A Model for Multi-Grained Parallelism †

John E. Savage
Brown University
Department of Computer Science, Box 1910
Providence, Rhode Island 02912
Phone: (401) 863-7642, Fax: (401) 863-7657
Email: jes@cs.brown.edu

Abstract

Multi-grained parallel computers can be very effective on computationally intensive problems that have important serial and parallel components. We examine the Mesh SuperHet, a model of this type consisting of the close coupling of a d-dimensional toroidal mesh of coarse-grained processors to a serial machine containing memory modules connected via a low-diameter network to a fast serial processor. We exhibit problems for which the Mesh SuperHet is superior to its serial or parallel components alone and develop tight performance bounds for sorting, the fast Fourier transform, and matrix multiplication. As multi-grained machines become more common, studies such as this will both reveal the fundamental limitations on such architectures and set the context for algorithm development.

1 Introduction

In this article we explore multi-grained parallel architectures consisting of closely coupled high-performance serial and parallel machines. We approach this topic believing that while many important problems exhibit high-degrees of parallelism, they also have important serial components which, if run on one processor of a parallel array, will run so slowly as to nullify the effect of parallelism.

Anecdotal evidence suggests that multi-grained architectures can greatly reduce computation time for large problems. It is reported that McRae solved a chemical process resource-allocation problem "40 times faster than he could on a supercomputer alone" [1] using a CRAY Y-MP connected via a high-speed link to a CM-2 Connection Machine. Others report a factor of 5 to 10 reduction in elapsed time [9] for such architectures. This evidence offers further support for multi-grained computers.

Our model for multi-grained parallelism is the *Mesh SuperHet* which we assert is a realistic model that provides a reasonable mix of serial and parallel computation together with a reasonable amount of structured communication between the two machines. It consists of closely coupled serial and parallel machines, the former having a supercomputer-style memory system and the latter being a p-processor, m-word/processor, d-dimensional toroidal mesh. (A 2-D version of the machine is shown in Figure 1.) We assume that the front-end memory system consists of a number of random-access memory modules of unlimited size (simulated via a memory hierarchy) that are connected via a low diameter network to the front-end processor. The front-end processor is assumed to execute instructions and access

†This work was supported in part by the Office of Naval Research under contract N00014-91-J-4052, ARPA Order 8225 and NSF Grant MiP-9020570. An abbreviated version of this paper was presented at the 6th Annual ACM Symposium on Parallel Algorithms and Architectures.

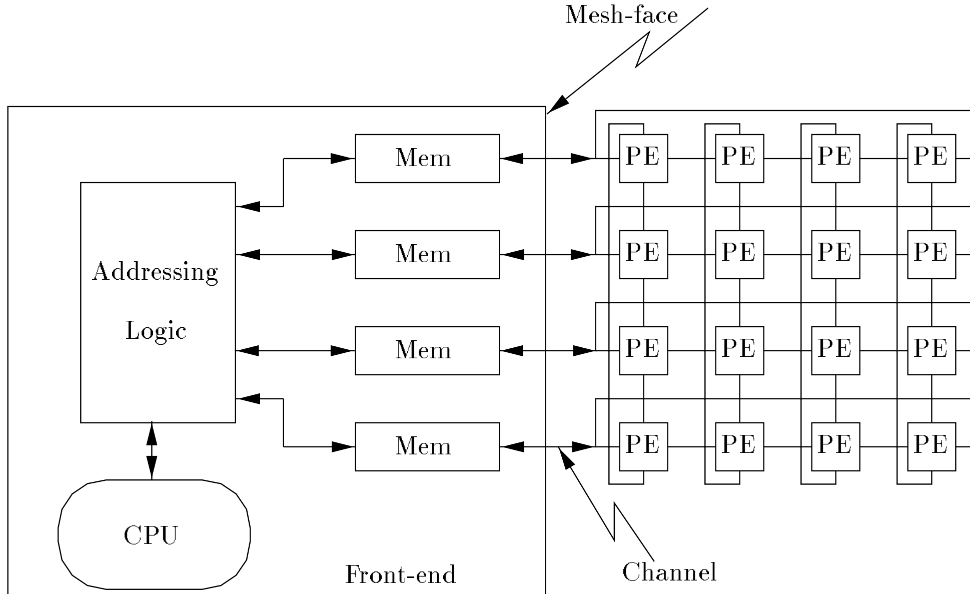

Figure 1: The two-dimensional Mesh SuperHet computer has p processors, m memory words per mesh processor, and is connected via $\sqrt{p}$ processors on the mesh face to a front-end that has $\sqrt{p}$ memory units of unlimited capacity that are addressable either by mesh processors or a serial front-end processor. The general Mesh SuperHet has a d-dimensional mesh with s processors along each dimension, $p = s^d$ processors all together, and $s^{d-1} = p^{1-1/d}$ processors (and front-end memory units) on a mesh face.

its memory modules ρ times as fast as a mesh processor, $\rho \geq 1$. The two machines are coupled by connecting each processor on a mesh face (each face has $p^{1-1/d}$ processors) with an individual memory module of the serial machine. We assume that the time to transmit n words between adjacent processors or between a mesh-face processor and a memory module requires $n + \lambda$ mesh processor cycles where λ is the mesh latency.

The assumed coupling between the serial and parallel machines offers a moderately generous bandwidth but requires that data generated on the mesh be organized carefully for front-end storage to avoid the creation of serial bottlenecks. We study close coupling between the serial and parallel machines in order to better understand its effect on the speedup attainable with the parallel machine. As we show, for some problems, such as sorting, the amount of coupling limits speedup whereas for others, such as the FFT and matrix multiplication, a full speedup is possible with limited coupling if the amount of memory per mesh processor, m, is sufficiently large.

This computational model allows us to study the effect of allocating some of the cost of multigrained parallel machine in a mesh and some in the traditional supercomputer front end. The assumptions made about the performance of the two types of machine and their coupling are realistic for modern high performance computers.

The effect of reducing the number of processors and memory modules that are connected

together and reducing the bandwidth of the channels between them directly affects the time to sort but only indirectly affects the time to compute the FFT and multiply matrices if processors have enough memory to compensate for the reduced bandwidth.

We show that the performance of algorithms is weakly dependent on the latency λ of the mesh and the ratio ρ between the speed of the front-end and mesh processors.

We focus attention on problems that are too large to store on the mesh and for which data must be moved between the front-end and the mesh before and after a computation. Mesh-based algorithms for the three problems examined here when they fit entirely on the mesh have been studied elsewhere (see Chapter 7 of [12]).

1.1 Contributions

We make four kinds of contributions. First, we exhibit a problem that is more quickly solved on the Mesh SuperHet than on either its serial or parallel component alone.

Second, we explore variations in the Mesh SuperHet architecture. We show that the model is robust under small changes in the number of interconnections between the mesh and the front-end memory modules. We also show that the impact of mesh communication latency λ on algorithm performance is small as long as λ is small by comparison with m, the number of words per mesh processor. Finally, we show that increasing the ratio ρ of the speed of the front-end and mesh processors has a small effect on the performance of algorithms.

Third, we develop a lower bound on computation time for sorting when the number of inputs, n, is large and show that a speedup of at most $O(p^{1-1/d}\log(mp))$ is possible on a p-processor, d-dimensional, m-word/processor Mesh SuperHet for this problem.

Fourth, we develop fast algorithms for three representative problems on this machine. We give an algorithm for sorting whose running time matches asymptotically that of the lower bound. We also give algorithms for the FFT and matrix multiplication that exhibit a full speedup when m is large relative to λ, p, and $1/\beta$, where β is the fraction of the mesh-face processors that are connected to front-end memory modules.

