10-1991

A Practical Hierarchical Model of Parallel Computation II: Binary Tree and FFT Algorithms

Todd Heywood

Sanjay Ranka
Syracuse University

SU-CIS-91-07

A Practical Hierarchical Model of Parallel Computation II: Binary Tree and FFT Algorithms

Todd Heywood and Sanjay Ranka

October 1991

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, New York 13244-4100

Abstract

A companion paper has introduced the Hierarchical PRAM (H-PRAM) model of parallel computation, which achieves a good balance between simplicity of usage and reflectivity of realistic parallel computers. In this paper, we demonstrate the usage of the model by designing and analyzing various algorithms for computing the complete binary tree, and the FFT/butterfly graph. By concentrating on two problems, we are able to demonstrate the results of different combinations of organizational strategies and different types of sub-models of the H-PRAM. The philosophy in algorithm design is to maximize the number of processors P that are *efficiently* usable with respect to an input size N, and to minimize the inefficiency when efficiency is not possible (when P is too large with respect to N). This can be done because of the H-PRAM's representation of general locality, i.e. both strict and neighborhood locality, and results in algorithms that can efficiently employ more processors (and are thus faster) than algorithms for models that only represent strict locality.

1 Introduction

In the companion paper [8] we introduced the Hierarchical PRAM (H-PRAM) model of parallel computation, and discussed its properties and benefits in detail. In this paper we demonstrate the usage of the H-PRAM via the design and analysis of various algorithms for computing both complete binary trees and FFT/butterfly graphs. By concentrating on two problems, we are able to demonstrate the results of different combinations of organizational strategies and different types of sub-models of the H-PRAM, e.g. EREW PRAM, LPRAM [1], and BPRAM [2].

It is assumed that the reader is familiar with the H-PRAM and its properties, as defined in [8].

General locality has been defined to mean both strict and neighborhood locality. We can employ general communication and synchronization locality on the *private* H-PRAM, or general synchronization locality (and *strict* communication locality) on the *shared* H-PRAM. Since the private H-PRAM provides the tightest simultaneous control over the four types of complexities (computation, communication, synchronization, and conceptual, as discussed in Section 2.2 of [8]), reduces the responsibilities of overlaying programming models (Section 2.2 of [8]), and provides a framework that may allow reduced communication bandwidth requirements on architectures as they scale up (Section 3.2 of [8]), we feel that algorithms should initially attempt to employ the private H-PRAM. However, problems that only submit to non-oblivious (data dependent) communication, or possess irregular communication patterns, may require a switch to the shared variant, which is the main reason for its existence. The private variant is assumed in this paper.

Section 2 of the section considers some preliminary algorithmic issues. In Section 3, we give two H-PRAM algorithms for computing a complete binary tree. The first serves as an introduction to the use of the H-PRAM, and the second introduces simple memory management, which results in improved performance for certain latencies.

In Section 4 we conduct a case study of the H-PRAM by designing and analyzing various H-PRAM algorithms for computing the FFT (or butterfly) graph. These algorithms differ both in design and in the types of sub-models which are employed, thus demonstrating the results of different combinations of organizational strategies and different types of sub-models of the H-PRAM. Section 5 concludes the paper with a discussion of general algorithmic issues.

2 Preliminaries

We always assume that $P \leq N$ for input size N and number of H-PRAM processors P. For simplicity, we also consider the size of the H-PRAM memory to be equal to the size of the data set under consideration (normally the input size, but some algorithms may use space $> N$). The reason for this is that, in the private H-PRAM, **partition** steps partition memory proportionately to the processors, and in algorithm design what we are really doing is partitioning the input data. This assumption is realistic since a mapping of the H-PRAM to an architectural model can simply assign N/P memory locations of the H-PRAM memory to the local memory of each architecture processor. The H-PRAM memory can be seen as P blocks of N/P memory locations, where the blocks are

the units that may be grouped and partitioned. Note that for any P'-processor sub-PRAM with N' memory, we have $N'/P' = N/P$ and $P' \leq N'$.

We employ the streamlined complexity analysis procedure introduced and justified in Section 2.3.2 of [8].

Recall from [8] that the latency $\ell(P')$ is the cost of a communication in a P'-processor sub-PRAM, and $s_\beta(Q, P')$ is the cost for a P'-processor sub-PRAM to β-synchronize Q sub-sub-PRAMs. Since latency is not necessarily the diameter of a P'-processor sub-network in an underlying architecture, $\text{diam}(P')$ is used when diameter is meant.

In H-PRAM algorithm analysis, it is often useful to fix the latency parameter ℓ in order to obtain more informative and concrete results. Since an exhaustive enumeration of results for all possible values of ℓ is out of the question, we wish to choose a couple of "representative" functions to fix ℓ to. In this paper we will consider the cases $\ell(P') = \log P'$ and $\ell(P') = \sqrt{P'}$, where P' is the number of processors being communicated amongst, since these functions (and functions of these functions) seem to cover a wide range of the latencies of existing and likely-to-be-built architectures.

It is also useful to fix the β-synchronization parameter s_β along with ℓ. Recall that when β-synchronization is solely a function of the number of sub-PRAMs being synchronized, $\text{diam}(Q) \leq s_\beta(Q, P') \leq \text{diam}(Q) \log Q$. As noted in Section 3.1 of [8], most hierarchical architectures have β-synchronization costs which are solely functions of Q. Thus, when $\ell(P') = \text{diam}(P') = \log P'$ we redenote $s_\beta(Q, P')$ as $s_\beta(Q)$. Although the range of $s_\beta(Q)$ is $\log Q \leq s_\beta(Q) \leq \log^2 Q$, for simplicity we fix $s_\beta(Q) = \log Q$ in this paper. This is very reasonable; for example, the hierarchical, logarithmic-diameter hypercube network can synchronize Q sub-cubes in $O(\log Q)$ time. If latency is not defined as diameter then we are at worst overestimating β-synchronization cost, since $\text{diam}(Q) \leq \ell(Q)$.

The two-dimensional mesh may have a β-synchronization cost that is also a function of the number of processors P' in the PRAM that is synchronizing the Q sub-PRAMs (see Section 3.1 of [8]). Although $s_\beta(Q, P')$ has been defined to be $\text{diam}(P') \leq s_\beta(Q, P') \leq \text{diam}(P') \log Q$ in this case, since the two-dimensional mesh is the only $\sqrt{P'}$-diameter architecture and it is known that P' mesh processors can synchronize in time $O(\sqrt{P'})$, when $\ell(P') = \sqrt{P'}$ we (redenote $s_\beta(Q, P')$ as $s_\beta(P')$ and) fix $s_\beta(P') = \sqrt{P'}$. Again, if latency is not defined as diameter then we are at worst overestimating β-synchronization cost.

The following fact, discussed in Section 2.3.2 of [8], will be used when β-synchronization is under consideration. The β-synchronization complexity of a sub-PRAM sub-algorithm *will not dominate* the sub-algorithm's complexity if the following two conditions are met:

1. $s_\beta(Q, P') \leq \ell(P')$ (note $Q \leq P'$ always).

2. The number of **partition** steps, thus the number of β-synchronizations, in the sub-algorithm is of the order of the number of communication steps in the sub-algorithm. Or equivalently, that it is of the order of N'/P' times the number of *permutation* steps on N' elements, which is the same as saying that the number of **partition** steps is of the order of the number of permutation steps on N' elements for all possible P' and N' (since $P' \leq N'$ always holds).

It is hard to conceive of any architecture where β-synchronization could not be done at the latency $\ell(P')$ of the sub-PRAM executing the **partition** step, so we assume Condition 1 to be true. Note that the cases we consider are (a) $\ell(P') = \log P'$ and $s_\beta(Q, P') = s_\beta(Q) = \log Q$, and (b) $\ell(P') = \sqrt{P'}$ and $s_\beta(Q, P') = s_\beta(P') = \sqrt{P'}$, so Condition 1 certainly holds for our purposes. Given this, what Condition 2 means is that β-synchronization will not dominate the complexity of any sub-PRAM algorithm that performs memory management, i.e. that executes permutation steps on the N' elements in the sub-PRAM memory (in time $(N'/P')\ell(P')$) "paired" with **partition** steps, since it will be subsumed by the cost of the permuting. If β-synchronization does not dominate the complexity of any of the sub-algorithms in a hierarchy, then clearly we can conclude that it does not dominate the complexity of the H-PRAM algorithm that they comprise.

Due to the numerous definitions of "optimal" in the literature, we need to make the following explicit definition.

Definition 2.1 *An* optimal *H-PRAM algorithm is one which is* efficient, *i.e. whose processor-time product is of the order of (within a constant factor of) the running time of the best known sequential algorithm for the problem at hand.*

We often refer to the "optimality range"; by this we mean the range of values that P may take with respect to N (or vice versa) such that optimality is achieved. We want to maximize the number of processors P that can efficiently be used on a problem of size N; the perfect (optimal!) optimality range is $P \leq N$.

Results are mainly compared to LPRAM [1] results for the same problem, thus comparing neighborhood and strict locality utilization (the LPRAM is a PRAM where, in addition to the global shared memory, processors have unbounded local memory).

3 Complete binary tree

Consider the computation of an N-leaf, height $\log N$ binary tree where the inputs to the problem reside at the leaves, and an internal node corresponds to the result of some computation $\oplus$ on the node's two children. Computation starts with the parents of the leaves and flows upward until the root is computed. This is a good introductory problem for the H-PRAM because of the tree hierarchy relation that it uses. The results of this section also hold for the parallel prefix problem by making two passes through the tree, first from the leaves to the root then from the root to the leaves [5].

We give two algorithms for computing a height $\log N$ ($\log N + 1$ levels) binary tree. The first is straightforward and intuitive, and serves as the introduction to usage of the H-PRAM; the second is an adjustment of the first that improves performance in certain cases (latencies), and serves as an introduction to simple memory management. Assume we have a P-processor H-PRAM and let k be an arbitrary value, $2 \leq k \leq P$, to be fixed later, where both P and k are powers of two (clearly N is a power of two). All sub-PRAMs are EREW PRAMs. We divide the N-leaf binary tree into $(\log P/\log k) + 1$ *layers*, where layers λ, $1 \leq \lambda \leq \log P/\log k$, have height $\log k$ and layer

$(\log P/\log k)+1$ has height $\log(N/P)$. The number of sub-trees in layer λ, $1 \leq \lambda \leq (\log P/\log k)+1$, is $k^{\lambda-1}$. If we represent each sub-tree in the partitioned binary tree as a node, then we obtain a k-ary tree of height $\log_k P = \log P/\log k$ whose P leaf nodes involve the computation of an N/P-leaf binary tree, and whose internal nodes involve the computation of a k-leaf binary tree.

The algorithm configures the H-PRAM to compute this k-ary tree. We create a k-ary hierarchy of sub-PRAMs, where each of the $\log_k P + 1$ levels of the hierarchy corresponds to a layer in the original binary tree, and sub-PRAMs in the same level of hierarchy computes the sub-trees in the corresponding layer of the binary tree. In other words, each sub-PRAM in levels λ, $1 \leq \lambda \leq \log_k P$, computes a k-leaf binary tree, and each sub-PRAM in level $\log_k P+1$ computes an N/P-leaf binary tree, with computation starting at the bottom of the hierarchy and proceeding upward.

The parameter k can be seen as a "partitioning parameter", dictating the sizes and numbers of sub-PRAMs used in the H-PRAM algorithm. The idea is that, given N and P, we will choose k to optimize the performance of the algorithm; this will be returned to subsequently.

In level λ, $1 \leq \lambda \leq \log_k P+1$, there are $k^{\lambda-1}$ sub-PRAMs, each of which has $P/k^{\lambda-1}$ processors. Note that this means that there is a single P-processor (sub)-PRAM in level 1 (as necessary) and P 1-processor sub-PRAMs in level $\log_k P + 1$. Thus each P'-processor sub-PRAM in level λ, $1 \leq \lambda \leq \log_k P$, creates k sub-sub-PRAMs of P'/k processors each, or equivalently, all of the sub-PRAMs in level λ collectively create k^λ level $\lambda+1$ sub-sub-PRAMs that have P/k^λ processors each; in other words $P'/k = P/k^\lambda$. Since by the definition of the private H-PRAM, memory is partitioned proportionately with the processors, each of the 1-processor sub-PRAMs in level $\log_k P+1$ initially has N/P leaves of the binary tree in its memory. The H-PRAM algorithm consists of the recursive use of one EREW PRAM sub-algorithm, $\mathsf{BT}(P', \lambda)$, which is given below.

$\mathsf{BT}(P', \lambda)$
    if $\lambda = log_k P + 1$ **then**
        <compute N/P-leaf binary tree with $P' = 1$ processor >
    else
        partition$\{k, P'/k,$ $\mathsf{BT}(P'/k, \lambda + 1)\}$
        <compute k-leaf binary tree with k processors; note that $k \leq P'$ >
    end-if-then-else
end-BT

The H-PRAM algorithm is invoked by $\mathsf{BT}(P, 1)$. The part of BT that computes an N/P-leaf binary tree is the standard sequential method. The part of BT that computes a k-leaf binary tree is the straightforward method of computing each level of the tree in parallel, where a processor computing an internal node (the leaves are in memory) reads the node's two children from memory, performs the operation $\oplus$ on them, and writes the result to memory. We note that only $k/2$

