The Uniform Memory Hierarchy Model of Computation

Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker
IBM Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

November 2, 1992

Abstract — The *Uniform Memory Hierarchy* (UMH) model introduced in this paper captures performance-relevant aspects of the hierarchical nature of computer memory. It is used to quantify architectural requirements of several algorithms and to ratify the faster speeds achieved by tuned implementations that use improved data-movement strategies.

A sequential computer's memory is modelled as a sequence $\langle M_0, M_1, ...\rangle$ of increasingly large memory modules. Computation takes place in M_0. Thus, M_0 might model a computer's central processor, while M_1 might be cache memory, M_2 main memory, and so on. For each module M_u, a bus B_u connects it with the next larger module M_{u+1}. All buses may be active simultaneously. Data is transferred along a bus in fixed-sized blocks. The size of these blocks, the time required to transfer a block, and the number of blocks that fit in a module are larger for modules farther from the processor. The UMH model is parameterized by the rate at which the blocksizes increase and by the ratio of the blockcount to the blocksize. A third parameter, the transfer cost (inverse bandwidth) function, determines the time to transfer blocks at the different levels of the hierarchy.

UMH analysis refines traditional methods of algorithm analysis by including the cost of data movement throughout the memory hierarchy. The *communication efficiency* of a program is a ratio measuring the portion of UMH running time during which M_0 is active. An algorithm that can be implemented by a program with positive asymptotic communication efficiency is said to be *communication-efficient.* The communication efficiency of a program depends on the parameters of the UMH model, most importantly on the transfer cost function. A *threshold* function separates those transfer cost functions for which an algorithm is communication-efficient from those that are too costly. Threshold functions for matrix transpose, standard matrix multiplication, and Fast Fourier Transform algorithms are established by exhibiting communication-efficient programs at the threshold and showing that more expensive transfer cost functions are too costly.

A parallel computer can be modelled as a tree of memory modules with computation occurring at the leaves. Threshold functions are established for multiplication of $N \times N$ matrices using up to N^2 processors in a tree with constant branching factor.

1 Overview

Theoretical computer science does not address certain performance issues important for creating scientific software. Careful tuning can speed up a program by an order of magnitude [GJMS88]. These improvements follow from taking into account various aspects of the memory hierarchy of the target machine. This paper presents a model of computation that captures these performance-relevant characteristics of computers.

Big-O analysis on the traditional Random Access Machine (RAM) model of computation [AHU74] ignores the non-uniform cost of memory accesses. Section 2 illustrates the gap between traditional theory and practice on a naive matrix multiplication program. For the RAM model, where every memory access takes one unit of time, the complexity[1] of this program is $O(N^3)$. However on a real computer[2], cache misses and address translation difficulties slow moderately large computations down considerably (by a factor of 40). On problems that are too big for main memory, page misses further reduce performance (first by another factor of 25 and, for still larger N, by a further factor of 500). A graph of time versus problem size looks more like $O(N^5)$ than $O(N^3)$. To achieve $O(N^3)$ performance, the programmer[3] must understand the computer's memory hierarchy.

A first step is the Memory Hierarchy (MH) model of Section 3. A sequential computer's memory is modelled as a sequence of (usually increasingly large) memory modules $\langle M_0, M_1, ...\rangle$, with buses connecting adjacent modules. All of the buses can be active at the same time. Data in module M_u is partitioned into blocks, and each block is further divided into subblocks. A *level-u block* is the unit of transfer along the bus connecting module M_u and next larger module M_{u+1}. A level-u block is also a *level-$(u+1)$ subblock*. The MH model has three parameters per memory module that tell how many data items are in each block, how many blocks the module can hold, and how many cycles it takes to transfer a block over the bus. Computations — arithmetic operations, comparisons, indexing, and branching —

[1]That is, the worst-case running time on a RAM as a function of problem size N, where N is the maximum of the matrix dimensions.

[2]In this case, an IBM RISC System/6000 model 530 computer, hereafter the RS/6000.

[3]Matrix multiplication is sufficiently simple that it may suffice to use a compiler with such optimizations as strip mining and loop interchange [PW86].

take place in M_0. If a program is written against the MH model, and the model's parameters reflect a particular computer, then the program can be translated to run efficiently on the computer. (Translating a MH program may be easy or hard depending on details of the machine, operating system and programming language.)

The many parameters of the Memory Hierarchy model can obscure algorithm analysis. Furthermore, an algorithm designer would like be able to write a single program that can be compiled to run well on a variety of machines. These considerations call for a model that is more accurate than the RAM model and yet is less complicated than the MH model. The Uniform Memory Hierarchy (UMH) model of Section 4 reduces the zoo of MH parameters to two constants (the aspect ratio and the packing factor) and a transfer cost function. This model characterizes memory hierarchies well enough to confront an algorithm designer with many of the problems faced by a performance tuner. Yet the model is simple enough to allow analysis. We believe that it will be possible to construct compilers that translate UMH programs to run efficiently on a broad class of computers.

UMH analysis of a program compares the performance on a RAM model to the performance on a UMH model. In order to focus on the different communication costs — and not the computational power — of the two models, the RAM model used in this paper is a MH model with an infinitely large module M_1. The *communication efficiency* of a program is the ratio of its RAM-complexity to its UMH-complexity; it is a function of the problem size. A program is said to be *communication-efficient* if it has asymptotic communication efficiency greater than 0. Whether or not an algorithm has a communication-efficient program depends on the transfer cost function. Section 4 defines threshold functions that separate transfer cost functions that allow an algorithm to be implemented communication-efficiently from those that do not.

Section 5 gives a transpose program with positive asymptotic communication efficiency[4] assuming constant transfer cost throughout the memory hierarchy, and it is shown to have communication efficiency almost but not quite 1 for "nicely aligned" matrices. (A matrix stored in level u is nicely aligned if the first element of each column begins a level-u block.) Section 5

[4]Simultaneous data transfers on the multiple buses overcome the $\Omega(N^2 \log\log N)$ lower-bound for the corresponding Block Transfer model [ACS87].

also shows than no program for transposing a square matrix can have an asymptotic communication efficiency of 1.

Section 6 shows how to rechoreograph the naive matrix multiplication program of Section 2 to be communication-efficient even if the transfer costs rapidly increase as one goes up the hierarchy. At the threshold function, matrix multiplication has communication efficiency of almost 1 for nicely aligned matrices, but again communication efficiency of 1 is shown to be unattainable.

Section 7 examines a Fast Fourier Transform (FFT) algorithm at its slowly increasing threshold function. The straightforward program for this algorithm has UMH complexity $O(N \log N \log\log N)$, but the RAM complexity is $O(N \log N)$, so its asymptotic communication efficiency is 0. Such a program is said to be *communication-bound*. Careful rechoreography of the algorithm results in a communication-efficient program.

Section 8 compares the UMH model to other models that incorporate non-uniform costs of memory references.

Section 9 incorporates parallelism in the model. A parallel computer is modelled as a uniformly branching tree of memory modules with processors at the leaves. Threshold functions are established for parallel matrix multiplication of $N \times N$ matrices for various numbers of processors. For instance, with N processors, a communication-efficient program requires constant transfer costs.

2 Naive Matrix Multiplication

This section examines the performance of a simple 4-line matrix multiplication program. Classical RAM analysis says that the running time of this program should be $O(N^3)$. Experimental data on an RS/6000 suggests that $O(N^5)$ might be a better approximation. This discrepency is explained by taking into account the memory hierarchy of the computer.

Program 1 computes $\mathsf{C} := \mathsf{C} + \mathsf{A}\ \mathsf{B}$, where A, B, and C are matrices[5]. If A, B, and C are square $N \times N$ matrices, the assignment statement is iterated

[5]Linear algebra packages, such as ESSL [IBM86] and LAPACK [ABetc92] use this "update" form since it facilitates problem decomposition. Analysis of $\mathsf{C} := \mathsf{A}\ \mathsf{B}$ is similiar.

```
NaiveMM(A[1:n,1:l], B[1:l,1:m], C[1:n,1:m]):
    REAL VALUE: A, B; VALUE RESULT: C
    INTEGER VALUE: n, m, l
    INTEGER: i, j, k
    FOR i FROM 1 TO n
        FOR j FROM 1 TO m
            FOR k FROM 1 TO l
                C[i,j] := C[i,j] + A[i,k]B[k,j]
    END
```

Program 1: A Naive Program for Standard Matrix Multiplication.

N^3 times. The *RAM*-complexity of Program 1 is:

$$N^3 + O(N^2).$$

This assumes that a multiply-and-add instruction, along with incrementing address and index variables, loading the needed data into registers, and executing a conditional branch, can all be performed as a single operation. The $O(N^2)$ term reflects the loop-setup cost.

To see how closely the *RAM* analysis corresponds to practice, we ran a FORTRAN version of Program 1 on square matrices that varied in size from 35×35 to 2000×2000. The results of this experiment are reported in Figure 1. The logarithm[6] of N is the horizontal axis of this graph; the vertical axis is the log of the running time of the program (in cycles) divided by N^3. The *RAM* analysis leads one to expect that the data points would approach the horizontal axis. This does not appear to be the case. A circle ($\circ$) plots a data point; a small question mark (?) plots an estimated[7] value. The solid line, a rough attempt to approximate the observed data with a straight line[8], represents a running time of $10^{-4}N^5$. The analysis of Section 4 will ratify this $O(N^5)$ behavior.

[6]In this paper, an unadorned log is the logarithm base 10, and lg is base 2.

[7]The last data point ($N = 2000$) represents a running time of over five days. It was not practical to gather more data.

[8]The slope of a line through points plotted on a log–log graph corresponds to the exponent of a polynomial approximation of the data.

