

# Embeddings

Suppose that we are interested in solving a problem on machine  $A$ . After some searching, we find that someone has designed an algorithm for the same problem on machine  $B$ . Unfortunately, machine  $A$  and machine  $B$  are different architectures. How can we make use of the algorithm available for the problem to simplify our task? Consider a  $p$ -processor configuration of machine  $A$  and machine  $B$ . Suppose that we can find a bijective mapping  $f$  from processors of  $B$  to processors of  $A$  such that whenever two processors in  $B$  are connected, the corresponding processors in  $A$  are also connected. We can run the same algorithm on machine  $A$  using the following strategy: Processor  $i$  in machine  $A$  runs the algorithm as if its id is  $f^{-1}(i)$ . To find a particular neighbor, processor  $i$  finds the relevant neighbor of  $f^{-1}(i)$  in machine  $B$ , and finds the processor in  $A$  to which this neighbor is mapped to. The algorithm will now run on machine  $A$  as if it were machine  $B$ . We do not need to expend any effort in designing an algorithm for  $A$ . Furthermore, if a parallel program is available on machine  $B$ , we can reuse the program on machine  $A$  with minimal modifications to redefine the id and neighbors of a processor. Even more important, we can use any algorithm developed for any problem on machine  $B$  to solve the same problem on machine  $A$  without any loss in speedup or efficiency.

Mappings from one model to another, such as the one described above, are important because they allow us to use the algorithms developed on one model to solve problems on another. Suppose that we have such a mapping from model  $B$  to model  $A$ . Model  $B$  is called the *source* and model  $A$  is called the *target* and the mapping itself is called an embedding of model  $B$  in model  $A$ . Note that our ability to run an algorithm designed for model  $B$  on a different model  $A$  without loss of efficiency does not mean that our algorithm is efficient on  $A$ . It merely states that efficiency is preserved when we move across the models, i.e. the algorithm will run as efficiently on  $A$  as it did on  $B$ . It may be possible to design an algorithm for  $A$  that is more efficient. Therefore, it makes sense to move algorithms across models using embeddings provided the algorithms are optimally efficient to begin with.

In this book, we have adopted the permutation network model as the primary model of parallel computation based on theoretical reasons and practical observations based on architectures of parallel computers. If so, why are embeddings of interest to us? For some problems, it is easier to develop algorithms by imagining a processor network that reflects the natural pattern of connectivity suggested by the problem at hand. For instance, tree is a natural network for developing algorithms for broadcasting. Two-dimensional meshes are natural for representation of matrices. Thus, if we learn how to embed an important virtual topology into our model of parallel computation, we are free to choose that virtual topology for designing an algorithm. It is of course, important to make sure that the algorithm is optimal with respect to the sequential algorithm.

It is with this view that we study embeddings in this course. Though our target architecture of a permutation network trivially allows embedding of any architecture into it, we will study embeddings using hypercubic permutations only. The reason for this is that instead of building an interconnection network that can support arbitrary permutations, it would be enough to build a network that only supports hypercubic permutation. However, we will use a slightly relaxed notion of hypercubic permutations. We call a permutation hypercubic if a) it is a permutation and b) every pair of processor that communicate differ in at most one bit (the bit position is not fixed and it can be different for different pairs of communicating processors).

## 1 Embedding Rings using Hypercubic Permutations

To study the embedding of a ring using hypercubic permutations, we must first study *binary reflected gray codes*. Consider a  $d$ -bit binary string. Since each position can have a 0 or a 1, the different possible  $d$ -bit strings can be used to denote  $2^d$  distinct values. Let us choose the decimal numbers  $0, 1, 2, \dots, 2^d - 1$  to be the  $2^d$  distinct values. A  $d$ -bit code assigns a unique decimal number in this range to each possible  $d$ -bit string. The  $d$ -bit binary numbers, for example, are a  $d$ -bit code.

If a code has the property that the bit strings that represent any two consecutive numbers differ in exactly one bit, such a code is called a gray code. In a  $p = 2^d$  processor parallel computer, the rank of each processor is represented by a  $d$ -bit string. The connection between gray codes and hypercubic permutations is that two processors are allowed to communicate if their bit representations differ in one bit and two consecutive strings in a gray code differ in one bit. Thus, gray codes are useful for mapping using hypercubic permutations.