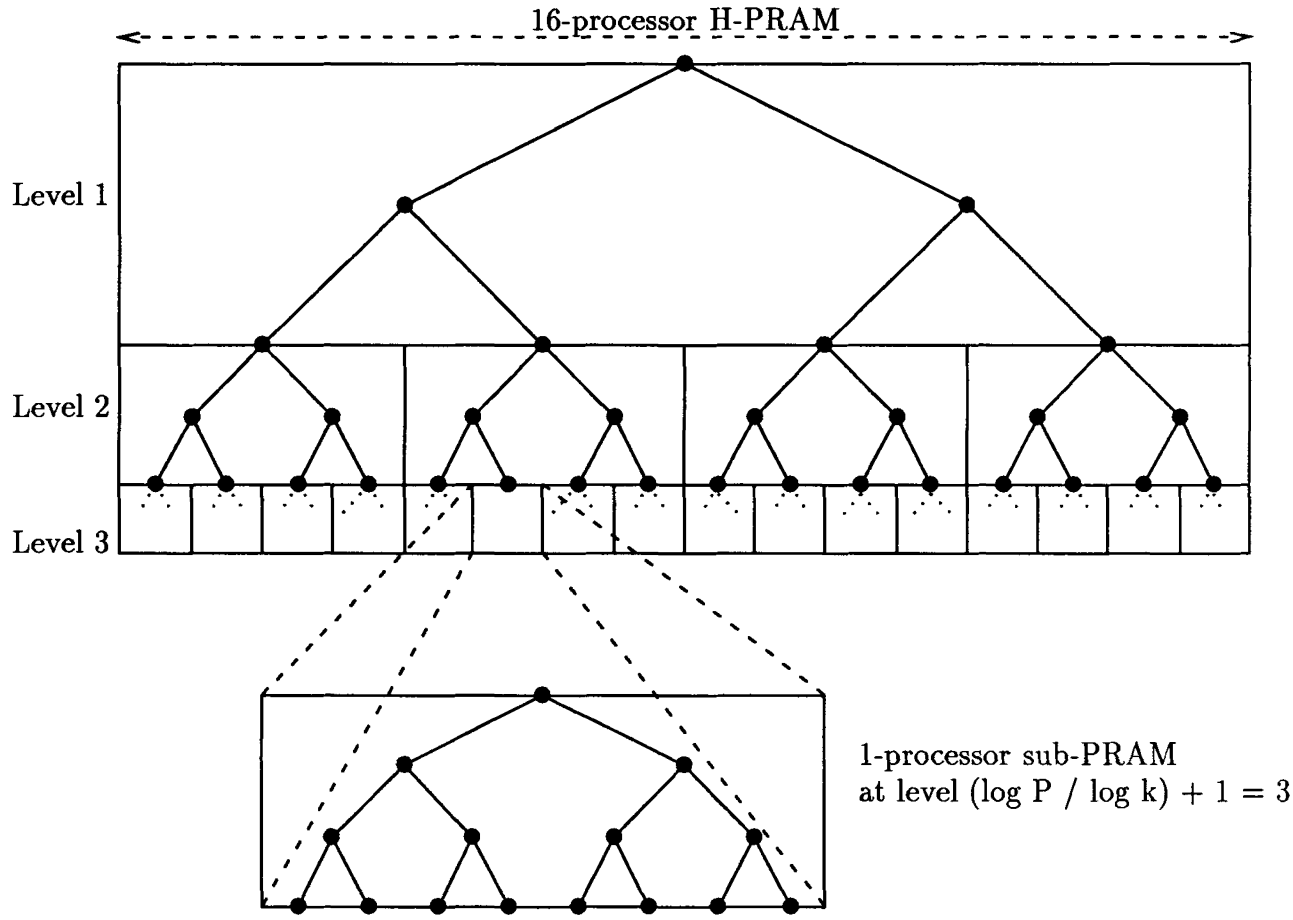

Figure 1: Hierarchy arising from $\mathsf{BT}(P,1)$ when $P = 16$, $N = 128$, and $k = 4$

processors, rather than k, are actually used to compute a k leaf binary tree. The quantity k is used in order to (slightly) simplify analysis and the adjusted algorithm discussed below.

Figure 1 shows the hierarchy that arises from $\mathsf{BT}(P,1)$ for $P = 16$, $N = 128$ (thus $N/P = 8$), and $k = 4$. Each box in the figure denotes a sub-PRAM, and the sub-tree within each box is the sub-tree computed by the sub-algorithm running on the corresponding sub-PRAM.

Lemma 3.1 *β-synchronization does not dominate the complexity of the* $\mathsf{BT}(P,1)$ *algorithm.*

Proof: Clearly, the number of **partition** steps (1), thus the number of β-synchronization steps, in a $\mathsf{BT}(P',\lambda)$ sub-algorithm is $\leq$ the number of communication steps. Thus the conditions for β-synchronization non-domination are satisfied for all sub-algorithms in the hierarchy, and we conclude that β-synchronization does not dominate the complexity of the $\mathsf{BT}(P,1)$ H-PRAM algorithm. □

Although we know that β-synchronization will not dominate $\mathsf{BT}(P,1)$, and thus can be dropped from further analysis, for this problem we keep it under consideration for demonstration purposes.

Theorem 3.1 $\mathsf{BT}(P,1)$ *can be computed in time*

$$O\left(\log P + \log k \cdot \sum_{\lambda=1}^{\log_k P} \ell(k^\lambda) + N/P + \sum_{\lambda=1}^{\log_k P} s_\beta(k,k^\lambda)\right)$$

Proof: Clearly $\mathsf{BT}(1,\log_k P+1)$ running on single processor, level $\log_k P+1$ sub-PRAMs, will take time $O(N/P)$ as $\ell(1)=1$. $\mathsf{BT}(P',\lambda)$ sub-algorithms running on $P'=P/k^{\lambda-1}$ processor sub-PRAMs in level λ, $1 \le \lambda \le \log_k P$, have $\log k$ (parallel) computation steps and $3\log k$ (parallel) communication steps. Thus the total cost of computation and communication steps in levels $\le \log_k P$ is

$$\begin{aligned} & O\left(\sum_{\lambda=1}^{\log_k P} (\log k + \log k \cdot \ell(P/k^{\lambda-1}))\right) \\ = \; & O\left(\log P + \log k \cdot \sum_{\lambda=1}^{\log_k P} \ell(P/k^{\lambda-1})\right) \end{aligned}$$

By reversing the sum, we get

$$O\left(\log P + \log k \cdot \sum_{\lambda=1}^{\log_k P} \ell(k^\lambda)\right)$$

Adding the $O(N/P)$ time for level $\log_k P+1$ sub-PRAMs to this gives the total time required for computation and communication. The final term in the complexity is that of β-synchronization, which is obtained by observing that $P'=P/k^{\lambda-1}$ processor sub-PRAMs in level λ, $1 \le \lambda \le \log_k P$ have one β-synchronization step on k sub-sub-PRAMs, for a β-synchronization complexity of $\sum_{\lambda=1}^{\log_k P} s_\beta(k, P/k^{\lambda-1})$. Again reversing the sum, we get $\sum_{\lambda=1}^{\log_k P} s_\beta(k,k^\lambda)$. □.

The form of this result is typical of H-PRAM algorithms which use hierarchies whose height is a function of the number of processors. We see that it is useful to consider specific latency functions in order to remove the sums and obtain a more concrete result (which can be compared with results for other models).

Theorem 3.2 *For* $\ell(P)=\log P$, $\mathsf{BT}(P,1)$ *can be computed in time*

$$O(\log^2 P + N/P)$$

Proof: Setting $\ell(x)=\log x$ in Theorem 3.1 simplifies the sum: $\sum_{\lambda=1}^{\log_k P} \log(k^\lambda) = O(\log k \cdot \log_k^2 P)$. So without β-synchronization complexity added in, the time is

$$\begin{aligned} & O(\log P + \log k(\log k \cdot \log_k^2 P) + N/P) \\ = \; & O(\log P + \log^2 P + N/P) \end{aligned}$$

regardless of the choice of k. Since $s_\beta(k,k^\lambda) = s_\beta(k) = \log k$ in this case, the β-synchronization component is $\sum_{\lambda=1}^{\log_k P} \log k = \log P$. Thus the complexity of $\mathsf{BT}(P,1)$ is $O(\log P + \log^2 P + N/P) = O(\log^2 P + N/P)$. □

Corollary 3.1 *For $\ell(P) = \log P$, $\mathsf{BT}(P,1)$ is optimal for $\epsilon P \log^2 P \leq N$, constant ϵ.*

Proof: Since the best sequential time for this problem is $O(N)$, $\mathsf{BT}(P,1)$ is optimal if $\log^2 P = O(N/P)$, i.e. if $P\log^2 P \leq c \cdot N$, some constant c, and $\epsilon = 1/c$. □

On the LPRAM, the algorithm for this problem takes time $O(\log^2 P + (N/P)\log P)$ [1] when $\ell(P) = \log P$. On the BPRAM [2], Phase PRAM [6] [7], and the model of Papadimitriou and Yannakakis [11] (which is essentially an arbitrary-pipelined LPRAM), which are all pipelined models, algorithms take time $O(\log^2 P/\log\log P + N/P)$ when $\ell(P) = \log P$ (and, for the Phase PRAM, which charges for (α) synchronization, when the synchronization cost is $O(\ell)$). If pipelining capability is taken away from these models, it adds a factor of $\log P$ to the times (although this comparison is somewhat unfair, as the algorithms are designed to take advantage of the pipelining). Note that the problem can be solved directly on the $\log P$ *diameter* hypercube in time $O(\log P + N/P)$ by embedding the tree in it.

Theorem 3.3 *For $\ell(P) = \sqrt{P}$, $\mathsf{BT}(P,1)$ can be computed in time*

$$O(\sqrt{P} + N/P)$$

Proof: Setting $\ell(P) = \sqrt{P}$ in Theorem 3.1 simplifies the sum: $\sum_{\lambda=1}^{\log_k P} \sqrt{k^\lambda} = O(\sqrt{P})$. So without β-synchronization complexity added in, the time is

$$O(\log P + \log k \cdot \sqrt{P} + N/P)$$

Since $s_\beta(k, k^\lambda) = s_\beta(k^\lambda) = \sqrt{k^\lambda}$ in this case, the β-synchronization component is $\sum_{\lambda=1}^{\log_k P} \sqrt{k^\lambda} = O(\sqrt{P})$. The $\log k$ term in the complexity indicates that k should be fixed at its minimum legal value, which is 2. Therefore we get

$$O(\log P + \sqrt{P} + N/P) = O(\sqrt{P} + N/P)$$

□

Corollary 3.2 *For $\ell(P) = \sqrt{P}$, $\mathsf{BT}(P,1)$ is optimal for $\epsilon \cdot P^{3/2} \leq N$, constant ϵ.*

Proof: The algorithm is optimal for $\sqrt{P} = O(N/P)$, which translates to $P^{3/2} \leq c \cdot N$, and $\epsilon = 1/c$. □

If the underlying latency $\ell(P) = \sqrt{P}$ architecture is a diameter $\sqrt{P}$ two-dimensional mesh, it may be best to fix $k = 4$ such that P'-processor sub-PRAMs will map to $\sqrt{P'} \times \sqrt{P'}$ sub-meshes. The complexity is affected by a factor of 2.

We note that $O(\sqrt{P} + N/P)$ is the fastest possible time for solving this problem on an architecture with $\ell(P) = \sqrt{P}$, and matches the complexity of the equivalent algorithms for the abovementioned pipelined models *without* using pipelining (all sub-PRAMs are EREW). On the LPRAM, the algorithm for this problem takes time $O(\sqrt{P}(\log P + N/P))$.

BT requires no explicit memory management, as it consists of only partitioning on the way down into the hierarchy, and computation as it proceeds back up where all data (leaves) needed by each sub-PRAM algorithm exists in the sub-PRAM memory. If we take responsibility for some memory management, we can adjust BT to obtain an H-PRAM algorithm that performs better for certain latencies, specifically $\ell(P) = \log P$ but not $\ell(P) = \sqrt{P}$ (for which BT achieves the fastest possible time anyway). The remainder of this section is devoted to this new algorithm, and serves as an introduction to the abstract control over memory management that the H-PRAM provides.

In BT, k processors are (actively) used by P'-processor sub-PRAMs to compute k-leaf binary trees in levels λ, $1 \leq \lambda \leq \log_k P$. In *level* $\log_k P$, $P' = k$; and in *levels* $< \log_k P$, $P' > k$. So it can be seen that we are unnecessarily charging $\ell(P')$ for communication even though only k processors are actually used. The adjustment to BT which results in the new algorithm, BT-Pack, is that we use k-processor sub-PRAMs to compute the k-leaf binary trees. The new algorithm works as before, except that as it proceeds back up through the hierarchy, a sub-PRAM with $> k$ processors (i.e. those in levels λ, $1 \leq \lambda \leq \log_k P - 1$) does *not* directly compute its k-leaf binary tree. Instead, it moves ("packs") the k leaves in its memory into a contiguous block of memory and performs a partition step such that this block becomes the private block of shared memory of a k-processor sub-sub-PRAM. The algorithm assigned to this sub-sub-PRAM computes the k-leaf binary tree with its k processors.

There are three sub-algorithms for this H-PRAM algorithm. Null-Alg is an algorithm that does absolutely nothing. Par-BT(x, y) computes a x-leaf binary tree with y processors (we will set $x = y = k$) via the straightforward method of computing each level of the tree in parallel, where a processor computing an internal node (the leaves are in memory) reads the nodes two children from memory, performs the operation $\oplus$ on them, and writes the result to memory. The last sub-algorithm is BT-Pack(P', λ), which is given below.

Recall from the beginning of this section that the input size N is equated to the H-PRAM's memory size, such that a partition step conceptually partitions data, and that the units of H-PRAM memory that are grouped and partitioned are blocks of N/P memory locations. A P'-processor sub-PRAM has memory (data) size $N' = P' \cdot N/P$. What the last partition step does is create two sub-sub-PRAMs: one with k processors and memory size $k \cdot N/P$, which will compute a k-leaf binary tree; and one with $P' - k$ processors and memory size $(P' - k) \cdot N/P$. Therefore, before the partition step we must make sure that all k leaves in the P'-processor sub-PRAM are moved into the block of memory that contains the first $k \cdot N/P$ memory locations, $0, \ldots, (kN/P) - 1$, of the sub-PRAM; specifying locations $i \cdot (N/P)$, $0 \leq i \leq k - 1$ just evenly spaces the leaves in memory. The partition step creates two sub-PRAMs; the k-processor one computes the k-leaf binary tree and the $(P' - k)$-processor one does nothing (Null-Alg).

The H-PRAM algorithm is invoked by BT-Pack$(P, 1)$.

Lemma 3.2 *β-synchronization does not dominate the complexity of the* BT-Pack$(P, 1)$ *algorithm.*

Proof: Identical to that of Lemma 3.1. □

```
BT-Pack(P', λ)
    if λ = log_k P + 1 then
        <compute N/P-leaf binary tree with P' = 1 processor >
    else if λ = log_k P then
        partition{k, P'/k,BT-Pack(P'/k, λ + 1)}
        <compute k-leaf binary tree with k processors; note that k = P' >
    else
        partition{k, P'/k,BT-Pack(P'/k, λ + 1)}
        <pack the k leaves: k processors read the leaves, and then write them to
        locations i · (N/P), 0 ≤ i ≤ k − 1 >
        partition{
            k: Par-BT(k, k);
            P' − k: Null-alg}
    end-if-then-else
end-BT-Pack
```

Again, although we know that β-synchronization does not dominate, we keep it under consideration in the following analysis for demonstration purposes.

