1. Motivation. The applications of Fourier Transform in physics.

The Fourier transform has long been used for characterizing linear systems and for identifying the frequency components making up a continuous waveform. However, when the waveform is sampled or the system is to be analyzed on a digital computer, the discrete version of the Fourier transform must be used.

The DFT plays an important role in many scientific and technical applications, including time series and waveform analysis, solutions to linear partial differential equations, the convolution and the correlation of time series, digital signal processing and image filtering.

The DFT is a linear transformation that maps *N* regularly sampled points from a cycle of a periodic signal, like a sine wave, onto an equal number of points representing the frequency spectrum of the signal (so in fact it is a bridge between the time domain and the frequency domain).

2. Mathematical theory. Continuous Fourier Transform

Recall the formula for the Continuous Fourier Transform (I'm sure, that everybody has learnt it at school)

Forward Fourier Transform $H(f) = \int_{-\infty}^{+\infty} h(t) e^{2\pi i f t} dt$

Inverse Fourier Transform $h(t) = \int_{-\infty}^{+\infty} H(f) e^{-2\pi i f t} df$, where $i = \sqrt{-1}$

Here the uppercase $H(f)$ represents the frequency-domain function, the lowercase $h(t)$ is the time-domein function.

Discrete Fourier Transform

The analogous discrete Fourier transform pair that applies to sampled versions of these functions can be written in the following form.

Consider a finite time series, sampled at an interval Δ: $h =< h[0], h[1], ..., h[N-1] >$

$h[k] = h[t_k] = h[k\Delta]$, $k = 0,1...\text{N-1}$

The discrete Fourier transform of the sequence *h* is the sequence $H =< H[0], H[1], ..., H[N-1] >$, where $H[j] = \sum_{k=0}^{N-1} h[k] e^{2\pi i k j / N}$, $j = 0,1...\text{N-1}$

Denote $W_N = e^{2\pi i / N}$, where e is the base of natural logarithms. The powers of W_N used in an DFT computation are also known as twiddle factors.

Now we can rewrite the formulation of DFT.

$$H[j] = \sum_{k=0}^{N-1} h[k] W_N{}^{jk}$$

$$h[k] = \frac{1}{N}\sum_{j=0}^{N-1} H[j] W_N^{-jk}$$

The computation of each *H [j]* according to this equation requires N complex multiplications. Therefore, the sequential complexity of computing the entire sequence *H* of length N is $\Theta(N^2)$ (it means that the amount of operations is proportional to the N², so that it grows like N²).

3. Improvement in DFT-development. FFT.

In 1965, Cooley and Tukey devised an algorithm to compute the DFT of an N-point series in a less amount operations. The revolutionary algorithm by Cooley and Tukey and its variations are referred to as the fast Fourier transform (FFT). This algorithm is based on the one interesting property of DFT – it's symmetry.

Recall $W_N = e^{2\pi i/N} \Rightarrow W_N^{\,N} = 1, W_N^{\,N/2} = -1$

If we associate $h[k] \Leftrightarrow H[j]$, then due to the property of symmetry

$h[-k] \Leftrightarrow H[-j]$

$h[k+l] \Leftrightarrow W^{-lk}\mathrm{H[j]}$

$\mathrm{W}^{\mathrm{lk}} h[k] \Leftrightarrow H[j+l]$

4. Fast Fourier Transform – serial algorithm.

Several different forms of the FFT algorithm exist. I would like to consider it's the simplest form, the one-dimensional, unordered, radix-2 FFT. Parallel formulations of higher-radix and multidimensional FFTs are similar to the simple algorithm because the underlying ideas behind all sequential FFT algorithms are the same.

Let's consider the computation of H[j] component.

Assume that N is a power of two: $N = 2^r$

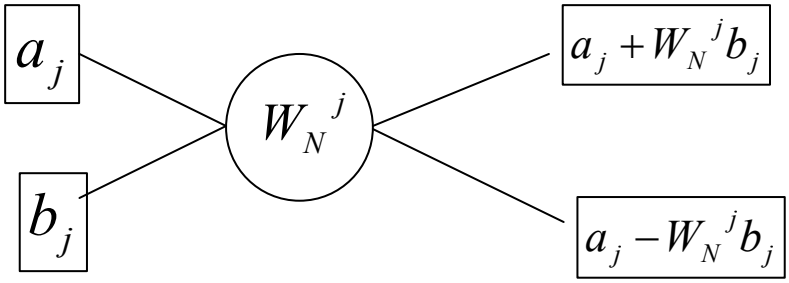
The FFT algorithm is based on the following step that permits an N-point DFT computation to be split into two (N/2)-point DFT computations:

$$H[j] = \sum_{k=0}^{N/2-1} h[2k]\mathrm{W}_{\mathrm{N}}^{\,2kj} + \mathrm{W}^{\mathrm{j}} \sum_{k=0}^{N/2-1} h[2k+1] W_N^{\,2kj}$$

Consider $H[j+N/2] = \sum_{k=0}^{N/2-1} h[2k]\mathrm{W}_{\mathrm{N}}^{\,2kj} - \mathrm{W}^{\mathrm{j}} \sum_{k=0}^{N/2-1} h[2k+1] W_N^{\,2kj}$

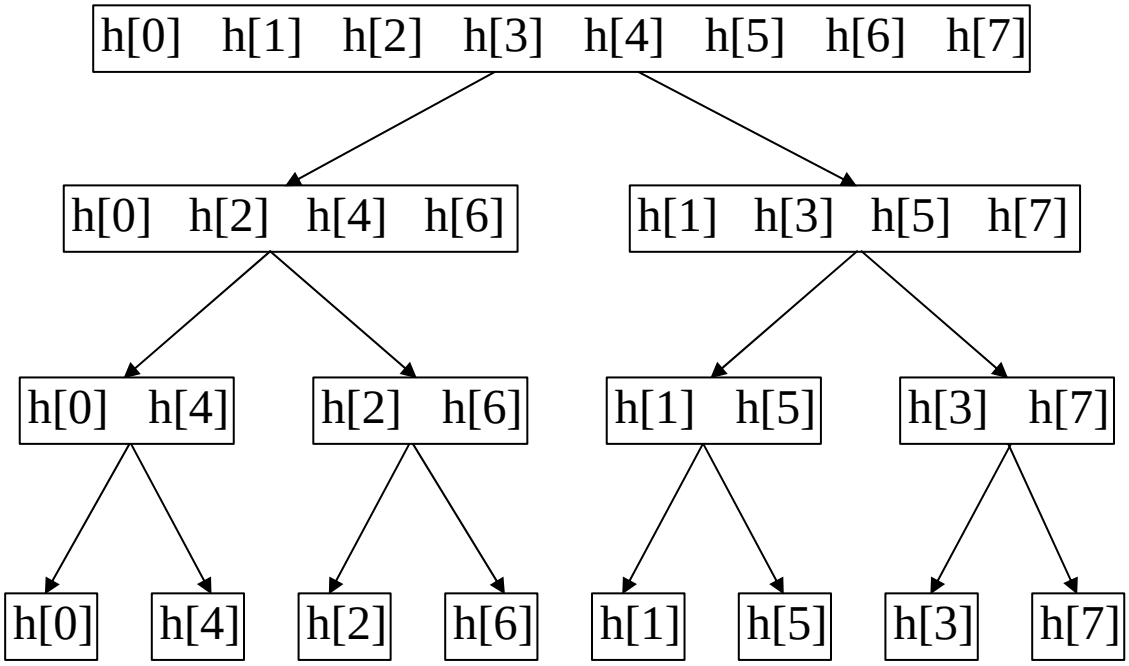
Why should we compute N operations for the first value and N operations for the second value, if we can compute each sum individually and then just to subtract or add them together?

