Chapter 23

Parallelizing Two-dimensional FFTs

23.1 The Computation of Multiple 1D FFTs

The need to compute a set of 1D FFTs arises naturally in many applications. If the FFTs have the same length and properties (e.g., all real or complex), an appropriate sequential FFT algorithm may be applied to them one after the other, or it may be applied to them all at once stage by stage. Since the same set of twiddle factors are applied, it is inefficient in this context to compute them on the fly. Instead, the twiddle factors should be pre-computed (once), stored, and reused for each FFT in the set. Using the latter approach, the storage for the twiddle factors is the same as that required for a single FFT. It is thus straightforward to adapt FFT software to compute multiple 1D FFTs on a uniprocessor machine,

The simplest parallel algorithm to compute multiple 1D FFTs is "embarrassingly parallel"; the set of M 1D FFTs (of length N each) can simply be evenly divided among the p processors, and each processor simply applies an appropriate sequential algorithm to compute its share of $\lceil M/p \rceil$ 1D FFTs. In this case, there is no communication, but the twiddle factors need to be pre-computed and stored in each processor. This was referred to as the "independent processors" approach in [46].

If the computation associated with each single FFT is divided among several processors in some way, additional inter-processor communication is required. This "cooperative processors" approach was also explored in [46], and two algorithms using this approach were compared with the "independent processors" idea on an nCUBE 2 hypercube consisting of 128 processors. As expected, the timing results reported in [46] confirmed efficiency values of 99 to 99.9% for the "independent processors" approach, which was in contrast to efficiency values of 29 to 49% for the two implementations of the cooperating processors.

Although the computation of a two-dimensional FFT may be viewed as computing multiple 1D FFTs in each direction (as shown in the next section), the parallelization of a two-dimensional FFT presents another challenge because the highly efficient "independent processors" approach cannot be used on both directions without incurring inter-processor data communications, and it is no longer the clear winner. The 2D FFTs are the focus of the remainder of this chapter.

23.2 The Sequential 2D FFT Algorithm

In this section a fast serial algorithm for computing the DFT on a two-dimensional image consisting of $N_1 \times N_2$ signals is reviewed. The signals are stored in an $N_1 \times N_2$ matrix \boldsymbol{x} . An entry in the signal matrix is denoted by x_{ℓ_1,ℓ_2} . The 2D-DFT of \boldsymbol{x} is defined by the following equation [70, 72]:

(23.1)
$$X_{r_1,r_2} = \sum_{\ell_1=0}^{N_1-1} \sum_{\ell_2=0}^{N_2-1} x_{\ell_1,\ell_2} \omega_{N_1}^{r_1\ell_1} \omega_{N_2}^{r_2\ell_2},$$
for $r_1 = 0, 1, \cdots, N_1 - 1$, and $r_2 = 0, 1, \cdots, N_2 - 1$.

If the equation above is used in a straightforward (naive) way, $\Theta(N_1N_2)$ arithmetic operations are required to compute each X_{r_1,r_2} , yielding a total cost of $\Theta(N_1^2N_2^2)$, or $\Theta(N^4)$ if $N_1 = N_2 = N$. Fortunately, this may be reduced very significantly by separating the 2D-DFT into a series of 1D-DFTs, which can each be implemented using a fast 1D-FFT algorithm. This process is shown below.

$$X_{r_1,r_2} = \sum_{\ell_1=0}^{N_1-1} \sum_{\ell_2=0}^{N_2-1} x_{\ell_1,\ell_2} \omega_{N_1}^{r_1\ell_1} \omega_{N_2}^{r_2\ell_2}$$

$$= \sum_{\ell_1=0}^{N_1-1} \omega_{N_1}^{r_1\ell_1} \left(\sum_{\ell_2=0}^{N_2-1} x_{\ell_1,\ell_2} \omega_{N_2}^{r_2\ell_2} \right)$$

$$= \sum_{\ell_1=0}^{N_1-1} \omega_{N_1}^{r_1\ell_1} \left(\tilde{X}_{\ell_1,r_2} \right)$$

$$= \sum_{\ell_1=0}^{N_1-1} \left(\tilde{X}_{\ell_1,r_2} \right) \omega_{N_1}^{r_1\ell_1}, \quad r_1 = 0, 1, \cdots, N_1 - 1, \quad r_2 = 0, 1, \cdots, N_2 - 1.$$

Thus, by effecting a series of (ordered) 1D-FFTs on the N_1 rows (of length N_2 each) of \boldsymbol{x} , the data in row ℓ_1 are transformed to \tilde{X}_{ℓ_1,r_2} for $0 \leq r_2 \leq N_2 - 1$. The total cost of this row-transform phase is $\Theta(N_1N_2\log_2 N_2)$. This is followed by a series of (ordered) 1D-FFTs on the N_2 columns (of length N_1 each) of \tilde{X} , and column r_2 of \tilde{X} is transformed to X_{r_1,r_2} for $0 \leq r_1 \leq N_1 - 1$. The cost of the column-transform phase is $\Theta(N_2N_1\log_2 N_1)$. The efficiency of the discrete Fourier transform of a digital image consisting of $N_1 \times N_2$ signals is thus improved from $\Theta(N_1^2N_2^2)$ to $\Theta(N_1N_2\log_2(N_1N_2))$. When $N_1 = N_2 = N$, the computing cost is reduced from $\Theta(N^4)$ to $\Theta(N^2\log_2 N)$. Therefore, the computational efficiency of the 2D-FFTs is even greater than that of the 1D-FFTs.