Theorem 3.4 BT-Pack$(P, 1)$ *can be computed in time*

$$O\left(\log P + \ell(k)\log P + \sum_{\lambda=1}^{(\log_k P)-1} \ell(k^\lambda) + N/P + \sum_{\lambda=1}^{\log_k P} s_\beta(k, k^\lambda) + \sum_{\lambda=1}^{\log_k P - 1} s_\beta(2, k^\lambda)\right)$$

Proof: Clearly BT-Pack$(1, \log_k P + 1)$ running on single processor, level $\log_k P + 1$ sub-PRAMs, will take time $O(N/P)$ as $\ell(1) = 1$. BT-Pack(P', λ) sub-algorithms running on $P' = k$ processor sub-PRAMs in level $\log_k P$ have $\log k$ computation steps and $3\log k$ communication steps, thus taking time $O(\log k + \ell(k)\log k)$. BT-Pack$(P', \lambda)$ sub-algorithms running on $P' = P/k^{\lambda-1}$ processor sub-PRAMs in level λ, $1 \leq \lambda \leq (\log_k P) - 1$, have 2 communication steps, thus taking time $O(\sum_{\lambda=1}^{(\log_k P)-1} \ell(P/k^{\lambda-1}))$. Therefore the total cost of computation and communication steps in levels $\leq \log_k P$ by the BT-Pack(P', λ) sub-algorithm is

$$O\left(\log k + \ell(k)\log k + \sum_{\lambda=1}^{(\log_k P)-1} \ell(P/k^{\lambda-1})\right)$$

Reversing the sum of the third term:

$$O\left(\log k + \ell(k)\log k + \sum_{\lambda=1}^{(\log_k P)-1} \ell(k^\lambda)\right)$$

The Par-BT and Null-Alg algorithms run in levels λ, $2 \leq \lambda \leq \log_k P$, concurrently with each other. Since Null-Alg does nothing, the time of a partition step which invokes the two algorithms is dominated by Par-BT, which has $3 \log k$ communication steps and $\log k$ computation steps on k processors. So the total time spent in Par-BT is $O((\log_k P - 1)(\log k + \ell(k) \log k))$. Putting the times for BT-Pack(P', λ) and Par-BT together, we get

$$\begin{aligned} & O\left(\log_k P(\log k + \ell(k)\log k) + \sum_{\lambda=1}^{(\log_k P)-1} \ell(k^\lambda) + N/P\right) \\ = \; & O\left(\log P + \ell(k)\log P + \sum_{\lambda=1}^{(\log_k P)-1} \ell(k^\lambda) + N/P\right) \end{aligned}$$

The final term in the complexity is that of β-synchronization. The β-synchronization from the k-ary hierarchy is the same as before, i.e. $\sum_{\lambda=1}^{\log_k P} s_\beta(k, k^\lambda)$, and we have the additional cost of β-synchronizing 2 sub-PRAMs in each level λ, $1 \leq \lambda \leq \log_k P - 1$, which is $\sum_{\lambda=1}^{\log_k P - 1} s_\beta(2, k^\lambda)$. □.

Theorem 3.5 *For $\ell(P) = \log P$,* BT-Pack$(P, 1)$ *can be computed in time*

$$O(\log^{3/2} P + N/P)$$

Proof: Setting $\ell(P) = \log P$ in Theorem 3.4 simplifies the sum: $\sum_{\lambda=1}^{(\log_k P)-1} \log(k^\lambda) = O(\log^2 P / \log k - \log P + \log k)$. So without β-synchronization complexity added in, the time is

$$\begin{aligned} & O\left(\log P + \log k \cdot \log P + (\frac{\log^2 P}{\log k} - \log P + \log k) + N/P\right) \\ = \; & O\left(\log k \cdot \log P + \frac{\log^2 P}{\log k} + \log k + N/P\right) \\ = \; & O\left(\log k \cdot \log P + \frac{\log^2 P}{\log k} + N/P\right) \end{aligned}$$

In this case we have $s_\beta(k, k^\lambda) = s_\beta(k) = \log k$ and $s_\beta(2, k^\lambda) = s_\beta(2) = \log 2 = 1$. Therefore the β-synchronization component is subsumed by the other components of the complexity. because $\sum_{\lambda=1}^{\log_k P} \log k = \log P$ and $\sum_{\lambda=1}^{\log_k P - 1} s_\beta(2) = \log_k P - 1$.

We wish to pick k to minimize the complexity

$$O(\log k \cdot \log P + \frac{\log^2 P}{\log k} + N/P)$$

Choosing $k = 2^{\sqrt{\log P}}$, and noticing that k is legal (i.e. a power of two such that $2 \leq k \leq P$) gives the result $O(\log^{3/2} P + N/P)$. □

Corollary 3.3 *For $\ell(P) = \log P$,* BT-Pack$(P, 1)$ *is optimal for $\epsilon P \log^{3/2} P \leq N$, constant ϵ.*

Proof: The algorithm is optimal if $P\log^{3/2} P = O(N)$, i.e. $P\log^{3/2} P \leq c \cdot N$, constant c, and $\epsilon = 1/c$. □

In comparison to the $O(\log^{3/2} P + N/P)$ time for BT-Pack$(P, 1)$, the LPRAM algorithm for this problem takes time $O(\log^2 P + (N/P)\log P)$ [1] when $\ell(P) = \log P$. The algorithms for the pipelined models of [2] [6] [7] [11] take time $\log^2 P/\log\log P + N/P$ when $\ell(P) = \log P$.

BT-Pack$(P, 1)$ gives no improvement over BT$(P, 1)$ for the case $\ell(P) = \sqrt{P}$.

4 Case study: FFT graph

In this section we conduct a case study of the utility of the private H-PRAM by designing and analyzing algorithms for computing the FFT, or butterfly, graph. We give algorithms that employ both two-level and multi-level ($\log_k P$) hierarchies, with different combinations of EREW PRAM, LPRAM, and BPRAM sub-models for these cases. (Although it has been noted in Section 2 of [8] that it may not be a good idea in practice to use extended PRAMs such as the LPRAM or BPRAM as sub-models, it is important to include them in a case study.) In addition to demonstrating different instances and contexts in which the H-PRAM can be used, a key goal of this section is to show how the H-PRAM can exploit general locality. Strict locality can sometimes be used to to obtain optimal algorithms, but the optimality only holds when N is significantly larger than P. We show that, in the context of the FFT graph problem, when N is not-so-large with respect to P, we can switch to employing neighborhood locality to achieve optimality. Furthermore, the configuration of the H-PRAM, representing the degree of locality employed, can be parameterized by N and P so that the "best" configuration is used, resulting in optimal algorithms for a wider range of N and P with respect to each other than achievable with strict locality alone.

The problem of computing the FFT graph is an ideal problem for a case study of the H-PRAM due to its regular structure, and high communication and synchronization requirements.

A directed acyclic graph, or "dag", can be used to represent a computational problem, where nodes represent inputs (if in-degree is 0), outputs (if out-degree is 0), or computations (if in-degree and out-degree are positive), and whose edges represent data dependencies, or communications. The time required to compute a dag is the time taken by an algorithm to compute all of the output (out-degree 0) nodes. The binary tree of the previous section is a dag whose leaves are input nodes, and root is the output node. The FFT graph, also known as the butterfly graph, is a dag whose computation can solve several problems; the Fast Fourier Transform and bitonic merge are two simple examples.

For $N = 2^m$, an N-point, height $\log N$ FFT graph has $\log N + 1$ levels of N nodes each, and can be represented algorithmically as follows. Let $x_{i,j}$ denote the jth node, $0 \leq j \leq N-1$, in level i, $0 \leq i \leq m$. Then the graph is defined by

- Inputs: $x_{0,0}, x_{0,1}, \ldots, x_{0,N-1}$
- Outputs: $x_{m,0}, x_{m,1}, \ldots, x_{m,N-1}$

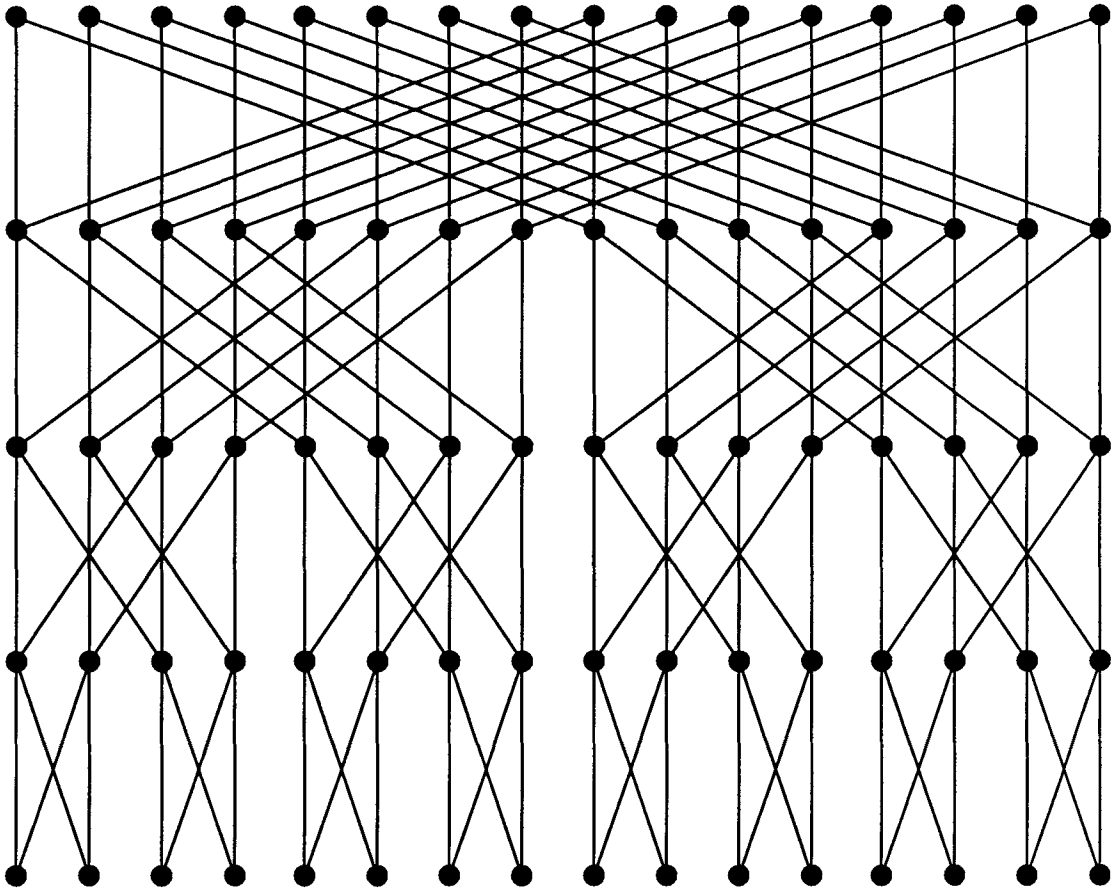

Figure 2: A 16-point FFT graph

- Computations: For $1 \leq i \leq m$, $x_{i,q} = f(x_{i-1,q}, x_{i-1,r})$

where f is a binary function computed in constant time, and q and r have binary representations that are identical except in the $(m-i)$th position. Figure 2 shows a 16-point FFT graph.

We assume again in this section that P is a power of two.

In Section 4.1 we give a simple extension of the BT algorithm to solve the FFT graph problem. The following sections consider more sophisticated H-PRAM algorithms for computing the FFT graph that undertake memory management; Section 4.2 considers algorithms that employ two-level hierarchies, while Section 4.3 addresses algorithms which use $(\log_k P)$-level hierarchies.

4.1 Binary tree extension

As with a binary tree, note that the N-point, height $\log N$ FFT graph can be divided into $\log_k P + 1$ stages, where stages λ, $1 \leq \lambda \leq \log_k P$ have height $\log k$, and stage $\log_k P + 1$ has height $\log(N/P)$. Furthermore, the result of this is that the FFT graph is partitioned into disjoint sub-graphs, where

a sub-graph in level λ ,$1 \leq \lambda \leq \log_k P$, consists of the first $\log k$ levels of a $(N/k^{\lambda-1})$-point FFT graph, and a sub-graph in level $\log_k P+1$ is a N/P-point FFT graph. Therefore we see that we can adjust the binary tree algorithm, BT$(P,1)$, to obtain an algorithm for computing the FFT graph, where stages correspond to levels of hierarchy, and sub-PRAMs to sub-graphs. The adjustments are that computation is done on the way down into the hierarchy, rather than on the way back up, and that FFT sub-graphs are computed rather than binary sub-trees.

Note that no memory management is required; as long as we store the jth point in a level of a FFT sub-graph in the jth memory location of the sub-PRAM that is computing (the first $\log k$ levels of) it, the memory (points) partition as required. The H-PRAM algorithm consists of one sub-algorithm FFT-BT-Ext(P', λ) that is used recursively, given below, and is invoked by FFT-BT-Ext$(P, 1)$.

FFT-BT-Ext(P', λ)
  if $\lambda = log_k P + 1$ then
    <compute (N/P)-point FFT with $P' = 1$ processor >
  else
    <compute first $\log k$ levels of $(N/k^{\lambda-1})$-point FFT
    with $P' = P/k^{\lambda-1}$ processors >
    partition$\{k, P'/k,$ FFT-BT-Ext$(P'/k, \lambda+1)\}$
  end-if-then-else
end-FFT-BT-Ext

Theorem 4.1 FFT-BT-Ext$(P,1)$ *has complexity*

$$O\left(\frac{N}{P}(\log P + \log k \cdot \sum_{\lambda=1}^{\log_k P} \ell(k^\lambda)) + \frac{N}{P}\log(N/P) + \sum_{\lambda=1}^{\log_k P} s_\beta(k, k^\lambda)\right)$$

Proof: The proof follows from that of BT (Theorem 3.1) after noting two differences between FFT-BT-Ext(P', λ) and BT(P', λ). First, in level $\log_k P + 1$ the FFT-BT-Ext sub-algorithms take time $O((N/P)\log(N/P))$ to compute an (N/P)-point FFT graph, rather than $O(N/P)$ for BT. Second, FFT-BT-Ext sub-algorithms operating in levels $\leq \log_k P$ take time that is an additional factor of N/P over that of BT sub-algorithms. Specifically, the time required for a level λ, $1 \leq \lambda \leq \log_k P$, FFT-BT-Ext sub-algorithm, which computes $\log k - 1$ levels (since one of the $\log k$ levels is the "input level") of $N/k^{\lambda-1}$ points using $P/k^{\lambda-1}$ processors, is

$$O\left(\frac{N/k^{\lambda-1}}{P/k^{\lambda-1}} \cdot (\log k + \log k \cdot \ell(P/k^{\lambda-1}))\right) = O\left(\frac{N}{P}(\log k + \log k \cdot \ell(P/k^{\lambda-1}))\right)$$

The proof follows from that of Theorem 3.1. □

Theorem 4.2 *For $\ell(P) = \log P$,* FFT-BT-Ext$(P, 1)$ *can be computed in time*

$$O\left(\frac{N}{P}(\log^2 P + \log(N/P))\right)$$

Proof: Follows from Theorem 3.2 and Theorem 4.1. □

Corollary 4.1 *For $\ell(P) = \log P$,* FFT-BT-Ext$(P, 1)$ *is optimal for $P^{\epsilon \cdot \log P} \leq N$, constant ϵ.*

Proof: Since the best sequential time for this problem is $O(N \log N)$, FFT-BT-Ext$(P, 1)$ is optimal if $\log^2 P = O(\log N)$, or $\log^2 P \leq c \cdot \log N$ for some constant c. This solves to $P^{\epsilon \cdot \log P} \leq N$, where $\epsilon = 1/c$. □

Theorem 4.3 *For $\ell(P) = \sqrt{P}$,* FFT-BT-Ext$(P, 1)$ *can be computed in time*

$$O\left(\frac{N}{P}(\sqrt{P} + \log(N/P))\right)$$

Proof: Follows from Theorem 3.3, and Theorem 4.1 above. □

Corollary 4.2 *For $\ell(P) = \sqrt{P}$,* FFT-BT-Ext$(P, 1)$ *is optimal for $2^{\epsilon \cdot \sqrt{P}} \leq N$, constant ϵ.*

Proof: The algorithm is optimal when $\sqrt{P} = O(\log N)$, or $\sqrt{P} \leq c \cdot \log N$, constant c. This solves to $2^{\epsilon \cdot \sqrt{P}} \leq N$, where $\epsilon = 1/c$. □

The algorithm for computing the FFT graph on the LPRAM [1] is optimal for $P \cdot 2^{\epsilon \cdot \ell(P)} \leq N$, so we see that the H-PRAM allows a larger number of processors to be efficiently used than on the LPRAM for $\ell(P) = \sqrt{P}$, but not $\ell(P) = \log P$ (since $P \cdot 2^{\epsilon \cdot \ell(P)} = P^{\epsilon+1} < P^{\epsilon \cdot \log P}$). The range $P^{\epsilon \cdot \log P} \leq N$ for $\ell(P) = \log P$ is not very good, and gives motivation to the development of the additional algorithms of the following two sections.