This scheme is called butterfly.



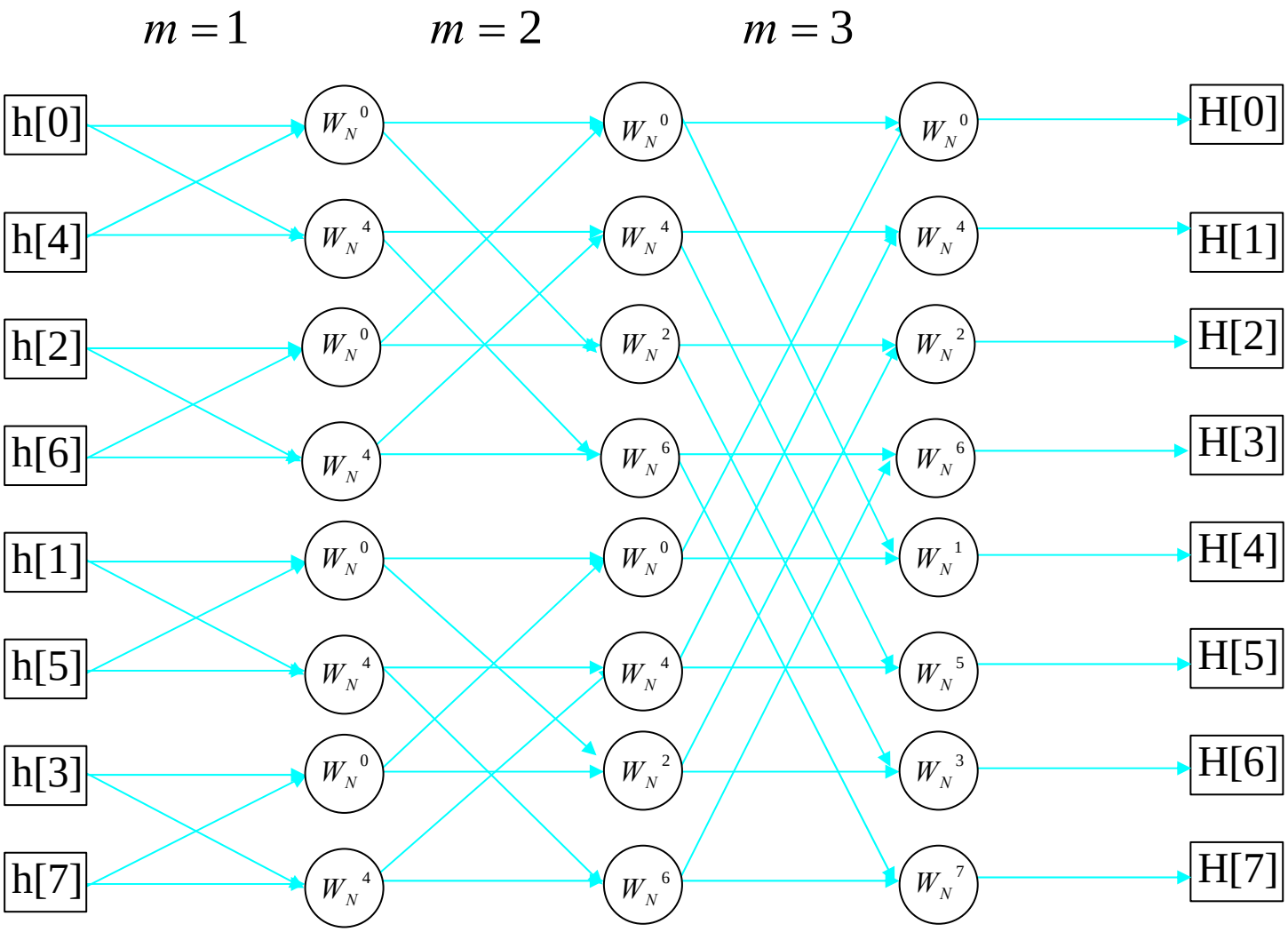
We can divide summations further and further and eventually (if *N* is a power of two) we can get each single summand. This leads to the recursive FFT algorithm. This FFT algorithm is called the radix-2 algorithm because at each level of recursion, the input sequence is split into two equal halves.

Let's consider how the recursive algorithm works on an 8-point sequence.



As we see, there is $\log_2 N$ levels of recursion (in these example $\log_2 N = 3$)

After this decomposition, using the butterfly-algorithm, we can compute the Fourier transform of our data.



The size of the input sequence over which an FFT is computed recursively decreases by a factor of two at each level of recursion. Hence, the maximum number of levels of recursion is $m = \log_2 N$ for an initial sequence of length *N*. The total number of arithmetic operations at each level is N and the overall sequential complexity of algorithm is ~($N \log_2 N$).

Thus, the overall complexity of the original DFT (of length N) is an order of N^2, of the FFT - $N \log_2 N$

As one can notice, the decomposition of time domain signal is nothing more then the reordering of the samples of the signal – so called bit reversal sorting algorithm.

Original sequence			Rearranged sequence	
decimal	binary		binary	decimal
0	000		000	0
1	001		100	4
2	010		010	2
3	011	⇒	110	6
4	100		001	1
5	101		101	5
6	110		011	3
7	111		111	7

Bit reversal does not affect the overall complexity of a parallel implementation of FFT.

