Empirical Analysis of Overheads in Cluster Environments*

Brian K. Schmidt

V. S. Sunderam

Department of Math and Computer Science
Emory University, Atlanta, GA 30322, USA
Phone: (404) 727-5926; Email: {bs,vss}@mathcs.emory.edu

Abstract

In concurrent computing environments based on heterogeneous processing elements interconnected by general-purpose networks, several classes of overheads contribute to lowered performance. The most obvious limitations are network throughput and latency, but certain other factors also play a significant role. In an attempt to gain some insight into the nature of these overheads, and to propose strategies to alleviate them, empirical measurements of native communication performance as well as application execution performance were conducted, using the PVM network computing system. These experiments and our analyses have identified load imbalance, the parallelism model adopted, communication delay and throughput, and within-host overheads as the primary factors affecting performance in cluster environments. Interestingly, we find that agenda parallelism and load balancing strategies contribute significantly more to better performance than improved communications or system tuning. Drawing general conclusions on how these inefficiencies may be overcome is inadvisable because of the tremendous variability of many parameters in general purpose network environments; we therefore propose several potential approaches, including model selection criteria, partitioning strategies, and software system heuristics, to reduce overheads and enhance performance in network based environments.

1. Introduction

Cluster computing, or concurrent processing on collections of loosely coupled computer systems, is a rapidly evolving technology with tremendous potential for high performance applications. Several cluster-based systems are in widespread use for applications ranging from computational quantum chemistry to Monte-Carlo simulations and materials science calculations. In general, such systems permit a collection of networked machines to be used as a unified, general-purpose, concurrent computing resource. While

* This material is based upon work supported by the National Science Foundation, under Award No. CCR-9118787, and the Applied Mathematical Sciences Program, Office of Basic Energy Sciences, U. S. Department of Energy, under Grant No. DE-FG05-91ER25105.

hardware multiprocessors have become commercially available and are effective platforms for highly compute-intensive applications, networked collections of computer systems can complement or augment the processing capabilities of traditional parallel machines. Typically, most contemporary computing environments are network based, and include a variety of processing elements, including general purpose workstations, different kinds of multiprocessors, and a few special-purpose machines such as graphics processors or vector computers. Further, with the advent of very high-speed networks, real-time resource sharing with geographically remote computer systems is imminent.

The network-based concurrent computing approach has proven to be viable and effective from several different standpoints:

- Computing Power: By providing (controlled) access to a much larger and richer hardware base, network-based concurrent environments can increase application performance by significant factors. In addition to increasing the overall computing resources that may be accessed by individual users, network environments also enable the exploitation of specialized resources.

- Expandability: Incremental scaling of a network-based concurrent computing environment is usually straightforward, and network bandwidth limitations that pose the main obstacle to scaling by larger factors are on the verge of increasing by one or two orders of magnitude with the advent of fiber optics. While it is unrealistic to expect communication speeds comparable to hardware buses or switches, high-speed networks can provide sufficient capacity and performance for a variety of applications. Further, under the right circumstances, the network-based approach can be effective in coupling several similar multiprocessors, resulting in a configuration that might be economically and technically difficult to achieve with hardware.

- Application Heterogeneity: Many existing and projected applications (e.g. as mentioned in [1]) are composed of sub-algorithms that differ widely in the model of computation, programming language, and computing and data handling requirements. On typical networks with a wide mix of architectures and capabilities, such applications can benefit by executing appropriate sub-algorithms on the best suited[1] processing elements.

In recent months, several systems that support cluster-based concurrent computing have emerged, and are in production use at a number of sites for general purpose concurrent applications. Among these are the PVM system[2], Network Linda[3], and the Express environment[4]. The approaches adopted, the programming and development interface supported, and performance levels, vary among these and other comparable systems. However, for several classes of applications, these systems have demonstrated their effectiveness in terms of both functionality and performance, sometimes achieving performance levels matching or exceeding that of supercomputers or hardware multiprocessors. Clusters are therefore highly cost-effective, widely available, and provide straightforward access interfaces as well as development and monitoring tools. For these reasons, cluster computing is expected to receive widespread attention in the near future, and will continue to be a viable alternative, or complement, to traditional high-performance computing platforms.

While the performance levels observed in cluster environments are usually adequate and often very good, there is frequently a noticeable degradation — as compared to execution on the same collection of computer systems, but without external loads, and ignoring communication overheads. This is, of course, to be expected; clusters are typically composed of general-purpose systems, interconnected by networks that carry various other traffic. Nevertheless, we believe that many of the overheads in cluster computing can be reduced to varying extents, by using a combination of partitioning strategies, simple approaches to scheduling and load balancing, and system level tuning. This paper describes our initial efforts to understand, in greater depth, the factors affecting cluster computing performance, and investigate strategies to improve performance in network based concurrent computing systems. We begin with an empirical analysis of the degree to which different factors affect cluster performance. Based on this analysis, we propose

1 in terms of computing paradigms, application languages, and resource requirements several potential approaches for improving performance, including partitioning and load balancing strategies, novel network protocols, and alternative software architectures. Preliminary findings from the use of such approaches are presented, but owing to the high degree of variability in nearly every component of network based concurrent systems, we believe that it is yet premature to suggest "optimal" techniques for performance enhancement. We do discuss qualitatively, the applicability of these strategies, both with respect to varying cluster platforms as well as to different classes of applications.

2. System Framework

The issues addressed in this work apply reasonably well to several types of distributed computing systems and infrastructures that support network-based concurrent computing. However, since our analyses, proposed techniques, and experimentation utilized the PVM system as their basis, the remainder of this paper will assume this environment. Details regarding the PVM system, experiences with its use, and descriptions of supporting toolkits may be found in [2,5,6]; in this section we describe the functions supported by PVM, parallel computing models that may be deployed, partitioning and scheduling strategies, and quantitative measures of primitive facilities.

2.1. The PVM Infrastructure

The PVM system essentially emulates a general-purpose concurrent computing framework on a networked collection of independent computer systems. A system level process executes on each machine in a user-configurable pool of processing elements, and through cooperative distributed algorithms, permits this collection of machines to be utilized as a coherent concurrent computing resource. Applications access this virtual machine via library routines embedded in imperative procedural languages such as C and Fortran. Support is provided for process management, for communication via connectionless and connection-oriented message passing, for synchronization based on barriers or variants of rendezvous, and for auxiliary functions. Internally, these library routines interact with the PVM service provider on each host (i.e. the PVM *daemon*); collectively, the daemons on machines in the host pool emulate a virtual concurrent machine. A schematic of the PVM system infrastructure is shown in Figure 1.

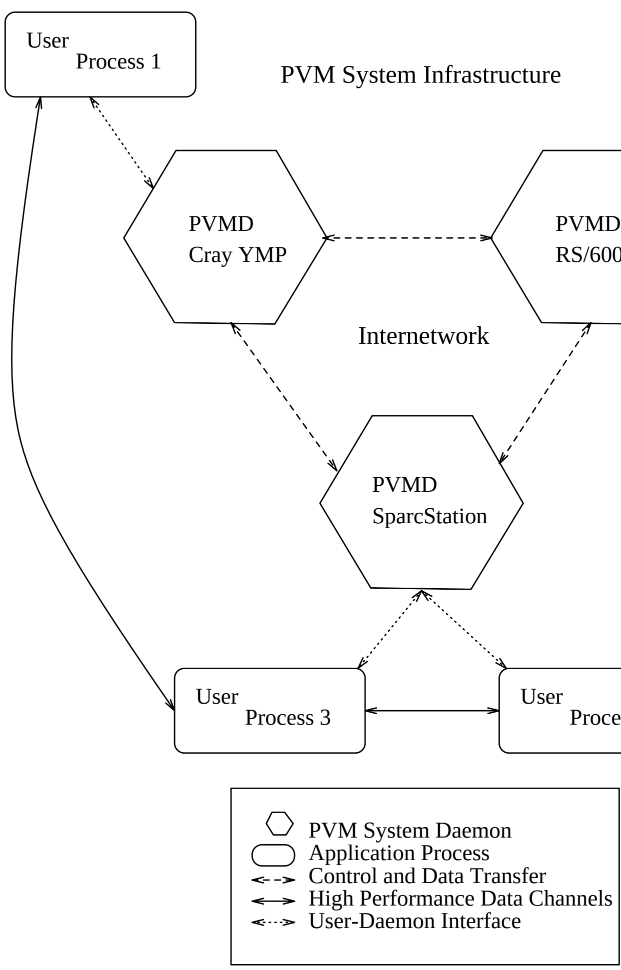

Figure 1: PVM Architectural Overview

An ‘instance’ of an application subtask or ‘component’ (realized as a process), is the unit of computational abstraction in the PVM system. Each process is an executing instance of an application component, where a component is a domain-specific module amenable to SPMD execution. Therefore, an application consists of several components, each of which may be dynamically manifested as multiple, concurrent instances that cooperate within and across component boundaries. Instances, therefore, are independent sequential threads that may spawn or terminate other instances, synchronize with one, several, or all instances, and exchange data among themselves. An illustrative example of this computing model is shown in Figure 2.

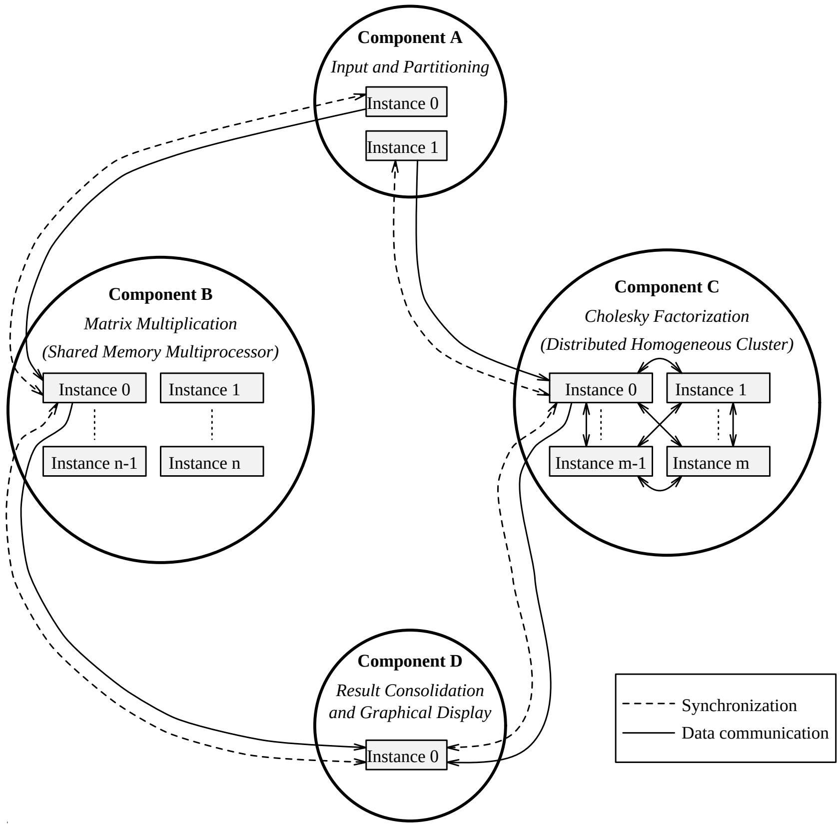

Figure 2: Example Structure of PVM Computing Model

Instances communicate via the use of messages; each message may contain multiple, typed, data areas. These message segments are built by provided library routines in a machine independent manner, and are optimized by the system to avoid unnecessary conversions when possible. Data transfer is always message based, but separate interfaces are provided for "packetized" and "stream" data, thereby supporting bursty as well as large volume, continuous communication. Message exchange is asynchronous, in that a sending process may continue execution prior to physical message reception by the destination process. Probe mechanisms are provided so that receiving instances need not suspend execution pending message arrival. Synchronization is provided in the form of named barriers that permit specification of a quorum, and by event rendezvous mechanisms.

2.2. Application Development

Applications utilizing the PVM infrastructure are programmed in a manner similar to other distributed memory multiprocessors, e.g. the hypercube. However, there are several significant differences:

- Process orientation: Under PVM, applications may use as many *processes* as required or base the number of instances on problem characteristics. Unlike processor-oriented parallel machines, instance to processor mapping need not be one-to-one.