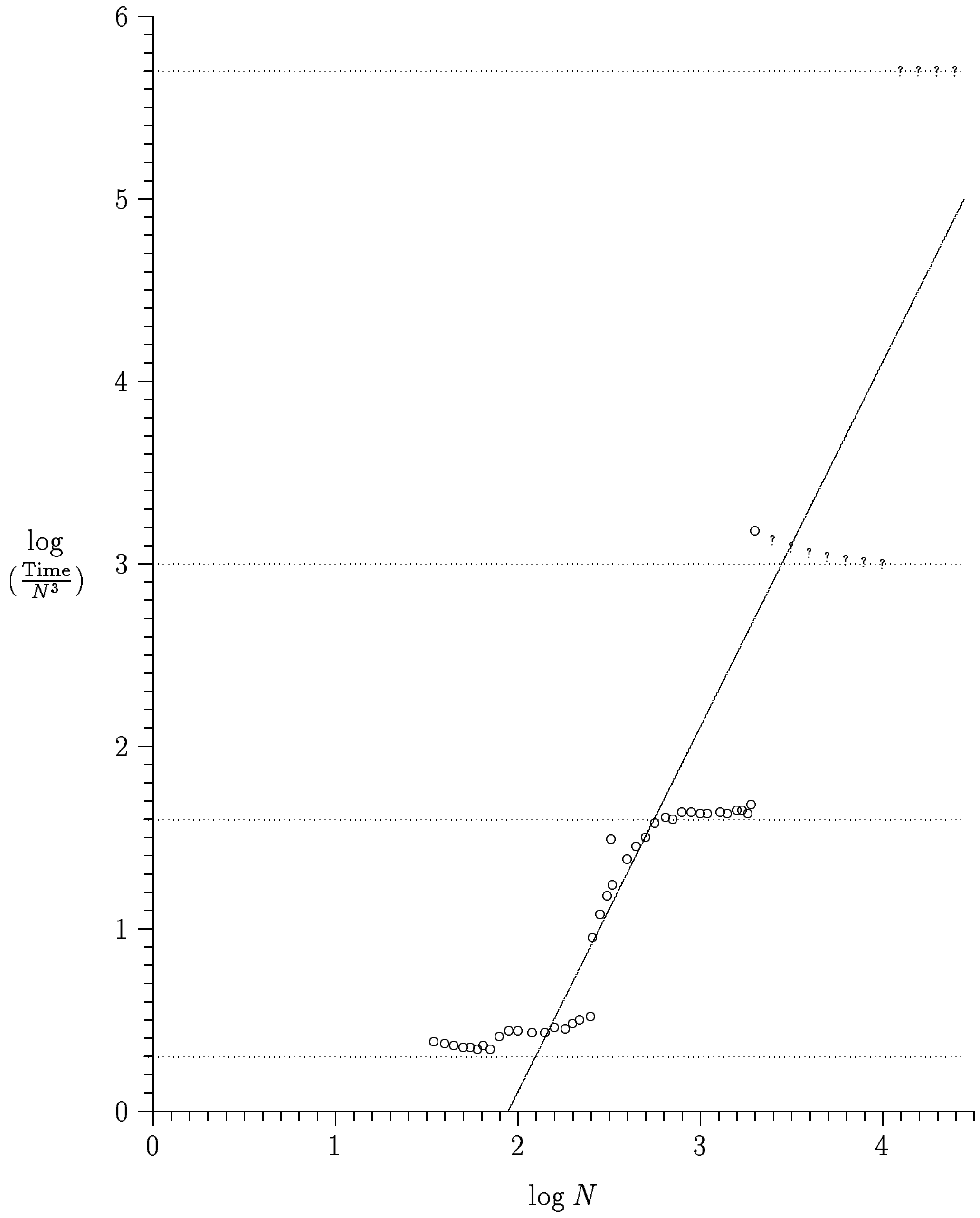


Figure 1: Execution Rate of Naive Matrix Multiplication on an RS/6000.

An alternate interpretation is represented by the horizotal lines. The horizontal axis represents one floating-point multiply-and-add instruction per cycle[9]. The first dotted line represents the cost of loading two data items per multiply-add. It fits the initial data points. A second dotted line, representing the cost of translating a virtual address into a real one for each multiply-add, fits the next group of points. A third represents the cost of a page miss for every 512 multiply-adds. This line picks up the final data point and the first group of estimated values. The final dotted line, representing the cost of a page miss for every multiply-add, is an estimate of the performance of Program 1 for matrices bigger than about $12,000 \times 12,000$. Verifying a single point on this line would require a 3.5 gigabyte disk and 1095 years. Before explaining why Program 1 exhibits this somewhat bizarre behavior, let's take a deeper look at matrix multiplication in general.

Calculation of the updating matrix product

$$\mathsf{C[1{:}n,1{:}m] := C[1{:}n,1{:}m] + A[1{:}n,1{:}l]\ B[1{:}l,1{:}m]}$$

can be visualized as a $\mathsf{n}\times\mathsf{m}\times\mathsf{l}$ rectilinear solid as in Figure 2. Matrices A and B form the left and top faces of the solid, and the initial value of C is on its back face. The final value of C is formed on the solid's front. Each unit cube in the interior of the solid represents adding the product of its projections on the A and B faces to a running sum. An element of the front face is the final value of the running sum computed in the unit cubes behind it. Evaluation of all of the unit cubes in any order will compute the result[10]. Program 1 first processes the upper left hand row of unit cubes (indexed by k from back to front), then moves to the second and subsequent rows (indexed by j from left to right) of the upper layer, and finally proceeds to the second and subsequent layers (indexed by i from top to bottom).

The vertical axis of Figure 1 represents the log of the average cost (in cycles) of evaluating a single unit cube on the RS/6000. Whenever a cube is computed, the corresponding elements of the A, B, and C matrices must be in registers. The compiled code leaves the value of an element of C in a register for the duration of the inner loop, but the other two values must be

[9]This is the peak speed of the RS/6000. The carefully-tuned matrix multiplication routine DGEMM in the LAPACK library achieves 90% of peak speed for large square matrices that fit in main memory.

[10]Numerical analysts prefer that the unit cubes behind a fixed element of the C face be evaluated from back to front, so that round-off effects can be better understood. This still leaves many more than N^{N^3} legal orders of evaluation.

Figure 2: The Matrix Multiplication Solid.

loaded. If a value is in cache, moving it into a register takes 1 cycle. If the value is not in cache, the *cache line* containing it and adjacent values must be transfered from slower memory into the cache. This requires at least 7 more cycles[11]. The cache consists of 512 cache lines, each containing 16 adjacent doublewords. Consider the cost of processing the i-th layer of the solid. Since there are repeated passes along the i-th row of A, the cache's replacement algorithm gives priority to keeping the entire row in cache. The matrices in our experiment were stored in column-major (FORTRAN) order, so the row of A takes up N lines of cache. If $N \leq 75$, there is room in cache not only for a row of A but also for the entire B matrix and the current row of C. Thus, the computation takes about 2 cycles per unit cube, the cost of bringing the A and B values into registers[12].

When N grows bigger than 75, even though the i-th row of the A matrix may remain in cache, the B matrix and the i-th row of C will not. When N gets a little larger, every 16 iterations of the inner loop will entail a cache miss, and the computation takes on average about 2.5 cycles per unit cube. When N grows to 481, the i-th row of A does not remain entirely in cache since a column of B also takes up 31 cache lines and the current element of C takes 1. By the time N gets to 640, each iteration of the inner loop entails a cache miss. When this happens, one says the cache is *thrashing.*

In fact, something more dramatic happens as well. In order to bring a data item from main memory into cache, its real address must be derived from its virtual address. A *Translation Lookaside Buffer* (TLB) speeds this mapping for some recently referenced pages of main memory. When $N > 256$, the row of A no longer fits in the RS/6000's TLB. By the time N reaches 512, evaluation of each unit cube entails at least one 40-cycle penalty to update the TLB.

As N grows beyond 2000, B and the i-th row of C will not fit in the computer's 48 megabytes of main memory. Soon, every 512 iterations[13] of the inner loop entails a page miss on the B matrix. Each assignment to the C matrix also causes a page misses. It takes perhaps a fiftieth of a second

[11]Actually, bringing a line into cache takes 14 cycles, but the last 6 are usually overlapped with computation.

[12]The floating-point pipeline of the RS/6000 can initiate one multiply-add operation per cycle. Concurrently, one floating-point register can be loaded per cycle. Since Program 1 requires two loads per multiply-add, it does not keep the floating-point pipeline busy.

[13]There are 512 doublewords to a page.

(500,000 cycles) to bring a page in from disk. Computing one layer takes about $1,000N^2 + 500,000N$ cycles. And when N grows so large that the i-th row of A no longer fits in main memory (this happens at about $N = 12,000$ since each element is on a separate page), there will be a page miss per unit cube, degrading performance by a further factor of 512.

Program 1 is a poor implementation of matrix multiplication. Well-known techniques [RR51, MK69, IT88, GJMS88, AG89] perform the same collection of multiply-adds in a different order. The matrix multiplication solid of Figure 2 is partitioned into pieces, for instance ones that are 40 units in each direction, which are computed one at a time. Processing a piece requires computation proportional to its volume. If the surface area of the piece is small enough to fit into, say, cache, then the cost of cache misses on the piece will be roughly proportional to the surface area. Thus, the program's efficiency is related to the volume-to-surface area ratio, suggesting the use of roughly cubic-shaped pieces[14]. To improve usage of all levels of the memory hierarchy — the registers, cache, address translation mechanisms, main memory, and disk — a recursive decomposition of pieces into subpieces into subsubpieces can be used.

State-of-the-art compiler technology (e.g., strip mining and loop interchange [PW86]) can automatically make these improvements to Program 1. On slightly more complicated problems, the capabilities of compilers are limited [CK89]. And for even more complex problems, there are improvements that we cannot expect a compiler to discover. The replacement of recursive transposes by a single bit-reversal permutation in the Fast Fourier Transform algorithm of Section 7 might be an example.

An algorithm designer who models computer memory as a single level (as in the RAM model) may not realize that such improvements are necessary. The standard advice, "Strive for spatial and temporal locality of reference," is rather vague. As we have seen, programs remain efficient for a range of problem sizes; then performance drops precipitously. If one has no intuition about how much "locality" is needed, then one might introduce far more overhead than needed, performing extra copies or invoking recursion excessively. This can destroy performance almost as surely as having too little locality. The Memory Hierarchy model of the next section confronts

[14]If two consecutive pieces share a surface, the communication cost is reduced, suggesting it might be better to use a pile of somewhat squat pieces.

the algorithm designer with problems that are of direct relevance to efficient programming.

