

**LogP Analysis of a Parallel Algorithm
for the Modified Gram-Schmidt Process**

Problem Report

Submitted to the College of Arts and Sciences

of

West Virginia University

In Partial Fulfillment of the Requirements for

The Degree of Master of Science

by

James R. Bercik

1995

Acknowledgments

I thank Dr. Frances Van Scoy, Dr. Srinu Kankanahalli and Dr. George Trapp for their guidance, suggestions and support. Also, thanks to Warren McCague for providing me with a random number generator.

Abstract. The LogP model is used to develop a parallel algorithm to compute the Modified Gram-Schmidt process. The algorithm is analyzed and implemented on the Connection Machine (CM-5). The LogP model is then used to predict the running times of the algorithm for different problem sizes and which are compared to actual running times on the CM-5.

Introduction	1
The LogP Model	2
The Gram-Schmidt Process and MGS.....	3
Example of MGS.....	4
History of MGS.....	6
Applications of MGS	6
Related Work	7
A Serial Algorithm for MGS.....	8
A Parallel Algorithm for MGS	9
LogP Analysis of ParallelMGS.....	13
Without Vector Units	13
With Vector Units	14
Evaluation of LogP.....	15
An Alternative Analysis	17
Amdahl's Law	20
Architecture of the CM-5	22
Processing Nodes	23
Control Processors	24
Networks.....	25
Implementations on the CM-5	26
CM Fortran	26
C*	27
Global/Local CM Fortran	29
C/CMMD	31
Summary	35
Appendix A - Serial MGS in C	37
Appendix B - Serial MGS in FORTRAN 77	40
References.....	42
Approval of Examining Committee	43

1 Introduction

There have been numerous models for analyzing parallel algorithms including various extensions of the PRAM but none of these have accurately represented the typical real world situation of a small number of processors with a large problem size and high communication costs between processors while remaining abstract enough to be practical for any parallel computer architecture. The LogP model (Culler et al., 1993) is a promising new model that does consider these issues.

The goal is to develop, analyze and implement a parallel algorithm under the LogP model and then evaluate the model based on the following criteria:

- Is it simple enough to analyze algorithms?
- Do the projected running times of algorithms closely resemble the actual running times?
- Does the model aid in the development of efficient parallel algorithms?
- Are communication costs accurately modeled?

I will study the problem of calculating a set of orthonormal vectors using the Modified Gram-Schmidt process. The algorithm provided is similar to the algorithm given by Zapata et al. (1991) which is for MGS QR factorization on hypercube SIMD computers.

2 The LogP Model

The LogP model is a realistic yet practical model for developing algorithms for distributed memory parallel machines. It considers communication costs and the number of processors without making assumptions about network topology structure or programming method (Culler et al., 1993).

The key parameters of the LogP model are:

L: Latency in communicating a message from source to destination

o: overhead or the amount of time for a processor to send or receive a message

g: gap or minimum time between message transmissions/receptions

P: the number of processors

The time for a small message to be sent from processor 1 and received by processor 2 is

$L + 2o + g$.

When developing algorithms under the LogP model, the following methods should be followed:

- assume a large number of data elements per processing node
- do not exploit specific network topologies
- do not allow extremely large messages

3 The Gram-Schmidt Process and MGS

Definition. If $S = \{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m\}$ is a nonempty set of vectors, then the vector equation

$$a_1 \mathbf{u}_1 + a_2 \mathbf{u}_2 + \dots + a_m \mathbf{u}_m = \mathbf{0}$$

has at least one solution,

$$a_1 = 0, a_2 = 0, \dots, a_m = 0$$

If this is the only solution, then S is a *linearly independent* set. If there are other solutions, then S is a *linearly dependent* set.

Definition. A set of vectors in an inner product space is called an *orthogonal set* if all pairs of distinct vectors in the set are orthogonal (the angle between them is $\Pi/2$). An orthogonal set in which each vector has a norm of 1 is called an *orthonormal set*.

The Gram-Schmidt Process takes a set of M linearly independent vectors of size N , $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m\}$ where $M \leq N$, and calculates a set of M orthonormal vectors of size N , $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m\}$. For this study we will consider the Euclidean inner product, represented by

$$\langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i=1}^{i=n} a_i b_i$$

and the Euclidean length or norm of a vector, \mathbf{a} , as

$$\|\mathbf{a}\| = \langle \mathbf{a}, \mathbf{a} \rangle^{1/2}$$

There are several different variations of the Gram-Schmidt Process including Classical Gram-Schmidt (CGS), Modified Gram-Schmidt (MGS) and Modified Gram-Schmidt with pivoting (MGSP). I have chosen to implement MGS since it economizes storage and is generally more stable than CGS and also because MGSP only shows a small improvement while increasing the amount of work to be done (Rice, 1966).

MGS can be represented mathematically as follows:

for $i=1, \dots, M$

1.) $\mathbf{v}_i = \mathbf{u}_i / \|\mathbf{u}_i\|$

2.) $\mathbf{u}_k = \mathbf{u}_k - \langle \mathbf{u}_k, \mathbf{v}_i \rangle \mathbf{v}_i \quad k=i+1, \dots, M$

3.1 Example of MGS

Step 0

In this step we have the original vectors stored in the matrix U and the matrix V set to 0's

$$\mathbf{V} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 1 & 1 & 2 & 2 & 1 \\ 2 & 3 & 1 & 6 & 2 & 3 & 1 & 6 \end{bmatrix}$$

Step 1, i=1

The first row of V is calculated and all rows of U are updated.

$$\mathbf{V} = \begin{bmatrix} 1/2\sqrt{2} & 1/2\sqrt{2} & 1/2\sqrt{2} & 1/2\sqrt{2} & 1/2\sqrt{2} & 1/2\sqrt{2} & 1/2\sqrt{2} & 1/2\sqrt{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1/2 & 1/2 & 1/2 & -1/2 & -1/2 & 1/2 & 1/2 & -1/2 \\ -1 & 0 & -2 & 3 & -1 & 0 & -2 & 3 \end{bmatrix}$$

Step 2, i=2

The second row of V is calculated and rows 2 and 3 of U are updated.

$$\mathbf{V} = \begin{bmatrix} \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} \\ -\frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1/2 & 1/2 & 1/2 & -1/2 & -1/2 & 1/2 & 1/2 & -1/2 \\ -2 & 1 & -1 & 2 & -2 & 1 & -1 & 2 \end{bmatrix}$$

Step 3, i=3

The third and final row of V is calculated.

$$\mathbf{V} = \begin{bmatrix} \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} \\ -\frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} \\ -\frac{1}{\sqrt{5}} & \frac{1}{2\sqrt{5}} & -\frac{1}{2\sqrt{5}} & \frac{1}{\sqrt{5}} & -\frac{1}{\sqrt{5}} & \frac{1}{2\sqrt{5}} & -\frac{1}{2\sqrt{5}} & \frac{1}{\sqrt{5}} \end{bmatrix}$$

3.2 History of MGS

In his doctoral dissertation on integral equations in 1879, J.P. Gram, a Danish actuary (Longley, 1984), attempted to improve on the expansion of Fourier series, which he noted was similar to the least squares methods. Rather than using Gauss's method, he "treated the dependent variable as a continuous function of the independent variable and said that the use of point intervals was a special case" (Anton,Rorres, 1991).

In 1905, Erhard Schmidt, a German mathematician (Anton,Rorres,1991), also discussed the Least Squares problem in his inaugural address at Göttingen University. The speech was later reprinted in 1907 when he gave credit to Gram for what has come to be known as the Gram-Schmidt Process (Longley, 1984).

Beginning in the 1950's mathematicians became interested in the usefulness of the Gram-Schmidt Process because of computers. Rice (1966), described the Modified Gram-Schmidt and Longley (1984) demonstrated that the Classical Gram-Schmidt process and Modified Gram-Schmidt process are mathematically identical.

3.3 Applications of MGS

The Gram-Schmidt process (and therefore MGS) can be used in calculating Legendre polynomials, Chebyshev polynomials, curve fitting of empirical data, smoothing, and calculating least square methods and other functional equations (Longley, 1984).

4 Related Work

Culler et al. (1993) gives a detailed description of the LogP model as well as analysis of several algorithms, with sections on algorithm design, data placement and work assignment. Several variations of the PRAM Model are also discussed.

Rice (1966) provides analysis of serial versions of CGS, MGS and MGSP as well as error analysis for each.

Zapata et al. (1991) describes an algorithm for MGS QR factorization on Single Instruction Stream Multiple Data (SIMD) computers using a SIMD computational model with an unshared distributed memory but assumes the number of processing nodes to be 1, M, N or MN where M and N are as described in chapter 3 above. The algorithm presented here is not for QR factorization and will be free of any assumptions about the number of processing nodes.

5 A Serial Algorithm for MGS

The pseudo code for the serial algorithm to compute MGS is given below. See the Chapter 8 for actual implementations. The goal is to start with a set of M linearly independent vectors and compute a set of M orthonormal vectors. Procedure SerialMGS accepts as input an $M \times N$ array, U , with its rows being the vectors that we wish to normalize. U is altered in the process and the normalized vectors are returned in the $M \times N$ matrix V .

The procedure loops from 1 to M to calculate to calculate M orthonormal vectors. There are two instances where it loops from 1 to N to compute the norm of a vector. Another loop from $i+1$ computes the updated values of U .

```
procedure SerialMGS(U,V,M,N)
for i=1 to M do
{
  for j=1 to N do          /* calculate dotproduct */
  norm  $\leftarrow$   $V_{i,j} * V_{i,j}$ 
  norm  $\leftarrow$  SQRT(norm)
  for j=1 to N do
   $V_{i,j} \leftarrow U_{i,j} / \text{norm}$ 
  for k=i+1 to M do
  {
    for j=1 to N do          /* calculate dotproduct */
    dotprdct  $\leftarrow U_{k,j} * V_{k,j}$ 
    for j=1 to N do
     $U_{k,j} \leftarrow U_{k,j} - \text{dotprdct} * V_{i,j}$ 
  }
}
```

SerialMGS has a complexity of $O(M^2 * N)$.

