application is to be unobtrusive. Note that any slave which may need to be withdrawn is in the active list. The master is already forced to wait for communication from all active slave because it needs their results. Thus a withdraw event cannot be missed. It is not as important to handle expansion events immediately. The master only checks for *Expand* messages in a non-blocking manner. If there are *Expand* messages to process, it does so. Otherwise it immediately continues. The master is not be required to wait on the inactive slaves, because an inactive slave may be running on a heavily loaded machine, and thus be slow to respond. There is no point in waiting for a possible *Expand* message from a slave that cannot be expanded.

4.3.1 Performance in the Quiet Case

If ADMopt is to be efficient, the quiet case must be efficient. Efficiency is defined as the percentage of the sum of the sequential performances of the participating machines the parallel computation is getting. Efficiency is calculated as follows. The speed of each machine running sequential opt on each data set using a standard neural-net is measured in terms of exemplars processed per second. This speed information is stored in a file which is read by ADMopt. When ADMopt is running, the master times each iteration and calculates and overall speed (in terms of exemplars per second) which the parallel application achieved for that iteration. Call this value *parallelSpeed*. The master then adds all the sequential speeds for the participating machines to find *sequentialSpeed*. Efficiency is *parallelSpeed* divided by *sequentialSpeed*.

The program has been run repeatedly on the worknet described in section 2. Using a 40 MB training set the computation can achieve a sustained computational speed of more than 74 MFLOPS, and an efficiency of 80%.

Figure 4 shows the efficiency of the calculation as a function of the number of machines. The graph shows the efficiency achieved using two different sized neural networks on a 40 MB training set. The size
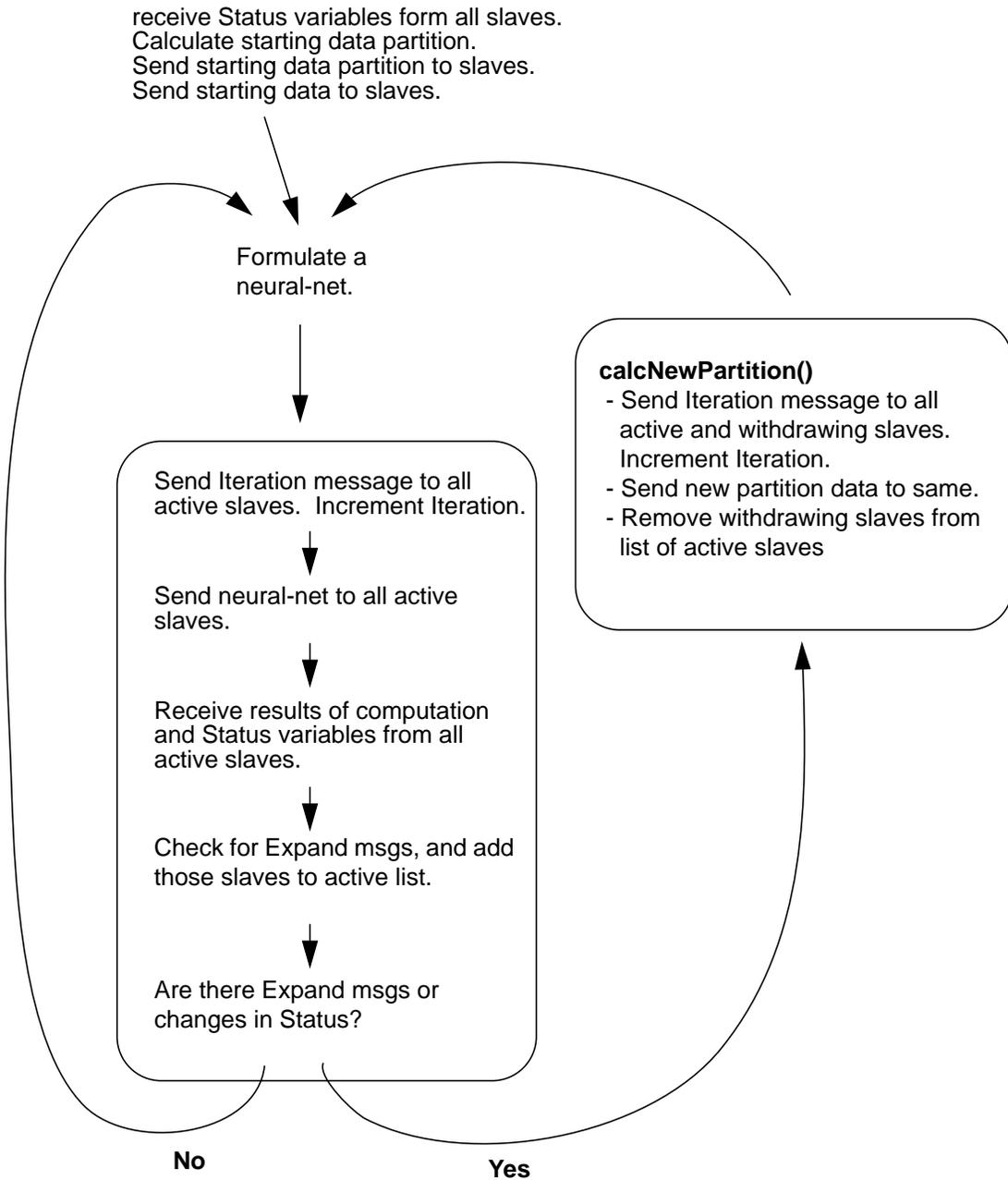
receive Status variables form all slaves.
Calculate starting data partition.
Send starting data partition to slaves.
Send starting data to slaves.