- Multiple component execution: Inherent support is provided for cooperative execution of multiple application modules (i.e. different programs) each of which may be replicated; on hardware multiprocessors, cumbersome programming or manual intervention is required to accomplish this.

- Exploiting heterogeneity: Since multifaceted virtual machines can be configured within the same framework, the potential for siting subtasks to the best suited architectures is significantly enhanced.

- Logical interconnection: For both synchronization and communication, direct logical visibility is provided between arbitrary sets of component instances. Interaction is via a logically complete graph, and synchronization points may be assigned abstract symbolic names.

PVM supports the explicitly parallel computing model, particularly with respect to partitioning and scheduling. Graphical tools are under development [7] to ease parallel program development and profiling, but the principal technique for representing concurrency is via appropriate use of host language control flow constructs, augmented by synchronization and communication primitives. Therefore, with regard to workload allocation — a critical factor influencing performance, as later sections show — our intention is that the findings and proposed strategies presented in this paper form the basis for integral system support of partitioning and scheduling within PVM.

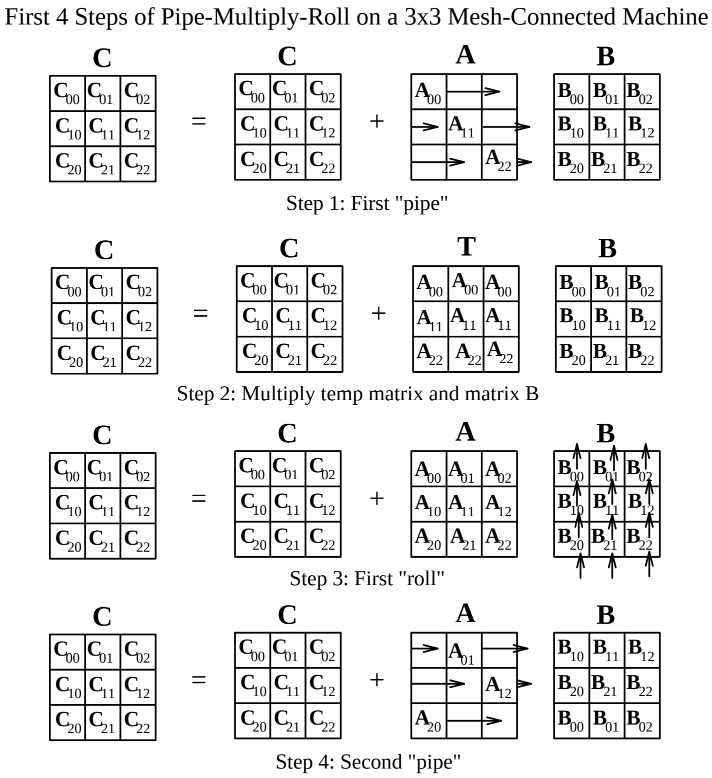

Figure 3(a): Matrix Multiplication using "Pipe-Multiply-Roll"

2.3. Application Programming paradigms

We begin with an illustration of how different concurrency models may be implemented under PVM. Using well known examples, we discuss common partitioning, scheduling, and organization techniques for concurrent processing. The examples that have been selected are standard textbook problems and are well understood; yet they exhibit characteristics that embody in abstract terms, the behavior of many applications. They are also fundamentally different with regard to communication patterns and computation-to-communication ratios — two very important factors affecting performance in message passing concurrent systems. In the following subsections, we present empirical results, and identify areas in which novel strategies may be employed to obtain gains in development ease as well as performance.

```
{Matrix Multiplication using Pipe-Multiply-Roll algorithm.}

{Processor 0 starts up other processes}
if (<my processor number> = 0) then
    for i := 1 to MeshDimension*MeshDimension
        initiate(<component name>,i)
    endfor
endif

forall processors Pij, such that 0 <= i,j < MeshDimension
    for k := 0 to MeshDimension-1
        {Pipe.}
        if myrow = (mycolumn+k) mod MeshDimension
            {Send A to all processors Pxy, such that x=myrow, y<>mycolumn}
            for y := 0 to MeshDimension-1, y <> mycolumn
                snd(<component name>,(Pxy, where x = myrow),999)
            endfor
        else
            rcv(999)                                         {Receive A}
        endif

        {Multiply.  Running totals maintained in C.}
        Multiply(A,B,C)

        {Roll.}
        {Send B to processor Pxy, such that x = myrow-1, y = mycolumn}
        snd(<component name>,(Pxy, where x = myrow-1, y = mycolumn),888)
        rcv(888)                                             {Receive B}
    endfor
endfor
```

Figure 3(b): Algorithm Outline for Matrix Multiplication

Matrix Multiplication

A classic technique for matrix multiplication on distributed memory architectures is the "Pipe-Multiply-Roll" algorithm described in [8]. This method multiplies matrix sub-blocks locally, and uses row-wise multicast of matrix **A** subblocks in conjunction with column-wise shifts of matrix **B** subblocks as shown in Figure 3(a). The communication pattern is therefore regular, and the computation is evenly balanced when a square grid of processing elements is used. Figure 3(b) outlines, in pseudo-code form, the PVM program used to perform block matrix multiplication.

Sorting Algorithms

In contrast to the matrix multiplication algorithm, most sorting algorithms for distributed memory architectures have a high communication to computation ratio. Furthermore, while the communication pattern is regular, it is not symmetric in many algorithms. Owing to these differences, typical DMM sorting algorithms can be considered to belong

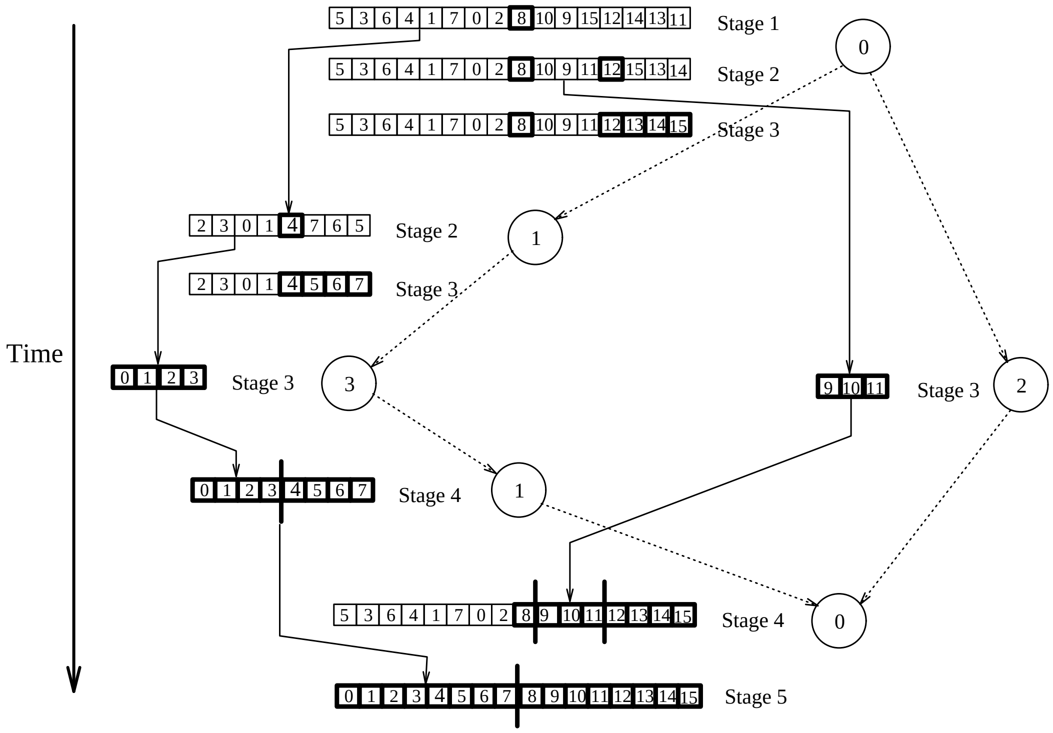

Figure 4(a): Split-Sort-Merge Algorithm on 4 Tree-Connected Processors

to a different application category in terms of structure and communication patterns. In this work, we use an adaptation of existing sorting algorithms [13,14,15,16] as a representative example of such a ("regular tree-structured") class, and perform empirical measurements of its performance in cluster environments. Figure 4 illustrates the method diagrammatically, as well as in the form of an algorithm skeleton.

Pure Data Parallelism

The applications presented thus far possess non-trivial, although straightforward, communication patterns. On the other hand, several applications exhibit what has been referred to as "embarrassing" parallelism. Essentially, the problem can be partitioned into completely independent portions, the algorithm applied to each, and partial results combined using simple combination schemes. However, this model permits dynamic load balancing, using *agenda parallelism* [10], thereby permitting processing elements to share the workload unevenly. Computing the Mandelbrot set is a commonly used representative of this category of parallel programs. We use this example in our experiments; a schematic outline of our implementation is shown in Figure 5.

```
{Processor 0 starts up other processes}
if (<my processor number> = 0) then
    for i := 1 to NumProcs
        initiate(<component name>,i)
    endfor
endif

{ Partition list based on a broadcast tree pattern. }
for i := 1 to N, such that 2^N = NumProcs
    forall processors P such that P < 2^i
        if P < 2^(i-1) then
            midpt: = PartitionList(list);
            {Send list[0..midpt] to P XOR 2^i}
            snd(<component name>,(P XOR 2^i),999)
            list := list[midpt+1..MAXSIZE]
        else
            rcv(999)                               {receive the list}
        endif
    endfor
endfor

{ Sort remaining list. }
Quicksort(list[midpt+1..MAXSIZE])

{ Gather/merge sorted sub-lists. }
for i := N downto 1, such that 2^N = NumProcs
    forall processors P such that P < 2^i
        if P > 2^(i-1) then
            snd(<component name>,(P XOR 2^i),888)  {Send list to P XOR 2^i}
        else
            rcv(888)                               {receive temp list}
            merge templist into list
        endif
    endfor
endfor
```

Figure 4(b): Algorithm Outline for Split-Sort-Merge

The primary objective in parallel processing is faster execution by using multiple processing elements that work cooperatively on a single problem. The level of efficiency in speeding up computations is, of course, dependent on several factors, ranging from inherent non-parallelism in the algorithm to the overheads of communication and synchronization among the multiple processors. In cluster-based environments, there are also external influences, since, in general, both the network and the processors may be in use by other applications. Therefore, analyzing communication overheads must take extraneous traffic into account and must also factor in the variable nature of network characteristics in terms of delay and throughput. Similarly, in trying to formulate computation requirements, unrelated processes executing on (usually) general purpose computers can contribute to unpredictable and large variations.

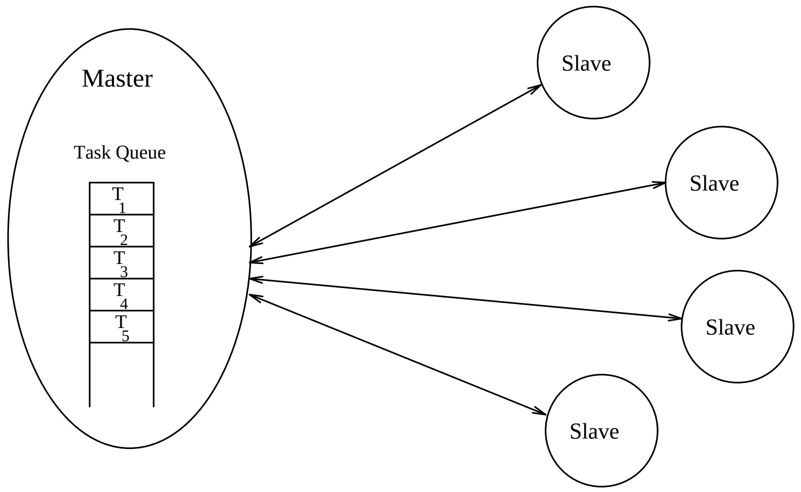

Figure 5(a): Schematic for Typical Data Parallel Applications

Since the above factors are so varied, constructing a precise model for efficiency in cluster environments appears to be a difficult, and possibly intractable, problem. In this project, we attempt to begin this task; our intention is to first verify if an accurate model can be constructed by making several simplifying assumptions. We use as our platform, a lightly loaded local area network where the computing elements are similar workstations, which, while not dedicated, do not execute external compute-intensive applications. By "lightly loaded network", we informally mean one which operates at 15-20% of capacity, and on which the traffic is fairly constant. External use of each workstation is limited to executing a window server with 5-10 window system clients, of which typically one is in interactive use at any instant. We would like to point out that this scenario is typical of that encountered in the vast majority of student laboratories, other university settings, and some industrial and commercial installations. However, we intend that these assumptions will nevertheless only be used to acquire a first-cut understanding of the issues involved in analyzing cluster environments.