3 The Memory Hierarchy Model

The following abstract model of computation reflects many of the memory and communication limitations of sequential computers. A *memory module* M_u is a triple $\langle s_u, n_u, l_u \rangle$. For any (possibly infinite) sequence $\sigma = \langle M_0, M_1, ... \rangle$ of memory modules, MH_σ is a *memory hierarchy.* Module M_u is said to be *level* u of the hierarchy. Intuitively, M_u is a box that can hold n_u blocks, each consisting of s_u data items, and a bus B_u that connects the box to the next module, M_{u+1}. Blocks are further partitioned into *subblocks.* The subblock is the unit of transfer along the bus from the previous module. B_u copies a level-u block atomically in l_u cycles to or from level-$(u+1)$, overwriting the old contents.

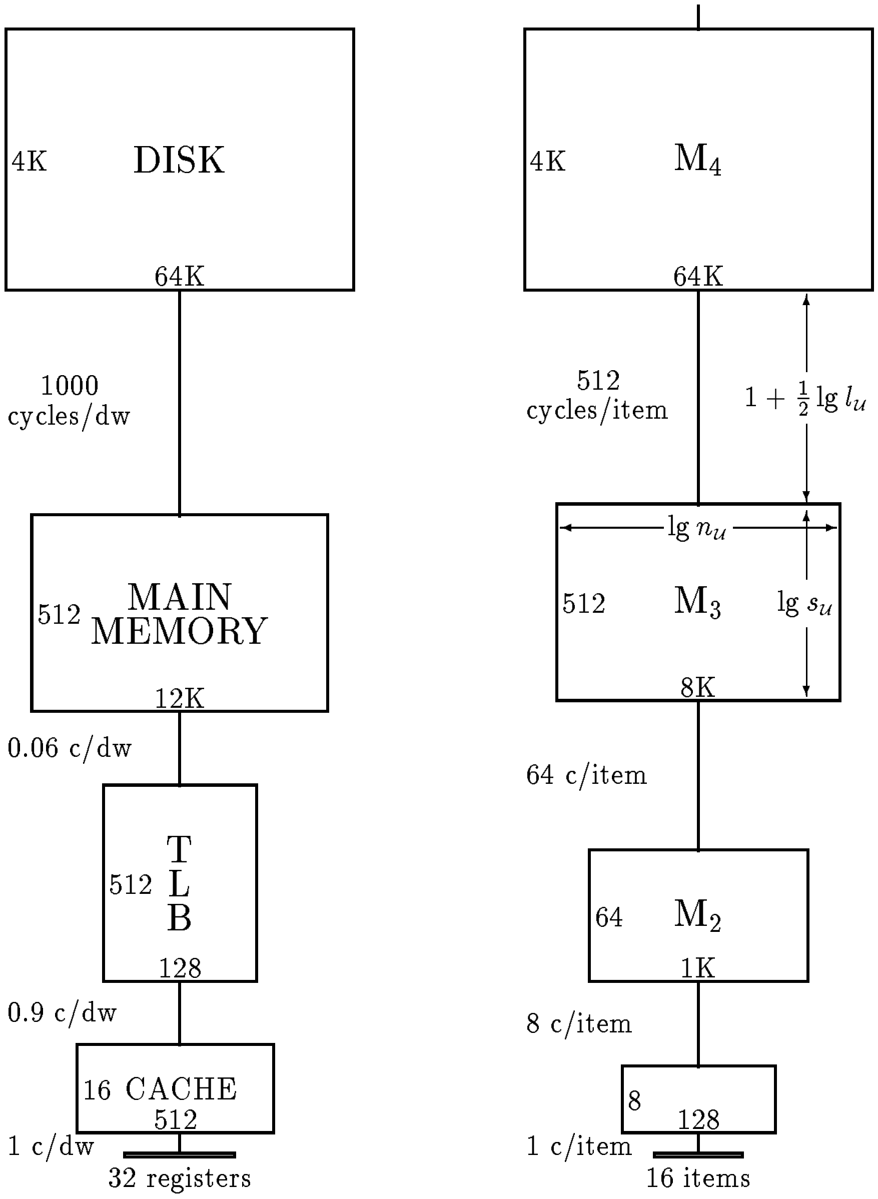
MH_σ can be depicted as a tower of modules with level 0 at the bottom [ACS90]. Two memory hierarchies are shown in Figure 3 (the second is a UMH, explained in the next section.) In the figure, the horizontal and vertical dimensions of the rectangle depicting M_u are proportional to the logs of n_u and s_u respectively. A logarithmic scale is used in the figure so that parameters that differ by a factor of over 1000 can be depicted visually.

The data items comprising a subblock at a given level are an indivisible unit as far as that level is concerned. These items can only be rearranged or modified by moving the subblock down to a lower level. To allow individual data items to be altered, level 0 is given the ability to perform computations on the data items, resulting in new data values. Since individual bits can be changed at level 0, the subblock of M_0 is the bit.

All buses can be active concurrently. The only restriction is that no data item in the block being transfered on bus B_u is available to buses B_{u-1} or B_{u+1} until the transfer of the entire block is complete.

It is useful to define other derived parameters. For convenience, the following list gives standard notation for both the primitive and the derived parameters:

- s_u, the *blocksize* or number of data items in a level-u block,



(a) RS/6000 Model 530 (b) $UMH_{16,8,8^u}$

Figure 3: Two Memory Hierarchies.

- n_u, the *blockcount* or number of blocks that M_u holds,
- l_u, the *latency* or time to move a block on B_u,
- $v_u \equiv n_u s_u$, the *volume* or number of data item held by M_u,
- $a_u \equiv n_u / s_u$, the *aspect ratio* of M_u,
- $p_u \equiv s_u / s_{u-1}$, the *packing factor* at M_u,
- $t_u \equiv l_u / s_u$, the *transfer cost* of B_u in cycles per item, and
- $b_u \equiv s_u / l_u$, the *bandwidth*, the inverse of transfer cost of B_u.

A memory hierarchy is not a complete model of computation, but instead is used to model the movement of data. To study any particular algorithm, reasonable assumptions need to be made about what a "data item" is and what operations level 0 can perform on data items. All computations considered in this paper are oblivious, in the sense that there is a fixed order in which the data items are accessed. If the MH model were to be used to analyze a non-oblivious computation such as merge sort, then it would be necessary to define carefully operations that control the movement of data through the hieararchy.

Figures 3(a) and 4 show the MH model of the RS/6000 used in the experiment of Section 2. For double-precision floating-point matrix operations, the basic data item will be an 8-byte doubleword, abbreviated *dw*. The RS/6000 has 32 floating-point registers that each hold a doubleword, an 8Kdw cache (512 128-byte cachelines), a 128-entry TLB (each entry providing virtual-to-real address mapping for a 512dw-page), 6Mdw (48 megabytes) of real memory, and about 256Mdw of disk storage [BW90, H92]. The disk is modeled, somewhat arbitrarily, as 64K 4Kdw-tracks. Level 0 can perform a multiply-add every cycle, and fixed-point and branching instructions are free[15]. It takes 8 cycles to load a value that is not in cache into a register and a total of 14 cycles to bring the cacheline into cache. The time to service

[15]If the data items are in registers and the 2-cycle pipelined multiply-add hardware is used effectively, the hardware can perform a floating-point multiply-add instruction every 40-nanosecond cycle. At the same time, separate fixed-point and branch units (with a separate instruction cache) do the branching, addressing and register loading needed to support the floating-point operations.

a TLB miss depends somewhat on the state of the translation table data structure; at minimum 32 cycles are required before the data can begin to move into cache. The time to service a page miss depends somewhat on the location of the disk-head.

level u	block-size s_u	block-count n_u	latency l_u	volume v_u	aspect ratio a_u	packing factor p_u	transfer cost t_u
4 Disk	4K	64K	–	256M	16	8	–
3 Main	512	12K	∼500K	6M	24	1	∼1K
2 TLB	512	128	∼32	64K	0.25	32	∼0.06
1 Cache	16	512	14 (or 8)	8K	32	16	∼0.9 (or 0.5)
0 CPU	1	32	1	32	32	64	1

Figure 4: RISC System/6000 Model 530 Memory Hierarchy.

In addition to giving the primitive parameters, Figure 4 shows some derived ones. The name "aspect ratio" comes from viewing M_u as a rectangle of height s_u and width n_u. The packing factor tells how many subblocks are packed into a block. Notice that, except for level 2, the aspect ratios and packing factors are moderately small integers[16]. This would still be true if the data item were a byte rather than a doubleword — aspect ratios listed as 32 would be 4 instead. The significance of these parameters is twofold. First, the performance of the algorithms we consider is rather insensitive to the actual values of aspect ratios and packing factors, provided they meet certain minimum values[17]. Second, for the machines we have experience with, these minimum values are usually met. Consequently, the UMH model defined in the next section assumes these parameters are small integer constants.

[16]The fact that the RS/6000 has a very small a_2 aspect ratio correlates with our experience that achieving peak performance on linear algebra kernels was made more difficult by the relatively small number of TLB entries. Fortunately, the very low transfer cost to the TLB mitigates its small blockcount.

[17]Experimental evidence [CSB86] can be interpreted to show that for a given volume cache, the minimum cache miss ratio and communication costs occur for aspect ratios between 4 and 16.

The story is quite different for the transfer costs t_u — the average number of cycles required to move a data item. There is no consistency among the t_u's at different levels. The transfer cost t_2 associated with the TLB is abnormally low since only the data's addressing information — not the data itself — must be processed when a TLB miss occurs. On the other hand, t_3 is very large because magnetic media are much slower than semiconductor memory. Since an algorithm's performance is sensitive to these costs, the UMH model is parameterized by a transfer cost *function*. The influence of this function is studied in depth.

4 UMH Analysis

While the Memory Hierarchy model is useful for tuning algorithms for particular machines, it is too baroque for clean algorithm analysis. The *Uniform* Memory Hierarchy (UMH) model makes simplifying assumptions that are representative of typical values. The UMH model assumes that all levels of the memory hierarchy have the same aspect ratio $a_u = \alpha$, and that all packing factors $p_u = \rho$. The transfer costs of buses in a UMH are given by a function $f(u)$. The argument u is a level number, and $f(u)$ is a non-negative number giving the transfer cost in cycles per item of bus B_u. Formally, the Uniform Memory Hierarchy $UMH_{\alpha,\rho,f(u)}$ is MH_σ, where $\sigma = \langle M_0, M_1, ...\rangle$ and $M_u = \langle \rho^u,\ \alpha\rho^u,\ \rho^u f(u)\rangle$. We require ρ to be an integer greater than 1, and we are generally satisified with programs that are communication-efficient for some small integral value for α. Generally, the transfer cost functions $f(u)$ are very simple, for instance $\mathbf{1}$ (the constant function), u (the identity), or ρ^u. A UMH not dissimilar from the RS/6000 is shown in Figure 3(b).