A simple way to generate a  $d$ -bit gray code is as follows: The one bit gray code is the same as the one bit binary code. i.e., The bit string 0 represents the decimal number 0 and the bit string 1 represents the decimal number 1. Suppose we want to create a two-bit gray code. We write down the bit strings corresponding to a 1-bit gray code (0 and 1) in a column and take a mirror reflection of it in a mirror placed at the bottom of the column. We then prefix a 0 to the original bit strings and prefix a 1 to the mirror reflected bit strings. This process is illustrated in figure 1. Why does this work? Any consecutive pair of bit strings from the original sequence still differs by one bit because they are both prefixed by a 0. The same is true in the mirror reflected sequence because they are all prefixed by a 1. Because of the mirror reflection, the last string in the original sequence and the first string in the mirror reflected sequence are the same except that the former is prefixed by a 0 and the later is prefixed by a 1. Therefore, they differ in exactly one bit. Similarly, the first string in the original sequence and the last string in the mirror reflected sequence are the same except the for the difference in the prefix. Therefore, they too, differ in exactly one bit.

This processor of construction can be used to generate  $d$ -bit gray codes for any  $d \geq 1$ . We first generate the  $(d - 1)$ -bit gray code recursively, take a mirror reflection of it and add a prefix of 0

<u>One bit</u>	<u>Two bit</u>	<u>Three bit</u>
0	00	000
1	01	001
	11	011
	10	010
		110
		111
		101
		100

Figure 1: Construction of two and three bit binary reflected gray codes.

to the original  $(d - 1)$ -bit gray code and add a prefix of 1 to its mirror reflection. This process of construction is illustrated in figure 1. Because the code is formed using repeated mirror reflections, it is called binary reflected gray code.

To see why the binary reflected gray codes are useful in the embedding of rings using hypercubic permutations, consider figure 2 in which the binary and gray codes are shown next to each other along with the decimal numbers they represent. The binary numbers can be viewed as the id's of processors in a ring. For instance, the left neighbor of 011 is 010 and the right neighbor of 011 is 100. The gray code can be viewed as the id's of processors on the parallel computer. Any two consecutive strings differ by one bit and this is true for the first and last strings as well. To embed a ring processor, we take the id of the processor in the ring, find the corresponding gray code, and assign it to this processor. This will ensure that whenever a link exists between two processors in a ring, the corresponding processors in the parallel computer have ranks that differ in one bit. Hence, operations on the ring such as each processor communicating with its left neighbor (or each communicating with its right neighbor) use hypercubic permutations on the parallel computer. This also means that left shift and right shift permutations can be effected using hypercubic permutations.

As an example, consider processor 4 in the ring. Its binary code is 100. From figure 2, 100 corresponds to a gray code of 110. If we view the bit string 110 itself as a binary number, its decimal value is 6. Thus, we map processor 4 in the ring to processor 6 in the parallel computer.

If we try to run a ring algorithm using this embedding, it is important to find the ring id of each physical processor and the physical ranks of its ring neighbors. This is because the actual physical processor should run the algorithm as if its rank is the rank of the ring processor it represents. However, the communication with its ring neighbors should actually be conducted with the physical processors to which the ring neighbors are mapped to.

Suppose that we have two functions `btog` and `gtob` that translate a binary number to a gray

<u>Decimal</u>	<u>Binary</u>	<u>Gray</u>
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Figure 2: Three bit binary and binary reflected gray codes.

code number and vice versa, respectively. If  $i$  is the rank of a physical processor, its ring id and the physical ranks of its left and right ring neighbors can be computed as follows:

$$\begin{aligned}
 \text{ring id} &= \text{gtob}(i) \\
 \text{left neighbor} &= \text{btog}[(\text{gtob}(i) - 1 + p) \bmod p] \\
 \text{right neighbor} &= \text{btog}[(\text{gtob}(i) + 1) \bmod p]
 \end{aligned}$$

A  $p$ -processor ( $p$  a power of 2) array can, of course, be embedded by first embedding a  $p$ -processor ring using hypercubic permutations and ignoring the link connecting the first and the last processors of the ring/array.

## 2 Embedding Meshes/Tori using Hypercubic Permutations

A mesh of any dimension, and in fact, a multidimensional torus can be embedded using hypercubic permutations. This can be done in a very simple fashion, which is a straightforward extension of embedding rings. Let us first restrict our attention to a two-dimensional torus. Further assume that the number of processors along each dimension is a power of two. i.e., consider a  $2^r \times 2^s$  torus for embedding into a  $2^{r+s}$ -processor parallel computer.