1.2 Related Computational Models

The Mesh SuperHet model is a generalization of the model of Atallah and Tsay [2] and that of Dehne, Fabri and Rau-Chaplin [5]. Atallah and Tsay connect a serial front end to a p-processor d-dimensional mesh with a small number of memory words per processor. They assume that at most $p^{1-1/d}$ (the number of processors on a mesh face) words can be transferred between the mesh and the serial machine per unit time without any restrictions on how the data is moved. They show that a speedup of $O(p^{1-1/d}\log p)$ is possible for sorting-related problems in computational geometry. Dehne, Fabri and Rau-Chaplin [5] consider a machine consisting of a d-dimensional, p-processor mesh in which the memory is large enough to hold the data for an entire problem. When a sorting problem fits entirely on a two-dimensional mesh, they show that a full speedup is possible if m satisfies $m = 2^{\Omega(\sqrt{p})}$.

Like Dehne et al., we assume that each processor on the mesh can have a large amount of memory. Like Atallah and Tsay, we assume the problem is so large that it cannot fit entirely on the processor array. Unlike Atallah and Tsay we severely restrict their model by forcing data to move through individual channels between machines.

Because this work concerns data movement between two machines it is related to that of Kung [8] who has explored conditions under which the computation and I/O time on serial machines are balanced for a number of problems. (Chapter 11 of [12] contains an accessible treatment of this subject.) He has examined problems such as matrix multiplication, Gaussian elimination, the FFT, and sorting; he also comments on these issues for two-dimensional meshes.

Finally, it should be noted that the Mesh SuperHet has much in common with the Cray Research, Inc. T3D machine. It has a supercomputer front-end and a three-dimensional mesh of fast processors with fast local communications. It differs from our model in that mesh processors are connected (in groups of 64) to the front-end via I/O channels, not directly to front-end memory modules.

2 Testing the Limits of the Mesh SuperHet

In this section we consider two questions: "Is the front-end processor always needed?" and "Are there problems solved more quickly on a Mesh SuperHet than on either the serial front end or the parallel computer alone?"

2.1 Simulating the Serial Front End

We note that although the large memory of the serial machine is used heavily in the algorithms developed in Sections 3, 4 and 5, the serial front-end processor is used only sparingly. This causes us to ask if this processor is always necessary. In practice it is necessary because, as explained earlier, many important computational problems either have an important serial component or make use of (legacy) code written for a serial processor. However, as the following theorem demonstrates, the work of the serial processor can done by the parallel processors as long as the serial processor is used infrequently.

Theorem 1 *Let T_{serial} be the number of steps executed by the serial front end of a d-dimensional, p-processor Mesh SuperHet in a T-step computation. If $T_{\text{serial}} = O\left(T/p^{1/d}\right)$, the computation can be done without the front-end processor in $O(T)$ steps.*

> **Proof** The serial processor can access any front-end memory module in one unit of time. To simulate this processor one mesh processor is designated as the replacement for the serial processor. During a serial phase of the computation this processor accesses a front-end memory module by dispatching one request to the corresponding mesh-face processor. Since the distance on the mesh face between any two processors is $O\left(p^{1/d}\right)$, $O(p^{1/d}T_{\text{serial}})$ steps suffice to simulate the serial front end and the result follows. ■

2.2 A Problem Better Solved on the Mesh SuperHet

While intuition says that a high-performance serial front-end processor can have a material influence on the execution time for problems having a mix of serial and parallel structure, it is difficult to exhibit a natural problem of this kind. For this reason we are content to exhibit a problem with this structure that is artificially created to demonstrate that problems exist.

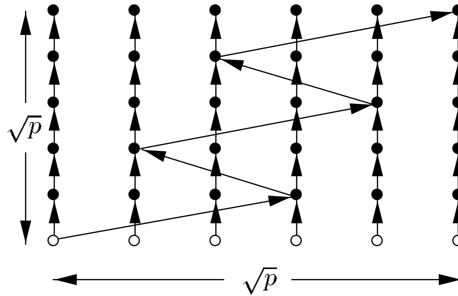

Figure 2: A graph describing a computation that is computed more efficiently on the Mesh SuperHet than on either the serial front-end or the mesh alone. One vertex in each row is special. It receives an input from a special vertex in the preceding row. Many columns separate special vertices in adjacent rows. The open vertices are associated with variables. Each closed vertex has one more input from a variable that is not shown.

Figure 2 exhibits a graph that we show is better solved on the 2-D p-processor Mesh SuperHet than on either its serial component (the mesh is not used) or parallel component (front-end memory modules are connected to mesh-face processors but the front-end processor is disabled). The graph has $\sqrt{p}$ columns each containing $\sqrt{p}$ vertices. As shown below, if the vertices in a given column are stored initially in a single front-end memory, $\sqrt{p}$ mesh-face processors can work almost independently on the columns except when values of special vertices (one per row, connected via edges to remote vertices) are computed. If the serial front-end processor can retrieve these remote values, the columns can be traversed in $\sqrt{p}$ steps. If not, the remote values have to be obtained by passing them through chains of processors on the mesh face which increases the total computation time to an amount proportional to p, as we show below.

To complete the description of the graph, we add a bit more detail. Each open graph vertex is associated with a unique variable. Each one-input solid vertex actually has two inputs, the output of its predecessor in the graph and the value of a unique variable not shown. Finally, each (special) binary vertex has three inputs, outputs of the two graph vertices to which it is connected plus a unique variable, also not shown. This graph has p input variables and $\sqrt{p}$ output vertices.

We assume that input variables associated with the jth column are stored in the jth memory module (See Figure 1). The graph can be computed level by level in $\Theta(\sqrt{p})$ steps using the $\sqrt{p}$ mesh-face processors to compute column values and the serial processor to fetch remote values. However, if the rapid access to the front-end memory modules provided by the serial processor is not available, it must be simulated by mesh processors, introducing a factor of approximately $\sqrt{p}$ (the number of columns separating special vertices in adjacent rows) in the computation time for each parallel step, making the computation time proportional to p.

This argument fails if the columns can be permuted to place logically adjacent columns in adjacent memory modules. A related problem with the desirable property has the same structure except that a parameter c_i, unknown when columns are assigned to memory modules, specifies the column containing the special vertex in the ith row, $1 \leq i \leq \sqrt{p}$. These

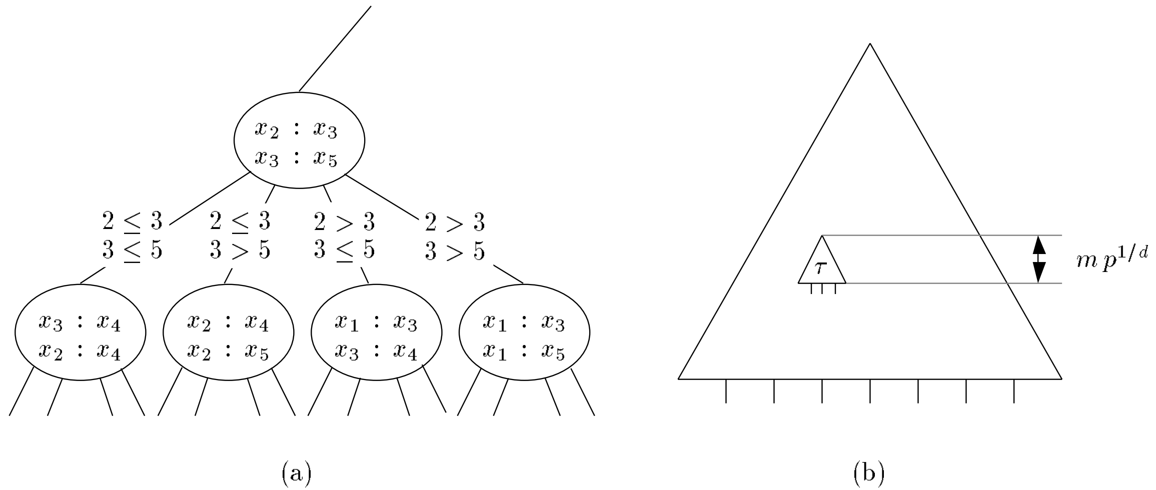

Figure 3: (a) Several levels of a parallel decision tree; (b) $mp^{1/d}$ levels of a large parallel decision tree.

parameters can be stored in one front-end memory module or retrieved from separate memory modules by the front-end in time $O(\sqrt{p})$. Thus, this computation can be done in time $O(\sqrt{p})$ on the Mesh SuperHet but only in time $\Omega(p)$ on either submachine alone, whether column variables are stored in the same module or not.

