Hiding Communication Costs in Bandwidth-Limited Parallel FFT Computation

Abhijit Sahay

Computer Science Division,
University of California, Berkeley

January 27, 1993

Abstract

This paper presents a novel computation schedule for FFT-type computations on a bandwidth-limited parallel computer. Using P processors, we are able to process an n-input FFT graph in the optimal time of $\frac{n \log n}{P}$ by carefully interleaving inter-processor communication steps with local computation. Our algorithm is suitable for both shared-memory and distributed memory machines and is analyzed in a simplification of the LogP model [5] suitable for studying bandwidth-limited parallel machines. Our parallel FFT algorithm incorporates several techniques that have long been used by parallel programmers to reduce communication costs and our analysis provides theoretical justification for the success of these techniques in the context of highly structured computations like FFTs. At another level, our algorithm can be viewed as an optimal simulation of large butterfly networks on arbitrary machines (as modeled under LogP.) Thus, we argue that computations thought to be inherently suited to butterfly networks can be executed with no loss in efficiency on arbitrary bandwidth-limited networks, given sufficient slack.

1 Introduction

The problem of computing the Fourier transform of a vector of complex numbers has numerous scientific applications and has been extensively studied [8, 4]. The Fast Fourier Transform (FFT) algorithm for the problem [3] is one of the most widely used algorithms in computer science. In addition to direct implementations of the FFT used in fields such as signal processing and computer algebra, the computational dependence graph of the FFT (also called the *butterfly* graph) has been used in the interconnection network for parallel machines due to its rich symmetries and scalability properties.

In this paper, we study parallelization of FFT-type computations, by which we mean computational tasks for which the dependence between subtasks can be represented by the butterfly. Such computations include direct FFTs, applications of FFTs for polynomial evaluation and integer multiplications [7, 14], bitonic sorting [2] and general computations on butterfly networks [10, 12]. We consider scheduling the n-input butterfly on a P-processor parallel machine in which interprocessor communication is costly. On real machines, the high cost of communication arises from three sources: network overhead, transit delay and limited bandwidth. If an algorithm requires infrequent communication of long messages (as is the case with our parallel FFT algorithm) the overhead and latency costs are generally small since they are incurred only once at every transmission. (For shared memory machines, these costs are negligible regardless of the communication pattern.) Bandwidth limitations impose a greater cost for both message passing and shared memory machines since they imply a cost per byte of data sent and these cannot be mitigated by pipelining or changing the size of message packets.

Accordingly, our primary focus is on minimizing bandwidth-induced communication costs. The specific model used for analyzing our algorithm is a slight simplification of the LogP model proposed by Culler et al [5]. This model characterizes a parallel computer by four parameters that abstractly capture computational bandwidth, and the bandwidth, latency and overhead of communication. In keeping with the authors' suggestion for bandwidth-constrained networks, we drop the overhead parameter for our analysis. Analysis in the complete model is a routine extension of the work presented here and offers no new insights.

Other models that incorporate bandwidth considerations such as Valiant's BSP model [15] or the LPRAM model of Aggarwal et al [1] emphasize the importance of reducing communication but not that of scheduling it carefully. The former assumes the presence of an intelligent router capable of routing h-relations (communication patterns in which no processor sends or receives more than h messages) in a contention-free manner and the latter accounts only for the number of communication steps. In real parallel programming, performance is often enhanced by paying attention to the communication schedule [5] and we use the LogP model for our analysis since it requires algorithms to specify their communication schedules explicitly.

1.1 Main Result

Our main contribution is a novel algorithm for partitioning the $n \log n$ nodes of the n-input butterfly among P processors, and for interleaving the computation and communication steps resulting from this allocation, so as to achieve a running time of $\frac{n \log n}{P}$ (and hence, *ideal speedup*) even in the presence of substantial bandwidth limitations. Thus, we are able to completely hide the communication cost of parallelization. Our algorithm uses $O(n)$ messages altogether which is optimal. Since the computation to communication ratio is logarithmic, it is not surprising that our algorithm works better for larger problem sizes. The smallest problem size for which our algorithm attains ideal speedup is large relative to P but not unreasonable for a large class of current machines which have small numbers of extremely powerful processors. Even when n is moderate, our schedule yields a speedup of at least $\frac{P}{1+1/(\log n)}$.

