Distributed Linear Algebra on Networks of Workstations

Thesis by
Peter Carlin
Advisor: K. Mani Chandy

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Computer Science Department
Pasadena, California 91125
July 17, 1994

Caltech CS-TR-94-14

Abstract

This thesis describes the development of a portion of a distributed linear algebra library for use on networks of workstations. The library was designed with special consideration towards three characteristics of networks of workstations: small numbers of processes, availability of multithreading, and high communication latency. Two aspects of the library are highlighted. First, modifications to message passing primitives to permit their use in a multithreaded environment. Second, modifications to basic linear algebra algorithms to improve their performance on networks of workstations. A model of distributed linear algebra on networks of workstations is developed, and used to predict the performance of the modified algorithms. These predictions are compared to experimental results on several networks of workstations.

Acknowledgments

I am indebted primarily to K. Mani Chandy, Carl Kesselman and Eric Van de Velde for guidance, support and inspiration.

The distributed linear algebra library described in this thesis was jointly developed with Tom Zavisca. We were able to work so well together because of his perseverance and sense of humor.

I would like to thank all those who have contributed to the development of CC++. This includes the entire Compositional Systems group at Caltech, as well as many people at Argonne. In particular, I would like to thank John Garnett, Tal Lancaster, James Patton and Mei-Hui Su for their assistance in using the current implementations of CC++.

I would like to thank Rajit Manohar, Paul Sivilotti, and John Thornley for discussion, suggestions and proofreading.

This research was supported in part by NSF CCR-9120008. I wish to thank Microsoft, Green Hills, the Barry Goldwater Foundation and Caltech for supporting my education.

Contents

List of Figures

Chapter 1

Introduction

1.1 Linear Algebra and Networks of Workstations

Linear algebra computations form an important part of computing. They are traditionally performed using expensive machines like vector supercomputers or multicomputers. For computations which require the raw performance available only on these machines, there is no alternative.

Many linear algebra computations do not require such expensive resources. Networks of workstations (NsW) are wide-spread and relatively inexpensive. Perhaps they can be utilized to perform such less demanding computations at much lower cost.

Previous work by Bjorstad, Coughran and Grosse [1] is encouraging. They choose a network of HP workstations connected by FDDI over a Cray Y-MP for modeling semiconductor devices by domain decomposition.

To further investigate the applicability of NsW for doing linear algebra, I have, in conjunction with Tom Zavisca, developed a distributed linear algebra library (named DLA) for NsW. Three important characteristics of NsW considered in designing DLA were:

Communication: Communication is slow relative to computation, both in latency and throughput. Furthermore, shared channel communication protocols like Ethernet and FDDI predominate. This degrades communication performance even further, as simultaneous communications are serialized.

Scale: Networks of workstations can be combined into computations involving at most tens of processors.

Multithreading: The operating system support for multithreading is well developed. This creates the potential for hiding communication by performing computation in parallel with the communication.

This thesis highlights two aspects of DLA developed on account of the above characteristics:

- Modifications to message passing primitives to allow multithreaded processes to use them.
- Modifications to standard linear algebra algorithms to improve performance on NsW.

1.2 Coping with Workstation Diversity

Workstations are available in a diverse combination of hardware, operating systems, and interconnection networks. In designing DLA, six steps were taken to reduce the costs of coping with this diversity:

CC++: The library is written in Compositional C++ (CC++), a parallel programming language developed at Caltech [2][4]. CC++ defines semantics for the creation and control of both single and multiple address space concurrency.

CC++ provides abstraction and portability across the architectures on which it is implemented. The cost is performance degradation over direct implementation in the underlying communication and thread libraries. A detailed knowledge of CC++ is not needed to read this thesis. Where used, the syntax is explained.

Process Structure: Workstations can be programmed in a panoply of styles and virtual topologies. The N dimensional process mesh topology and SPMD programming style common to most linear algebra algorithms is selected.

Architecture Independence: Rather than develop algorithms for a specific network of workstations (NW) and over-optimize for one particular scenario, the algorithms were developed based on a small set of performance parameters. These parameters represent properties of the operating system and hardware important to the algorithms. A suite of CC++ programs was written to extract these from a given NW.

Heterogeneity: Although a NW can be composed of multiple types of workstations, here only NsW composed of identical processors are considered.

Uniprocessor Workstations: Multiprocessor workstations are becoming widely available, but here attention is restricted to uniprocessor workstations.

Thread Scheduling: Preemptive scheduling is useful for real-time applications; however it carries overhead and is unnecessary for linear algebra computations. Here attention is restricted to round-robin scheduling with thread switches only on suspension.

1.3 Organization

The remainder of this thesis is organized as follows: Chapter 2 presents the performance parameters used in developing the new algorithms. Chapter 3 presents the modifications made to the basic message passing system to allow for multithreaded processes. Chapter 4, the focus, details modifications to basic linear algebra algorithms to increase their performance on NsW. Their performance relative to the standard algorithms is modeled for specific NsW using the parameters extracted in Chapter 2. The algorithms are implemented in DLA and their relative performance measured and compared to the theoretical predictions.

Chapter 2

Performance Parameters

This chapter presents the parameters used to model linear algebra algorithm performance on NsW. These parameters represent those aspects of the workstations, operating systems and interconnection networks important to linear algebra algorithms.

Time Arithmetic Operation - τ_a While the underlying linear algebra computation, matrix vector product for instance, remains constant, algorithms might differ in the amount of copying of received results, zeroing of temporaries and other attendant work. Therefore, a measure of the cost of an arithmetic operation is needed.

τ_a is the cost of a double precision arithmetic operation. All arithmetic operations are considered identical. This parameter takes into account the pointer arithmetic and memory dereferencing associated with a floating point operation. All other parameters will be expressed in units of τ_a.

Communication, Asynchronous - $\tau_{ca}(L, C)$ The most important difference between algorithms will be in their communication structures. Communication can be synchronous or asynchronous. In a synchronous communication, neither the send nor the corresponding receive complete until both have been started. In an asynchronous communication, the receive can not complete until after the send has been initiated, but the send may complete before the corresponding receive is initiated. Here only asynchronous communication is considered. The time needed for communication is influenced by the length of the message and the network load. The network load is composed of a background load, due to machines not taking part in the computation, and load caused by the computation. Here the background load is assumed constant during the computations being compared.

$\tau_{ca}(L, C)$ is then the time needed to asynchronously send a message as a function of L, the length of the message, and C, the number of such messages being sent simultaneously across the whole computation. The relationship between C and τ_{ca} varies considerably across systems, but in all cases τ_{ca} increases monotonically with C. τ_{ca} also increases monotonically with L. τ_{ca} is decomposed into $\tau_{cab} + \tau_{caf}$.

Communication, Asynchronous Begin - $\tau_{cab}(L, C)$ A potentially useful consequence of multithreading is the ability to hide communication latency. With the appropriate hardware and software, some workstations can complete a communication and perform computation in parallel. Usually the outgoing message must be copied out of its original location in memory before control is returned to the processor for more computation. If this were not done, modification of that location during the computation might nondeterministically change the message.

$\tau_{cab}(L, C)$ represents the time necessary to begin a communication. After τ_{cab}, the processor is available for computation and the memory sent can be altered, but the message has not yet reached the receiving process.

Communication, Asynchronous Finish - $\tau_{caf}(L, C)$ is the time needed to finish an asynchronous communication after $\tau_{cab}(L, C)$ is complete, including the time required for processing the message in the receiving process.

Overhead from Multithreading - $\tau_{par}(N)$,$\tau_{parfor}(N)$ Multithreading means multiple threads of control are created. Although we have constrained ourselves to uniprocessor workstations, meaning only one thread can be executing at a time, multithreading is still useful. One thread may be suspended pending completion of a communication action, leaving another able to perform computation.

The CC++ constructs for creation of multiple threads are `par` and `parfor`. τ_{par} and τ_{parfor} represent the overhead of creating and managing multiple threads as a function of N, the number of threads created.

Constraining Thread Interleaving - τ_{sync}, τ_{atomic} The possible interleaving of created threads must often be constrained to preserve determinacy in the result of the computation. The mechanisms in CC++ through which this is done are `sync` and `atomic`. A `sync` is a single-assignment variable, used for synchronization between threads on the same process, while `atomic` enforces mutual exclusion in access to C++ objects.

τ_{sync} and τ_{atomic} represent the overhead associated with performing these synchronizations.

2.1 Measured Parameters

A suite of CC++ programs was developed to measure these parameters. This suite is described in detail in [5]. The table summarizes the parameters on the following NsW, where all parameters are normalized in units of τ_a for that particular NsW:

Setup I SPARCstation iPCs running SUNOS 4.1.2, communicating via an Ethernet

Setup II RS6000s running AIX 3.2, communicating via an Ethernet

Parameter	Setup I	Setup II
$\tau_{ca}(L, C)$	24000+4000*C+(200+20*C)*L	90900*C + 91*C*L
$\tau_{cab}(L, C)$	9000+85*L	6360 + 12*L
$\tau_{caf}(L, C)$	15000+4000*C+(115+20*C)*L	84540*C + (91*C-12)*L
$\tau_{par}(N)$	2540+1060*(N-1)	3360+1450*(N-1)
$\tau_{parfor}(N)$	2720+1390*N	3840+1690*N
τ_{sync}	705	380
τ_{atomic}	140	210

Chapter 3

Multithreaded Message Passing

This chapter describes the communication structure designed for DLA to allow communication between multithreaded SPMD processes on NsW. A DLA computation consists of a user-specified process duplicated across a run-time created N-dimensional virtual process mesh. This mesh is mapped to a user-specified network of workstations.

Each process is a CC++ procedure, in which multiple threads may be created. This poses the challenge of creating communication primitives which can be accessed by multiple threads without introducing non-determinism. A multiple-reader/multiple-writer (MRMW) channel object was created to solve this problem. To improve communication performance, postable receives were implemented as well.

3.1 MRMW Channel with Postable Receives

Single-reader/single-writer asynchronous channels in CC++ have been described in [7]. Our MRMW channel adds message tags to allow a receiver to select from a particular sender, and appropriate use of atomicity to allow multiple receivers to use the channel 'simultaneously'.

The receiving end of each channel belongs to a single process. It is a multiple-receiver channel because multiple threads may be reading from the receiving end. Many processes, and many threads inside a process can have access to a sending end of a channel; messages are merged in a non-deterministic but fair manner.

Each message is tagged with the process from which it came. Thus, a receiver may request a message from a particular process, but not from a particular thread inside that process. Similarly, all send actions occur to the receiving processes, but not to a particular thread inside that process.

There are three types of receive actions: `Receive`, `PReceive` and `Post_Receive`. All receive actions specify a process from which the message is desired. The first message in the queue from that process, or the next message to arrive if none is currently present, is returned. All messages are removed from the queue after a receive action has taken place, and memory management of the messages themselves is the responsibility of the channel user. The receive actions differ in how copying the incoming message is handled, and whether the receive is blocking or not.

A `Receive` action returns when the requested message has arrived, thus blocking if no message is present. It returns a pointer to the message, which the receiver is responsible for deleting.

A `PReceive` action is also blocking, but specifies a location, `msg_dest`, to which the message should be copied. If the `PReceive` occurs before the message is queued, the incoming message will be placed directly in `msg_dest`, without an intermediate copy. If the `PReceive` occurs after the message has been queued, the incoming message is copied to that location.

A `Post_Receive` has the same behavior with respect to copying as `PReceive`, but is non-blocking. `Post_Receive` returns a pointer to a sync integer which will be defined after the message has been copied to the specified location.

It is important to note that the intermediate copies being discussed are from the memory space the CC++ program sees. A high-level language programmer has no control over copies created below their control. Minimizing the number of copies on send and receive sides is the best way to limit message latency. Imposing additional latency at the CC++ level is not acceptable if avoidable, as is the case here.

The class interface for the DLA communication object is shown below:

```
global class CommProcess {
public:
  Channel* Incoming_channels;

  void ASend(int direction, int destination, Message& input);

  Message* Receive(in direction, int source);
  void PReceive(int direction, int source, Message* msg_dest);
  sync int* Post_Receive(int direction, int source, Message* msg_dest);

  CommProcess(Configuration config);
  ~CommProcess();
};
```

Each process contains a channel from every other process whose location in the process grid differs in only one index, i.e, its Cantor space neighbors. To reduce the overhead associated with the channels, all the channels from processes in the same dimension of the process grid are grouped together into one `Channel` object. Thus, all communication functions take additional `direction` arguments to determine which channel should be used.

An individual `Channel` object is shown in Figure 3.1. It stores the incoming messages in two lists: a list of unclaimed messages which have arrived (*msg_list*) and a list of pending requests for which messages have not arrived (*wait_list*). `sync` pointers are used to implement blocking receives: requests for a message that have not arrived will suspend on a `sync` variable until that message arrives. `atomic` functions are used to prevent multiple receivers from traversing the lists simultaneously.