3 Sorting on the Mesh SuperHet

We derive a lower on the time to sort with comparison-based sorting algorithms on the Mesh SuperHet and then describe an asymptotically optimum sorting algorithm. We assume without loss of generality that all words to be sorted are distinct.

3.1 A Lower Bound on the Time to Sort

Our lower bound on the time to sort uses the parallel decision tree model of computation [7, pp. 185], an extension of the well-known serial decision tree model. As suggested in Figure 3(a), a parallel decision tree performs multiple comparisons in each step and then executes one of several branches based on the outcomes.

For every comparison-based sorting algorithm there is a parallel decision tree. Each leaf of such a treee corresponds to the order that the inputs $\{x_1, x_2, \ldots, x_n\}$ have when sorted. Since there are $n!$ permutations of these words, every parallel decision tree has at least $n!$ leaves. We note that the paths through a parallel decision tree may not all have the same length and that several different paths may be associated with the same set of comparisons, although performed in different orders.

While the following lower bound on sorting time is stated for the Mesh SuperHet, it applies to a more general class of machine, namely, those whose front-end can do $O(p^{1-1/d}$

$\log(mp)$) comparisons per parallel machine step, whose parallel machine has mp storage locations, and whose mesh-face can pass $\beta p^{1-1/d}$ words per parallel step, $0 \leq \beta \leq 1$.

Theorem 2 *Let $T_{sort}(n,m,p,d)$ be the time, measured in mesh processor steps, required by a comparison-based algorithm to sort n elements, $n \geq 4$, on a p-processor, m-word/processor, d-dimensional Mesh SuperHet in which $\beta p^{1-1/d}$ mesh-face processors are connected to front-end memory modules, $0 \leq \beta \leq 1$. Then, $T_{sort}(n,m,p,d)$ satisfies the following inequality where ρ is the ratio of the speed of the front-end processor to that of a mesh processor.*

$$T_{sort}(n,m,p,d) \geq \frac{n \log n}{4(\rho + (1+\beta)p^{1-1/d}\log(2mp))}$$

Proof Divide time up into intervals of $mp^{1/d}$ mesh processor steps. In any given interval let e be the number of words entering and l the number leaving the mesh. Clearly, $e + l \leq \beta mp$ because at most $\beta p^{1-1/d}$ words can move across the mesh face in one step. Since at the beginning of the interval there are at most mp words in the array, at most $e + l + mp \leq (1+\beta)mp$ words can interact on the mesh during any interval. Thus, at most $((1+\beta)mp)!$ permutations of these elements can be identified by the mesh during an interval. The serial processor can make at most ρ comparisons on each mesh step and at most $\rho mp^{1/d}$ comparisons in one interval. Since there are two outcomes for each of these comparisons, at most $2^{\rho mp^{1/d}}$ outcomes can be identified by the front-end processor in one interval. Thus, at most $((1+\beta)mp)!2^{\rho mp^{1/d}}$ outcomes can be simultaneously identified by both the front end and the mesh in one interval.

Let τ be a subtree of the parallel decision tree corresponding to the comparisons made by mesh processors and the serial front-end process in one interval, as suggested in Figure 3(b). It follows that the number of distinguishable leaves of any such subtree τ is at most $((1+\beta)mp)!2^{\rho mp^{1/d}}$.

Consider the full decision tree associated with a comparison-based sorting algorithm whose running time is $T_{sort}(n,m,p,d)$. This time contains $T_{sort}(n,m,p,d)/mp^{1/d}$ intervals of length $mp^{1/d}$. Because at most $((1+\beta)mp)!2^{\rho mp^{1/d}}$ outcomes can be identified in one interval it follows that at most

$$(((1+\beta)mp)!2^{\rho mp^{1/d}})^{T_{sort}(n,m,p,d)/mp^{1/d}}$$

outcomes can be identified in time $T_{sort}(n,m,p,d)$. Because the decision tree must have at least $n!$ leaves, $T_{sort}(n,m,p,d)$ must satisfy the following lower bound.

$$T_{sort}(n,m,p,d) \geq mp^{1/d} \log n! / \log\left((1+\beta)mp)!2^{\rho mp^{1/d}}\right)$$

The desired result follows from the observations that $(q/4)\log_2 q \leq \log(q!)$ for $q \geq 4$ and $\log_2(q!) \leq q \log_2 q$. ■

This lower bound shows that a speedup of at most about $p^{1-1/d}\log(mp)$ is possible. A full speedup of p is not achievable because it takes time proportional to $mp^{1/d}$ to move mp

words on and off the mesh, which is comparable to the time to merge mp words on the mesh. Note the weak dependence of the bound on ρ and β. ρ, the ratio of the speed of the front-end processor to that of a mesh processor, must be at least as large as $2p^{1-1/d}\log(mp)$ when $\beta = 1$ for the speed of the front-end to affect the speedup of the combined machine. (For $p = 64$, $d = 2$ and $m = 1$ Mbyte, ρ must be at least 368.)

It is interesting to note that, as shown by Dehne *et al.* [5], that the n words can be sorted with a full speedup on a 2-D mesh without a front-end unit if the following conditions hold: a) the n words fit into the memory of the p processors, that is, $n = mp$, and b) the amount of memory per processor is large enough relative to the number of processors, that is, $m = 2^{\Omega(\sqrt{p})}$.

We are particularly interested in the case for which n is much larger than mp, that is, the mesh unit cannot solve the problem alone but is used to speed up an otherwise serial problem.

3.2 An Asymptotically Optimal Sorting Algorithm

In this section we generalize the Atallah-Tsay sorting algorithm [2] (see Figure 4) to the Mesh SuperHet. Our principal challenge is to manage data movement between the machines so as to avoid creating serial bottlenecks.

The new algorithm sorts a list of n elements by dividing the list into q sublists, recursively sorting the sublists, and then merging these lists together. Without loss of generality we assume that $n = q^t(mp/3)$, t an integer, and $q = \left\lceil (mp/3)^{1/2d} \right\rceil$. The algorithm merges the q sublists using the front-end and mesh cooperatively. It finds the approximately $(mp/3)$ smallest words in the q lists and moves them to the mesh where they are merged. (The complex details of this process are central to our extension of the Atallah-Tsay algorithm.) This step is repeated with the next approximately $(mp/3)$ smallest elements until all lists are exhausted. Identification, removal and merging of these words is done through coordinated action by the serial front end and the mesh. Our extension to the Atallah-Tsay algorithm involves a careful implementation of the procedure `Merge` (see Figure 4).

Since the front-end processor executes ρ steps in the time it takes for a mesh processor to execute one step, we measure the time for a computation in multiples of the mesh processor execution time.