4. Parallel FFT-algorithms

Due to wide application of FFT in scientific and engineering fields, there has been a lot of interest in implementing FFT on parallel computers. Nowadays, several opportunities of parallelizing FFT have been found, however this issue hasn't been investigated completely. I am going to talk about two parallel formulations of FFT: the binary-exchange algorithm and the transpose algorithm.

The Binary-Exchange algorithm

1. Decomposition is induced by partitioning the input vector => each task starts with one element of the input vector and computes the corresponding element of the output vector.

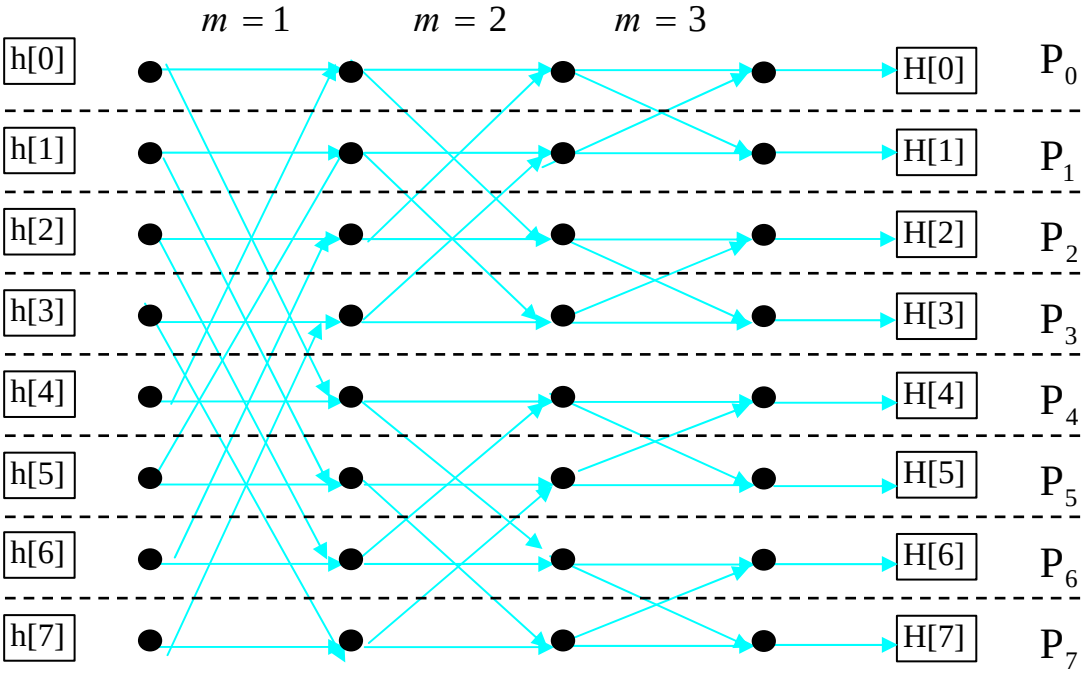
2. Assign to each task the same label as the index of its input element => in every iteration of the algorithm exchange of data takes place between those pairs of task, whose labels differ in one bit position. I'll explain it more detailed some time later.

4.1 The Binary-Exchange algorithm (a full bandwidth network)

First, consider the case, when we have a full bandwidth network, it means, that we implement this algorithm on a parallel computer on which a bisection width is an order of p, p is a number of parallel processes. The algorithm will be described assuming a hypercube network. However, the performance and scalability analysis would be valid for any parallel computer with an overall simultaneous data-transfer capacity of O (p) (accurate to a constant).

One Task Per Process

Let's consider the simplest mapping in which one task is assigned to each process. $N = 2^r$



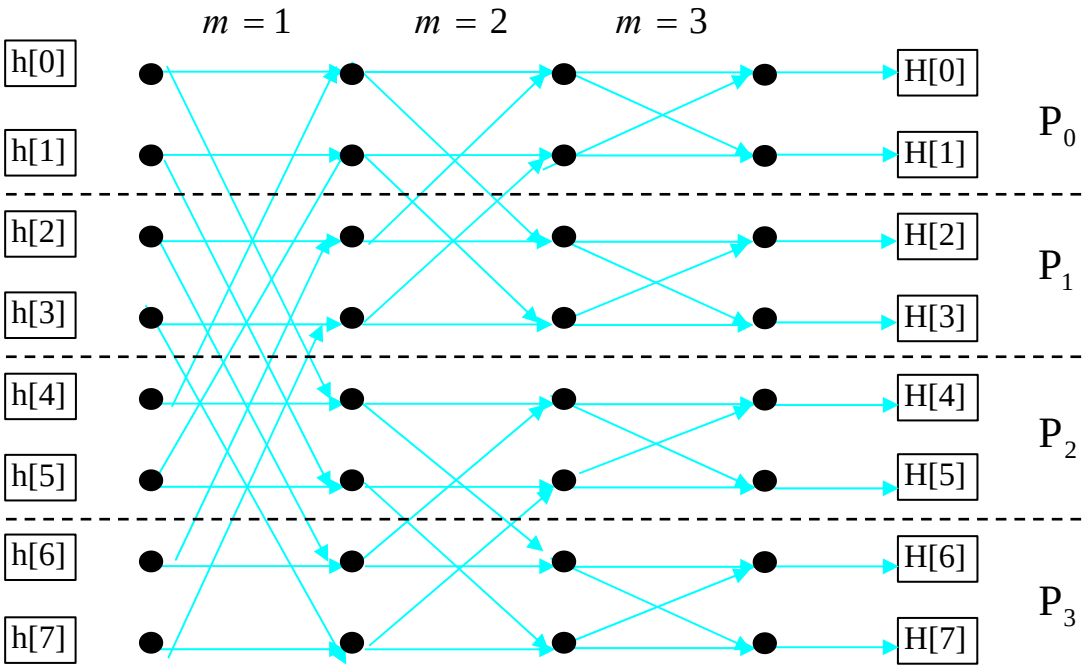
To perform the updates, process P_i requires an element from a process whose label differs from i at only one bit. Recall that in a hypercube, a node is connected to all those nodes whose labels differ from its own at only one bit position. Thus, the parallel FFT computation maps naturally onto a hypercube with a one-to-one mapping of processes to nodes.

In the first iteration, the labels of each pair of communicating processes differ only at their most significant bits. For instance, processes P_0 communicate with P_4. Similarly, in the second iteration, the labels of processes communicating with each other differ at the second most significant bit, and so on.

In each iteration of this algorithm, every process performs one complex multiplication and addition, and exchanges one complex number with another process. Thus, there is a constant amount of work per iteration. Hence, it takes time an order of $\log N$ to execute the algorithm in parallel by using a hypercube with N nodes.

