LARGE-GRAIN PIPELINING ON HYPERCUBE MULTIPROCESSORS†

Chung-Ta King,[1] *Lionel M. Ni*[1,2]

[1]Department of Computer Science
Michigan State University
East Lansing, Michigan 48824

[2]Division of Mathematics and Computer Science
Argonne National Laboratory
Argonne, IL 60439

ABSTRACT: A new paradigm, called *large-grain pipelining*, for developing efficient parallel algorithms on distributed-memory multiprocessors, e.g., hypercube machines, is introduced. Large-grain pipelining attempts to maximize the degree of overlapping and minimize the effect of communication overhead in a multiprocessor system through macro-pipelining between the nodes. Algorithms developed through large-grain pipelining to perform matrix multiplication are presented. To model the pipelined computations, an analytic model is introduced, which takes into account both underlying architecture and algorithm behavior. Through the analytic model, important design parameters, such as data partition sizes, can be determined. Experiments were conducted on a 64-node NCUBE multiprocessor. The measured results match closely with the analyzed results, which establishes the analytic model as an integral part of algorithm design. Comparison with an algorithm which does not use large-grain pipelining also shows that large-grain pipelining is an efficient scheme for achieving a greater parallelism.

1. INTRODUCTION

In recent years, configuring a computer system with multiple VLSI processors becomes a significant trend in parallel processing research. Among the available systems, hypercube multiprocessors have attracted a great amount of attention [GuHS86, HaMS86]. These hypercube machines are characterized by a point-to-point, hypercube interprocessor connection. Each node in the system has its own processor and local memory, and runs independently of other nodes. There are direct connections from the nodes to the host/IO processors which take charge of program development, peripheral controlling, and data downloading and uploading. This kind of architecture is referred to as *distributed-memory multiprocessors*.

The major bottleneck in current hypercube multiprocessors is the communication. Due to the distributed nature, communication between processors are usually carried out through explicit message passing. This induces nonnegligible overhead in transferring data between processors. In addition, according to current implementations, the host controls peripheral devices. Programs and data must be downloaded from the host to nodes, and results must be uploaded back to the host. These downloading and uploading operations cause a severe I/O bottleneck which limits the computation rate.

Therefore, to design parallel algorithms on hypercube multiprocessors, the designer must take into account the communication effect and determine an efficient partitioning and mapping scheme in order to reduce the amount of communication and balance the processor loads. Moreover, a scheduling strategy is needed, which arranges the order of job execution to fully utilize the available resources. In this paper, a new algorithm design paradigm, called *large-grain pipelining*, is introduced. Large-grain pipelining exploits pipelined computations between processors. Through pipelining, the degree of overlapping can be maximized, the effect of communication overhead can be minimized, and, most importantly, the I/O rate can be balanced with the computation rate.

One important characteristic of the pipelined computation is the concept of information flows [NeSn87]. From the view point of information flows, one can identify three important ingredients in large-grain pipelining. First, the flows of data connect the processors in the system into an array or a network. Processors in this network receive streams of data from processors upstream and generate streams of data to processors downstream. The network can be regular, such as a

† This research was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38 and in part by the DARPA ACMP project.

DISCLAIMER

mesh, a binary tree, or a hexagon, or irregular. Furthermore, the shape of the network may be changed as the computation progresses. Next, to form streams of data, data in the computation are partitioned into *blocks* and are piped into the pipelines one by one. Data blocks are the most basic units of processing. It follows that large-grain pipelining is most suitable for problems whose data can be partitioned naturally and operated upon in parallel, i.e., those that can be solved by *data parallel algorithms* [HiSt86]. Pipelined computations will be further elaborated in Section 2 of this paper, followed by examples for performing matrix multiplication in Section 3.

The third important ingredient of large-grain pipelining is the determination of design parameters which include the sizes of data blocks and the number and lengths of pipelines. An optimal set of parameters can maximize the degree of overlapping and balance the I/O rate with the computation rate, which, in turn, optimizes the algorithm. Through accurate analytic modeling, these design parameters can be obtained. It turns out that the sizes of data blocks are usually much greater than that of a single data element to offset the effect of communication overhead. Details of the analytic model will be presented in Section 4, together with experimental results from a 64-node NCUBE. These results confirm the validity of the model and the effectiveness of the large-grain pipelining concept.

2. MODELING PIPELINED COMPUTATION

A *job* can be considered as a function which takes inputs and generates outputs. This situation can best be illustrated through *P-nets* [King88], as shown in Figure 1. Figure 1(a) describes the job of a matrix multiplication, $A \times B = C$, where circles represent *data modules* (the matrices) and squares represent *computation modules* (the multiplication). One way of performing the multiplication is to partition A and B into four submatrices as follows:

$$A \times B = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = C$$

where $C_{ij} = C_{ij}^{(0)} + C_{ij}^{(1)} = A_{i0}B_{0j} + A_{i1}B_{1j}$, $0 \le i,j \le 1$. The P-net shown in Figure 1(b) defines the necessary operations to calculate the matrix product in this case. Four *phases* are identified: *data downloading*, *submatrix multiplication*, *submatrix addition*, and *result uploading*.