Theorem 3 *When $mp \geq 3\,2^d$, a set of n elements can be sorted on the p-processor, d-dimensional Mesh SuperHet with m words per mesh processor in time $T_{sort}(n, m, p, d)$ where*

$$T_{sort}(n, m, p, d) = O\left(\frac{n \log n}{p^{1-1/d}\log(mp/3)}\left(1 + \frac{\lambda}{m}\right) + \frac{n}{p}\log m\right)$$

The effect of latency λ is small. In particular it can be ignored if m is large by comparison with λ, an assumption we make. In this case the second term in the above bound dominates when

$$\frac{n \log n}{p^{1-1/d}\log(mp/3)} = O\left(\frac{n}{p}\log m\right)$$

or when $\log n = O\left(\log(mp/3)\log(m)/p^{1/d}\right)$. When n is a polynomial in mp (the uninteresting case), namely, $n = (mp)^k$ for k a constant, a full speedup is possible when m is large

```
Sort(List u of n words)
  Divide u into sublists {v_j, 1 ≤ j ≤ q};
  Recursively sort these into lists {w_j, 1 ≤ j ≤ q};
  Merge sorted sublists {w_j, 1 ≤ j ≤ q};

Merge {w_j, 1 ≤ j ≤ q}
  Until done
    Find (mp/3)th smallest word;
    Extract mp/3 smallest words from q lists;
    Move mp/3 smallest words to mesh;
    Shorten lists {w_j, 1 ≤ j ≤ q} ; (Update pointers)
    Sort words on mesh;
    Move sorted words to front end;
```

Figure 4: A sketch of the Atallah-Tsay sorting algorithm.

enough relative to p, namely, when $\log m = \Omega(k\,p^{1/d})$. When n grows more rapidly with mp (the interesting case) or when m grows more slowly with p (unlikely), the speedup is at most $O(p^{1-1/d}\log(mp/3))$.

We now give a proof of this result. Consider the planes of the mesh that are parallel to the mesh-face, the plane connected to the front-end memory units. We assign a common linear order to the $p^{1-1/d}$ processors on each of these parallel mesh planes and to the front-end memory units. (We speak of the ith such processor on the mesh face and the ith corresponding memory unit.) We also order the planes from front to back thereby giving every processor a unique index.

The general sorting procedure divides the list of n words into q sublists each containing n/q words. It sorts these sublists and then merges them. We show that the merging step can be done in $O\left(n(1+\lambda)/p^{1-1/d}\right)$ steps using the front-end and the mesh. This gives the following recurrence for $T_{sort}(n,m,p,d)$, the time used by our algorithm.

$$T_{sort}(n,m,p,d) = qT_{sort}\left(\frac{n}{q},m,p,d\right) + O\left(\frac{n}{p^{1-1/d}}\left(1+\frac{\lambda}{m}\right)\right)$$

The base case for our recursive procedure is the sorting of lists of $n/q^t = mp/3$ words. Below we give an algorithm to sort a list of $mp/3$ words that takes time $T_{sort}(mp/3,m,p,d)$ where

$$T_{sort}(mp/3,m,p,d) = O\left(\frac{mp}{p^{1-1/d}}\left(1+\frac{\lambda}{m}\right)+m\log m\right)$$

which is the base case for the induction. The desired conclusion follows from these two results.

Lemma 1 *There exists a Mesh SuperHet algorithm with running time $T_{sort}(wp,m,p,d)$ satisfying the bound given below that sorts a list of wp words, $m/3 \le w \le m$. The wp words are stored initially as lists of $wp^{1/d}$ words in contiguous locations in the $p^{1-1/d}$ front-end*

memory modules. After being sorted, they are stored in the same modules with the same number of words per module.

$$T_{sort}(w, m, p, d) = O\left((w+\lambda)p^{1/d} + w\log w\right)$$

Proof The wp words are moved across the mesh face and onto the memories of the mesh processors in such a fashion that each mesh memory has w words. After a delay of λ processor cycles data moves across the mesh face. Since there are $wp^{1/d}$ words per front-end memory module, through pipelining the wp words can be moved onto the mesh in time $O\left((w+\lambda)p^{1/d}\right)$. Each mesh processor then sorts its w words sequentially in $O(w\log w)$ steps after which these sorted lists are merged on the mesh using an extension of Batcher's bitonic sorting algorithm [3], as discussed below.

Batcher's bitonic sorting algorithm sorts p words in $O(p^{1/d})$ steps on a p-processor, d-dimensional mesh by executing a set of data-independent compare/exchange operations (two keys are compared and exchanged if the first is larger than the second). Preparata and Vuillemin [10] show how to map Batcher's algorithm, a normal algorithm, from the hypercube to meshes. (Savage [12, page 307] provides an accessible treatment of these results.)

To create a merging algorithm from Batcher's algorithm, each compare/exchange operation is replaced by a merge/split operation. A merge/split operation is supplied two w-word sorted lists and produces two w-word sorted lists in which all elements of the second list are greater than or equal to all words of the first. Each merge/split operation is implemented on a serial mesh processor as a merge operation in $O(w)$ steps.

Ignoring the time for data movement, this merging algorithm takes time $O\left(wp^{1/d}\right)$. Since the time to move one w-word block between adjacent processors is $w+\lambda$ and Batcher's algorithm requires $O\left(p^{1/d}\right)$ (parallel) such data movement steps, the total time to move data and merge it is $O\left((w+\lambda)p^{1/d} + w\log w\right)$.

Finally, since Batcher's algorithm doesn't necessarily leave the sorted sequence in the common order of mesh processors, another application of Batcher's algorithm may be necessary to move these lists to their desired final positions. This can be done by associating unique keys to each sorted list of w words.

It follows that the total time to sort wp words and leave them in final positions on the front end is $O\left((w+\lambda)p^{1/d} + w\log w\right)$. ■

As shown below, our new sorting algorithm invokes the `Merge` procedure of Figure 4 $n/(mp/3)$ times, extracting $mp/3$ words from the q lists each time. It does this by moving between $mp/3$ and mp words from front-end memory units to the mesh each time, processing them (discarding words if necessary), and moving $mp/3$ sorted words back to front-end memory units.

The `Merge` procedure is used to merge q sorted lists. These lists are organized into **blocks** of $(mp/3)^{1-1/d}$ words arranged in ascending order. Blocks are placed on disks in such a fashion that the first $z = (m/3)^{1-1/d}$ words (z an integer) are placed in ascending order in consecutive locations of the first front-end memory unit, the next z are placed in

ascending order in the same locations of the second front-end memory unit, etc. Thus, a block can be moved between the front end and the mesh in z steps.

To extract words from the q sorted lists, `Merge` invokes the procedure **Find** based on the algorithm of Frederickson and Johnson [6]. The Frederickson/Johnson algorithm, which is too complicated to describe in this space, finds the kth smallest word in q sorted lists. It has a serial running time that is independent of the length of these lists, unlike that of Blum et. al. [4] which finds the kth smallest word in an unsorted list of n words in time $O(n)$ on a serial processor. The following theorem describes the performance of the Frederickson/Johnson algorithm.

Theorem 4 ([6]) *Let $t = \min(k, q)$. An algorithm exists that finds the k smallest item in an $b \times q$ matrix of sorted columns in at most $\gamma(q + t\log(k/t))$ steps for some constant $\gamma > 0$.*

We apply this algorithm to our problem with $k = (mp/3)$ and $q = \left\lceil (mp/3)^{1/2d} \right\rceil$ from which it follows that $t = q$ and its running time is $(\gamma/\rho)(q + q\log(k/q)) \leq 2(\gamma/\rho)q\log(mp/3)$ which is $O\left((mp/3)^{1/2d}/\rho)\log(mp/3)\right)$ in the time for one mesh step.

We now describe our modification of the `Extract` procedure of Figure 4. It has one form when the first $mp/3$ items are extracted and a second one when the remaining items are removed.