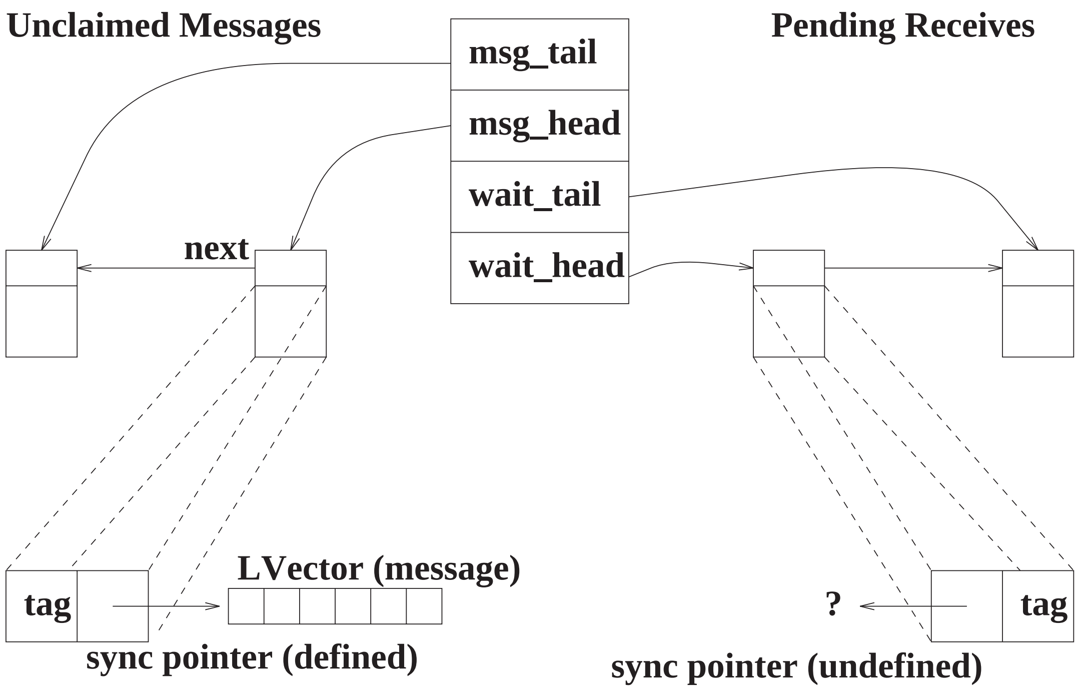

Figure 3.1: Multiple Reader, Multiple Writer Channel

Chapter 4

Algorithms

4.1 Introduction

This chapter presents modifications which improve the performance of three basic linear algebra algorithms on NsW. The modifications cope with the high communication costs and take advantage of multithreading and the small number of processors present on NsW.

Algorithms for matrix multiplication, dense matrix vector product and sparse diagonal matrix vector product on NsW are developed. Execution time and memory use are used to judge performance. For each algorithm, the parameters measured in Chapter 2 are used to theoretically compare the developed and standard algorithms for specific NsW. A comparison is made of these theoretical predictions to the measured performance of the algorithms as implemented in DLA on those NsW.

The algorithms discussed are fundamental to linear algebra. Because the actual computation is not being modified, only incremental increases in performance over the standard algorithms can be expected. The small increases in performance achieved will be multiplied by the many times these routines are used.

Some notes about the presentation of the algorithms and the notation used to describe them are necessary:

Timings: All theoretical timings are in units of τ_a, the time required for execution of one double precision arithmetic operation.

Performance Graphs: The modified and basic algorithms are primarily compared by graphing the difference in execution time as a fraction of the total execution time. This both eliminates constant overheads not modeled in the theoretical analysis of the algorithms, and highlights the important aspects of the comparison.

A drawback is that it does not show how performance for either algorithm compares to that on a single processor. Therefore, supplemental graphs comparing distributed performance to sequential performance are included. This serves to demonstrate that modification affects domains which both need improvement and are of sufficient size to be computationally useful.

Another drawback is that examining the difference in execution times amplifies noise in experimental data, and potentially obscures offsetting errors in the models.

Program Duplication: All program fragments are implicitly duplicated across all processes in the process mesh. By convention, P,Q refer to the number of process rows and columns respectively, while p and q refer to the local process row and column number.

Program Notation: Program fragments are given in CC++, with the added data types, operations and communication primitives defined by DLA and enumerated below.

Data Types: The following DLA data types are used:

- **Vector(Size,Orientation)** A vector of size *Size* distributed across *Orientation* (*ROW* or *COL*). Although the same declaration is made in each process, the local variable contains only the local part of the vector, as defined by the data distribution. The local size will then be $Size/P$ or $Size/Q$ depending on *Orientation*.
- **Matrix(SizeP,SizeQ)** A matrix with global rows *SizeP* and global columns *SizeQ*. Like *Vector*, only the local part of the matrix (size $SizeP/P \times SizeQ/Q$) is stored in each process.
- **LVector(Size)** A local vector with *Size* entries.

Operations: Local operations on DLA data types are expressed in shorthand. For instance, if A is a matrix, and X,Y are vectors, `Y+=A*X` performs the matrix vector product on local portions of A and X and adds the result to the local portion of vector Y.

Communication Primitives: The following DLA communication primitives are used:

- **ASend(dir,pid,lv v or m)** Asynchronous send. Send local vector *lv*, or local portion of vector *v*, or matrix *m* to process *pid* in direction *dir*. *dir* can be *ROW* or *COL*.
- **lv or v or m=Receive(dir,pid)** Blocking receive. Store the first unclaimed message, or next incoming (if none currently present) message from process *pid* in direction *dir* in local vector *lv*, local portion of vector *v* or matrix *m*.
- **Recursive_double(dir,v or lv)** In-place recursive double of local vector *lv* or local portion of vector *v*, with processes whose id is identical except for differences in the *dir* index.
- **Shift(car,m)** Send local portion of matrix *m* in cardinal compass direction *car*, receive replacement piece from direction opposite *car*. Edge processes wraparound.
- **Shift(v,amount)** Shift local portion of vector *v* *amount* indices, also with wraparound.

Linear Distributions For ease of explanation, all data distributions are linear and assumed to be load balanced.

4.2 Matrix Matrix Product

Indicative of its wide use, matrix multiplication is a much discussed procedure. Much effort has gone into developing sequential algorithms of order N^x, $2 < x < 3$. Most parallelization efforts have been directed at the 'trivial' N^3 mesh algorithm to multiply a $M \times M$ matrix (B) with a $M \times M$ matrix (C) on a $P \times P$ process grid to produce a $M \times M$ matrix (A). In this section, a variant of the mesh algorithm is developed which mitigates the high cost of communication on NsW.

Standard Mesh Algorithm

The basic distributed mesh solution on a $P \times P$ process grid is given by Cannon's algorithm [6] and illustrated for $P = 2$ in Figure 4.1. The initial configuration of the matrices is not a trivial decomposition. In process row i matrix B is shifted left by i processes, while in process column j matrix C is shifted up by j processes. This configuration requires $O(P)$ communications to setup from a block decomposition. The mesh algorithm leaves the matrices back in the initial configuration, and thus this cost can be amortized over repeated multiplications. This initial configuration is common to both algorithms analyzed here, and is thus not considered in the comparative performance.

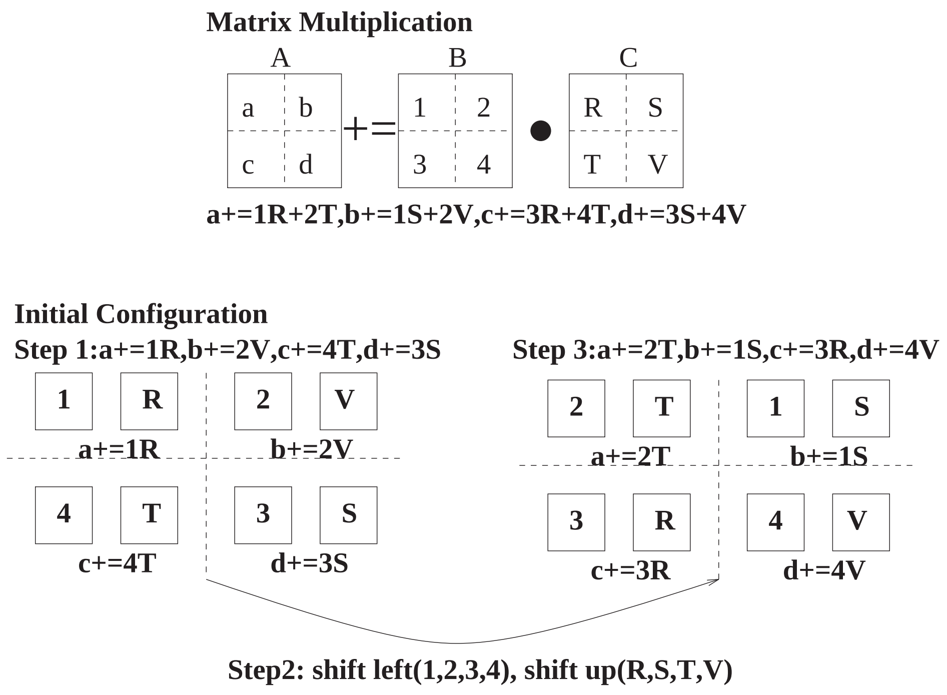
Matrix Multiplication
A
B
C
a
b
c
d
+=
1
2
3
4
●
R
S
T
V
a+=1R+2T,b+=1S+2V,c+=3R+4T,d+=3S+4V
Initial Configuration
Step 1:a+=1R,b+=2V,c+=4T,d+=3S
a+=1R
b+=2V
c+=4T
d+=3S
Step 3:a+=2T,b+=1S,c+=3R,d+=4V
a+=2T
b+=1S
c+=3R
d+=4V
Step2: shift left(1,2,3,4), shift up(R,S,T,V)

Figure 4.1: Matrix Multiplication: Cannon's algorithm

The computational part of Cannon's algorithm proceeds with a local matrix-matrix product on each process, followed by all processes shifting matrix B to the left and matrix C up. This is repeated P times:

```
Matrix A(M,M); Matrix B(M,M); Matrix C(M,M);
for (n=0;n<P;n++) {
  A+=B*C;
  Shift(WEST,B) || Shift(NORTH,C);
}
```

The Execution time is:

$$\tau_{mm1} = P[compute + Shift(West) + Shift(North)]$$
$$\tau_{mm1} = P\left[\frac{2M^3}{P^3} + \tau_{ca}\left(\frac{MN}{P^2}, 2P^2\right)\right]$$

Cannon's algorithm requires memory to store matrices A,B and C and one temporary matrix used during each shift operation.

$$MEM_{mm1} = 4\frac{M^2}{P^2}$$

Communication Hiding

Cannon's algorithm requires each process to wait for each shift operation to complete before beginning computation of the next local product. An algorithm which can compute while waiting for a shift operation would perform better on a NW.

Furthermore, communications in both ROW and COL directions are occurring simultaneously. On an architecture with dedicated communication channels from each process to neighboring processes, this is possible. However, on a network such as Ethernet or FDDI where all processes share a channel, these communications will be serialized.

To hide the communication, the local portions of matrices B and C are each decomposed into two portions: Bf, Bs and Cf, Cs. The local portion of matrix A is decomposed into four portions: Aff,Afs,Asf,Ass, as follows:

Bf Rows $0 \leq r < \frac{M}{2P}$ of B.

Bs Rows $\frac{M}{2P} \leq r < \frac{M}{P}$ of B.

Cf Columns $0 \leq c < \frac{M}{2P}$ of C.

Cs Columns $\frac{M}{2P} \leq c < \frac{M}{P}$ of C.

Aff Columns $0 \leq c < \frac{M}{2P}$ of Rows $0 \leq r < \frac{M}{2P}$ of A.

Asf Columns $0 \leq c < \frac{M}{2P}$ of Rows $\frac{M}{2P} \leq r < \frac{M}{P}$ of A.

Afs Columns $\frac{M}{2P} \leq c < \frac{M}{P}$ of Rows $0 \leq r < \frac{M}{2P}$ of A.

Ass Columns $\frac{M}{2P} \leq c < \frac{M}{P}$ of Rows $\frac{M}{2P} \leq r < \frac{M}{P}$ of A.

$A+ = B*C$ is then $Aff+ = Bf*Cf$, $Afs+ = Bf*Cs$, $Asf+ = Bs*Cf$ and $Ass+ = Bs*Cs$. The computation of each decomposed portion of A can now occur concurrently with a shift of one of the four portions of matrices B and C.

This approach is illustrated in Figure 4.2 for $P = 2$. Each process must perform twice as many communication actions as before, and the algorithm requires the ability to use column as well as row oriented matrices. However, each communication is only half as large, and can now be done in parallel with computation. Furthermore, only half as many messages are being transmitted simultaneously.