In general, there are two major techniques in exploiting parallelism: *concurrency* and *pipelining* [HwBr84]. Concurrency exploits spatial parallelism by utilizing several processors operating on multiple subjobs simultaneously. Thus, if a hypercube multiprocessor has 8 processors, then the 8 submatrix multiplications specified in Figure 1(b) can be performed concurrently as shown in Figure 2(a). However, as mentioned in Section 1, these data are usually downloaded from the host. Thus, the amount of concurrency obtained is really dependent upon the host-to-node communication bandwidth and the degree of overlapping when the host downloads several processors. In other words, if the host downloading is performed sequentially and the communication overhead is very high, then operations in these 8 processors might be carried out sequentially in time, even though they are executed by different processors.

Pipelining exploits temporal parallelism in which streams of data flowing from processor to processor so that different processors can work on different stages of the computation simultaneously. Using this technique, an 8-processor hypercube machine can be organized into 4 pipelines, as shown in Figure 2(b) for one of the pipelines. Each pipeline has 2 *stages*. Although, within each pipeline, operations between stages are executed sequentially, the number of active I/O channels the host has to maintain when downloading submatrices of B is reduced from 8 to 4. Note also that the calculation of $C_{10}^{(0)}$ does not depend on $C_{00}^{(0)}$, but on the availability of A_{10} and B_{00}. Thus, processor 0 can send B_{00} to pro-

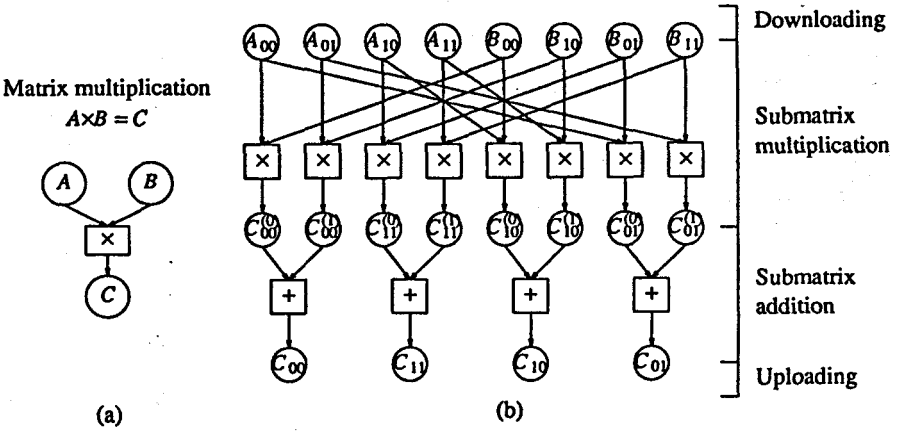
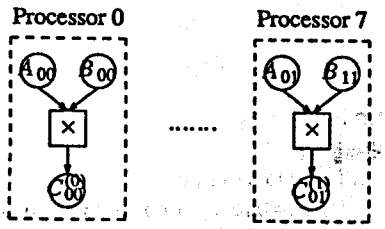
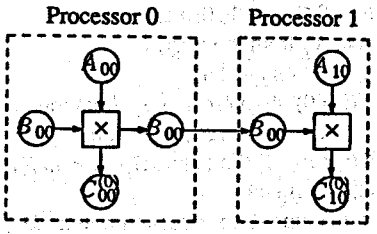


Figure 1. Matrix multiplication and its P-net representation



(a) concurrent operation



(b) pipelined operation

Figure 2. Submatrix multplication using multiple processors

cessor 1 immediately after it receives B_{00} and before it calculates C_{00}^0. In this way, computations in these two processors are overlapped through pipelining.

As discussed in Section 1, one important ingredient of large-grain pipelining is the partitioning of data into data blocks. Thus, submatrices of B can be further partitioned into sub-submatrices to form data streams. Also, when downloading, the host can feed all the pipelines in a round-robin fashion. In this way, even though the downloading is performed sequentially, operations among the processors can overlap. Note that the granularity (the sizes of data blocks) plays a very important role in this situation. The smaller the granularity is, the higher the degree of overlapping will be. However, since there is a fixed overhead associated with the setup of each message, a small granularity results in a large number of messages and a high communication cost. The choice of a suitable granularity is a very important issue and will be discussed in Section 4. Note also that the pipelined computation discussed here is most suitable for developing data parallel algorithms in which data are partitioned and operated upon independently.

From the above discussion, one can see that multiple pipelining is a very general pattern of computation. Even the concurrent computation depicted in Figure 2(a) can be considered as a pipelined computation with 8 pipelines, each has only one stage. Following the flow concept, each stage in the pipeline can be modeled as a computational unit with multiple input and output streams (see Figure 3). Computation of the whole system is then the cascade of these units along different directions. From this point of view, a formal analytic model of pipelined computations can be derived [King88], which can be used to estimate the execution time of the computation.

3. EXAMPLES OF LARGE-GRAIN PIPELINING

In this section, examples are introduced to illustrate the application of large-grain pipelining. Consider the problem of matrix multiplication, $A \times B = C$, where A, B, and C are $M \times M$ matrices. Suppose that A and B are stored in the host initially, and that C will be stored back to the host. Then, input data must be downloaded from the host to the nodes and results must be uploaded back to the host. Downloading and uploading operations introduce extra communication overhead, which differentiate our algorithms from those reported elsewhere [ChSm87, FoOH87].

To perform the multiplication, the following partitioning scheme is used: The hypercube is configured into an $n_1 \times n_2 = N$ mesh, and matrices A and B are partitioned along columns and rows into $n_1 \times n_2$ and $n_2 \times n_3$ submatrices, respectively. Note that, if n_1 and n_2 are both power of 2, then the mesh can be embedded perfectly onto a hypercube so that any node is only one hop away from its four neighbors [SaSc85]. Furthermore, nodes lie in the same column or row form a subcube.