Under the assumptions of low, and near-constant external network loads, we attempt to parametrize the communications overheads for cluster-based computing on local networks. We use as our basis, the conventional approach adopted in analyzing communications for most message-passing, distributed memory multiprocessors [11]. This approach is to define communication time as a simple linear function of the number

```
{Master Mandelbrot algorithm.}

{Initial placement}
for i := 0 to NumWorkers - 1
    initiate(<worker name>,i)                {Start up worker i}
    snd(<worker name>,i,999)                 {Send task to worker i}
endfor

{Receive-send}
while (WorkToDo)
    rcv(888)                                 {Receive result}

    {Send next task to available worker}
    snd(<worker name>,<available worker>,999)

    display result
endwhile

{Gather remaining results.}
for i := 0 to NumWorkers - 1
    rcv(888)                                 {Receive result}
    terminate(<worker name>,i)               {Terminate worker i}
    display result
endfor

-------------------------------------------------------------------------

{Worker Mandelbrot algorithm.}

while (true)
    rcv(999)                                 {Receive task}}
    result := MandelbrotCalculations(task)   {Compute result}
    snd(<master name>,0,888)                 {Send result to master}
endwhile
```

Figure 5(b): Data Parallel Algorithm Outline

of bytes transmitted, with a constant additive factor representing "startup" overheads. Therefore

$$T_{comm} = \alpha + \beta\, N$$

where α is the startup time, and β, the cost per byte. Returning to the PVM architectural model in Figure 1, it can be seen that PVM provides two alternative mechanisms for message transfer; one, using datagram (connectionless) protocols is routed via the system daemons, while the other, based on stream (connection-oriented) transport, is direct between communicating processes. Although the datagram communication mechanism is a three-stage process, our experiments demonstrated that this mode of communication could be modeled by a single equation as shown above. In order to estimate the coefficients α and β, a number of experiments were conducted under varying network and host load conditions, for different message sizes. The results of these experiments are shown in Table 1. The message "packing" times compiled in column two of the table account for the PVM system overhead associated with composing messages in a machine independent manner in preparation for delivering the data to the communication subsystem, and the other columns report actual message transfer times.

Size (bytes)	Packing	Datagram Transmission	Stream Transmission
1	0.093182	4.97248	1.59715
32	0.088342	4.2199	1.47073
64	0.095959	3.85209	1.54337
512	0.155628	4.42586	1.61310
1024	0.255457	5.24203	1.98495
4096	0.670895	14.61778	10.20227
16384	2.212407	45.43572	20.08871
131072	22.738652	332.53938	184.47232

Table 1: Measured Time (milli-seconds) for PVM Communication

From the experimental results, the coefficients were determined by curve fitting. For the test environment comprised of 12-MIPS Sun Sparcstations interconnected by 10 Mb/s Ethernet, we found that the PVM communication costs[2] are as follows:

$$\alpha_{pack} = 0.0 \; msec$$

$$\beta_{pack} = 0.00017 \; msec/byte$$

$$T_{pack} = \alpha_{pack} + \beta_{pack} N \; msec$$

$$= 0.00017 N \; msec$$

$$\alpha_{DG} = 4.527 \; msec$$

2 We take α_{pack}, β_{pack}, and T_{pack} to represent the variables in the equation for the cost of "packing" messages with PVM.

$$\beta_{DG} = 0.0025 \; msec/byte$$

$$T_{send-DG} = \alpha_{DG} + \beta_{DG} N \; msec$$

$$= 4.527 + 0.0025N \; msec$$

$$\alpha_{stream} = 1.661 \; msec$$

$$\beta_{stream} = 0.0014 \; msec/byte$$

$$T_{send-stream} = \alpha_{stream} + \beta_{stream} N \; msec$$

$$= 1.661 + 0.0014N \; msec$$

$$T_{comm-DG} = T_{pack} + T_{send-DG}$$

$$= 4.527 + 0.00267N \; msec$$

$$T_{comm-stream} = T_{pack} + T_{send-stream}$$

$$= 1.661 + 0.00157N \; msec$$

It is interesting to note that the above equations fit the observed measurements very well, and that few outlying observations were encountered.

2.4. Issues in Application Performance

To reiterate our objectives, in this project we wish to analyze factors that affect performance in heterogeneous networked systems and to construct reasonably accurate models of application behavior. The motivations for this effort are fourfold:

- To gain an understanding into the basic characteristics of concurrent algorithms that execute in network computing systems.

- To predict, based on the model, overall execution times, and to verify the extent to which the model will scale while retaining a close relationship to observed measurements.

- To determine the break-even number of processors (beyond which no speedups are obtainable for a given problem size), and to determine the range of processor values which exhibit greatest efficiency levels.

- To identify bottlenecks and/or the effect of parametric changes, e.g. processor speeds, network throughput, etc.

In this initial attempt, we begin with the traditional performance model adopted in typical distributed memory multiprocessor analyses. Essentially, the time taken for an application is dependent on the computation time, communication time (including synchronization), and load imbalance. The last factor is usually a measure of the equitable distribution of workload among the processing elements. However, in networked systems, where processors are typically not dedicated to the application under consideration, there is another side to load imbalance *viz* the effect of external CPU, memory, or network loads. For example, in a cluster of identical workstations, equal amounts of work may well require different execution times, owing to externally generated computation, swapping, and network activity. Of course, these influences vary dynamically, and it is therefore difficult to incorporate them accurately into any theoretical model. As will be described in Section 4, our proposed approach to addressing this issue is based on a combination of flexible partitioning and scheduling schemes, controlled by instantaneous monitoring of conditions contributing to external effects. In the following analyses, we revisit the three representative application algorithms discussed earlier. Load imbalance due to uneven distribution of workload is not an issue in any of these; in matrix multiply and sorting, workload partitioning is equal, while the dynamic load balancing scheme in the Mandelbrot computations prevent workload imbalance except at the end of application execution.

3. Experimental Results

In order to determine whether models could be constructed that are simple and approximate, but yet pragmatic enough to be of value, experiments were conducted and are reported in this section. Our methodology was to conduct empirical studies, and compare the results from these against predicted application behavior obtained by modeling the same applications. The algorithms outlined in the previous section were implemented under PVM, and timings measured. To form the basis for comparison, the same applications were also executed on an Intel iPSC/860 distributed memory multiprocessor. In this section we report on predicted versus measured results for the application algorithms and compare these findings against similar measurements on the hypercube (a multiprocessor with non-multiprogrammed processing elements and a dedicated interconnection network).

In the experiments that were conducted, we instrumented the applications in order to measure the time spent in various components of the algorithms, and we used the following method for obtaining the measurements. All measurements are elapsed (or wall-clock) times. The time during which a process was blocked awaiting message arrival (i.e. waiting for a subsequent unit of work) is considered *idle* time, and was measured as the elapsed time between the beginning and end of a **receive** operation. Idle time is essentially a means of quantifying the degree of load imbalance exhibited by the algorithm. We consider the time a process spends doing useful work which can be contributed to the final result (e.g. multiplication of sub-blocks in the Pipe-Multiply-Roll algorithm) to be *computation* time. *Communication* time includes the time spent in packing a message as well as transmission to the intended receiver, and was measured by timing the actual PVM primitives as the algorithm executed. *Total* time was measured by a single process that waited for all instances to finish execution. This process was also responsible for gathering local timing results from all instances; computation, communication, and idle times reported are arithmetic means of corresponding individually measured timings.

3.1. Matrix Multiplication

Modeling the Pipe-Multiply-Roll algorithm described earlier, using communication and computation as the only factors, yields the following:

$$T_{comp} = P \left[\left(\frac{N}{P} \right)^3 T_{mult} + \left(\frac{N}{P} \right)^3 T_{add} \right]$$

$$= \frac{N^3}{P^2} \left(T_{mult} + T_{add} \right)$$

$$T_{comm} = P \left\{ P \left[\alpha + 4\beta \left(\frac{N}{P} \right)^2 \right] + \left[\alpha + 4\beta \left(\frac{N}{P} \right)^2 \right] \right\} , \; P > 1$$

$$= P(P+1) \left[\alpha + 4\beta \left(\frac{N}{P} \right)^2 \right]$$

where T_{mult} and T_{add} are the times for a single multiplication and addition respectively[3], T_{comp} is computation time, and T_{comm} is communication time. In the above, $N = \langle \textit{matrix dimension} \rangle$ and $P = \sqrt{\langle \textit{numprocs} \rangle}$. We specifically include N^3 additions (as opposed to the $N^2(N-1)$ additions that are actually required) since the general method of implementation employs this strategy, i.e. $C[i][j]$ is first set to 0 and then intermediate products are repeatedly summed in $C[i][j]$.

The graphs in Figures 6(a) and (b) show the relationship between the predicted and observed measurements for matrix multiplication on PVM, using datagram and stream communication protocols respectively. While the two curves are similar, it may be seen that as the number of processors increases, the observed timings appear to diverge from the predicted ones.

---

3 The times for a single addition/multiplication were measured empirically. In the workstation environment $T_{mult} + T_{add} = 6.741\ \mu sec$, and on the hypercube $T_{mult} + T_{add} = 0.4\ \mu sec$ [11].

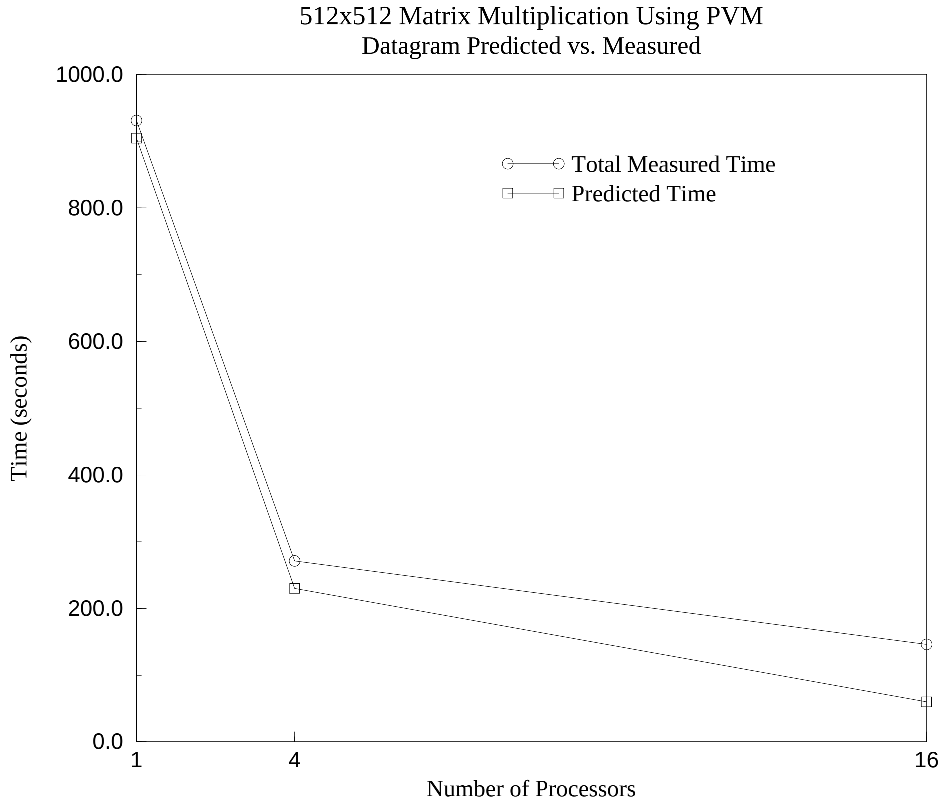

Figure 6(a): Predicted vs. measured time for matrix multiplication using PVM with datagram communication.