4.2 Two-level algorithms

We begin by noticing (in more detail than the previous section) how the FFT graph can be partitioned, so that we can partition the H-PRAM accordingly in order for sub-PRAMs to compute sub-graphs. Let k be an arbitrary value, whose bounds will be fixed subsequently, which is a power of two. Then the N-point, height $\log N$ FFT graph can be partitioned into $\log N / \log k = \log_k N$ stages of height $\log k$, where each stage consists of N/k *independent* k-point FFT graphs. Within a stage, the k-point FFT graphs are generally "intertwined", i.e. have intersecting edges, but share no common nodes, so are thus disjoint from each other. The output of one stage is the input of the next.

Figure 3 shows a 16-point FFT graph with $k = 4$. It is thus partitioned into $\log 16 / \log 4 = 2$ stages of height $\log 4 = 2$. In each stage, one independent ($k = 4$)-point FFT sub-graph is outlined in bold.

The idea behind a two-level H-PRAM algorithm is to compute the FFT graph stage by stage, with the level 1 P-processor PRAM acting as the coordinator and executing $\log_k N$ partition steps,

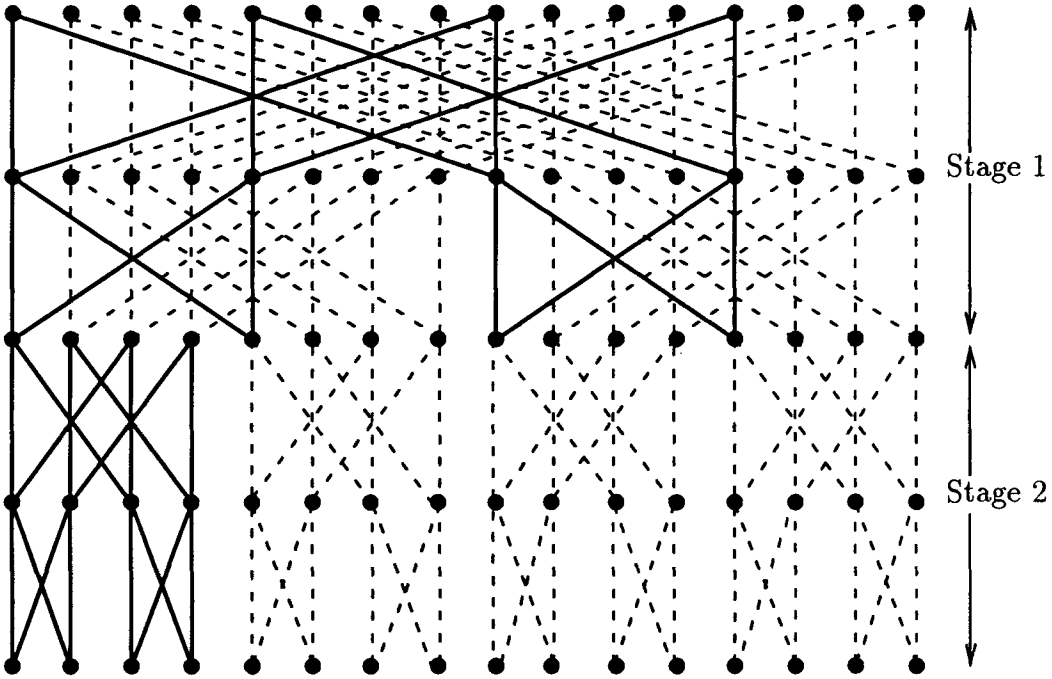

Figure 3: A partitioned (with $k = 4$) 16-point FFT graph. Within each stage, one independent ($k = 4$)-point FFT sub-graph is outlined in bold.

and employing N/k level 2 sub-PRAMs to compute the k-point FFT (sub)-graphs. However, since the k-point FFT graphs within a stage are generally intertwined, meaning that the points of a sub-graph are not contiguous in (level 1) memory, the k-point sub-graphs must be "untangled" so that level 2 sub-PRAMs can compute them. This means that memory management is required; but if we do this, then we can obtain parameterized (through k) control over the sizes of *all* FFT (sub)-graphs that are computed in the algorithm. In other words, we can parameterize general locality, and k can be set to obtain the optimum degree of locality for certain ℓ, P, and N.

The memory management required is as follows. At the beginning and end of every stage, the jth point of the corresponding level is stored in the jth memory location, $0 \leq j \leq N-1$, in the level 1 PRAM. Then untangling the k-point FFT graphs at each stage consists of permuting the N points, i.e. permuting the H-PRAM memory, such that the points of each k-point FFT graph are in contiguous memory locations. Then blocks of k memory locations (points) will become the shared memories of level 2 sub-PRAMs. After the level 2 sub-PRAMs compute the k-point FFTs and exit, the memory is "unpermuted" so that points are returned to their correct order in the current level in the N-point FFT graph.

The required permutation is the k-shuffle, and the "unpermutation" the k-unshuffle. The k-shuffle (k-unshuffle) is equivalent to performing $\log k$ perfect shuffles (unshuffles). The permutations are "segmented", i.e. number the permutations are applied independently to multiple blocks of contiguous points, whose number and size depends on the stage number. To be more specific, in stage λ, $1 \leq \lambda \leq \log_k N$, the k-shuffle is applied to each of $k^{\lambda-1}$ blocks of $N/k^{\lambda-1}$ points prior to computing the k-point FFTs of that stage, and the k-unshuffle is applied to the same blocks following the k-point FFT computations. (Note that in stage $\log_k N$ the permutations are identity permutations and thus are not necessary.) The fact that the memory management permutations are "segmented" ones is used in Section 4.3 to design multi-level H-PRAM algorithms.

The *general* two-level, P-processor H-PRAM algorithm for computing the N-point FFT graph consists of two sub-algorithms: the coordinator sub-algorithm that runs on the level 1 PRAM, performing the memory management, and the sub-algorithm that computes a k-point FFTs on a level 2 sub-PRAM. Since, in each stage, there are N/k FFT graphs to be computed, each partition step in the level 1 sub-algorithm creates N/k level 2 sub-PRAMs. Note that the minimum number of sub-PRAMs created must be one ($N/k \geq 1$) and the maximum must be P ($N/k \leq P$), therefore the bounds on the value of k are $N/P \leq k \leq N$. The number of processors in a level 2 sub-PRAM is $P' = \frac{P}{N/k} = \frac{k}{N/P}$.

The *specific* H-PRAM algorithms are those that we have when the types (PRAM, LPRAM, BPRAM) of sub-models are fixed, thus fixing the specifics of the sub-algorithms that run on those sub-models. We give a general, high-level description of the two-level H-PRAM algorithm, and then consider the specific instances of it that occur when various combinations of sub-model types are fixed. Let Direct-FFT(P') be the sub-algorithm that is assigned to level 2 sub-PRAMs to compute k-point FFT graphs with P' processors. The sub-algorithm running on the level 1 PRAM is FFT-Two-Level(P'), and is given below; the H-PRAM algorithm is initiated by FFT-Two-Level(P).

FFT-Two-Level(P')

for $\lambda = 1$ to $\log_k N$ do

$<$ k-shuffle on $k^{\lambda-1}$ blocks of points/memory $>$

partition$\{N/k, \frac{k}{N/P}$, Direct-FFT$(\frac{k}{N/P})$ $\}$

$<$ k-unshuffle on $k^{\lambda-1}$ blocks of points/memory $>$

end-for-do

end-FFT-Two-Level

We use the notation TYPE1-TYPE2 to refer to the specific H-PRAM algorithms that use a TYPE1 sub-model in level 1 (on which FFT-Two-Level runs) and TYPE2 sub-models in level 2 (on which Direct-FFT algorithms run). We consider three types of sub-models: EREW PRAM (EREW for short), LPRAM, and BPRAM. There are three specific H-PRAM algorithms presented: EREW-EREW, EREW-LPRAM, and BPRAM-EREW. The reason that the LPRAM is not considered as a TYPE1 sub-models is that it has no advantages over the EREW PRAM for performing permutations. As noted in Section 3.2 of [8], whether to classify the BPRAM as a computational or architectural model is unclear. We explore its use as a TYPE1 sub-model since its algorithm for "rational permutations" can implement our memory management, and we want to see the results of using block pipelining to move data around (permute it) for memory management. While the EREW-EREW and EREW-LPRAM algorithm results are presented in detail, we only briefly discuss the BPRAM-EREW results since the rather complicated complexity of the BPRAM rational permutation (sub)-algorithm combined with dynamic configurability of the H-PRAM leads to tedious and complicated analysis. We do not consider algorithms that employ the BPRAM as a TYPE2, variable-sized (parameterized by k) sub-model.

We first consider the β-synchronization complexity of FFT-Two-Level algorithms, as it is independent of the types of sub-models used.

Theorem 4.4 *The β-synchronization component in* FFT-Two-Level *algorithm complexities will never dominate.*

Proof: FFT-Two-Level runs in level 1 on all P H-PRAM processors, executing $\log_k N = \log N / \log k$ partition steps. Since there are $2\log_k N$ permutation steps, the conditions for β-synchronization non-domination are met. □

Thus, we drop consideration of β-synchronization complexity from the subsequent analyses of the FFT-Two-Level algorithms. This results in a significantly cleaner and simpler process.

We first concentrate on the EREW-EREW FFT-Two-Level algorithm.

4.2.1 EREW-EREW FFT-Two-Level

Theorem 4.5 *The* EREW-EREW FFT-Two-Level *algorithm can be computed in time*

$$O\left(\frac{N\log N}{P}\cdot\left(\ell(\frac{k}{N/P})+\frac{\ell(P)}{\log k}\right)\right)$$

Proof: FFT-Two-Level, running on the level 1, P-processor sub-PRAM, executes $2\log_k N$ permutations and $\log_k N$ partition steps which create N/k sub-PRAMs. Permuting N elements takes time $O((N/P)\cdot\ell(P))$, therefore the time taken by the FFT-Two-Level sub-algorithm is $O(\log_k N\cdot(N/P)\cdot\ell(P))$. A Direct-FFT sub-algorithm operating in level 2 on $P'=\frac{k}{N/P}$ processors computes a k-point FFT graph in the straightforward way in time

$$\begin{aligned}O\left(\frac{k\log k}{P'}+\frac{k\log k}{P'}\cdot\ell(P')\right) &= O\left(\frac{N\log k}{P}+\frac{N\log k}{P}\cdot\ell(\frac{k}{N/P})\right)\\ &= O\left(\frac{N\log k}{P}\cdot\ell(\frac{k}{N/P})\right)\end{aligned}$$

and level 2 Direct-FFT sub-algorithms are invoked $\log_k N$ times. Therefore the total time taken by the H-PRAM algorithm is

$$O\left(\log_k N\cdot\left(\frac{N\log k}{P}\cdot\ell(\frac{k}{N/P})+(N/P)\cdot\ell(P)\right)\right)$$

Which, by changing $\log_k N$ to $\log N/\log k$ and rearranging, gives the result. □

Again, k must be $N/P\le k\le N$.

Theorem 4.6 *The* EREW-EREW FFT-Two-Level *algorithm is optimal for* $P\cdot 2^{\epsilon\cdot\ell(P)}\le N$, *constant* ϵ.

Proof: The optimal parallel time is $O((N/P)\log N)$ so we see that the EREW-EREW FFT-Two-Level algorithm is optimal when

$$\ell(\frac{k}{N/P})+\frac{\ell(P)}{\log k}=O(1)$$

In order for this to hold, we need $\ell(\frac{k}{N/P})=O(1)$, which means that k must be $O(N/P)$ by the definition of the latency function (see Section 2.3 of [8]). So we choose $k=N/P$. We also need

$$\frac{\ell(P)}{\log k}=\frac{\ell(P)}{\log(N/P)}=O(1)$$

or equivalently $\ell(P)\le c\cdot\log(N/P)$, constant c. This solves to $P\cdot 2^{\epsilon\cdot\ell(P)}\le N$, where $\epsilon=1/c$. □

The range $P\cdot 2^{\epsilon\cdot\ell(P)}\le N$ is the same as that of the FFT graph algorithm for the LPRAM [1]. This is expected, since $k=N/P$ means that there are 1-processor sub-PRAMs in level 2, and the LPRAM is an instance of the H-PRAM corresponding to a two-level private H-PRAM with 1-processor sub-PRAMs in level 2.

Note that this is an improvement in the optimality range over that of FFT-BT-Ext for $\ell(P)=\log P$ but not for $\ell(P)=\sqrt{P}$.

The fact that $k = N/P$ means that only strict locality is employable to achieve optimality, for FFT-Two-Level. If $k > N/P$, sub-PRAMs have > 1 processor; this would be using neighborhood locality. Although choosing $k > N/P$ will not achieve optimality, it may result in an improved complexity (over that from choosing $k = N/P$) when the given N and P are not within the optimality range. In other words, neighborhood locality may be useful for *reducing inefficiency* when N is not so much larger than P.