Note that if an in-place unordered FFT is used in both row-transform phase and column-transform phase, then the entry x_{ℓ_1,ℓ_2} would be finally overwritten by X_{m_1,m_2} , where the binary representation of m_1 bit-reverses that of ℓ_1 , and m_2 is related to ℓ_2 in the same way.

For obvious reasons, the approach based on equation (23.2) is called the "rowcolumn" method [41, 70] or the "separable" method [46] in the literature.

23.2.1 Programming considerations

As noted by Duhamel and Vetterli [41], the matrix which contains the data of a 2D transform grows quickly. For example, if $N_1 = N_2 = 1024$, there are more than one million complex numbers in the 1024×1024 matrix. Depending on the programming language used, this large matrix is stored either column-by-column or row-by-row in computer memory. In order to minimize the number of memory accesses, an efficient method must be used to access blocks of consecutive rows or blocks of consecutive columns in a manner compatible with the storage scheme. To achieve this, the row-column FFT is often performed by including a matrix transposition between the FFTs on the columns and the FFTs on the rows in order to allow access to the data by blocks [41]. A fast method for matrix transposing was proposed by Eklundh in [45]. The two possible implementations of the row-column method are depicted in Figure 23.1.



23.2.2 Computing a single 1D FFT stored in a 2D matrix

If the data of a single 1D FFT of length $N = 2^n$ is stored in a 2D matrix of dimension $N_1 \times N_2$, where $N = N_1 \times N_2$, $N_1 = 2^{n_1}$, and $N_2 = 2^{n_2}$, it can also be computed by independent 1D FFTs on the rows and columns of the matrix. However, note that in this context the twiddle factors used are derivatives of ω_N^ℓ rather than derivatives of $\omega_{N_1}^\ell$ and $\omega_{N_2}^\ell$. For example, a 2D matrix was used in [51] to store a 1D FFT for distribution to processors that are connected by a hypercube or a 2D mesh network.

Consequently, except for using different twiddle factors, one can use the 2D FFT algorithm to compute a single 1D FFT if the latter is stored in a 2D matrix. It follows that parallel 2D FFT algorithms can be used to compute a single 1D FFT as well.

23.2.3 Sequential algorithms for matrix transposition

The standard method depicted in Figure 23.2 applies to any block partitioned matrix. Since each A_{ij} block could be of dimensions 1×1 , $k \times k$, or $m \times \ell$, the matrix A may be square or rectangular of any size.



The divide-and-conquer method depicted in Figure 23.3 is recursive by nature. A square or rectangular matrix may be divided into four submatrices at each level.





The method depicted in Figure 23.4 was originally proposed by Eklundh [45] to facilitate the out-of-core (when core memory was still in use and the memory size is very limited) matrix transposition. The idea is that two rows are read in each time, the appropriate elements are permuted, the modified two rows are then written out to the disk. The example in Figure 23.4 shows that the first permutation step involves

row 1 and row 2, the second step involves row 3 and row 4, the third step involves row 1 and row 3, and the fourth step involves row 2 and row 4. (Each row can be a single row as originally proposed, or it can be a block row if each A_{ij} is a submatrix instead of a single element.)

Note that unlike the other two algorithms, some blocks will be moved multiple times in Eklundh's method. For example, block A_{14} was moved twice in Figure 23.4.

Note that this more complicated permutation scheme can be easily described using the binary representation of the row and column indices of block A_{ij} , which is denoted as A[i, j] below. Suppose $i = b_2 b_1 b_0$, and $j = u_2 u_1 u_0$; then three pairwise exchanges may switch $A[b_2 b_1 b_0, u_2 u_1 u_0]$ with $A[b_2 b_1 u_0, u_2 u_1 b_0]$, followed by $A[b_2 u_1 u_0, u_2 b_1 b_0]$, and finally with $A[u_2 u_1 u_0, b_2 b_1 b_0]$.

Although this method requires that the number of (block) rows and (block) columns be the same, since i and j must have the same number of bits to effect pair-wise exchange throughout, there is no restriction on the dimension of each block itself. That is, A[i, j] could again be a single element, a square, or a rectangular matrix.



23.3 Three Parallel 2D FFT Algorithms for Hypercubes

Readers are assumed to be familiar with the hypercube multiprocessors introduced in Chapter 18 and the large number of parallel 1D FFTs described in previous chapters. As one would expect, different parallel algorithms are available to handle the column-oriented mapping scheme, row-oriented mapping scheme, and the 2D-block mapping scheme. Three sample algorithms are used to introduce some basic ideas in the following sections.