In order to better understand the reasons for this behavior, a more detailed analysis of the observed times was performed. The results are shown in the graphs in Figures 7(a) and (b).

The computation times were nearly identical for both the stream and datagram protocols, and they also match the predicted computational load very closely, which implies that the theoretical model is an accurate predictor of CPU activity. Measured communication times were also within reasonable limits of each other, but it is important to note that the PVM message-passing semantics does not exactly mimic a blocking **send**, i.e. a sending process does not wait for an acknowledgement from the destination process before continuing execution. Hence, the measured communication time may be significantly lower than the actual time spent packing, transmitting, and unpacking the various messages; and a portion of idle time will in fact include transmission time. Thus,

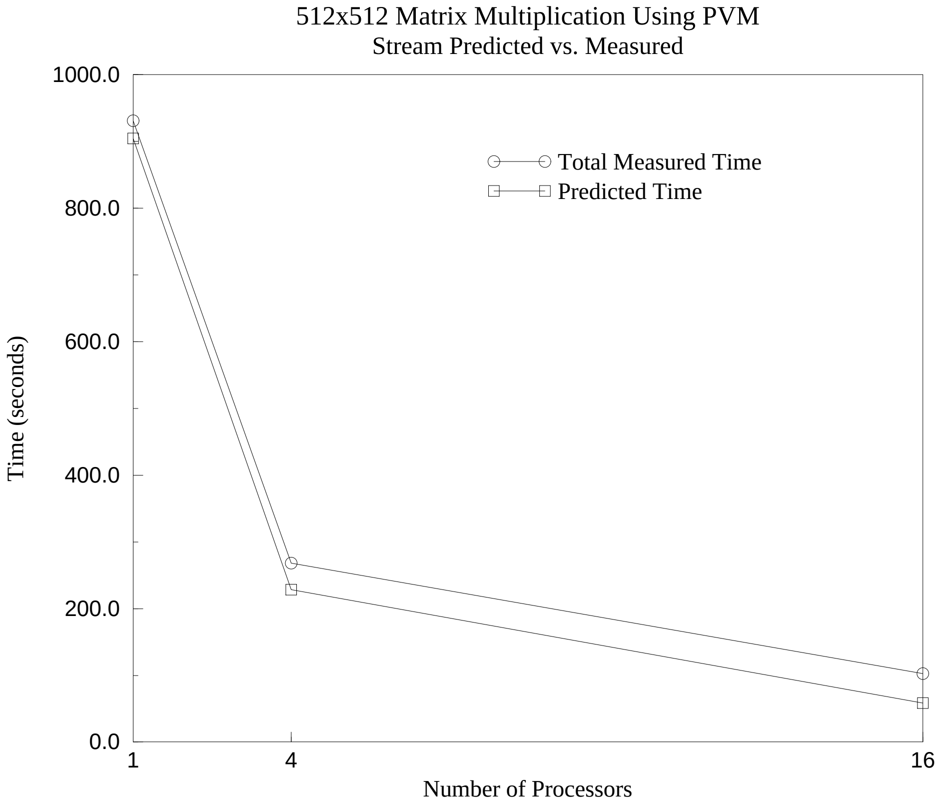

Figure 6(b): Predicted vs. measured time for matrix multiplication using PVM with stream communication.

the observed idle time is the key point of interest here, and it is clear that a large amount of load imbalance was incurred by the algorithm, especially in the case where datagram protocol was utilized. While some idle time is due to propagation delay, the remaining and more significant amount is attributable to the fact that the communication medium was actually a shared channel, forcing messages to be serialized in unspecified order, whereas the model assumes bandwidth-unlimited communication. Therefore, the deviation of the observed measurements from the predicted values is primarily due to contention for the physical communication medium, and clearly a communication pattern which is regular and symmetric (and at approximately equally spaced intervals) is a poor choice for a network cluster computing environment that utilizes a shared communication medium.

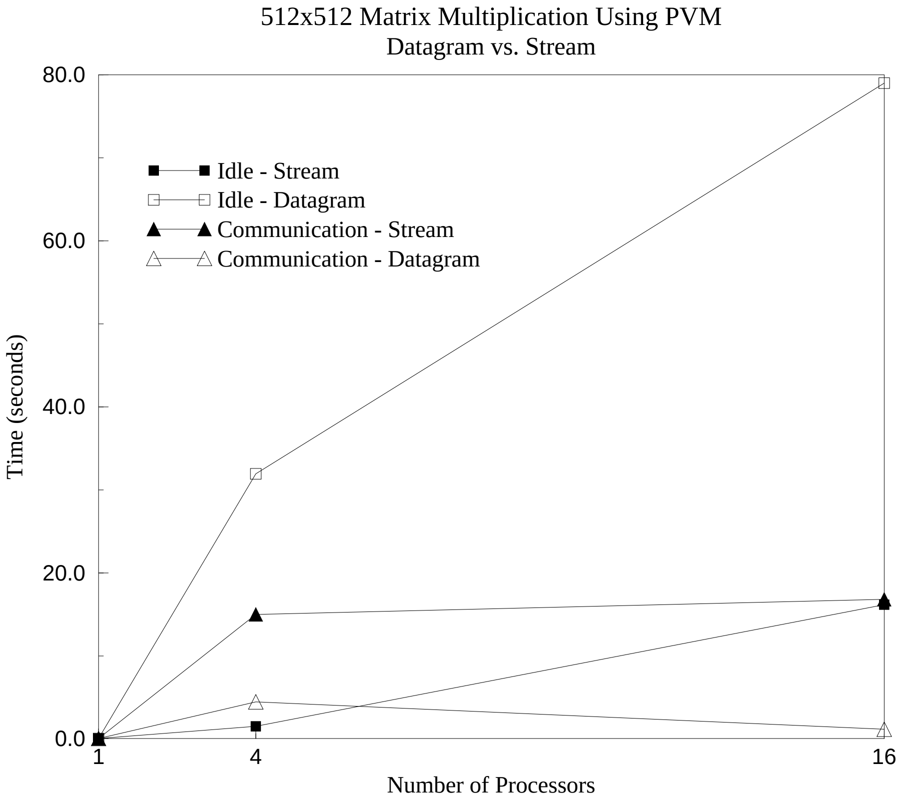

Figure 7(a): Measured idle and communication time for matrix multiplication using PVM.

Since networked machines used as concurrent multicomputers communicate over a shared channel, they share many of the same problems that plague bus-based multiprocessors: scalability, bandwidth requirements, and contention. As machines are added to the network, the bandwidth requirements increase while available bandwidth remains constant, resulting in poor scalability. Also, contention for the channel must be arbitrated by some means, which implies that messages are either serialized (as in MPP's that utilize a global bus) or interleaved - increasing latency in either case. Scalability and bandwidth issues will primarily be addressed by new networks, e.g. fiber optics, which possess a much higher capacity. In order to reduce contention and, in so doing, latency and idle time, it is desirable to utilize algorithms which have irregular communication patterns in which on average at most one processing element is attempting to use the network at any given time. Concurrency can be much more effectively utilized to hide latency in such algorithms with staggered, or even "chaotic", communication patterns. Such algorithms are better suited to network environments than the matrix multiplication

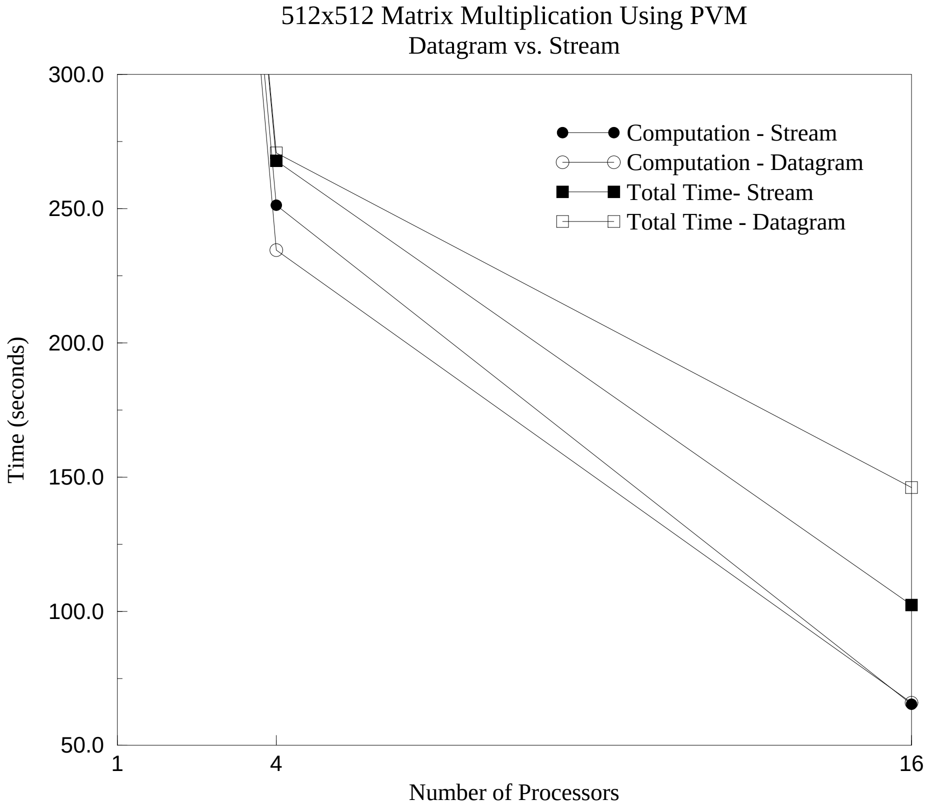

Figure 7(b): Measured computation and total time for matrix multiplication using PVM.

algorithm discussed. While regularity and symmetry may be a desirable property for parallel algorithms on true multiprocessors, the above experience indicates that they can be detrimental in cluster computing where communication cost factors so heavily into performance degradation. In the following section we present other strategies that lead to a better match between observed and predicted results.

Figure 8 shows a comparison between predicted and observed times for matrix multiplication (using the same algorithm) on the Intel iPSC/860 hypercube. Here again we see divergence of the measured values from the predicted total times; however, both computation and communication times appear to be consistent with the expected values. This is surprising, since total time should simply be the sum of computation and communication times, and the reason for this behavior is not immediately apparent. Idle time was not significantly high enough to warrant concern; in fact, it was most likely a product of poor clock resolution. We conjecture that this discrepancy is due to the following

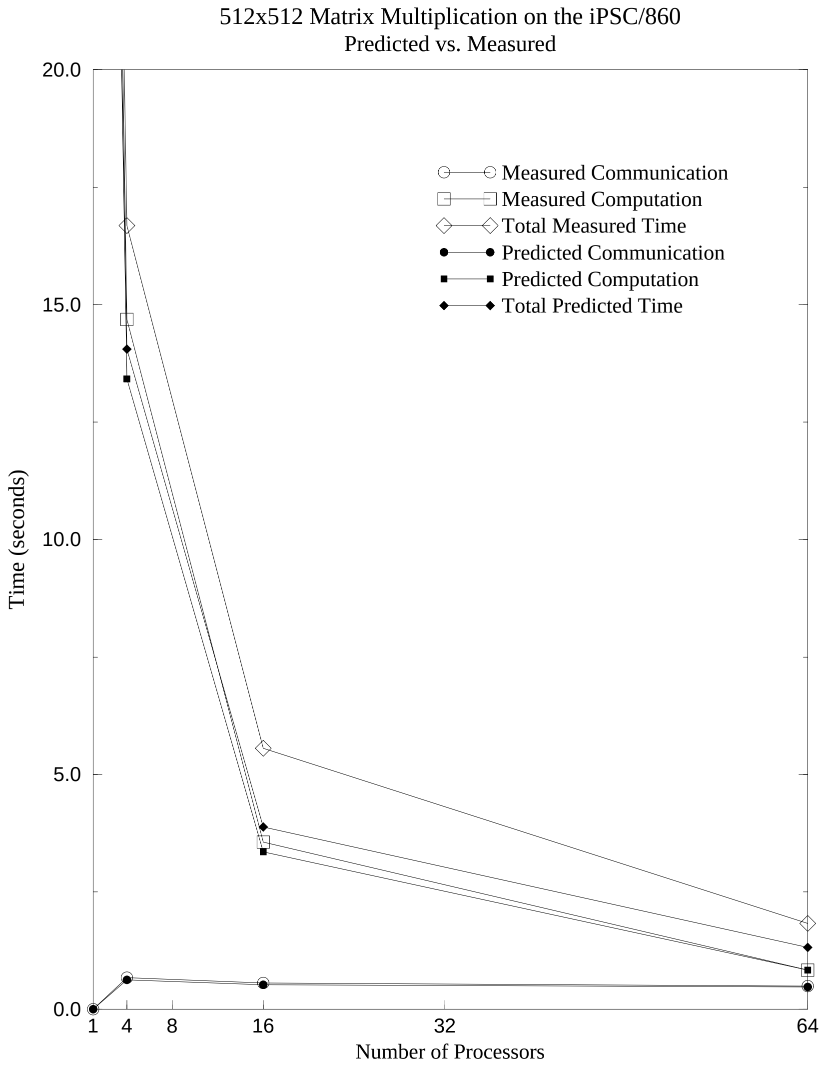

Figure 8: Predicted vs. measured times for matrix multiplication on a hypercube.

factors that are specific to the Intel iPSC/860 hypercube.