Following these lines, we now consider what we call the *term minimization strategy* for configuring the H-PRAM (i.e. for choosing k, given N and P). Very simply, it consists of minimizing the non-optimal factor

$$\ell(\frac{k}{N/P}) + \frac{\ell(P)}{\log k}$$

by choosing k such that the two terms are equal to each other. To do this, we need to fix the latency function.

Lemma 4.1 *When $\ell(P) = \log P$, the equality*

$$\ell(\frac{k}{N/P}) = \frac{\ell(P)}{\log k}$$

holds when k is chosen such that

$$\log k = \frac{1}{2}\left(\log(N/P) + \sqrt{\log^2(N/P) + 4\log P}\right)$$

and furthermore, this k is within the required bounds $N/P \leq k \leq N$.

Proof: k is chosen such that

$$\log(\frac{k}{N/P}) = \frac{\log P}{\log k}$$

which simplifies to

$$\log^2 k - \log(N/P)\log k - \log P = 0$$

Applying the quadratic formula gives the solution for $\log k$:

$$\log k = \frac{1}{2}\left(\log(N/P) + \sqrt{\log^2(N/P) + 4\log P}\right)$$

It needs to be checked that $N/P \leq k \leq N$, or equivalently that $\log(N/P) \leq \log k \leq \log N$. We first check that $\log(N/P) \leq \log k$:

$$\begin{aligned}
\log(N/P) &\leq \frac{1}{2}\left(\log(N/P) + \sqrt{\log^2(N/P) + 4\log P}\right) \\
\log(N/P) &\leq \sqrt{\log^2(N/P) + 4\log P} \\
\log^2(N/P) &\leq \log^2(N/P) + 4\log P
\end{aligned}$$

which clearly holds. We now check that $\log k \leq \log N$:

$$\begin{aligned}
\frac{1}{2}\left(\log(N/P) + \sqrt{\log^2(N/P) + 4\log P}\right) &\leq \log N \\
\sqrt{\log^2(N/P) + 4\log P} &\leq \log N + \log P \\
(\log N - \log P)^2 + 4\log P &\leq (\log N + \log P)^2 \\
4\log P &\leq 4\log N \log P
\end{aligned}$$

which clearly holds. Finally, we note that k is a power of two, as required. □

Theorem 4.7 *For $\ell(P) = \log P$, the* EREW-EREW FFT-Two-Level *algorithm can be computed in time*

$$O\left(\frac{N\log N}{P}(\sqrt{\log^2(N/P) + 4\log P} - \log(N/P))\right)$$

Proof: Choosing k as in Lemma 4.1, we have $\log(\frac{k}{N/P}) = \frac{\log P}{\log k}$, so the non-optimal factor in the complexity is

$$\begin{aligned}
&\leq 2\log(\frac{k}{N/P}) \\
&= 2\left(\frac{1}{2}(\log(N/P) + \sqrt{\log^2(N/P) + 4\log P}) - \log(N/P)\right) \\
&= \sqrt{\log^2(N/P) + 4\log P} - \log(N/P)
\end{aligned}$$

and the Theorem follows. □

Corollary 4.3 *For $\ell(P) = \log P$, the* EREW-EREW FFT-Two-Level *algorithm can be computed in time*

$$O\left(\frac{N\log N\log(N/P)}{P}\right)$$

for $P \cdot 2^{2\sqrt{\log P}} \leq N$, and in time

$$O\left(\frac{N\log N\sqrt{\log P}}{P}\right)$$

for $P \cdot 2^{2\sqrt{\log P}} \geq N$.

Proof: If $4\log P \leq \log^2(N/P)$, then

$$\begin{aligned}
&\sqrt{\log^2(N/P) + 4\log P} - \log(N/P) \\
\leq\ &\sqrt{2\log^2(N/P)} - \log(N/P) \\
=\ &0.41 \cdot \log(N/P)
\end{aligned}$$

and $4\log P \leq \log^2(N/P)$ simplifies to

$$\begin{aligned}
\sqrt{4\log P} &\leq \log(N/P) \\
2^{2\sqrt{\log P}} &\leq N/P \\
P \cdot 2^{2\sqrt{\log P}} &\leq N
\end{aligned}$$

Similarly, if $4\log P \geq \log^2(N/P)$ (i.e. if $P \cdot 2^{2\sqrt{\log P}} \geq N$), then

$$\begin{aligned} & \sqrt{\log^2(N/P) + 4\log P} - \log(N/P) \\ \leq\ & \sqrt{8\log P} - \log(N/P) \\ \leq\ & \sqrt{8\log P} = 2.83 \cdot \sqrt{\log P} \end{aligned}$$

The Corollary follows from substituting $O(\log(N/P))$ or $O(\sqrt{\log P})$ for $\sqrt{\log^2(N/P) + 4\log P} - \log(N/P)$ in Theorem 4.7. □

Compare the times in Corollary 4.3 to the complexity which was obtained when $k = N/P$:

$$O\left(\frac{N\log N}{P} \cdot \frac{\log P}{\log(N/P)}\right)$$

which is also the complexity of the FFT graph algorithm on the LPRAM [1]. We see that neighborhood locality can indeed be used to improve performance for N not so "significantly" larger than P (i.e. when N and P are such that optimality is not achievable).

We now apply the term minimization strategy to the EREW-EREW FFT-Two-Level algorithm for the other latency we consider in this paper, $\ell(P) = \sqrt{P}$.

Lemma 4.2 *When* $\ell(P) = \sqrt{P}$,

$$\ell(\frac{k}{N/P}) \approx \frac{\ell(P)}{\log k}$$

when k *is chosen such that* $k = N/\log^2 N$. *However, in order for this* k *to be legal* ($N/P \leq k \leq N$), *it is necessary that* $N \leq 2^{\sqrt{P}}$.

Proof: We want to choose k such that

$$\sqrt{\frac{k}{N/P}} = \frac{\sqrt{P}}{\log k}$$

which simplifies to $k\log^2 k = N$. Since $k = O(N/\log^2 N)$ when $k\log^2 k = N$, we choose to fix $k = N/\log^2 N$. Now it needs to be checked that $N/P \leq k \leq N$. Clearly $N/\log^2 N \leq N$ so we check that $N/P \leq N/\log^2 N$. This simplifies to $N \leq 2^{\sqrt{P}}$, which must hold in order to apply the term minimization strategy. It is also necessary that k be a power of two; if $N/\log^2 N$ is not then we pick the nearest power to it. □

Theorem 4.8 *For* $\ell(P) = \sqrt{P}$, *the* EREW-EREW FFT-Two-Level *algorithm can be computed in time*

$$O\left(\frac{N}{P}\sqrt{P}\right)$$

when $N \leq 2^{\sqrt{P}}$.

Proof: Choosing k as in Lemma 4.2 under its constraint $N \leq 2^{\sqrt{P}}$, we know that $\sqrt{\frac{k}{N/P}} \approx \frac{\sqrt{P}}{\log k}$. Since

$$\frac{\sqrt{P}}{\log k} = \frac{\sqrt{P}}{\log(N/\log^2 N)}$$

is the slightly larger term, the non-optimal factor in the complexity is

$$\leq \frac{\sqrt{P}}{\log(N/\log^2 N)}$$

so the complexity is

$$O\left(\frac{N \log N}{P} \cdot \frac{\sqrt{P}}{\log(N/\log^2 N)}\right) = O\left(\frac{N}{P}\sqrt{P}\right)$$

□

When $\ell(P) = \sqrt{P}$ then, the term minimization strategy can be employed to get a $O((N/P)\sqrt{P})$ time algorithm for $N \leq 2^{\sqrt{P}}$. In comparison, the complexity when k was chosen to be N/P was

$$O\left(\frac{N}{P}\sqrt{P} \cdot \frac{\log N}{\log(N/P)}\right)$$

which is also the complexity of the FFT graph algorithm on the LPRAM.

4.2.2 EREW-LPRAM FFT-Two-Level

We turn to the EREW-LPRAM FFT-Two-Level algorithm, where the level 1 sub-model remains an EREW PRAM, but the level 2 sub-models that are used to compute k-point FFT graphs are now LPRAMs. Recall that the level 2 sub-PRAMs have $P' = \frac{k}{N/P}$ processors each in FFT-Two-Level. The EREW PRAM algorithm for computing a k-point FFT graph with P' processors used $O((k/P')\log k) = O((N/P)\log k)$ computation *and* communication steps. The LPRAM algorithm in [1] uses $O((k/P')\log k) = O((N/P)\log k)$ computation steps and $O(\frac{k \log k}{P' \log(k/P')}) = O(\frac{N \log k}{P \log(N/P)})$ communication steps. In the following we consider the Direct-FFT sub-algorithm in the general FFT-Two-Level algorithm to be this LPRAM algorithm. Using the LPRAM as a sub-model of the H-PRAM allows one to exploit strict and neighborhood locality simultaneously.

Theorem 4.9 *The* EREW-LPRAM FFT-Two-Level *algorithm can be computed in time*

$$O\left(\frac{N \log N}{P}\left(\frac{\ell(k/(N/P))}{\log(N/P)} + \frac{\ell(P)}{\log k}\right)\right)$$

Proof: The FFT-Two-Level sub-algorithm, running on the level 1, P-processor sub-PRAM, takes the same time as it did for the EREW-EREW algorithm:

$$O\left(\log_k N \cdot \frac{N}{P}\ell(P)\right) = O\left(\frac{N \log N}{P} \cdot \frac{\ell(P)}{\log k}\right)$$

From the above discussion of the number of steps in the LPRAM FFT algorithm, it is straightforward that a Direct-FFT sub-algorithm operating in level 2 on $P' = \frac{k}{N/P}$ processors computes a k-point FFT graph in time

$$O\left(\frac{N}{P}\log k + \frac{N \log k}{P \log(N/P)} \cdot \ell(\frac{k}{N/P})\right)$$

and level 2 Direct-FFT sub-algorithms are invoked $\log_k N = \log N / \log k$ times. Therefore the total time taken up by Direct-FFT in the FFT-Two-Level algorithm is

$$O\left(\frac{N \log N}{P}\left(1 + \frac{\ell(k/(N/P))}{\log(N/P)}\right)\right) = O\left(\frac{N \log N}{P} \cdot \frac{\ell(k/(N/P))}{\log(N/P)}\right)$$

Adding this and the above time for the level 1 sub-algorithm together gives the result. □

In the following, we consider the circumstances under which the EREW-LPRAM FFT-Two-Level algorithm is optimal.

Lemma 4.3 *The* EREW-LPRAM FFT-Two-Level *algorithm is optimal if k can be chosen such that the following bounds hold simultaneously.*

1. $\ell(\frac{k}{N/P}) \leq c_1 \cdot \log(N/P)$
2. $k \geq 2^{\epsilon_2 \cdot \ell(P)}$

where c_1 and c_2 are positive constants in the running time, c_1 corresponding to the Direct-FFT *sub-algorithm and c_2 corresponding to the* FFT-Two-Level *sub-algorithm, and $\epsilon_2 = 1/c_2$.*

Theorem 4.10 *For $\ell(P) = \log P$, the* EREW-LPRAM FFT-Two-Level *algorithm is optimal for*

$$P^{\epsilon/(c+1)+1} \leq N$$

where c and ϵ are positive constants such that $\epsilon \leq 1 + 1/c$. ($c = c_1$ and $\epsilon = \epsilon_2 = 1/c_2$, where c_1 and c_2 are the constants in the running time used in Lemma 4.3).

Proof: We know from Lemma 4.3 that the algorithm is optimal when k can be chosen such that (1) $\log(\frac{k}{N/P}) \leq c_1 \log(N/P)$, which solves to $k \leq (N/P)^{c_1+1} = (N/P)^{c+1}$, and (2) $k \geq 2^{\epsilon_2 \log P}$, which solves to $k \geq P^{\epsilon_2} = P^{\epsilon}$. Remembering that k must be within the bounds $N/P \leq k \leq N$, we conclude that the algorithm is optimal if k can be chosen such that

$$\max(N/P, P^{\epsilon}) \leq k \leq \min(N, (N/P)^{c+1})$$

Since N and P are powers of two, we can assume that k also is, as required.

Therefore, optimality exists when $\max(N/P, P^{\epsilon}) \leq \min(N, (N/P)^{c+1})$. Clearly, $N/P \leq N$ and $N/P \leq (N/P)^{c+1}$, so we need to show the conditions under which we know that $P^{\epsilon} \leq \min(N, (N/P)^{c+1})$ holds. Let $\min(N, (N/P)^{c+1}) = (N/P)^{c+1}$. Then $P^{\epsilon} \leq (N/P)^{c+1}$ when

$$\begin{aligned} P^{\epsilon+c+1} &\leq N^{c+1} \\ P^{\epsilon/(c+1)+1} &\leq N \end{aligned}$$

which gives the claimed optimality range. Now consider that $\min(N, (N/P)^{c+1}) = N$. In this case,

$$\begin{aligned} N &\leq (N/P)^{c+1} \\ P^{c+1} &\leq N^{c} \\ P^{1+1/c} &\leq N \end{aligned}$$

Then $P^\epsilon \leq \min(N, (N/P)^{c+1}) = N$ if $P^\epsilon \leq P^{1+1/c}$, i.e. if $\epsilon \leq 1 + 1/c$, which gives the constraints on the constants, and the Theorem follows. □

Note that the smaller ϵ is and the larger c is, the better the optimality range

$$P^{\epsilon/(c+1)+1} \leq N$$

Also, ϵ stands for $1/c_2$ and c for c_1, where c_1 and c_2 are constants in the running time (Lemma 4.3). Therefore we see that the larger the constant factors in the running time are, the better the optimality range.

Assume that $c_1 = c_2$ and denote the common constant by c (so $\epsilon = 1/c$); the optimality range of the EREW-LPRAM FFT-Two-Level algorithm becomes $P^{1/c(c+1)+1} \leq N$. Assume $c = 1$ for demonstration purposes; then the range is $P^{3/2} \leq N$. Now let $c = 2$; here the EREW-LPRAM FFT-Two-Level algorithm's optimality range is $P^{7/6} \leq N$ (remember that the perfect optimality range would be $P \leq N$). We think that this is good evidence for the utility of neighborhood locality, at least for the problem of computing the FFT graph with an underlying logarithmic latency architecture.

The other latency function we consider in this paper is $\ell(P) = \sqrt{P}$, so attention is now turned to how the EREW-LPRAM FFT-Two-Level algorithm behaves in this case.