To derive the algorithm, first consider the four steps per iteration:

```
Matrix A(M,M); Matrix B(M,M); Matrix C(M,M);
for (n=0;n<P;n++) {
  par { Ass+=Bs*Cs; Shift(WEST,Bf); }
  par { Asf+=Bs*Cf; Shift(NORTH,Cs); }
  par { Afs+=Bf*Cs; Shift(NORTH,Cf); }
  par { Aff+=Bf*Cf; Shift(WEST,Bs); }
}
```

The CC++ semantics of a `par` are that the block terminates when all statements in the block terminate, and that statements in the block are executed in parallel in a weakly fair interleaved manner. Now assign indices to the matrix portions, checking whether each pair of indices match. Each iteration begins with the ith piece of each portion of the matrix in the local process:

```
Matrix A(M,M); Matrix B(M,M); Matrix C(M,M);
for (n=0;n<P;n++) {
  par {Ass+=Bs(i)*Cs(i);     {ASend(WEST,Bf(i));  Bf(i+1)=Receive(EAST);}}
  par {Asf+=Bs(i)*Cf(i);     {ASend(NORTH,Cs(i)); Cs(i+1)=Receive(SOUTH);}}
  par {Afs+=Bf(i+1)*Cs(i+1); {ASend(NORTH,Cf(i)); Cf(i+1)=Receive(SOUTH);}}
  par {Aff+=Bf(i+1)*Cf(i+1); {ASend(WEST,Bs(i));  Bs(i+1)=Receive(EAST);}}
}
```

Finally, factor the loop so the matrices return to their starting positions:

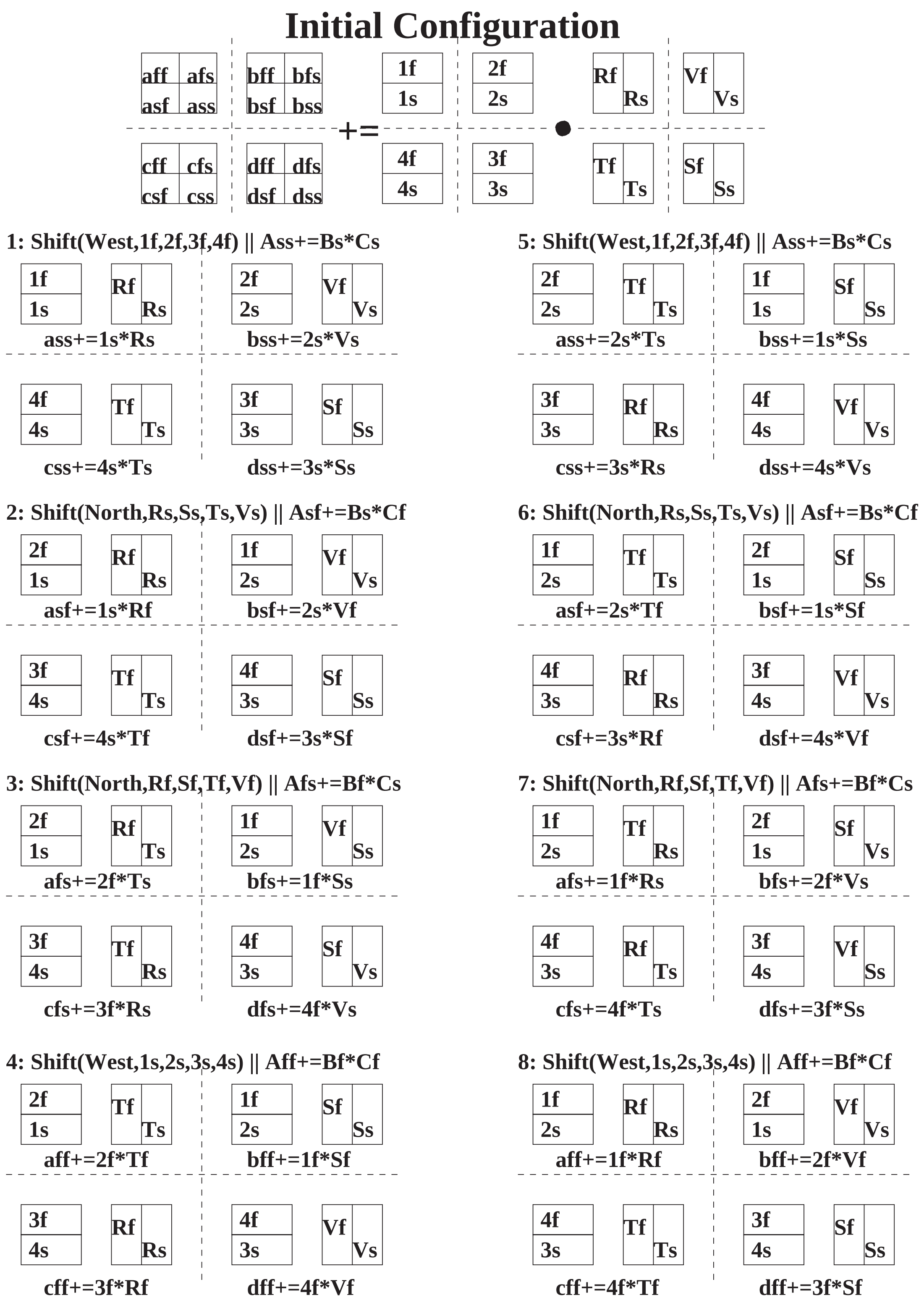

Figure 4.2: Matrix Multiplication: Communication Hiding Algorithm

```
Matrix A(M,M); Matrix B(M,M); Matrix C(M,M);
par {Ass+=Bs*Cs; {ASend(WEST,Bf);  Bf=Receive(EAST);} }
par {Asf+=Bs*Cf; {ASend(NORTH,Cs); Cs=Receive(SOUTH);} }
for (n=0;n<(P-1);n++) {
  par {Afs+=Bf*Cs; {ASend(NORTH,Cf); Cf=Receive(SOUTH);} }
  par {Aff+=Bf*Cf; {ASend(WEST,Bs);  Bs=Receive(EAST);} }
  par {Ass+=Bs*Cs; {ASend(WEST,Bf);  Bf=Receive(EAST);} }
  par {Asf+=Bs*Cf; {ASend(NORTH,Cs); Cs=Receive(SOUTH);} }
}
par {Afs+=Bf*Cs; {ASend(NORTH,Cf); Cf=Receive(SOUTH);} }
par {Aff+=Bf*Cf; {ASend(WEST,Bs);  Bs=Receive(EAST);} }
```

The execution time of this algorithm is:

$$\tau_{mm2} = 4P[compute \parallel shift]$$

The parallel execution of *compute* and *shift* introduces the overhead of a `par` of size 2. Two threads are created, one to handle *shift* and one to handle *compute*. In order to overlap the communication and computation, however, the communication needs to start first. In CC++ the `par` does not specify which thread executes first; this issue of controlling thread interleaving is seen in all algorithms presented here.

Assuming the communication thread executes first, $\tau_{cab}\left(\frac{M^2}{2P^2}, P^2\right)$ is required to initiate the communication, after which time the computation can begin execution. When the computation completes, if the remaining communication ($\tau_{caf}\left(\frac{M^2}{2P^2}, P^2\right)$) has completed, the `Receive` can complete; otherwise the remaining communication must be awaited. The execution time of τ_{mm2} is thus:

$$\tau_{mm2} = 4P\left[\tau_{par}(2) + \tau_{cab}\left(\frac{M^2}{2P^2}, P^2\right) + \frac{M^3}{2P^3} max\left(0, \tau_{caf}\left(\frac{M^2}{2P^2}, P^2\right) - \frac{M^3}{2P^3}\right)\right]$$

This algorithm uses less temporary memory than Cannon's, since only half a matrix is communicated on each shift.

$$MEM_{mm2} = 3.5 * \frac{M^2}{P^2}$$

Theoretical Performance Comparison

The difference in execution time between Cannon's and the communication hiding algorithm is:

$$\begin{aligned}\tau_{mm1} - \tau_{mm2} = {} & P\left[\tau_{ca}\left(\frac{M^2}{P^2}, 2P^2\right)\right] - \\ & 4P\left[\tau_{par}(2) + \tau_{cab}\left(\frac{M^2}{2P^2}, P^2\right) + max\left(0, \tau_{caf}\left(\frac{M^2}{2P^2}, P^2\right) - \frac{M^3}{2P^3}\right)\right]\end{aligned}$$

The potential benefit of communication hiding comes at the overhead of more communications (P vs $4P$). For constant P, this will result in performance *degradation* for small M, until communication startup costs become negligible compared to message transmission times. Performance improves as $max(0, \tau_{caf} - compute)$ goes to 0, reaching a peak before the communication (growing at M^2) becomes negligible compared to the computation (growing at M^3). The difference in performance, as a percentage of total execution time, then tails off at $\frac{1}{M}$.

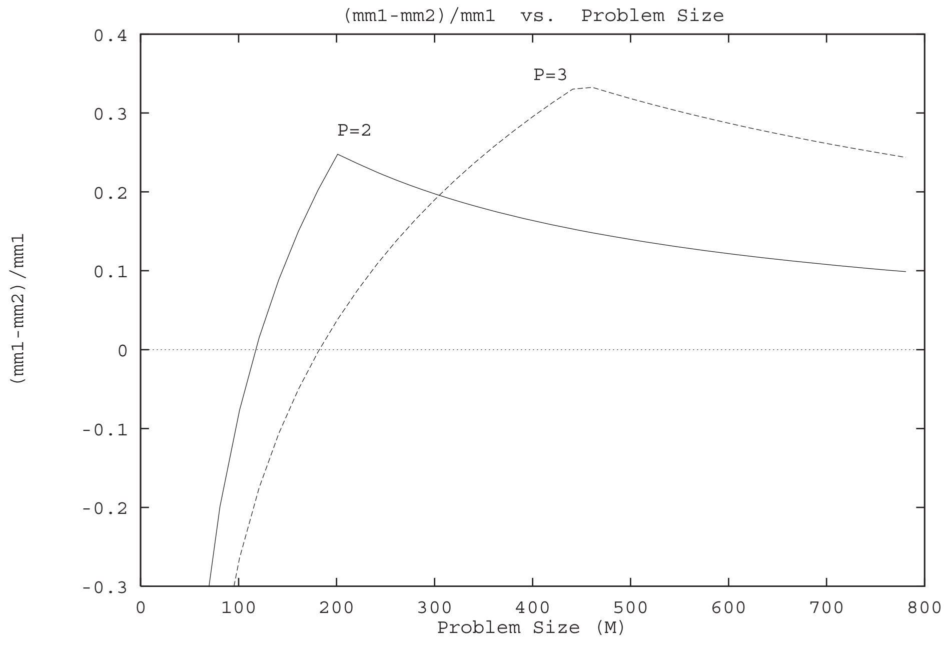

Figure 4.3: Matrix Multiplication: Theoretical Performance on Setup I

Figure 4.3 shows $mm1 - mm2$ as a fraction of $mm1$ for $P = 2, 3$ using the parameters measured in Chapter 2 for Setup I.

As P increases, the memory size at which maximum performance improvement occurs increases. This is seen in Figure 4.4, where $(mm1 - mm2)/mm1$ is plotted versus $\frac{M}{P}$, again using Setup I parameters. Since $MEM_{mm2} = 3.5\frac{M^2}{P^2}$, a $\frac{M}{P}$ of 1000 means 27MB of memory per process. The upper bound on the P for which significant performance improvement can be achieved is determined by the maximum allowable memory size per process.

Experimental Performance Comparison

The algorithms described above were implemented in DLA. Figures 4.5 and 4.6 show the measured $mm1 - mm2$ as a fraction of $mm2$ on Setup I for $P = 2$ and $P = 3$. The maximum performance improvement is slightly less in magnitude and at different problem sizes than predicted, for both $P = 2$ and $P = 3$. The decay after the peak is also faster than predicted.

This discrepancy might be caused by an inaccurate equation for network performance under heavy load. As the background load increases, the maximum performance improvement is shifted up and right (to larger problem sizes). The effects of a perturbation in background load between that present when $\tau_{ca}(L, C)$ is measured and that seen by the algorithm are magnified by the large load matrix multiplication itself generates.

NsW are not usually isolated for the purpose of ensuring maximum computing power. An algorithm which is less affected by network load is therefore preferable. If network load increases, $mm2$ might hide the increased communication cost behind computation, depending on M. $mm1$ will always suffer degradation.