- While the **send** primitive on the hypercube is a blocking procedure, the time a process is suspended is equivalent only to the time it takes to deliver the entire message to the physical communication layer within the local node (as was mentioned earlier). Our experiments, however, indicate that the time to pack and transmit a message is roughly equivalent to measuring the time for the **send** to execute, but this is not surprising since the amount of transmission time that is omitted from the measured result is simply the transfer time for the last byte of the message (and perhaps some minimal unpacking) — which is trivial when compared to the total communication time. In fact this lost time is probably smaller than the clock resolution and thus represents no significant difference from the actual cost, but this is only true in the case of single hop transmissions and must necessarily carry larger weight where multi-hop transfers are used.

- The matrix multiplication algorithm that we employed was explicitly designed to operate on a two-dimensional mesh interconnection network, but the hypercube topology is that of a *boolean N-cube*. While a mesh can always be embedded within a hypercube by some appropriate renaming of the nodes, such a translation was not performed. Thus, many messages required multiple hops to reach their destinations, which implies that measured communication time should align with predicted time (assuming a mesh-connected machine) but will in reality be somewhat higher because the multi-hop propagation delay is not included. Despite this inconsistency, the model fits the measured values within reasonable limits and can thus be effectively utilized as a close approximation of run-time processing.

3.2. Sorting

The Split-Sort-Merge algorithm, which was outlined earlier, is a new algorithm which was developed in order to fully exploit the topology of a hypercube. It utilizes the familiar broadcast tree pattern to distribute the workload as quickly as possible (assuming that the list to be sorted initially resides on a single node). Based on what is known about the quicksort algorithm's partitioning scheme, the workload should be fairly evenly distributed. Thus, the algorithm should perform well and exhibit good speedup in the presence of low communication costs, such as that found on the communication pathways of a hypercube. This algorithm can be modeled in a manner similar to that used above, and the following results are obtained, where T_{oper} is the time to perform a single sorting operation[4] (i.e. a test and swap):

$$T_{comp} = \left(\frac{N}{P} \log_2 \frac{N}{P} \right) T_{oper}$$

4 Benchmark studies yielded the following average times for a test and swap operation: 2.54 μ*sec* on the network cluster, and 0.45 μ*sec* on the hypercube.

$$
\begin{aligned}
T_{comm} &= \sum_{i=1}^{\log_2 P} [\alpha + \frac{N}{2^i}(4\beta)] \\
&= \frac{(\log_2 P)(1+\log_2 P)}{2}\alpha + 4\beta N \sum_{i=1}^{\log_2 P} \frac{1}{2^i} \\
&= (\log_2 P + \log_2^2 P)\frac{\alpha}{2} + 4\beta N \left[\frac{P-1}{P} \right]
\end{aligned}
$$

Timing studies were performed with an implementation of this algorithm on the same test beds that were used in the matrix multiplication experiments, and Figure 9 summarizes the results for a cluster environment using datagram and stream protocols. It is clear from the graph that the match between predicted and measured values is quite close, i.e. within 50 ms of each other, but there is still some deviation. In order to understand why the algorithm behaved in this manner, we conducted further experiments, examining the various components of execution as before, and the results for a network cluster and the Intel iPSC/860 hypercube are shown in Figures 10 and 11 respectively.

It is interesting to note in these graphs that in each of the experiments, the observed computation times match the predicted values closely, even though the predictions are based on the assumption that exactly $(N/P)\log_2(N/P)$ operations are performed, when in reality the computations are more loosely defined, i.e. $O[(N/P)log_2(N/P)]$. It is evident, however, that communication cost was the primary factor affecting performance and leads to a divergence from expected results. This phenomenon is partially accounted for by contention for the shared physical medium in the cluster environment; but as we explain, this is not the main cause. Although the algorithm partitions the list in a quicksort fashion, which on average results in dividing it in half on each iteration, in practice we see wide variation from the median over a small sample space but narrow divergence when a large number of partitions are necessary. Thus, it is not surprising that average computation times match so well since they apply to $1/P$ of the list, which is understandably quite large since in general $P \ll N$, and the median will approach the expected value. Communication time, on the other hand, involves only $\log_2 P$ partitions, and thus there will be large differences between expected message sizes and actual sizes. At first this may seem to be a trivial concern since the list is always broken into two parts and the

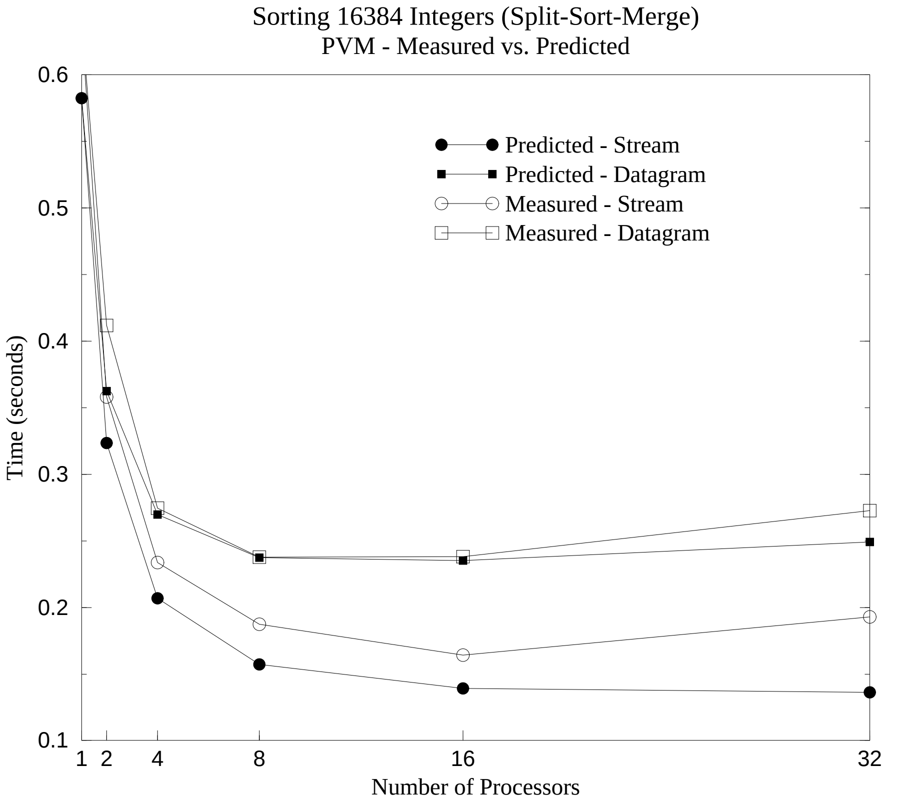

Figure 9: Predicted vs. measured times for sorting using PVM.

total volume of communication is constant. Unfortunately, the algorithm assumes a symmetric communication pattern, but in practice the broadcast tree will become unbalanced in terms of workload and idle time. This imbalance is propagated down through the tree, and results in poor correlation with predicted values. If the tree were significantly deeper, i.e. $P \approx N$, then we could expect to see much closer alignment to expected results as we do in the case of computation time. This is unrealistic, however, both in terms of cost and scalability since the algorithm's performance degrades so rapidly. Thus, this algorithm provides an interesting theoretical solution to the sorting problem on a DMM, but in practice the communication overhead greatly outweighs the benefits of additional parallelism.

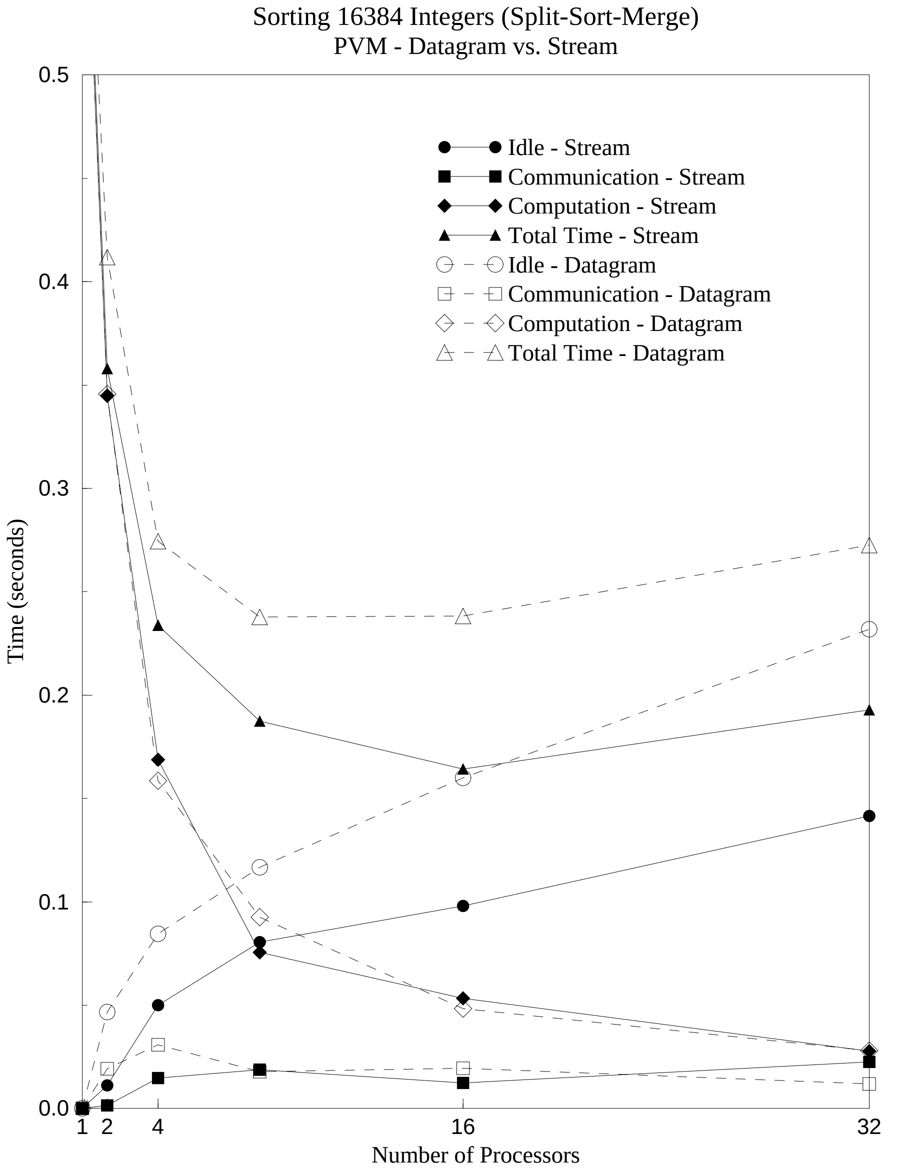

Figure 10: Comparison of PVM datagram and stream protocols for sorting.