One way to perform the multiplication is to load submatrices of A into the corresponding processors in the mesh. Then, submatrices of B are piped into the hypercube from the host, as shown in Figure 4(a). Each column of n_1 nodes forms a pipeline and there are n_2 pipelines altogether. Let $node_{ij}$ denote the processor located at row i and column j of the mesh. Then, the algorithm is described in Figure 5.

It can be seen from Figure 4(a) that each pipeline will receive a stream of B submatrices from the host. After all

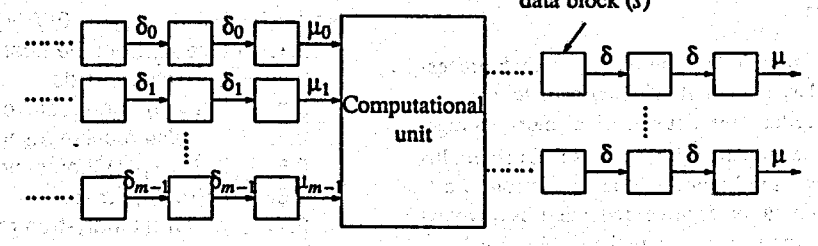
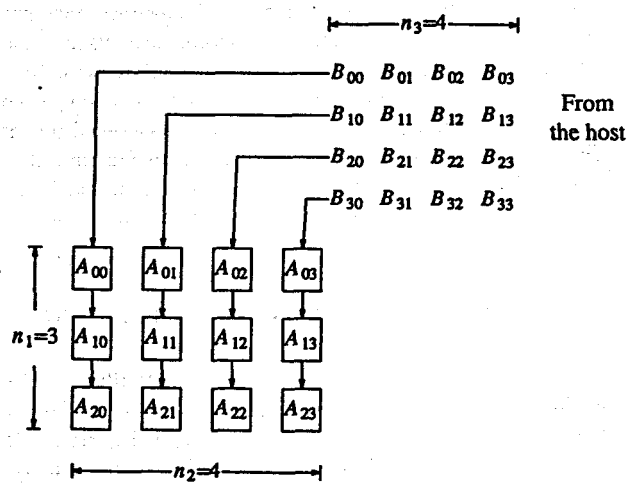
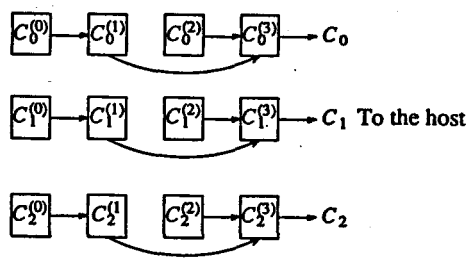


Figure 3. Model of a computational unit



(a) submatrix multiplication



(b) submatrix addition

Figure 4. A matrix multiplication using pipelining concept

submatrices of B have passed through the pipelines, each processor will have collected a portion of a submatrix of C with size $(M/n_1)\times M$. To obtain the final result, a binary-tree reduction is performed among the processors in the same row (Figure 4(b)). Note that the pattern of binary-tree reduction can be perfectly embedded into a hypercube. The pipelining concept can also be applied here in which each $C_i^{(j)}$ can be further partitioned into n_4 smaller sub-submatrices. Then, these small sub-submatrices are piped through the reduction tree to accumulate the final results.

The major problem with Algorithm *I* is that each processor has to store a large amount of intermediate data — $C_{i0}^{(j)}$, ..., $C_{i,n_3-1}^{(j)}$. To reduce the amount of memory storage, we can perform the tree reduction during the submatrix multiplication phase, as shown in Figure 6. That is, as soon as a C submatrix is generated, a binary-tree reduction is invoked among the row processors to accumulate the complete submatrix. This complete C submatrix is then uploaded back to the host. No intermediate submatrices of C need be stored in the processors.

<Algorithm I>:

Host:

1.1. for $i := 0$ to n_1-1 do
1.2. for $j := 0$ to n_2-1 do send A_{ij} to $node_{ij}$;
1.3. for $k := 0$ to n_3-1 do
1.4. for $j := 0$ to n_2-1 do send B_{jk} to $node_{0j}$;
1.5. for $i := 0$ to n_1-1 do receive C_i from $node_{i,n_2-1}$.

Node: ($node_{ij}$, $0\le i\le n_1-1$, $0\le j\le n_2-1$)

2.1. receive A_{ij} from the host;
2.2. for $k := 0$ to n_3-1 do
2.3. if ($i = 0$) then receive B_{jk} from the host
2.4. else receive B_{jk} from $node_{i-1,j}$;
2.5. if ($i \ne n_1-1$) then send B_{jk} to $node_{i+1,j}$;
2.6. $C_{ik}^{(j)} := A_{ij}\times B_{jk}$;
2.7. perform binary-tree reduction with $node_{il}$, $0\le l\le n_2-1$, to obtain $C_i := C_i^{(0)}+...+C_i^{(n_2-1)}$ in $node_{i,n_2-1}$, where $C_i^{(l)} := [C_{i0}^{(l)}, ..., C_{i,n_3-1}^{(l)}]$;
2.8. if ($j = n_2-1$) then send C_i to the host.