The only known result on ideal speedup of FFT computations is for the PRAM model where $P \leq n$ processors can be used with a speedup of P. For models that include communication costs, speedup of $P/2$ using P processors was demonstrated by Papadimitriou and Yannakakis for their delay model [11] (which incorporates network latency but does not address bandwidth requirements.) Essentially the same algorithm has been shown to yield $P/2$ speedup for the BSP [15] and the LPRAM [1].

Even though the FFT and similar algorithms have been implemented on a variety of parallel computers and even though techniques embodied in our algorithm – balanced data and load distribution, pipelining, overlapping communication and computation – have been used extensively by parallel programmers [6, 9], attempts to quantify the savings made possible by such techniques have been limited. Our results can be viewed as a justification of these methods. Similar analyses of matrix computations appears in [13].

Finally, our algorithm represents a general purpose simulation of butterfly networks on arbitrary machines as modeled under LogP. We thus show that a parallel machine with a small number of powerful processors can be used without any loss in performance for the simulation of specialized networks even if the host machine has low communication bandwidth.

The rest of this paper is organized as follows. Section 2 defines the problem and specifies the variant of LogP that we use in the analysis. Section 3 describes the data allocation and load distribution rules used by our algorithm and presents the analysis for a naive computation schedule. Section 4 describes the main algorithm and proof of its optimality. Section 5 refines the analysis for small n and presents a precise bound on the running time for cases where the speedup is less than P. Section 6 contains additional comments and concluding remarks.

2 The Problem and The Model

The n-input butterfly is a directed graph with $n(\log n + 1)$ nodes arranged in n rows and $(\log n + 1)$ columns. [1] Nodes are labeled (r, c) with $0 \leq r < n$ and $0 \leq c \leq \log n$. For $0 \leq r < n$ and $0 \leq c < \log n$, node (r, c) has directed edges to nodes $(r, c+1)$ and $(\bar{r}_c, c+1)$ where r and $\bar{r}_c$ differ only in the $(c+1)$-th most significant bit of their binary representations. Figure 1 shows an 8-input butterfly.

The nodes in column 0 are the problem inputs and those in column $\log n$ represent the outputs of the computation. Each non-input node represents an indivisible computation that must be carried out on a single processor. Processing a node requires the results of the computations represented by the node's immediate predecessors; if any of these are carried out at a different processor, the computed value(s) must be communicated before the node can be processed. Our goal is to allocate the nodes of the butterfly among P processors and to design computation schedules for each processor so as to minimize the time at which the last output node is processed.

The model we use for analyzing our algorithms is a variant of the LogP model [5], a model which reflects the communication bottlenecks of real machines realistically. In this model of distributed-memory multiprocessors, processors communicate by point-to-point messages The model specifies the performance characteristics of the interconnection network, without specifying its structure. The main parameters of the model are:

P: the number of processor/memory modules.

[1] We assume throughout that n and P are powers of 2 and all logarithms are to base 2.

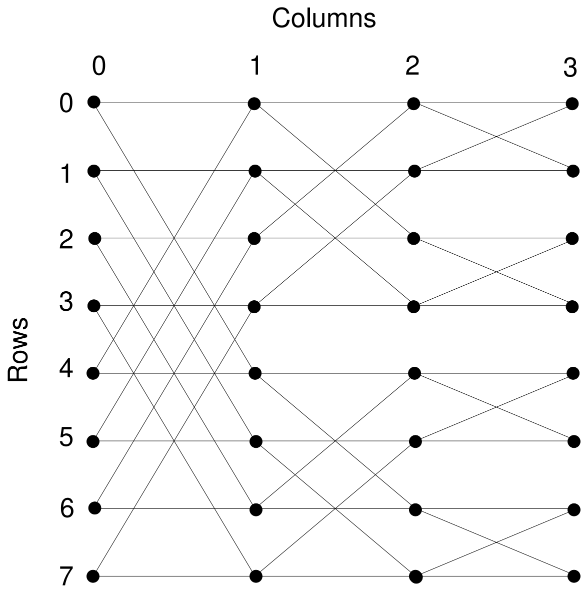

Figure 1: *An 8-input butterfly.*

L: an upper bound on the *latency*, or delay, incurred in communicating a message containing a word from its source module to its target module.

o: the *overhead*, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.

g: the *gap*, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth.