Any processor in the  $2^{r+s}$ -processor parallel computer is represented by a bit string of length  $r + s$ . We can split this bit string into two strings, one consisting of the first  $r$  bits and the other consisting of the remaining  $s$  bits. The torus has  $2^r$  rows and  $2^s$  columns. Since the rows and columns of a torus are rings, we can embed each row and each column of the torus using hypercubic permutations on the appropriate number of processors. Suppose that we take a specific row, say  $i$ . We can represent  $i$  in gray code using  $r$  bits. For this fixed  $r$ -bit combination, all possible bit strings of length  $s$  form a  $2^s$ -processor subset. The row is a  $2^s$ -processor ring which can be embedded in this  $2^s$ -processor subset. specifically, if  $(i, j)$  is the id of a processor in the torus, it is mapped to

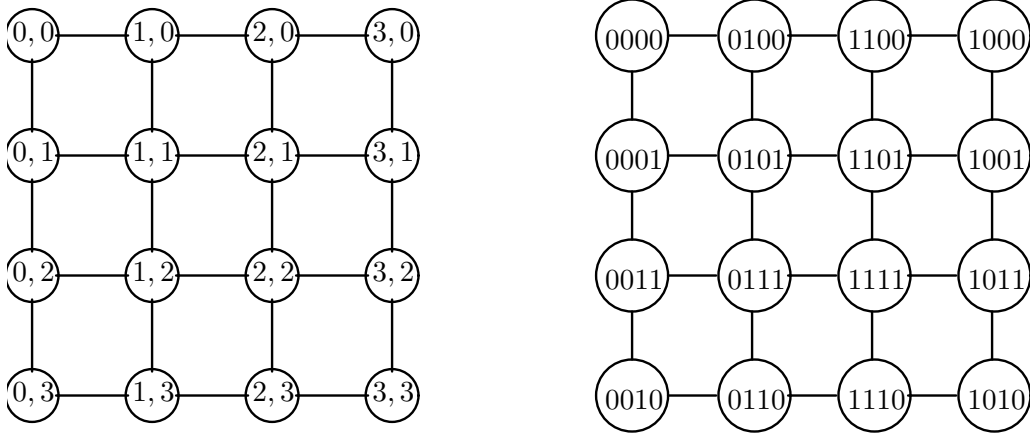


Figure 3: Embedding a 16-processor mesh using hypercubic permutations.

the processor obtained by concatenating the bit string  $\mathbf{btog}(i)$  with  $\mathbf{btog}(j)$ . The embedding of a  $4 \times 4$ -processor mesh into a 16-processor parallel computer is shown in figure 3.

Why does this embedding work? First of all, let us make sure that each processor in the torus is mapped to a unique physical processor. Let ‘ $\parallel$ ’ be an operator denoting the concatenation of bit strings. Consider two torus processors  $(i, j)$  and  $(i', j')$ . These are mapped to  $\mathbf{btog}(i) \parallel \mathbf{btog}(j)$  and  $\mathbf{btog}(i') \parallel \mathbf{btog}(j')$ , respectively. If the two torus processors are distinct,  $i \neq i'$  and/or  $j \neq j'$ . Since  $\mathbf{btog}$  is a one-to-one function, the corresponding physical processors are distinct as well.

It is easy to see that this embedding preserves the links between neighboring processors. For example, the west neighbor of  $(i, j)$  is  $((i - 1 + 2^r) \bmod 2^r, j)$ . The processor  $(i, j)$  is mapped to  $\mathbf{btog}(i) \parallel \mathbf{btog}(j)$  and  $((i - 1 + 2^r) \bmod 2^r, j)$  is mapped to  $\mathbf{btog}((i - 1 + 2^r) \bmod 2^r) \parallel \mathbf{btog}(j)$ . Since  $\mathbf{btog}(i)$  and  $\mathbf{btog}((i - 1 + 2^r) \bmod 2^r)$  differ in exactly one bit, the ranks of the two physical processors differ in one bit. If  $l = x \parallel y$  is the id of a physical processor in binary representation ( $l$  is  $r + s$  bits long,  $x$  is  $r$  bits long etc..)

$$\begin{aligned}
 \text{torus id} &= \mathbf{gtob}(x) \parallel \mathbf{gtob}(y) \\
 \text{west neighbor} &= \mathbf{btog}[(\mathbf{gtob}(x) - 1 + 2^r) \bmod 2^r] \parallel y \\
 \text{east neighbor} &= \mathbf{btog}[(\mathbf{gtob}(x) + 1) \bmod 2^r] \parallel y \\
 \text{north neighbor} &= x \parallel \mathbf{btog}[(\mathbf{gtob}(y) - 1 + 2^s) \bmod 2^s] \\
 \text{south neighbor} &= x \parallel \mathbf{btog}[(\mathbf{gtob}(y) + 1) \bmod 2^s]
 \end{aligned}$$