23.3.1 The transpose split (TS) method

The transpose split (TS) method used in [22, 24] parallelizes the row-column 2D FFT algorithm. In the example shown in Figure 23.5, the four processors P_0 , P_1 , P_2 , and P_3 , are each allocated a block of consecutive columns or rows. Clearly, only the matrix transposition phase(s) will incur inter-processor communication, and an efficient parallel algorithm for matrix transposition is all that is needed.

Figure 23.5 The TS (transpose split) method with column or row data allocation.



Each processor performs 1D FFTs on allocated columns:

23.3.2 The local distributed (LD) method

The *local distributed* (LD) method in [22, 24, 46] does not have a matrix transposition phase. For the example above, each processor first independently computes multiple 1D FFTs on allocated columns (or rows). In the next phase, since each individual row (or column) of the updated signal matrix is shared by all four processors as depicted in Figure 23.6, an appropriate "parallel" 1D FFT algorithm from previous chapters is used to transform the multiple rows (or columns) all at once. (Note that only one set of twiddle factors is needed.) An implementation proposed in [46] used the sequential "split-radix" algorithm to the rows, and the parallel "radix-4" to the columns.

Figure 23.6 The LD (*local distributed*) method with column or row wise data allocation.



Since a processor may apply each stage of the FFT transformation to all 1D FFTs at once, the same communication algorithm for a single parallel 1D FFT may be easily modified to include the data needed for all 1D FFTs in each message, i.e., the message size is increased, but the number of messages remains identical to that incurred by parallelizing a single 1D FFT. Therefore, on machines with large communication bandwidth, the communication cost is expected to impact the performance of 2D FFT less significantly.

23.3.3 The 2D block distributed method

The basic idea of the 2D block distributed method [22] is depicted by a simple example on a 2-by-2 processor grid in Figure 23.7 below.



Even if one wishes to do so, it is not always possible to configure the available p processors as a $\sqrt{p} \times \sqrt{p}$ grid. For example, if $p = 2^d$ and d is an odd number, the p processors cannot be arranged as a square grid. In what follows, assume $p = 2^d = 2^{d_1+d_2} = 2^{d_1} \times 2^{d_2}$. A desirable objective is that the 2D block distributed method work for all possible values of d_1 and d_2 . To achieve this, it is apparent that the dimensions of the rectangular processor grid should be input parameters to the program.

When the p processors form a hypercube, the processors on each row and each column of the grid form a subcube, hence the name *subcube-grid* [26, 27, 29]. One may then choose any convenient dimensions, because the performance of the 2D block distributed method is not affected by the dimensions of the subcube-grid. The reasons for this are that the following observations hold, regardless of how the subcube-grid is configured.

- each processor has $(N_1 \times N_2)/p$ data elements,
- each message is either of length $(N_1 \times N_1)/p$ or one half of it (depending on the chosen parallel 1D FFT algorithm),
- all subcube-doubling message exchanges involve only neighboring processors,
- the total number of messages is $d_1 + d_2 = d$ always,
- the subcube-doubling communication algorithm does not cause traffic congestion.

Note that the generalized 2D block method includes the LD method as a special case corresponding to a $1 \times p$ or $p \times 1$ subcube-grid.

23.3.4 Transforming a rectangular signal matrix on hypercubes

Although a square signal matrix was shown in all examples in the previous sections, in reality the signal matrix may not be square. A little reflection leads to the conclusion that all three algorithms work without significant modification when the matrix is not square. The only proviso is that the TS (transpose split) method requires that its parallel matrix transposition algorithm handle rectangular matrices of any dimension.

23.4 The Generalized 2D Block Distributed (GBLK) Method for Subcube-grids and Meshes

Recall that the signal data for a 2D FFT are stored naturally in an $N_1 \times N_2$ matrix, and that by viewing the hypercube as various 2D subcube-grids, the generalized 2D block distributed (GBLK) method may be regarded as partitioning the matrix on a corresponding subcube-grid. Furthermore, the performance of the GBLK method is not affected by the aspect ratio of the subcube-grid for reasons discussed in the last section. In other words, given a hypercube consisting of $p = 2^d = 2^{d_1+d_2}$ processors, the $N_1 \times N_2$ data matrix may be mapped to any subcube-grid of dimensions $\gamma_1 \times \gamma_2$, where $\gamma_1 = 2^{d_1}$, and $\gamma_2 = 2^{d_2}$. Given below are the four possible subcube-grids for p = 8, together with the corresponding data mappings.

Figure 23.8 The four GBLK data mappings on 8-node subcube-grids.



23.4.1 Running hypercube (subcube-grid) programs on meshes

The 512 computing nodes on the Intel Touchstone DELTA computer [44] are connected as a 16-by-32 two-dimensional mesh, and disjoint sub-meshes of dimensions (*row*, *col*), with $row \leq 16$ and $col \leq 32$, can be allocated to individual users [44]. For example, Figure 23.9 shows three 8-processor physical sub-meshes allocated from a 4-by-8 mesh.