Multiple Tasks Per Process

Let's now consider a mapping in which the N tasks of an N-point FFT are mapped onto p processes, where N> p. For the sake of simplicity, we will assume that both N and p are powers of two, i.e., $n = 2^r$ and $p = 2^d$, r and d - natural number. We partition the sequences into blocks of N/p contiguous elements and assign one block to each process.



As you can observe, the d most significant bits of the index of any element of the sequence are the binary representation of the label of the process that the element belongs to. This property of the mapping plays an important role in determining the amount of communication performed during the parallel execution of the FFT algorithm.

Elements with indices differing at their d (in our example d= 2) most significant bits are mapped onto different processes. However, all elements with indices having the same d most significant bits are mapped onto the same process. Recall that in the first iteration, the labels of each pair of communicating processes differ only at their most significant bits. Similarly, in the second iteration, the labels of processes communicating with each other differ at the second most significant bit, and so on.

As a result, elements combined during the first *d* iterations belong to different processes, and pairs of elements combined during the last *(r - d)* iterations belong to the same processes. Hence, this parallel FFT algorithm requires interprocess interaction only during the first $d = \log p$ of the $\log n$ iterations. There is no interaction during the last *r - d* iterations. Furthermore, in each iteration from the first d, all the elements that a process requires come from exactly one other process.

Each interaction operation exchanges *N/p* words of data.

Denote:

t_s - is the latency or the startup time for the data transfer

t_w - is the per-word transfer time, which is inversely proportional to the available bandwidth between the nodes

t_c - is the time of a complex multiplication and addition pair

Therefore, the time spent in communication in the entire algorithm is $t_s \log p + t_w(n/p) \log p$. A process updates *n/p* elements during each of the $\log n$ iterations. If a complex multiplication and addition pair takes time t_c, then the parallel run time of the binary-exchange algorithm for n-point FFT on a p-node hypercube network is

$$T_p = t_c \frac{N}{p} \log_2 N + t_s \log_2 p + t_w \frac{N}{p} \log_2 p$$

Speedup

$$S = \frac{t_c N \log_2 N}{T_p} = \frac{pN \log_2 N}{N \log N + (t_s/t_c) p \log_2 p + (t_w/t_c) N \log_2 p}$$

Efficiency

$$E = \frac{1}{1 + (t_s p \log_2 p)/(t_c N \log_2 N) + (t_w \log_2 p)/(t_c \log_2 N)}$$

Hence, knowing these parametres of our parallel computer, we can estimate the time of computation, speedup and efficiency.

Scalability Analysis

There are two observations:

1. For a given problem size (problem size is the number of basic computation steps in the best sequential algorithm to solve the problem on a single processing element), as we increase the number of processing elements, the overall efficiency of the parallel system goes down. This phenomenon is common to all parallel systems.
2. In many cases, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.

Following from these two observations, we define a scalable parallel system as one in which the efficiency can be kept constant as the number of processing elements is increased, provided that the problem size is also increased. It is useful to determine the rate at which the problem size

must increase with respect to the number of processing elements to keep the efficiency fixed. For different parallel systems, the problem size must increase at different rates in order to maintain a fixed efficiency as the number of processing elements is increased. This rate determines the degree of scalability of the parallel system.

In our case, the overall isoefficiency function is given by: $W = \max\left\{ p\log_2 p, K\frac{t_s}{t_c} p\log_2 p, K\frac{t_w}{t_c} p^{Kt_w/t_c}\log_2 p \right\}$ where $K = \frac{E}{1-E}$.

Assume that $t_c = 2, t_w = 4, t_s = 25$

Figure shows the isoefficiency curves given by this function for *E = 0.20, 0.25, 0.30, 0.35, 0.40*, and *0.45*. The asymptotic isoefficiency functions for *E = 0.20, 0.25*, and *0.30* are $\Theta(p\log p)$. The isoefficiency function for *E = 0.40* is $\Theta(p^{1,33}\log p)$, and that for *E = 0.45* is $\Theta(p^{1,64}\log p)$.

Isoefficiency functions of the binary-exchange algorithm on a hypercube with t_c = 2, t_w = 4, and t_s = 25 for various values of E.

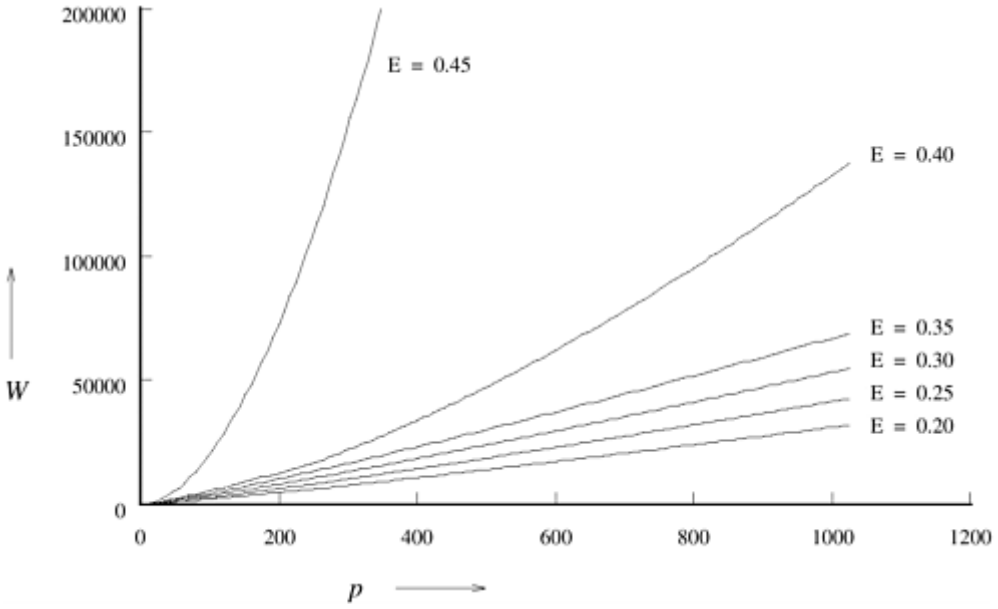
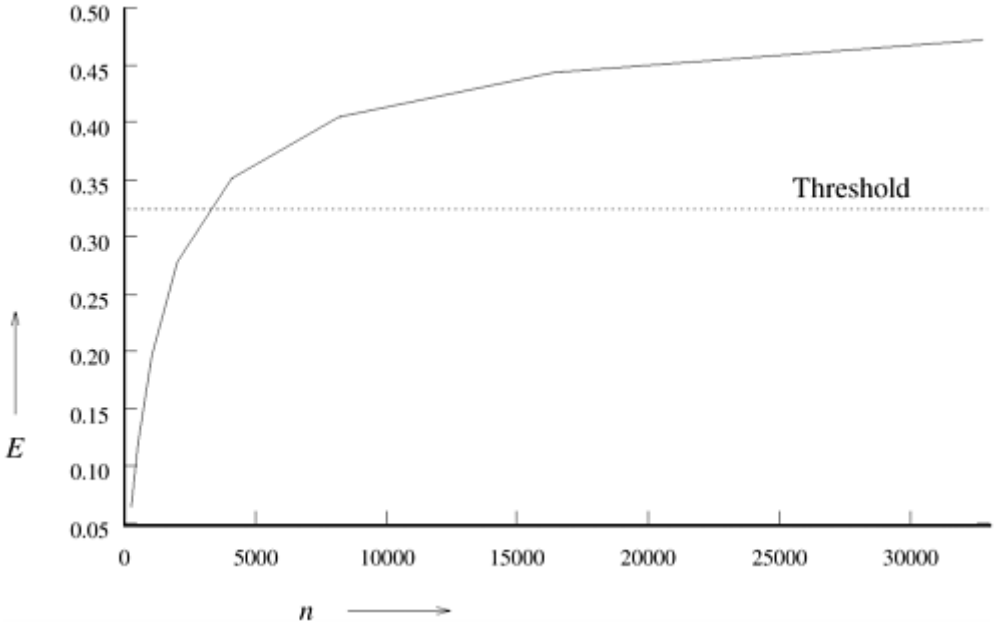