6 A Parallel Algorithm for MGS

Procedure ParallelMGS accepts as input an $M \times N$ array U , with its rows being the vectors that we wish to normalize. U is altered in the process and the normalized vectors are returned in the $M \times N$ matrix V . N/P elements of each row of U and V are stored on each node. Assume that the nodes are numbered from 1 to P and that P and N are powers of 2. To calculate the norm of a vector, we calculate the inner product of the N/P sections of the vectors on each node in parallel and then perform a summing operation using a reduction method and store the result on the first node. The square root of this value is then taken to arrive at the norm.

procedure ParallelMGS(U,V,M,N)

$P = \text{numprocessors}$

Step 1 : **for** $i=1$ **to** M **do**

{

Step 1.0 $dp_{pid}=0$

Step 1.1 : **for** $j = 1$ **to** N/P **do in parallel** /* calculate dotproduct of node vectors*/

$dp_{pid} \leftarrow U_{i,j} * U_{i,j} + dp_{pid}$

Step 1.2 : $norm_1 \leftarrow \text{sum of } dp_j \text{ for } j = 1..P$

Step 1.3 : $norm_1 \leftarrow (norm_1)^{1/2}$

Step 1.4 : Broadcast $norm_1$ to all nodes

Step 1.5 : **for** $j = 1$ **to** N/P **do in parallel**

$V_{i,j} \leftarrow U_{i,j}/norm_j$

Step 1.6 : **for** $k=i+1$ **to** M **do**

{

Step 1.6.0 $dp_{pid}=0$

Step 1.6.1 : **for** $j = 1$ **to** N/P **do in parallel** /* calculate dotproduct of node vectors*/

$dp_{pid} \leftarrow U_{k,j} * V_{i,j} + dp_{pid}$

Step 1.6.2 : $dotprdct_1 \leftarrow \text{sum of } dp_j \text{ for } j = 1..P$

Step 1.6.3 : Broadcast $dotprdct_1$ to all nodes

Step 1.6.4 : **for** $j = 1$ **to** N/P **do in parallel**

$U_{k,j} \leftarrow U_{k,j} - dotprdct_{pid} * V_{i,j}$

}

}

6.2 An Example of ParallelMGS

In the following example, the same vectors from example 3.1 are used to demonstrate how the data would be laid out and to show the communication patterns of ParallelMGS. Each node contains 2 elements from each vector so node 1 contains elements 1 and 2, node 2 contains elements 3 and 4, node 3 contains elements 5 and 6, and node 4 contains elements 7 and 8. Vector 0 is on the bottom and vector 2 is on the top.

Step 1: Initial Layout

In the initialization, all V vectors are set to 0 and the initial U vectors are input. The norm and dotproduct are set to 0 on each node.

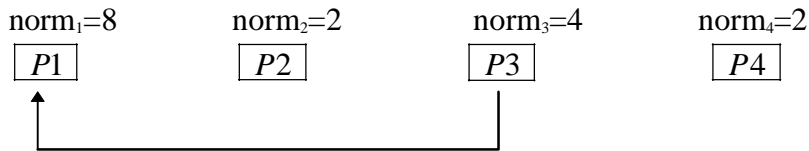
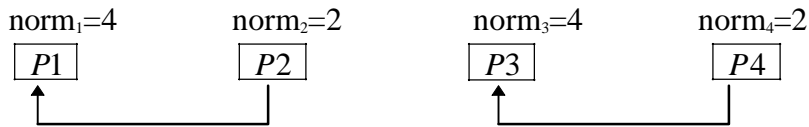
	norm ₁ =0 dp ₁ =0	norm ₂ =0 dp ₂ =0	norm ₃ =0 dp ₃ =0	norm ₄ =0 dp ₄ =0
V	+0.00 +0.00 +0.00 +0.00 +0.00 +0.00	+0.00 +0.00 +0.00 +0.00 +0.00 +0.00	+0.00 +0.00 +0.00 +0.00 +0.00 +0.00	+0.00 +0.00 +0.00 +0.00 +0.00 +0.00
U	+2.00 +3.00 +1.00 +2.00 +1.00 +1.00	+1.00 +6.00 +2.00 +1.00 +1.00 +1.00	+2.00 +3.00 +1.00 +2.00 +1.00 +1.00	+1.00 +6.00 +2.00 +1.00 +1.00 +1.00
	P1	P2	P3	P4

Step 2: Calculate DotProduct and Norm

To calculate the norm of row 1 of U, we square and then sum the elements of U on each node, storing the result into dp_i for i=1,P.

dp ₁ =2	dp ₂ =2	dp ₃ =2	dp ₄ =2
P1	P2	P3	P4

We now perform a summing operation using a reduction method on the dp_i values, storing the final result into $norm_1$. This is the inner product of row 1 of U with itself.

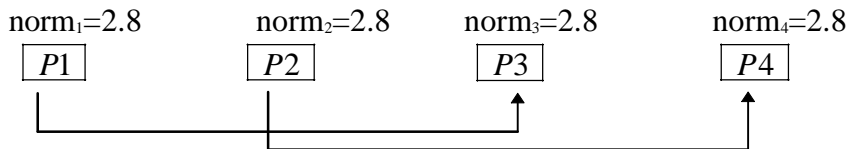
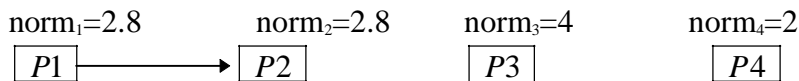


Taking the square root of $norm_1$ leaves the norm of row 1 of U in $norm_1$.



Step 3: Broadcast Norm to all Nodes

The norm value on node $P1$ is broadcast to the three other nodes in two steps.



Step 4: Calculate First Row of V

Each element of row 0 of V is set equal to the corresponding element of in row 0 of U divided by the norm. All rows of U are updated. Note the values shown are rounded to nearest hundredth.

	norm ₁ =2.8	norm ₂ =2.8	norm ₃ =2.8	norm ₄ =2.8																								
V	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	+0.00	+0.00	+0.00	+0.00	+0.35	+0.35	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	+0.00	+0.00	+0.00	+0.00	+0.35	+0.35	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	+0.00	+0.00	+0.00	+0.00	+0.35	+0.35	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	+0.00	+0.00	+0.00	+0.00	+0.35	+0.35
+0.00	+0.00																											
+0.00	+0.00																											
+0.35	+0.35																											
+0.00	+0.00																											
+0.00	+0.00																											
+0.35	+0.35																											
+0.00	+0.00																											
+0.00	+0.00																											
+0.35	+0.35																											
+0.00	+0.00																											
+0.00	+0.00																											
+0.35	+0.35																											
U	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>-1.00</td><td>+0.00</td></tr><tr><td>-0.50</td><td>+0.50</td></tr><tr><td>+1.00</td><td>+1.00</td></tr></table>	-1.00	+0.00	-0.50	+0.50	+1.00	+1.00	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>-2.00</td><td>+3.00</td></tr><tr><td>+0.50</td><td>-0.50</td></tr><tr><td>+1.00</td><td>+1.00</td></tr></table>	-2.00	+3.00	+0.50	-0.50	+1.00	+1.00	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>-1.00</td><td>+0.00</td></tr><tr><td>-0.50</td><td>+0.50</td></tr><tr><td>+1.00</td><td>+1.00</td></tr></table>	-1.00	+0.00	-0.50	+0.50	+1.00	+1.00	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>-2.00</td><td>+3.00</td></tr><tr><td>+0.50</td><td>-0.50</td></tr><tr><td>+1.00</td><td>+1.00</td></tr></table>	-2.00	+3.00	+0.50	-0.50	+1.00	+1.00
-1.00	+0.00																											
-0.50	+0.50																											
+1.00	+1.00																											
-2.00	+3.00																											
+0.50	-0.50																											
+1.00	+1.00																											
-1.00	+0.00																											
-0.50	+0.50																											
+1.00	+1.00																											
-2.00	+3.00																											
+0.50	-0.50																											
+1.00	+1.00																											
	P1	P2	P3	P4																								

Step 5: Calculate Second Row of V

The norm is recalculated and broadcast. Each element of row 1 of V is set equal to the corresponding element of in row 1 of U divided by the norm. Rows 1 and 2 of U are updated.