A 1-by-8 submesh allocated from a 4-by-8 mesh

2 3



From E. Chu [25], The International Journal of High Performance Computing Applications, 13(2):124–145, 1999. With permission.

A mesh has fewer communication channels than a hypercube, and it is not possible to have the allocated processors in Figure 23.9 form a subcube-grid. However, since a mesh is a connected network, there is a path between any two processors, a hypercube program implementing the subcube-doubling communication algorithm will run correctly on a mesh as shown by the 8-node examples in Figure 23.10. Regardless of whether a subcube-grid or a mesh is used, the matrix elements can always be distributed to processors using the same mapping scheme, and the communication algorithm can still be understood as passing the same sequences of messages between the same designated pairs of processors, and the length of each message remains unchanged.

Thus a different physical network topology will not affect the "correctness" of the algorithm or the software. However, a different physical network topology can

- (i) increase the "physical distance" (measured by the number of hardware channels or hops) between communicating processors, and
- (ii) cause "contention of communication channels" when logically-disjoint message paths overlap badly on the physical network

and hence compromise the effectiveness of the logical topology in achieving its objective.

If the *hop (distance) penalty* is low, the first problem will not affect the performance much. However, the contention of communication channels may be a serious problem

because it can cause severe traffic congestion. In the next section, the extent of traffic congestion is directly related to the physical distance a message travels when using the subcube-doubling technique on a mesh, and the question of how to reduce traffic congestion by using an optimal aspect ratio to configure the physical mesh (at runtime) is addressed.



23.5 Configuring an Optimal Physical Mesh for Running Hypercube (Subcube-grid) Programs

The objective in configuring an optimal physical mesh is to minimize communication overhead due to the multi-hop (distance) penalty and traffic congestion. Since circuit-switching is used by the DELTA mesh and other currently available message-passing multiprocessors to manage the network, the contention of communication channels is resolved in a particular manner. It is, therefore, useful to show directly in Section 23.5.3 that the effect of channel contention on a circuit-switching physical mesh is also minimized by the optimal aspect ratio derived in Theorem 23.1 in Section 23.5.1.

23.5.1 Minimizing multi-hop penalty

To support the subcube-doubling communication algorithm on an arbitrary γ_1 -by- γ_2 subcube-grid using a μ_1 -by- μ_2 physical mesh, consider first how to minimize the total physical distance the messages travel. In the following analysis it is assumed that $\gamma_1 = 2^{d_1}$, $\gamma_2 = 2^{d_2}$, $\mu_1 = 2^{\delta_1}$, $\mu_2 = 2^{\delta_2}$, and $p = \gamma_1 \times \gamma_2 = \mu_1 \times \mu_2 = 2^d$, where d_1, d_2 ,

 δ_1 and δ_2 are non-negative integers. It is also assumed without loss of generality that the *p* processors are numbered consecutively row by row in both the subcube-grid and the mesh: e.g., processors $P_0, \dots, P_{\gamma_2-1}$ form the first row of the subcube-grid, and processors P_0, \dots, P_{μ_2-1} form the first row of the mesh.

When the subcube-doubling algorithm is used for concurrent message exchanges among all pairs of processors, the communication requirement is the same for every processor (see Figures 23.8 and 23.10). It is thus sufficient to examine the requirement of processor P_0 in what follows.

Figures 23.8 and 23.10 demonstrate that regardless of the aspect ratio of the subcube-grid or the mesh, P_0 communicates with P_i , $i = 2^{\ell}$, $0 \leq \ell \leq d - 1$. However, the physical distance between P_0 and each such P_i varies with the physical meshes used to run the program. For example, according to Figure 23.10, using a 2-by-4 mesh, P_0 is one hop away from either P_1 or P_4 , and two hops away from P_2 ; using a linear array, P_0 is one hop away from P_1 , two hops away from P_2 , and four hops away from P_4 .

Theorem 23.1 is next stated and proved, which shows that the total physical distance between P_0 and all designated P_i 's is a function of d and δ_1 .

Theorem 23.1 Assume that the p processors denoted by P_0, \dots, P_{p-1} are arranged row by row on a $\mu_1 \times \mu_2$ physical mesh, where $\mu_1 = 2^{\delta_1}$, and $\mu_2 = 2^{\delta_2}$. If processor P_0 communicates with the $d = \delta_1 + \delta_2$ processors required by the subcube-doubling technique, then the total physical distance is given in hops by $H_{mesh}(d, \delta_1) = 2^{\delta_1} + 2^{d-\delta_1} - 2$, and is minimized when $\delta_1 = d/2$, i.e., $\mu_1 = \mu_2 = \sqrt{p}$, assuming that d is an even number.