Formulate a
neural-net.

calcNewPartition()
 - Send Iteration message to all
   active and withdrawing slaves.
   Increment Iteration.
 - Send new partition data to same.
 - Remove withdrawing slaves from
   list of active slaves

Send Iteration message to all
active slaves. Increment Iteration.

Send neural-net to all active
slaves.

Receive results of computation
and Status variables from all
active slaves.

Check for Expand msgs, and add
those slaves to active list.

Are there Expand msgs or
changes in Status?

No

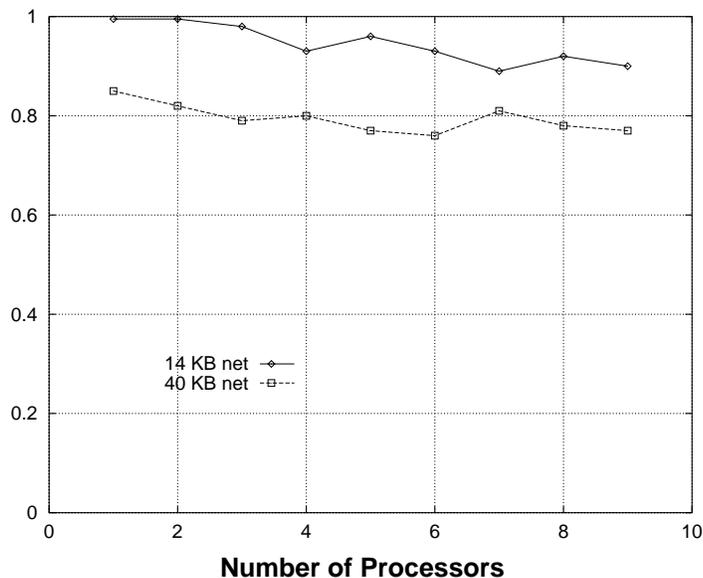**Yes**

Figure 3: *Algorithm for ADMopt Master*

Figure 4: *Efficiency vs. Number of Processors. ADMopt run on 2 to 9 machines using a 40 MB training set and neural-nets of two different sizes. The computation using the larger network spends a larger percentage of time communicating. The 40 KB network is the size which would actually be used by the speech researchers.*

of the neural-net determines the number of calculations which must be performed on each exemplar, so using a larger neural net will increase the computation time each iteration, thus *increasing* the ratio of computation to communication. Recall that the neural-net is a matrix which is being broadcast to all the slaves at each iteration; the slaves then send a gradient of the same size as the neural network back to the master. Thus the size of the network also determines the amount of communication performed each iteration, so using a larger net also serves to *decrease* the computation to communication ratio. Figure 4 shows that the computation is noticeably less efficient when using the larger network, and this points to the communication as the bottleneck. The non-scalability of the program is due, at least in part, to the non-scalability of the ethernet which connects the machines. Ethernet is simply not a scaling network. There is only one physical wire over which all messages are sent, so as the amount of communication increases, the number of packet collisions increases, thus the increasing overhead and lowering efficiency of the computation. Preliminary monitoring of the number of collisions seen by the workstation running the master indicates that the number of collisions is directly proportional to the size of the neural-net.