The model assumes that $g \geq 2o$, since if a processor is sending and receiving messages, a gap of $2o$ is enforced automatically between consecutive sends or receives. Furthermore, it is assumed that the network has a *finite capacity*, such that at most $\lceil L/g \rceil$ messages can be in transit from any processor or to any processor at any time. If a processor attempts to transmit a message that would exceed this limit, it stalls until the message can be sent without exceeding the capacity limit. Since L is only an upper bound the latency experienced by any message is unpredictable, and messages directed to a given target module may not arrive in the same order as they are sent. In order to be considered correct, an algorithm must produce correct results under all interleavings of messages consistent with the upper bound of L on latency. The algorithm we present does not violate the capacity constraints of the network and its running time can only improve if messages are received out of order. In the rest of the paper, therefore, we shall not mention network capacity and assume that each message incurs the maximum latency of L.

For many machines, the network bandwidth is small and such machines are modeled under LogP by having a large g. If g is large compared to o, a simpler 3-parameter version has been suggested in which in which the network overhead parameter is omitted (or set to 0), thus allowing processors to perform g units of computation between consecutive sends or receives. Of course, every real machine requires the processor's involvement in network transactions, precluding the possibility of such an ideal overlap. Nevertheless, the approximation is good for bandwidth-limited machines.

Finally, we assume without loss of generality that time is measured in units of butterfly operations, i.e. We assume that each butterfly node can be processed in unit time on any of the processors.

3 A Simple Algorithm

We assume that the processors are numbered $0, \ldots, P-1$ and that $n \geq P^2$. [2] Let $m = n/P$ and $l = m/P$. The following observations about the n-input butterfly are standard and will be important in understanding our algorithm:

- For each $j > 0$, the nodes in columns 0 through j comprise $n/2^j$ disjoint 2^j-input butterflies with the i-th butterfly being $A_i = \{(r, c) : \ r \equiv i \pmod{n/2^j}, 0 \leq c \leq j\}$.
- For each $k < \log n$ the nodes in columns k through $\log n$ comprise 2^k disjoint $\frac{n}{2^k}$-input butterflies with the i-th butterfly being $B_i = \{(r, k) : \ (i-1)\frac{n}{2^k} \leq r < i\frac{n}{2^k}, k \leq c \leq \log n\}$.

3.1 Processor Allocation

The processor assignment scheme for our algorithm is given by the following rules:

1. For $0 \leq c \leq \log m$, node (r, c) is assigned to processor $r \pmod P$. This yields a *cyclic* allocation in which row 0 is assigned to processor 0, row 1 to processor 1 and so on.
2. For $\log m < c \leq \log n$, node (r, c) is assigned to processor $\lfloor \frac{r}{m} \rfloor$. This yields a *blocked* allocation in which the first block of n/P rows are assigned to processor 0, the next to processor 1, etc.

Using the decomposition rules given above, it is easily verified that the nodes assigned to a processor by rule 1 (which we shall call the processor's Phase I computation load) comprise a single m-input butterfly and those assigned to it by rule 2 (referred to as its Phase II load) comprise the non-input nodes of l disjoint P-input butterflies. Moreover, for each of the P-input butterflies of Phase II, exactly one input node appears as an output in each of the P m-input butterflies of Phase I. Thus, the only communication requirement imposed by this load distribution is that the outputs of each processor's Phase I computation be evenly distributed among the P processors prior to Phase II computation. The entire computation thus requires each pair of processors to exchange l messages, resulting in a total of $n - m$ messages. It has been shown that any processor assignment scheme incurs $\Omega(n)$ messages [1], implying that, up to constant factors, our scheme is optimal with respect to total communication.