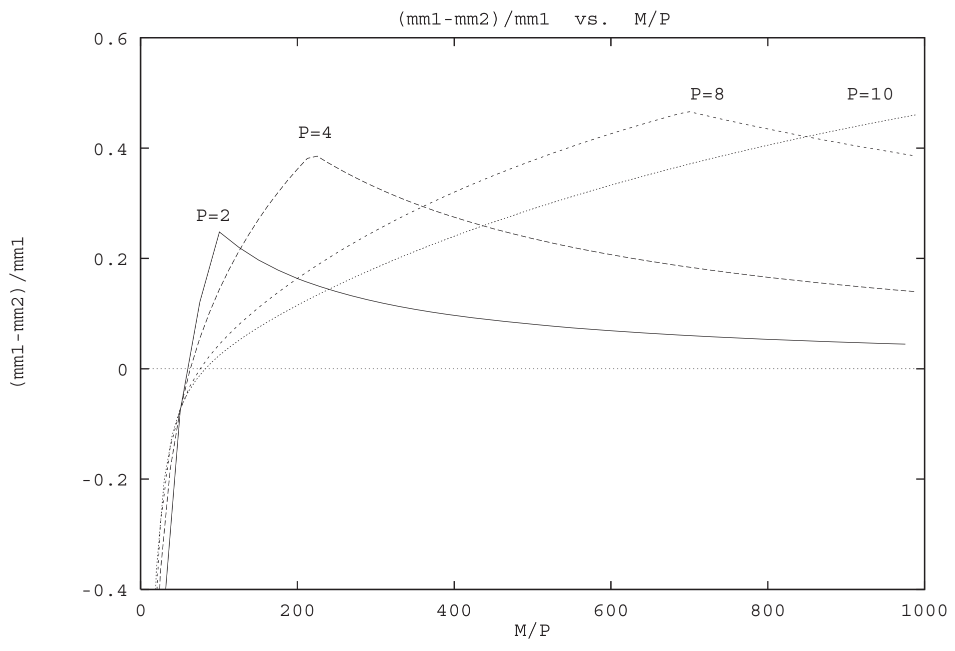

Figure 4.4: Matrix Multiplication: Theoretical Performance: Problem Size per Process

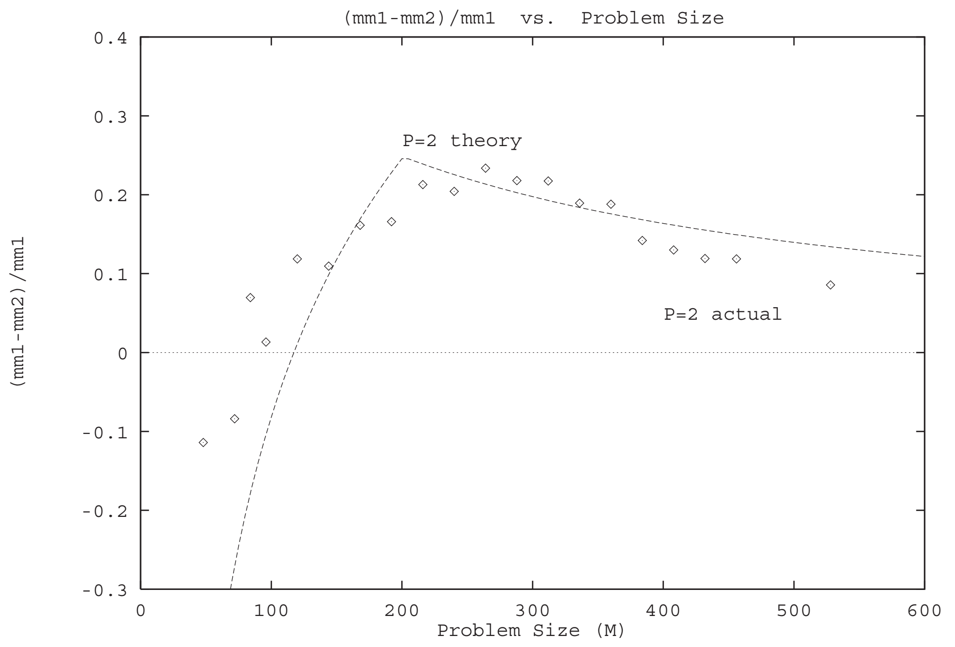

Figure 4.5: Matrix Multiplication: Experimental Performance on Setup I for P=2

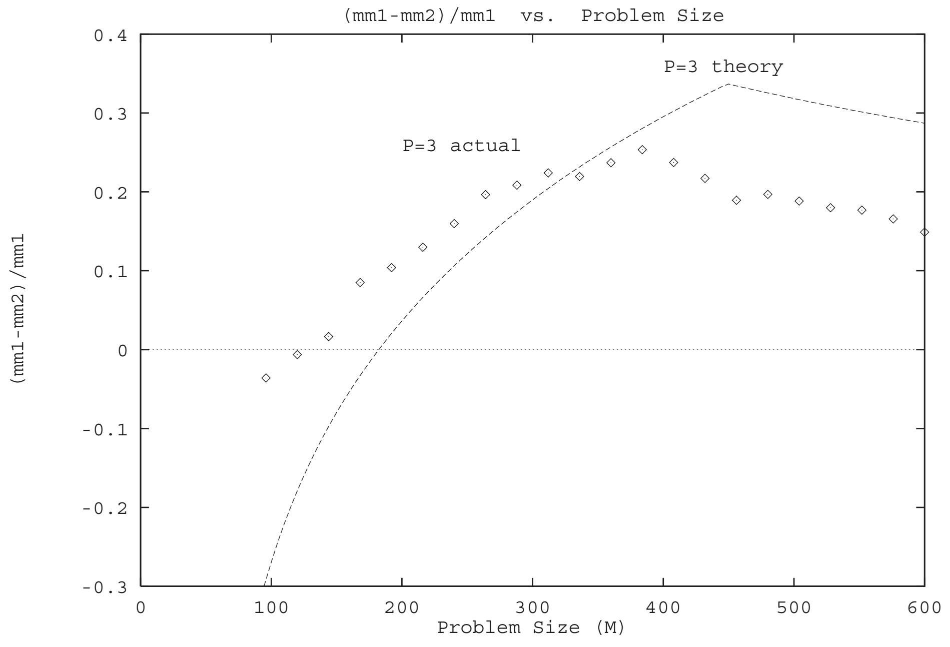

Figure 4.6: Matrix Multiplication: Experimental Performance on Setup I for P=3

4.3 Dense Matrix Vector Product

Standard Algorithm

In the standard algorithm for matrix vector product [9], illustrated in Figure 4.7, the M by N matrix A is distributed over a $P \times Q$ process grid, while the N by 1 input vector X is column distributed across the same process grid.

Each process first executes a local matrix vector product and then performs a recursive double with processes in its row. The resultant M by 1 output vector Y is then row distributed. A program which implements this algorithm is:

```
Vector Y(M); Matrix A(M,N); Vector X(N); LVector T(M/P);
T=A*X;
T=RecursiveDouble(COL,T);
Y+=T;
```

The execution time of this algorithm is:

$$\begin{aligned}
\tau_{mv2D} &= [compute] + [recursive\_double] + [accumulate] \\
\tau_{mv2D} &= \left[\frac{M}{P} + \frac{2MN}{PQ}\right] + \left[\tau_{ca}\left(\frac{M}{P}, Q\right) log_2 Q\right] + \left[\frac{M}{P}\right]
\end{aligned}$$

This algorithm requires memory for storage of the matrix A, vectors X and Y, and the temporary vector T. The recursive double can be done in-place using vector T and does not require any added

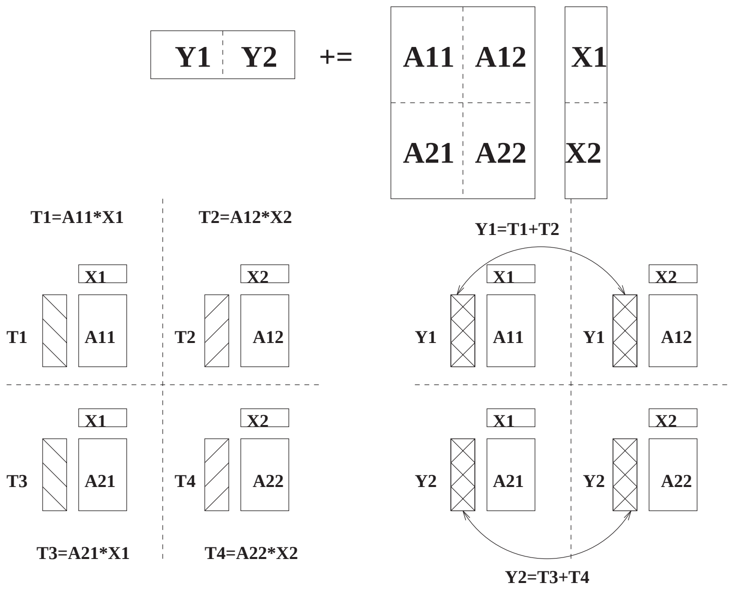

Figure 4.7: Matrix Vector Product: Standard Algorithm

memory. The total memory needed per process is:

$$MEM_{mv2D} = \frac{MN}{PQ} + \frac{N}{Q} + \frac{2M}{Q}$$

Recursive doubling is a communication requiring synchronization after each step. On each process, the next step of the recursive doubling procedure can not be initiated until the previous one has completed. Thus, the ability to overlap communication with computation can not be exploited.

Communication Hiding Algorithm

An algorithm which creates more communication, but allows that communication to be possibly hidden by computation is now developed. This algorithm is illustrated in Figure 4.8, where the matrix A is distributed over a $1 \times Q$ process grid, so that each process receives a M by $\frac{N}{Q}$ portion of A. Both X and Y are distributed column wise.

Let y_{ij} be the partial result vector formed by $A_{ij}X_j$. Multiplication of the local matrix and input vector on process q results in process q containing $\forall v : v\epsilon 0..P-1 :: y_{vq}$. Process q must accumulate $\sum_{j=0}^{Q-1} y_{qj}$ for the result to be column distributed. Each process must thus send partial results to all other processes.

Such a broadcast operation involves $(Q-1)Q$ communications, while the recursive double in the standard algorithm requires $Q * log_2 Q$. For $Q > 3$, this will result in more communication. However, because calculation of y_{vq} is independent of the calculation of $y_{(v+1)q}$, the communication of y_{vq} to process v can be performed in parallel with the computation of $y_{(v+1)q}$.

This can be expressed as follows, where q is the local process identifier:

```
Vector Y(M); Matrix A(M,N); Vector X(N); LVector T(M/Q);
parfor (int reldest=1;reldest<(Q+1);reldest++) {
  int j=(q+reldest)%Q; int src=(q+Q-reldest)%Q;
  // reldest is destination relative to q for this partial product
```

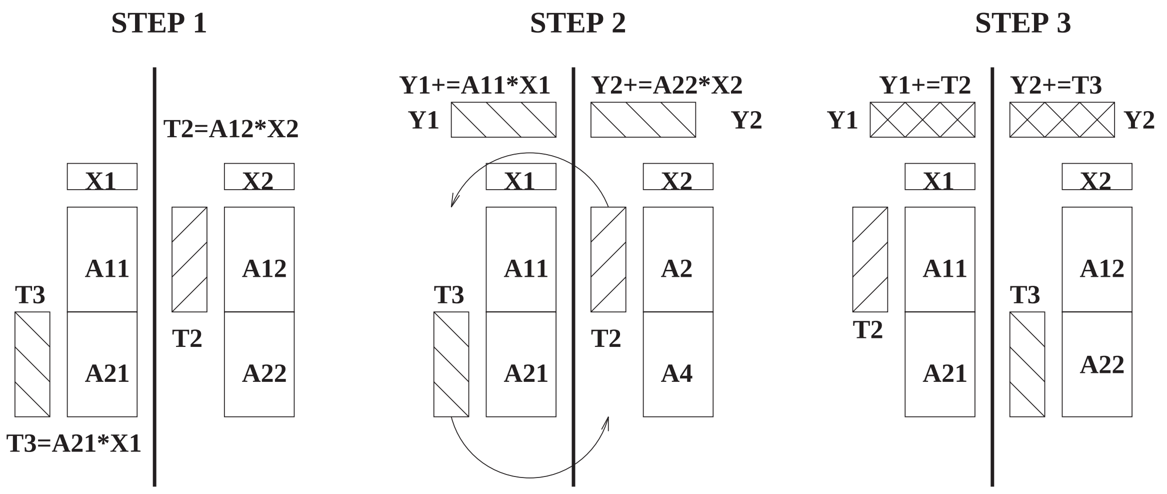

Figure 4.8: Matrix Vector Product: Communication Hiding Algorithm

```
  // j is absolute destination for this partial product
  // src is absolute source for received partial product
  T=Aqj*X;
  if (j==q)
    Y.Add_to(T);
  else {
    ASend (COL,j,T);
    Y.Add_to(Receive(COL,src));
  }
}
```

Each iteration of the `parfor` block exchanges $A_{jq}X_q$ for $A_{qj}X_j$. The semantics of `parfor` in CC++ indicate that the Q iterations of the block body will execute in parallel. On a uniprocessor each iteration will be handled by one thread. The possible interleavings of the iterations must be constrained such that at most one iteration is accumulating portions of the result into vector Y at any one time. The `atomic` function `Add_to(LVector)` is used to guarantee this, as in CC++ at most one thread can be accessing an object through one of that object's `atomic` member functions at any one time.

Ideally, each local product will be computed and the send initiated, then that iteration will wait for the corresponding receive to arrive while another iteration computes the next local product. As soon as the receive arrives, the appropriate iteration will accumulate it into the result vector, preventing the unprocessed receive message buffers from accumulating. Figure 4.9 shows this ideal interleaving of the iterations for one process when $Q = 5$.

Like `par`, `parfor` does not specify the interleaving of the various instances of the loop body. All threads might compute $Aqj * X$, then relinquish control. This would result in Q temporary buffers, each of size $\frac{M}{Q}$ being created, and the communication only taking place after all computation was complete. More likely in a non-preemptive environment, a thread may not be returned control as soon as the receive arrives, leading to memory pileup.