	norm ₁ =1.4	norm ₂ =1.4	norm ₃ =1.4	norm ₄ =1.4																								
V	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>-0.35</td><td>+0.35</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	+0.00	+0.00	-0.35	+0.35	+0.35	+0.35	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>+0.35</td><td>-0.35</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	+0.00	+0.00	+0.35	-0.35	+0.35	+0.35	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>-0.35</td><td>+0.35</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	+0.00	+0.00	-0.35	+0.35	+0.35	+0.35	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+0.00</td><td>+0.00</td></tr><tr><td>+0.35</td><td>-0.35</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	+0.00	+0.00	+0.35	-0.35	+0.35	+0.35
+0.00	+0.00																											
-0.35	+0.35																											
+0.35	+0.35																											
+0.00	+0.00																											
+0.35	-0.35																											
+0.35	+0.35																											
+0.00	+0.00																											
-0.35	+0.35																											
+0.35	+0.35																											
+0.00	+0.00																											
+0.35	-0.35																											
+0.35	+0.35																											
U	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+1.00</td><td>+10.0</td></tr><tr><td>-0.50</td><td>+0.50</td></tr><tr><td>-2.0</td><td>+1.0</td></tr></table>	+1.00	+10.0	-0.50	+0.50	-2.0	+1.0	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+1.00</td><td>+1.00</td></tr><tr><td>+0.50</td><td>-0.50</td></tr><tr><td>-1.00</td><td>+2.00</td></tr></table>	+1.00	+1.00	+0.50	-0.50	-1.00	+2.00	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+1.00</td><td>+1.00</td></tr><tr><td>-0.50</td><td>+0.50</td></tr><tr><td>-2.00</td><td>+1.0</td></tr></table>	+1.00	+1.00	-0.50	+0.50	-2.00	+1.0	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>+1.00</td><td>+1.00</td></tr><tr><td>+0.50</td><td>-0.50</td></tr><tr><td>-1.00</td><td>+2.00</td></tr></table>	+1.00	+1.00	+0.50	-0.50	-1.00	+2.00
+1.00	+10.0																											
-0.50	+0.50																											
-2.0	+1.0																											
+1.00	+1.00																											
+0.50	-0.50																											
-1.00	+2.00																											
+1.00	+1.00																											
-0.50	+0.50																											
-2.00	+1.0																											
+1.00	+1.00																											
+0.50	-0.50																											
-1.00	+2.00																											
	P1	P2	P3	P4																								

Step 6: Calculate Third Row of V

The norm is recalculated and broadcast. Each element of row 2 of V is set equal to the corresponding element of in row 2 of U divided by the norm. Row 2 of U is updated.

	norm ₁ =4.4	norm ₂ =4.4	norm ₃ =4.4	norm ₄ =4.4																								
V	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>-0.45</td><td>+0.22</td></tr><tr><td>-0.35</td><td>+0.35</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	-0.45	+0.22	-0.35	+0.35	+0.35	+0.35	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>-0.22</td><td>+0.45</td></tr><tr><td>+0.35</td><td>-0.35</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	-0.22	+0.45	+0.35	-0.35	+0.35	+0.35	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>-0.45</td><td>+0.22</td></tr><tr><td>-0.35</td><td>+0.35</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	-0.45	+0.22	-0.35	+0.35	+0.35	+0.35	<table border="1" style="border-collapse: collapse; text-align: left;"><tr><td>-0.22</td><td>+0.45</td></tr><tr><td>+0.35</td><td>-0.35</td></tr><tr><td>+0.35</td><td>+0.35</td></tr></table>	-0.22	+0.45	+0.35	-0.35	+0.35	+0.35
-0.45	+0.22																											
-0.35	+0.35																											
+0.35	+0.35																											
-0.22	+0.45																											
+0.35	-0.35																											
+0.35	+0.35																											
-0.45	+0.22																											
-0.35	+0.35																											
+0.35	+0.35																											
-0.22	+0.45																											
+0.35	-0.35																											
+0.35	+0.35																											
	P1	P2	P3	P4																								

7 LogP Analysis of ParallelMGS

In this chapter, the ParallelMGS algorithm will be analyzed using the LogP model. An asymptotic upper bound is arrived at for architectures with and without vector units and then an alternative analysis is done. In Chapter 10, the upper bounds are compared with actual running times on the CM-5.

7.1 Without Vector Units

With the exception of a square root operation, steps 1.1 - 1.4 and 1.6.2.1 - 1.6.3 are identical. The basic algorithm here is the inner product of two vectors. Steps 1.1 and 1.6.1 require N/P parallel steps and no communication between the nodes.

Steps 1.2 and 1.6.2 are performing the sum of P elements on P processors using a reduction method. This requires $\log_2 P$ steps with a maximum of $\log_2 P$ additions to be performed at any one step and a maximum of $\log_2 P$ messages to be sent at any step. The distance that the messages must travel during this operation increase exponentially with each step. At each step, a maximum of $L+2o+g$ cycles are used to send messages and we assume that the additions can be done in unit time. This results in a complexity of $k_1 * ((L+2o+g) * \log_2 P)$

Assume that the SQRT function can be evaluated in constant time

Steps 1.4 and 1.6.3 follow a similar communication pattern as 1.2 and 1.6.2 only no adding is being done. The complexity of this step is $k_2 * (L+ 2o + g) * \log_2 P$.

The overall complexity for calculating the norm or the dot product is then

$$k_3 * (L + 20 + g) * \log_2 P).$$

Step 1.5 we assume that division can be done in constant time and since no communication is required this step takes N/P cycles.

Step 1.6.4 similar to step 1.5, this step requires N/P cycles

The overall complexity of ParallelMGS is then

$$M^2 * \text{MAX}[k_3 * (L + 20 + g) * \log_2 P, k_4 * N/P].$$

In most cases $N \gg \log_2 P$ and our complexity will be $O(M^2 * N/P)$.

7.2 With Vector Units

Some machines, including the CM-5, are equipped with vector units (VUs) on each processing node. If we allow D to be the depth of the vector unit, Steps 1.5 and 1.6.4 will require $N/(P*D)$ cycles. This will result in a complexity of $O(M^2 * N/(P*D))$.

8 Evaluation of LogP

Included below are several graphs that plot the actual running time of the CM FORTRAN implementation of ParallelMGS versus the estimated LogP runtime. Thinking Machines Corporation claims that a CM-5 with 33 Mhz SPARC RISC chips and vector units has a peak performance of 160 Mflops per processing node. Culler et al. (1993) note that the linpack benchmarking of the CM-5 showed a performance of 3.2 Mflops per processing node. For the LogP times, the asymptotic bound, $O(M^2*N/P)$, has been divided by the linpack benchmark, L. The resulting formula is $M^2*N/(P*L)$. Note that since we are using actual Mflops for the processing nodes in a CM-5, the factor for the vector units becomes irrelevant.