Proof: Since each row of the physical mesh is a linear array, the physical distance between P_0 and P_j , $j = 2^{\ell_2}$, $0 \le \ell_2 \le \delta_2 - 1$, is exactly 2^{ℓ_2} hops. Since each column of the physical mesh is also a linear array, the physical distance between P_0 and P_i , $i = 2^{\delta_2 + \ell_1}$, $0 \le \ell_1 \le \delta_1 - 1$, is exactly 2^{ℓ_1} hops. Therefore, the total distance between the *d* pairs of communicating processors can be computed by

(23.3)
$$H_{mesh}(d, \delta_1) = \sum_{\ell_2=0}^{\delta_2-1} 2^{\ell_2} + \sum_{\ell_1=0}^{\delta_1-1} 2^{\ell_1} = 2^{d-\delta_1} + 2^{\delta_1} - 2.$$

Minimizing $H_{mesh}(d, \delta_1)$ with respect to δ_1 yields $\delta_1 = d/2$. Hence $\mu_1 = \mu_2 = 2^{d/2} = \sqrt{p}$.

23.5.2 Minimizing traffic congestion

To quantify the traffic congestion caused by the subcube-doubling communication on the physical mesh, a traffic weight $w_{i,j}^{(k)}$ is associated with each communication channel $C_{i,j}$ which physically connects processors P_i and P_j on the mesh, and $w_{i,j}^{(k)}$ is defined to be the number of overlapped communication paths on that channel during the k^{th} communication step. Since the subcube-doubling communication is performed independently within each row and within each column of the mesh, it is sufficient to examine the extent of traffic congestion within one row and one column, which are linear arrays of sizes μ_2 and μ_1 on a μ_1 -by- μ_2 mesh. Figure 23.11 shows the overlapped communication paths caused by each subcube-doubling communication step on a linear array consisting of processors P_0 , P_1 , \cdots , P_7 . The values of $w_{i,i+1}^{(k)}$ defined for each channel connecting the neighboring processors P_i and P_{i+1} on the linear array are given in Table 23.1, where $0 \le i \le 6$ and $1 \le k \le 3$. The extent of traffic congestion can be quantified by the total weight $\sum_{k=1}^{3} \sum_{i=0}^{6} w_{i,i+1}^{(k)} = 28$ for this example.

Figure 23.11 The overlapped communication paths incurred by the subcube-doubling algorithm (p = 8).



From E. Chu [25], The International Journal of High Performance Computing Applications, 13(2):124–145, 1999. With permission.

Table 23.1 Traffic weights for communication channels $C_{i,i+1}$ in Figure 1.								
Step (k)	$w_{0,1}^{(k)}$	$w_{1,2}^{(k)}$	$w_{2,3}^{(k)}$	$w_{3,4}^{(k)}$	$w_{4,5}^{(k)}$	$w_{5,6}^{(k)}$	$w_{6,7}^{(k)}$	$\sum_{i=0}^{6} w_{i,i+1}^{(k)}$
k = 1	1	0	1	0	1	0	1	4
k = 2	1	2	1	0	1	2	1	8
k = 3	1	2	3	4	3	2	1	16
$\sum_{k=1}^{3} \sum_{i=0}^{6} w_{i,i+1}^{(k)}$								28

From E. Chu [25], The International Journal of High Performance Computing Applications, 13(2):124–145, 1999. With permission.

Now consider the total traffic weight caused by the subcube-doubling algorithm on a linear array consisting of $\mu = 2^{\delta}$ processors in Lemma 23.2 below.

Lemma 23.2 The total traffic weight imposed by the subcube-doubling communication on a linear array of size $u = 2^{\delta}$ is given by $W(\delta) = \sum_{k=1}^{\delta} \sum_{i=0}^{\mu-2} w_{i,i+1}^{(k)} = 2^{2\delta-1} - 2^{\delta-1}$.

Proof: Observe that during the k^{th} subcube-doubling communication step, each pair of communicating processors is $m = 2^{k-1}$ hops apart. Since the traffic weights on the m channels connecting processors P_0, P_1, \dots, P_m is summed up by $\sum_{\ell=1}^m \ell$, and the traffic weights on the m-1 channels connecting processors $P_m, P_{m+1}, \dots, P_{2m-1}$ is summed up by $\sum_{\ell=1}^{m-1} \ell$, the total weight on channels connecting each disjoint group of

 $2m = 2^k$ processors, can be computed by

(23.4)
$$\sum_{\ell=1}^{m} \ell + \sum_{\ell=1}^{m-1} \ell = m^2 = 2^{2k-2}.$$

Since there are $2^{\delta-k}$ groups of 2^k processors performing the k^{th} subcube-doubling communication step independently, the total weight from all k communication steps on the entire linear array can be computed by

(23.5)
$$W(\delta) = \sum_{k=1}^{\delta} \sum_{i=0}^{\mu-2} w_{i,i+1}^{(k)} = \sum_{k=1}^{\delta} 2^{\delta-k} \left(\sum_{\ell=1}^{2^{k-1}} \ell + \sum_{\ell=1}^{2^{k-1}-1} \ell \right) = 2^{2\delta-1} - 2^{\delta-1} .$$

Theorem 23.3 The total traffic weight imposed by the subcube-doubling communication on a μ_1 -by- μ_2 mesh, where $\mu_1 = 2^{\delta_1}$, $\mu_2 = 2^{\delta_2}$, and $p = 2^{\delta_1 + \delta_2} = 2^d$, is given by $W_{mesh}(d, \delta_1) = 2^{d-1} \left(2^{\delta_1} + 2^{d-\delta_1} - 2\right)$, and is minimized when $\delta_1 = d/2$, i.e., $\mu_1 = \mu_2 = \sqrt{p}$, assuming that d is an even number.