We can extend the embedding of a two dimensional torus to embedding a torus of any dimension in a straightforward fashion. Suppose we have a  $2^{r_1} \times 2^{r_2} \times \dots \times 2^{r_k}$ -processor  $k$ -dimensional torus. Split the physical processor id bit string into  $k$  consecutive substrings of length  $r_1, r_2, \dots, r_k$ . The id of a processor in the torus is a sequence of  $k$  values, one for each dimension. Embed each dimension separately such that dimension  $j$  is embedded using  $r_j$  bits.

### 3 Embedding Binary Trees using Hypercubic Permutations

Can a  $p$ -leaf binary tree be embedded using hypercubic permutations? Let  $p$  be a power of two. The tree has  $2p - 1$  processors. We choose a  $2p$ -processor parallel computer.

Let us partition the processors of the parallel computer into two groups, depending on if the number of 1 bits in the processor id is odd or even. Call a processor *even* if it has an even number of 1 bits and *odd* otherwise. Clearly, there are  $p$  even and  $p$  odd processors. In any embedding, the root of the tree should be mapped to a physical processor. Without loss of generality, suppose that it is mapped to an even processor. If we want to embed the binary tree using hypercubic permutations, the two children of the root must be mapped to odd processors. By repeatedly using the same argument, all the processors at a fixed level of the tree are all mapped to either odd processors or they are all mapped to even processors. Tree processors in alternate levels starting with the level containing the root are all mapped to even processors. The rest of the levels are mapped to odd processors. Without loss of generality, assume that the leaf-level is mapped to even processors. Then, the number of even processors required to satisfy this assignment is

$$p + \frac{p}{4} + \frac{p}{4^2} + \dots$$

But we only have  $p$  even processors. Therefore, it turns out that a binary tree cannot be embedded using hypercubic permutations.

The problem arises because the levels of the tree do not all have the same number of processors. In particular, the leaf level has approximately half the processors of the binary tree and they all must be mapped to either odd or even processors. This problem could be circumvented if we somehow find a way of assigning half the processors at every level to even processors and the remaining half to odd processors. Such an assignment will equally use even and odd processors. Using such a strategy, it is possible to embed a binary tree such that whenever two logical processors are connected in the tree, the corresponding physical ranks have processors whose ranks differ in at most two bits. The key to this approach is to map one of the children of the root to the same type processor as the root is assigned to. For example, we can assign the root to an even processor, its left child to an odd processor with one bit difference in rank, and its right child to an even processor which differs in exactly two bits. We will not discuss this embedding here as there is another embedding which is simple, uses less processors, and is most often used in practice.

Most tree algorithms use one level of the tree at a time. For such algorithms, it does not matter if multiple tree processors are mapped to a single physical processor as long as no two tree processors at the same level are mapped to the same physical processor. To ensure this, we need at least  $p$  processors because the leaf-level of the tree has  $p$  processors. Consider the following assignment of processors illustrated in figure 4. The  $p$  leaves of the tree are mapped to the  $p$  processors such that leaf  $i$  (numbering the leaves from left to right starting with 0) is mapped to processor  $i$ . Each internal processor in the tree is mapped to the same physical processor as its left child is mapped

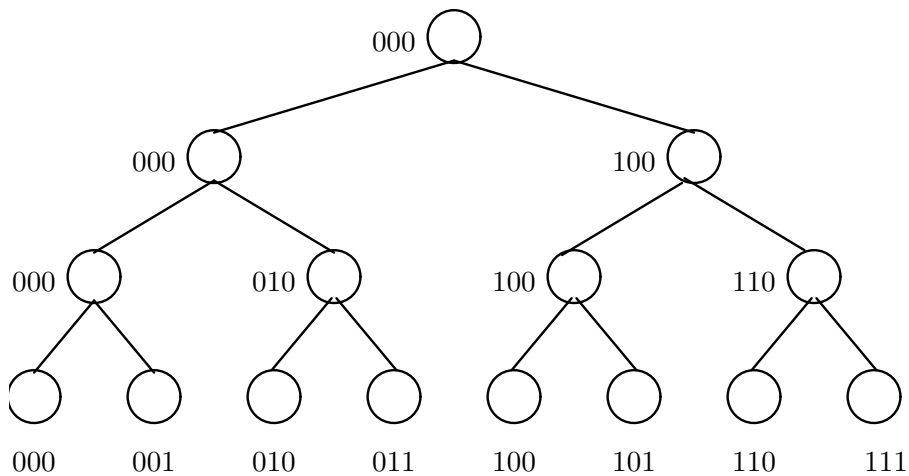


Figure 4: Embedding a 16-processor tree using hypercubic permutations.