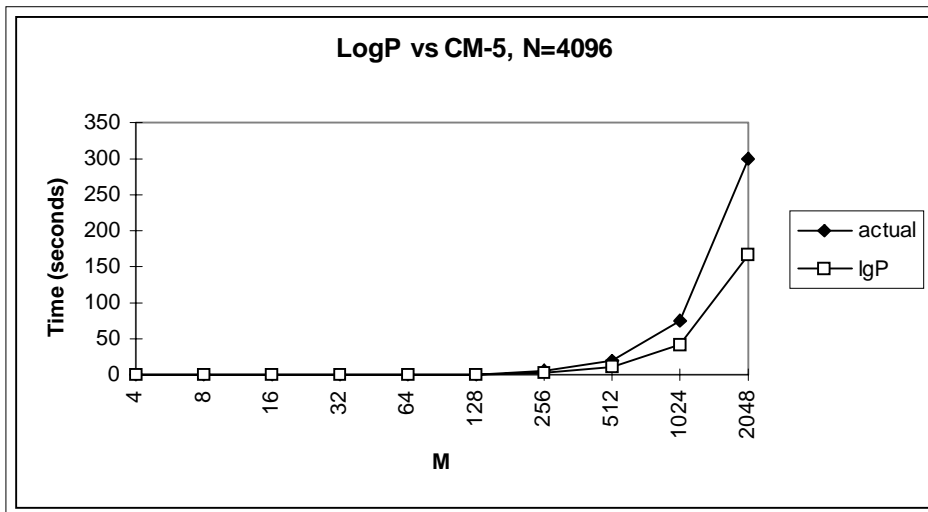


Figure 8.1 LogP vs CM-5, N=4096

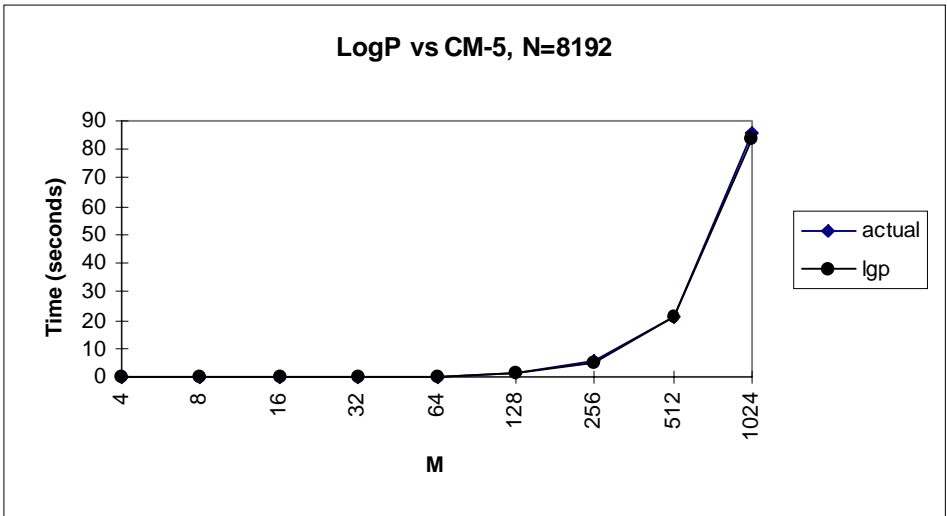


Figure 8.2 LogP vs CM-5, N=8192

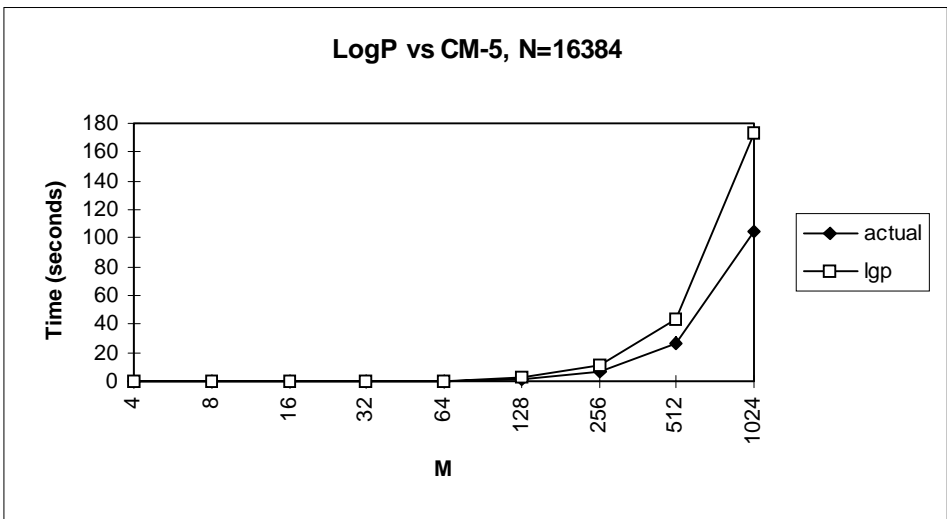


Figure 8.3 LogP vs CM-5, N=16384

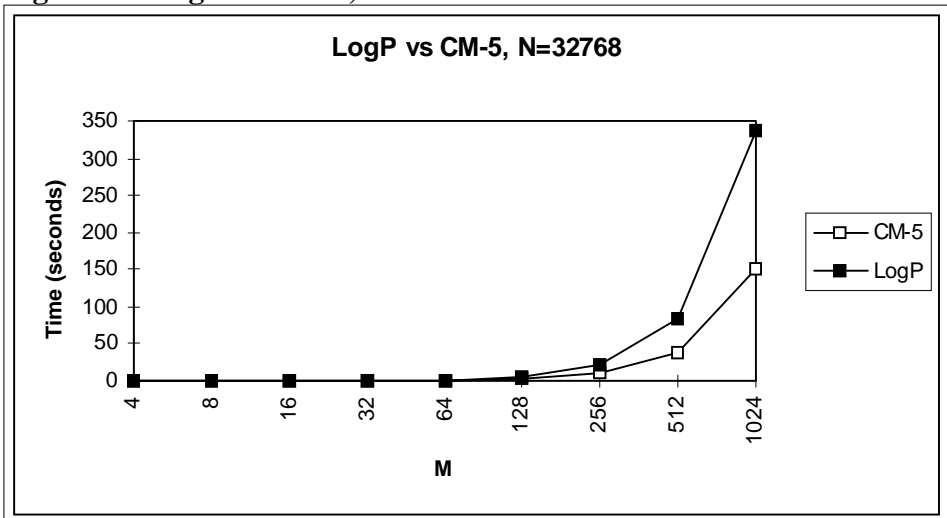


Figure 8.4 LogP vs CM-5, N=32768

9 An Alternative Analysis

In the above LogP analysis, we arrived at an asymptotic upper bound of $O(M^2 \cdot N/P)$. Figures 8.1 through 8.4 show evidence that when $N \gg M$, we have indeed arrived at an upper bound. This is not always the case, however. When $N=M$ the LogP estimates look more like a lower bound than an upper bound (see Figure 9.1). This could be caused by the fact that when $M \geq N$ the ratio of communications to computations increases greatly because the parts of the code that were parallelized were the loops of size N and also, the communication penalty “fell out” during the analysis.

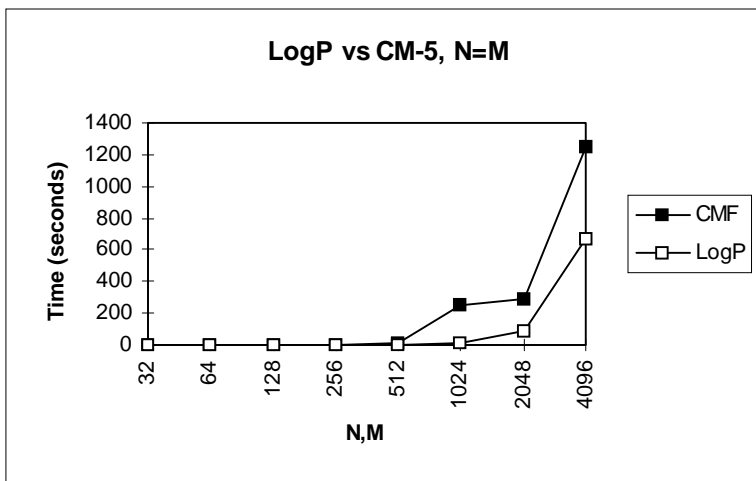


Figure 9.1 LogP vs CM-5, N=M

It appears that, when using the LogP model for analyzing parallel algorithms, separate analysis of the communication and computation would allow for better comparisons of different approaches to the same problem because both computation and communication costs could be compared. This is because what we really do when analyzing an algorithm is estimate the number of floating point operations. However, the number of messages is a completely different thing and could be measured in different units.

If the communication and computation times are analyzed separately we have a new bound of $O(M^2 \cdot N/P) + O(M^2 \log_2 P)$. To arrive at an estimated running time using the new bound, the average number of nodes that must be visited for each message, the network channel width, the size of the message and the number of cycles spent sending and receiving by a node are required. The number of leaves in a true 4-ary fat tree must be 4^i , where i is an integer. In our case, we have 32 leaves. The tree could have a root with two trees with 16 leaves as children or it could be that the nodes before the leaves have only two children. In the first case, the average distance that a message must travel is $H_1=4.6875$ and in the later case the average distance traveled a distance of $H_2=5.3125$.

To calculate the time for 1 message we will slightly modify the formula given by Culler et al. (1993).

$$T(M,H) = (T_{\text{snd}} + \lceil M/w \rceil + H \cdot r + T_{\text{rcv}}) \cdot \text{cycles/ns}$$

The values for the above parameters are as follows:

Cycle/ns	w	Tsnd	Trcv	r cycles	H1	H2
25	4	1300	1300	8	4.6875	5.3125

Using these numbers we arrive at the time to send one 64 bit message as

$$T_1(64,32) = ((1300 + 64/4 + 4.6875 \cdot 8 + 1300) \cdot 25) \cdot 10^{-9} \text{ sec/ns} = 0.0000663 \text{ sec/message}$$

$$T_2(64,32) = ((1300 + 64/4 + 5.3125 \cdot 8 + 1300) \cdot 25) \cdot 10^{-9} \text{ sec/ns} = 0.0000665 \text{ sec/message}$$

With these numbers our graph for $N=M$ changes drastically and yet we still have an acceptable upper bound for the cases when $M < N$. See figure 9.2 and 9.3

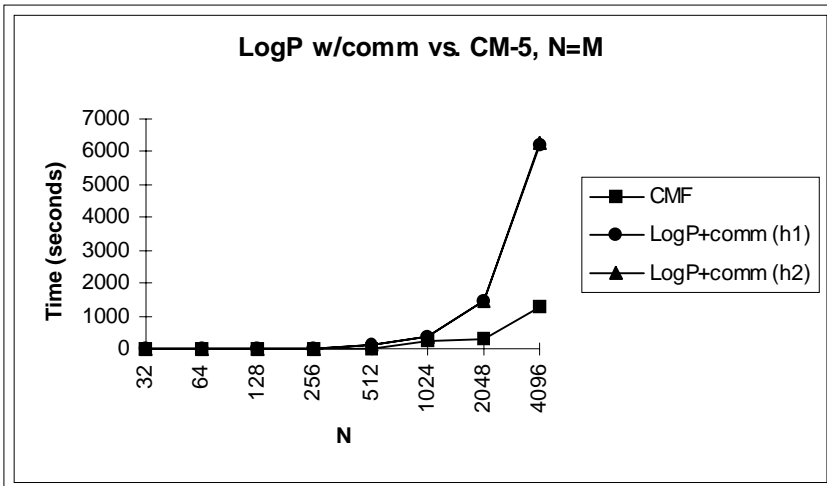


Figure 9.2 LogP w/ comm. vs Cm5

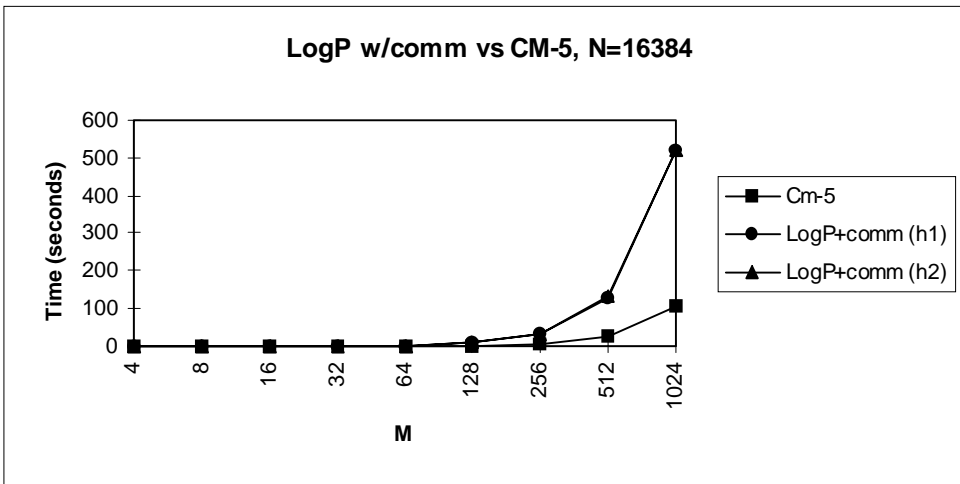


Figure 9.3 LogP w/comm vs CM-5, N<M

10 Amdahl's Law

Amdahl's Law gives us an equation for the maximum speed up, S , of a parallel program where P is the ideal speedup (the number of processors) and F is the fraction of the sequential program that can be parallelized. The equation is :

$$S = 1 / ((1-F) + (F/P))$$

According to this equation, to achieve between 16 and 32 times speed up on a 32 processing node system, at least 97% of the serial code must be parallelized. See Figure 10.1.