Let ∞ be a value larger than all other values and let μ be the $(mp/3)$th smallest word in the current set of q lists stored in the front-end memory units.

Our first version of `Extract` copies to the mesh all blocks that contain μ or words smaller than μ. Because at most $q = \left\lceil (mp/3)^{1/2d} \right\rceil$ blocks may contain larger words and each block has $(mp/3)^{1-1/d}$ words, at most $(mp/3)^{1-1/d}q \leq mp/3$ words larger than μ are moved to the mesh. Combined with the $mp/3$ words less than or equal to μ, at most $2mp/3$ words are moved to the mesh. These words are placed in the memory of the mesh processors with at most m words per memory. Through pipelining this data movement occurs in time $O\left((m + \lambda)p^{1/d}\right)$. In $O\left(p^{1/d}(1 + \lambda)\right)$ time the value of μ is then broadcast to all mesh processors. In time $O(\log m)$ each processor identifies each of its words larger than μ. Blocks on the mesh whose mirror images on the front-end contain a word larger than μ (these blocks will be moved to the mesh in the next round) are copied back to their original positions in the front-end memory units with values less than or equal to μ replaced by ∞ in time $O\left((m + \lambda)p^{1/d}\right)$. The front-end is also given the identity of these blocks. (This corresponds to shortening the lists in Figure 4.)

On the mesh all words greater than μ are replaced by ∞, leaving the words on each processor in sorted order. The words are then sorted using Batcher's algorithm in time $O\left((m + \lambda)p^{1/d}\right)$. The $mp/3$ words other than ∞ are written back to the front-end memory units into positions reserved for the merged sorted list in time $O\left((m + \lambda)p^{1/d}\right)$.

The second and later applications of `Extract` use μ, the $mp/3$th smallest word among those that have not been been removed. A block containing such words may also contain words previously replaced by ∞ (words that were less than or equal to a previous value of μ) or words larger than the current value of μ. (See Figure 5.) There are at most $2q$ such blocks. Since each block has $(mp/3)^{1-1/d}$ words, there are $2q\,(mp/3)^{1-1/d}$ such words which is at most $2mp/3$ (equivalently $\lceil a \rceil \leq a^2$ or $a \geq \sqrt{2}$ for $a = (mp/3)^{1/2d}$) when $mp/3 \geq 2^d$.

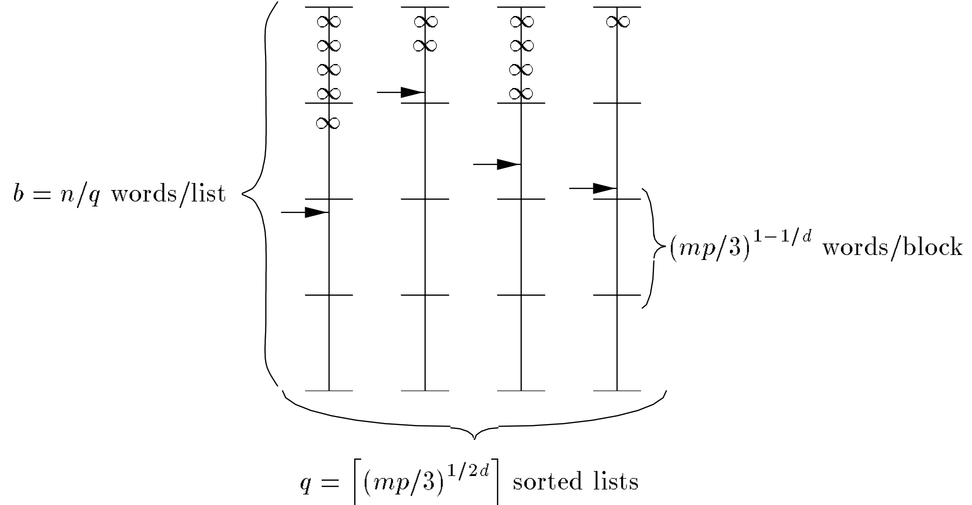

Figure 5: Sorted columns on which **Extract** operates. Columns are subdivided into sorted blocks containing $(mp/3)^{1-1/d}$ words. Each block is uniformly distributed over the front-end memory units, $(m/3)^{1-1/d}$ words per unit. Pointers identify the next word larger than the next $(mp/3)$ elements in each column. Blocks containing words less than or equal to the $mp/3$th smallest word not yet removed, μ, may contain words that are larger than μ and words marked as ∞ that were removed on previous steps.

Adding to these words the $mp/3$ words less than or equal to μ and greater than the previous value of μ, at most mp words are moved to the mesh. We then write back to the front-end memory those blocks containing words larger than μ after replacing words less than or equal to μ by ∞. On the mesh we then replace words larger than μ with ∞, sort all words on the mesh, move the $mp/3$ sorted words other than ∞ to their final positions on front-end memory units, and shorten the lists. As discussed in the proof of Lemma 1, the time for these executions of the `Extract`, `Move`, `Shorten`, and `Sort` procedures is $O((m+\lambda)p^{1/d})$.

It follows that the time for each pass of the `Merge` procedure satisfies the following bound when $mp \geq 3\,2^d$.

$$T_{merge} = O\left(\left(\frac{1}{\rho}\right)(mp/3)^{1/2d}\log(mp/3) + (m+\lambda)p^{1/d}\right)$$

Thus, $T_{merge} = O\left((m+\lambda)p^{1/d}\right)$ when the second term dominates, that is, when $\log a \leq ab$ for $a = (mp/3)^{1/2d}$ and $b = (\rho/2d)(1+\lambda/m)m^{1-1/d}3^{1/d}$. Since $\log a \leq a$ when $a \geq 1$, the inequality $\log a \leq ab$ holds when $b \geq 1$, that is, when $\rho(1+\lambda/m)m^{1-1/d}3^{1/d} \geq 2d$ which is easily satisfied for typical values of d, such as 2 or 3. For example, it is satisfied if $\rho \geq 2d$ (the front end is $2d$ times faster than a mesh processor) or $\rho = 1$ and $m \geq 3d$ (the front end is the same speed as a mesh processor but each processor has at least $3d$ memory locations). Thus, without loss of generality, $T_{merge} = O\left((m+\lambda)p^{1/d}\right)$ holds when $mp \geq 3\,2^d$.

Since `Merge` is invoked $3n/mp$ times, under these conditions the total time to merge the q sorted lists is $O\left(n(1+\lambda/m)/p^{1-1/d}\right)$ and Theorem 3 follows.

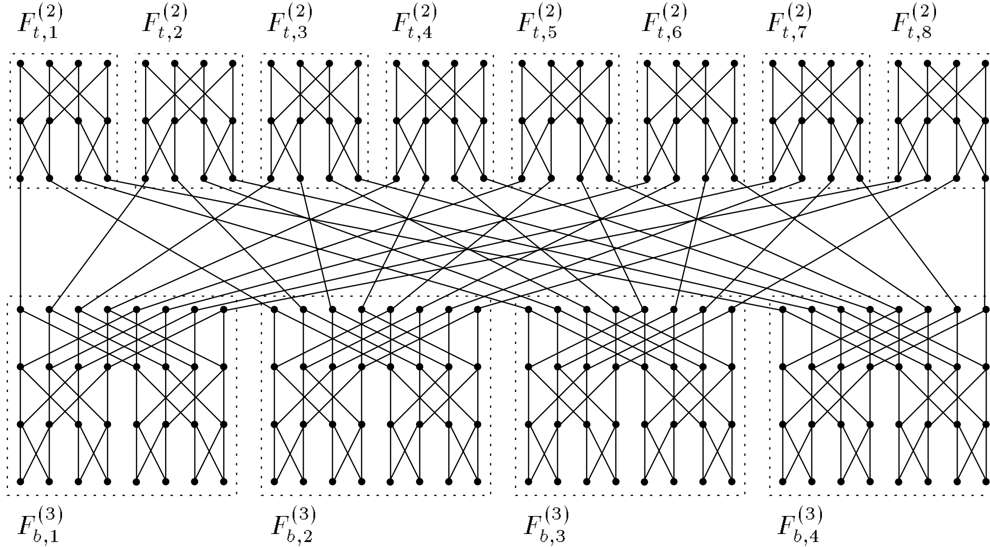

Figure 6: Decomposition of the FFT graph $F^{(5)}$ on 32 inputs into 4 copies of $F^{(3)}$, the FFT graph on 8 inputs, and 8 copies of $F^{(2)}$ on 4 inputs. The edges between bottom and top FFT subgraphs in $F^{(5)}$ are not in the graph but are added to show vertices common to two FFT subgraphs.