Assuming the ideal execution profile is achieved, the execution time is:

$$\tau_{mv1D} = [thread\_overheads] + [computation] + [communication] + [accumulate]$$

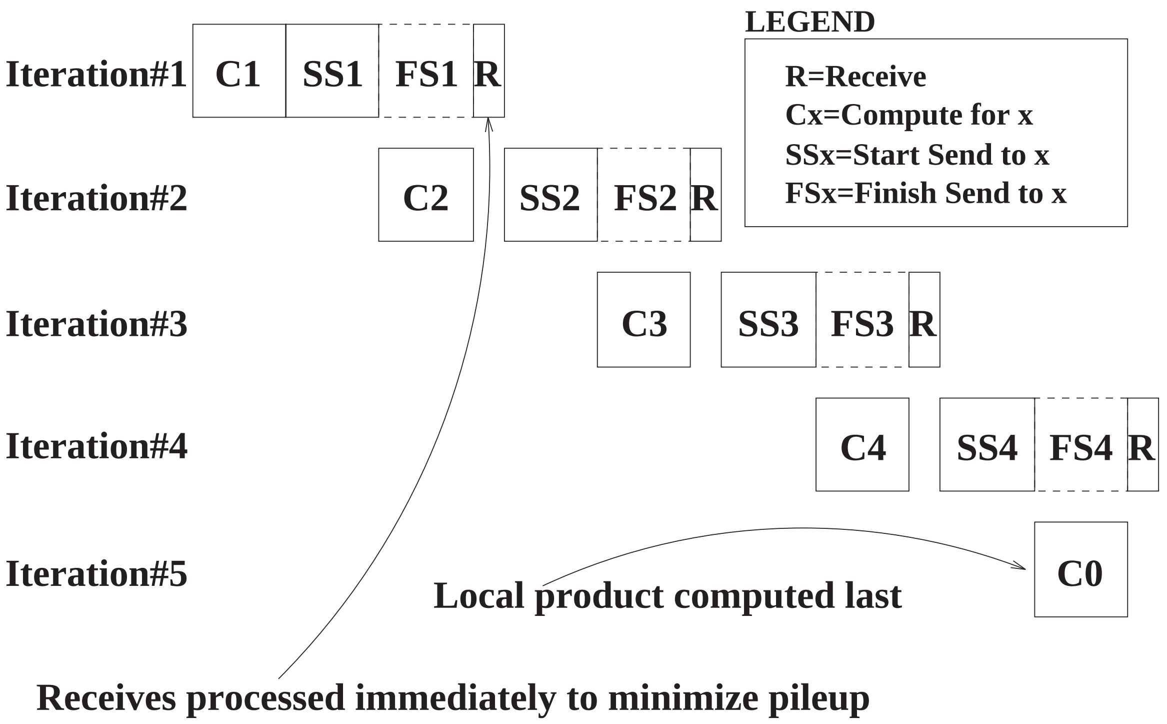

Figure 4.9: Matrix Vector Product: Ideal Execution Trace of Communication Hiding Algorithm

$$
\begin{aligned}
\tau_{mv1D} \quad = \quad & [\tau_{parfor}(Q) + Q * \tau_{atomic}] + \left[\frac{2MN}{Q} + M\right] + \\
& \left[(Q-1)\tau_{cab}\left(\frac{M}{Q}, Q\right) + (Q-1)max\left(0, \tau_{caf}\left(\frac{M}{Q}, Q\right) - \frac{2MN}{Q^2}\right)\right] + \left[Q\frac{M}{Q}\right]
\end{aligned}
$$

This algorithm requires memory for storage of the matrix A, vectors X and Y, and the temporary vectors T. In the best case, only two temporary vectors exist at any time (one for sending, one for receiving). However, both received and computed buffers may accumulate up to the size of the input vector. The memory needed per process is thus:

$$
\begin{aligned}
\texttt{BEST CASE} \qquad MEM_{mv1D} &= \frac{MN}{Q} + \frac{N}{Q} + \frac{3M}{Q} \\
\texttt{WORST CASE} \qquad MEM_{mv1D} &= \frac{MN}{Q} + \frac{N}{Q} + \frac{M}{Q} + 2M
\end{aligned}
$$

Theoretical Performance Comparison

The developed NW algorithm executes on a 1 by Q process mesh whereas the standard algorithm executes on a P by Q process mesh. If $Q = 1$ then the standard algorithm requires no communication. If $P = 1$ then it requires $log_2 Q$ communication. What P,Q should be used when comparing $mv2D$ to $mv1D$?

These algorithms are basic linear algebra routines, used as part of more complex solvers. In such a situation, the process grid is often constrained for the maximum benefit to the complex solver.

Here the performance for a 1 by Q mesh is analyzed. Assuming $M{=}N$, the difference in execution time between $mv2D$ and $mv1D$ is:

$$\begin{aligned} mv_{2D} - mv_{1D} &= \tau_{ca}\left(\frac{M}{P}, Q\right) log_2 Q - \tau_{parfor}(Q) - Q * \tau_{atomic} - \\ &\quad (Q-1)\tau_{cab}\left(\frac{M}{Q}, Q\right) - (Q-1)max\left(0, \tau_{caf}\left(\frac{M}{Q}, Q\right) - \frac{2MN}{Q^2}\right) \end{aligned}$$

The cost of hiding the communication is the replacement of a recursive double with a broadcast, meaning $mv1D$ will not be preferable for large Q. To be worthwhile, assuming the completion of the communication is hidden by computation and M is large enough to make thread overheads negligible:

$$\begin{aligned} log_2 Q \tau_{ca}\left(\frac{M}{Q}, Q\right) &> (Q-1)\tau_{cab}\left(\frac{M}{Q}, Q\right) \\ log_2 Q \tau_{caf}\left(\frac{M}{Q}, Q\right) &> ((log_2 Q) - Q - 1)\tau_{cab}\left(\frac{M}{Q}, Q\right) \end{aligned}$$

Both $mv1D$ and $mv2D$ perform Q communications at each step, and will thus exhibit the same degradation in communication performance due to network saturation. The effects of the degraded communication will be mitigated in the $mv1D$ algorithm, however, as the communication cost is partially hidden by computation.

Figures 4.10 and 4.11 show the theoretical difference $mv2D - mv1D$ as a fraction of $mv2D$ on Setups I and II. In these comparisons $M = N$.

Both architectures exhibit initial performance degradation, as there is virtually no computation to hide the increased communication. As more computation becomes available with which to hide the communication, performance improves. The improvement reaches a maximum, and then decreases at $\frac{1}{N}$, as the communication becomes negligible compared to the computation for coarse grain sizes.

Figure 4.12 shows the theoretical difference in scaling of $mv2D$ and $mv1D$ on Setup I for $Q = 6$. The most performance improvement occurs in the transition region between communication startup dominated (low scaling) and computation dominated (high scaling) areas. The memory size per process is reasonable in this region (2MB at $M = 1200$), meaning for this NsW $mv1D$ is predicted to be extremely applicable.

As seen in Figure 4.10, as Q is increased, the expected performance improvement occurs for larger M. It also occurs at larger $\frac{M}{Q}$, although this increase is slower. For larger Q than displayed here, the maximum performance improvement decreases. There is a Q above which the improvement is too coarse grained to be realizable in the available memory. For the NW in Setup I, supposing a realistic limit for the matrix under multiplication of 10MB per node, this occurs above $Q = 20$.

The performance improvement is inversely proportional to network throughput, so networks with more background load will exhibit increased improvement. If new network technologies, such as ATM, are able to improve faster than processor speeds, then the improvements achieved by $mv1D$ will be reduced.

Experimental Performance Comparison

The algorithms described above were implemented in DLA. Figures 4.13, 4.14, and 4.15 show the measured difference $mv2D - mv1D$ as a fraction of $mv2D$ on Setups I and II. The results have the form and approximate magnitude of that predicted. In Setup II the performance is better than expected, whereas in Setup I it is worse, particularly for $Q = 2$.

Figure 4.16 shows the scaling of $mv2D$ versus that of $mv1D$ for $Q = 6$ on Setup I. The difference