Figure 13.6 shows the efficiency curve of n-point FFTs on a 256-node hypercube with the same hardware parameters. The efficiency *E* is computed for various values of *n*, when *p=256*. The figure shows that the efficiency initially increases rapidly with the problem size, but the efficiency curve flattens out beyond the threshold.

Figure 13.6. The efficiency of the binary-exchange algorithm as a function of n on a 256-node hypercube with t_c = 2, t_w = 4, and t_s = 25.



As we can see there is a limit on the efficiency that can be obtained for reasonable problem sizes, and that the limit is determined by the ratio between the CPU speed and the communication bandwidth of the parallel computer being used. This limit can be raised by increasing the bandwidth of the communication channels. However, making the CPUs faster without increasing the communication bandwidth lowers the limit. Hence, the binary-exchange algorithm performs poorly on a parallel computer whose communication and computation speeds are not balanced. If the hardware is balanced with respect to its communication and computation speeds, then the binary-exchange algorithm is fairly scalable, and reasonable efficiencies can be maintained while increasing the problem size at the rate of □(p log p).

4.2 The Binary-Exchange algorithm (a limited bandwidth network)

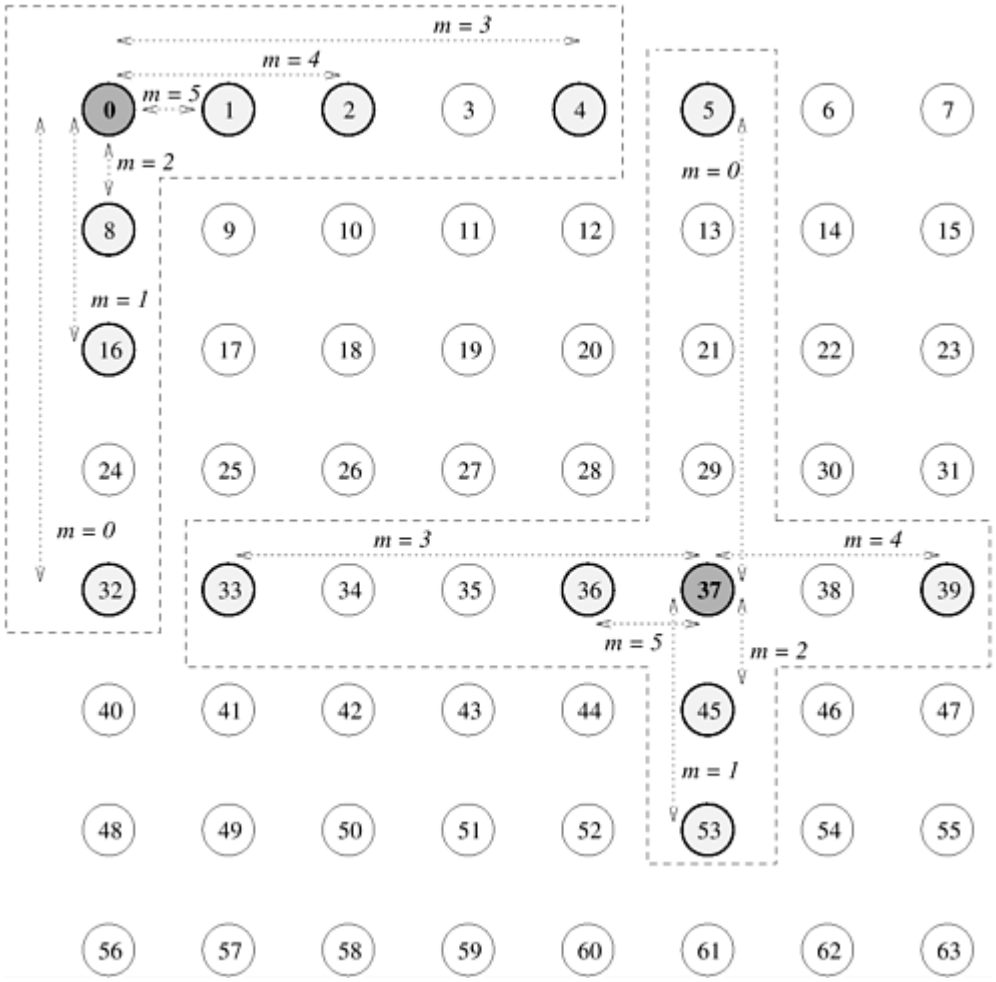
Now we consider the implementation of the binary-exchange algorithm on a parallel computer whose cross-section bandwidth is less than $\Theta(p)$. The algorithm and its performance characteristics will be illustrated using a mesh interconnection network. Assume that *n* tasks are mapped onto *p* processes running on a mesh with $\sqrt{p}$ rows and $\sqrt{p}$ columns, and that $\sqrt{p}$ is a power of two. Let $N = 2^r$ and $p = 2^d$. Also assume that the processes are labeled in a row-major fashion and that the data are distributed in the same manner as for the hypercube; that is, an element with index ($b_0 b_1 \cdots b_{r-1}$) is mapped onto the process labeled ($b_0 \cdots b_{d-1}$).