4.3.2 The Impact of Migration

If ADMopt is to be efficient, the computation must also maintain efficiency over migrations. Figure 5 shows the impact of a two controlled migration events, the withdrawal from and later expansion to a single processor, on the 70+ MFLOP calculation. The computation quickly rises to a peak speed which it maintains through the ninth iteration. During the tenth iteration, one of the processors was loaded down with other work, so as to trigger a migration event. The speed drops dramatically as the calculation waits on the one slowed processor. The actual size of the drop (and length of the wait) depends on how heavily loaded the withdrawing machine is. The next iteration is less efficient as well because the slaves must stop computing to redistribute the data. The computation then continues with eight machines and quickly rises to a new, albeit lower peak speed. When the withdrawn machine becomes available again, the computation again must slow to redistribute. Note that the drop in speed for expansion is not as great as the drop for withdrawal. This difference is consistently seen, and is due to the fact that withdrawal is slowed down by
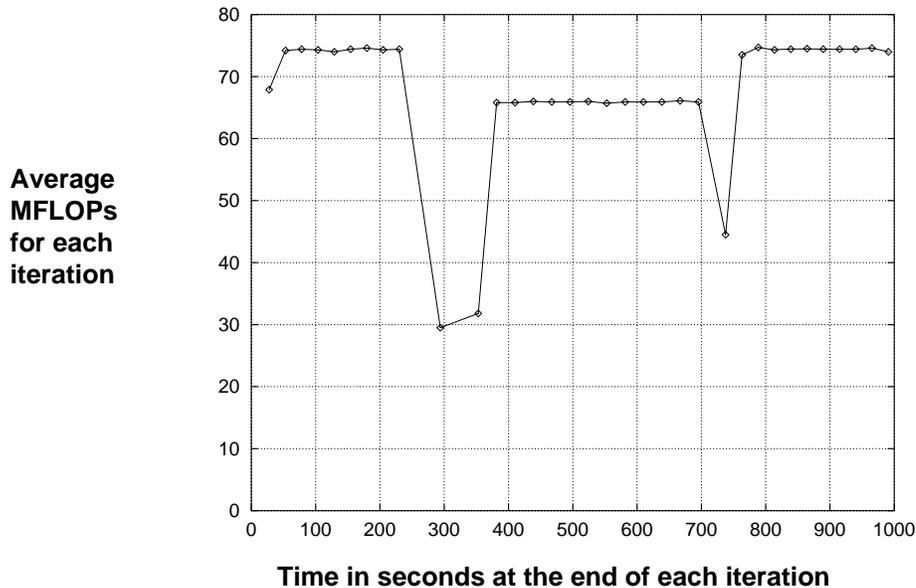
Figure 5: *Impact of migration on the computation. A migration event occurs between the ninth and tenth iterations, causing a dip in speed. The system then pauses to redistribute and continues computing at a new speed. When the machine becomes available again, the system again adapts and continues at its former rate.*

the heavily loaded withdrawing machine. When expansion occurs, the redistribution is not competing with any other user work, and so proceeds more quickly. Figure 6 shows that the system maintains efficiency over the two migration events. There are decreases corresponding to the two migration events, but the efficiency quickly recovers. This is as expected.

ADMopt is not yet unobtrusive. As currently implemented, the host check is performed at the end of each iteration. So, if a workstation owner tries to reclaim a workstation which ADMopt is utilizing, the current iteration must finish before ADMopt even begins to withdraw from the machine. The withdrawal shown in figure 3 takes almost a full minute. This is a known area for improvement and is discussed under Future Work.

5.   Other Approaches

The Distributed Systems Research Group (DSRG) at OGI is investigating several methods for adapting computations to the dynamic nature of a worknet. The DSRG is currently pursuing two systems software approaches which have the distinct advantage of being transparent to the user and application programmer. This is quite the opposite of ADM. There are also alternative user-level approaches which are commonly used to solve load balancing problems, which are applicable to this problem space. The following subsections compare these various methods in the context of the four requirements.

5.1   Transparent Approaches

In this context, *systems software* is defined as a software layer which makes adaptation transparent to the application. The PVM system itself cannot do anything to address the migration requirements because PVM application processes are static, UNIX processes. Once started on a certain machine, they cannot be moved to another machine by PVM. A conceptually simple extension to PVM is the addition of a mechanism to allow a process to transparently migrate from one host to another. Along with the addition of a host monitor, this would form the foundation for "reactive" PVM process migration. The PVM system
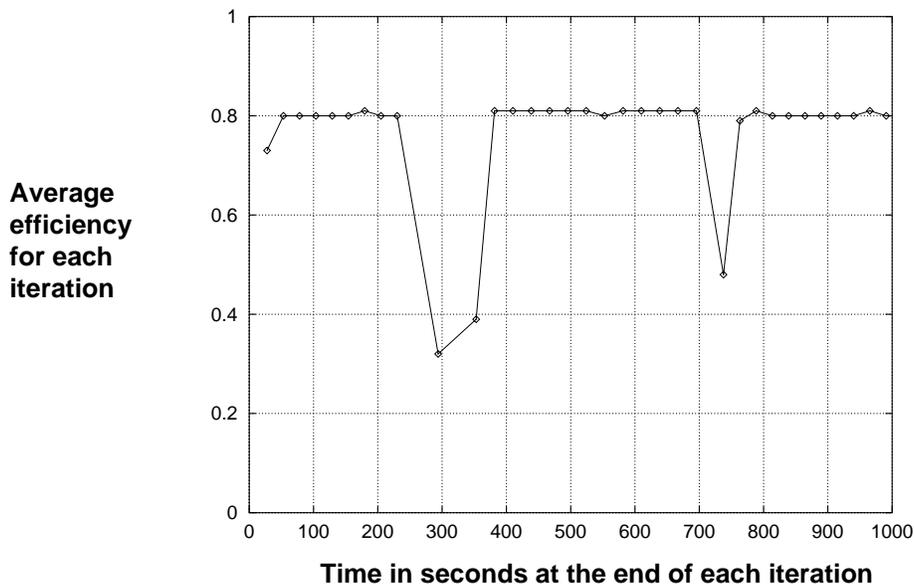
Figure 6: *Impact of migration on efficiency. These efficiency data correspond to the MFLOPS data of Figure 5.*