4 The Fast Fourier Transform Algorithm

The fast Fourier transform (FFT) algorithm computes the discrete Fourier transform (DFT) on n inputs in time $O(n \log n)$ on a serial computer when $n = 2^k$ for integer k. We now develop a Mesh SuperHet algorithm for it, assuming m and p are powers of 2. We show that when each mesh processor has enough memory, a full speedup is possible. This surprising result, which differs from that for sorting, follows from the fact that a full speedup is possible in the computation of an FFT on mp inputs (the number that will fit on the mesh) when m is sufficiently large.

Our algorithm uses the well-known decomposition of the FFT into FFT subgraphs shown in Figure 6 [13]. The FFT graph on 2^a inputs, $F^{(a)}$, can be decomposed into 2^r top subgraphs $\{F_{t,i}^{(a-r)} \mid 1 \leq i \leq 2^r\}$ on 2^{a-r} inputs and 2^{a-r} bottom subgraphs $\{F_{b,j}^{(r)} \mid 1 \leq j \leq 2^{a-r}\}$ on 2^r inputs.

Let $mp = 2^r$. For $n = 2^k$ our algorithm computes $F^{(k)}$, the FFT graph on n inputs, by decomposing it into $m = 2^r$ top subgraphs $\{F_{t,i}^{(k-r)} \mid 1 \leq i \leq 2^r\}$ on 2^{k-r} inputs and $n/mp = 2^{k-r}$ bottom subgraphs $\{F_{b,j}^{(r)} \mid 1 \leq j \leq 2^{k-r}\}$ on 2^r inputs. The bottom subgraphs $\{F_{b,j}^{(r)} \mid 1 \leq j \leq 2^{k-r}\}$ are computed individually by moving mp inputs to the mesh, performing the FFT computation, and moving the results to the front-end memory units. Below we describe the computation of $F_{b,j}^{(r)}$. We then compute the top subgraphs $\{F_{t,i}^{(k-r)} \mid 1 \leq i \leq 2^r\}$ recursively by decomposing them into top subgraphs $\{F_{t,i}^{(k-2r)} \mid 1 \leq i \leq 2^r\}$ and bottom subgraphs $\{F_{b,j}^{(r)} \mid 1 \leq j \leq 2^{k-r}\}$. This process is repeated until $F^{(k)}$

is decomposed into $\lceil(\log n)/(\log mp)\rceil$ layers with n/mp FFT subgraphs on r inputs in each layer except possibly the top layer. Without serious loss of generality we assume that $\log n$ is divisible by $\log mp$.

Let $T_{FFT}(n, m, p, d)$ be the time to compute $F^{(k)}$. Below we show that $T_{FFT}(mp, m, p, d)$, the base case for $T_{FFT}(n, m, p, d)$, satisfies $T_{FFT}(mp, m, p, d) = O\left(m \log(mp) + (m + \lambda)p^{1/d}\right)$. Furthermore, the time to move mp data items on and off the mesh and permute the outputs of an FFT subgraph so that they are accessible in parallel by subsequent FFT graphs is $O\left((m + \lambda)p^{1/d}\right)$. Thus, mp inputs can be moved to the mesh, the FFT computed on them, and the outputs moved back to front-end memory units in time $O\left(m \log(mp) + (m + \lambda)p^{1/d}\right)$. The first term in this expression dominates and a full speedup of $O(p)$ for this computation is possible when $\log(mp) \geq (1 + \lambda/m)p^{1/d}$.

From the decomposition given above it follows that $T_{FFT}(n, m, p, d)$ satisfies the following recursion.

$$\begin{aligned} T_{FFT}(n, m, p, d) &= (mp)T_{FFT}(n/mp, m, p, d) \\ &\quad + (n/mp)\, O\left(\frac{mp\log(mp)}{p}\left(1 + \frac{(1 + \lambda/m)p^{1/d}}{\log mp}\right)\right) \end{aligned}$$

The solution to this recurrence is stated in the following theorem.

Theorem 5 *The FFT algorithm on n inputs can be computed on the p-processor, d-dimensional Mesh SuperHet with m words per mesh processor and interprocessor latency of λ in time $T_{FFT}(n, m, p, d)$ where*

$$T_{FFT}(n, m, p, d) = O\left(\frac{n \log n}{p}\left(1 + \frac{(1 + \lambda/m)p^{1/d}}{\log mp}\right)\right)$$

A full speedup is possible when $\log(mp) \geq (1 + \lambda/m)p^{1/d}$.

The fact that a full speedup holds can be seen intuitively as follows: the FFT on n inputs can be decomposed into $(\log n)/(\log mp)$ layers with n/mp FFT graphs on mp inputs in each layer. If each FFT graph in one layer can be computed in time $O\left(mp \log(mp)/p\right)$, then the FFT graph on n inputs can be computed in time $O\left((n/mp)(\log n/\log mp)(mp\log(mp)/p)\right)$ which is $O(n \log n/p)$.

Computation of Bottom FFT Graphs We now describe our algorithm to compute a bottom subgraph $F^{(r)}$ on $mp = 2^r$ inputs when $m = 2^d$, $p = 2^c$, and $r = c+d$. Our algorithm decomposes $F^{(r)}$ into $m = 2^d$ top subgraphs on $p = 2^c$ inputs, $\{F_{t,j}^{(c)} \mid 1 \leq j \leq 2^d\}$, and p bottom subgraphs on m inputs, $\{F_{b,j}^{(d)} \mid 1 \leq j \leq 2^c\}$. It first moves mp inputs from the front end to the mesh in time $O\left((m + \lambda)p^{1/d}\right)$ placing m inputs in each of the p processor memories. The subgraphs $F^{(d)}$ are computed in parallel on individual processors in time $O(m \log m)$. The m top subgraphs $F^{(c)}$ on p inputs (whose outputs are the outputs of $F^{(r)}$) are computed m/p per processor. The inputs to the top subgraphs are the outputs of the p bottom subgraphs. It follows that to compute the m/p FFT subgraphs produced at each processor (each on p inputs) requires moving m/p outputs from each of the p processors to a single processor. This data movement corresponds to a matrix transposition. We now describe how this is done.

Data Transposition on the Mesh As mentioned in Section 3.2, a common linear order is assigned to the front-end memory units and the $p^{1-1/d}$ processors in each plane of the mesh. Let these processors be indexed from 0 to $p^{1-1/d}-1$. Since the planes are ordered from the first to the $p^{1/d}$th, a linear order is given to each of the p processors. Let the processors be indexed from 0 to $p-1$ and let the locations in each memory unit be indexed from 0.