Figure 5. Algorithm *I* for matrix multiplication

It is very important here to consider the balance between granularity and communication overhead. It might turn out that communication cost is so high that, combining two or more $C_{ik}^{(j)}$'s into a larger submatrix, e.g., use $[C_{i0}^{(j)}\ C_{i1}^{(j)}]$ in the reduction, will give a better performance. In other words, the tree reduction is performed every two or more iterations instead of one. Thus, the choice of optimal granularity is essential. In our approach, granularity is determined through accurate performance estimations which will be discussed in more detail in Section 4.

<Algorithm II>:

s-1Host:

1.1. for $i := 0$ to n_1-1 do
1.2. for $j := 0$ to n_2-1 do send A_{ij} to $node_{ij}$;
1.3. for $k := 0$ to n_3-1 do
1.4. for $j := 0$ to n_2-1 do send B_{jk} to $node_{0j}$;
1.5. for $k := 0$ to n_3-1 do
1.6. for $i := 0$ to n_1-1 do receive C_{ik} from $node_{i,n_2-1}$.

Node: ($node_{ij}$, $0\le i\le n_1-1$, $0\le j\le n_2-1$)

2.1. receive A_{ij} from the host;
2.2. for $k := 0$ to n_3-1 do
2.3. if ($i = 0$) then receive B_{jk} from the host
2.4. else receive B_{jk} from $node_{i-1,j}$;
2.5. if ($i \ne n_1-1$) then send B_{jk} to $node_{i+1,j}$;
2.6. $C_{ik}^{(j)} := A_{ij}\times B_{jk}$;
2.7. perform binary-tree reduction with $node_{il}$, $0\le l\le n_2-1$, to obtain $C_{ik} := C_{ik}^{(0)}+...+C_{ik}^{(n_2-1)}$ in $node_{i,n_2-1}$;
2.8. if ($j = n_2-1$) then send C_{ik} to the host.

Figure 6. Algorithm *II* for matrix multiplication

<Algorithm III>:

Host:

1.1. for $i := 0$ to n_1-1 do
1.2. for $j := 0$ to n_2-1 do send A_{ij} to $node_{ij}$;
1.3. for $k := 0$ to n_3-1 do
1.4. for $j := 0$ to n_2-1 do send B_{jk} to $node_{0j}$;
1.5. for $k := 0$ to n_3-1 do
1.6. for $i := 0$ to n_1-1 do receive C_{ik} from $node_{i,n_2-1}$.

Node: ($node_{ij}$, $0 \le i \le n_1-1$, $0 \le j \le n_2-1$)

2.1. receive A_{ij} from the host;
2.2. for $k := 0$ to n_3-1 do
2.3. if ($i = 0$) then receive B_{jk} from the host
2.4. else receive B_{jk} from $node_{i-1,j}$;
2.5. if ($i \neq n_1-1$) then send B_{jk} to $node_{i+1,j}$;
2.6. $C_{ik}^{(j)} := A_{ij} \times B_{jk}$;
2.7. if ($j \neq 0$) then
2.8. receive $C_{ik}^{(j-1)}$ from $node_{i,j-1}$
2.9. $C_{ik}^{(j)} := C_{ik}^{(j)} + C_{ik}^{(j-1)}$;
2.10. if ($j = n_2-1$) then send $C_{ik}^{(j)}$ to the host
2.11. else send $C_{ik}^{(j)}$ to $node_{i,j+1}$.

Figure 7. Algorithm *III* for matrix multiplication

Binary-tree reduction is not the only way to accumulate submatrices of C. We consider next a linear reduction scheme, as described in Figure 7. After having calculated a C submatrix, each processor sends that C submatrix to its right neighbor. The accumulation progresses from left to right in a linear fashion until the complete C submatrix is obtained in the right most processor of the row.

Note that in Algorithm *III* a processor will perform a submatrix multiplication before it pauses to wait for the C submatrix from the left neighbor (Statement 2.6 - 2.9 in Figure 7). This is because submatrix multiplication is a computation intensive operation ($O(n^3)$). If every processor performs its multiplication after it receives the C submatrix from the left neighbor, then the delay within each stage will be very long. Following the same argument, one can also expect that the linear reduction will behave badly when n_2 is large, because the reduction path is very long. Performance analyses of these algorithms are given in the next section, where considerations in designing algorithms using the pipelining concept will be further elaborated.

4. PERFORMANCE ANALYSIS

In this section, performance of the three algorithms introduced in the previous section is analyzed. An analytic model is introduced to model the execution of the algorithms and estimate their execution times. To verify the accuracy of the analytic model and the effectiveness of the pipelining concept, the algorithms were implemented on a 64-node NCUBE multiprocessor [HaMS86]. A very close match between the analyzed and measured performance is observed. Comparison with an algorithm without using the pipelining concept was conducted, and the results indicate the superior of algorithms using the pipelining concept.

4.1. NCUBE Multiprocessor

NCUBE is a distributed-memory multiprocessor with a point-to-point hypercube interprocessor connection. Our major concern here is to measure system parameters, such as message startup delay, single-byte transmission time, and elementary operation (e.g., $a \times b + c$) time. All time quantities will be measured in terms of *ticks*, where 1 *tick*=1024/*processor clock rate* (in *Hz*). In general, communication overhead is modeled as [GrRe86]

$$\sigma_h + \tau_h \times s \quad \text{for communication initiated by the host} \tag{4.1}$$

$$\sigma_n + \tau_n \times s \quad \text{for communication initiated by the node} \tag{4.2}$$

where

σ_h = message startup delay in the host

τ_h = single-byte transmission time for messages initiated by the host

σ_n = message startup delay in the node

τ_n = single-byte transmission time for messages initiated by the node

s = size of the message (in *bytes*)

Message startup delay includes such overhead as invoking the *send()* and *recv()* routines and setting up the DMA. Transmission time includes such overhead as transferring the message to and from the operating system's buffer and over the communication link [MuBA86].