3.3. Mandelbrot Computation

Experiments were also conducted to examine the behavior of an algorithm that utilizes the task queue model of parallelism. We chose the computation of the Mandelbrot set for our experiments, since the computation pattern is so well known, and since the work of each processing element is independent of the others. The communication model is trivial, consisting of a pre-determined number of master-slave exchanges. Our

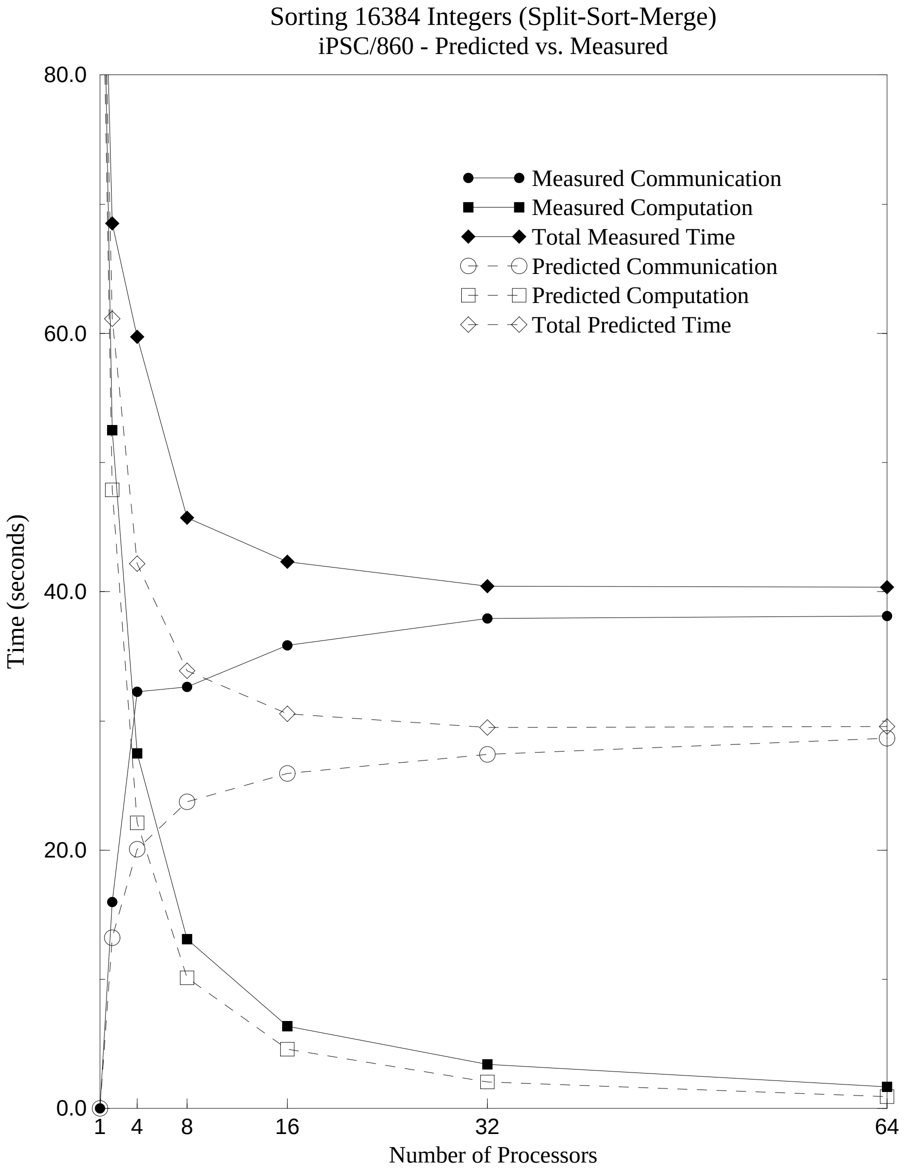

Figure 11: Predicted vs. measured time for sorting on a hypercube.

results are presented in Figure 12, which shows that observed and expected values differ only negligibly, and thus the model approximates actual behavior with a high degree of accuracy.

There is no substantial idle time incurred by any of the processors, but this is unsurprising since dynamic load balancing is inherent in this programming paradigm. An interesting point to note is that predicted times actually exceed measured values for

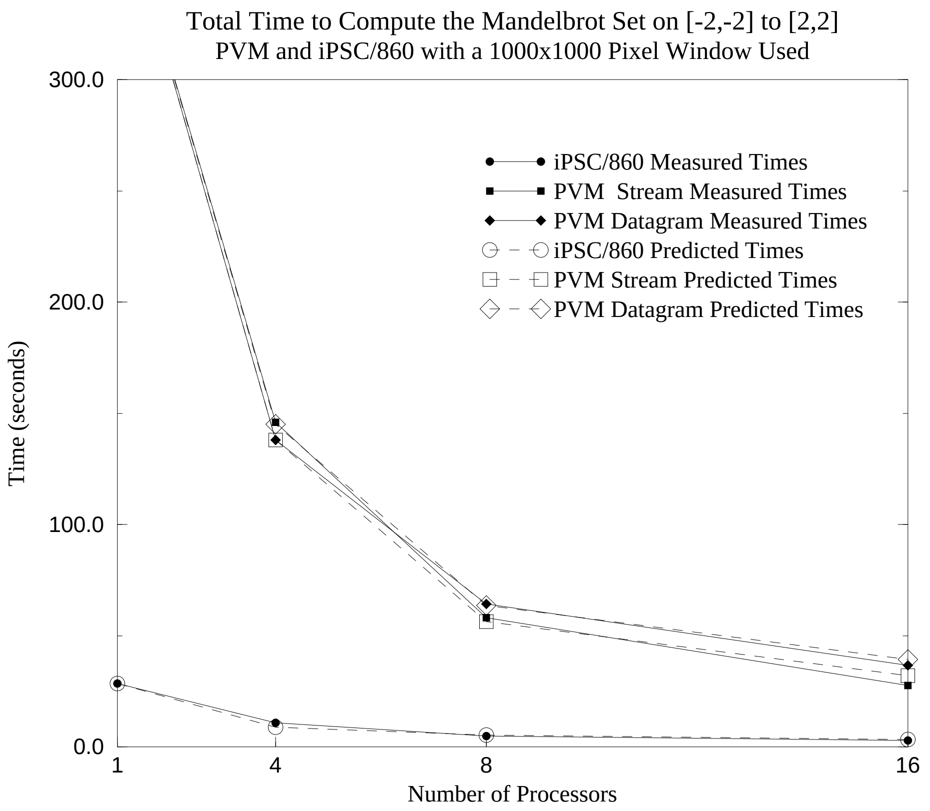

Figure 12: Predicted vs. measured times for Mandelbrot calculations.

large numbers of processors. This can be explained, since the theoretical model assumes that $T_{total} = T_{comp} + T_{comm}$, but in actual practice, computation and communication will be overlapped by an amount directly proportional to the number of processing elements. In addition to providing a close match between predicted and observed results, the Mandelbrot computation experiment also suggests that an algorithm which is based on dynamic load balancing will yield superior results and allow effective modeling of its behavior.

4. Proposed Techniques and Strategies

In the previous section, we described experimental results obtained from execution of example applications, and compared these against the predicted behavior of PVM programs that used a simple and straightforward model. The relatively close conformance between expected and observed values are encouraging. At least for these classes of applications, approximate modeling of program characteristics seems viable and useful. These analyses and experiments lead to the next stage of our work, i.e. experimentation with strategies and techniques that improve the performance of PVM programs, and achieve greater efficiency, given the constraints of operating in a general purpose environment that is subject to varying external influences.

4.1. Efficiency Analysis

Based on the results presented in section 3, it is apparent that the theoretical model provides an adequate framework upon which to analyze the efficiency and expected performance of application algorithms. In addition to modeling overall execution time, other benefits may also be drawn from this methodology. For example, examination of the properties of the formulas for the expected times can lead to a good prediction of the number of processing elements to utilize in order to maximize efficiency. Communication cost is often inversely related to computation time, but generally increases with the number of processors. As a consequence, many algorithms do not scale well; for, as the number of processing elements increases, the cost of communication can become prohibitive and impede any further performance gains. A graph of total time for a scalable algorithm will exhibit a constant, downward slope, but for most algorithms the slope of this curve will begin to increase and flatten out. At this point, the use of additional processors would be wasteful, and would not contribute to increased speed. In order to avoid this situation it would be useful to be able to predict the number of processing elements that should be used, in order to ensure minimum execution time and reasonable resource usage.

Analyzing the slope of the curves associated with the formulas to model the behavior of the algorithms that were presented in the preceding sections, we predicted the level of parallelism that should be employed in order to maximize efficiency. This was accomplished analytically by searching for the point at which the slope of execution time begins to decrease more slowly than some predetermined bound, where this bound represents the tolerance level for inefficiency. By examining the graphs in figures 6, 8, 9, 11, and 12, it becomes clear that as the curve for total execution time begins to flatten, further parallelism would be wasteful; and thus the most efficient use of concurrency lies in this area of the curve. Predictions based on the formulas fell into the "flattening" regions of the curves of measured execution time, and thus these results, as well as our intuition, were strongly supported by the data. Although this is a simplistic model, it is powerful enough to provide us with an easily computed heuristic for determining the degree to which an application will scale. Therefore, such an analysis of the run-time efficiency of these classes of parallel applications can provide an effective means of ensuring high performance while maintaining efficient use of resources.

4.2. Alternative Protocols

From the analysis of PVM communication costs presented in the previous section, it can be observed that both communication mechanisms (datagram and stream based) have several drawbacks. The datagram communication method has the advantage that connection maintenance overheads are avoided and scaling bottlenecks are not encountered. However, since this is a three-stage scheme, considerable overheads are incurred, and communication is often prohibitively expensive. The stream based approach exhibits superior performance, but it too, has several drawbacks. Connection establishment is expensive, and therefore this scheme is unsuitable for one-time, short message exchanges. On some systems, the number of such stream connections that can be used is subject to certain restrictions, a factor that could lead to scaling limitations.

In order to address these drawbacks, we are investigating alternative protocol architectures aimed at providing high performance services for network-based computing. One such effort is the DCL protocol suite; a preliminary description of this protocol may be found in [12]. The basic premise in this work is that distributed systems inherently require a different subset of communication services, which can be provided on local networks by devising a protocol that operates directly above the data link layer. A schematic of such a protocol architecture is shown in Figure 13.

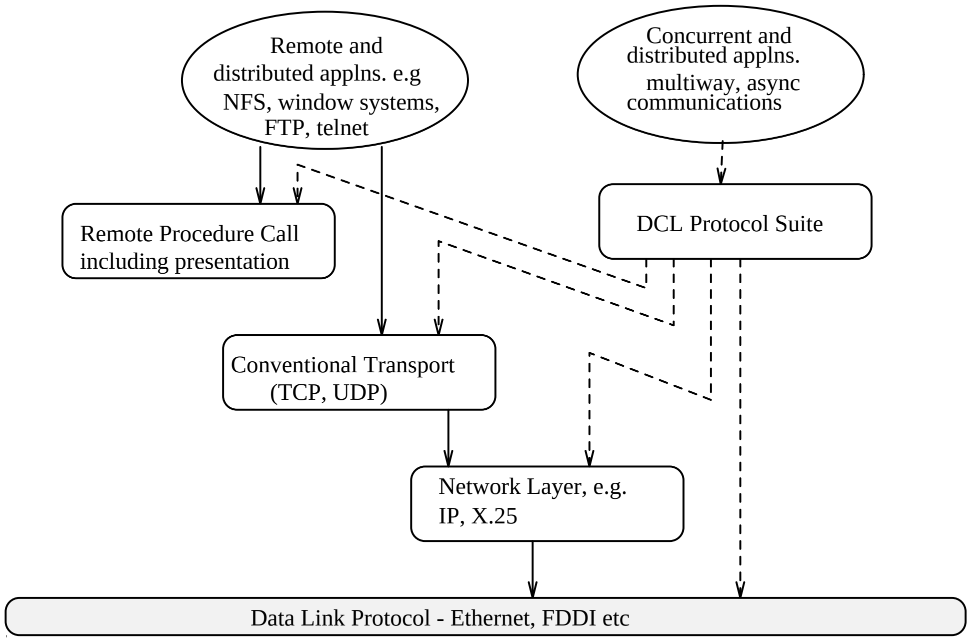

Figure 13: Communication model for DCL protocol suite.