Proof: As noted earlier, the subcube-doubling communication is performed independently within each row and each column of the μ_1 -by- μ_2 mesh. Lemma 23.2 implies

(23.6)
$$W_{row}(\delta_2) = 2^{2\delta_2 - 1} - 2^{\delta_2 - 1}$$

and

(23.7)
$$W_{column}\left(\delta_{1}\right) = 2^{2\delta_{1}-1} - 2^{\delta_{1}-1}.$$

Since there are $\mu_1 = 2^{\delta_1}$ rows and $\mu_2 = 2^{\delta_2}$ columns, the total traffic weight on the μ_1 -by- μ_2 mesh is given by

(23.8)

$$W_{mesh}(d, \delta_{1}) = 2^{\delta_{2}} \times W_{column}(\delta_{1}) + 2^{\delta_{1}} \times W_{row}(\delta_{2})$$

$$= 2^{d-1} \left(2^{\delta_{1}} + 2^{d-\delta_{1}} - 2\right)$$

$$= 2^{d-1} \times H_{mesh}(d, \delta_{1}) .$$

Therefore, the value $\delta_1 = d/2$ that minimizes $H_{mesh}(d, \delta_1)$ in Theorem 23.1 also minimizes $W_{mesh}(d, \delta_1)$.

Corollary 23.4 follows immediately from Theorems 23.1 and 23.3.

Corollary 23.4 If the given physical mesh consists of $p = 2^d$ processors, where d is an odd number, then $H_{mesh}(d, \lfloor d/2 \rfloor) = H_{mesh}(d, \lceil d/2 \rceil)$, and $W_{mesh}(d, \lfloor d/2 \rfloor) = W_{mesh}(d, \lceil d/2 \rceil)$.

The results above are depicted in Figure 23.12 for the 8-processor example, i.e., either a 2-by-4 mesh or a 4-by-2 mesh should be used to run the hypercube program regardless of how the matrix is partitioned among the processors.

Figure 23.12 Optimal 8-node meshes for running hypercube programs.



23.5.3 Minimizing channel contention on a circuit-switched network

When a message is to be sent from one processor to another on a circuit-switched network, a header packet is sent to reserve all of the channels required to build its path. After this "circuit" is established, the message is transmitted, and an end-of-message indicator releases the channels [44]. Therefore, when the paths of several concurrent messages overlap, the establishment of each corresponding circuit must wait for the shared channel(s) to be released from one previously built route. Such wait time can be eliminated if there is no overlapped concurrent communication paths. From the analysis of subcube-doubling communication on a mesh in the previous section, it is clear that there are exactly "m" m-hop paths overlapping each other when a processor sends a message to a destination m hops away within each row or each column of the physical mesh; i.e., the number of overlapped paths is the same as the physical mesh chosen to minimize the total physical distance a processor's messages travel in Theorem 23.1 also minimizes the total number of overlapped communication paths, and hence the effect of channel contention on a circuit-switched network.

Thus, Theorems 23.1, 23.3, and Corollary 23.4 imply that when a hypercube program is run (or emulated) on a mesh,

for best performance, a closest-to-square physical mesh should be used.

The user still has the flexibility of choosing a particular data mapping to facilitate memory access, and/or to simplify data structures, and/or for programming convenience.

23.6 Pipelining Subcube-doubling Communications on All Hypercube Channels

The idea of pipelining subcube-doubling communications on all hypercube channels was used by Calvin in [22] to overlap communication and computation in implementing parallel 2D FFT algorithms. To help explain this idea, recall from Chapter 18 the d communication steps in the basic subcube doubling algorithm—they are depicted again in Figure 23.13 (d = 3 in the example). Note that while there are $d(2^{d-1})$ channels in a hypercube of dimension d, only 2^{d-1} channels are used per communication step.

To make use of *all* available channels simultaneously, each processor is required to *pipeline* its outgoing messages to all its neighbors. For example, P_0 is shown to pipeline its messages to P_1 , P_2 , and P_4 using a *non-blocking* send in each step as shown in Figure 23.14, so does P_1 as well as every other processor.

Since the pipelining technique typically involves sending multiple shorter messages instead of a single long message, the startup time caused by multiple sends must be "overlapped" (or "masked") by arithmetic work to a large extent if the pipelining method is to be effective. Instead of simply displaying a data mapping which can accomplish this objective, the computation of N_1 1D FFTs of length N_2 on $p = 2^d = 8$ processors is used as an example to "construct" and "demonstrate" such a mapping step by step. **Figure 23.13** The *d* synchronous exchanges in the subcube-doubling algorithm (d = 3).