There are three levels of specification of a computation on a UMH:

1. An *algorithm*. The algorithm specifies the mathematics of a computation. Execution order and data movement strategies are left unspecified. The algorithms in this paper are described informally, but they may be formally specified in a concurrent programming notation or as computational circuits, dataflow graphs, or program dependence graphs [FOW87]. The matrix multiplication solid of Figure 2 is an

example of an algorithm; there are different algorithms for matrix multiplication that use fewer than $O(N^3)$ operations [AHU74].

2. A *program*. A program specifies the order of evaluation and the data movement strategy of an algorithm. The process of finding a program for an algorithm will be called *choreographing* the algorithm. The order of evaluation can be given as a sequential *RAM* program, as with Naive Matrix Multiplication in Program 1. Data movement strategies are often be given implicitly, e.g., as demand fetching with least-recently-used replacement. Programs in this paper will be written in a procedural notation described below.

3. A *schedule*. Given a particular set of input values to a program, a schedule tells exactly when each block is moved along each bus. Even if a program is oblivious, the schedule may depend on the *alignment* of the data, that is, where each data item is in relationship to the block boundaries at each level of the memory hierarchy. It is not necessary to write a schedule to run a program; rather, the schedule is a trace of what happens when a program runs. Schedules are used in proofs.

The programs for our algorithms are specified as one or more procedures for each module in the memory hierarchy. Typically, a procedure is called from a procedure one level above and makes a sequence of calls to procedures one level below. To achieve greater efficiency, the arguments to a call may be moved down before the results of a previous call have been moved back up. Each memory module can be viewed as solving a sequence of problems with a three-stage pipeline. In the first stage, the inputs to a procedure are moved down from the level above. In the second stage, the procedure is invoked. This will entail writing subproblems down to (and reading their solutions up from) the level below. In the final stage, the solution is moved back up to the next level. Because the second stage uses a different bus, its activity can be concurrent with that of the first and last. Computation, which occurs at M_0, is overlapped with communication.

A program specifies, for each bus, the order in which data items move down the bus and the order in which they move up. The interleaving of these two streams must also be specified. Unless another interleaving is given, we assume that data moves down at the last possible moment that will still enable it to arrive just-in-time when it is needed at M_0, and data moves up at the first possible free cycle. This strategy prevents a memory module from being flooded with more data than it can hold and is consistent with the pipelining of procedure calls. Our model of computation is given the ability to achieve appropriate schedules for such programs[18].

Problem specification entails defining where problem instances are located. We adopt the convention that $N \times N$ input matrices initially reside at level $\lceil \log_\rho N \rceil$ of a UMH; on an MH they reside at the lowest level in which they fit. For a given choice of MH or UMH parameters, the $(U)MH$ complexity of a program is the function of the problem size N that gives the maximum running time, taken over all problem instances of size N, of the best schedule for that instance.

Our interest is often not only in the big-O complexity of a program, but in whether it wastes even a constant factor of its RAM speed. Thus, we make the following definition:

- The *communication efficiency* $c(N)$ of a program is the ratio of the RAM complexity to the UMH complexity[19]. For the purpose of calculating $c(N)$, the RAM complexity of a program is taken to be the MH complexity on a two-level hierarchy with the UMH's M_0 module and an arbitrarily large M_1. Thus, the RAM complexity is the cost of computation and moving data along the B_0 bus[20].

The communication efficiency of a UMH program is analogous to measurements in practice of the fraction of a computer's peak speed sustained by an application. A program is said to be *communication-efficient* if $c(N)$

[18]One approach to implementing such a model would have a *controller* for each bus to execute the procedure(s) for the adjoining module. An alternate approach would have an *addressing unit* responsible for dispatching transfer requests to each of the buses. It is easy to verify that such a unit need only dispatch a constant number of transfer requests per cycle provided the transfer cost is nonincreasing and the packing factor is at least 2. In order to extend the model to handle nonoblivious computations, careful attention would have to be given to specifying the computational power of the addressing unit or controllers.

[19]For a given problem size, the instances with the worst running times on the RAM and on the UMH may be different. We could have chosen to define $c(N)$ to be the worst ratio over all instances of size N. However, $c(N)$ would then be unduly biased by instances that were particularly easy on a RAM but of average difficulty on a UMH.

[20]In general, when we describe an algorithm, we will also specify M_0 and give B_0 enough bandwidth so that our definition of RAM complexity coincides with the usual notion.

is bounded below by a positive constant. Otherwise, if $c(N)$ gets arbitrarily close to zero, the program is *communication-bound.* An algorithm is *communication-efficient* if it has a communication-efficient program; it is *communication-bound* if every program for it is communication-bound.

As an example, consider Program 1 for multiplying $N \times N$ matrices naively on $UMH_{16,8,8^u}$, the model chosen in Figure 3(b) to resemble the RS/6000. Given N, let $\nu = \lfloor \log_8(N/16) \rfloor$. Since $N \geq 16(8^\nu)$, the blockcount of level ν, there is not room in level ν to hold an entire row of the A matrix plus a block of B. Thus, level ν will thrash, that is, a new level-ν block will be moved in for each multiply-add. Notice that $\nu = \lfloor \log_8(N/16) \rfloor > \log_8(N/16) - 1 = \log_8(N/128)$. On $UMH_{16,8,8^u}$, bringing a block into level ν requires $8^{2\nu} > (N/128)^2$ cycles. Thus, the complexity of Program 1 on $UMH_{16,8,8^u}$ is at least $N^3(N/128)^2 = 2^{-14}N^5$. This matches our observed $10^{-4}N^5$ performance on the RS/6000 remarkably well. Since the RAM complexity of Program 1 is $O(N^3)$, the communication efficiency of Program 1 is $O(N^{-2})$; it is communication-bound on $UMH_{16,8,8^u}$. However, the standard matrix multiplication algorithm can be rechoreographed as in Section 6 to yield a communication-efficient program on this model.

Whether or not an algorithm A is communication-efficient on $UMH_{\alpha,\rho,f(u)}$ depends on $f(u)$. In many cases it is possible to nicely separate transfer cost functions for which A is communication-efficient from those for which it is communication-bound. To this end, we define:

- Given α and ρ, a function $f(u)$ is a *threshold function* for algorithm A if (1) A is communication-efficient on $UMH_{\alpha,\rho,f(u)}$, and (2) for any function $g(u)$ such that $\inf \frac{f(u)}{g(u)} = 0$, A is communication-bound on $UMH_{\alpha,\rho,g(u)}$.

It is not hard to show that if A is communication-efficient on a $UMH_{\alpha,\rho,f(u)}$ and $\inf \frac{f(u)}{g(u)} > 0$, then A is communication-efficient on $UMH_{\alpha,\rho,g(u)}$. Thus, a threshold function for an algorithm partitions all transfer cost functions into two sets — those which support communication-efficient programs for the algorithm and those which do not — according to whether $\inf \frac{f(u)}{g(u)}$ is positive or 0.

A candidate for a threshold function can be constructed to ensure that the time required to move the input and results along the bus to the next smaller module is bounded by the computation time. To be precise:

- The *candidate threshold function* $f_c(u)$ for an algorithm A is defined as the minimum, over all problem sizes N that reside in level $u+1$, of $RAM(N)/Data(N)$, where $RAM(N)$ is the RAM complexity[21] of A and $Data(N)$ is the maximum number of data items in the input and output of instances of size N.

The following lemma shows that the candidate threshold function satisfies property (2) for a threshold function.

lemma 4.1 *Let A be an algorithm that depends on all of its input data, and let $f_c(u)$ be the candidate threshold function for A. Suppose $g(u)$ is a function such that* $\inf \frac{f_c(u)}{g(u)} = 0$*. Then for any α and ρ, A will be communication-bound on $UMH_{\alpha,\rho,g(u)}$.*

proof: Given any program P for A, we must show that for any $\epsilon > 0$, there exists an N such that $RAM(N)/UMH(N) < \epsilon$, where $UMH(N)$ is the complexity of P on $UMH_{\alpha,\rho,g(u)}$.

Since $\inf \frac{f_c(u)}{g(u)} = 0$, we can choose v such that $\frac{f_c(v)}{g(v)} < \epsilon$. By the definition of $f_c(u)$, we can find N such that instances of size N reside in level $(v+1)$ and $f_c(v) = \frac{RAM(N)}{Data(N)}$. On $UMH_{\alpha,\rho,g(u)}$, just moving the input and output data items along the topmost bus B_v will take at least $g(v)Data(N)$ on some instance of size N. Thus, $UMH(N) \geq g(v)Data(N)$, and so $\frac{1}{g(v)} \geq \frac{Data(N)}{UMH(N)}$.

We now have $\epsilon > \frac{f_c(v)}{g(v)} \geq \frac{RAM(N)}{Data(N)} \frac{Data(N)}{UMH(N)} = \frac{RAM(N)}{UMH(N)}$, as required. •

It is not *a priori* clear that A will be communication-efficient on its candidate threshold function $f_c(u)$. However, this turns out to be the case for the algorithms considered below.

5 Matrix Transposition

[21]Since an algorithm dictates the computations to be performed, if M_0 is specified as in the previous footnote, then the RAM complexity of any program for A will be the same. Thus, the RAM complexity of an algorithm is well-defined.

Matrix transposition, $\mathsf{B} := \mathsf{A}^{\mathrm{T}}$, is typical of algorithms that people have tuned to memory hierarchies, improving performance significantly. This section shows that the constant function $\mathbf{1}$ is a threshold function for transposition. Transposing square matrices can be choreographed on $UMH_{6,\rho,\mathbf{1}}$ to be communication-efficient, and for nicely aligned matrices, to have asymptotic communication efficiency almost but not quite 1. For transfer cost functions asymptotically slower than a constant, transpose is communication-bound; its running time is the time to move the data to and from the highest level. The first step in analyzing transposition is to specify the algorithm and the details of the model that will be considered.