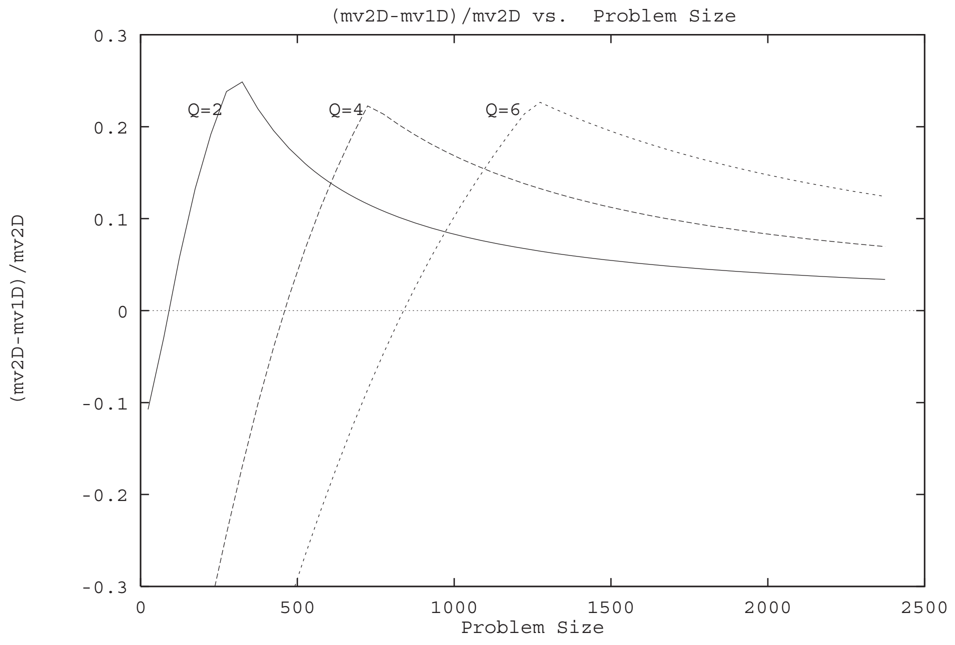

Figure 4.10: Matrix Vector Product: Theoretical Performance on Setup I

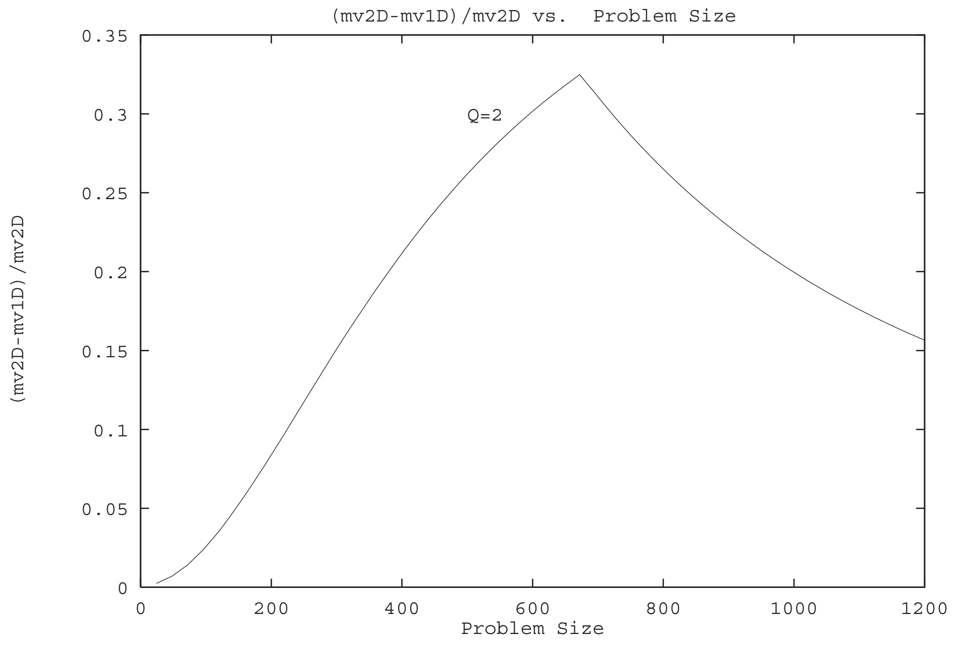

Figure 4.11: Matrix Vector Product: Theoretical Performance on Setup II

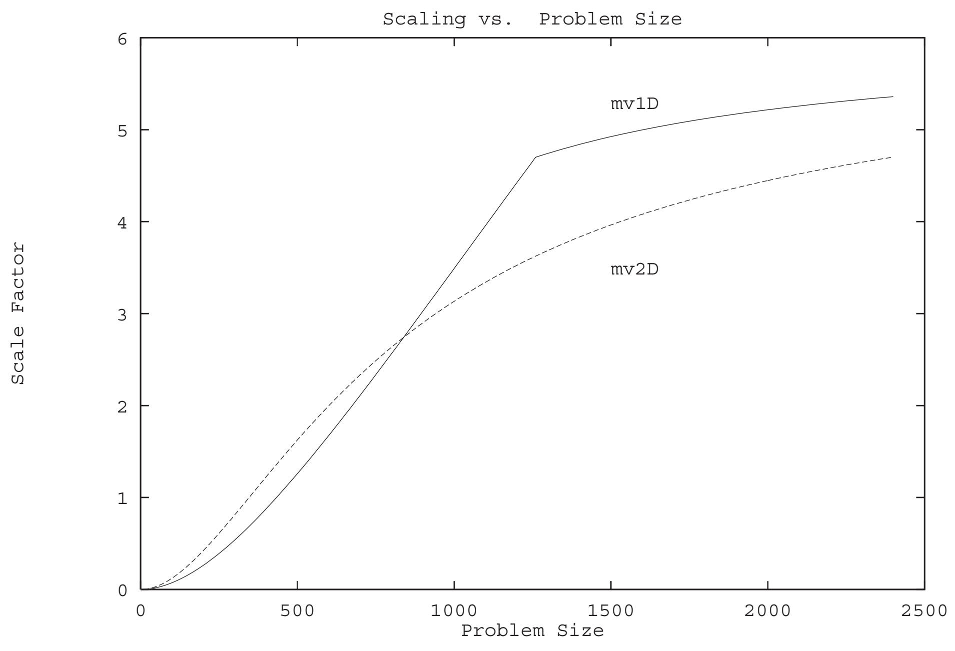

Figure 4.12: Matrix Vector Product: Theoretical Scaling on Setup I for Q=6

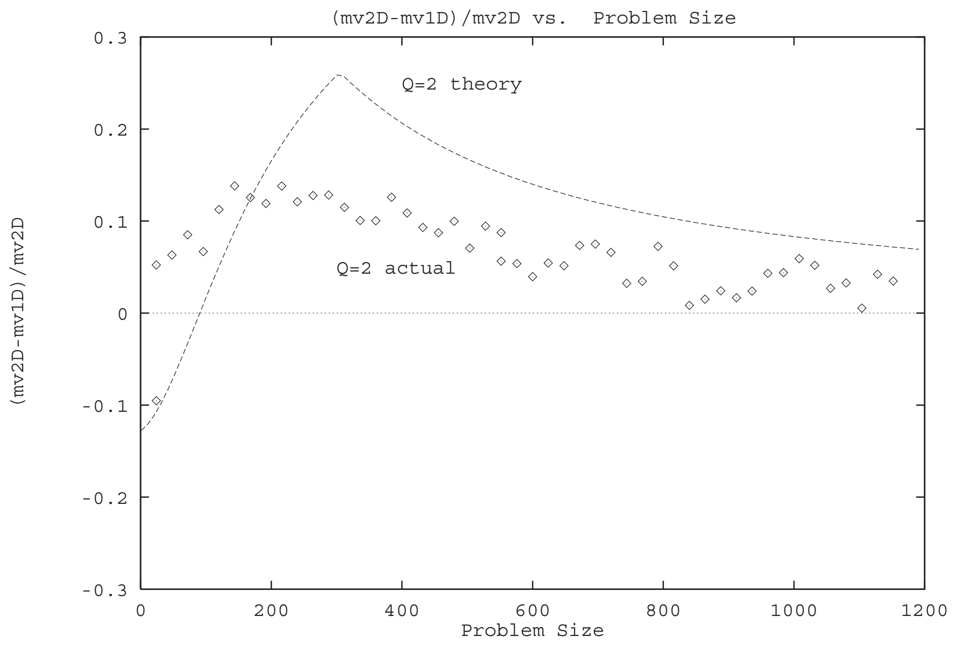

Figure 4.13: Matrix Vector Product: Experimental Performance on Setup I for Q=2

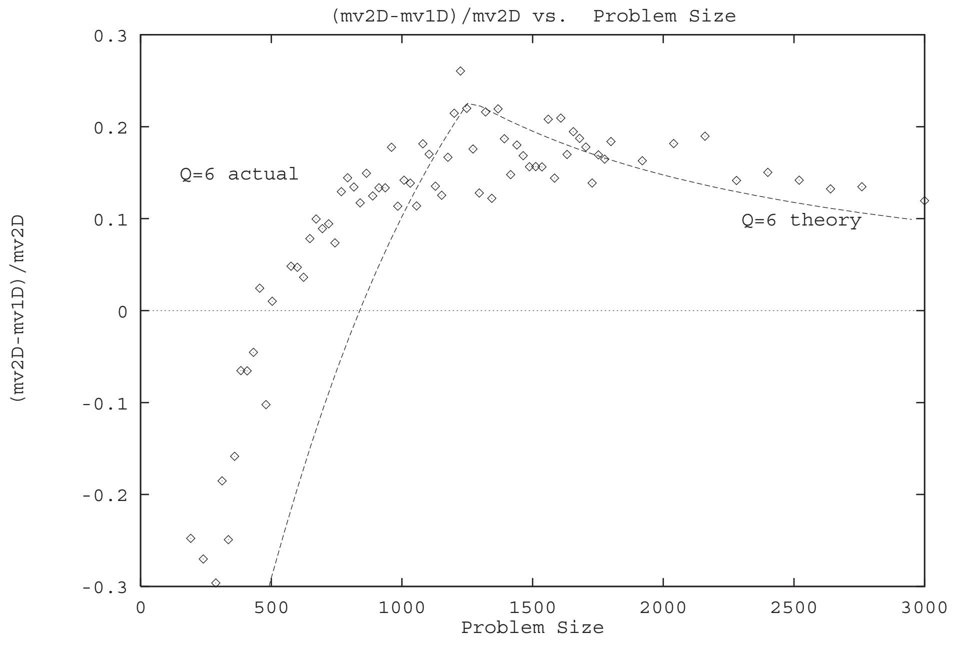

Figure 4.14: Matrix Vector Product: Experimental Performance on Setup I for Q=6

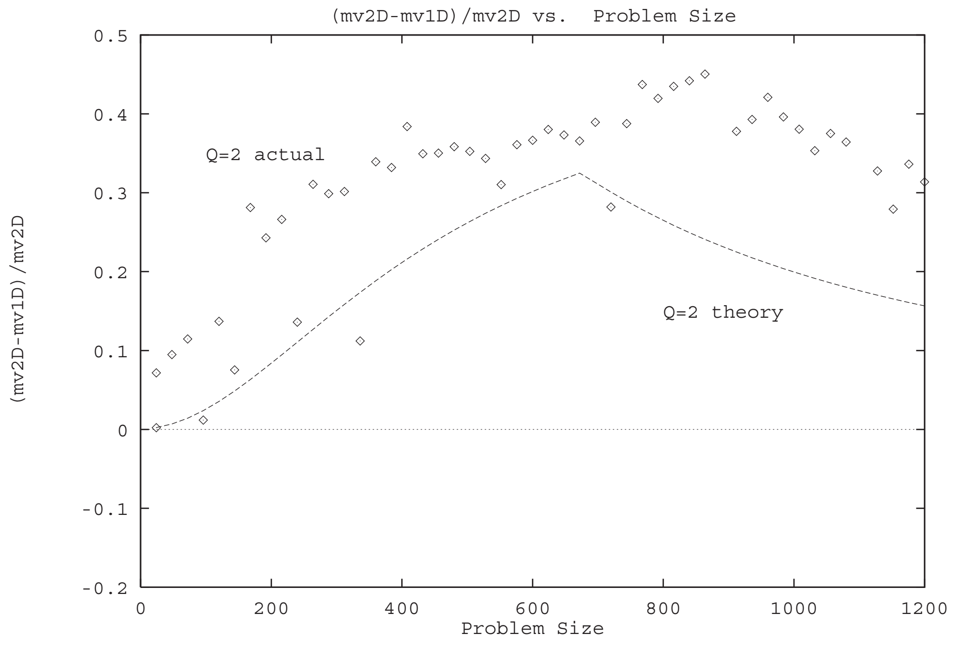

Figure 4.15: Matrix Vector Product: Experimental Performance on Setup II

in scaling is slightly better than expected. Problem sizes larger than those shown are not tractable on $Q = 1$ due to memory limitations, therefore a scaling comparison is not meaningful.

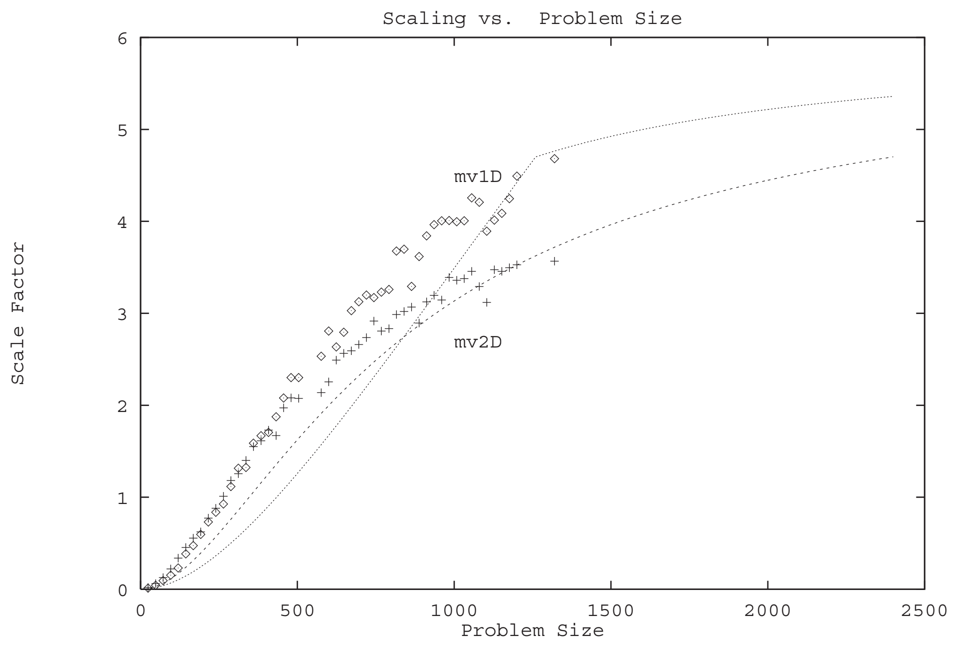

Figure 4.16: Matrix Vector Product: Experimental Scaling on Setup I for Q=6

4.4 Sparse Diagonal Matrix Vector Product

Sparse Diagonal Matrix Vector Multiplication (SDMV) multiplies a $M \times N$ matrix (A), with fill only along D diagonals, by a N by 1 vector (X), producing a M by 1 result vector (Y). Sparse matrices can have very low fill, thus the overhead associated with complex matrix representations are often worthwhile. Here the diagonal form introduced in [9] is used. The matrix is represented as an array of diagonals:

```
class SDMatrix {
public:
  int grows,gcols; // Global Size
  int cols;        // Local Size
  int ndiags;      // Number of Diagonals
  Type** diags;    // Array of size ndiags, each of an array of size cols.
  int* diag_nums;  // Array of size ndiags, gives diagonal number of diags[k]
};
```

Multicomputer Shift Algorithm

On a $1 \times Q$ process grid, where the diagonals, input and output vectors are distributed across the process columns, SDMV is commonly implemented via the vector shift operation, as given in [9]:

```
Vector Y(M); SDMatrix(M,N); Vector X(N);
```

```
Shift(Y,A.diag_nums[0]);
for (int d=0;d<A.ndiags;d++) {
  Y+=A.diags[d]*X;
  if (d==(A.ndiags-1)) Shift(Y,A.grows-A.diag_nums[d]);
  else Shift(Y,A.diag_nums[d+1]-A.diag_nums[d]);
}
```

This algorithm is illustrated in Figure 4.17 for a simple example.

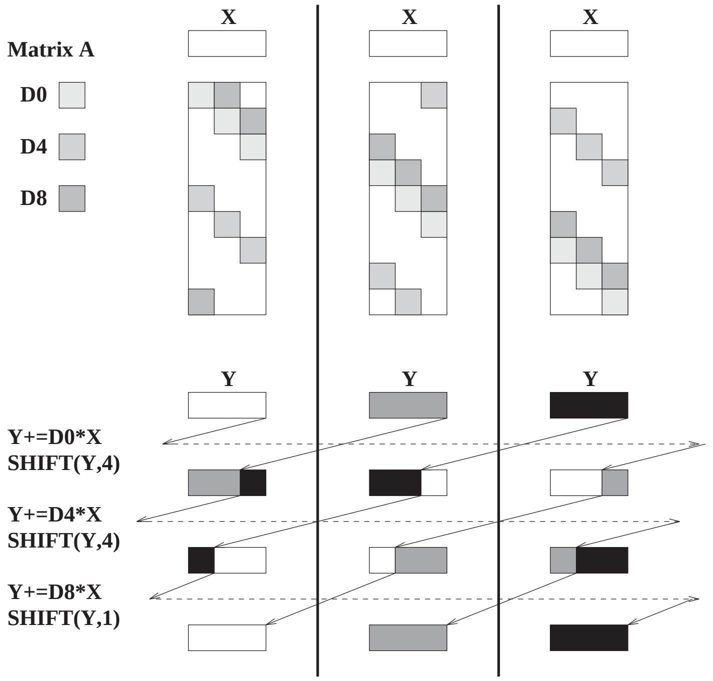

Figure 4.17: SDMV: Shift Algorithm

D shift operations are performed, during each of which no computation can be performed. If the diagonal number of the ith diagonal is labeled $dn(i)$, then the execution time is:

$$\tau_{Shift} = \frac{2ND}{Q} + \tau_{ca}(dn(0), Q) + \sum_{i=1}^{D-1} \left(\tau_{ca}\left(dn(i) - dn(i-1), Q\right)\right) + \tau_{ca}(M - dn(D), Q)$$

In this analysis the diagonals are assumed to be evenly spaced through the matrix, meaning each shift is of length $\frac{M}{D}$. The execution time is then:

$$\tau_{Shift} = \frac{2ND}{Q} + D\tau_{ca}\left(\frac{M}{D}, Q\right)$$

The memory use is the distributed matrix, the distributed result vector, the distributed input vector, and an extra distributed vector. The extra distributed vector is used with the result vector as a wraparound buffer to implement shift without copying the unshifted portion of the vector.

$$MEM_{Shift} = \frac{ND}{Q} + \frac{N}{Q} + \frac{2M}{Q}$$

High Diagonal Algorithm

Rather than Shift the result vector after each diagonal has been multiplied by the local part of X, the product of the local portions of all diagonals and X can be accumulated into a buffer of size M. A portion of this buffer is then sent to each process. This results in $Q-1$ communications instead of D. For most applications, D is between 5 and 20, and definitely bounded by 50. On a multicomputer, with Q in the hundreds, this change results in performance degradation. On a network of workstations, however, with Q in the same order of magnitude as D, a significant amount of communication time can be saved.