The host communication overhead is studied first. Note that when downloading submatrices, the host must first gather all elements of the submatrix and load them into a consecutive memory buffer before the send primitive can be called. Similarly, when receiving submatrices, the host has to scatter the elements of a submatrix from a consecutive message buffer to their respective locations in the matrix. Thus the experiment performed not only measured the overhead in (4.1) but also measured the overhead involved in gathering/scattering operations. Results from the experiment show that the host communication can be modeled perfectly by

$$5.5 + 0.069 \times s \tag{4.3}$$

Note that this value is independent of the number of receiving nodes, which indicates that the host cannot do any useful computation during the interval given by (4.3).

Next, node communication overhead is measured and can be modeled linearly by

$$5.0 + 0.013 \times s \tag{4.4}$$

Note that data gathering/scattering operations are not necessary in the nodes. Finally, the times to execute simple operations $\tau_b=a+b$ and $\tau_c=a\times b+c$, where a, b, and c are floating numbers, are measured and found to be 0.15 and 0.24 ticks, respectively. In summary, we will use the following system parameters on the NCUBE:

σ_h	σ_n	τ_h	τ_n	τ_b	τ_c
5.5	5	0.069	0.013	0.15	0.24

<Note>: all quantities are measured in *ticks*

4.2. Analysis of Pipelined Matrix Multiplication

Given the set of system parameters, the execution time (T) of the algorithms discussed in Section 3 can be derived. Due to the limitation of space, only the execution time of Algorithm *I* will be derived here. The derivations for Algorithm *II* and *III* are very similar. A more formal and accurate analytic model is presented in [King88]. To simplify the analysis, let us assume that matrices A, B, and C are all square matrices with size $M\times M$. The extension to non-square matrices is straightforward. Note that matrices A and B are partitioned into $n_1\times n_2$ and $n_2\times n_3$ submatrices, respectively. Assume that n_1, n_2, and n_3 can divide M evenly. Then, the sizes of submatrices are:

$$\text{size of } A_{ij} \text{ is } \frac{bM^2}{n_1 n_2} \text{ for } 0\le i<n_1,\ 0\le j<n_2$$

$$\text{size of } B_{ij} \text{ is } \frac{bM^2}{n_2 n_3} \text{ for } 0\le i<n_2,\ 0\le j<n_3$$

$$\text{size of } C_{ij} \text{ is } \frac{bM^2}{n_1 n_3} \text{ for } 0\le i<n_1,\ 0\le j<n_3$$

where $b=4$ *bytes* is the size of a floating number on NCUBE. Define the following notations:

$$t_0=\tau_c\frac{M^3}{n_1n_2n_3} \qquad t_1=\tau_b\frac{M^2}{n_1}$$

$$\theta_{12}=\tau_h\frac{bM^2}{n_1n_2} \qquad \psi_{12}=\tau_n\frac{bM^2}{n_1n_2}$$

$$\theta_{23}=\tau_h\frac{bM^2}{n_2n_3} \qquad \psi_{23}=\tau_n\frac{bM^2}{n_2n_3}$$

$$\theta_{13}=\tau_h\frac{bM^2}{n_1n_3} \qquad \psi_{13}=\tau_n\frac{bM^2}{n_1n_3}$$

$$\theta_1=\tau_h\frac{bM^2}{n_1} \qquad \psi_1=\tau_n\frac{bM^2}{n_1}$$

The first step in analyzing the performance of a pipelined algorithm is to identify the critical path of the computation. Note that in Algorithm *I*, all processors perform the same function, and, according to the order of data downloading, processors at the right most column are the ones that start the latest among all processors at the same row. Thus, we need only focus on the operations in $node_{i,n_2-1}$, $0\le i\le n_1-1$. Consider $node_{0,n_2-1}$ first. Operations in this processor depend on two events: the arrival of a B submatrix and the setup of the receive routine. Note that, before the host can download $B_{n_2-1,0}$, it must send all submatrices of A as well as B_{00}, ..., $B_{n_2-2,0}$ to the corresponding nodes. Thus, $B_{n_2-1,0}$ will arrive at $node_{0,n_2-1}$ at

$$\mu^{(0)}=N(\sigma_h+\theta_{12})+n_2(\sigma_h+\theta_{23})$$

where $N=n_1n_2$ is the number of nodes used. The interval between successive B submatrix arrivals is

$$\eta^{(0)}=n_2(\sigma_h+\theta_{23})$$

which is the time the host loads all nodes in the top row exactly once. On the other hand, $node_{0,n_2-1}$ will setup the receive routine for $B_{n_2-1,0}$ only after it has received A_{0,n_2-1} from the host. Thus, this node will be ready for $B_{n_2-1,0}$ at

$$\alpha^{(0)}=n_2(\sigma_h+\theta_{12})+\psi_{12}+\sigma_n$$

The second term, ψ_{12}, accounts for the time to copy the received A submatrix (A_{0,n_2-1}) from the system's buffer, and σ_n is the time to invoke the receive routine for $B_{n_2-1,0}$. Depending on which event occurs first, the processor can start the submatrix multiplication phase at