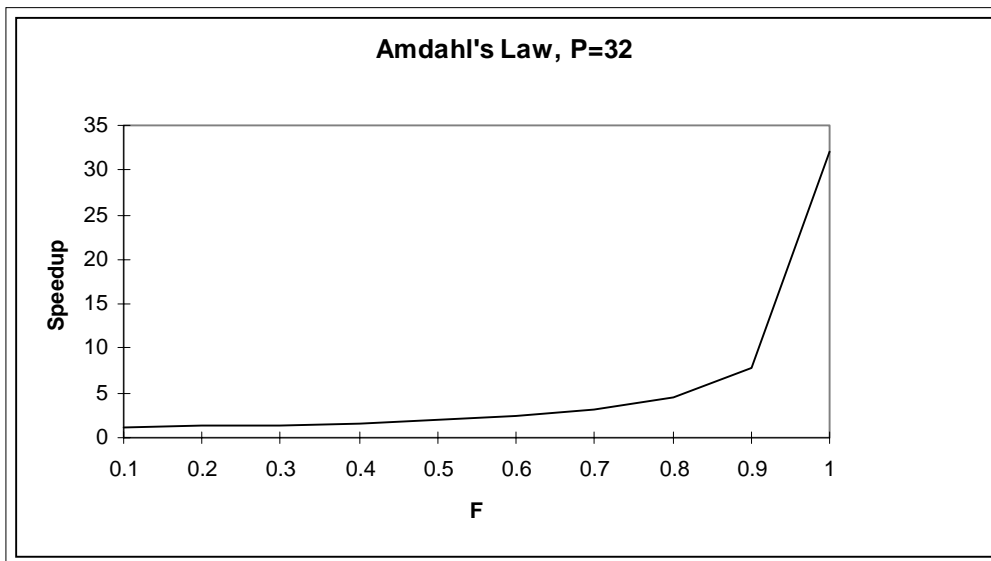


Figure 10.1 Amdahl's Law

In our case, $P=32$ and of our 6 loops, 4 were re-written in parallel, so, $F = 4/6 = 2/3$.

Using these values for F and P we arrive at the maximum speedup for ParallelMGS as

$$S = 1 / ((1-2/3) + ((2/3) / 32)) = 2.82$$

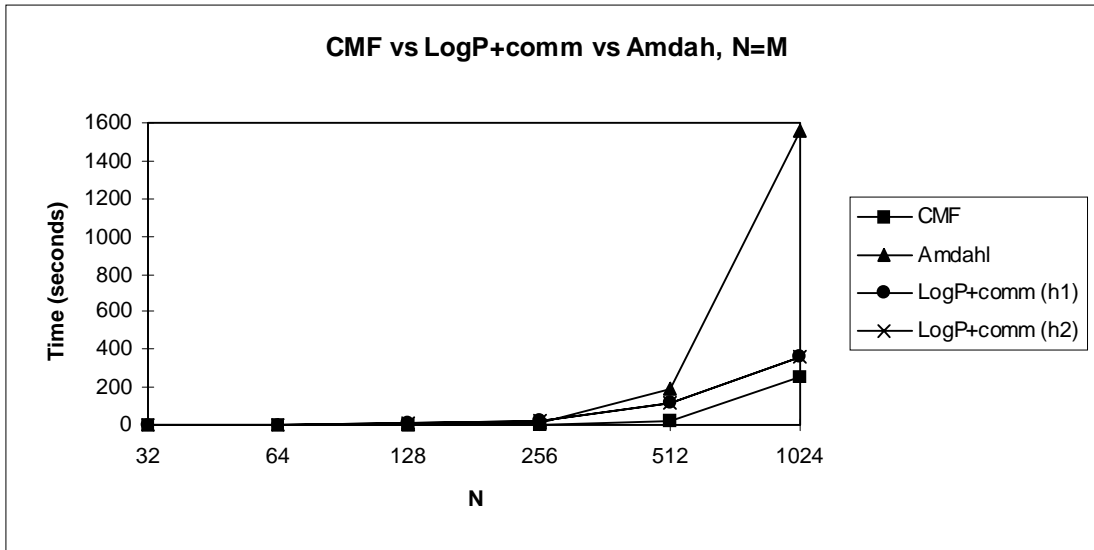


Figure 10.2 Amdahl's Law

Figure 10.2 shows a comparison of CM FORTRAN, LogP+comm and Amdahl's Law. The numbers for Amdahl's law were obtained by dividing the FORTRAN 77 time by the maximum speed up $S=2.8$. In the graph, the Amdahl run times are unexpectedly much slower than the CM FORTRAN and LogP+comm. This suggests that LogP+comm is a better method of estimating the speedup of a parallel algorithm than Amdahl's Law. It should be noted that when mapping Amdahl's Law to real machines, the results will vary with compilers and architectures. For example, The FORTRAN 77 times were obtained on a Sparc Station 10 Series Model 52 with 2 RISC 51 Mhz processors. Had we used a parallel machine consisting of 32 Sparc Station 10's, the resulting Amdahl time might better match the parallel time.

11 Architecture of the CM-5

The CM-5 is a parallel computer that supports both SIMD and Multiple Instruction Stream Multiple Data (MIMD) programming styles. The CM-5 has between 16 and 1024 processing nodes, at least one control processor, distributed memory and three high-bandwidth networks in place of the traditional bus. The processing nodes can be grouped into partitions which are managed by a control processor known as a Partition Manager. Figure 11.1 shows the organization of the CM-5.

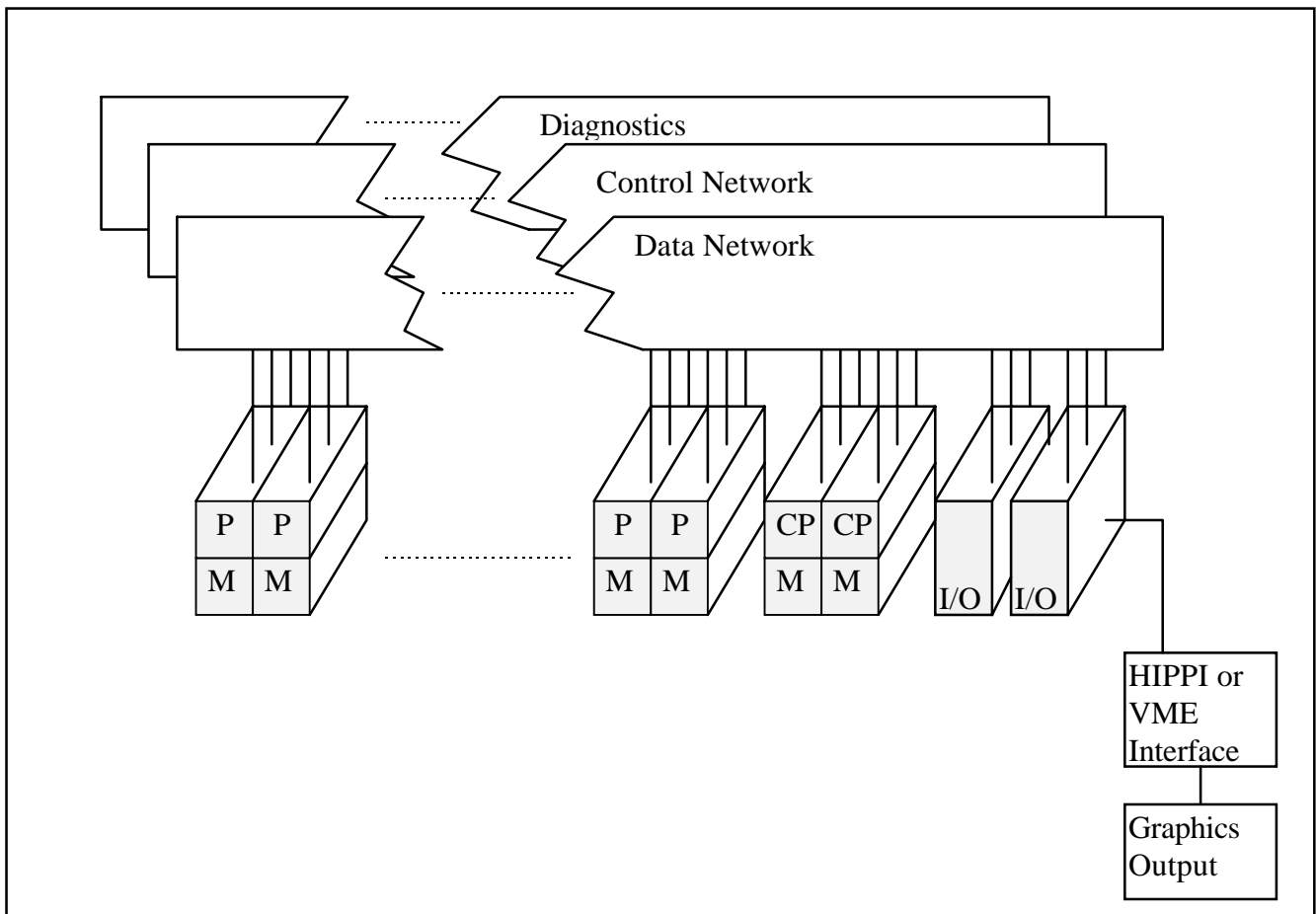


Figure 11.1 CM-5 Organization

11.1 Processing Nodes

Each CM-5 processing node consists of a RISC microprocessor, four high performance vector units each having a 72-bit path to an 8 or 32Mbyte of DRAM memory bank with a peak memory bandwidth of 160 Mbytes/sec per VU. Both the processor and the VUs are connected to a 64 bit bus. All connections to the rest of the system pass through the network interface (Thinking Machines Corporation, 1993a). Figure 11-2 illustrates a processing node of the CM-5.