The experimental DCL protocol is designed specifically for concurrent applications on local area networks. The protocol is inclusive in that it subsumes the functions of the transport and network layers, whose interfaces are not directly accessible to applications. In addition to reliable data transport, the DCL protocol contains support for group communications, session and token management, synchronization, and recovery. Further, several frequently used distributed functions like mutual exclusion, consensus, election, and reliable broadcast are available as protocol service primitives.

One of the key features of DCL is its intended operation directly above the data link layer of local networks. This is a significant departure from traditional protocol layering schemes, but is justifiable for the following reasons:

- The layered nature of most protocol suites is usually preserved in their implementation, leading to inefficiencies caused by boundary crossings. While layering is a convenient abstraction and helps reduce complexity, primary network and transport layer functions such as routing and sequencing are almost nonexistent in local network environments. Therefore, incorporating the necessary functions within a session level protocol results in performance benefits with little sacrifice in modularity and function separation.

- Recently, efficient access points to the data link interface have been made available (e.g. the packet filter mechanism, the NIT protocol), thereby permitting higher level protocols (including those implemented as user processes) to effectively use low level communication facilities.
- Operation directly above the data link level enables more effective utilization of, and control over, link level features such as network broadcast and multicast, and also allows the direct use of local network addresses for machine identification.

The experimental implementation of DCL performs up to 40% faster in terms of end-to-end latency and throughput, as compared to the datagram and stream protocols used by PVM. The test implementation of this protocol showed that communication costs could be greatly reduced; estimates for α and β on the same experimental platform were 0.87 ms and 0.00077 ms/byte respectively. If this protocol were utilized by the PVM system, application speedups would improve significantly. Speedup curves for the previously examined applications using PVM with datagram and stream protocol are shown in Figures 14(a) and (b) respectively, and estimated curves based on the above communication costs (using the DCL protocol) are shown in Figure 14(c).

Both the agenda parallel algorithm (Mandelbrot set calculation) and the regular, symmetric matrix multiplication algorithm approach linear speedup, and the tree-based sorting algorithm approaches logarithmic speedup. Such results are encouraging and demonstrate that a network cluster computing environment is capable of behaving in a near-ideal manner. Notwithstanding the above results, these expected gains do not take into account the load imbalance due to external factors. More improvements may be possible if such conditions were monitored, and the algorithm dynamically adapted in response.

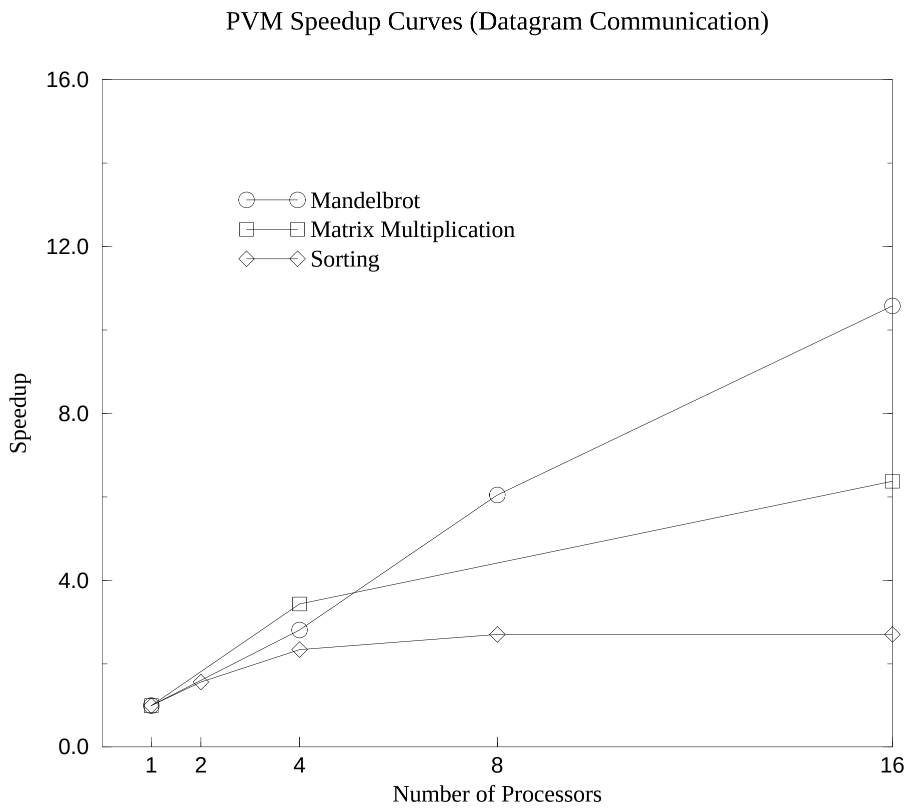

Figure 14(a): Speedup curves for previous applications using PVM datagram protocol.

4.3. Partitioning and Scheduling Schemes

As discussed in Section 3, load imbalance caused by external factors contributes to performance degradation. In order to test the extent to which this factor degrades performance, the matrix multiply application was reorganized in two different ways and executed under the same conditions as earlier. The first method was "hierarchical PMR", a variant of the original pipe-multiply-roll algorithm. In this method, a two-level hierarchy was used, with the block multiplications at the first level themselves being implemented as a PMR algorithm at the second level. The results of this experiment were disappointing, and indeed worse than the regular PMR timings. This degradation was found to be caused by a combination of increased, low-volume communication, and exacerbated effects of load imbalance owing to smaller granularity.

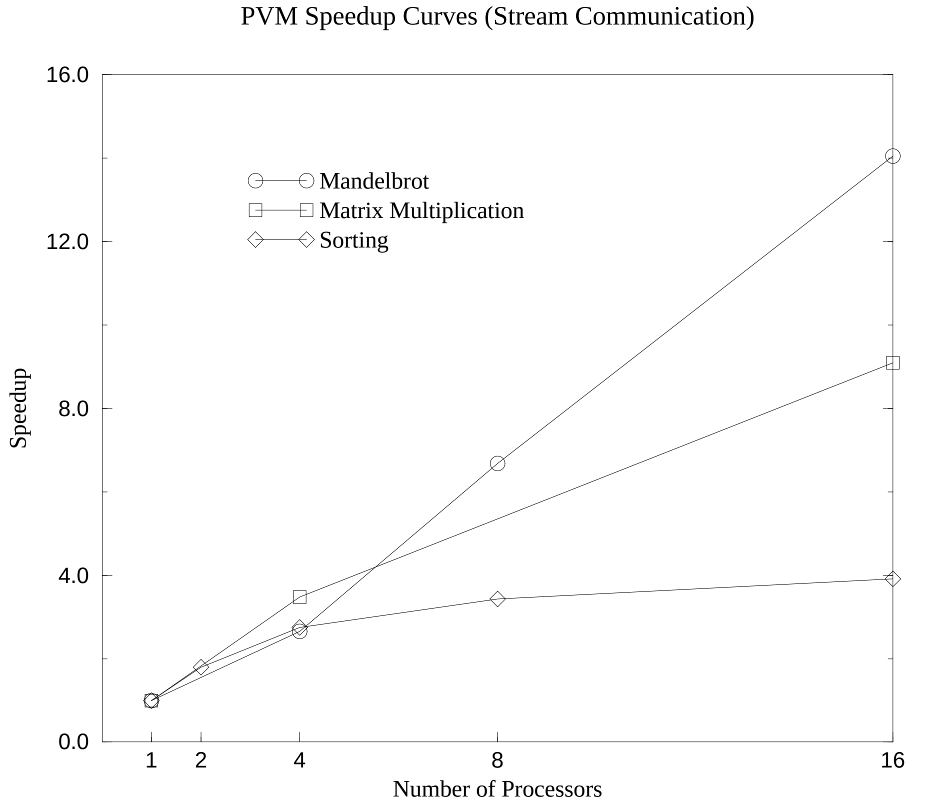

Figure 14(b): Speedup curves for previous applications using PVM stream protocol.

The second approach was to reimplement matrix multiplication as a task queue algorithm, while retaining the block multiplication paradigm and maintaining the same volume of communication. This was done by having a master process maintain a queue of pending work, where each task was the multiplication of a matrix subblock. Using a dynamic load balancing scheme, the master process assigned tasks to processing elements as they became available, collected results, and entered new tasks into the queue. This scheme assigns more work to lightly loaded processing elements, and thereby minimizes overall execution time. The results of this experiment were extremely encouraging, and resulted in an almost identical correspondence with the predicted results. Thus, it is empirically demonstrable that agenda parallelism is possibly the best model for achieving maximal performance gains in a network environment. A comparison of the different strategies and their relationships to expected behavior is shown in the graph in Figure 15.

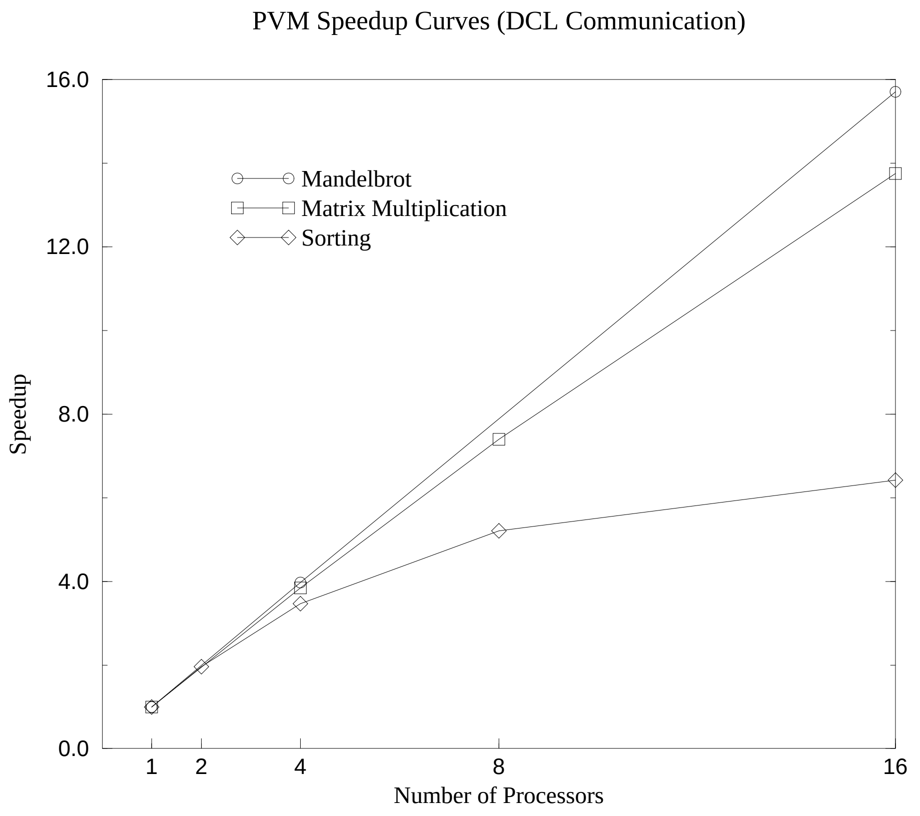

Figure 14(c): Speedup curves for previous applications using DCL protocol.