to. Thus, all the  $(\log p + 1)$  processors in the leftmost path of the tree are mapped to physical processor with rank 0. This mapping has a load of  $\log p + 1$ , where load is the maximum number of logical processors simulated by a physical processor. However, the effective load is 1 if only one level of the tree is used at a time.

We show that this mapping respects hypercubic permutations. Let us first show that the links connecting the leaf processors to their parents use only hypercubic permutations. The parent of a leaf processor is assigned the physical processor with rank of  $2i$  (for some  $0 \leq i < \frac{p}{2}$ ) and its tree children in the parallel computer are itself and  $2i + 1$ . Since  $2i$  and  $2i + 1$  differ exactly in one bit, this mapping conforms to hypercubic permutations. Thus, all the tree links connecting the leaf-level to the previous level conform to hypercubic permutations.

This argument can be applied repeatedly to higher levels in the tree. The level next to the leaf-level is mapped to the processors  $** \dots *0$ . If we ignore the last bit as it is fixed for all the processors, it is as if this level is mapped to consecutive processors in the same manner as described above. Thus, the links connecting this level to its previous level also map to processors whose ranks differ in one bit. In general, level  $j$  (suppose root is at level 0) is mapped to processors of the form  $** \dots *00 \dots 00$ , where the number of 0's to the right is  $\log p - j$ . Since all the tree links conform to hypercubic permutations, the tree is embedded using only hypercubic permutations.

To run a tree algorithm, each physical processor should know at what levels of the tree it should participate and the actual ranks of its parent and children for that level. This is easy to obtain. Processor  $i$  should participate in levels  $(\log p - 1), (\log p - 2), \dots, (\log p - j)$ , where  $j$  is the number of trailing zeros in  $i$ . Suppose that processor  $i$  is participating in level  $k$  of the binary tree. Its parent and children for this level are:

parent : change  $(\log p - k)^{th}$  bit to zero.  
left child : self.  
right child : change  $(\log p - k - 1)^{th}$  bit to one.

A number of other models can also be embedded using hypercubic permutations. These include the butterfly networks, pyramids and meshes of trees.

# Matrix Algorithms

Algorithms for matrix problems often have a higher computation complexity than communication complexity. Thus, optimally efficient algorithms for such problems can be designed even for networks with low connectivity, such as the mesh. Therefore, we will develop our algorithms for the mesh-connected parallel computer. We use the results on embedding of a mesh in a permutation network using hypercubic permutations to extend our algorithms to a permutation network. We will examine the problem of computing the product of two matrices.

Before we proceed to study algorithms for matrices, we should consider how matrices should be distributed in our parallel computer. This gives us the distribution of the input as well as the distribution according to which output should be generated. Suppose that we want to distribute an  $m \times n$  matrix on a  $\sqrt{p} \times \sqrt{p}$  mesh, where  $m \geq \sqrt{p}$  and  $n \geq \sqrt{p}$ . One simple way is to rank the processors with id's ranging from 0 to  $p - 1$ , and distribute the rows of the matrix evenly to all the processors. In this scheme, each processor gets  $\frac{m}{p}$  rows of the matrix. A similar scheme is to distribute the columns of the matrix to the processors based on a linear ordering of the processors.

Notice that we are developing the matrix algorithms on a two-dimensional mesh topology, and a two-dimensional mesh and a matrix have the same structure. It turns out to be beneficial to use this similarity to perform the input distribution. An  $m \times n$  matrix can be treated as a  $\sqrt{p} \times \sqrt{p}$  matrix where each element is a submatrix of size  $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ . We will use this natural representation to distribute matrices to processors in a parallel computer. This distribution of matrices is known as block distribution.

## 4 Cannon's Matrix Multiplication Algorithm

Suppose that we are given two matrices  $A$  and  $B$  and we need to compute their matrix product  $C = A \times B$ . To keep things simple, let us assume that  $A$  and  $B$  are square matrices. To focus attention on the parallel component of the matrix multiplication algorithm, further assume that the input matrices  $A$  and  $B$  are square matrices of size  $n \times n$  and the size of the mesh is  $n \times n$ .