The problem considered here will be that of assigning to a matrix B the transpose of a separate[22] matrix A. The matrices are stored in column-major order. The transpose algorithm[23] analyzed here moves each element of A into level 0 and moves it back up into the proper position of B.

It is natural to let the data item of the UMH model be the individual matrix entry. Level 0 has whatever abilities it needs to keep track of indices. The transfer cost of bus B_0 is one cycle per item. Since each element of A must move down into level 0 (one cycle) and move back up (a second cycle), the RAM complexity of transposing $N \times N$ matrices is $2N^2$. Since there are N^2 input data items and N^2 result data items, the candidate threshold function is the constant function $\mathbf{1}$. The next subsection shows that it is indeed a threshold function.

5.1 A Communication-Efficient Transpose Program

Program 2, based on well-known methods, has an asymptotic communication efficiency on $UMH_{\alpha,\rho,\mathbf{1}}$ of nearly 1, provided the data are aligned with respect to block boundaries at all levels of the hierarchy. In the unaligned case, the program is slower but still communication-efficient.

Theorem 5.1 *Suppose that $N = \rho^{w}$, that $\alpha \geq 3$, and that $N \times N$ square matrices A and B are aligned so that the first element of each begins a level-w*

[22]The results of this section also hold for transposing a square matrix in place. The details are messier.

[23]There might be faster transpose algorithms that compress data to reduce communication time, or that avoid bringing certain elements into level 0.

```
MT$_{u+1}$ (A[1:n,1:m], B[1:m,1:n]):
    REAL VALUE: A; RESULT: B; INTEGER VALUE: n, m
    {This procedure resides in module $u+1$, and will only be called
    with n $\leq \rho^{u+1}$ and m $\leq \rho^{u+1}$ }
    INTEGER: i$_0$, i$_1$, j$_0$, j$_1$
    FOR i$_0$ FROM 1 TO n BY $\rho^u$
        i$_1$ := MIN(i$_0$+$\rho^u$-1, n)
        FOR j$_0$ FROM 1 TO m BY $\rho^u$
            j$_1$ := MIN(j$_0$+$\rho^u$-1, m)
            MT$_u$ ( A[i$_0$:i$_1$, j$_0$:j$_1$], B[j$_0$:j$_1$, i$_0$:i$_1$] )
    END
MT$_0$ (a, b):
    REAL VALUE: a; RESULT: b
    b := a
    END
```

Program 2: Matrix Transposition.

block. Then $\mathsf{MT}_w(\mathsf{A}, \mathsf{B})$ *on* $UMH_{\alpha,\rho,1}$ *takes at most* $(2+2/\rho^2)N^2$ *cycles to transpose.*

proof: We exhibit a schedule and verify that it is valid. Let A_0^w be A, and, for $u = w-1, w-2, ..., 0$ and $i = 0, ..., \rho^{2(w-u)}-1$, partition A_i^{u+1} into ρ^2 submatrices, each $\rho^u \times \rho^u$, denoted $\mathsf{A}_{i\rho^2}^u, ..., \mathsf{A}_{i\rho^2+\rho^2-1}^u$. Let B_i^u be the submatrix of B into which $(\mathsf{A}_i^u)^T$ will be copied; A_i^u and B_i^u are the arguments to the $(i+1)$-th call to MT_u. The schedule is as follows:

- A_i^u moves down bus B_u into M_u during cycles $(2i-1)\rho^{2u}$ to $2i\rho^{2u}-1$. It is transmitted by column from left to right. It displaces B_{i-2}^u.
- The data items of A_i^u are copied to B_i^u all the way down at M_0 during cycles $2i\rho^{2u}$ to $(2i+2)\rho^{2u}-1$.
- B_i^u moves up B_u to M_{u+1} during cycles $(2i+2)\rho^{2u}$ to $(2i+3)\rho^{2u}-1$. It is transmitted by column from left to right, and it overwrites the subblocks of M_{u+1} that held A_{i-1}^u.

To simplify the indices, time runs from cycle $-\rho^{2w-2}$, when A_0^{w-1} begins its downward descent into M_{w-1}, to cycle $2\rho^{2w}+\rho^{2w-2}-1$, when the last submatrix completes its upward ascent. Thus, the total time is as required. To validate this schedule, we must show the following for each $u < w$: (1) no submatrix moves down bus B_u before it arrives at M_{u+1}; (2) no submatrix moves up bus B_u before it is complete; (3) at any time, M_u is required to hold at most three submatrices; and (4) no two transfers on bus B_u overlap.

The first submatrix $\mathsf{A}_{i\rho^2}^u$ of A_i^{u+1} is scheduled to start moving down B_u at cycle $(2i\rho^2-1)\rho^{2u}$, which is before A_i^{u+1} has fully arrived at M_{u+1}. However, $\mathsf{A}_{i\rho^2}^u$ is a subset of the first ρ^u columns of A_i^{u+1}. These columns, which start arriving at cycle $(2i-1)\rho^{2u+2}$ and require ρ^{u+1} cycles apiece, finish arriving at cycle $(2i\rho^2-\rho^2+\rho)\rho^{2u}$. Since $\rho > 1$, $\mathsf{A}_{i\rho^2}^u$ is in M_{u+1} before it is scheduled to move down B_u. The remaining submatrices of A_i^{u+1} are not scheduled until A_i^{u+1} has fully arrived, but well before A_i^{u+1} is displaced by B_{i+1}^{u+1}. This verifies requirement (1); requirement (2) is similar.

To verify (3), notice that traffic on bus B_u occurs in epochs of $2\rho^{2u}$ cycles. During epoch i, which goes from cycle $2i\rho^{2u}$ to cycle $2(i+1)\rho^{2u}-1$, A_i^u is transposed into B_i^u (clobbering A_{i-1}^u) by the hierarchy below bus B_u. Meanwhile, submatrix B_{i-1}^u moves up bus B_u during the first half of epoch i, and is replaced by A_{i+1}^u during the second half. So, M_u need only be big enough to hold A_i^u, B_i^u, and *one* of B_{i-1}^u and A_{i+1}^u. Thus, $\alpha = 3$ suffices.

Finally, (4) is verified by observing that bus B_u is used for the upward movement of B_{i-1}^u in the first half of epoch i and for the downward movement A_{i+1}^u during the second half. A half-epoch is exactly enough time to move each submatrix. $\bowtie$

If the columns of the matrices are not aligned on block boundaries, Program 2 will take longer and require bigger modules. There are two sources of performance degradation. First, a subblock of data might span a subproblem boundary. Such a subblock might have to be moved along the bus to the next lower level several times. Unfortunately, the most likely situation is that nearly every subblock at every level above the first will span subproblem boundaries. In this case, a column of a subproblem takes up two subblocks instead of one. If α is 3, thrashing will result and Program 2 will not be communication-efficient. If we assume $\alpha \geq 6$, then communication-efficiency is restored, though the program will run only about half as fast as in the aligned case.

A second source of performance degradation is that N may not be a power of ρ, resulting in some undersized subproblems. For instance, if $N = \rho^{w-1}+1$, there will be four calls to MT_{w-1}. The degradation due to these extra calls is at most a factor of 4, since transposing a partial block is no slower than transposing a full block. In fact, the performance is a bit better, as stated in the following theorem.

Theorem 5.2 *Transposing an $N \times N$ matrix stored in level $w = \lceil \log_\rho N \rceil$ by MT_w of Program 2 takes time at most $6N^2$ on $UMH_{\alpha,\rho,1}$ with $\alpha \geq 6$.*

> **proof sketch:** Consider the movement of level-u blocks on the bus connecting level u to level u+1. A block will be moved at most once for each $\rho^u \times \rho^u$ subproblem of which it is a part. A level-u block can belong to at most three subproblems: the beginning of the block might end a column of one subproblem, the middle can span an entire column of an undersized subproblem (one having fewer than the usual ρ^u elements in its columns), and the end can begin a column of a third subproblem. (It follows from $N > \rho^{w-1} \geq \rho^u$ that there cannot be two adjacent undersized subproblems.) Closer analysis reveals that at most half the blocks can span three distinct subproblems, and the remainder will span at most two[24]. Thus, each bus will be utilized for at most $5N^2$ cycles: $2.5N^2$ for moving blocks of A down, and $2.5N^2$ for moving blocks of B up. The remaining factor of N^2 is more than sufficient to account for the startup and ending latencies of the hierarchical data movement. $\diamond$

Thus, transposing square matrices is communication-efficient. If we adopt the convention that $M \times N$ matrices reside at level $\log_\rho(\max(M, N))$, then Program 2 will also be communication-efficient for non-square matrices.

Our proofs reveal that properly sized and aligned matrix transpose will take time close to $2N^2$, and otherwise it can take approximately $4N^2$ or $5N^2$ cycles. This is a large gap. Is there a better program that doesn't have such a penalty for awkward matrices? The question is rhetorical; the point

[24]Let b be a level-u block spans a column of an undersized subproblem. If $N \geq 2\rho^u$, each column of the original matrix will contain columns from at least two full length subproblems and (at most) one undersized subproblem. Thus, the blocks before and after b don't contain columns from undersized problems. On the other hand, if $N < 2\rho^u$, then both the initial and final elements of b are part of the same subproblem, albeit from different columns, except when the first elements of b contain the last elements of a subproblem.

is, working with the UMH model leads algorithm designers to think about certain issues. These issues are ones that must be addressed by programmers who want to attain peak performance. A model that exposes the problem should facilitate finding a robust and portable solution.

We do not wish to imply that our model captures all those issues that have practical importance. In particular, the limited associativity of some caches and TLB's may be a large concern. There may also be issues involved with getting the addressing code or vector operations performing optimally. Further, the memory module sizes and transfer speeds of a particular machine will differ from the UMH machine in ways that may affect the range of possible solutions.

5.2 A Limit on Communication Efficiency

This subsection shows asymptotic communication efficiency of 1 is not attainable on $UMH_{\alpha,\rho,1}$ for transpose. There is unavoidable latency in getting the operation started up.