Theorem 4.11 *For $\ell(P) = \sqrt{P}$, the* EREW-LPRAM FFT-Two-Level *algorithm is optimal for*

$$2^{\epsilon\cdot\sqrt{P}} \leq N$$

where ϵ is a positive constant such that, for a positive constant c, $\epsilon \geq 1 + 1/c$. ($c = c_1$ and $\epsilon = \epsilon_2 = 1/c_2$, where c_1 and c_2 are the constants in the running time used in Lemma 4.3).

Proof: We know from Lemma 4.3 that the algorithm is optimal when k can be chosen such that (1) $\sqrt{\frac{k}{N/P}} \leq c_1 \cdot \log(N/P)$, which solves to

$$k \leq (c_1)^2 \cdot (N/P)\log^2(N/P) = c^2(N/P)\log^2(N/P)$$

and (2) $k \geq 2^{\epsilon_2\cdot\sqrt{P}} = 2^{\epsilon\sqrt{P}}$. Remembering that k must be within the bounds $N/P \leq k \leq N$, we conclude that the algorithm is optimal if k can be chosen such that

$$\max(N/P, 2^{\epsilon\sqrt{P}}) \leq k \leq \min(N, c^2 \cdot (N/P)\log^2(N/P))$$

Since N and P are powers of two, we can assume that k also is, as required.

Therefore, optimality exists when

$$\max(N/P, 2^{\epsilon\sqrt{P}}) \leq \min(N, c^2 \cdot (N/P)\log^2(N/P))$$

Clearly, $N/P \leq N$ and $N/P \leq c^2 \cdot (N/P)\log^2(N/P)$, so we need to show the conditions under which we know that

$$2^{\epsilon\sqrt{P}} \leq \min(N, c^2 \cdot (N/P)\log^2(N/P))$$

holds. Clearly, we need that $2^{\epsilon\sqrt{P}} \leq N$; this establishes the optimality range stated in the Theorem. Now consider the other case, where

$$\min(N, c^2 \cdot (N/P)\log^2(N/P)) = c^2 \cdot (N/P)\log^2(N/P)$$

We need to show that $2^{\epsilon\sqrt{P}} \leq c^2 \cdot (N/P)\log^2(N/P)$. Operating under the knowledge that $2^{\epsilon\sqrt{P}} \leq N$, we know that

$$c^2 \cdot \frac{2^{\epsilon\sqrt{P}}}{P}\left(\log(2^{\epsilon\sqrt{P}}/P)\right)^2 \leq c^2 \cdot (N/P)\log^2(N/P)$$

Then $2^{\epsilon\sqrt{P}} \leq c^2(N/P)\log^2(N/P)$ if

$$\begin{aligned}
2^{\epsilon\sqrt{P}} &\leq c^2 \cdot \frac{2^{\epsilon\sqrt{P}}}{P}\left(\log\left(\frac{2^{\epsilon\sqrt{P}}}{P}\right)\right)^2 \\
\frac{1}{c}\sqrt{P} &\leq \log\left(\frac{2^{\epsilon\sqrt{P}}}{P}\right) \\
\frac{1}{c}\sqrt{P} &\leq \epsilon\sqrt{P} - \log P \\
\log P &\leq \sqrt{P}(\epsilon - \frac{1}{c})
\end{aligned}$$

This holds if $\epsilon \geq 1 + 1/c$, which is the constraint on the constants, and the Theorem follows. □

As before, notice that the smaller ϵ is (and thus the larger c is, since it is necessary that $\epsilon \geq 1 + 1/c$), the better the optimality range. Remember that ϵ stands for $1/c_2$ and c for c_1, where c_1 and c_2 are constants in the running time (Lemma 4.3).

This optimality range, $2^{\epsilon\sqrt{P}} \leq N$, matches that of FFT-BT-Ext for $\ell(P) = \sqrt{P}$, and is an improvement over the one for the LPRAM FFT graph algorithm [1] (and, equivalently, the EREW-EREW FFT-Two-Level algorithm where $k = N/P$), which is $P \cdot 2^{\epsilon\sqrt{P}} \leq N$.

4.2.3 BPRAM-EREW FFT-Two-Level

For the reasons noted at the beginning of this section, we only briefly discuss the results of the BPRAM-EREW FFT-Two-Level algorithm. These results are analogous to those obtained for the EREW-EREW FFT-Two-Level algorithm. Optimality is only obtainable for the $k = N/P$ case of single-processor sub-PRAMs in level 2, and occurs for the range $P \cdot \ell(P) \leq N$. This matches the result for computing the FFT graph on the BPRAM [2], which is to be expected because the BPRAM is an instance of a two-level private H-PRAM corresponding to a BPRAM sub-model in level 1 with single-processor sub-PRAMs in level 2. Term minimization may be used to obtain slightly improved performance for $N < P \cdot \ell(P)$, but is overly complicated.

4.3 Multi-level algorithms

Although the general two-level algorithm is simple and avoids the sums in its analyses that arise from multi-level hierarchies, it does not exploit all of the general locality inherent in the problem

of computing the N-point FFT graph. The general multi-level ($\log_k P$) algorithm presented in this section, although more involved, does allow the exploitation of all inherent general locality; it can be seen as a combination of the two-level algorithm and the binary tree extension algorithm FFT-BT-Ext. As in the previous section, we will consider specific instances of the general algorithm which are obtained by fixing the types of sub-models of the H-PRAM used.

The previously unexploited locality can be seen in the memory management of FFT-Two-Level, specifically in the permutations on memory/points that "untangle" and "retangle" the k-point FFT graphs prior to and following each of the $\log_k N$ stages. Recall that these permutations were performed in the level 1 P-processor (sub)-PRAM, and thus that communication steps implementing the permutation were charged at the full latency $\ell(P)$ of the H-PRAM. However, the permutations were "segmented", i.e. multiple independent permutations were applied to multiple independent blocks of contiguous points/memory, where block sizes grew successively smaller with successive stages. The idea behind the multi-level algorithm is to partition the the H-PRAM in correspondence with the blocks that permutations are applied to, so that each block, and only one block, is within a sub-PRAM's private memory when a permutation is applied. Then permuting can be done at the latency of the sub-PRAM rather than that of the P-processor PRAM at level 1 in the hierarchy.

A better way of explaining is in terms of the structure of the N-point FFT graph first noticed in Section 4.1. Again consider the graph to be partitioned into $\log_k N = \log N / \log k$ stages of height $\log k$. Then note that *after* stage λ, $1 \leq \lambda \leq \log_k N$, has been computed, the remainder of the N-point FFT graph that is yet to be computed consists of k^λ distinct (disjoint) FFT graphs, each of which has N/k^λ points. Broadly speaking, the general multi-level algorithm will compute a stage and then call itself recursively on each of the remaining FFT sub-graphs.

As an example, consider Figure 3. Here we have $N = 16$ and $k = 4$. Note that after stage 1 has been computed, there are $k^1 = 4$ disjoint FFT sub-graphs of $N/k^1 = 4$ points. After stage 2 has been computed, there are $k^2 = 16$ disjoint FFT sub-graphs of $N/k^2 = 1$ point.

The algorithm computes N/k k-point FFT graphs as before, but also partitions the H-PRAM into a k-ary hierarchy. At each level in the hierarchy, sub-PRAM algorithms will permute (untangle), compute k-point FFT graphs (via a **partition** step, as before), and unpermute (retangle). Each stage λ of the N-point graph will be computed by sub-PRAMs in level λ of the hierarchy.

Since the P-processor H-PRAM is being partitioned into a k-ary hierarchy, and $P \leq N$, we have to partition the N-point FFT graph slightly differently. The straightforward way is to say that it is partitioned into $\log_k P + 1 = \log P / \log k + 1$ stages, where stage λ, $1 \leq \lambda \leq \log_k P$, has height $\log k$, and stage $\log_k P + 1$ has height $\log(N/P)$. Here, in level λ of the hierarchy, $1 \leq \lambda \leq \log_k P$, k-point FFT graphs will be computed, and in level $\log_k P + 1$ 1-processor sub-PRAMs will compute (N/P)-point FFT graphs.

However, we wish to allow for the case that $k > P$ (although $k \leq N$), since maximizing the partitioning flexibility (parameterized by k) maximizes the H-PRAM's ability to adapt to different N, P, and cost functions. Define $\log_k x = 1$ when $k > x$. Then we say that the N-point FFT graph

is partitioned into $\lceil \log_k P \rceil + 1$ stages, where stage λ, $1 \leq \lambda \leq \lceil \log_k P \rceil$, has height $\log k$, and stage $\lceil \log_k P \rceil + 1$ has height $\log N - \lceil \log_k P \rceil \log k$. If $k > P$, there will be two stages, the first having height $\log k$ and the second having height $\log N - \log k = \log(N/k)$.

We give a pseudo-code outline of the general multi-level algorithm below, followed by further discussion, after which an understanding of the **FFT-Multi-Level** algorithm and its correctness should be immediate when it is given. The sub-algorithm **Seq-FFT(x)** is a standard sequential algorithm that computes an x-point FFT graph on a 1-processor sub-PRAM.

```
Pseudo-FFT-Multi-Level(P', λ)
    < Compute N/k^λ k-point FFT graphs >
    < Comment: note that there are k^(λ-1) sub-PRAMs in level λ,
    and k^(λ-1) · N/k^λ · k = N >
    if λ < log_k P then
       partition{k, P'/k, Pseudo-FFT-Multi-Level(P'/k, λ + 1) }
    else
       if k ≤ P then
          < Comment: note that P'/k = 1 and N/k^λ = N/P >
          partition{k, 1, Seq-FFT(N/P)}
       else
          < Comment: note that N/k^λ = N/k >
          partition{P, 1, Seq-FFT(N/k)}
       end-if-then-else
    end-if-then-else
end-Pseudo-FFT-Multi-Level
```

The (pseudo) H-PRAM algorithm is invoked by **Pseudo-FFT-Multi-Level(P, 1)**. As for FFT-BT-Ext, there are $k^{\lambda-1}$ sub-PRAMs in levels λ, $1 \leq \lambda \leq \log_k P$. If $k \leq P$, there are $k^{\lambda-1} = P$ sub-PRAMs in level $\lambda = \log_k P + 1$. If $k > P$, there will be a two-level hierarchy with one P-processor sub-PRAM in level 1 and P 1-processor sub-PRAMs in level 2. Consider the point where $\lambda = \log_k P$. Note that if $k \leq P$ we have computed $\log P$ levels of the FFT graph and have yet to compute the remaining $\log(N/P)$ levels. $k^\lambda = P$ sub-PRAMs in level λ then compute these remaining levels by independently computing $N/k^\lambda = N/P$ point FFT graphs. If $k > P$, then when $\lambda = \log_k P = 1$ we have computed $\log k$ levels and have yet to compute the remaining $\log(N/k)$ levels. There are $k > P$ independent $N/k < N/P$ point FFT graphs to be computed. Since P is the maximum number of sub-PRAMs usable, P of them are created, each of which computes an (N/k)-point FFT graph.

We now turn attention to the computation of k-point FFT graphs in levels $1 \leq \lambda \leq \log_k P$.

Sub-PRAMs in level λ collectively compute stage λ of the N-point FFT graph in the same way that the two-level algorithm did; by permuting, partitioning, and unpermuting. There are $k^{\lambda-1}$ sub-PRAMs running Pseudo-FFT-Multi-Level in level λ; each is responsible for computing N/k^{λ} k-point FFT graphs. Therefore each stage is computed as N/k independent k-point FFT graphs since $(N/k^{\lambda}) \cdot k^{\lambda-1} = N/k$.

The general FFT-Multi-Level algorithm is given below. Note that when a $P' = P/k^{\lambda-1}$ processor sub-PRAM partitions itself to create N/k^{λ} sub-sub-PRAMs for computing k-point FFT graphs, the sub-sub-PRAMs have

$$\frac{P'}{N/k^{\lambda}} = \frac{P/k^{\lambda-1}}{N/k^{\lambda}} = \frac{Pk}{N} = \frac{k}{N/P}$$

processors, as in the two-level algorithm. As before, the Direct-FFT(x) sub-algorithm computes a k-point FFT graph using x processors.

FFT-Multi-Level(P', λ)
    < k-shuffle on the $N/k^{\lambda-1}$ points in memory >
    partition{N/k^{λ}, $\frac{k}{N/P}$, Direct-FFT($\frac{k}{N/P}$) }
    < k-unshuffle on the $N/k^{\lambda-1}$ points in memory >
    if $\lambda < \log_k P$ then
        partition{k, P'/k, FFT-Multi-Level($P'/k, \lambda+1$) }
    else
        partition{ min(k, P), 1, Seq-FFT(N/k^{λ}) }
    end-if-then-else
end-FFT-Multi-Level

We again use the notation TYPE1-TYPE2 to refer to specific instances of the general algorithm, where FFT-Multi-Level runs on TYPE1 sub-models, and Direct-FFT runs on TYPE2 sub-models. Two specific algorithms are considered: EREW-EREW and EREW-LPRAM.

We first consider the β-synchronization complexity of FFT-Multi-Level since it is independent of the types of sub-models used.

Theorem 4.12 *The β-synchronization complexity of* FFT-Multi-Level *will never dominate.*

Proof: The FFT-Multi-Level(P', λ) sub-algorithms in the hierarchy each have two partition steps and two permutation steps. Therefore the conditions for β-synchronization non-domination are met for each sub-algorithm, and we conclude that β-synchronization does not dominate the complexity of the FFT-Multi-Level(P, 1) H-PRAM algorithm. □

Therefore we again drop consideration of β-synchronization from the subsequent analyses of the FFT-Multi-Level algorithms.