As in case of the hypercube, communication takes place only during the first $\log p$ iterations between processes whose labels differ at one bit.

What is the difference?

Unlike the hypercube, the communicating processes are not directly linked in a mesh. Consequently, messages travel over multiple links and there is overlap among messages sharing the same links. Figure 13.7 shows the messages sent and received by processes 0 and 37 during an FFT computation on a 64-node mesh. As the figure shows, process 0 communicates with processes 1, 2, 4, 8, 16, and 32. Note that all these processes lie in the same row or column of the mesh as that of process 0. Processes 1, 2, and 4 lie in the same row as process 0 at distances of 1, 2, and 4 links, respectively. Processes 8, 16, and 32 lie in the same column, again at distances of 1, 2, and 4 links. More precisely, in $\log\sqrt{p}$ of the log p steps that require communication, the communicating processes are in the same row, and in the remaining $\log\sqrt{p}$ steps, they are in the same column.

Figure 13.7. Data communication during an FFT computation on a logical square mesh of 64 processes. The figure shows all the processes with which the processes labeled 0 and 37 exchange data.



Since a process performs *n/p* such calculations in each of the $\log n$ iterations, the overall parallel run time is given by the following equation:

$$
\begin{aligned}
T_P &= t_c \frac{n}{p} \log n + 2 \sum_{m=0}^{d/2-1} \left(t_s + t_w \frac{n}{p} 2^m \right) \\
&= t_c \frac{n}{p} \log n + 2 \left(t_s \log \sqrt{p} + t_w \frac{n}{p} (\sqrt{p} - 1) \right) \\
&\approx t_c \frac{n}{p} \log n + t_s \log p + 2 t_w \frac{n}{\sqrt{p}}
\end{aligned}
$$

The speedup and efficiency are given by the following equations:

$$
\begin{aligned}
S &= \frac{t_c n \log n}{T_P} \\
&= \frac{pn \log n}{n \log n + (t_s/t_c) p \log p + 2(t_w/t_c) n \sqrt{p}} \\
E &= \frac{1}{1 + (t_s p \log p)/(t_c n \log n) + 2(t_w \sqrt{p})/(t_c \log n)}
\end{aligned}
$$

4.3 The transpose algorithm

Another parallel algorithm involves matrix transposition, and hence is called the transpose algorithm.

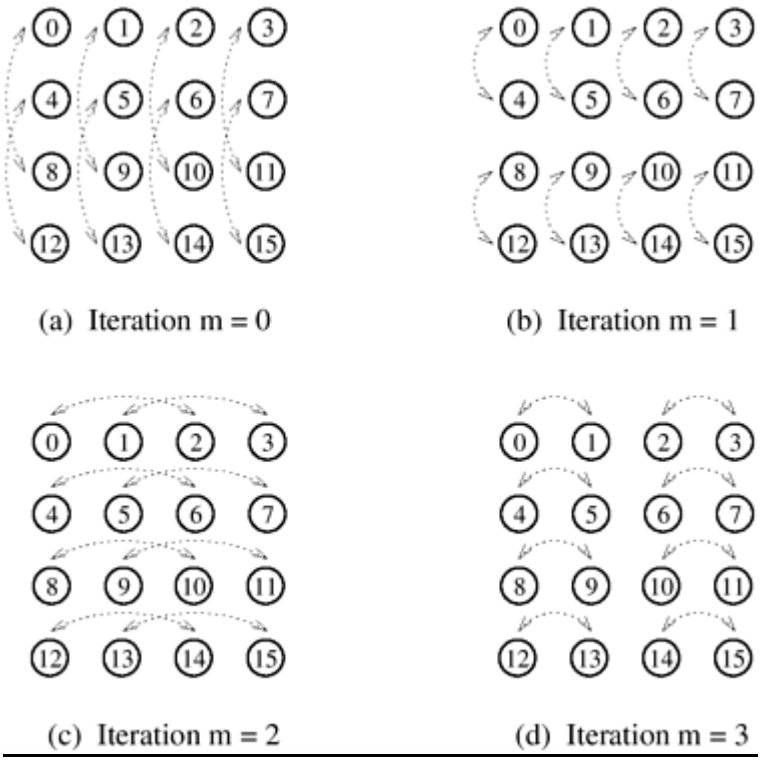
The transpose algorithm is particularly useful when the ratio of communication bandwidth to CPU speed is low and high efficiencies are desired. On a hypercube or a p-node network with Θ(p) bisection width, the transpose algorithm has a fixed asymptotic isoefficiency function of □(p^2 log p). That is, the order of this isoefficiency function is independent of the ratio of the speed of point-to-point communication and the computation.

Two-dimentional transpose algorithm

The simplest transpose algorithm requires a single transpose operation over a two-dimensional array; hence, we call this algorithm the two-dimensional transpose algorithm.

Assume that $\sqrt{n}$ is a power of 2, and that the sequences of size n are arranged in a $\sqrt{n} \times \sqrt{n}$ two-dimensional square array, as shown in the picture for *n = 16*. Recall that computing the FFT of a sequence of n points requires log n iterations. If the data are arranged as shown in Figure 13.8, then the FFT computation in each column can proceed independently for $\log\sqrt{n}$ iterations without any column requiring data from any other column. Similarly, in the remaining $\log\sqrt{n}$ iterations, computation proceeds independently in each row without any row requiring data from any other row. The figure illustrates that if data of size *n* are arranged in a $\sqrt{n} \times \sqrt{n}$ array, then an n-point FFT computation is equivalent to independent $\sqrt{n}$-point FFT computations in the columns of the array, followed by independent $\sqrt{n}$-point FFT computations in the rows.

Figure 13.8. The pattern of combination of elements in a 16-point FFT when the data are arranged in a 4 x 4 two-dimensional square array.