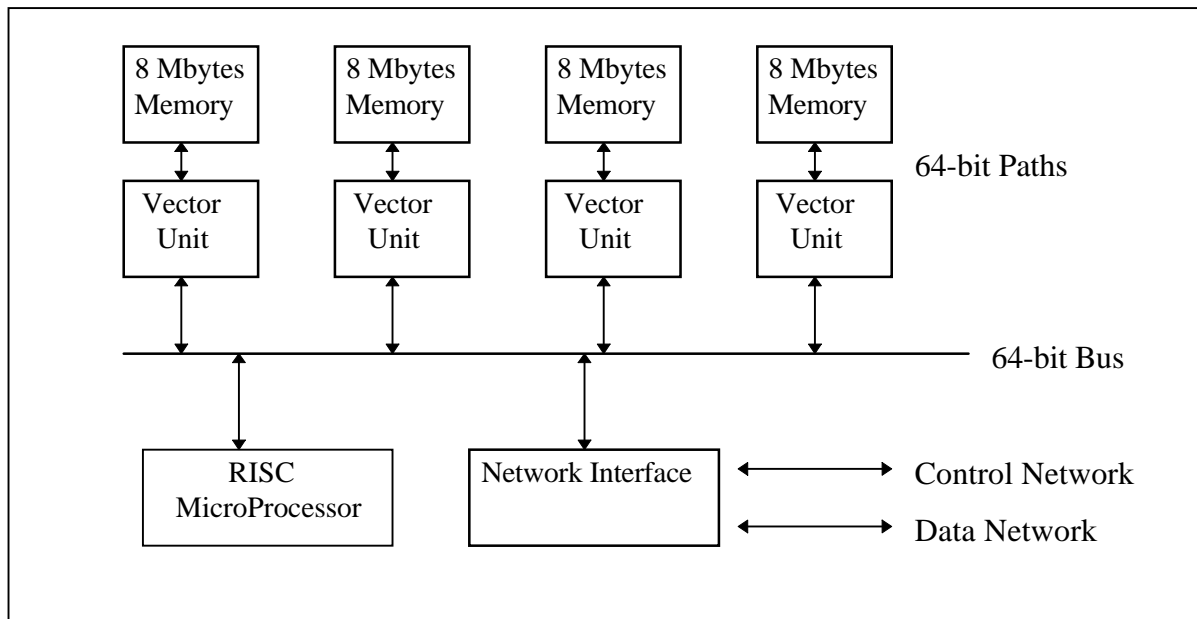


Figure 11-2: CM-5 Processing Node

The microprocessor issues instructions to the VUs either one at a time or by broadcasting the same instruction to all four VUs. Each VU provides up to 40 Mflops of 64-bit performance for a total of 160 Mflops peak performance for each processing node.

11.2 Control Processors

A control processor consists of a RISC microprocessor, memory, I/O devices and a CM-5 Network Interface which provides access to the Control and Data Networks. Other than the network interface, the control processor is a standard workstation. Each control processor runs the CMOST operating system, which is an extension of UNIX with additional features for the parallel processing CM-5 (Thinking Machines Corporation, 1993a). Figure 11-3 illustrates a control processor of the CM-5.

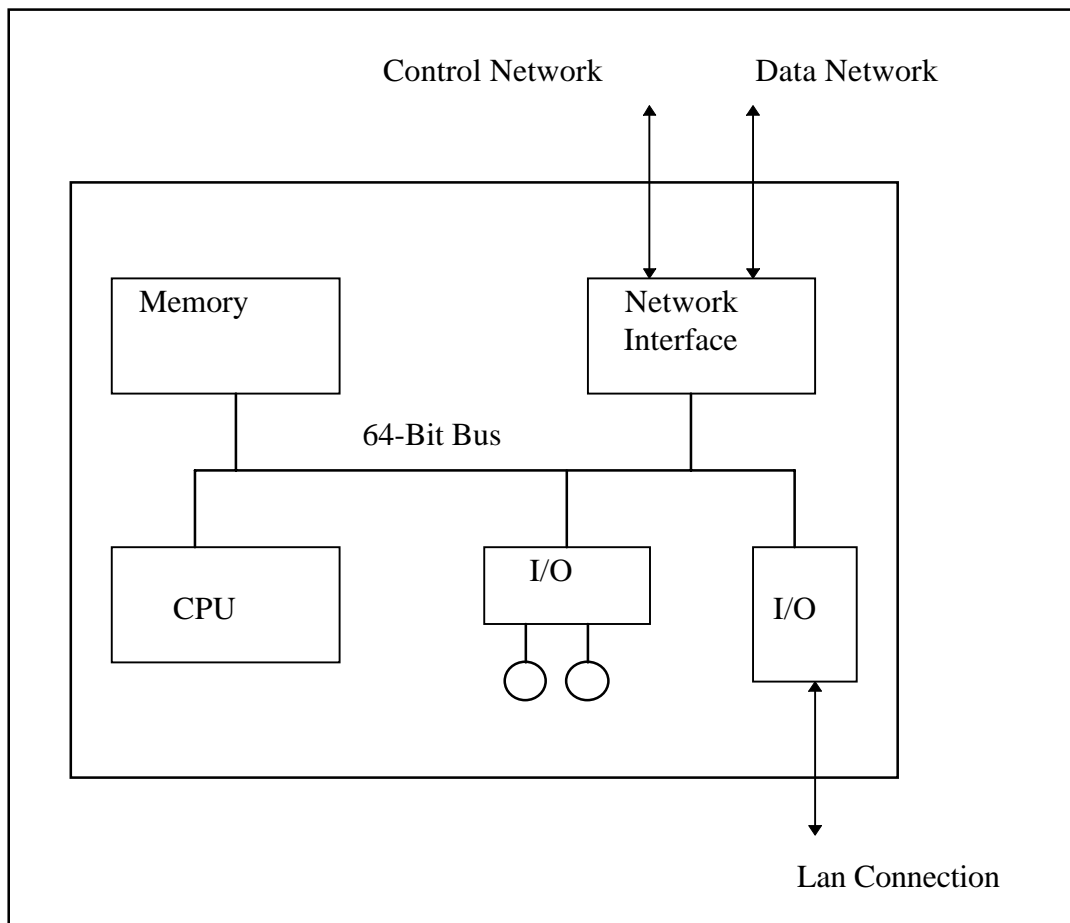


Figure 11-3 CM-5 Control Processor

11.3 Networks

Every processor is connected to two communication networks that allow for high bandwidth communication over the optimal route. The control network can synchronize the nodes, perform broadcasting and parallel prefix operations. The data network is a 4-ary fat tree where each node has 4 children. The height of a tree is $\log_4(\# \text{ of addresses spanned})$.

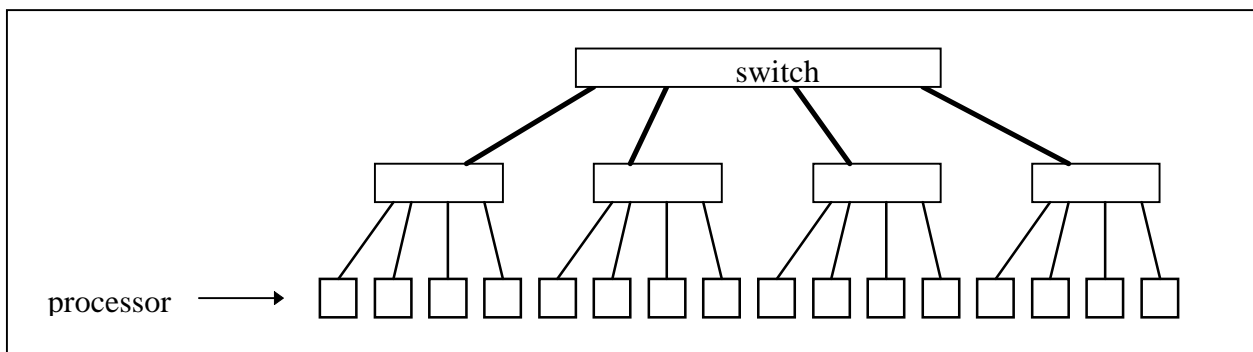


Figure 11-4 A 16 node CM-5 4-ary fat tree network

A 16 node CM-5 fat tree network is shown in Figure 11-4. Each leaf represents a network interface on a processor. The nodes in the tree are switches implemented with VLSI chips. Each node has two or four parents and four children. The closer to the root a node is, the more switches it will have.

12 Implementations on the CM-5

This section contains descriptions of implementations of parallelMGS on a 32 node CM-5 with vector units. The ParallelMGS algorithm is implemented using the CM FORTRAN, Global/Local Cm FORTRAN, C*, and C/CMMD programming languages. The code for Serial MGS implemented in FORTRAN 77 and C is included in the appendices.