The input matrices are initially distributed such that the  $(i, j)^{th}$  processor contains  $a_{ij}$  and  $b_{ij}$ . The objective is to generate  $c_{ij}$  in  $(i, j)^{th}$  processor ( $i$  is the row number  $j$  is the column number). Consider the problem of generating  $c_{00}$  on processor  $(0, 0)$ .  $c_{00}$  is given by

$$c_{00} = \sum_{i=0}^{n-1} a_{0i} \times b_{i0}$$

$a_{00}$  and  $b_{00}$  are already available in  $(0, 0)$  and can be multiplied to give the first term in the above sum. To form the second term,  $a_{01}$  and  $b_{10}$  are needed which are located in the processor to the east of and to the south of  $(0, 0)$  respectively. It can easily be observed that  $a_{0k}$  and  $b_{k0}$  needed to

**Algorithm 1** *Cannon's Algorithm*

```

Move  $a$ 's in row  $i$  to the west by  $i$  steps.
Move  $b$ 's in column  $i$  to the north by  $i$  steps.
for  $i = 0$  to  $n - 1$  do
     $c = c + a * b$ .
    send  $a$  to the west.
    send  $b$  to the north.

```

Figure 5: Algorithm for multiplying two matrices on a mesh.

form the  $(k + 1)^{th}$  term in the sum are located in  $k^{th}$  processor to the east of and to the south of  $(0, 0)$  respectively. Therefore, by continuously moving the  $a$ 's to the west and the  $b$ 's to the north, the operands required for computing the consecutive terms can be brought into  $(0, 0)$  in consecutive time steps. This is the purpose of the matrix multiply loop in Figure 5.

Unfortunately, this is not the case with the other processors. However, the following conditions hold: The processor  $i$  steps to the east of  $(i, j)$  contains  $a_{i,(j+i) \bmod m}$  and the processor  $j$  steps to the south of  $(i, j)$  contains  $b_{(i+j) \bmod m, j}$ . Together, these can form the  $[(i + j + 1) \bmod n]^{th}$  term in computing  $c_{ij}$ . Recall that

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \times b_{kj}$$

Therefore, by moving the  $a$ 's to the west  $i$  times and the  $b$ 's to the north  $j$  times, the terms required for computing  $[(i + j + 1) \bmod n]^{th}$  term for  $c_{ij}$  can be brought into  $(i, j)$ . It is important to note that, since all the processors  $(i, j)$  in a row have the same value of  $i$  and the motion of  $a$ 's is to the west (i.e. not crossing rows), we can achieve the required movements of  $a$ 's for each processor simultaneously and without needing any extra storage in the process. Same is true for the movement of  $b$ 's also. Once this preconditioning is done, consecutive terms can be formed by fetching the  $a$ 's from the east and  $b$ 's from the north, as before. This requires a precondition step to prepare the matrices for multiplication, as shown in Figure 5.

It should be noted that the time required for this algorithm is  $O(n)$ , compared to  $O(n^3)$  of the sequential algorithm. As  $n^2$  processors are used, it follows that this algorithm is optimally efficient.

Now, consider extending this algorithm for the more realistic case of multiplying two  $n \times n$  matrices on a  $\sqrt{p} \times \sqrt{p}$  size mesh of processors, where  $n > \sqrt{p}$ . We can treat an  $n \times n$  matrix as a  $\sqrt{p} \times \sqrt{p}$  matrix where each element of the matrix is a submatrix of size  $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ . The algorithm would remain the same, except that instead of moving an element, we move an entire submatrix and instead of performing element-wise multiplication, we perform multiplication of submatrices. Thus, the algorithm consists of  $\sqrt{p}$  phases. In each phase, we move submatrices of size  $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$  and perform local matrix multiplication on submatrices of size  $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ . Computing the product of two

submatrices takes  $O\left(\frac{n^3}{p\sqrt{p}}\right)$  time. Communicating a submatrix to a neighbor takes  $O\left(\tau + \mu\frac{n^2}{p}\right)$  time. The total complexity is given by

$$\begin{aligned}\text{Computation Time} &= O\left(\frac{n^3}{p}\right) \\ \text{Communication Time} &= O\left(\tau\sqrt{p} + \mu\frac{n^2}{\sqrt{p}}\right)\end{aligned}$$