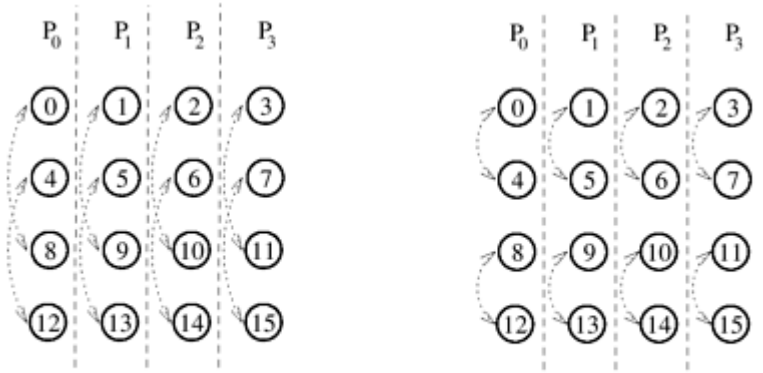
If the $\sqrt{n} \times \sqrt{n}$-array of data is transposed after computing the $\sqrt{n}$-point column FFTs, then the remaining part of the problem is to compute the $\sqrt{n}$-point columnwise FFTs of the transposed

matrix. The transpose algorithm uses this property to compute the FFT in parallel by using a columnwise striped partitioning to distribute the $\sqrt{n} \times \sqrt{n}$ -array of data among the processes. For instance, consider the computation of the 16-point FFT, where the 4×4 -array of data is distributed among four processes such that each process stores one column of the array.

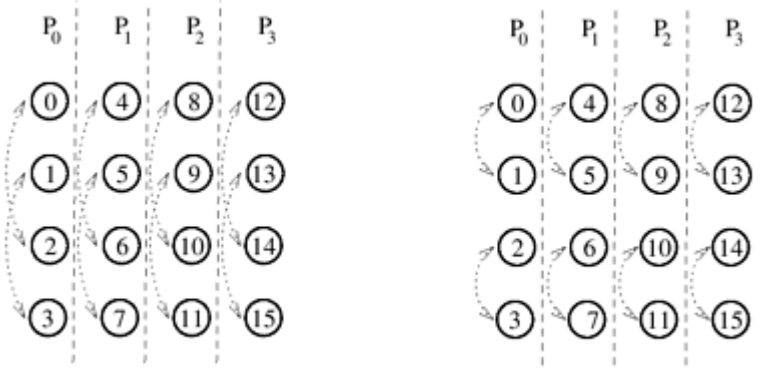
In general, the two-dimensional transpose algorithm works in three phases.

1) A $\sqrt{n}$ -point FFT is computed for each column of the initial array.
2) The array of data is transposed.
3) A $\sqrt{n}$ -point FFT is computed for each column of the transposed array.

Figure shows that the first and third phases of the algorithm do not require any interprocess communication. In both these phases, all $\sqrt{n}$ points for each columnwise FFT computation are available on the same process. Only the second phase requires communication for transposing $\sqrt{n} \times \sqrt{n}$ -the matrix.



(a) Steps in phase 1 of the transpose algorithm (before transpose)



(b) Steps in phase 3 of the transpose algorithm (after transpose)

In this transpose algorithm, one column of the data array is assigned to one process. Before analyzing the transpose algorithm further, consider the more general case in which p processes are used and $1 \leq p \leq \sqrt{n}$. The $\sqrt{n} \times \sqrt{n}$ -array of data is striped into blocks, and one block of $\frac{\sqrt{n}}{p}$ rows is assigned to each process. In the first and third phases of the algorithm, each process computes $\frac{\sqrt{n}}{p}$ FFTs of size $\sqrt{n}$ each. The second phase transposes the $\sqrt{n} \times \sqrt{n}$ -matrix, which is distributed among *p* processes with a one-dimensional partitioning. Such a transpose requires an all-to-all personalized communication.

Now we derive an expression for the parallel run time of the two-dimensional transpose algorithm. The only inter-process interaction in this algorithm occurs when the $\sqrt{n} \times \sqrt{n}$ array of data partitioned along columns or rows and mapped onto p processes is transposed. The parallel run time of the transpose algorithm on a hypercube or any □(p) bisection width network is given by the following equation:

$$T_p = t_c \frac{N}{p} \log_2 N + t_s (p-1) + t_w \frac{N}{p}$$

The expressions for speedup and efficiency are as follows:

$$S \approx \frac{pN \log_2 N}{N \log_2 N + (t_s / t_c) p^2 + (t_w / t_c) N}$$

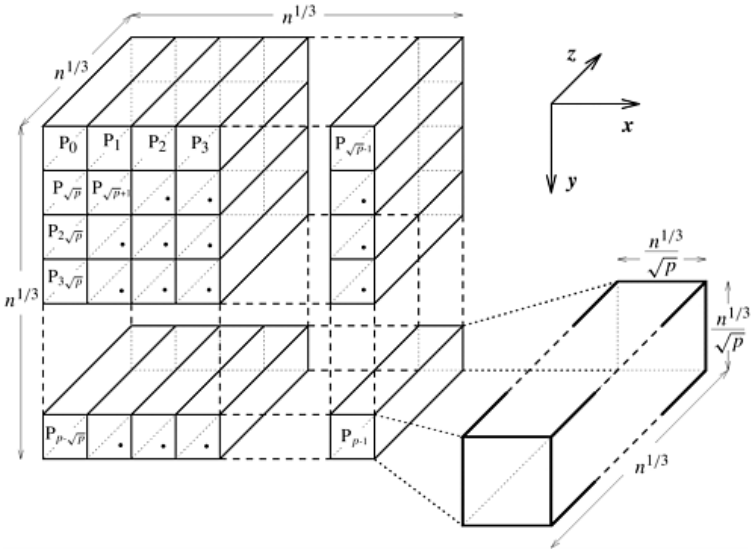
$$E \approx \frac{1}{1 + (t_s p^2) / (t_c N \log_2 N) + t_w / (t_c \log_2 N)}$$

Comparison with the Binary-Exchange Algorithm

The transpose algorithm has a much higher overhead than the binary-exchange algorithm due to the message startup time t_s, but has a lower overhead due to per word transfer time t_w. As a result, either of the two algorithms may be faster depending on the relative values of t_s and t_w. If the latency t_s is very low, then the transpose algorithm may be the algorithm of choice. On the other hand, the binary-exchange algorithm may perform better on a parallel computer with a high communication bandwidth but a significant startup time.