Figure 23.14 Pipelining subcube-doubling "send" on all channels (d = 3).



Referring to Figure 23.15, observe that P_0 has been assigned the first block of $\frac{N_2}{p}$ columns (p = 8 in the example). Instead of applying FFT steps to the entire block all at once and exchanging a single message with one neighbor, the data in P_0 are now partitioned into $d = \log_2 p$ portions (d = 3 in the example), and P_0 interleaves its local computation on each portion of data with message passing to each neighbor as described in Algorithm 23.6.

Algorithm 23.1 The actions by P_0 in s	tep 1.
begin	
$d := \log_2 p$	d is the hypercube dimension
$\underline{\mathbf{for}}\ k = 1\ \underline{\mathbf{to}}\ d$	iterate on d portions of data
P_0 performs local computation on	the k^{th} portion;
P_0 sends the updated data from t	his portion to its k^{th} neighbor;
end for	
end	

After P_0 completes the initial step, ideally the message P_0 expects from P_4 has already arrived, and P_0 can now use the incoming data to update the first portion of its data. (If the data is always ready when the processor needs it, the communication time is said to be fully masked.) Referring to Figure 23.15 again, observe that after P_0 updates the first portion of data, P_0 immediately sends the newly updated data from this portion to P_1 . Since the data in each block row are divided evenly among the p processors, P_0 must rotate its d neighbors each step. By this time, ideally the data P_0 needs to update the second portion of data has arrived from P_2 . After updating the second portion, P_0 immediately sends the necessary data from this portion to P_4 , and so on. The actions by P_0 in the second step are described in Algorithm 582. With the understanding that the list of appropriate neighbors is rotated by one position (see Figure 23.15), the generic description of step 2 may be used to describe step 3. (For d > 3, the same description may be used for step 2, step 3, ..., and step d.)

Algorithm 23.2 A	generic	description	of actions	by .	P_0	in steps	2, 3,	$\cdots, d.$
------------------	---------	-------------	------------	------	-------	----------	-------	--------------

<u>begin</u>

$$\begin{split} d &:= \log_2 p \\ \underline{\text{for}} \ k = 1 \ \underline{\text{to}} \ d \\ P_0 \ \text{receives data from an appropriate} \\ & \text{neighbor;} \\ P_0 \ \text{updates an appropriate portion of data;} \\ P_0 \ \text{sends the updated data from this portion} \\ & \text{to an appropriate neighbor;} \\ \underline{\text{end for}} \end{split}$$

end

d is the hypercube dimension iterate on d portions of data communication cost is masked if data have arrived when needed

For a hypercube of dimension d, step d + 1 is the last step.

Algorithm 23.3 A generic description of actions by	y P_0 in step d+1 – the last step.
<u>begin</u>	
$d := \log_2 p$	d is the hypercube dimension
$\underline{\mathbf{for}}\ k = 1\ \underline{\mathbf{to}}\ d$	iterate on d blocks of data
P_0 receives data from an appropriate	communication cost is masked if
neighbor;	data have arrived when needed
P_0 updates an appropriate portion of data;	
end for	
end	

Figure 23.15 Pipelining subcube-doubling "send" from P_0 on all channels (p = 8).



Observe from Figure 23.15 that the data mapping has been partially constructed in the process of developing the algorithm for P_0 . For p = 8, one only needs to follow the actions of P_7 , which mirrors that of P_0 , and the data mapping is completed in Figure 23.16.

Figure 23.16 Pipelining subcube-doubling "send" from P_7 on all channels (p = 8).







With the entire map constructed in Figure 23.16, one can now visualize the actions of any processor from the map. Keep in mind that all processors perform the same actions (pairing up with appropriate neighbors) "concurrently." As one more example, the actions by P_1 are shown in Figure 23.17.

Figure 23.17 Pipelining subcube-doubling "send" from P_1 on all channels (p = 8).



It should be understood that the generic description of the d + 1 steps of the algorithm given in this section is not tailored to P_0 's actions at all, but instead reflects the concurrent actions of all p processors. Note that in order to mask the communication cost, the matrix must be sufficiently large so that each processor can be kept busy computing before the message it waits for arrives. An analytical model was used in [22] to derive the minimum size of such a matrix, which, as expected, is a function of the number of processors and the hardware parameters of the machine being used. The method described above for computing multiple 1D FFTs may be viewed as employing an "all processor-to-all neighbor" communication scheme depicted in Figure 23.18. Although the simplest problem of computing many 1D FFTs is used in this section to make various aspects of this communication scheme easily understood, the method is not designed and should not be used for this simple case—because the "independent processor" method incurs no communication at all. However, this method is useful for FFT of higher dimensions, which is revisited in the next section.



23.7 Changing Data Mappings During Parallel 2D FFT Computation

The data mappings required in implementing the following four methods for computing the 2D FFT are depicted in Figures 23.19–23.22.