The product of a diagonal numbered d (from the main diagonal) and the local portion of X accumulates the result in positions d to $d+\frac{N}{Q}$, where positions greater than N wraparound. Positions $\frac{iM}{Q}$ to $\frac{(i+1)M}{Q}$ belong to the portion of the result vector local to process $(q+i) \bmod Q$.

The algorithm can be implemented as follows:

```
Vector Y(M); SDMatrix(M,N); Vector X(N); LVector T(N);
T=0;
for (int d=0;d<NUM_DIAGS;d++) {
  (T+d)+=A[d]*X;
}
Y+=T[0..N/Q];
for (dest=1;dest<Q;dest++) {
  Send(COL,(q+dest)%Q,T[dest*N/Q..(dest+1)*N/Q]);
  Y+=Receive(COL,(q+Q-dest)%Q);
}
```

This algorithm is illustrated in Figure 4.18 on the toy example used previously.

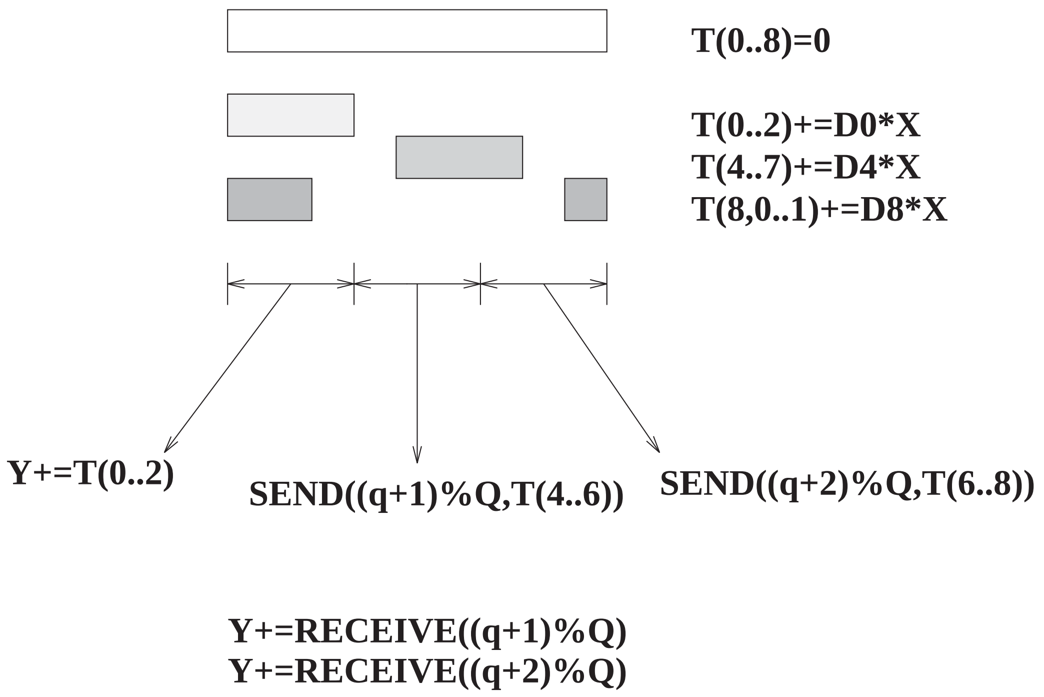

Figure 4.18: SDMV: High Diagonal Algorithm

The execution time is the actual matrix computation (with added zeroing) plus $(Q-1)$ communications and the time needed to accumulate the received results. Each communication is of size $\frac{M}{Q}$.

$$\tau_{HighD} = \frac{2ND}{Q} + M + (Q-1)\tau_{ca}\left(\frac{M}{Q}, Q\right) + M$$

The memory use is the distributed matrix, the distributed result vector, the distributed input vector and a temporary vector the size of the full result vector.

$$MEM_{HighD} = \frac{ND}{Q} + \frac{M}{Q} + \frac{N}{Q} + M$$

The non-scalable memory use of the full input vector clearly needs to be avoided.

Communication Hiding

All the diagonals which partly or fully contribute to the result destined for process q from process p can be accumulated together into a buffer of size $\frac{M}{Q}$. This buffer can be sent to q and then reused to compute the result destined for process $q+1$.

Furthermore, since the computation of the buffer for process q and process $q+1$ are independent, the computation of a new buffer can be done concurrently with the communication of the previous buffer. Thus, $Q-1$ sends of time τ_{ca} in the previous algorithm are replaced with $Q-1$ sends of time τ_{cab}.

The cost is overhead in thread management, any unhidden communication, and time required to determine which sections of which diagonals should be multiplied with the input vector the produce the result destined for a given process. This is represented by the function *compute_for* in the code below. This overhead is linearly proportional to the number of diagonals, and is thus negligible for realistic problem sizes.

```
Vector Y(M); SDMatrix(M,N); Vector X(N); LVector T(N/Q);
parfor (int reldest=1;reldest<(Q+1);reldest++) {
  int j=(q+reldest)%Q; int src=(q+Q-reldest)%Q;
  // reldest is destination relative to q for this partial product
  // j is absolute destination for this partial product
  // src is absolute source for received partial product
  T=compute_for(A,X,reldest%Q);
  if (j==q)
    Y.Add_to(T);
  else {
    ASend (COL,j,T);
    Y.Add_to(Receive(COL,src));
  }
}
```

In addition, the function *Add_to*, which accumulates into the vector Y, must be atomic so as to make the result deterministic; this imposes additional overhead.

The execution of this algorithm on one process for the example worked previously is shown in Figure 4.19

The execution time is:

$$\tau_{CH} = [zero + compute] + Parfor + [accumulate] + [(Q-1) * communication]$$

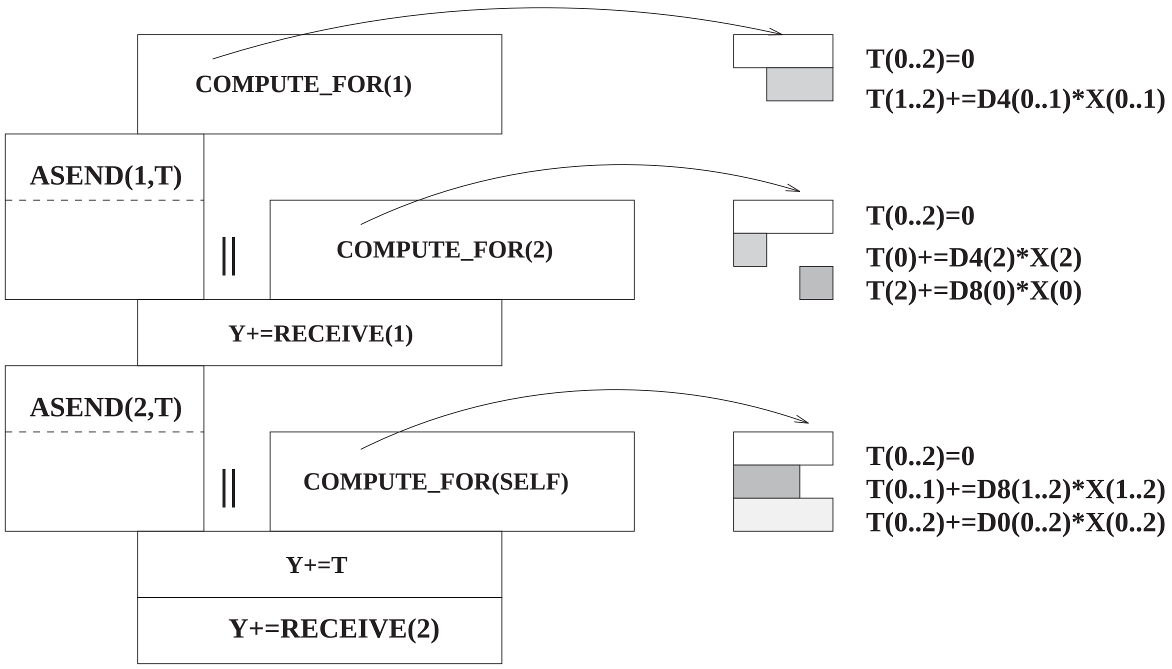

Figure 4.19: SDMV: Communication Hiding Algorithm

$$\begin{aligned}\tau_{CH} \quad = \quad & \left[M + \frac{2ND}{Q}\right] + \tau_{parfor}(Q) + \left[Q\left(\tau_{atomic} + \frac{M}{Q}\right)\right] + \\ & \left[(Q-1)\tau_{cab}\left(\frac{M}{Q}, Q\right) + (Q-1)max\left(0, \tau_{caf}\left(\frac{M}{Q}, Q\right) - \frac{2ND}{Q^2}\right)\right] +\end{aligned}$$

The risk of using a `parfor` in CC++ is that the interleaving of created threads cannot be specified. Each thread might create the temporary buffer and then be switched out. This would result in the same non-scalable memory of size M on each process. Furthermore, the receives may arrive before they are requested, potentially piling up in another M memory. Thus, the memory usage of the communication hiding algorithm is:

$$\begin{aligned}\texttt{BEST CASE} \qquad MEM_{CH} \quad &= \quad \frac{ND}{Q} + \frac{2M}{Q} + \frac{N}{Q} \\ \texttt{WORST CASE} \qquad MEM_{CH} \quad &= \quad \frac{ND}{Q} + \frac{M}{Q} + \frac{N}{Q} + M\end{aligned}$$

Theoretical Performance Comparison

The difference in execution time between *Shift* and *HighD* is:

$$\tau_{Shift} - \tau_{HighD} \quad = \quad D\tau_{ca}\left(\frac{M}{D}, Q\right) - (Q-1)\tau_{ca}\left(\frac{M}{Q}, Q\right) - 2M$$

For extremely small M, communication startup dominates the length dependent‘ communication term, therefore the difference will be proportional to $D - Q + 1$.

As communication startup becomes negligible, $\tau_{ca}(L, Q) \approx aL$, where a is network throughput. Then the difference will approach $aD\frac{M}{D} - a(Q-1)\frac{M}{Q} - 2M$, or $aM\left(\frac{1}{Q} - \frac{2}{a}\right)$. As Q increases, the difference will decrease, eventually becoming negative for $Q > \frac{a}{2}$.

Figure 4.20 shows the theoretical difference $Shift - HighD$ as a fraction of $Shift$ for several Q, using the performance parameters for Setup I obtained in Chapter 2.

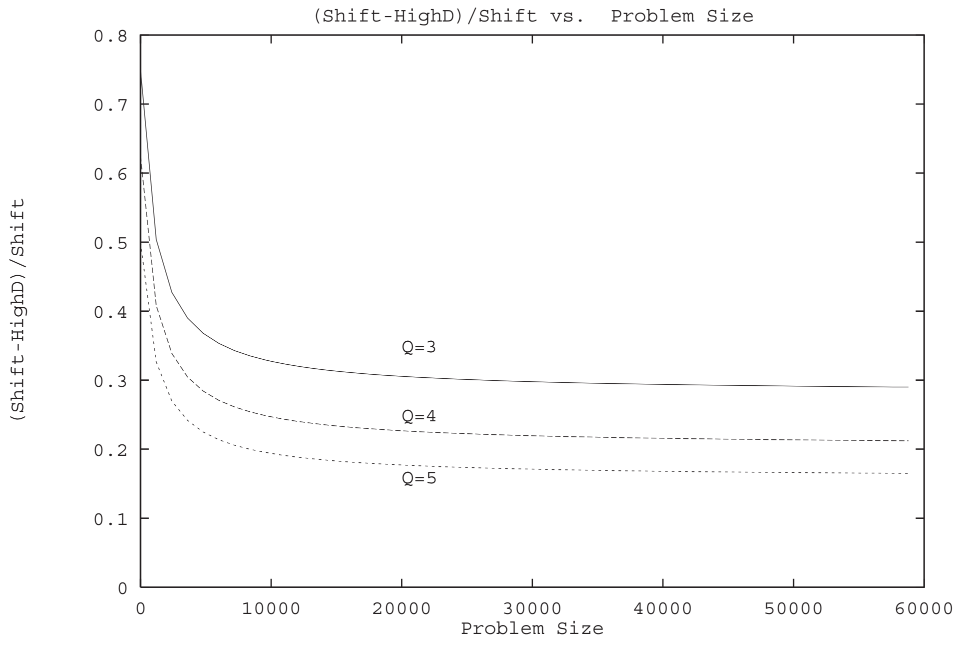

Figure 4.20: SDMV: Theoretical Performance Shift vs HighD on Setup I, D=8

The difference in execution time between $HighD$ and CH is:

$$\tau_{HighD} - \tau_{CH} \quad = \quad (Q-1)\left[\tau_{caf}\left(\frac{M}{Q}, Q\right) - max\left(0, \tau_{caf}\left(\frac{M}{Q}, Q\right) - \frac{2ND}{Q^2}\right)\right] - \tau_{parfor}(Q)$$

CH is preferable to $HighD$, both because it saves memory and because, once the overhead associated with `parfor` is negligible compared to the communication needed, the execution time is shorter.