12.1 CM FORTRAN

CM FORTRAN is a version of FORTRAN 77 which includes the array processing extensions of the ANSI and ISO standard FORTRAN 90 (Thinking Machines Corporation, 1993b). This is a data parallel or SIMD language, i.e., each parallel statement is executed on all of the nodes at the same time.

```
C-----
C File      : mgs.fcm
C Purpose   : CM Fortran implementation of Modified Gram-Schmidt
C           : Process
C Author    : Jim Bercik
C Date      : 8-23-94
C-----
      SUBROUTINE MGS(M,N,U,V)
      IMPLICIT NONE
      INTEGER M,N
      REAL U(M,N),V(M,N),NORM_SQRD
CMF$      LAYOUT U(:SERIAL,:NEWS),V(:SERIAL,:NEWS)
      INTEGER I,K
      DO 10 I=1,M
      NORM_SQRD=DOTPRODUCT(U(I,:),U(I,:))
      IF (NORM_SQRD .EQ. 0) THEN
          PRINT*, 'VECTORS ARE LINEARLY DEPENDENT... EXITING'
          GOTO 60
      ENDIF
      V(I,:)=U(I,:)/SQRT(NORM_SQRD)
      DO 20 K=I,M
          U(K,:)=U(K,:)-(DOTPRODUCT(U(K,:),V(I,:))*V(I,:))
      20 CONTINUE
      10 CONTINUE
      60 RETURN
      END
```

12.2 C*

C* is a Data parallel dialect of C for the CM-5. Each parallel statement is executed on all of the nodes at the same time. C* is based on the ANSI C language and adds a small set of extensions for parallel programming (Thinking Machines Corporation, 1993b).

```
/* Author : Jim Bercik
   Date   : 10-22-94
   File   : mgs.cs
   Purpose: C* implementation of the ParallelMGS
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cscmm.h>
#include <cstimer.h>
#include <cm/cmssl.h>
#include <cm/cmssl_math.h>

/* declaration */
void ParallelMGS(double:current *, double:current *,
                int *, int *);

void main()
{
    int n,i,m, rank,ier,num_tests,ctr;
    shape [][]myshape;
    FILE *fp;
    ier = 0;

    /* printf("What is m : ");
       scanf("%d%c", &m);
       printf("What is n : ");
       scanf("%d%c", &n);
    */
    printf("CSTAR MGS \n\n");
    rank = 2;
    if ((fp = fopen("mgs.dat", "r"))==NULL)
    {
        printf("CANNOT OPEN FILE");
        exit(1);
    }
    fscanf(fp, "%d", &num_tests);
    fprintf(stderr, "Num Tests: %d\n", num_tests);
    fscanf(fp, "%d %d", &m, &n);

    for(ctr=1;ctr<=num_tests;++ctr)
```

```

{
myshape = allocate_shape(&myshape, rank, m, n);
with(myshape)
{
double:myshape U, V;
ier = CMSSL_fast_rng (&U, 1.0);
CMC_timer_clear(0);
CMC_timer_start(0);
ParallelMGS(&U, &V,&m, &n);
CMC_timer_stop(0);
printf("M= %d N= %d\n",m,n);
CMC_timer_print(0);
fscanf(fp,"%d%d",&m,&n);
}
deallocate_shape(&myshape);
}
}

void ParallelMGS(double:current *U, double:current *V, int *m, int *n)
{
int i,j,k;
double norm,dotprod;
shape [*n] vecshape;
i=0;

for(i=0;i<*m;++i)
{
with(vecshape)
{
norm=0.0;
norm += ([i][pcoord(0)]*U)*([i][pcoord(0)]*U);
norm=sqrt(norm);
[i][.]*V=[i][.]*U/norm;
for (k=i;k<*m;++k)
{
dotprod=0;
dotprod+=([k][.]*U)*([i][.]*V);
[k][.]*U=[k][.]*U-([i][.]*V)*dotprod;
}
}
}
}
}

```

12.3 Global/Local CM FORTRAN

Global/Local CM FORTRAN is a version of CM FORTRAN that allows a global program to execute in data parallel fashion and call local subroutines that run independently on each node. The local program must use the CMMD message passing library to perform communication tasks and uses the four vector units as its parallel processors. The local subroutine must be written in CM FORTRAN or C and the global routine must be in CM FORTRAN. CMMD is the message passing library that gives the programmer explicit control of each node.

```
C-----
C File      : mgs_l.fcm
C Purpose:  local portion of Global/Local CM Fortran
C           implementation of Modified Gram-Schmidt Process
C Author   : Jim Bercik
C Date    : 11-30-94
C-----
      SUBROUTINE MGS_l(M,N,U,V,NUM_NODES)
      IMPLICIT NONE
      include "/usr/include/cm/cmmd_fort.h"
      INTEGER M,N,NUM_NODES
      REAL U(:,:),V(:,:),NORM,tmp
CMF$    LAYOUT U(:SERIAL,:NEWS),V(:SERIAL,:NEWS)
      INTEGER I,K,J,dotprdet
      INTEGER ORIGINAL_PARTITION_SIZE,Pid,P

C reset partition size according to problem size
      NORM = 0
      Pid=CMMD_self_address()
      ORIGINAL_PARTITION_SIZE=CMMD_partition_size()
      P=CMMD_reset_partition_size(NUM_NODES)

C only execute on the nodes that we want to use.
      IF( Pid .LT. P .AND. P .EQ. NUM_NODES) THEN
      DO 10 I=1,M
          tmp=0
          DO 5 j=1,N/P
5             tmp=tmp+U(i,j)**2
              NORM=CMMD_reduce_float(tmp,CMMD_combiner_add)

              NORM=SQRT(NORM)
          IF (NORM .EQ. 0.0 ) THEN
c             vectors are linearly dependent
              V(1,:)=0.0
              V(2,:)=0.0
              V(3,:)=0.0
```

```

        GOTO 60
    ENDIF
    V(I,:) = U(I, :)/NORM
    DO 20 K=I, M
        tmp=0
        DO 25 J=1, N/P
25         tmp=tmp+U(K, J)*V(I, J)
            dotprdct=CMMD_reduce_float(tmp, CMMD_combiner_add)
            U(K, :) = U(K, :) - (dotprdct*V(I, :))
20         CONTINUE
10        CONTINUE
            P=CMMD_reset_partition_size(ORIGINAL_PARTITION_SIZE)
        ENDIF

60       RETURN

    END

```

12.4 C/CMMD

CMMD is the message passing library that gives the programmer explicit control of each node. CMMD calls must reside in C programs, FORTRAN 77 programs or in the local portion of Global/Local CM FORTRAN programs (Thinking Machines Corporation, 1993b).

```
/*
  Author: Jim Bercik
  Purpose: driver for C/CMMD implementation of ParallelMGS
  File: mgsdrv.c
  Date: 10-12-94
  To Compile: 1.) cc -c MGS.cmmd.c util.c
              2.) cmmd-ld -comp cc -node MGS.cmmd.o util.o
*/

#include <stdio.h>
#include <stdlib.h>
#include <cm/cmmd.h>
#include <math.h>

void mgs();
double random();

void main()
{
  FILE *fp;
  int M,N,size,i,j,ctr,num_tests,vec_size;
  int Pid,num_nodes,P,orig_partition_size;
  int seed=56113;
  double *O,*U;
      orig_partition_size=CMMD_partition_size();
      CMMD_fset_io_mode(stdout,CMMD_independent);

  fprintf(stderr," # nodes %d\n ",orig_partition_size);
  if ((fp =fopen("mgs.dat","r"))==NULL)
  {
    printf("CANNOT OPEN FILE");
    exit(1);
  }
  fscanf(fp,"%d",&num_tests);
  fprintf(stderr,"Num Tests: %d\n",num_tests);
  fscanf(fp,"%d %d",&M,&N);

  /* determine the number of nodes to use */
  num_nodes=orig_partition_size;
  for (i=orig_partition_size;i>=1;i--)
```

```

    {
        if(N <= i*32)
            num_nodes=i;
    }
    P=CMMD_reset_partition_size(num_nodes);
/*    fprintf(stderr," # nodes to use %d\n ",P);*/
    Pid=CMMD_self_address();
    if(Pid < P && P == num_nodes)
    {
        vec_size=N/P;
/* ---- Allocate Memory Dynamically ----- */

        size = sizeof(double);
        U = (double *) malloc(M*vec_size*size);
        O = (double *) malloc(M*vec_size*size);
        if (U == NULL || O == NULL)
        {
            printf("CANNOT ALLOCATE MEMORY");
            exit(1);
        }
/* ----- */

for(ctr=1;ctr<=num_tests;++ctr)
{
    for (i=0;i<M*vec_size;++i) /* fill U with random #'s in [0,1] */
        U[i]=random(&seed);

    fprintf(stderr,"\n\n");
    fprintf(stderr,"Test %d : M=%d N=%d\n",ctr,M,N);

    CMMD_node_timer_clear(0);
    CMMD_node_timer_start(0);

    mgs(M,N,U,O,P);
    CMMD_node_timer_stop(0);
    fprintf(stderr,"Busy: %f seconds \n",
                CMMD_node_timer_busy(0));
    fprintf(stderr,"Elapsed: %f seconds \n",
                CMMD_node_timer_elapsed(0));

    fscanf(fp,"%d%d",&M,&N);
    U = (double *) realloc(U,M*vec_size*size);
    O = (double *) realloc(O,M*vec_size*size);
    if (U == NULL || O == NULL)
    {
        fprintf(stderr,"M=%d vec_size=%d\n",M,vec_size);
        fprintf(stderr,"CANNOT ALLOCATE MEMORY");
        exit(1);
    }
}
} /* end if */
}