4.3.1 EREW-EREW FFT-Multi-Level

Theorem 4.13 *The* EREW-EREW FFT-Multi-Level *algorithm can be computed in time*

$$O\left(\frac{N}{P}\cdot\left(\log P\cdot\ell(\frac{k}{N/P})+\sum_{\lambda=1}^{\log_k P}\ell(k^\lambda)+\log(N/P)\right)\right)$$

Proof: Clearly the Seq-FFT sub-algorithm running on 1-processor sub-PRAMs in level $\log_k P+1$ will take time $O((N/P)\log(N/P))$ or $O((N/k)\log(N/k))$ depending on whether $k\le P$ or $k>P$, respectively. Since $N/P > N/k$ when $k > P$ we just use the $O((N/P)\log(N/P))$ form. The EREW PRAM Direct-FFT sub-algorithm is executed $\log_k P$ times (in levels $2,\ldots,\log_k P+1$ of the hierarchy); each time employing $P'=\frac{k}{N/P}$ processors computing a k-point FFT graph in the straightforward way. Thus the total time taken by Direct-FFT sub-algorithms is

$$\begin{aligned}
& O\left(\log_k P\cdot\left(\frac{k\log k}{P'}+\frac{k\log k}{P'}\cdot\ell(\frac{k}{N/P})\right)\right)\\
= \; & O\left(\frac{\log P}{\log k}\cdot\left(\frac{N}{P}\log k\cdot\ell(\frac{k}{N/P})\right)\right)\\
= \; & O\left(\frac{N\log P}{P}\cdot\ell(\frac{k}{N/P})\right)
\end{aligned}$$

Lastly, the FFT-Multi-Level sub-algorithm runs in levels λ, $1\le\lambda\le\log_k P+1$, and executes permutations (in addition to partition steps, already accounted for). Permuting N' elements with P' ($\le N'$) processors takes time $O((N'/P')\ell(P'))$ on an EREW sub-model. In level λ there are $k^{\lambda-1}$ sub-PRAMs running FFT-Multi-Level; each has $P/k^{\lambda-1}$ processors and $N/k^{\lambda-1}$ points in its memory. Thus the total time taken by FFT-Multi-Level *sub*-algorithms in the EREW-EREW FFT-Multi-Level *algorithm* is

$$O\left(\sum_{\lambda=1}^{\log_k P}\frac{N/k^{\lambda-1}}{P/k^{\lambda-1}}\cdot\ell(P/k^{\lambda-1})\right)=O\left(\frac{N}{P}\sum_{\lambda=1}^{\log_k P}\ell(P/k^{\lambda-1})\right)$$

By reversing the sum, we get

$$O\left(\frac{N}{P}\sum_{\lambda=1}^{\log_k P}\ell(k^\lambda)\right)$$

Adding the total times of the three sub-algorithms gives the result. □

Lemma 4.4 *The* EREW-EREW FFT-Multi-Level *algorithm is optimal if k can be chosen such that the following bounds hold simultaneously.*

1. $\ell(\frac{k}{N/P})\le c_1\cdot\frac{\log N}{\log P}$
2. $\sum_{\lambda=1}^{\log_k P}\ell(k^\lambda)\le c_2\cdot\log N$

where c_1 and c_2 are positive constants in the running time, c_1 corresponding to the Direct-FFT *sub-algorithm and c_2 corresponding to the* FFT-Multi-Level *sub-algorithm.*

We again consider the explicit latency functions $\ell(P) = \log P$ and $\ell(P) = \sqrt{P}$ in order to remove the sums from the analyses and get simpler and more informative results.

Theorem 4.14 *Let $c = c_1$ and $\epsilon = 1/c_2$, where c_1 and c_2 are the constants in the running time used in Lemma 4.4, and let $a = (\log P + c)/\log P$. Then, for $\ell(P) = \log P$ the* EREW-EREW FFT-Multi-Level *algorithm is optimal for*

$$P^{(1+\sqrt{1+4a\epsilon})/2a} \leq N$$

under the restriction that $\epsilon \leq 1$ ($c_2 \geq 1$).

Proof: We know from Lemma 4.4 that the algorithm is optimal if k can be chosen such that (1) $\log(\frac{k}{N/P}) \leq c_1 \cdot \log N/\log P$, which, letting $c = c_1$, solves to

$$\begin{aligned} \log k &\leq c \cdot \frac{\log N}{\log P} + \log(N/P) \\ k &\leq \frac{N}{P} \cdot N^{c/\log P} \end{aligned}$$

and (2)

$$\sum_{\lambda=1}^{\log_k P} \log(k^\lambda) \leq c_2 \cdot \log N$$

which, letting $\epsilon = 1/c_2$ and noting that

$$\sum_{\lambda=1}^{\log_k P} \log(k^\lambda) = O\left(\frac{\log^2 P}{\log k}\right)$$

solves as

$$\begin{aligned} \frac{\log^2 P}{\log k} &\leq c_2 \cdot \log N \\ \epsilon \cdot \frac{\log^2 P}{\log N} &\leq \log k \\ P^{\epsilon \cdot \log P/\log N} &\leq k \end{aligned}$$

Remembering that k must be within the bounds $N/P \leq k \leq N$, we conclude that the algorithm is optimal if k can be chosen such that

$$\max\left(N/P, P^{\epsilon \cdot \log P/\log N}\right) \leq k \leq \min\left(N, \frac{N}{P} \cdot N^{c/\log P}\right)$$

Since N and P are powers of two, we can assume that k also is, as required. Thus, the algorithm is optimal when

$$\max\left(N/P, P^{\epsilon \cdot \log P/\log N}\right) \leq \min\left(N, \frac{N}{P} \cdot N^{c/\log P}\right)$$

Clearly, $N/P \leq N$ and $N/P \leq \frac{N}{P} \cdot N^{c/\log P}$. Also $P^{\epsilon \cdot \log P/\log N} \leq N$ when

$$\begin{aligned} \epsilon \cdot \frac{\log^2 P}{\log N} &\leq \log N \\ \sqrt{\epsilon} \cdot \log P &\leq \log N \end{aligned}$$

which holds for all $P \leq N$ when $\epsilon \leq 1$ (or $c_2 \geq 1$ since $\epsilon = 1/c_2$), which gives the restriction in the Theorem.

The final case that must hold is

$$P^{\epsilon \cdot \log P/\log N} \leq \frac{N}{P} \cdot N^{c/\log P}$$

which we now proceed to simplify to the optimality range stated in the Theorem.

$$\begin{aligned} P^{\epsilon \cdot (\log P/\log N)+1} &\leq N^{(c/\log P)+1} \\ \log P \left(\frac{\epsilon \cdot \log P}{\log N} + 1\right) &\leq \log N \left(\frac{c}{\log P} + 1\right) \\ \frac{\epsilon \cdot \log P}{\log N} + 1 &\leq \frac{\log N}{\log P}\left(\frac{\log P + c}{\log P}\right) \end{aligned}$$

For presentational purposes, let $x = \log N/\log P$ and let $a = (\log P + c)/\log P$. Then we have

$$\begin{aligned} \frac{\epsilon}{x} + 1 &\leq xa \\ \epsilon + x &\leq x^2 a \\ x^2 a - x - \epsilon &\geq 0 \end{aligned}$$

Applying the quadratic formula to the equation gives

$$x = \frac{\log N}{\log P} \geq \frac{1 + \sqrt{1 + 4a\epsilon}}{2a}$$

Therefore, with $a = (\log P + c)/\log P$, k is choosable such that the algorithm is optimal when

$$\begin{aligned} \log P \cdot \frac{1 + \sqrt{1 + 4a\epsilon}}{2a} &\leq \log N \\ P^{(1+\sqrt{1+4a\epsilon})/2a} &\leq N \end{aligned}$$

and the Theorem follows. □

Now the behavior of the EREW-EREW FFT-Multi-Level algorithm when $\ell(P) = \sqrt{P}$ is considered.

Theorem 4.15 *For $\ell(P) = \sqrt{P}$, the* EREW-EREW FFT-Multi-Level *algorithm is optimal for $2^{\epsilon \cdot \sqrt{P}} \leq N$, where $\epsilon = 1/c_2$ and c_2 is a constant in the running time (corresponding to the c_2 in Lemma 4.4).*

Proof: We know from Lemma 4.4 that the algorithm is optimal if k can be chosen such that (1) $\sqrt{\frac{k}{N/P}} \leq c_1 \cdot \log N/\log P$, which solves to

$$k \leq (c_1)^2 \cdot \frac{N \log^2 N}{P \log^2 P}$$

and (2)

$$\sum_{\lambda=1}^{\log_k P} \sqrt{k^\lambda} \leq c_2 \cdot \log N$$

which, noting that

$$\sum_{\lambda=1}^{\log_k P} \sqrt{k^\lambda} = O(\sqrt{P})$$

solves to

$$\begin{aligned} \sqrt{P} &\leq c_2 \cdot \log N \\ 2^{\epsilon\sqrt{P}} &\leq N \end{aligned}$$

where $\epsilon = 1/c_2$. Remembering that k is constrained by $N/P \leq k \leq N$, we see that the algorithm is optimal for $2^{\epsilon\sqrt{P}} \leq N$ if k can be chosen such that

$$N/P \leq k \leq \min(N, (c_1)^2 \cdot \frac{N \log^2 N}{P \log^2 P})$$

This is clearly possible since N/P is $\leq$ both terms of the min function. Since N and P are powers of two, we can assume that k also is, as required. □

We again note that $2^{\epsilon\sqrt{P}} \leq N$ is a substantial improvement in the optimality range, for $\ell(P) = \sqrt{P}$, over the $P \cdot 2^{\epsilon\sqrt{P}} \leq N$ optimality range of both the EREW-EREW FFT-Two-Level algorithm, and the LPRAM FFT graph algorithm [1]. It matches the FFT-BT-Ext and EREW-LPRAM FFT-Two-Level ranges for $\ell(P) = \sqrt{P}$.

The term minimization strategy can be employed to choose k for the EREW-EREW FFT-Multi-Level algorithm similarly to the way it was for the EREW-EREW FFT-Two-Level algorithm, giving very similar results. We only summarize them here. For $\ell(P) = \log P$, the complexities are

$$O\left(\frac{N \log P \log(N/P)}{P}\right)$$

for $P \cdot 2^{2\sqrt{\log P}} \leq N$, and

$$O\left(\frac{N \log P \sqrt{\log P}}{P}\right)$$

for $P \cdot 2^{2\sqrt{\log P}} \geq N$. For $\ell(P) = \sqrt{P}$ the complexity is

$$O\left(\frac{N}{P}(\sqrt{P} + \log(N/P))\right)$$

We again note that term minimization is meant to be used when N and P are values such that optimal time $O((N/P)\log N)$ cannot be achieved.

4.3.2 EREW-LPRAM FFT-Multi-Level

Theorem 4.16 *The* EREW-LPRAM FFT-Multi-Level *algorithm can be computed in time*

$$O\left(\frac{N}{P}\cdot\left(\log P+\frac{\log P}{\log(N/P)}\cdot\ell(\frac{k}{N/P})+\sum_{\lambda=1}^{\log_k P}\ell(k^\lambda)+\log(N/P)\right)\right)$$

Proof: The only difference from the EREW-EREW algorithm is that the sub-models that k-point FFT graphs are computed on are now LPRAMs rather than EREW PRAMs. The total time taken by the FFT-Multi-Level *sub*-algorithms and Seq-FFT sub-algorithms is the same, which we know from Theorem 4.13 to be

$$O\left(\frac{N}{P}\sum_{\lambda=1}^{\log_k P}\ell(k^\lambda)+\frac{N}{P}\log(N/P))\right)$$

The LPRAM Direct-FFT sub-algorithm is executed $\log_k P$ times (in levels $2,\ldots,\log_k P+1$ of the hierarchy); each time employing $P'=\frac{k}{N/P}$ processors computing a k-point FFT graph. The LPRAM algorithm in [1] uses $O((k/P')\log k)=O((N/P)\log k$ computation steps and $O(\frac{k\log k}{P'\log(k/P')})=O(\frac{N\log k}{P\log(N/P)})$ communication steps. Thus the total time taken by Direct-FFT sub-algorithms is

$$\begin{aligned}
& O\left(\log_k P\cdot\left(\frac{k\log k}{P'}+\frac{k\log k}{P'\log(k/P')}\cdot\ell(\frac{k}{N/P})\right)\right)\\
= \; & O\left(\frac{\log P}{\log k}\cdot\left(\frac{N}{P}\log k+\frac{N\log k}{P\log(N/P)}\cdot\ell(\frac{k}{N/P})\right)\right)\\
= \; & O\left(\frac{N}{P}\cdot\left(\log P+\frac{\log P}{\log(N/P)}\cdot\ell(\frac{k}{N/P})\right)\right)
\end{aligned}$$

Adding the total times of the three sub-algorithms gives the result. □

Lemma 4.5 *The* EREW-LPRAM FFT-Multi-Level *algorithm is optimal if k can be chosen such that the following bounds hold simultaneously.*

1. $\ell(\frac{k}{N/P})\le c_1\cdot\frac{\log N\log(N/P)}{\log P}$
2. $\sum_{\lambda=1}^{\log_k P}\ell(k^\lambda)\le c_2\cdot\log N$

where c_1 and c_2 are positive constants in the running time, c_1 corresponding to the Direct-FFT *sub-algorithm and c_2 corresponding to the* FFT-Multi-Level *sub-algorithm.*

We turn to considering the explicit latency functions $\ell(P)=\log P$ and $\ell(P)=\sqrt{P}$, as usual, in order to remove the sums from the analyses and get simpler and more informative results.

For $\ell(P)=\log P$, finding the optimality range $f(P)\le N$ requires solving a cubic equation. Because of the difficulty of this, we define the range in terms of the cubic equation in the following Theorem, and subsequently calculate a few explicit range results.

Theorem 4.17 *Let $c = c_1$ and $\epsilon = 1/c_2$, where c_1 and c_2 are the constants in the running time used in Lemma 4.5, let $x = \log N / \log P$, and let z denote the positive solution to the cubic equation*

$$cx^3 + (1-c)x^2 - x - \epsilon \geq 0$$

Then, for $\ell(P) = \log P$, the EREW-LPRAM FFT-Multi-Level *algorithm is optimal for $P^z \leq N$, under the restriction that $\epsilon \leq 1$ ($c_2 \geq 1$).*

Proof: We know from Lemma 4.5 that the algorithm is optimal if k can be chosen such that (1) $\log(\frac{k}{N/P}) \leq c_1 \cdot \frac{\log N \log(N/P)}{\log P}$, which, letting $c = c_1$, solves to

$$\begin{aligned}
\log k &\leq c \cdot \frac{\log N \log(N/P)}{\log P} + \log(N/P) \\
&= \log(N/P)(c \cdot \frac{\log N}{\log P} + 1) \\
k &\leq (\frac{N}{P})^{c \cdot \log N / \log P + 1} \\
&= \frac{N}{P} \cdot \frac{N^{c \cdot \log N / \log P}}{P^{c \cdot \log N / \log P}} \\
&= \frac{N}{P} \cdot \frac{N^{c \cdot \log N / \log P}}{N^c} \\
k &\leq \frac{N}{P} \cdot N^{c \cdot (\log N / \log P - 1)}
\end{aligned}$$

and (2)

$$\sum_{\lambda=1}^{\log_k P} \log(k^\lambda) \leq c_2 \cdot \log N$$

which, letting $\epsilon = 1/c_2$ and noting that

$$\sum_{\lambda=1}^{\log_k P} \log(k^\lambda) = O\left(\frac{\log^2 P}{\log k}\right)$$

solves as

$$\begin{aligned}
\frac{\log^2 P}{\log k} &\leq c_2 \cdot \log N \\
\epsilon \cdot \frac{\log^2 P}{\log N} &\leq \log k \\
P^{\epsilon \cdot \log P / \log N} &\leq k
\end{aligned}$$

Remembering that k must be within the bounds $N/P \leq k \leq N$, we conclude that the algorithm is optimal if k can be chosen such that

$$\max\left(N/P, P^{\epsilon \cdot \log P / \log N}\right) \leq k \leq \min\left(N, \frac{N}{P} \cdot N^{c \cdot (\log N / \log P - 1)}\right)$$

Since N and P are powers of two, we can assume that k also is, as required. Thus, the algorithm is optimal when

$$\max\left(N/P, P^{\epsilon \cdot \log P/\log N}\right) \leq \min\left(N, \frac{N}{P} \cdot N^{c\cdot(\log N/\log P - 1)}\right)$$

Clearly, $N/P \leq N$ and $N/P \leq \frac{N}{P} \cdot N^{c\cdot(\log N/\log P-1)}$. Also $P^{\epsilon \cdot \log P/\log N} \leq N$ when

$$\begin{aligned} \epsilon \cdot \frac{\log^2 P}{\log N} &\leq \log N \\ \sqrt{\epsilon} \cdot \log P &\leq \log N \end{aligned}$$

which holds for all $P \leq N$ when $\epsilon \leq 1$, (or $c_2 \geq 1$ since $\epsilon = 1/c_2$), which gives the restriction in the Theorem.