Once the maximum possible communication hiding has been reached, the difference in execution time rises linearly with M, as it is $(Q-1)\tau_{caf}\left(\frac{M}{Q}\right)$. SDMV also grows linearly with M, thus the difference in execution time will be a constant fraction of the total time. This constant will decrease with Q, as demonstrated in Figure 4.21, which shows $HighD - CH$ as a fraction of $HighD$ on Setup I for several Q.

Experimental Performance Comparison

The algorithms described above were implemented in DLA. Figure 4.22 shows the measured $Shift - HighD$ as a fraction of $Shift$ on Setup I. Performance for $Q = 5$ is worse than expected. For both $Q = 3$ and $Q = 5$, the performance of $HighD$ appears to degrade with increasing M, rather than approaching a constant. This is most likely due to the non-scalable memory use in $HighD$.

Figure 4.23 shows the measured $HighD - CH$ as a fraction of $HighD$ using Setup I, for $Q = 4$,$D = 8$. The form of the results are as expected, but the constant approached as M increases is a factor of two higher than expected. In addition, the data is quite noisy.

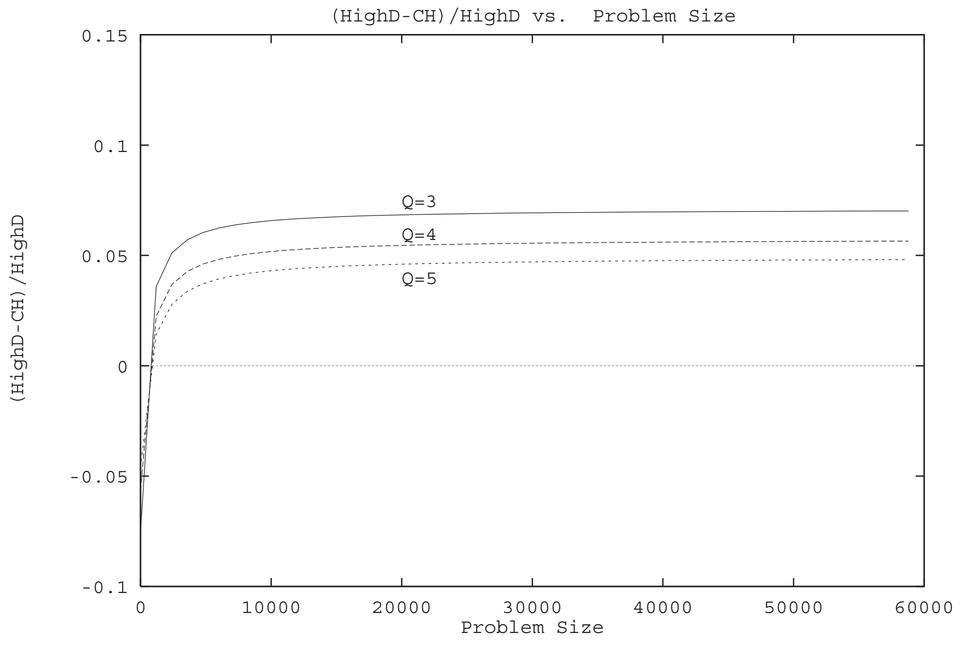

Figure 4.21: SDMV: Theoretical Performance HighD vs CH on Setup I, D=8

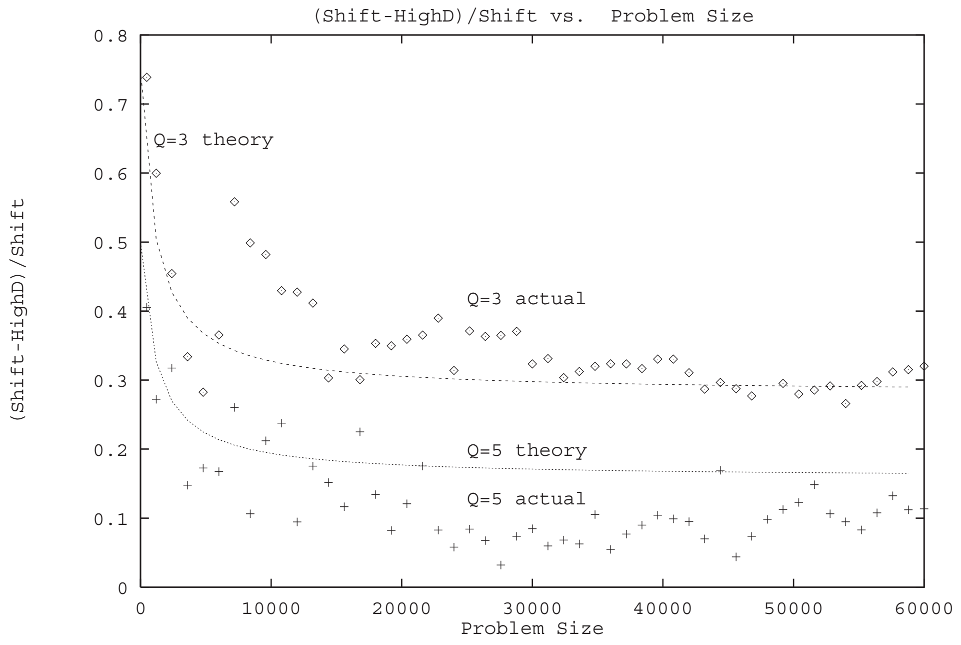

Figure 4.22: SDMV: Experimental Performance Shift vs HighD on Setup I

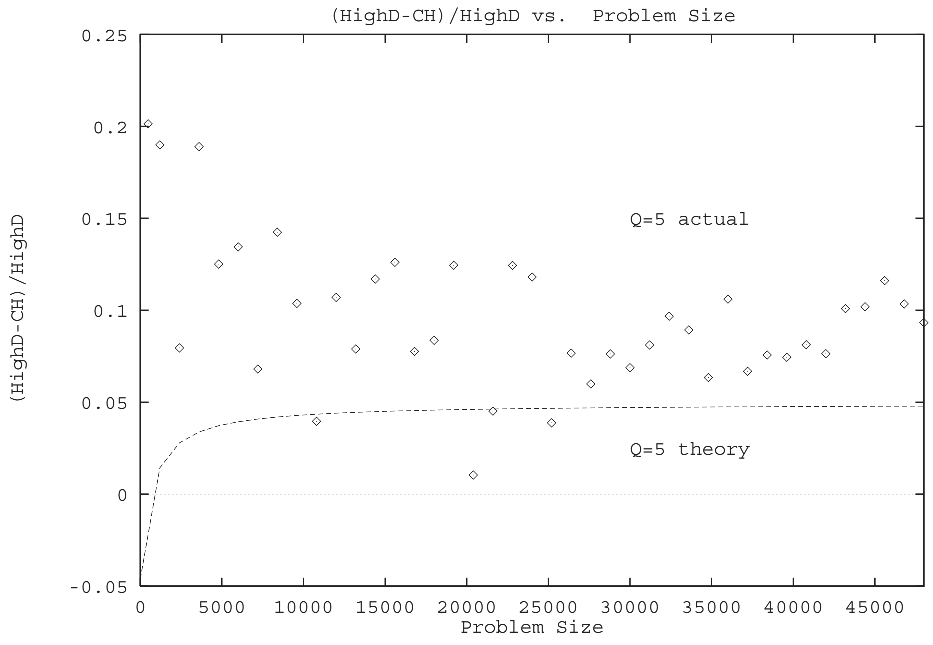

Figure 4.23: SDMV Experimental Performance: HighD vs CH on Setup I for Q=5,D=8

The constant performance improvement for large M is important, as SDMV is a linear computation. Since the required communication also grows linearly with M, performing SDMV in a distributed environment is only viable to enable computation for M too large for the memory of a single processor. Since the $HighD$ algorithm uses as much memory on each process as is used sequentially on a single processor, the $HighD$ algorithms is realistically unusable.

The CH algorithm is valuable, however. Figure 4.24 shows the measured $Shift - CH$ as a fraction of $Shift$ on Setup I for $Q = 5$, using problem sizes too large for $Q = 1$.

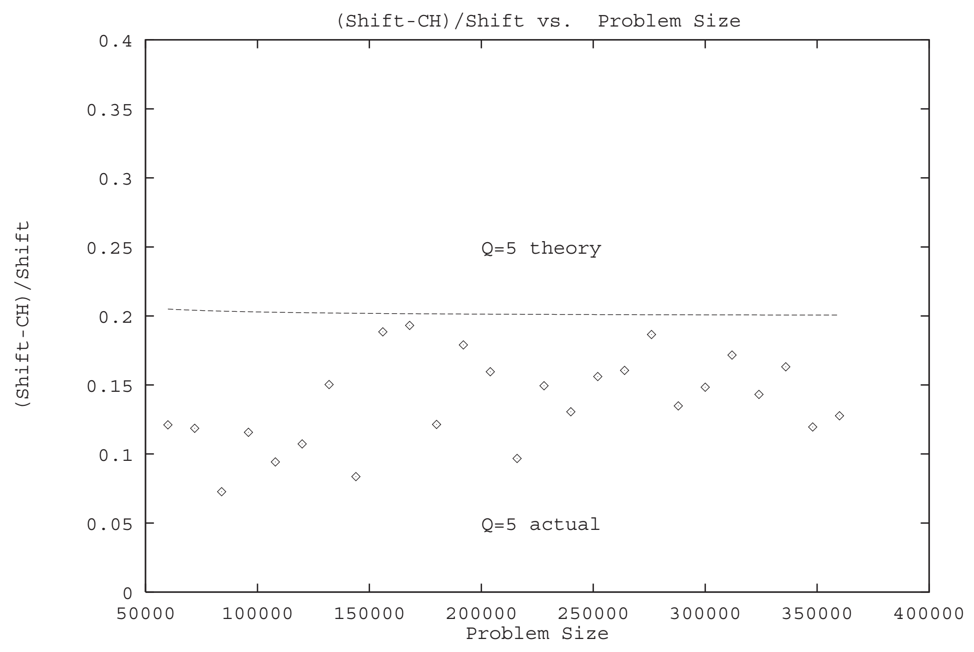

Figure 4.24: SDMV Experimental Performance: Shift vs CH on Setup I for Q=5,D=8

Chapter 5

Conclusion

The algorithmic modifications developed improve performance of basic linear algebra routines on NsW. The performance model predicts in what situations improvement will occur, and approximately what their magnitudes will be. Exact prediction of execution times would require a considerably more complex model.

Therefore, this thesis takes a step towards effectively utilizing NsW for concurrent computation: it provides a performance model for regular process grid topology, SPMD style distributed linear algebra programs on multithreaded, homogeneous, uniprocessor NsW. With this performance model, and the attendant suite, a user can measure the parameters of interest, and select the appropriate algorithm for a particular NW.

To fully exploit the diversity of NsW, restrictions on process topologies, programming style and heterogeneity need to be relaxed. Details of thread interleavings need to be better hidden from consideration of the higher level linear algebra programmer.

The algorithmic modifications developed will be used only if the performance gains are worth the complexity involved in achieving them. Added complexity in basic linear algebra routines reduces the complexity that can be dealt with at higher levels. NsW will be used for linear algebra only if their price/performance potential can be achieved without significant investment in dealing with the complexity caused by workstation diversity.

Bibliography

[1] Petter E. Bjorstad, W.M. Coughran, Jr. and Eric Grosse. *Parallel Domain Decomposition Applied to Coupled Transport Equations.* 7th Domain Decomposition Conference, Penn State. Oct. 93.

[2] Peter Carlin, Carl Kesselman, and K. Mani Chandy. *The Compositional C++ Langugage Definition.* CS-TR-92-02, California Institute of Technology, 1994.

[3] Peter Carlin and Tom Zavisca. *DLA: A Distributed Linear Algebra Library for CC++.* Internal Note

[4] Peter Carlin and Paul A.G. Sivilotti. *A Tutorial for CC++.* CS-TR-94-02, California Institute of Technology, 1994.

[5] Peter Carlin. *A CC++ Performance Suite.* Internal Note

[6] Vipin Kumar [et al.]. *Introduction to parallel computing : design and analysis of algorithms.* Benjamin/Cummings Publishing Co, 1994.

[7] Paul A.G. Sivilotti. *A Verified Integration of Imperative Parallel Programming Paradigms in an Object Oriented Language.* CS-TR-93-21, California Institute of Technology, 1993.

[8] C.A. Thekkath and H.M. Levy. *Limits to Low-Latency Communication on High Speed Networks.* ACM Transactions on Computer Systems, Vol 11. Number 2, 1993. pp 179-203.

[9] Eric F. Van de Velde. *Concurrent Scientific Computing.* Springer-Verlag, 1994.

[10] D.W. Walker. *Message Passing Interface.* Parallel Computing, Vol 20. Number 4, 1994.

[11] B.W. Abeysundara, and A.E. Kamal. *High-Speed Local Area Networks and their Performance - A Survey.* ACM Computing Surveys, Vol 23. Number 2, pp 221-264 1991.