itself could monitor the status of the various components of the virtual machine and move processes when the situation demands. This is an ongoing research project at OGI under the title of Migratable PVM (MPVM). This method has the significant advantage of transparency from the perspective of the user. It satisfies the migration requirements by stopping the process on a withdrawing workstation, transferring the entire process to a new machine, and restarting it. MPVM only allows MPVM only allows redistribution at the granularity of a full UNIX processes. This may be too coarse to allow reasonable efficiency. There are no small units of work to redistribute evenly among the remaining processors, only a fixed number of large UNIX processes. So if a machine must be withdrawn, the load from this application on some other machine may need to be doubled. If the computation is started on four machines and a fifth becomes available, MPVM cannot give the new machine any work. The redistribution requirements are thus unsatisfied.

The granularity issue can be addressed by overdecomposing the problem. Instead of starting one processes on each machine, start several processes on each machine. Each process will contain a smaller portion of data and will be smaller in size, and there will be a larger number of them to shuffle around. This will allow work to be redistributed at a finer granularity than before overdecomposition. However, it will increase the amount of interprocess communication, which is expensive under PVM. Thus there is a trade off between redistribution granularity and computational efficiency. MPVM with overdecomposition also increases the number of heavyweight context switches as each processor multitasks among the several application processes it now has. Both the additional heavyweight interprocess communication and context switches are overheads that were not present before overdecomposition. There is also a limitation to how MPVM can migrate processes. It can only migrate processes between machines of the same architecture. The worknet thus becomes divided into disjoint pools of machines among which MPVM can migrate processes, and thus redistribute work. If a pool which consisted of many machines when the computation began, shrinks to only a few machines, those few machines of that type may become overloaded and the whole computation will slow down.

MPVM satisfies the migration requirements with restrictions regarding heterogeneity, but does not allow fine grain redistribution. The addition of overdecomposition improves its redistribution precision, but incurs the overhead of additional communication and context switches. These shortcomings can be

addressed by a different system-level extension to PVM: user level processes. A system of this type, called ULP, is also an ongoing research project within the DSRG. ULP would allow a number of computational threads to exist within a single UNIX process, with one UNIX process per processor. These lightweight user-level processes could be context switched more quickly than full UNIX processes. They could also be easily migrated between homogeneous machines. Because they represent a smaller unit of work, the user-level processes facilitate fine grained redistribution, either dynamic or static. The shared aspect of their execution would allow local communication to be very efficient, thereby largely negating the penalty of increased local communication. However, the large number of user-level processes may generate a larger amount of remote communication than the equivalent single process because the neighbors of any thread will not necessarily be on that same processor. Also, these user-level processes cannot be migrated across platforms, and so do not allow heterogeneous withdrawal and expansion. Since threads can only be migrated within homogeneous pools, even for redistribution purposes, dynamic work redistribution is restricted as in MPVM.

An alternative to these transparent approaches is to take an application level approach. Application level approaches heterogeneity issue by distributing work as data rather than as processes. Migration takes the form of data redistribution. Data can be moved from one machine to another and across platforms with relative impunity. Faster processes can be given more data to compute over while slower machines are given less. These methods employ one process per processor, and so avoid extra context switches. They also eliminate the extra communication required by overdecomposition. The most serious disadvantage is they complicate the process of writing the application.