Theorem 5.3 *Suppose that $N = \rho^{w}$, that α and ρ are at least 2, and that* A *and* B *are $N \times N$ matrices stored in column-major order at level w of the memory hierarchy and are aligned so that the first element of each begins a level-w block. Any program that choreographs the transpose algorithm on $UMH_{\alpha,\rho,1}$ requires at least $(2 + c)N^2$ cycles, where $c = 1/(6\rho^4\alpha)$.*

proof: A block at some arbitrary level $v < w$ of the result matrix B, being a partial column of B, contains elements from ρ^{v} different columns, and hence blocks, of A. We will argue that before ρ^{v} subblocks of A have been transferred down out of the top level (level w), there's not much useful communication to be done on the bus B_{v}. Specifically, there aren't yet any complete blocks of B to be moved up, and there's not enough room at and below level v to hold much data that is moved down. To show that a constant fraction of the RAM time is wasted, we will focus on a carefully chosen level v that is a few levels down from the top of the hierarchy. The details follow.

Let d be the unique integer such that $\rho^{d-1} \leq 2\alpha < \rho^{d}$. Let $v = w - (d + 2)$. There are two facts we will need to pull out of a hat later:

1. $\rho^{w-1}\rho^{v} \geq N^2/(2\rho^4\alpha)$. This is derived as follows:

$$N^2 = \rho^{2w} = (\rho^{w-1}\rho)(\rho^{v}\rho^{d+2}) = \rho^{w-1}\rho^{v}\rho^{4}\rho^{d-1} \leq \rho^{w-1}\rho^{v}\rho^{4}2\alpha.$$

   Dividing both sides by $2\rho^4\alpha$ gives the desired inequality.

2. $\frac{4}{3}\rho^{2v}\alpha < N^2/(3\rho^4\alpha)$. This is derived as follows:

$$N^2 = \rho^{2w} = \rho^{2v}\rho^{2d+4} = \rho^{2v}\rho^{4}\rho^{2d} > \rho^{2v}\rho^{4}4\alpha^2.$$

   Dividing both sides by $3\rho^4\alpha$ gives the desired inequality.

The result matrix B is partitioned into ρ^{w+d+2} level-v blocks. Each level-v block of B contains elements from ρ^{v} different columns of A, and hence from ρ^{v} different level-$(w-1)$ blocks of A (since each column of A is a distinct level w-block). Consider the state of the computation just before timestep $\rho^{w-1}\rho^{v}$. At most $\rho^{v}-1$ different columns of A can have been involved in the computation so far, since moving a subblock down from level w takes ρ^{w-1} timesteps. Thus, no level-v blocks of B have been completely filled in with their final values. Consequently, in the remaining time, every level-v block of B must be moved up B_{v+1}. Further, almost all of the A matrix must be moved down that same bus. Only the z data items that are currently stored at level v or below are excepted, where $z = \sum_{u=0}^{v}\alpha\rho^{2u} = \alpha\frac{\rho^{2v+2}-1}{\rho^2-1} < \alpha\frac{\rho^2}{\rho^2-1}\rho^{2v} \leq \frac{4}{3}\alpha\rho^{2v}$ such items. Thus, the total time T required to compute B is at least $\rho^{w-1}\rho^{v}$ (the current timestep) plus $N^2 - z$ (to move the remaining data down into level v) plus N^2 (to move all the complete blocks up). Substituting for z, we obtain $T > 2N^2 + \rho^{w-1}\rho^{v} - \frac{4}{3}\alpha\rho^{2v}$. By the two magic formulae given earlier, $T > 2N^2 + N^2/(2\rho^4\alpha) - N^2/(3\rho^4\alpha) = (2 + 1/(6\rho^4\alpha))N^2$. ♣

If α is sufficiently large, then contrary to our convention, the matrices might fit in a level lower than level w. The techniques of this proof can be adapted to show that even if the matrices are stored in the lowest level into which they fit, asymptotic communication efficiency of 1 cannot be achieved. Furthermore, if we hold the transfer cost of B_0 at 1 (so the RAM-complexity remains the same) but reduce all other transfer costs by a constant factor, the communication-efficiency of a transpose program will improve slightly but still be less than 1. The startup latency of transpose can only be eliminated by making the transfer cost function be asymptotically zero.

5.3 Rational permutations

Matrix transposition is an instance of a more general class of permutations, the *rational permutations* of Aggarwal, Chandra and Snir [ACS87]. Let $N = 2^n$ and $I = \{0, 1, ..., N-1\}$. For each i in I, let $[i_{n-1}, ..., i_0]$ be the binary expansion of i. Suppose σ is a permutation on $\langle n-1, ..., 0\rangle$. Then σ induces a permutation R_σ on I defined by $R_\sigma([i_{n-1}, ..., i_0]) = [i_{\sigma(n-1)}, ..., i_{\sigma(0)}]$. Finally, if A and B are arrays over the index set I, then we say that B is obtained from A via the rational permutation R_σ if for all i in I, $\mathsf{A}(i) = \mathsf{B}(R_\sigma(i))$. A program that given A produces B is said to implement the rational permutation induced by σ.

For example, if n is even and $\sigma(i) = i + n/2 \pmod n$, then the array obtained from a $\sqrt{N} \times \sqrt{N}$ matrix via the rational permutation R_σ is the transpose of A. Another example, which will be used in Section 7, is the *bit-reversal permutation* induced by $\sigma(i) = (n-1) - i$.

Theorem 5.4 *Given $N = 2^n$, $\alpha \geq 6$, σ a permutation on $\langle n-1, ..., 0\rangle$, and N-element arrays A and B residing at level $\lceil \log_{\rho^2} N \rceil$, the rational permutation R_σ taking A into B can be performed in at most $6N$ cycles on $UMH_{\alpha,\rho,\mathbf{1}}$.*

proof sketch: The techniques of the transpose algorithm are adapted to perform the rational permutation. Specifically, at an arbitrary level $\nu > 0$, a rational permutation problem on $\rho^{2\nu}$ elements is decomposed into a sequence of rational permutation subproblems on $\rho^{2(\nu-1)}$ elements, and the subproblems are passed down to level ν-1. In the nicely aligned case, the decomposition can be done so that each subproblem and solution fit into exactly $\rho^{\nu-1}$ subblocks. In the unaligned case, a constant multiple more blocks may be required.

We will describe the decomposition for the aligned case; the unaligned case uses the same decomposition, but the data may span more subblocks. In the aligned case, ρ is a power of two and each level-ν subblock holds data from A or B whose indices differ only on the least significant $k = (\nu - 1) \lg \rho$ bits. Let $T_\nu = \{0, 1, ..., k-1\} \cup \{\sigma(0), \sigma(1), ..., \sigma(k-1)\}$. We define a T_ν-set to be a maximal subset of the index set I whose elements differ only in bit positions in T_ν. T_ν has between k and $2k$ distinct elements, depending on σ, and so a T_ν-set has at most 2^{2k} elements. Let S be a subset of A indexed by a T_ν-set. Since T_ν includes the low-order k bits and A is aligned, any level-ν subblock that contains one element of S must lie entirely in S. Similarly, since T_ν includes $\sigma(0), \sigma(1), ..., \sigma(k-1)$, $R_\sigma(\mathrm{S})$ completely fills up any subblock that it intersects. Furthermore, every T_ν-set is the union of a collection of $T_{\nu-1}$-sets, so problems can be recursively decomposed into subproblems. Thus, a T_ν-set plays the same role for a rational permutation that a square submatrix plays in the transpose program. The fact that a T_ν-set may be smaller than $\rho^{2\nu-1}$ elements is not a concern — several T_ν-sets can be bundled together to form a subproblem of size $\rho^{2(\nu-1)}$ elements.

The remaining details are essentially the same as for transposition. $\diamondsuit$

A generalization of the class of rational permutations (also called "bit permute" permutations) is the class of *bit permute with complement* permutations [C92]. Under such a permutation, the i-th input element is permuted to an address determined by rearranging the bits of i and exclusive-oring the result with some constant c. Techniques of the previous proof can be adapted to achieve a communication efficient bit permute with complement permutation on $UMH_{\alpha,\rho,\mathbf{1}}$.

6 Matrix Multiplication

The $O(N^3)$ standard matrix multiplication algorithm depicted in Figure 2 is basic to many scientific subroutine libraries, such as ESSL [IBM86] and LAPACK [ABetc92]. This section shows that $\rho^u/4$ is a threshold function for this algorithm. Thus, communication-efficient implementations exist if and only if the transfer costs are no more than a constant higher than $\rho^u/4$. It also shows, as with matrix transposition, that a communication efficiency of 1 cannot quite be achieved with such transfer cost functions.

6.1 Communication-Efficient Matrix Multiplication

The candidate threshold function for multiplication of $N \times N$ matrices is approximately[25] $\rho^u/4$ because three input matrices must travel down, and

[25]The smallest problem that resides in level u+1 has $N = \rho^u + 1$. The candidate threshold function $f_c(u)$ is exactly $(\rho^u+1)^3$ cycles for a size N instance divided by $4(\rho^u+1)^2$ data items in the instance. This equals $(\rho^u + 1)/4$ cycles per item.

```
MM_{u+1}(A[1:n,1:l], B[1:l,1:m], C[1:n,1:m]):
    REAL VALUE: A, B; VALUE RESULT: C
    INTEGER VALUE: n, m, l
    INTEGER: i_0, i_1, j_0, j_1, k_0, k_1
    FOR i_0 FROM 1 TO n BY $\rho^u$
        i_1 := MIN(i_0+$\rho^u$-1, n)
        FOR j_0 FROM 1 TO m BY $\rho^u$
            j_1 := MIN(j_0+$\rho^u$-1, m)
            FOR k_0 FROM 1 TO l BY $\rho^u$
                k_1 := MIN(k_0+$\rho^u$-1, l)
                MM_u ( A[i_0:i_1, k_0:k_1], B[k_0:k_1, j_0:j_1], C[i_0:i_1, j_0:j_1] )
    END
MM_0 (a, b, c):
    REAL VALUE: a, b; VALUE RESULT: c
    c := c + ab
    END
```

Program 3: Matrix Multiplication.

one result matrix must travel up, the topmost bus in the N^3 time it takes on a *RAM*. This subsection presents a program for standard matrix multiplication with $O(N^3)$ running time on $UMH_{6,\rho,\rho^u/4}$. That, together with lemma 4.1, shows that $\rho^u/4$ is a threshold function.

Program 3, based on techniques going back at least thirty years [RR51], recursively dices the matrix multiplication solid of Section 2 so as be communication efficient on $UMH_{6,\rho,\rho^u/4}$.