$$T_0^{(0)}=\max\left\{\mu^{(0)},\alpha^{(0)}\right\}+\psi_{23}$$

where ψ_{23} is the time to copy $B_{n_2-1,0}$ from the system buffer. From now on, in each iteration, $node_{0,n_2-1}$ will (1) send the received B submatrix to $node_{1,n_2-1}$, which takes $\sigma_n+\psi_{23}$; (2) perform a submatrix multiplication, which takes t_0; and (3) receive a B submatrix from the host, which takes $\sigma_n+\psi_{23}$. Let

$$c^{(0)}=2(\sigma_n+\psi_{23})+t_0$$

Then, the iteration time at this node will depend on both $\eta^{(0)}$ and $c^{(0)}$, and is given by

$$\delta^{(0)}=\max\left\{\eta^{(0)},c^{(0)}\right\}$$

Given $T_0^{(0)}$ and $\delta^{(0)}$, the time that $node_{0,n_2-1}$ finishes the submatrix multiplication phase can be found to be

$$T_m^{(0)}=T_0^{(0)}+(n_3-1)\delta^{(0)}+(\sigma_n+\psi_{23})+t_0$$

Furthermore, due to the binary-tree reduction, the submatrix addition phase will take

$$T_a^{(0)}=\log_{n_2}(\sigma_n+2\psi_1+t_1)$$

assuming for simplicity no further partitioning for C submatrices. It follows that the host will receive the first submatrix of C at

$$\mu_0=T_m^{(0)}+T_a^{(0)}+(\sigma_n+\psi_1)$$

In general, for $node_{i,n_2-1}$, $0\le i\le n_1-1$, the time that the submatrix multiplication can start is dependent upon (1) when the first B submatrix, $B_{n_2-1,0}$, arrives, i.e.,

$$\mu^{(i)}=T_0^{(0)}+i(\sigma_n+2\psi_{23})$$

where the second term is the time to transfer one B submatrix from $node_{i,n_2-1}$ to $node_{i+1,n_2-1}$; and (2) when it receives the A submatrix and is ready for $B_{n_2-1,0}$, i.e.,

$$\alpha^{(i)} = (i+1)n_2(\sigma_h+\theta_{12}) + \psi_{12} + \sigma_n$$

Assume that $\mu^{(i)} \geq \alpha^{(i)}$, then we have

$$T_0^{(i)} = \mu^{(i)} \quad T_m^{(i)} = T_m^{(0)} + i(\sigma_n+2\psi_{23}) \quad \delta^{(i)} = \delta^{(0)}$$

and the host will receive the C submatrix from $node_{i,n_2-1}$ at

$$\mu_i = \mu_0 + i(\sigma_n+2\psi_{23})$$

Now consider the host. The first C submatrix will arrive at the host at μ_0, and subsequent C submatrices arrive with an interval of $(\sigma_n+2\psi_{23})$. On the other hand, after receiving one C submatrix, the host needs a time interval of $\theta_1 + \sigma_h$ to copy the submatrix from the system buffer and setup the next receive routine. It follows that the host will finish the uploading phase (and the whole execution) at

$$T = \mu_0 + (n_1-1)\max\left\{(\sigma_n+2\psi_{23}), (\theta_1+\sigma_h)\right\} + \theta_1$$

4.3. Performance Analysis

In this subsection, the execution time derived in Section 4.2 is first validated through experimental results obtained from the NCUBE. Having established the validity, the model can then be used to assist in algorithm designs, such as determining the optimal partition configuration and partition size. Next, the effectiveness of the algorithms introduced in Section 3 is compared with an algorithm without using large-grain pipelining.

The design parameters n_1 and n_2 determine the configuration of the pipelines, where n_2 gives the number of pipelines and n_1 gives the number of stages in each pipeline. If the number of processors (N) is fixed, then we need only consider n_1 or n_2, because $N=n_1n_2$. Figure 8 shows the relationship between the execution time and the pipeline

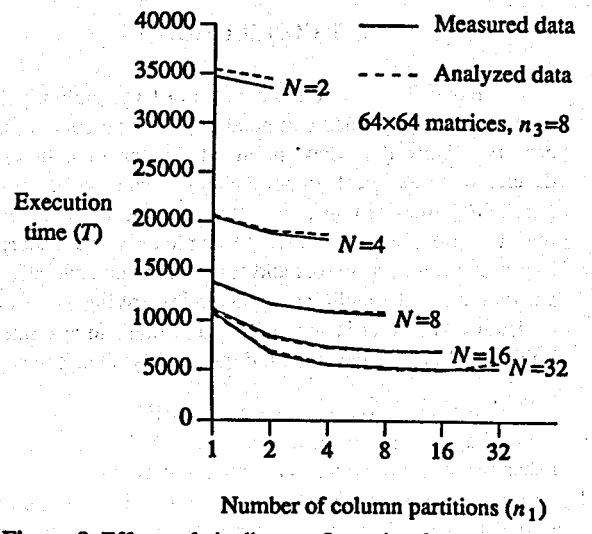


Figure 8. Effects of pipeline configuration for Algorithm *I*

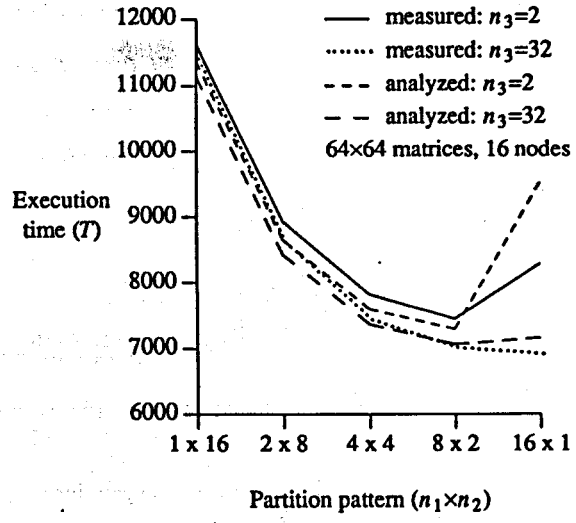


Figure 9. Effects of partition size for Algorithm *I*

configuration for Algorithm *I*. The optimal configuration for a given N can be found from the corresponding curve by choosing its minimum point. It can be seen from the figure that our analytic model predicts the trend correctly and that errors are within 5% in the region of interest, i.e., around the "knees".