5.2 Application Level Approaches

The worker-coordinator programming model is a popular application level method for solving the traditional load balancing problem. The worker-coordinator model (WC) is conceptually simple, and is mentioned in [GB+93] as the pool of tasks paradigm and covered in several variations in [GKR91] under source initiated load balancing algorithms. WC is often programmed in a master slave style, but not all master slave programs are WC. The WC model implies very specific behavior from the master and the slaves. The central premise is that processors request work when they become idle. One process serves as the coordinator. It starts up a number of worker processes, gives each an initial "chunk" of work, often referred to as a task, then handles future requests for work and collates results. It does not participate in the main computation. Each worker process requests another task whenever it finishes the current task. Thus faster processors will request work more often and thus perform a larger portion of the computation. Fast processors, and thus the computation as a whole, is not required to wait on the slowest processor (except for the worst case where the slowest processor grabs the last task which it slowly computes while the other processors wait for the final result). Realize that this model cannot remove work from slow processors, nor will it wait for loaded processors to become free before giving them more work. It only decreases the rate at which loaded processors are given more work. It also has no way to add new processors to the computation beyond the original starting number. The migration requirements are therefore not met, and WC programs will never become unobtrusive. A critical decision when writing an application in a WC style is determining the task size. As with overdecomposition, there is a trade-off between precision of work distribution and computational efficiency. If the task size is too small, fast processors may spend more time requesting and receiving data than they spend computing. If the task size is too large, the computation may end up waiting on slower processors. As long as an appropriate task size can be found, the redistribution requirements are satisfied. Determining the appropriate task size in WC is similar to determining how often to call the host check function in ADM. Both influence computational efficiency.

If the WC model is made a little more sophisticated, it can fulfill more of the requirements. The

worker-coordinator model does an acceptable job of giving the correct amount of work to each machine. So, if the migration requirements can be met, the redistribution requirements will follow naturally. Often, worker processes just blindly apply an algorithm to any data they are given. Whenever this condition is true, the following ideas can be applied. First, the method requires a host monitor function which can detect the state of any workstation in the worknet. Expansion can be satisfied by simply creating a new worker on the expanding machine. Withdrawal can be satisfied by killing any obtrusive worker, as long as its work is given to someone else. So, when the host monitor informs the coordinator that some worker is now obtrusive, the coordinator can kill the offending worker and give its work to another processor. Likewise when a new workstation becomes available, the coordinator can create a new worker on the new processor and give it work to do. Now all four requirements are met. This extended worker-coordinator will not work unless the application allows for the "blind" workers. A blind worker does not need any information about the overall computation other than its current task. This is often true of data parallel applications. Also, WC requires a more powerful host monitoring program than ADM, because WC's host monitor must be able to inform the coordinator of the state of any node in the system even if there is no worker on that node. If these two criteria are met, WC meets all four requirements.

WC is actually closely related to ADM in many subtle ways, and not all of the distinctions have been worked out. Many of the mechanisms which can be added to one can also be added to the other. For instance, ADM could kill off obtrusive processes if it had the more powerful host monitor. However, ADM would need to first evacuate the obtrusive process's data, or there would have to be another copy available. Likewise, WC could use ADM's host monitoring function if it left a skeleton running on all processors. More interestingly, ADM almost seems like a push-driven method, while WC is a pull-driven method. WC waits for processes to request data before it is sent. ADM tries to predict which machines will need more data and gives it to them before they need it. ADM moves all the data when it gives each processor an initial partition, and moves potentially large portions whenever a migration event occurs. The WC method moves small amounts of data constantly throughout the computation. The idea behind ADM is that it does not do any data movement during the quiet case, but once migration events occur, it is not clear which method will move more data overall. In WC there is a trade off is between computational efficiency and communication overhead, and is centered on task size. In ADM the trade off is between computational efficiency and withdrawal responsiveness, and involves the temporal granularity of the host check, which is determined by a "task size" which is basically one iteration over the data set.

WC does have a weakness which has no obvious parallel in ADM. This weakness is that the coordinator is a potential bottleneck as the number of workers becomes large. The speed of the computation is potentially limited by the speed at which the coordinator can process requests from the workers. The coordinator bottleneck is not preset in ADM because there is no central process handing out work. The adaptation functionality is distributed among all the processes. ADMopt *is* written in a master-slave style with the adaptation functionality split between the master and the slaves. However, this split is a result of the specific application, not the ADM method. Opt contains an inherently sequential portion, the conjugate-gradient optimization. ADMopt performs this portion on a single node, then broadcasts the resulting neural network. The application could be written in a SPMD style where each process redundantly performs the conjugate-gradient optimization. The master-slave style is used for historical reasons.

6 Future Work

Three main areas of future work are foreseen. The first area involves improvements, optimizations, and further experiments ADMopt. The second is the automation of the ADM functionality by creating a library of ADM functions. Finally, ADM needs the adaptation decision making power of a global sched-

uler. Toward this end, we plan to establish an interface between PVM and DQS.

6.1 Experiments with ADMopt

The work done so far has not proved that ADM is a viable adaptation method. ADMopt achieves impressive speed on a quiet worknet, so it is certainly worth pursuing. The response of the program to controlled migration events is understood and is encouraging. Many areas for improvement have been identified. One of the key questions still to be answered is whether or not ADMopt can be made to be unobtrusive.