The final case that must hold is

$$P^{\epsilon \cdot \log P/\log N} \leq \frac{N}{P} \cdot N^{c\cdot(\log N/\log P-1)}$$

which we now proceed to simplify to the optimality range stated in the Theorem.

$$\begin{aligned} P^{\epsilon\cdot(\log P/\log N)+1} &\leq N^{c\cdot(\log N/\log P-1)+1} \\ \log P\left(\frac{\epsilon \cdot \log P}{\log N} + 1\right) &\leq \log N\left(c \cdot \left(\frac{\log N}{\log P} - 1\right) + 1\right) \\ \frac{\epsilon \cdot \log P}{\log N} + 1 &\leq \frac{\log N}{\log P}\left(c \cdot \left(\frac{\log N}{\log P} - 1\right) + 1\right) \end{aligned}$$

For presentational purposes, let $x = \log N/\log P$. Then we have

$$\begin{aligned} \frac{\epsilon}{x} + 1 &\leq x \cdot (c(x-1)+1) \\ \epsilon + x &\leq x^2 \cdot (cx - c + 1) \\ cx^3 + (1-c)x^2 - x - \epsilon &\geq 0 \end{aligned}$$

which is the cubic equation stated in the Theorem. Let z be the positive solution of the equation. Then $x = \frac{\log N}{\log P} \geq z$, which solves to

$$\begin{aligned} z \cdot \log P &\leq \log N \\ P^z &\leq N \end{aligned}$$

and the Theorem follows. A very loose upper bound can be established for z. We are looking for a solution $x > 1$ to the cubic equation. Reformulate it as

$$cx^3 + (1-c)x^2 - x \geq \epsilon$$

and note that this holds if

$$cx^3 - cx^2 + 1 - x \geq \epsilon$$

holds (for $x > 1$). Then we get

$$\begin{aligned} cx^2(x-1) - 1 \cdot (x-1) &\geq \epsilon \\ (x-1)(cx^2-1) &\geq \epsilon \end{aligned}$$

Assume that $c \geq 1$ (clearly reasonable since c is a constant in the running time). Then the last equation holds if

$$(x-1)(cx^2 - c) \geq \epsilon$$

which reformulates as

$$\begin{aligned} (x-1)(x^2-1) &\geq \epsilon/c \\ (x-1)(x-1)(x+1) &\geq \epsilon/c \end{aligned}$$

This last equation holds if

$$\begin{aligned} (x-1)^3 &\geq \epsilon/c \\ x &\geq 1 + (\epsilon/c)^{1/3} \end{aligned}$$

Thus $z \leq 1 + (\epsilon/c)^{1/3}$ is a solution, and provides a very loose (due to the simplifying assumptions) upper bound on z. □

By calculating solutions to the cubic equation for certain c_1 and c_2 (constants in the running time; see Lemma 4.5), it can be seen that the optimality range for the EREW-LPRAM FFT-Multi-Level algorithm is better than that of the EREW-LPRAM FFT-Two-Level algorithm for $\ell(P) = \log P$. For example, for $c_1 = c_2 = 1$ we get the range $P^{4/3} \leq N$ (compare to $P^{3/2} \leq N$ for the two-level algorithm), and for $c_1 = c_2 = 2$ we get the range $P^{8/7} \leq N$ (compare to $P^{7/6} \leq N$ for the two-level algorithm). Again we see that the larger the constants in the running time, the better the optimality range.

Now we turn attention to the behavior of the EREW-LPRAM FFT-Multi-Level algorithm for $\ell(P) = \sqrt{P}$.

Theorem 4.18 *For $\ell(P) = \sqrt{P}$, the* EREW-LPRAM FFT-Multi-Level *algorithm is optimal for $2^{\epsilon\sqrt{P}} \leq N$, where $\epsilon = 1/c_2$ and c_2 is a constant in the running time (corresponding to the c_2 in Lemma 4.5).*

Proof: We know from Lemma 4.5 that the algorithm is optimal if k can be chosen such that (1) $\sqrt{\frac{k}{N/P}} \leq c_1 \cdot \frac{\log N \log(N/P)}{\log P}$, which solves to

$$k \leq (c_1)^2 \cdot \frac{N \log^2 N \log^2(N/P)}{P \log^2 P}$$

and (2)

$$\sum_{\lambda=1}^{\log_k P} \sqrt{k^\lambda} \leq c_2 \cdot \log N$$

which, noting that

$$\sum_{\lambda=1}^{\log_k P} \sqrt{k^\lambda} = O(\sqrt{P})$$

solves to

$$\begin{aligned} \sqrt{P} &\leq c_2 \cdot \log N \\ 2^{\epsilon\sqrt{P}} &\leq N \end{aligned}$$

where $\epsilon = 1/c_2$. Remembering that k is constrained by $N/P \leq k \leq N$, we see that the algorithm is optimal for $2^{\epsilon\sqrt{P}} \leq N$ if k can be chosen such that

$$N/P \leq k \leq \min\left(N, (c_1)^2 \cdot \frac{N \log^2 N \log^2(N/P)}{P \log^2 P}\right)$$

This is clearly possible since N/P is $\leq$ both terms of the min function. Since N and P are powers of two, we can assume that k also is, as required. □

Note that this optimality range is the same as for the FFT-BT-Ext, EREW-LPRAM FFT-Two-Level, and EREW-EREW FFT-Multi-Level algorithms.

5 Conclusions and discussion

Although a key emphasis of this paper was to demonstrate usage of the H-PRAM, and the results of using different combinations of sub-model types and hierarchical structures, it is informative to summarize a few of the "best" complexity results here. For the binary tree problem: when $\ell(P) = \log P$, time $O(\log^{3/2} P + N/P)$ (optimal for $\epsilon P \log^{3/2} P \leq N$, $\epsilon = 1/c$, c a constant in the running time). When $\ell(P) = \sqrt{P}$, time $O(\sqrt{P} + N/P)$ (optimal for $\epsilon P^{3/2} \leq N$, $\epsilon = 1/c$, c a constant in the running time).

For the FFT graph problem: when $\ell(P) = \sqrt{P}$, optimal time $O((N/P)\log N)$ is achieved for $2^{\epsilon\sqrt{P}} \leq N$, $\epsilon = 1/c$, c a constant in the running time. This is an improvement over the $P \cdot 2^{\epsilon\sqrt{P}} \leq N$ optimality range of the case where only strict locality is used (the LPRAM [1]). When $\ell(P) = \log P$ and β-synchronization cost is logarithmic in the number of sub-PRAMs being synchronized, optimality is achieved for $P^z \leq N$, where z approaches 1 as two constants c_1 and c_2 in the running time grow. When $c_1 = c_2 = 2$ the optimality range is $P^{8/7} \leq N$.

The algorithms presented in this and the following sections demonstrate a methodology of building on existing algorithms to obtain new results. In other words, the H-PRAM provides a means of organizing various existing algorithms such that communication and synchronization overhead is "minimized", which can result in a composite algorithm that performs better than its individual sub-algorithms do alone. Although we give H-PRAM algorithms for basic problems for study and comparison purposes, the H-PRAM should allow conceptually simple construction of "larger", more complex parallel algorithms comprised of many simple sub-algorithms. It allows us to build on the work which already exists with respect to the PRAM. In other words, we really do

not want to design H-PRAM versions of common (and optimal) PRAM algorithms, i.e. reinvent the wheel, even if those PRAM algorithms are not optimal on the H-PRAM, but to use them as building blocks for designing "larger" (and optimal) H-PRAM algorithms. We anticipate that basic algorithms, such as for prefix and list ranking, would be provided as primitives (employing network topology) in any implementation of an H-PRAM to architecture mapping. The point here is that the H-PRAM allows the modular construction of large, (controlled) asynchronous, complex systems from simple synchronous PRAM algorithms, and we wish to make full use of the large body of work that exists on PRAM computations.

We want to stress that algorithm design and analysis seems relatively simple; it appears that the implicit hierarchical organization controls any additional conceptual complexity over the PRAM model. What can be difficult, as seen in the following section, is choosing the best configuration of the H-PRAM given an H-PRAM algorithm, input size N, and number of processors P. However, in a computer system that supports the H-PRAM, automated tools could do this. The general philosophy in (private) H-PRAM algorithm design should be to obtain algorithms that have the *greatest possible partitioning flexibility.* This means defining the loosest possible upper and lower bounds on the values that the "partitioning parameter" k can take on. The algorithm complexities will be in terms of k, N, and P. Then, given a value for N (the input size that a user want to compute on) and a value for P (the number of processors available to the user), the value for k that minimizes the complexity can be chosen; as stated, potentially by automated tools.

In theoretical work one can attempt to find a value, or range of values, for k (as a function of N and P) such that good (optimal) performance is obtained for the widest possible range of values for N and P. This is what we will do in the following section, in effect to maximize the number of processors that are *efficiently* usable with respect to an input size N, and to minimize the inefficiency when optimality is not possible (when P is too large with respect to N). This is possible because of the H-PRAM's representation of general locality, i.e. both strict and neighborhood locality.

When N and P are such that optimality ranges hold for multiple fixed instances of the latency parameter (e.g. $\log P$, $\sqrt{P}$), then the H-PRAM algorithm is architecture independent without loss of efficiency across architectures with those latencies. Similarly, when optimality is not possible but the inefficiency is within a certain bound for multiple latencies, the algorithm could be considered architecture independent with bounded inefficiency across architectures with those latencies. Kruskal, Rudolph, and Snir [9] have proposed a complexity theory of parallel algorithms based on preservation of efficiency across multiple architectures.

There is the potential that *general* H-PRAM algorithms can be designed, with unspecified types of sub-models and unspecific sub-algorithms (one defines *what* the sub-algorithms do but not *how* they go about it), in order to gain an additional degree of architecture independence. Once a target architecture is known, one can choose the type(s) of sub-models(s) that most reflect it (e.g. PRAM, LPRAM, BPRAM), then choose specific sub-algorithms that have been designed for that/those sub-model(s). In other words, one general H-PRAM algorithm may have various specific instances of it, as demonstrated by the FFT graph algorithms of the following section.

The private H-PRAM provides a memory management paradigm that lies between the extremes of totally automated (PRAM) and totally manual (networks). This is one reason for our belief, as stated in the introduction, that the H-PRAM provides a good balance between simplicity of usage and reflectivity of realistic architectures. The paradigm is one where memory is seen as a linear block of memory locations, and there is responsibility for organizing that block into groups (by permuting the memory), but not for the details of implementing the organizing (i.e. for routing data in a network).

Clearly, the private H-PRAM with a *tree* hierarchy relation is naturally suited for problems that can be solved by divide-and-conquer. Preparata and Vuillemin [10] have defined ASCEND and DESCEND classes of algorithms, which operate in a divide-and-conquer manner, and noticed that quite a few algorithms either belong to, or are comprised of sub-algorithms which belong to, these classes. Cypher [4] has generalized these classes to a class (the "bit-block" class) that is less restrictive of the operations in a divide-and-conquer algorithm.

Chan [3] has pointed out that many problems in scientific and numerical computation have natural hierarchical solutions, and advocated the development of hierarchical parallel algorithms and architectures for this domain.

Not all problems will submit to solution on the private H-PRAM; there may be difficulties in designing algorithms for this variant such that data required by processors are always in the private shared memory of the sub-PRAMs that the processors belong to. Problems that *only* submit to non-oblivious (data dependent) communication *may* require a switch to the shared variant, which is one reason for its existence. We conjecture that H-PRAM algorithms will generally be "control oblivious", i.e. the partitioning will *not* be data dependent, but will instead depend on the cost parameters, number of processors P, and input size N.

We are continuing to investigate algorithms for the private variant of the H-PRAM.

References

[1] A. Aggarwal, A.K. Chandra and M. Snir, Communication complexity of PRAMs, *Theoretical Computer Science*, Vol. 71, 1990, pp. 3–28.

[2] A. Aggarwal, A.K. Chandra and M. Snir, On communication latency in PRAM computations, Tech. Rep. RC 14973 (#66882) 9/27/89, IBM T.J. Watson Research Center, Yorktown Heights, NY.

[3] T.F. Chan, Hierarchical algorithms and architectures for parallel scientific computing, *Proc. ACM Intl. Conference on Supercomputing*, 1990, pp. 318–329.

[4] R. Cypher, Efficient communication in massively parallel computers, Ph.D. Thesis, Dept. of Computer Science, Univ. of Washington, 1989.

[5] E. Dekel and S. Sahni, Binary trees and parallel scheduling algorithms, *IEEE Trans. on Computers*, March 1982, pp. pp. 307–315.

[6] P.B. Gibbons, A more practical PRAM model, *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 158–168.

[7] P.B. Gibbons, The asynchronous PRAM: a semi-synchronous model for shared memory MIMD machines, Ph.D. thesis, Computer Science Division, University of California, Berkeley, California, Dec. 1989.

[8] T. Heywood and S. Ranka, A practical hierarchical model of parallel computation I: The model, Technical Report SU-CIS-91-06 School of Computer and Information Science, Syracuse University, Feb. 1991, Revised: Oct. 1991.

[9] C.P. Kruskal, L. Rudolph and M. Snir, A complexity theory of efficient parallel algorithms, *Theoretical Computer Science*, Vol. 71, 1990, pp. 95–132.

[10] F.P. Preparata and J. Vuillemin, The Cube-Connected Cycles: a versatile network for parallel computation, *Commun. of the ACM*, May 1981, pp. 300–309.

[11] C.H. Papadimitriou and M. Yannakakis, Towards an architecture independent analysis of parallel algorithms, *SIAM Journal on Computing*, April 1990, pp.322–328.