The first m/p of the top FFT subgraphs are assigned to the first processor, the second m/p top subgraphs are assigned to the second processor, etc. Because the jth input to the ith top FFT subgraph is the ith output of the jth bottom FFT subgraph, we must move outputs of the bottom FFT subgraphs to those processors at which the top FFT subgraphs are computed. This is done by moving to each of the processors a group of m/p of the outputs produced by each bottom subgraph (every pth output, as suggested in Figure 6). This step, which amounts to a matrix transposition, can be done in parallel in $O((m+\lambda)p^{1/d})$ steps using a routing algorithm, as explained in the proof of Lemma 1. The top subgraphs are then computed in $O((m/p)(p\log p)) = O(m\log p)$ steps because each processor computes m/p FFT subgraphs on p inputs serially. It follows that $F^{(r)}$ on $2^r = mp$ inputs can be computed on the mesh in time $O(m\log(mp) + (m+\lambda)p^{1/d})$.

Data Transposition on the Mesh SuperHet As described above, $F^{(k)}$ on $n = 2^k$ inputs is computed from its decomposition into $mp = 2^r$ top subgraphs $\{F_{t,i}^{(k-r)} \mid 1 \leq i \leq 2^r\}$ on $n/mp = 2^{k-r}$ inputs and $n/mp = 2^{k-r}$ bottom subgraphs $\{F_{b,j}^{(r)} \mid 1 \leq j \leq 2^{k-r}\}$ on $mp = 2^r$ inputs. In turn, each top subgraph $F_{t,i}^{(k-r)}$ is decomposed into top FFT subgraphs on $k-2r$ inputs and bottom FFT subgraphs on r inputs. This process is repeated until $F^{(k)}$ is decomposed into $\log n/\log(mp)$ layers each containing $2^{k-r} = n/mp$ FFT subgraphs on r inputs.

This is a straightforward recursive construction except for the fact that if the outputs of the bottom FFT subgraphs are stored in the same order in the front-end memory units, then many top FFT subgraphs will need to retrieve each of their inputs from the same memory unit, as can be seen by consulting Figure 6. This would serialize the computation. To permit parallel access to the inputs of FFT subgraphs on $mp = 2^r$ inputs in one layer, we cyclically rotate the outputs of successive such FFT subgraphs in the previous layer by sufficiently many places such that if an output from one FFT subgraph is sent to the sth front-end memory unit, the same output in the next FFT subgraph is sent to the $(s+1)$st (with wraparound) memory unit. Because the jth input to the ith top FFT subgraph is the ith output of the jth bottom FFT subgraph, this insures that the inputs to an FFT subgraph on 2^r inputs into which the top FFT subgraph is divided receives its inputs in parallel. This provides the desired result.

Reducing the Size of the Mesh Face If the number of channels crossing the mesh face is reduced from $p^{1-1/d}$ to $\beta p^{1-1/d}$, $0 \leq \beta \leq 1$, the effect on the above algorithm is to increase the time to move data onto and off of the mesh from $O\left((m+\lambda)p^{1/d}\right)$ to $O\left((m+\lambda)p^{1/d}/\beta\right)$. For sufficiently large m a full speedup continues to be possible.

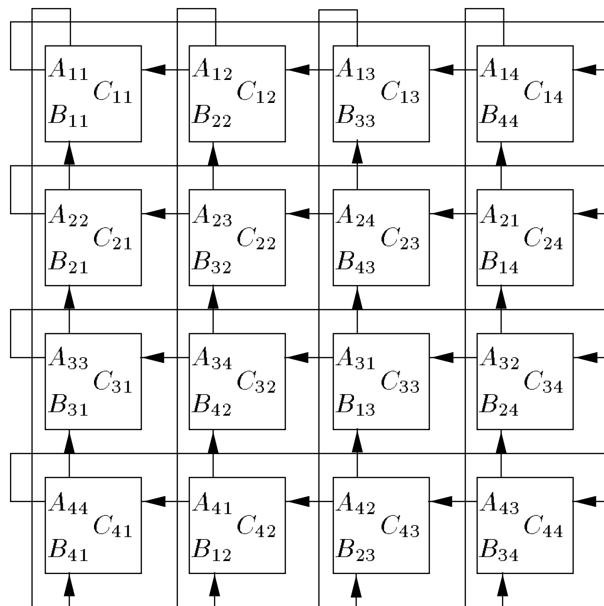

Figure 7: Placement of matrix elements in a 2-D mesh before starting the computation of the product $C = A \times B$. Elements in C remain stationary while those in A and B move horizontally and vertically, respectively.

5 Matrix Multiplication

In this section we describe an algorithm for the multiplication of two $n \times n$ matrices on the p-processor, d-dimensional Mesh SuperHet when at most $\beta p^{1-1/d}$ channels are used, $0 < \beta \leq 1$. This algorithm provides a full speedup when m is sufficiently large relative to p. Kung [8] has made similar observations for serial, I/O-limited machines.

Our algorithm extends to d dimensions the 2-D systolic algorithm to multiply two $s \times s$ matrices described by Preparata and Vuillemin [11]. Their 2-D algorithm (which ignores latency) places the entries in the $s \times s$ matrices $A = \{A_{i,j}\}$ and $B = \{B_{i,j}\}$ in an $s \times s$ 2-D mesh, one element per processor, as shown in Figure 7. (The first row of A is placed in the first row of the mesh. The ith row of A is rotated left $i-1$ places before placing it in the ith row. A similar translation of columns of B generates the placement shown.) The (i,j) entry in the product matrix $C = AB$ is formed by taking the product of entries of A and B in the (i,j) mesh processor and adding it to the current value of $C_{i,j}$ (which is initially 0), rotating all rows of A left one place cyclically and all columns of B up one place cyclically, and repeating the first two steps until all products are formed and accumulated. The number of steps executed by this algorithm is $s-1$ if all the data is initially on the mesh in the correct positions and data can be moved horizontally and vertically simultaneously. Since the classical serial algorithm executes $s^2(2s-1)$ steps, this 2-D systolic algorithm exhibits a speedup of $O(p)$ where $p = s^2$. It requires $s-1$ data movement steps.

This 2-D algorithm can be extended in the obvious way to compute the product of two

$s\sqrt{m/3} \times s\sqrt{m/3}$ matrices on a p-processor ($p = s^2$), m-word/processor 2-D mesh in which two $\sqrt{m/3} \times \sqrt{m/3}$ submatrices instead of two elements is stored in the memory of each mesh processor. Thus, the processor in row i column j contains a $\sqrt{m/3} \times \sqrt{m/3}$ submatrix from each of A, B, and C. The product of submatrices of A and B is formed and added to the submatrix of C which is initially the zero matrix. The rotation of submatrices of A and B described above is then done and the process repeated.

This algorithm requires at most $2s\left(\sqrt{m/3}\right)^3$ computation steps to multiply two $s\sqrt{m/3} \times s\sqrt{m/3}$ matrices and $O(s(m + \lambda))$ data movement steps where λ is the latency of the mesh. Thus, the running time of this algorithm, including the time for data movement between mesh processors, is $O\left(s\left(m^{3/2} + \lambda\right)\right) = O\left(\sqrt{p}\left(m^{3/2} + \lambda\right)\right)$.

This algorithm can be extended to run on the $s \times s \times \cdots s$ d-dimensional mesh, d even and $s = p^{1/d}$, in which each processor is identified by a d-tuple $\underline{a} = (a_1, a_2, \ldots, a_d)$, $0 \leq a_i \leq s-1$, and processors whose d-tuples differ in one place and whose values differ by 1 are adjacent. The processors with the same values for a_{d-1} and a_d form a $(d-2)$-dimensional submesh. When the submeshes are treated as processor nodes, they form a 2-D mesh.