Since a weakness of ADM is obtrusiveness, strategies to reduce obtrusiveness could be tested on ADMopt. It is possible to call the host check function more often than after each iteration. However, interrupting the computation more often involves saving more information regarding the current state of the computation. This is a move toward MPVM, which captures all state and data of a process at once, and migrates the entire process. The current distinction between MPVM and ADM, is the level at which state is captured. MPVM is transparent because is captures state under the application level. This same low-level state includes architecture specific details (such as registers and program counters) and so prevents heterogeneous migration. ADM captures state at the application level, and so can migrate across platforms, but the application programmer must add the required functionality to the application.

The repartitioning itself needs to be improved. The current data movement code is naive, and should be replaced with a method that attempts to minimize data exchanges. An alternative to exchanging data via messages is to have each process read its new portion of the data from the original input file. This alternative would have the distinct advantage of freeing the obtrusive process from the need to send off all its data—it could immediately stop computing. This is possible with opt because the exemplars are never modified. However, it is *not* possible unless the data file is in a globally accessible location, or replicated on each machine. The message passing method does not require any shared files. Reading from a file may, or may not be faster than message passing, tests would have to be run.

Another way to avoid the cost of data movement is to give each process a full copy of the input data. Redistribution is then simply a matter of telling each process what portion over which it should compute. Repartitioning is very fast, and the obtrusive process can quickly stop computing. However, each process now requires significantly more memory, and the withdrawn process will occupy a large portion of its hosts swap space and this itself may prove to be obtrusive. Sending each process all of the input data would also increase the application start-up time: using a 40MB training set on nine machines would require 360MB of data to be transferred, instead of just 40MB. ADMopt was actually first developed in this manner before the data movement code was included, and the method is still a compile time option. Its effectiveness has not been investigated because PVM had difficulty broadcasting the required amount of data, and workstations were running out of memory.

6.2 Implementation as a Library

To alleviate the complexity of writing a parallel application using ADM, we propose capturing the functionality in a library. The library would include all the PVM calls and would present an interface to the programmer of a few, easy to use calls. This would remove the need to program explicit message passing.

The system would be implemented mach as PVM is, using a daemon which would run on each host and a library to be linked to application programs. The daemon would include the asynchronous host monitor and could facilitate globalization of local host state information. The provided functions would include the following:

signalHandler	a function which catches the signal from the host monitoring daemon and sets the redistributeFlag

withdrawCheck	a macro which expands into a check of the redistributFlag and a call to redistributeData when necessary
getGlobalData	a function which would implement the message passing necessary to return the latest copy of the global data structure used by the computation (for example, the neural-net in opt)
getNewPartition	a function called by redistributeData which performs the message passing and calculation of a new data partition
forMyData	a macro which expands into a for-loop over the correct indices which can be used by the programmer to perform the application algorithm on each processor's data portion
redistributeData	the function called by withdrawCheck which in turn calls getNewPartition then performs the necessary message passing to perform data redistribution
initialize	a function which does all necessary ADM setup, including establishing contact with the local ADM daemon and PVM, and acquiring the initial data partition and data

Details such as function arguments have yet to be worked out. The above list might very well change when implemented. Interestingly, the implementation of ADM as a library, makes it more similar to Piranha. The use of ADM becomes a process a structuring the application in a certain way and making calls to predefined functions which handle all the parallelism with the mechanisms hidden from the programmer.

A decent job has been done handling an application with a basically one-dimensional structure. To be truly useful, ADM would have to be applicable to two and three dimensional problems. The partition calculation and redistributeData functions would have to be capable of handling the corresponding data structures. If the application computes over a two-dimensional data structure such as an array, a hierarchical partitioning method can be used. This method would allow both static and dynamic adaptation. The basic method is as follows. Divide the processors into groups. The total computing power of each group is calculated, and the data set is divided along one dimension (into strips) according to the power of each group. Then each strip is subdivided according to the power of each machine in the group. See figure 7. This method can be extended to three dimensions.

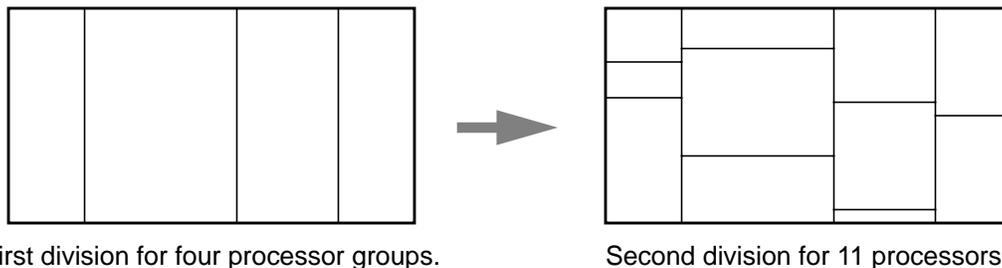First division for four processor groups.   Second division for 11 processors

Figure 7: *Illustration of hierarchical data division.*

With two-dimensional functionality, the application of ADM to problems such as a Laplace solver would be possible.