Theorem 6.1 *Suppose that $N = \rho^w$ and that $N \times N$ matrices* A, B, *and* C *are aligned so that the first element of each begins a level-w block. Then for any $\alpha \geq 6$,* MM_w(A,B,C) *on $UMH_{6,\rho,\rho^u/4}$ updates* C *by the product of* A *and* B *with communication efficiency $\frac{1}{1+1/\rho^3}$.*

proof: As with the proof of Theorem 5.1, we exhibit a schedule and verify its validity. Let P_0^w be the given problem C := C + A B. Program 3 recursively decomposes problem P_i^{u+1} into ρ^3 subproblems denoted $P_{i\rho^3}^u$ to $P_{i\rho^3+\rho^3-1}^u$. The schedule follows:

- Problem P_i^u moves down B_u into M_u during cycles $(i-\frac{3}{4})\rho^{3u}$ to $i\rho^{3u}-1$. The appropriate submatrix of A is transmitted by column from left to right followed by the submatrices for B, and then C. In the case of $u=0$, this means that one item will be moved up and three down B_0 every cycle.
- Module M_0 performs the multiplications for problem P_i^u during cycles $i\rho^{3u}$ to $(i+1)\rho^{3u}-1$.
- The resulting submatrix of C for problem P_i^u moves up B_u to M_{u+1} during cycles $(i+1)\rho^{3u}$ to $(i+\frac{5}{4})\rho^{3u}-1$.

Time runs from cycle $-\frac{3}{4}\rho^{3w-3}$, when problem P_0^{w-1} begins its downward descent into M_{w-1}, to cycle $\rho^{3w}+\frac{1}{4}\rho^{3w-3}-1$ when the result of problem $P_{\rho^3-1}^{w-1}$ completes its upward accent. Thus, the total time is as required. It is easy to verify that no bus is scheduled for more than one transfer at a time, and that the most 2 problems are resident in a module at any one time and hence $\alpha=6$ suffices.

In order to verify that there is enough time to move down the data for problem P_i^u, notice that P_i^u consists of three $\rho^u\times\rho^u$ submatrices. Since the data is nicely aligned, it comprises $3\rho^u$ level-u blocks. Blocks on bus B_u take $\rho^u/4$ cycles per item and each block contains ρ^u items. Thus, moving the problem down takes $\frac{3}{4}\rho^{3u}$ cycles. Similarly, the results of problem P_i^u travel up B_u in the allotted time.

Finally, we must show that data is available at the specified module when it is scheduled to be moved. The first subproblem $P_{i\rho^3}^{u-1}$ of problem P_i^u begins its descent $\frac{3}{4}\rho^{3u-3}$ cycles before P_i^u has finished arriving at level u. However, this descent begins after all of the A and B submatrices and more than half of the C submatrix have arrived. Thus, a compete subproblem is available. The subsequent subproblems all overlap the multiplications of P_i^u. The upward movement of results of the subproblems proceeds analogously. ♡

Program 3 will be communication efficient on unaligned square matrices. However, this may require $\alpha=12$ (since submatrices may take up twice as many blocks) and the communication efficiency drops, but by a factor of less than 16 (since twice as many blocks are sent per subproblem and $\frac{7}{8}$-ths of the top-level subproblems may be almost empty).

6.2 A Limit on Communication Efficiency

The following theorem shows that the *UMH* complexity any program implementing standard matrix multiplication is bigger than the *RAM* complexity by an additive $O(N^3)$ startup latency even if the transfer cost function is an arbitrary constant factor faster that the candidate function.

Theorem 6.2 *Suppose $N = \rho^{w}$ and A, B, and C are $N \times N$ matrices stored in column-major order at level w of the memory hierarchy and are aligned so that the first element of each begins a level-w block. For any $c > 0$ and any α, matrix multiplication,* $\mathsf{C} := \mathsf{C} + \mathsf{A}\ \mathsf{B}$, *on $UMH_{\alpha,\rho,c\rho^u}$ has communication-efficiency at most $\frac{1}{1+3c^2/4\rho^3}$.*

proof: The transfer cost t_{w-1} on the topmost bus is cN/ρ cycles per item. Bringing down a subblock, which contains N/ρ data items, takes cN^2/ρ^2 cycles. It takes c^2N^3/ρ^3 cycles to move the first cN/ρ subblocks down from the top level. Suppose these subblocks come from a columns of A and b columns of B. Each of these aN/ρ element of the A matrix are multiplied by at most 1 element from each of the b blocks of B since B is in column-major order. Thus, at most abN/ρ multiplications are possible using this data. This expression is maximized when $a = b = cN/2\rho$. Therefore, at most $c^2N^3/4\rho^3$ multiplications can be performed in the first c^2N^3/ρ^3 cycles. Thus, the entire computation requires at least $N^3 + 3c^2N^3/4\rho^3$ cycles on the *UMH*, while taking only N^3 cycles on the corresponding *RAM*. ♠

Notice that the proof holds for the nonupdate form of matrix multiplication ($\mathsf{C} := \mathsf{AB}$). However, it would not go through if B were in row major order. This suggests that it might be possible to compute $\mathsf{C} := \mathsf{AB}^{\mathrm{T}}$ with communication efficiency 1.

7 Fast Fourier Transforms

This section builds on the techniques of previous sections to address a significantly more intricate problem. The forward *Discrete Fourier Transform* (DFT) of a vector x of N complex numbers called *points* is the vector

y of N complex numbers, defined by

$$y_p = \sum_{q=0}^{N-1} \omega_N^{pq} x_q \quad (0 \leq p < N),$$

where ω_N is the N-th root of unity, $e^{-2\pi i/N}$. A *Fast Fourier Transform* (FFT) is a technique for efficiently computing a DFT [V-L92]. Many FFT algorithms make $\log N$ passes through the data. On $UMH_{\alpha,\rho,cu}$, the model of interest, such techniques require moving $\Omega(N \log N)$ data items on the topmost bus, which takes $\Omega(N \log^2 N)$ time. Thus, these algorithms are communication-bound.

Alternatively, if $N = K_1 K_2$, there is a N-point FFT algorithm that consists of K_2 independent FFT's on K_1 points, multiplication of the intermediate results by powers of ω_N affectionately called *twiddle factors*, and then K_1 independent FFT's on K_2 points. An FFT algorithm with RAM complexity $O(N \log N)$ results from recursively choosing K_1 and K_2 within a constant factor of each other. Further, it requires only $O(N)$ data movement on the topmost bus of a UMH. This algorithm is therefore appropriate for our search for a communication-efficient program.

This section will establish that the identity function (in fact, any linear function) is a threshold function for this FFT algorithm. After the standard "Four-Step" program [B90] is shown to be communication-bound for any linear transfer cost function, it will be rechoreographed to be communication-efficient. To simplify presentation, we will only consider the case where $N = 2^{2^r}$ for some integer r, where at each stage of the recursion $K_1 = K_2$ down to $K_1 = K_2 = 2$, and where $\rho = 2$. To clarify explanations, we will leave ρ in symbolic form where possible.

The Four-Step FFT program performs the following steps, treating its input as a $\sqrt{N} \times \sqrt{N}$ matrix stored in column-major order:

1. perform a $\sqrt{N}$-point FFT on each row.

2. multiply the entry in the jth row and kth column by $e^{-2\pi ijk/N}$.

3. transpose the matrix.

4. perform a $\sqrt{N}$-point FFT on each row.

This algorithm takes $O(N \log N)$ time on a RAM.

The input to a problem of size N is assumed to reside in level $\lceil \log_{\rho^2} N \rceil$ on a UMH. Since moving $O(N)$ data down the topmost bus in a communication-efficient program can take at most $O(N \log N)$ time, the candidate threshold function $f_c(\lceil \log_{\rho^2} N \rceil - 1)$ is $O(\log N)$. Therefore, $f_c(u)$ is $O(u)$. However, the next theorem shows that the Four-Step program is communication-bound at the candidate function.

Theorem 7.1 *For any $c > 0$, any α, and any $\rho \geq 2$, the communication efficiency of the Four-Step program on $UMH_{\alpha,\rho,cu}$ is $O(\frac{1}{\log \log N})$.*

> **proof:** The running times of step 2, $T_s(N)$, and of step 3, $T_t(N)$, are at least the cost of moving the data up into the lowest level v into which all the data fit. This level will be some constant k levels below level $w = \lceil \log_{\rho^2} N \rceil$, where k depends on α. Thus, the cost of moving the N data items is $Nc(w - 1 - k) \approx cN(\log_{\rho^2} N - 1 - k)$. This shows $T_t(N)$ and $T_s(N)$ are $\Omega(N \log N)$. Hence we obtain the following recurrence for $T_{fs}(N)$, the UMH complexity of Four-Step program:
>
> $$\begin{aligned} T_{fs}(N) &\geq 2\sqrt{N}\, T_{fs}(\sqrt{N}) + T_s(N) + T_t(N) \\ &\geq 2\sqrt{N}\, T_{fs}(\sqrt{N}) + \Omega(N \log N). \end{aligned}$$
>
> This recurrence implies that $T_{fs}(N)$ is $\Omega(N \log N \log \log N)$. The result follows immediately. $\triangle$

The remainder of this section shows how to rechoreograph the Four-Step program to be communication-efficient on a UMH whose transfer cost function is the identity function. Since the candidate threshold function is $O(u)$, this theorem, together with lemma 4.1, confirms that the identity function is indeed a threshold function.

Theorem 7.2 *There is communication-efficient program for the Fast Fourier Transform on $N = 2^{2^r}$ points aligned at a block boundary at level w ($w = 2^{r-1}$) on $UMH_{\alpha,2,u}$ for any $\alpha \geq 7$.*

> **proof:** In this proof, we will define the bit-reversed FFT (BRFFT), which can be obtained without recursive transposes. Since the bit-reversal permutation is its own inverse, an FFT is obtained from the BRFFT by applying a bit-reversal. Although this permutation is $O(N \log N)$, it is not done

recursively and so only affects the running time by a factor of at most 2 (instead of $\log\log N$). Next, multiplying by twiddle factors is examined. This step is combined with the beginning of the subsequent BRFFT step. Finally, a recursive decomposition of BRFFT in which subproblems fill up entire subblocks is described.