To multiply $s^k\sqrt{m/3} \times s^k\sqrt{m/3}$ matrices A and B, $k = d/2$, store $m/3$ entries of these matrices in each the p processor memories. Each matrix associated with a $(d-2)$-dimensional mesh is an $s^{k-1}\sqrt{m/3} \times s^{k-1}\sqrt{m/3}$ matrix. To form the product of matrices A and B, we view them as 2-D matrices and recursively invoke the algorithm to multiply matrices on a $(d-2)$-dimensional mesh. Because the outer 2-D algorithm performs $O(s)$ computations (each of which takes time $O\left(s^{k-1}m^{3/2}\right)$) and $O(s)$ data movements (each of which is done in time $O(\left(s^{k-1}(m + \lambda)\right))$, the total time to multiply two $s^k\sqrt{m/3} \times s^k\sqrt{m/3}$ matrices on the mesh is $O\left(s^k(m^{3/2} + \lambda)\right) = O\left(\sqrt{p}(m^{3/2} + \lambda)\right)$. Thus, two matrices that fit on the d-dimensional mesh can be multiplied with a full speedup when m is sufficiently large relative to p and λ.

Consider now the computation of the product $W = U \times V = (W_{r,s})$ of two $n \times n$ matrices $U = (U_{r,s})$ and $V = (V_{r,s})$ where we view U, V and W as $(n/q) \times (n/q)$ matrices, $q = s^k\sqrt{m/3}$, with entries that are $q \times q$ matrices for $1 \leq r, s \leq (n/q)$.

The product $W = U \times V$ is formed by taking the block inner products of rows of U with columns of V. These inner products are done with $(n/q)^3$ products and $(n/q)^2(n/q - 1)$ sums of $q \times q$ block matrices. The matrix multiplication algorithm uses the mesh to form such inner products.

To compute $W_{r,s} = \sum_{t=1}^{n/q} U_{r,t}V_{t,s}$, for each value of $1 \leq t \leq n/q$ move $U_{r,t}$ and $V_{t,s}$ to the mesh, multiply and discard them but add their product P to the partial sum of $W_{r,s}$ which is initially the zero matrix. $W_{r,s}$ is stored on the front-end memory after its computation is complete.

Suppose now that a fraction β of the processors on a mesh face are connected to a like number of front-end memory units. The time necessary to move one $q \times q$ matrix onto or off of the mesh is $O\left((m + \lambda)p^{1/d}/\beta\right)$.

It follows that the time to compute the product of two $n \times n$ matrices on the Mesh SuperHet is at most $(n/q)^3$ times the time to move two $s^k\sqrt{m/3} \times s^k\sqrt{m/3}$ matrices to the mesh (which is $O\left((m + \lambda)p^{1/d}/\beta\right)$, multiply them (which is $O\left(\sqrt{p}(m^{3/2} + \lambda)\right)$), and move the results back to the front-end (which is $O\left((m + \lambda)p^{1/d}/\beta\right)$. Here $q = s^k\sqrt{m/3} = \sqrt{mp/3}$. The bound given below follows from this analysis.

Theorem 6 *Let $T_{matrix}(n, m, p, d)$ be the number of steps to multiply two $n \times n$ matrices on the p-processor d-dimensional Mesh SuperHet with β channels between the mesh and front end when λ is the mesh interprocessor latency. Then for d even, $T_{matrix}(n, m, p, d)$ satisfies the following upper bound:*

$$T_{matrix}(n, m, p, d) = O\left(\frac{n^3}{p}\left[1 + \frac{\lambda}{m^{3/2}} + \frac{p^{1/d}}{\beta\sqrt{mp}}\left(1 + \frac{\lambda}{m}\right)\right]\right)$$

As this result demonstrates, a full speedup can be achieved by matrix multiplication on the Mesh SuperHet when $m \geq \max\left(\lambda^{2/3}, p^{2/d-1}/\beta^2\right)$. This result demonstrates that interprocessor latency, λ, is not important as long as it is small by comparison with $m^{3/2}$.

6 Conclusion

We have studied multigrain parallel computation using as model the Mesh SuperHet. It consists of the interconnection of serial and parallel processors. The parallel processor is a p-processor, d-dimensional toroidal mesh whereas the serial front end contains a serial processor and $p^{1-1/d}$ random-access memory modules. The processors on the face of the mesh are each connected directly to a memory module. The front-end processor executes ρ times as many instructions per unit time as each mesh processor. The time to move n words between two mesh processors or between a mesh processor and a memory module is $n + \lambda$ where λ is the latency of the machine. Each mesh processor has m words of local memory.

We have addressed the question of whether the serial front-end processor is essential and have studied three problems on the Mesh SuperHet, namely, sorting, the FFT, and matrix multiplication.

We have shown that a (somewhat artificial) problem exists that executes more quickly (in time $O(\sqrt{p})$) on the 2-D Mesh SuperHet than on either its serial or parallel components alone (in time $\Omega(p)$). We have also shown that the serial front end can be disabled without more than a constant-factor loss in the running time T of a problem if it uses the front end for fewer than $O(T/p^{1/d})$ steps.

In our examination of sorting, the FFT, and matrix multiplication we have derived a non-trivial lower bound on the time to sort and use the trivial factor of $O(p)$ speedup for the other two problems. We exhibit algorithms for these three problems that are asymptotically optimal when the amount of mesh-processor memory is sufficiently large. In particular, we show that on large problems when the amount of memory per processor, m, is large enough, a speedup of at most $O(p^{1-1/d}\log(mp/3))$ is possible whereas a speedup of $O(p)$ is possible for the FFT and matrix multiplication, even when the number of mesh-face channels is very small. We also show that the effect on speedup of ρ and λ is small for these problems if m is large enough.

As multi-grained computers become more prevalent, they will be used for a large variety of computationally intensive tasks. Thus, it is important to understand the opportunities of and limitations on this architecture. Not only must experiments be conducted to determine how best to exploit the architectural details of existing machines, studies such as this must be done to understand the fundamental limitations on such architectures and set the context for algorithm development.

Bibliography

[1] "Super Linkage," *Science* 251 (March 15, 1991), 1311.

[2] M. J. Atallah and J. -J. Tsay, "On the Parallel-Decomposability of Geometric Problems," *Algorithmica* 8 (1992), 209–231.

[3] K. E. Batcher, "Sorting Networks and Their Applications," *Procs. AFIPS Spring Joint Computer Conference* 32, 307–314.

[4] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time Bounds for Selection," *J. Computer and Systems Sciences* 7 (1973), 448–461.

[5] F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers," *Procs. ACM Symposium on Computational Geometry*, 298–307.

[6] G. N. Frederickson and D. B. Johnson, "The Complexity of Selection and Ranking in X+Y and Matrices with Sorted Columns ," *Journal of Computer and System Sciences* 24 (1982), 197–208.

[7] J. JáJá, in *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[8] H. T. Kung, "Memory Requirements for Balanced Computer Architectures," *Journal of Complexity* 1 (1985), 147–157.

[9] H. Nicholas, G. Giras, V. Hartonas-Garmhausen, M. Kopko, C. Maher, and A. Ropelewski, "Distributing the Comparison of DNA and Protein Sequences Across Heterogeneous Supercomputers," *Procs. Supercomputing '91* (Nov. 18-22, 1991), 139–146.

[10] F. P. Preparata, "The Cube-Connected-Cycles: A Versatile Network for Parallel Computation," *Comm. ACM* 24 (May 1981), 300–309.

[11] F. P. Preparata and J. E. Vuillemin, "Area-Time Optimal VLSI Networks for Multiplying Matrices," *Inf. Proc. Let.* 11 (1980), 77–80.

[12] J. E. Savage, *Models of Computation: Exploring the Power of Computing*, Addison Wesley, Reading, MA, 1998.

[13] J. E. Savage and S. Swamy, "Space-Time Tradeoffs on the FFT Algorithm," *IEEE Trans. on Info. Th.* IT-24 (Sept. 1978), 563–568.