The Generalized Transpose Algorithm

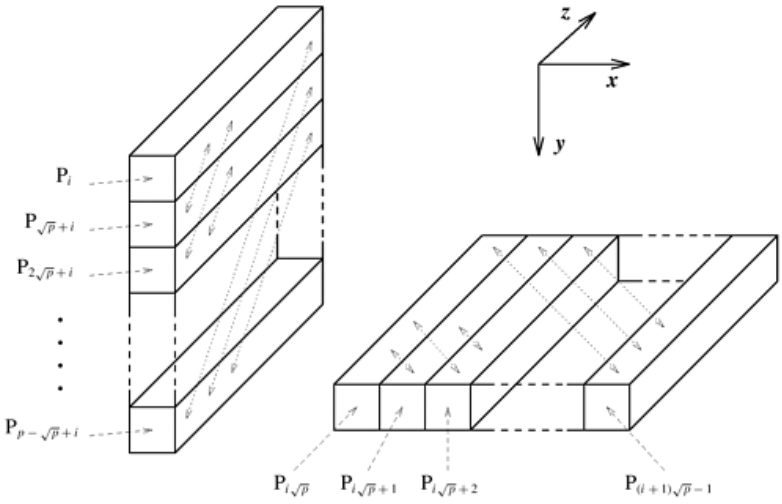
In the two-dimensional transpose algorithm, the input of size *n* is arranged in a $\sqrt{n} \times \sqrt{n}$ two-dimensional array that is partitioned along one dimension on p processes. These processes, irrespective of the underlying architecture of the parallel computer, can be regarded as arranged in a logical one-dimensional linear array. As an extension of this scheme, consider the n data points to be arranged in an $n^{1/3}$ x $n^{1/3}$ x $n^{1/3}$ three-dimensional array mapped onto a logical $\sqrt{p} \times \sqrt{p}$ two-dimensional mesh of processes. simplify the algorithm description, we label the three axes of the three-dimensional array of data as x, y, and z. In this mapping, the x-y plane of the array is checkerboarded into $\sqrt{p} \times \sqrt{p}$ parts. Thus, each process has $(n^{1/3}/\sqrt{p}) \times (n^{1/3}/\sqrt{p}) \times n^{1/3} = n/p$ elements of data.



In this case, $n^{1/3}$-point FFTs are computed over the elements of the columns of the array in all three dimensions, choosing one dimension at a time. We call this algorithm the three-dimensional transpose algorithm. This algorithm can be divided into the following five phases:

1. In the first phase, $n^{1/3}$-point FFTs are computed on all the rows along the z-axis.
2. In the second phase, all the $n^{1/3}$ cross-sections of size $n^{1/3}$ x $n^{1/3}$ along the y-z plane are transposed.
3. In the third phase, $n^{1/3}$-point FFTs are computed on all the rows of the modified array along the z-axis.
4. In the fourth phase, each of the $n^{1/3}$ x $n^{1/3}$ cross-sections along the x-z plane is transposed.
5. In the fifth and final phase, $n^{1/3}$-point FFTs of all the rows along the z-axis are computed again.

The communication (transposition) phases in the three-dimensional transpose algorithm



(a) Transpose in the i^{th} column of processors during phase 2

(b) Transpose in the i^{th} row of processors during phase 4

Parallel run-time

$$\mathrm{T_p} = t_c \frac{N}{p} \log_2 N + 2\mathrm{t_s}(\sqrt{p} - 1) + 2t_w \frac{N}{p}$$

Speedup

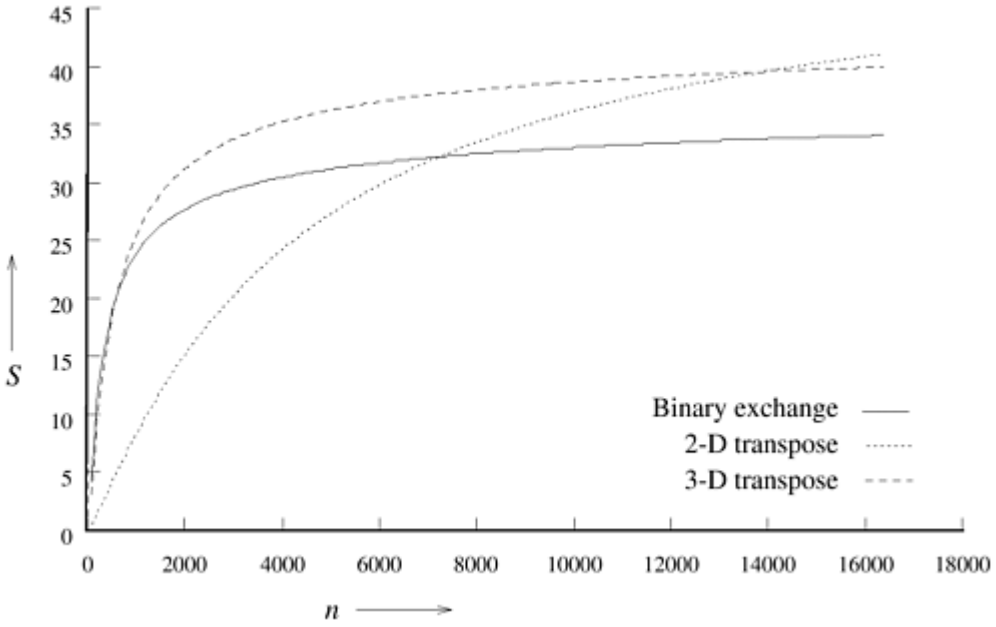
$$S \approx \frac{pN\log_2 N}{N\log_2 N + 2(t_s/t_c)p(\sqrt{p}-1) + 2(t_w/t_c)N}$$

4.5 Comparison of described algorithms

The binary-exchange algorithm and the two-dimensional transpose algorithms can be regarded as two extremes. The former minimizes the overhead due to t_s but has the largest overhead due to t_w. The latter minimizes the overhead due to t_w but has the largest overhead due to t_s . The variations of the transpose algorithm for 2 < q < log p lie between these two extremes. For a given parallel computer, the specific values of t_c, t_s , and t_w determine which of these algorithms has the optimal parallel run time

Note that, from a practical point of view, only the binary-exchange algorithm and the two- and three-dimensional transpose algorithms are feasible. Higher-dimensional transpose algorithms are very complicated to code. Moreover, restrictions on n and p limit their applicability.

A comparison of the speedups obtained by the binary-exchange, 2-D transpose, and 3-D transpose algorithms on a 64-node hypercube with $t_c = 2$, $t_w = 4$, and $t_s = 25$.



The figure shows that for different ranges of n, a different algorithm provides the highest speedup for an n-point FFT. For the given values of the hardware parameters, the binary-exchange algorithm is best suited for very low granularity FFT computations, the 2-D transpose algorithm is best for very high granularity computations, and the 3-D transpose algorithm's speedup is the maximum for intermediate granularities.

To sum up, the binary-exchange algorithm yields good performance on parallel computers with sufficiently high communication bandwidth with respect to the processing speed of the CPUs. Efficiencies below a certain threshold can be maintained while increasing the problem size at a moderate rate with an increasing number of processes. However, this threshold is very low if the communication bandwidth of the parallel computer is low compared to the speed of its processors.