6.3 Interfacing with Other Systems

Section 4.1 mentioned that the policy decisions required regarding when the parallel computation should

adapt, could not easily be made by ADM. This is due to the distributed nature and "per application" nature of the information returned by the host monitor. Because of this, ADM is not very good at allowing several parallel applications coexist on the same worknet. What is needed is a higher entity which can control all ADM applications on a worknet; what is needed is a global scheduler. The DSRG is currently working on defining an interface between message passing systems such as PVM and the Distributed Queuing System [Rev92] (DQS).

The need for a global scheduler can be clarified by an example. One of the basic assumptions underlying ADM is that all other work on the worknet is local work on the nodes. What happens if there are other processes trying to adapt themselves to the same worknet? What if more than one ADM program is running? The programs will almost certainly interfere with one another. Consider two ADM programs running on the same worknet. Assume each manages to claim a disjoint set of workstations, but they are both monitoring all workstations. Some time later a workstation becomes idle. It is possible that one program could claim the workstation without the other one even noticing, but not likely. It is more likely that both programs will notice, and both will expand onto the workstation. When they next check the activity of that workstation, they will both notice that the machine is doing other work in addition to their own and will withdraw from it. This leaves the workstation free, and the process can repeat. This might very well cause both ADM programs to thrash—both will continually expand to and withdraw from this one node, spending enough time adapting to seriously impact computationally efficiency and thus overall execution time.

The aforementioned problem arrises because each ADM program is performing its own monitoring of the network, and the information thus gathered is inherently out of date. ADM programs make adaptation decisions based on the state of the worknet when they last checked it. That state may change before the program actually adapts. Therefore, when more than one ADM program is running, a higher level entity is needed to tell each program which machines to utilize so they do not interfere with each other. This entity could take the form of a global scheduler. The DSRG is just beginning to research the required interface between systems like PVM and global scheduling systems like DQS. DQS is capable of controlling the execution of a large number of programs running on a network, but it can only start and stop them, it has no means of moving them around. DQS and ADM could work together to harmoniously execute a number of adaptive parallel programs. DQS could be set up to run on the local worknet. Users would then submit their ADM jobs along with a specification file summarizing the resources needed by the job to DQS. DQS could select certain jobs to run based on their total computational needs and give each job a list of machines to utilize. This list of machines would have to be twofold: a list of all potential machines, and a subset of the first list on which to actually begin computing. A basic function of DQS is to continually monitor all machines in the system and to make intelligent decisions regarding which jobs should run where. So, DQS could inform an ADM job of the need to expand or migrate by giving it a new set of machines to utilize. ADM would then perform the necessary adaptation.

Cooperation like that described is necessary if ADM is to be widely used as a general method. Restricting users to one active ADM job per active on a worknet is unacceptable. More careful analysis and definition of the above interface is planned.

7 Related work

Dataparallel C is a parallel programming system which performs automatic load balancing and runs on worknets. It uses virtual processors as the unit of work which is distributed among the physical processors. It performs runtime load balancing by migrating the virtual processors, which include both code and data, from one physical processor to another, even on heterogeneous networks. However, it requires a special language and a more restrictive SIMD programming model. ADM is not restricted to SIMD programming,

and is thus slightly more general. Dataparallel C uses a global stack to capture process state. This global stack requires that processes execute in SIMD fashion. Like MPVM it is capturing state at a level beneath the application, but this low level information is kept above any architecture dependent features, so it can accomplish heterogeneous migration. ADM functions entirely at the application level and is not tied to any system structure like a global stack, and can be more general.

Piranha is a system for adaptive parallel programming which is implemented on top of Linda. Piranha is very similar to ADM in that the application programmer structures the application program in a certain way and writes certain functions which allow the application to adapt to a dynamic environment. There is an inherent trade-off between the efficiency of computation and the speed at which Piranha can withdraw from a workstation. One can be increased only at the expense of the other, and the programmer is required to strike a balance between the two. Like ADM, increasing withdrawal speed involves deceasing the temporal granularity at which work can be stopped. Decreasing this granularity involves saving more state information, as in ADM. Linda provides a parallel environment through shared associative object memories, called tuples, and operations on those objects. It takes the form of a coordination language which is combined with a computation language like C. The effect of Linda is something like distributed-shared memory, and the user does not have to write code to explicitly pass messages. Since ADM is directed at explicit message passing systems, Piranha handles a slightly different problem domain.