3.2 Computational Schedule

The computation schedule for our algorithm is simple: each processor processes the nodes assigned to it in increasing order of column number. With this schedule, each processor finishes its Phase I computation by time $m \log m$ at which time it can start sending its Phase I outputs to other processors. If processor i sends messages to processors $i+1, \ldots, i+P-1 \pmod P$ in order, each processor would have received all its messages in $L + (m - l - 1)g$ additional time steps. Finally, Phase II computation requires $m \log p$ steps, yielding the following:

Theorem 3.1 *In the LogP model with no overhead, an n-input butterfly can be processed in time* $\frac{n}{p} \log n + (\frac{n}{p} - \frac{n}{p^2} - 1)g + L$.

[2] The implication that each processor hold at least P data points is reasonable for all current parallel machines.

We see that, ignoring the additive contribution of latency, our simple algorithm is within a factor $(1 + \frac{g}{\log n})$ of optimal. It is easily verified that entirely cyclic or entirely blocked allocation would increase the number of messages by a factor of $\log P$. If we had chosen a bad communication schedule for exchanging messages, for example, one in which every processor sent messages to processor 0, then to processor 1, etc. the number of messages would stay the same but the communication time would be increased by a factor of P. Thus, even with this simple algorithm we see the virtue of careful load distribution, balanced communication patterns and pipelining of messages to hide network latency.

Our algorithm can be viewed as a simulation of the butterfly network on the LogP model. If each node of the butterfly represents a processor and in a single step, processors can compute locally or communicate with a neighbor, we see that with our processor allocation, a single step of the butterfly takes no more than $(n \log n + ng)/P$ time, which is close to optimal. The overlapped algorithm of the next section improves this to the absolutely optimal $n \log n/P$.

4 Hiding Communication Costs

The computation schedule of the algorithm just presented forces processors to idle while messages are in transit. It is clear that higher speedup would be achieved if message transmission could be overlapped with computation, another technique that is commonly employed in parallel programming. The scheme we propose now attempts to generate the messages to be sent as early as possible during Phase I computation. Unless n is extremely small, we expect that by the end of Phase I computation, a significant fraction of the communication would have been completed. This would allow Phase II computation to be started without any delay and reduce (or eliminate) the idling time for each processor. We make this idea more precise below.

4.1 Output-Driven Computation Schedules

Our new algorithm employs the same processor assignment scheme but an entirely different computation schedule for Phase I. The computation schedule for Phase II turns out to have little effect on the running time and we consider it in detail in Section 5. Recall that processor i's Phase I computation load comprises a single m-input butterfly, viz. the set $A_i = \{(ap + i, c) : 0 \leq a < m, 0 \leq c \leq \log m\}$. The computation schedule for processor i is determined by a permutation $\pi^{(i)}$ of the set $\{a : 0 \leq a < m\}$. Processor i processes its Phase I outputs in the order determined by $\pi^{(i)}$, always processing the smallest number of intermediate nodes required to generate the next output in the sequence. We call this computation schedule the *output-driven* schedule determined by $\pi^{(i)}$. As soon as an output node has been processed, the processor transmits it to the processor that requires it for its Phase II computation. The output permutations $\{\pi^{(i)}\}$ are chosen from a class of permutations that permit outputs to be generated at an optimal rate.

As an example, consider the case $m = 8, P = 4$ shown in Figure 2. Each processor has an 8-input butterfly in Phase I, the outputs of which are to be distributed among the 4 processors in blocks of 2. It is clear that the first output cannot be generated until time 7 since each output in column $\log m = 3$ has 2 predecessors in column 2, each of which has 2 distinct predecessors in column 1, for a total of 6 predecessors that must be processed before the output can be processed. However, the next output can be generated at time 8 and the next two at time 11 and 12, as shown in Figure 2, if the output permutation were, for example, (2 3 0 1 6 7 4 5). Of course, not all processors should be driven by this permutation, for this would lead to every processor first sending messages to processor 1, then to processor 0, then to processor 3 and finally to processor 2.