4.4. Dynamic Resource Monitoring

In the experiment using dynamic load balancing described above, the resource characteristics of individual processing elements are determined only by their early or late completion of application tasks which they are assigned. In order to be able to more precisely determine the usable resource levels on the different processors in a given PVM machine configuration, we have developed a set of extensions to the PVM system. Essentially, these extensions permit instantaneous measurements of various system parameters, including CPU load average, available and occupied memory, interrupt levels, and I/O activity, in addition to one-time measurements of message packing and transmission times as well as raw processor speeds for primitive arithmetic operations. We are currently attempting to formulate a model using these parameters which, when combined with a similar parameterization of application characteristics, will identify the best suited location for a given task or subtask. We intend that this scheme operate at

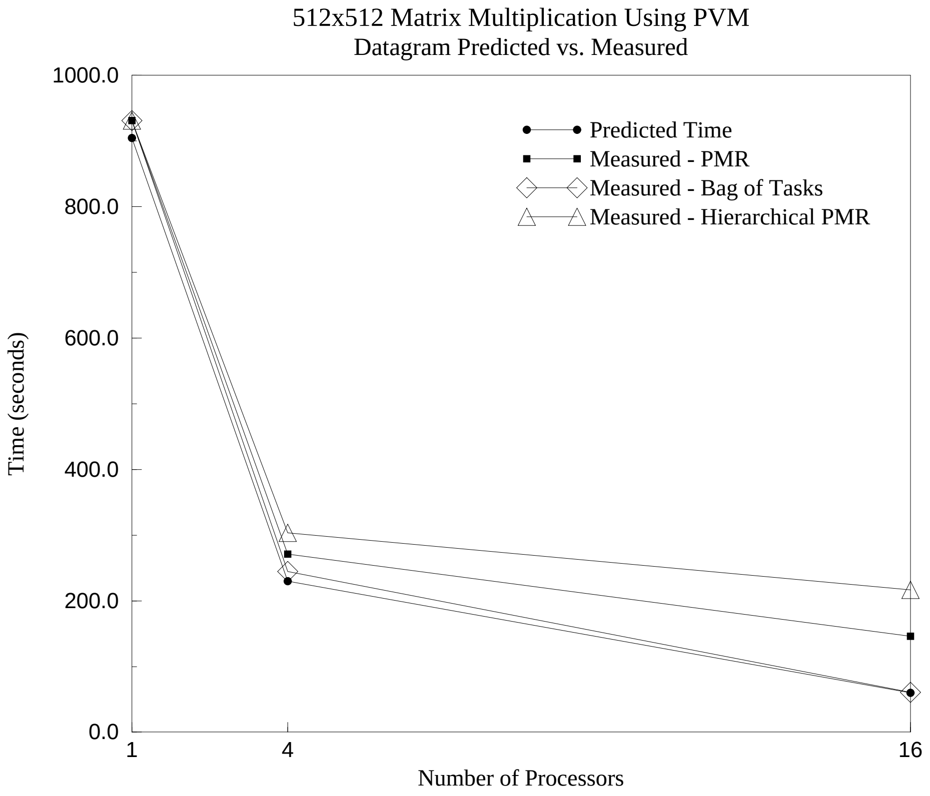

Figure 15: Comparison of differing matrix multiplication strategies.

different levels of granularity — e.g. the process level for initial placement, the data sub-block level when the agenda parallelism model is used, and perhaps even a finer level as explained in the next subsection.

4.5. Domain Specific Concurrent Libraries

One promising approach that we are investigating is the concept of domain-specific libraries that could be integrated into the PVM framework. One motivation for this approach is derived from the fact that many concurrent applications are composed of several distinct algorithms, each typically being driven by the results of the previous one. These algorithms themselves are well-defined computations such as matrix factorization, solution of PDE’s, or Fourier transforms — algorithms that are well understood and for which different parallel solutions have been proposed. By providing predefined, concurrent versions of these basic modules, each implemented in multiple ways, an application's user need only invoke the appropriate module from within a sequential program — the PVM and library subsystems could execute these modules concurrently, utilizing the best algorithm available, depending on user supplied flags, and heuristics based on the problem size, the nature of the network, etc. This approach also permits partitioning and scheduling schemes that are most appropriate, given the instantaneous resource usage levels, which will be readily available within such a framework. A preliminary architectural design of such a model is shown in Figure 16.

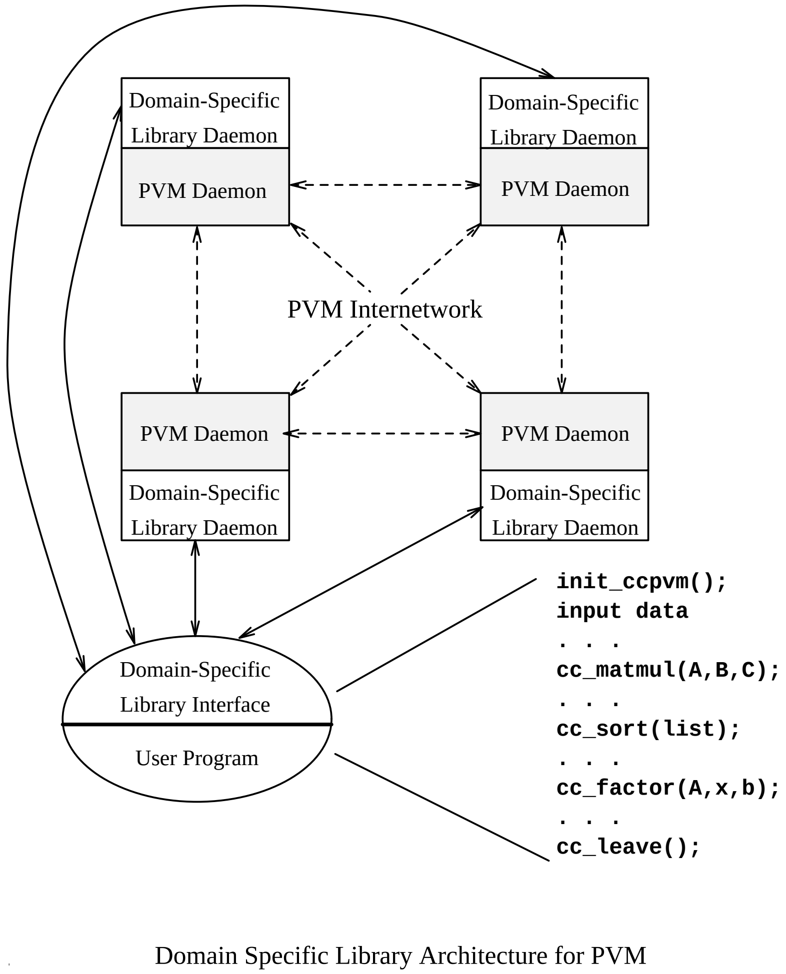

Figure 16: Model of domain specific library structure under PVM.

5. Discussion

The objectives of this work were to investigate the issues in modeling concurrent applications executing in heterogeneous networked environments. Owing to the multitude of factors to which such environments are subject, a complete and formal analysis is thought to be difficult and possibly intractable. Our approach has been pragmatic and heuristic in nature, and our initial experiences are promising. For the representative applications that we studied, simple models based on communication and computation have proven to be adequate and informative. These models may be used to predict application behavior as well as to determine the appropriate number of processing elements that will result in near optimal efficiency. Further, based on our results, we have proposed several strategies that may be employed in order to increase the effectiveness of network computing. Of course, these findings are only preliminary, and it may not be possible to generalize them for a wider range of applications or indeed, more general networked environments. However, certain general recommendations can be made with regard to network based systems:

- Network limitations — while network capacities in cluster environments are usually much lower than in multiprocessors, it *is* possible to utilize this capacity at near-theoretical maximum rates (as our raw communications tests indicate). Since communication startup times are comparatively high, best network utilization is achieved when messages are large and less frequent. The above is true on both present-day local and wide-area networks as well as on imminent high-speed fiber networks. A (perhaps self-evident) performance pointer is therefore to utilize large messages, and direct task-to-task communication, wherever possible.

- Contention — Rather than capacity *per se*, our findings indicate that it is the (commonly) shared use of network media that poses significant bottlenecks in cluster computing. This competition for communications capacity may be categorized as (a) traffic due to external loads; (b) paging activity caused directly by concurrent applications in diskless environments; and (c) messages used by concurrent applications for communication and synchronization. The first factor is generally not significant, unless local network segments contain a large number of heavily used machines or multiple concurrent applications are being executed simultaneously. Paging activity can contribute significantly to network contention, and should be addressed if at all possible. The factor that is directly controllable is contention by application messages, and this effect can be alleviated by staggered communication patterns or even a token-based mechanism that prevents simultaneous message transmission. While the last suggestion might seem archaic, we have found that overall performance actually improves significantly when such a mechanism is used. In general however, it is clear that regular and symmetric algorithms are not well suited for cluster environments when network capacity is relatively small.

- Load imbalance — Almost by definition, heterogeneous cluster computing environments are prone to severe load imbalances, owing to disparities in machine capabilities as well as to external loads. Our experiments indicate that even in reasonably well controlled situations (i.e. small, constant external loads on identical machines), load imbalance emerges as a primary cause of lowered overall performance. This is due to the intermittent and irregular nature of many system activities including paging, routing daemons, etc, and cannot generally be predicted well enough to account for in a partitioning or scheduling strategy. As our experiences show, dynamic load balancing, combined with the *agenda parallel* model is very effective in combating load imbalance. Another approach which appears promising is a software architecture as shown in Figure 15, where workload allocation is performed at smaller granularity, and relative to current resource conditions on individual machines.

- Protocol support — Our preliminary experiments with alternative transport protocols used to support concurrent cluster computing appear to indicate that this aspect is critical, and should be investigated further. Currently, cluster systems utilize protocols that are *pairwise* in nature — while these perform well for one-to-one communication, they are inherently poorly suited for the *many-to-many* mode of communication that typically characterizes concurrent applications. We believe that specialized, lightweight, multi-party protocols will greatly enhance the performance of cluster systems, and are continuing to develop the DCL suite further.

To summarize, this paper has described initial work in the performance measurement and modeling of cluster computing environments. We have attempted to identify different classes of parameters that affect performance, and have investigated possible approaches to address each. However, this effort is only a beginning; substantial further research is required to acquire a deeper understanding of the issues, to construct a comprehensive and accurate model, and to devise strategies to achieve optimal performance. In future work in this direction, we intend to analyze a more comprehensive set of application categories, machine and network types, distributed primitives and protocols, and also evolve methods for precisely identifying external processor and network load factors, and integrate them into a generalized model for network concurrent computing.

6. References

[1] G. C. Fox, "Parallel Computing Comes of Age: Supercomputer Level Parallel Computations at Caltech", *Concurrency: Practice and Experience*, Vol. 1, No. 1, pp. 63-104, September 1989.

[2] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing", *Concurrency: Practice and Experience*, Vol. 2, No. 4, pp. 315-339, December 1990.

[3] S. Ahuja, N. Carriero, D. Gelernter, "Linda and Friends", *IEEE Computer*, Vol. 19, No. 8, August 1986.

[4] J. Flower, A. Kolawa, "The Express Programming Environment", *Parasoft Corporation Report*, July 1990.

[5] G. A. Geist and V. S. Sunderam, "Network Based Concurrent Computing on the PVM System", to appear in *Concurrency: Practice and Experience*, 1992.

[6] V. S. Sunderam, "Heterogeneous Environments for Network Concurrent Computing", to appear in *Journal of Future Generation Computer Systems*, 1992.

[7] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. S. Sunderam, "Graphical Development Tools for Network-Based Concurrent Supercomputing", *Proc. of ACM Supercomputing 1991*, pp. 435-444, November 1991.

[8] G. C. Fox, *et. al.*, "Solving Problems on Concurrent Processors", Vol I, Prentice Hall, Englewood Cliffs, 1988.

[9] M. J. Quinn, "Designing Efficient Algorithms for Parallel Computers", McGraw Hill, New York, 1987.

[10] N. Carriero, D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed", *ACM Computing Surveys*, Vol. 21, No. 3, September 1989.

[11] L. Bomans and D. Roose, "Benchmarking the iPSC/2 Hypercube Multiprocessor", *Concurrency: Practice and Experience*, Vol. 1, No. 1, pp. 3-18, September 1989.

[12] V. S. Sunderam, "An Inclusive Session Level Protocol for Distributed Applications", *Proc. ACM Sigcomm Symposium on Communication Architectures and Protocols*, pp. 307-316, September 1990.

[13] M. J. Quinn, "Parallel Sorting Algorithms for Tightly Coupled Multiprocessors", *Parallel Computing*, 6, pp. 349-357, 1988.

[14] D. J. Evans, "A Parallel Sorting-Merging Algorithm for Tightly Coupled Multiprocessors", *Parallel Computing*, 14, pp. 111-121, 1990.

[15] D. Y. Yeh and D. Y. Shyong, "An Efficient Sorting Algorithm on a Hypercube Multiprocessor", *Proceedings of the Third Annual Parallel Processing Symposium*, pp. 337-343, March 1989.

[16] X. Guan and M. A. Langston, "Time-Space Optimal Parallel Merging and Sorting", *IEEE Transactions on Computers*, Vol. 40, No. 5, pp. 596-602, May 1991.