The Condor system from the University of Wisconsin is very successful at unobtrusively making use of idle workstations. It can transparently move a sequential job among homogeneous workstations to make use of idle cycles. The one restriction placed on the application programmer is that all file operations must be idempotent. In contrast to ADM, it supports neither parallel jobs nor heterogeneous networks, but its transparency is an advantage. The important difference between ADM and Condor is that Condor uses a rollback method of migration while ADM uses a "roll-forward" method. Condor creates a migratable image at regular intervals as the program executes. When a migration event occurs, the last stored image is used. Thus the migratable image was created *before* the migration event occurs. ADM's "roll-forward" method is as follows. When a workstation owner reclaims a workstations, ADM completes its current iteration, thus creating a migratable state *after* the migration event actually occurs.

8 Conclusions

The use of networks of heterogeneous workstations as parallel computers is a very attractive proposition. However, their use as a distributed parallel computer presents a number of challenges which need to be investigated. This paper has investigated those issues in the context of data parallel programming using explicit message passing. Four requirements were derived which we feel must be satisfied if data parallel applications are to adapt to the dynamic number of processors presented by the network of workstations. These requirements seem to provide a good template for use when comparing adaptive methods, but there is more which must be considered. They have proved to be necessary, but not sufficient conditions for a successful adaptation method. They have indicated the weaknesses of ADM, and allowed us to formulate possible improvements to the method.

ADM has been applied to an application. It has been shown to be efficient, both when the worknet is quiet and over single adaptation events. The performance gains have shown that the worknet is certainly a resource for computing power that should be exploited.

Coding the ADM application was not easy. It is not at all clear that application programmers will be willing to use this method if the work required cannot be reduced. Forming the essential functionality into a library should address this problem.

The obtrusiveness of the method still needs to be investigated. Tests should be run to see if changing

the frequency of the host check can reduce obtrusiveness, while maintaining computational efficiency. Basically, tests need to be run which evaluate how the systems behaves when the parameters of the approach are varied.

There are a wide variety of parallel programming tools, languages and models in existence, but few address the same problems as ADM. Dataparallel C and Piranha are two systems which do, and they realize different amounts of success. More work must be done with ADM before they can be accurately compared.

9   Acknowledgments

10   References

[BC89]   Etienne Barnard and Ronald Cole, "A Neural-net training program based on conjugate-gradient optimization", Technical Report CSE-89-014, Oregon Graduate Institute, 1989

[BLL]   Allan Bricker, Michael Litzkow, and Miron Liuney. Condor Technical summary. Computer Sciences Department, University of Wisconsin, Madison, WI.

[Bro92]   Eugene D. Brooks III. Massively Parallel Computing. The 1992 MPCI Yearly Report: Harnessing the Killer Micros, Massively Parallel Computing Initiative, August 1992.

[CG+]   N. Carriero, D. Gelernter, D. Kaminsky, J. Westbrook. Adaptive Parallelism with Piranha.

[CG89]   N. Carriero and D. Gelernter, Linda in Context. *Communications of the ACM*, 32(4), April 1989.

[DS92]   A. Deshpande and M. Schultz. Efficient Parallel Programming with Linda. *Supercomputing 92,* 1992.

[GB+93]   A. Giest, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. PVM 3.0 User's Guides and Reference Manual. February 1993.

[GK92]   D. Gelernter and D. Kaminsky, "Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha", Sixth ACM International Conference on Supercomputing, Washington, D.C., July 19-23 1992.

[GKR91]   A. Y. Grama, V. Kumar, and V. N. Rao. Experimental Evaluation of Load Balancing Techniques for the Hypercube. *Parallel Computing '91*, pages 497–513, Elsevier Science Publishers B. V., 1992.

[GS92]   G. A. Geist and V. S. Sunderam. Network-Based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.

[Kon93]   R. Konuru, J. Walpole, S. Otto. A User-Level Process Package for Concurrent Computing. Technical report in progress. Oregon Graduate Institute, 1993.

[LL]   Mike Litzkow and Miron Livny. Experience with the Condor distributed batch system. Computer Sciences Department, University of Wisconsin, Madison, WI.

[LLM88]   M. Litzkow, M. Livny, and M. Mutka. Condor—A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computer Systems*, pages 104-111, San Jose, CA, June 1988.

[NQ93]   Nenad Nedeljkovic and Michael J. Quinn, Data-Parallel Programming on a Network of Heterogeneous WorkstationsConcurrency: Practice and Experience, 5(4):257-268, June 1993.

[Rev92]   Louis S. Revor. DQS Users guide. Computing and Telecommunications Division, Argonne National Laboratory, September 1992.