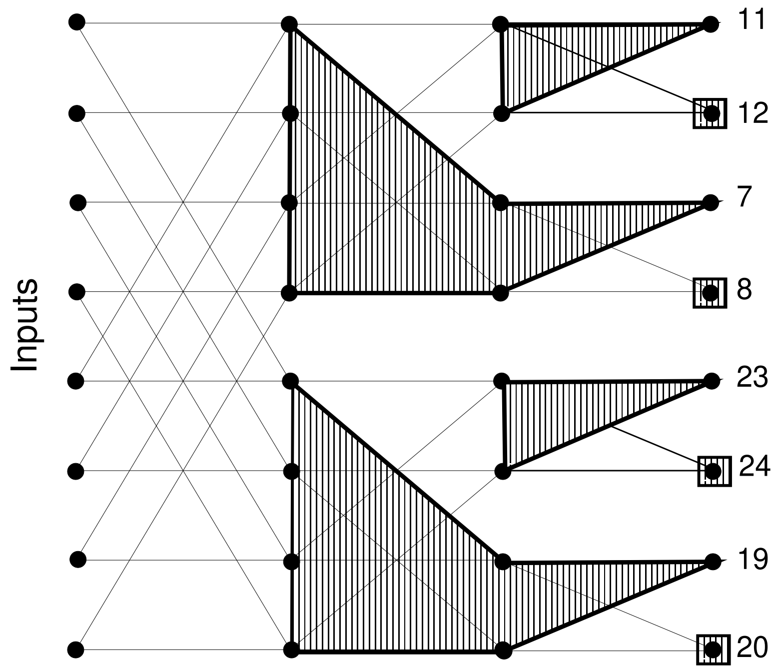

Figure 2: *Output permutation and schedule for P_2 for $m = 8$, $P = 4$; $\sigma^{(2)} = (1032)$; $\pi^{(2)} = (23016745)$. Shaded portion indicates incremental work in producing successive outputs. Numbers represent times at which outputs are generated.*

Processor 2 would thus not start receiving its messages until late and would not be able to receive its messages at the rate they were being sent to it.

This delay can be avoided by noting there are many output permutations that yield outputs at the optimal rate. Let T_m be the full binary tree with m leaves labeled $0, \ldots, m-1$. Consider a permutation of the leaves in which for each node, either all the leaves of the left subtree appear before any leaf of the right subtree, or vice versa. The recursive structure of the butterfly ensures that the time steps at which outputs are produced by a schedule driven by such a permutation do not depend on the actual permutation but merely its structure. Our interest in these permutations is chiefly due to the fact that these computation schedules produce outputs at an optimal rate: the first output is produced at the earliest possible time and each subsequent output is produced with minimum additional delay. The abundance of these permutations allows us to choose different optimal permutations for various processors in such a way that the reception schedules are also balanced. We make these ideas more precise below.

4.2 Binary Permutations

Definition 4.1 *Let $S_i = \{0, 1, \ldots, 2^i - 1\}$, $i \geq 0$ and let $\pi = (\pi_0, \ldots, \pi_{2^i-1})$ be a permutation of S_i. π is said to be* **binary** *if $i = 0$ or if $i > 0$ and the following conditions hold:*

- $\lfloor \frac{\pi_0}{2^{i-1}} \rfloor = \lfloor \frac{\pi_j}{2^{i-1}} \rfloor$ *for $j = 1, \ldots, 2^{i-1} - 1$.*
- $\pi^{(L)} = (\pi_0 \pmod{2^{i-1}}, \ldots, \pi_{2^{i-1}-1} \pmod{2^{i-1}})$ *and* $\pi^{(R)} = (\pi_{2^{i-1}} \pmod{2^{i-1}}, \ldots, \pi_{2^i-1} \pmod{2^{i-1}})$ *are binary permutations of S_{i-1}.*

As a small example, (4 5 6 7 3 2 0 1) is a binary permutation of S_3 while (4 5 6 7 3 0 2 1) is not, since (3 0 2 1) is not a binary permutation of S_2.

4.2.1 Maintaining Symmetry in Reception

As already noted, if each $\pi^{(i)}$ is binary, the message transmission schedules for the various processors are identical. However, there is still the possibility of undesirable asymmetry in the reception schedules. As an illustration, if each processor employed the identity permutation so that its outputs are generated in increasing order of row number, all messages destined for processor i would be sent out before any of the messages destined for processor $i+1$. Thus, processor $P-1$ would idle for a much longer time than processor 0, adversely affecting the algorithm's completion time. To ensure symmetry in reception, we would like that at each processor receive a message whenever any other processor does. To this end, we use the following more general decomposition property of binary permutations which is easily proved by induction:

Proposition 4.1 *Let $i = q + r$ with q and r non-negative and let $\pi = (\pi_0, \ldots, \pi_{2^i-1})$ be a binary permutation of S_i. Then*

- $\lfloor \frac{\pi_{k2^r}}{2^r} \rfloor = \lfloor \frac{\pi_{k2^r+j}}{2^r} \rfloor$ *for $1 \leq j < 2^r$ and for $0 \leq k < 2^q$.*
- *For each $k, 0 \leq k < 2^q, \pi^k = (\pi_{k2^r} \pmod{2^r}, \ldots, \pi_{(k+1)2^r-1} \pmod{2^r})$ is a binary permutation of S_r.*
- *The sequence $(\lfloor \frac{\pi_{k2^r}}{2^r} \rfloor), k = 0, 1, \ldots, 2^q - 1$ is a binary permutation of S_q.*

Thus, for any decomposition $i = q + r$, a binary permutation of S_i can be thought of as being uniquely determined by a binary permutation of S_q and 2^q independent binary permutations of S_r. Thus, for example, if $q = 2,\ r = 1$, the binary permutation (4 5 6 7 3 2 0 1) of S_3 decomposes into the binary permutation (2 3 1 0) of S_2 and the $2^q = 4$ binary permutations $(0\ 1), (0\ 1), (1\ 0), (0\ 1)$ of S_1.

To describe the binary permutations $\pi^{(i)}$, consider the decomposition $\log m = \log P + \log l$. For $i = 0, \ldots, P-1$, let $\sigma^{(i)} = (\sigma_0^{(i)}, \ldots, \sigma_{P-1}^{(i)})$ be the binary permutation of $S_{\log P}$ given by $\sigma_j^{(i)} = \bar{i} \oplus j$ and let $\pi^{(i)}$ be the binary permutation of $S_{\log m}$ determined by $\sigma^{(i)}$ and P copies of the identity permutation of $S_{\log l}$. In other words, processor i groups its m outputs into P blocks of l each, one block to be sent to each of the P processors. It uses $\sigma^{(i)}$ to determine the order in which blocks of outputs will be processed, and within a block processes outputs in increasing order of row number. Figure 2 shows a typical output driven schedule.

4.3 Running Time Analysis

Our choice of binary permutations ensures that the communication patterns for the various processors are identical and symmetric: whenever processor i sends a message to processor j, processor k sends one to processor $k \oplus i \oplus j$. Thus, it suffices to analyze the completion time of any processor. It is most convenient to study processor $P-1$ since its output permutation is the identity. It is clear that Phase I computation completes at time $m \log m$ and that Phase II computation requires time $m \log P$. If T denotes the time at which the last message is sent from a processor, the completion time for the algorithm is $\max\{T + L, m \log m\} + m \log P$. Our goal is to determine T. To simplify notation, we will identify the output node $(ap + (p-1), \log m)$ with the index a.

For $0 \leq a < m-l$ let t_a denote the time at which output node a is processed [3], and let s_a be the time at which it is sent. We have $s_0 = t_0$ and for $a \geq 1$, $s_a = \max\{t_a, s_{a-1}+g\}$. Hence, if a^* is the largest index for which $s_a = t_a$, then $T = t_{a^*} + g(m-l-a^*-1)$. In order to compute a^*, we need the following lemmas:

Lemma 4.1 *For the output driven schedule determined by a binary permutation, $t_0 = m-1$. If $a = q2^{r-1}$ with q odd, $t_a = t_{a-2^{r-1}} + r2^{r-1}$.*

Proof: The first output can be processed only after all of its $m-2$ predecessors: 2 in column $\log m - 1$, 4 in column $\log m - 2$,..., $m/2$ in column 1. For general a, it is easy to see that all of a's predecessors in columns 1 through $\log m - r$ have been processed by time $T_{a-2^{r-1}}$ and that exactly $r/2$ nodes in each of the columns $\log m - r + 1$ through $\log m$ are processed in generating output a. $\Box$

Corollary 4.1 *Let $a = b_t b_{t-1} \ldots b_0$ be the binary representation of a. Then $t_a = (m-1) + \sum_{i=0}^{t} b_i (i+1)2^i$.*