```

```

/* -----
Function mgs

Purpose: function to compute the modified gram-shmidt process
-----*/

void mgs(M, N, U, O,P)
int M,N,P;
double *U,*O;

{
int i,j,k,l,vec_size;
double norm,dot_prod,tmp;

vec_size=N/P;
for(i=0;i<M;++i)
{

tmp=0.0; /* calculate dotproduct */
for(l=0;l<vec_size;++l)
tmp=tmp + U[i*vec_size+l] * U[i*vec_size+l];
norm=CMMD_reduce_double(tmp,CMMD_combiner_add);
norm=sqrt(norm);
if(norm == 0.0)
{
fprintf(stderr,"Vectors are Linearly Dependent... Exiting\n");
fprintf(stderr,"norm= %f\n",norm);
return;
}
for (k=0;k<vec_size;++k)
O[i*vec_size+k]=U[i*vec_size+k]/norm;
for(k=i;k<M;++k)
{

tmp=0.0;
for(l=0;l<vec_size;++l) /* calculate dotproduct */
tmp=tmp + U[k*vec_size+l] * O[i*vec_size+l];
dot_prod=CMMD_reduce_double(tmp,CMMD_combiner_add);
for(j=0;j<vec_size;++j)
U[k*vec_size+j]=U[k*vec_size+j]-O[i*vec_size+j]*dot_prod;
}

}

return;
}

/* -----
Function: random
Purpose: generates random # between 0 and 1
-----*/

```

```
double random (seed)
int *seed;
{
  int a=16807,m=2147483647,q=127773,r=2836,lo,hi,test;
  hi=(int)*seed/(int)q;
  lo = *seed % q;
  test=a*lo-r*hi;
  if (test > 0)
    *seed=test;
  else
    *seed=test+m;
  return (double) *seed/(double) m;
}
```

13 Summary

We have studied the development, analysis and implementation of an algorithm using the LogP model and have demonstrated that when we estimate computation and communication separately, the LogP model can provide a reasonable prediction of an algorithms running times for various problem sizes. These predictions have shown to be a more reliable number than dividing the runtimes of the serial implementation by the maximum speedup that Amdahl's Law provides.

Analyzing algorithms using the LogP model has shown to be reasonably simple when asymptotic bounds are used. More effort is required in the algorithm design process than when using a PRAM because of the requirement that we have a finite number of processors and a large number of data elements per processor. This leads to a data layout that can be used on real parallel machines.

M=N	CMF	GL CMF	CSTAR	CMMD	F77	C	LogP + comm	
							(h1)	(h2)
N=32	0.236	0.763	0.704	0.1	0.1	0.04	0.3398	0.3408
N=64	0.282	3.13	2.713	2	0.5	0.28	1.3604	1.36448
N=128	1.104	11.85	10.195	7.5	4.4	2.19	5.4518	5.46816
N=256	4.362	47.201	47.391	9.09	42	17.23	21.889	21.95456
N=512	17.378	188.476	245.15	40.822	538	144.53	113.9025	114.24613
N=1024	255.562	758.953	1929.45	171.267	4394	1192.59	358.0089	359.13728
N=2048	286.26	5773.164	14312.502	1286.26		9907.54	1474.2978	1478.4922
N=4096	1252.1443	46166.84				out of mem	6232.7357	6249.513

Figure 13.1 Modified Gram Shmidt implementation times

The CM FORTRAN implementation of the ParallelMGS algorithm showed a significant speed up over the serial versions, with approximately 34 times speedup over the C implementation for $M=N=2048$. However, as shown in the table in Figure 13.1, the other parallel implementations did not fare as well. The blank fields in the table show times that were either too long to obtain or too long for the system constraints. It is possible that the CMF compiler is better optimized than the other parallel language compilers.

Appendix A - Serial MGS in C

```
/* Jim Bercik
   File: mgs.c
   Purpose: function to compute the modified gram-shmidt process
*/
#include <stdlib.h>
#include <math.h>

void mgs(int M, int N, double *U, double *V);
double dp(double *a, double *b, int N);

void mgs(int M, int N, double *U, double *V)
{
    int i, j, k;
    double norm, dot_prod;

    for(i=0; i<M; ++i)
    {
        norm=sqrt(dp(U+i*N, U+i*N, N));
        if(norm == 0.0)
        {
            printf("Vectors are Linearly Dependent...Exiting\n");
            printf("norm= %f\n", norm);
            return;
        }

        for (k=0; k<N; ++k)
            V[i*N+k]=U[i*N+k]/norm;
        for(k=i; k<M; ++k)
        {
            dot_prod=dp(U+k*N, V+i*N, N);
            for(j=0; j<N; ++j)
                U[k*N+j]=U[k*N+j]-V[i*N+j]*dot_prod;
        }
    }

    return;
}
```

```

/* File: util.c
   Purpose: conatins utility routines for mgs.c
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/times.h>

double random(int *seed);
double dp(double *a,double *b,int n);
void timestart(struct tms *timer, long int *r_time);
void timestop(char *msg,struct tms timer,long int r_time);

#define TICKS 100.    /* this is set to CLK_TCK in <limits.h> */

double random (int *seed)
{
    int a=16807,m=2147483647,q=127773,r=2836,lo,hi,test;
    hi=(int)*seed/(int)q;
    lo = *seed % q;
    test=a*lo-r*hi;
    if (test > 0)
        *seed=test;
    else
        *seed=test+m;
    return (double) *seed/(double) m;
}

double dp(double *a,double *b,int N)
{
    int i;
    double dot_product=0.0;
    for(i=0;i<N;++i)
        dot_product+=a[i]*b[i];
    return(dot_product);
}

void fill_arr(int M,int N,double *U)
{
    int i;
    int seed=56112;
    for (i=0;i<M*N;++i)
        U[i]=random(&seed);
    return;
}

void timestart(struct tms *timer1, long int *r_time)
{
    *r_time=times(timer1);
}

void timestop(char *msg,struct tms timer1,long int r_time)

```

```
{
    struct tms tbuf2;
    long real2;
    real2=times(&tbuf2);

    fprintf(stderr,"%s\n real %.2f\n user %.2f\n sys %.2f\n",msg,
        (real2-r_time)/TICKS,
        (tbuf2.tms_utime-timer1.tms_utime)/TICKS,
        (tbuf2.tms_stime-timer1.tms_stime)/TICKS);
}
```

Appendix B - Serial MGS in FORTRAN 77

```
C Jim Bercik
C FILE :mgs.S.f
C Purpose: function to compute the modified gram-shmidt
C          process
```

```
PROGRAM MGSDVR
IMPLICIT NONE

INTEGER N,M
PARAMETER(M=2048,N=2048)
DOUBLE PRECISION U(M,N),O(M,N)
DOUBLE PRECISION DPR,TMP(M,N),random,d
INTEGER I,K,J,SEED,SECONDS1,SECONDS
C initialize vectors
SEED=563172
CALL DFILLMAT(U,M,N,SEED)
PRINT*, ' F77 MGS '
PRINT*, ' M = ',m, ' N= ',N
CALL MGS(M,N,U,O)
C=====
SUBROUTINE MGS(M,N,U,O)
INTEGER M,N
DOUBLE PRECISION O(M,N), U(M,N),NORM_SQRD
DOUBLE PRECISION DPR
c CALL MATCOPY(U,O,M,N)
DO 10 I=1,M
    NORM_SQRD=DPR(U,I,U,I,M,N)
    IF (NORM_SQRD .EQ. 0) THEN
        PRINT*, 'VECTORS ARE LINEARLY DEPENDENT'
        GOTO 60
    ENDIF
    DO 15 K=1,N
15      O(I,K)=U(I,K)/SQRT(NORM_SQRD)
        DO 20 K=I,M
            DP=DPR(U,K,O,I,M,N)
            DO 30 J=1,N
                U(K,J)=U(K,J)-(DP*O(I,J))
30          CONTINUE
20      CONTINUE
10     CONTINUE
60     RETURN
END
```

```

C Jim Bercik
C File: util.f
C Purpose: contains utility function for msg.S.f
C-----
      DOUBLE PRECISION FUNCTION DPR(V,I,U,J,M,N)
C  computes the doptprduct of the Ith row of matrix V and the Jth
C  row of matrix U
      IMPLICIT NONE
      INTEGER I,J,N,K,M
      DOUBLE PRECISION V(M,N), U(M,N)
      DPR = 0.0e+0
      DO 10 K=1,N
10          DPR = DPR+V(I,K)*U(J,K)
      RETURN
      END
C-----
      SUBROUTINE DFILLMAT(A,M,N,SEED)
      IMPLICIT NONE
      INTEGER M,N,SEED,I,J
      DOUBLE PRECISION A(M,N),RANDOM
      DO 10 I=1,M
          DO 20 J=1,N
20              A(I,J)=RANDOM(SEED)
10          CONTINUE
      RETURN
      END
C-----
      double precision function random(seed)
      integer seed
      integer a,m,q,r,lo,hi,test
      a=16807
      m=2147483647
      q=127773
      r=2836
      hi=seed/q
      lo=mod(seed,q)
      test=a*lo-r*hi
      if ( test .gt. 0) then
          seed=test
      else
          seed=test+m
      end if
      random=float(seed)/float(m)
      return
      end

```

References

- [1] Anton, H., Rorres, C. (1991) *Elementary Linear Algebra Applications Version*, New York.
- [2] Culler, D., Karp, R., Paterrson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., von Eicken, T. (1993) *LogP: Towards a Realistic Model of Parallel Computation*. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
- [3] Longley, James W. (1984) *Least Squares Computations Using Orthogonalization Methods*.
- [4] Rice, J. (1966) Experiments on Gram-Schmidt Orthogonalization. *Math-Comp*. 10:325-328.
- [5] Thinking Machines Corporation (1993a), *CM-5 Users Guide*. Cambridge, MA.
- [6] Thinking Machines Corporation (1993b), *Connection Machine CM-5 Technical Summary*. Cambridge, MA
- [7] Zapata E. L., Lamas, J.A., Rivera, F.F., Plata, O.G. (1991) Modified Gram-Schmidt QR Factorization on Hypercube SIMD Computers. *Journal of Parallel and Distributed Computing*. 12:60-69.

Approval of Examining Committee

Srini Kankanahalli, Ph.D.

George E. Trapp, Ph.D.

Frances Van Scoy, Ph.D.