It can be seen in Figure 8 that a square configuration always results in good performance [FoOH87]. However, due to the host downloading and uploading operations, the algorithm favors partitions with few pipelines, i.e., a small n_2. On the other hand, when n_1 is small, C submatrices will be large, and so will be the reduction tree. In this case, the communication overhead is too high for efficient execution.

Parameter n_3 determines the partition size of data. The effect of partition sizes on the performance of Algorithm *I* is illustrated in Figure 9. It is evident from Figure 9 that the granularity does have a bearing on the algorithm performance. A large granularity forces the operations in the processors to be executed nearly sequentially, which reduces the degree of overlapping between processors. On the other hand, a small granularity can increase the degree of overlapping, but, due to the fixed overhead in transmitting messages, communication cost will also rise. Though it is not shown, the best partition size in this case is around 16.

Next, performance of the three algorithms introduced in Section 3 is studied. In Figure 10, their execution times are plotted against the pipeline configurations. It can be seen that Algorithms *II* and *III* have the same performance around the optimal regions, and both are superior to Algorithm *I*. This is because the latter has a reduction tree which involves submatrices of large sizes. Note that Algorithm *III* is expected to perform worse when n_2 is large, due to a long path to accumulate C submatrices. However, from Figure 10, we can see that Algorithm *III* performs as good as Algorithm *II* does. A reasonable explanation is that operations in the processors can be fully overlapped in Algorithm *III*.

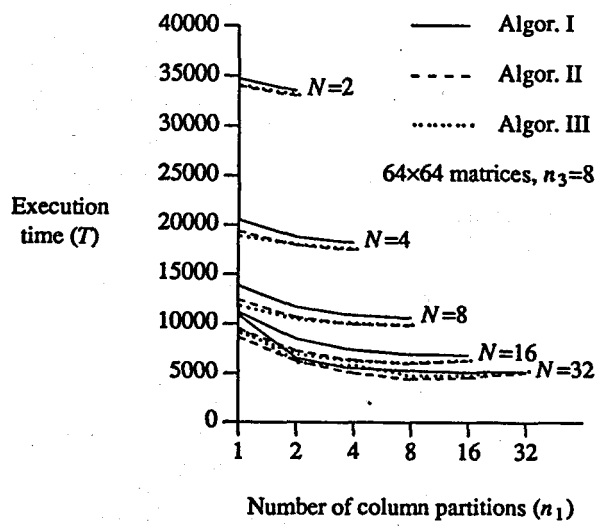


Figure 10. Comparison of algorithms using pipelining

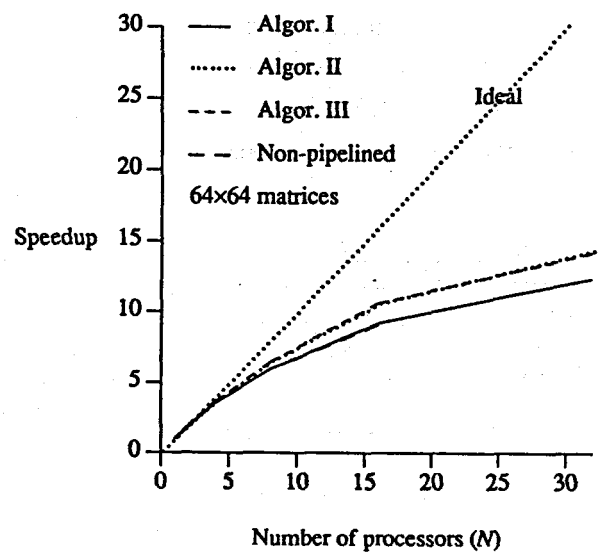


Figure 12. Comparison of speedups

Finally, an algorithm which does not use large-grain pipelining [NiKP87] is compared with the three pipelined algorithms presented in this paper. In the non-pipelined algorithm (see Figure 11), both submatrices of A and B are loaded into the nodes initially. Note that the communication patterns of ring and binary-tree reduction can be perfectly embedded in a hypercube. In each iteration, every node performs a submatrix multiplication, then adds the resultant C submatrices with all nodes in the same row, and finally sends the local B submatrix to the next node in the same column in a ring fashion. The measured speedups are plotted in Figure 12, where the speedup of an algorithm is defined as

$$\text{speedup} = \frac{\text{execution time using } N \text{ nodes}}{\text{execution time using one node}}$$

It can be seen from Figure 12 that Algorithm *II* and *III* both perform better than the non-pipelined one does. This is because the pipelined algorithms can better utilize the overlapped operations and balance the computation with communication through the choice of optimal design parameters. However, the improvement is not so significant as expected. The major reason is because of the communication overhead in current generation hypercube multiprocessors. The overhead in setting up a message is still too high, during which interval the host cannot perform any useful computation. Besides, the gathering/scattering operations in the host are also expensive. More operating system supports are needed to reduce the overhead.