Lemma 4.2 *If $a \equiv 0 \pmod{2^{g-1}}$ then $s_a = t_a$; otherwise $s_a > t_a$.*

Proof: The proof is by induction on a. Since $s_0 = t_0$, the base case holds. For the induction step, it suffices to show that $t_{i2^{g-1}+k} < t_{i2^{g-1}} + kg$ whenever $0 < k < 2^{g-1}$ and that $t_{(i+1)2^{g-1}} \geq t_{i2^{g-1}} + g2^{g-1}$. Letting $k = \sum_{i=0}^{g-2} b_i 2^i$, Corollary 4.1 gives $t_{i2^{g-1}+k} - t_{i2^{g-1}} = \sum_{i=0}^{g-2} b_i(i+1)2^i < kg$, establishing the first inequality. Similarly, letting $(i+1)2^{g-1} = q2^{r-1}$, q odd, we see that $t_{(i+1)2^{g-1}} - t_{i2^{g-1}} = (r+1)2^r - \sum_{i=g-1}^{r-1}(i+1)2^i \geq g2^{g-1}$ which proves the second inequality.

Lemma 4.2 allows us to characterize a^* easily as the highest index less than $m-l$ which evenly divides 2^{g-1}. Further, we can use 4.1 to explicitly compute t_{a^*} and hence T. We omit the calculations and present only the final result.

Theorem 4.1 *Let n and P be powers of 2 with $n > P^2$. Let T denote the time at which the last message is sent by a processor in the output driven schedule determined by a binary permutation. Then:*

$$g \geq \log m \quad \Rightarrow \quad T = (m-1) + (m-l-1)g \tag{1}$$

$$\log \frac{n}{P^2} + 1 < g < \log m \quad \Rightarrow \quad T = m \log m + 2^g - g - 1 - lg \tag{2}$$

$$g \leq \log l + 1 \quad \Rightarrow \quad T = m \log m + (2^g - 1) - l(\log l + 1) \leq m \log m \tag{3}$$

4.4 Implications for Speedup

An immediate implication of Theorem 4.1 is that for $n \geq 2^g/gP^2$, there is *no idling* (except possibly for L steps) even though $O(n)$ messages are delivered by the network! Thus our schedule guarantees a speedup of P for large n despite the fact that the number of messages grow proportional to the problem size. Even if n is smaller (but at least P^2), our algorithm guarantees a speedup of at least $\frac{P}{1+\frac{1}{\log n}}$ as long as $g < \log P$. If $g \geq \log P$ and $n \leq 2^g P$, our output driven schedules cannot hide all communication and we analyze this situation more precisely in the next section.

[3] Note that $\pi^{(i)}$ is chosen such that processor i sends out the first $m-l$ outputs that it processes.

5 The Case of Small n

If the problem size is small and the bandwidth constraints stringent, some idling is inevitable since there isn't enough computation time in Phase I to hide the communication cost. For such cases, we will analyze the running time of the entire algorithm by considering overlap of messages with Phase II computation as well.

Suppose that $n < p2^g$. In this case, $a^* = 0$ as implied by Lemma 4.2. Thus, messages are sent at time steps $(m-1) + ig$, $i = 0, \ldots, m-l-1$. Recall that these messages contain the inputs for Phase II computation and that each processor's Phase II load comprises l disjoint P-input butterflies. The computation schedule for Phase II computation is *input driven*: a processor processes as many nodes as possible upon the arrival of an input and waits only if no more nodes can be processed until the arrival of the next input. For each of its Phase II butterflies, a processor has exactly one input available locally (as an output of its Phase I computation) and we will consider these inputs to be the first ones to "arrive". An input for which a processor waits will be called a *bottleneck*. Our goal is to identify the last bottleneck.

For $i = 0, 1, \ldots, m-1$, let δ_i denote the number of nodes that can be processed using the first i inputs to arrive but that cannot be processed using only the first $i-1$ inputs to arrive. The following necessary condition for an input to be a bottleneck is obvious.

Lemma 5.1 *If the i-th input to arrive is a bottleneck, then $\delta_{i-1} < g$.*

In addition, the following implications of our choice of binary permutations can be verified:

Fact 5.1 $\delta_{il+j} = \delta_{il}$ *for* $j = 1, 2, \ldots, l-1$.