• The TS (*transpose split*) method: Two different data mappings are required in phases I and II (see Figure 23.19). Note that by distributing consecutive rows to the processors in Phase II, one has effectively transposed the data matrix as desired. Accordingly, an efficient parallel algorithm for changing the data mapping is an efficient parallel algorithm for matrix transposition, and such an algorithm will be presented in the next section.

Figure 23.19 The TS method: Different data mappings used in phases I and II (p = 8).



- The LD (*local distributed*) method: Identical data mappings are used in phases I and II (see Figure 23.20).
- The GBLK (generalized block distributed) method: Identical data mappings are used in phases I and II (see Figure 23.21).
- Calvin's method [22]: Two different data mappings are required in phases I and II (see Figure 23.22). This method appears to be called the LD method with overlap in [22]. It is not clear how the cost for changing the mapping can be masked from the very brief description in [22].

23.8 Parallel Matrix Transposition By Changing Data Mapping

As indicated in the previous section, an efficient parallel algorithm for changing the mappings from distributing the matrix columns to distributing the matrix rows is an



Figure 23.21 The GBLK method: Identical data mappings used in phases I and II (p = 8).



Figure 23.20 The LD method: Identical data mappings used in phases I and II (p = 8).

Figure 23.22 Calvin's method: Different data mappings used in phases I and II (p = 8).



efficient parallel algorithm for matrix transposition. Observe from Figure 23.23 that a data mapping by columns may be viewed as distributing the $N_1 \times N_2$ matrix Aon a $1 \times p$ subcube-grid, and a data mapping by rows may be viewed as distributing the same matrix on a $p \times 1$ subcube-grid, where $p = 2^d$. That is, each row of the matrix is initially shared by $p = 2^d$ processors, and is finally stored in its entirely in a single processor. This objective can be accomplished by halving the number of processors on each row of the subcube-grid, and doubling the number of processors on each column of the subcube-grid in d steps. Observe from Figure 23.23 that each time the subcube-grid changes its dimensions this way, all that is required is that every processor exchange one half of its data with a directly connected processor. Accordingly, the total communication cost for transposing an $N_1 \times N_2$ matrix is exactly d concurrent exchanges of $\frac{1}{2} \frac{N_1 \times N_2}{p}$ complex numbers among all pairs of processors. (If desired, the initial column mapping can be restored by reversing the steps with the same communication cost.)

Note that the communication cost in Phase II of the LD (local distributed) method for a 2D FFT (without inter-processor data permutation) requires d concurrent exchanges of $\frac{N_1 \times N_2}{p}$ complex numbers [98], and the communication cost of this matrix transposition algorithm is one half of that amount. Therefore, if the initial column mapping needs not to be restored, the TS (transpose split) method incurs half the communication cost of the LD method; if the initial column mapping must be restored, the communication cost of the TS method becomes the same as that of the LD method.

23.9 Notes and References

As noted in Section 23.3.3, the performance of the 2D block distributed method is not affected by the *aspect ratio* of the subcube-grid for reasons identified there. However, this is not the case for many parallel matrix algorithms, i.e., *the choice of the*

Figure 23.23 Parallel matrix transposition by changing data mapping (p = 8).



Interpreting the hypercube as a 1-by-8 subcube-grid



Distributing matrix to p = 8 processors by columns



Interpreting the hypercube as a 2-by-4 subcube-grid



Interpreting the hypercube as a 4-by-2 subcube-grid



Interpreting the hypercube as a 8-by-1 subcube-grid











 P_0

P1 P2 P3 P4 P5

 P_7 Distributing matrix to p = 8 processors by rows.

 P_6

aspect ratio for the subcube-grid can severely impact the performance of parallel algorithms, and the subcube-grid is an important and versatile physical network topology. For example, Chu and George show in [26, 27, 29] that an optimal aspect ratio can be determined at run time for a class of fundamental numerical algorithms including Gaussian elimination with partial pivoting, QR factorization (with column pivoting [27]), Gauss-Jordan inversion, and multiple least squares updating algorithms. The significant net saving in execution time and storage usage gained from using an optimal subcube-grid was demonstrated by numerical experiments on iPSC/2 and iPSC/860 hypercubes in [26, 27, 29].

Furthermore, the authors reported in [27] the iPSC/2 and iPSC/860 execution times to demonstrate an efficient data relocation algorithm which dynamically changes the data mapping between the subcube-grids, and the same algorithm was used in the last section for changing the aspect ratio from $1 \times p$ to $p \times 1$, which effectively transposes the distributed matrix among the p processors as desired.

The interplay of optimal physical and logical network topologies in the design and implementation of parallel matrix algorithms was investigated further by Chu in [25].

Other interesting algorithms for computing the 2D FFTs include the class of vectorradix algorithms as well as the class of polynomial transform algorithms. The basic principles underlying these two classes of sequential algorithms were reviewed in [41], and their parallel implementation on hypercube and mesh machines was recently examined by Angelopoulos and Pitas in [2]. Readers are referred to [65, 66, 79, 80] for more details on the vector-radix algorithms, and [69, 70] on the polynomial transforms originally proposed by Nussbaumer for the computation of 2D cyclic convolutions.