The pipelining concept is expected to benefit from novel architectural designs in the second generation hypercubes [ShFi87]. For instance, if some processors in the hypercube can access the disk directly, then several data streams can flow into the hypercube simultaneously, which results in a true multiple pipelining scheme. More research is needed to assess the idea of pipelining under new environments.

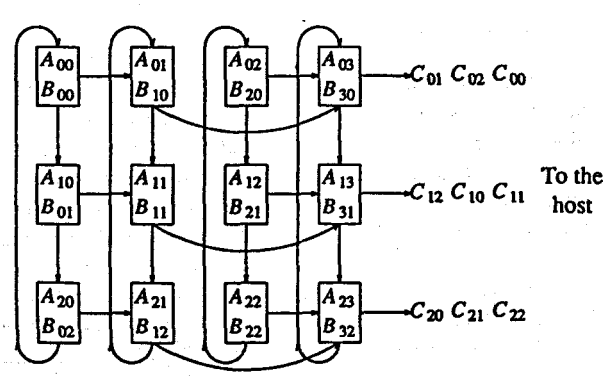


Figure 11. Matrix multiplication without pipelining

5. CONCLUSION

We have introduced the concept of large-grain pipelining — a model of pipelined computations on hypercube multiprocessors. From the view point of information flows, we described three most important ingredients of large-grain pipelining. Also demonstrated, through algorithms for matrix multiplication, are the ways to implement the concept on hypercube multiprocessors and to bring the considerations for partitioning and pipelining together in arriving good design decisions. The latter is accomplished through an accurate analytic model of the algorithm and the underlying architecture.

Experiments on a 64-node NCUBE show a very close match between the measured and analyzed data. This establishes the analytic model as an integral part in determining the optimal design parameters for a given algorithm. Experimental results also indicate the effectiveness of the pipelining con-

cept in improving the performance. With more efficient communication support in second generation hypercube multiprocessors, the improvement using large-grain pipelining is expected to be even greater.

The concept of large-grain pipelining is very similar to systolic arrays [KuLe78], although large-grain pipelining concentrates on asynchronous and coarse-grain computations. The versatile and dynamic interconnection of the hypercube allows the system to form pipelines of different configurations and to adapt to different processing streams. Therefore, it is possible to transform a systolic algorithm into one which uses large-grain pipelining. Main issue here is how to group data and cells in the systolic arrays so that the granularity is large enough to be processed on a hypercube multiprocessor. Research is now undergoing to study the transformation procedure in a formal way.

REFERENCES

[ChSm87] V. Cherkassky, R. Smith, "Efficient Mapping and Implementation of Matrix Algorithms on a Hypercube," *Technical Report*, Department of Electrical Engineering, University of Minnesota, 1987.

[FoOH87] G.C. Fox, S.W. Otto, A.J. Hey, "Matrix Algorithms on a Hypercube I: Matrix Multiplication," *Parallel Computing*, Jan. 1987, pp. 17-31.

[GrRe86] D.C. Grunwald, D.A. Reed, "Benchmarking Hypercube Hardware and Software," *Technical Report*, UIUCDCS-R-86-1303, Department of Computer Science, University of Illinois at Urbana-Champaign, 1986.

[GuHS86] J.L. Gustafson, S. Hawkinson, K. Scott, "The Architecture of a Homogeneous Vector Supercomputer," *Proc. of 1986 Int'l Conf. on Parallel Processing*, August 1986, pp. 649-652.

[HaMS86] J.P. Hayes, T.N. Mudge, Q.F. Stout, S. Colley, J. Palmer, "Architecture of a Hypercube Supercomputer," *Proc. of 1986 Int'l Conf. on Parallel Processing*, August 1986, pp. 653-660.

[HiSt86] W.D. Hillis, G.L. Steele, "Data Parallel Algorithms," *Comm. ACM*, December 1986, pp. 1170-1183.

[HwBr84] K. Hwang, F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Co., 1984.

[King88] C.T. King, "Parallel Computation Modeling for Distributed-Memory Multiprocessors," *Ph.D. Dissertation*, Department of Computer Science, Michigan State University, 1988.

[KuLe78] H.T. Kung, C.E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proc.*, 1978, pp. 32-63.

[MuBA86] T.N. Mudge, G.D. Buzzard, T.S. Abdel-Rahman, "A High Performance Operating System for the NCUBE," *Proc. of the 2nd Conf. on Hypercube Multiprocessors*, 1986.

[NeSn87] P.A. Nelson, L. Snyder, "Programming Paradigms for Nonshared Memory Parallel Computers," in *The Characteristics of Parallel Algorithms*, L.H. Jamieson, D.B. Gannon, R.J. Douglass, eds., MIT Press, 1987.

[NiKP87] L.N. Ni, C.T. King, P. Prins, "Parallel Algorithm Design Considerations for Hypercube Multiprocessors," *Proc. of 1987 Int'l Conf. on Parallel Processing*, 1987, pp. 717-720.

[SaSc85] Y. Saad, M.H. Schultz, "Topological Properties of Hypercubes," *Technical Report*, YALEU/DCS/RR-389, Department of Computer Science, Yale University, June 1985.

[ShFi87] Y. Shih, J. Fier, "Hypercube Systems and Key Applications," in *Parallel Processing for Supercomputing and AI*, K. Hwang, G. DeGroot, eds., 1987.