Fact 5.2 *If $b_0 b_1 \ldots b_t$ is the binary representation of i and j is the smallest index such that $b_j = 0$ ($j = t+1$ if there is none such), then $\delta_{il} = 2(2^j - 1)$.*

Fact 5.3 *For each i, $\delta_i \geq \delta_{i-1}$.*

Let $i^* = \min\{i : \delta_i \geq g\}$ (with $i^* = m-1$ if there is no such i.) It follows from Lemma 5.1, and Fact 5.3 that if there are any bottlenecks at all, then i^* must be the last bottleneck. This gives a simple characterization of the running time for the entire algorithm. If there are no bottlenecks, the running time must be $m \log n$; if there are any bottlenecks, we can precisely locate the last one and hence determine its arrival time. Also, from the δ-sequence, we know exactly how many nodes remain to be processed after its arrival and can hence compute the overall running time. This is formalized in the following.

Theorem 5.1 *Let n and P be powers of 2 with $n \geq P^2, n < 2^g P$. Let $m = \frac{n}{P}$. Let $j^* = \lceil \log(g+2) \rceil - 1$. If $g > 2p-2$, let $i^* = m-1$ and $R = 2p-2$; otherwise let $i^* = m(1 - 1/2^{j^*})$ and $R = m \log P - m(j^* - 3 + \frac{1}{2^{j^*-2}}$. Then, the running time of our algorithm is*

$$max\{m \log n, (m-1) + L + (i^* - l)g + R\}$$

6 Conclusion

We have presented a computational schedule for parallel execution of the FFT graph that permits efficient overlap of computation and communication, effectively hiding all communication costs for large problems. We can also view this computational schedule as a communication-optimal simulation of a butterfly network on bandwidth-limited machines in the presence of appropriate slack. Thus, computations naturally suited to the butterfly can be implemented efficiently on low-bandwidth machines with 'large nodes'. Indeed, the only communication step required is an all-to-all personalized communication which can be easily and efficiently implemented for most networks.

Our schedule employs careful data distribution, pipelining of messages, and a dispersion of messages over a computation-intensive period – all techniques that are rewarded in parallel programming, especially for scientific computations. Our result can be viewed as helping explain the success enjoyed by such techniques.

References

[1] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Communication Complexity of PRAMs. In *Theoretical Computer Science*, pages 3–28, March 1990.

[2] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Coneference*, volume 32, pages 307–314, 1968.

[3] J. M. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp*, 19:297–301, 1965.

[4] J.M. Cooley, P.A. Lewis, and P.D. Welch. History of the fast Fourier transform. In *Proc. IEEE*, volume 55, pages 1675–1677, 1967.

[5] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993. (to appear) Also appears as UCB/CS/92 713 report.

[6] J. J. Dongarra, R. van de Geijn, and D. W. Walker. A Look at Scalable Dense Linear Algebra Libraries. In J. Saltz, editor, *Proceedings of the 1992 Scalable High Performance Computing Conference*. IEEE Press, 1992.

[7] C.M.. Fiduccia. Polynomial evaluation via the division algorithm – the fast Fourier transform revisited. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*, pages 88–93, 1972.

[8] W.M. Gentleman and G. Sande. Fast Fourier transforms for fun and profit. In *AFIPS 1966 Fall Joint Computer Conference*, volume 29, pages 563–578, Washington, D.C., 1967. Spartan.

[9] S. L. Johnsson, C.T. Ho, M. Jacquemin, and A. Ruttenberg. Computing fast Fourier transforms on Boolean cubes and related networks. In *Advanced Algorithms and Architectures for Signal Processing II*, pages 223–231. Society of Photo-Optical Instrumentation Engineers, 1987.

[10] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architecture: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, CA, 1992.

[11] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. In *Proceedings of the Twentieth Annual ACM Symposium of the Theory of Computing*, pages 510–513. ACM, 1988.

[12] Abhiram G. Ranade. How to emulate shared memory. In *Proceedings of the 28th IEEE Annual Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[13] A. Sahay. Overlapping communication and computation in parallel matrix algorithms, in preparation.

[14] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

[15] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the Association for Computing Machinery*, 33(8):103–11, August 1990.