The BRFFT is the FFT followed by a bit-reversal permutation BR, one of the rational permutations described in Section 5.5. The permutation BR applied to a vector can be formed by treating the data as a matrix, transposing the matrix (which swaps the high-order bits with the low-order ones in the vector's addresses), applying a BR to each row of the matrix (which reverses the order of the high-order address bits), and finally applying BR to each column (reversing the lower-order bits). Thus, the BRFFT could be computed by a "Seven-Step" program, namely (1) FFT each row, (2) scale (i.e., multiply by the twiddle factors), (3) transpose, (4) FFT each row, (5) transpose, (6) BR each row, (7) BR each column. However, steps (3) through (5) can be replaced the single step, (3-5) FFT each column. Next, step (6) can be moved ahead of step (3-5), since the columns remain identical (although arranged in a permuted order) when the rows are each bit-reversed. Finally, step (6) can be further moved to be before step (2), assuming the twiddle factors are suitably permuted. Thus, we have a "Five-Step" program for BRFFT, (1) FFT each row (6) BR each row, (2) scale, (3-5) FFT each column (7) BR each column. Now notice that the first two steps are simply a set of BRFFT's, as are the last two. This yields a "Three-Step" BRFFT program, namely (a) BRFFT each row, (b) scale by twiddle factors, and (c) BRFFT each column.

The next refinement will combine the scaling step (b) with the BRFFT's by making use of three subroutines at all levels u for which u is a power of 2.

- $\mathsf{Full}_u(\mathsf{A},\mathsf{S})$ — Given A and S, arrays of ρ^{2u} complex numbers at level u, multiply each element of A by the corresponding element of S and then perform a BRFFT on A. $\mathsf{Full}_u(\mathsf{A},\mathsf{S})$ is implemented as $\mathsf{Rows}_u(\mathsf{A},\mathsf{S})$ followed by $\mathsf{Columns}_u(\mathsf{A},\mathsf{T}_u)$, where T_u is the set of twiddle factors needed for a BRFFT on ρ^{2u} points.
- $\mathsf{Rows}_u(\mathsf{A},\mathsf{S})$ — Treating A and S as two-dimensional $\rho^u \times \rho^u$ matrices, execute $\mathsf{Full}_{u/2}(\mathsf{A}[\mathsf{i},1:\rho^u],\mathsf{S}[\mathsf{i},1:\rho^u])$ for each i from 1 to ρ^u, that is, perform a scaled BRFFT on each row.
- $\mathsf{Columns}_u(\mathsf{A},\mathsf{S})$ — Treating A and S as two-dimensional $\rho^u \times \rho^u$ ma-

trices, execute $\mathsf{Full}_{u/2}(\mathsf{A}[1:\rho^u,\mathsf{j}],\mathsf{S}[1:\rho^u,\mathsf{j}])$ for each j from 1 to ρ^u, that is, perform a scaled BRFFT on each column.

Performing BRFFT on an array A of $N = 2^{2^r}$ points is now simply a matter of calling Full_w (A,J), where J is the all-1 matrix and $w = 2^{r-1}$.

Notice that no data is transferred between levels in order for Full_u to call Rows_u or $\mathsf{Columns}_u$. When $\mathsf{Columns}_u$ in turn calls $\mathsf{Full}_{u/2}$, it passes down one complete block of A and one of S. This data reaches level $u/2$ as square matrices of $\rho^{u/2}$ level-$(u/2)$ blocks. A sequence of such calls can be scheduled using the now-familiar pipeline technique. On the other hand, Rows_u cannot call $\mathsf{Full}_{u/2}$ without reformatting its data, since each call on $\mathsf{Full}_{u/2}$ gets one point from each of ρ^u level-u blocks. To repack this data into $\rho^{u/2}$ level-$(u/2)$ blocks would require moving it down all the way to M_0.

Rather than repacking the data, we will rechoreograph Rows_u to avoid the problem. First, we expand the call to $\mathsf{Full}_{u/2}$ and its calls to $\mathsf{Rows}_{u/2}$ and $\mathsf{Columns}_{u/2}$ inline, obtaining:

```
FOR i FROM 1 TO $\rho^u$
    FOR j FROM 1 TO $\rho^{u/2}$
        $\mathsf{Full}_{u/4}$(A[i,j,1:$\rho^{u/2}$], S[i,j,1:$\rho^{u/2}$])
    FOR j FROM 1 TO $\rho^{u/2}$
        $\mathsf{Full}_{u/4}$(A[i,1:$\rho^{u/2}$,j], $\mathsf{T}_{u/2}$[1:$\rho^{u/2}$,j])
```

In the above, A and S are treated as $\rho^u \times \rho^{u/2} \times \rho^{u/2}$ arrays. Next, we distribute the outer loop across the two inner loops and interchange the loop order of both nested loops. It can be verified that these program transformations do not change the semantics of the program. The result is:

```
FOR j FROM 1 TO $\rho^{u/2}$
    FOR i FROM 1 TO $\rho^u$
        $\mathsf{Full}_{u/4}$(A[i,j,1:$\rho^{u/2}$], S[i,j,1:$\rho^{u/2}$])
FOR j FROM 1 TO $\rho^{u/2}$
    FOR i FROM 1 TO $\rho^u$
        $\mathsf{Full}_{u/4}$(A[i,1:$\rho^{u/2}$,j], $\mathsf{T}_{u/2}$[1:$\rho^{u/2}$,j])
```

Now block (stripmine) each inner loops into two nested loops, and treat A and S as $\rho^{u/2} \times \rho^{u/2} \times \rho^{u/2} \times \rho^{u/2}$ arrays, yielding:

```
FOR j FROM 1 TO $\rho^{u/2}$
    FOR i$_2$ FROM 1 TO $\rho^{u/2}$
        FOR i$_1$ FROM 1 TO $\rho^{u/2}$
            Full$_{u/4}$(A[i$_1$,i$_2$,j,1:$\rho^{u/2}$], S[i$_1$,i$_2$,j,1:$\rho^{u/2}$])
FOR j FROM 1 TO $\rho^{u/2}$
    FOR i$_2$ FROM 1 TO $\rho^{u/2}$
        FOR i$_1$ FROM 1 TO $\rho^{u/2}$
            Full$_{u/4}$(A[i$_1$,i$_2$,1:$\rho^{u/2}$,j], T$_{u/2}$[1:$\rho^{u/2}$,j])
```

Finally, the two innermost loops can now be written as calls to $\mathsf{Rows}_{u/2}$ as follows:

```
FOR j FROM 1 TO $\rho^{u/2}$
    FOR i$_2$ FROM 1 TO $\rho^{u/2}$
        Rows$_{u/2}$(A[1:$\rho^{u/2}$,i$_2$,j,1:$\rho^{u/2}$], S[1:$\rho^{u/2}$,i$_2$,j,1:$\rho^{u/2}$])
FOR j FROM 1 TO $\rho^{u/2}$
    FOR i$_2$ FROM 1 TO $\rho^{u/2}$
        Rows$_{u/2}$(A[1:$\rho^{u/2}$,i$_2$,1:$\rho^{u/2}$,j], T$_{u/2}$[1:$\rho^{u/2}$,j]$^{\mathsf{R}}$)
```

where the superscript R indicates that the vector $\mathsf{T}_{u/2}[1{:}\rho^{u/2},j]$ is to be repeated $\rho^{u/2}$ times.

The arguments of the calls to $\mathsf{Rows}_{u/2}$ are subarrays comprising $\rho^{u/2}$ level-$(u/2)$ blocks. Furthermore, the innermost loops for Rows_u (the loops indexed by i_2) make a sequence of calls to $\mathsf{Rows}_{u/2}$ using data that is arranged sequentially in level-u blocks. To start up this sequence of calls, we need only move down one subblocks from each of $2\rho^{u/2}$ level u blocks, then one subsubblock of each of these, and so on, until the data for the first call reaches level $u/2$. Thereafter, additional subproblems can be pipelined down (and the results moved up) at a rate of one subproblem every $3\rho^u(u-1)$ cycles, the time to move $3\rho^u$ data items along the slowest bus (the one just below level u).

Choose constant $c \geq 3$ sufficiently large so that the base cases Rows_1 and $\mathsf{Columns}_1$ can be executed in $cN\lg N$ cycles (where $N = \rho^2 = 4$). We will show by induction that a call to Rows_u on $N = \rho^{2u}$ data points, where u is a power of 2, requires $cN\lg N$ cycles in M_0. Rows_u entails a sequence of $2N^{\frac{1}{2}}$ calls to $\mathsf{Rows}_{u/2}$. By the above discussion, the data can be choreographed to arrive at level $u/2$ at a rate of up to one problem every $3\rho^u(u-1) < c(N^{\frac{1}{2}})\lg(N^{\frac{1}{2}})$ cycles. Making the inductive assumption that each call to

$\mathsf{Rows}_{u/2}$ requires $c(N^{\frac{1}{2}})\lg(N^{\frac{1}{2}})$ cycles at M_0, we see that executing Rows_u requires $2N^{\frac{1}{2}}c(N^{\frac{1}{2}})\lg(N^{\frac{1}{2}}) = cN\lg N$ cycles in M_0. Another induction shows that $\mathsf{Columns}_u$ takes $cN\lg N$ cycles and Full_u takes $2cN\lg N$ cycles in M_0. Using the pipeline techniques of earlier proofs, once problems start arriving at M_0, it remains active until the computation is complete.

Finally, we argue that the startup latency is quite small. An $O(N^{\frac{1}{2}}\log N)$ preprocessing step brings all the twiddle factors to the appropriate levels (T_u goes to level u). The time to bring the first data items to M_0 — essentially the time to move $N^{\frac{1}{2}}$ blocks down the topmost bus plus the time to move one block down each subsequent bus — is $O(N^{\frac{3}{4}}\log N)$. ∇

It is not at all clear this program would be competitive on real computers. Some authors [GJ87, B90] try to avoid bit-reversal permutations like the plague; in practice, a single transpose is more efficient than a bit-reversal. The penalty of recursive transposes is not large — $O(\log\log N)$. Bailey [B90] reports good results for the Four-Step program for N from 2^8 to 2^{20} on Cray supercomputers[26]. For problems of this size it is not necessary to invoke the transpose recursively; simultaneous FFTs of the rows are performed by the Cray library routine.

It is probably possible to improve upon the communication-efficiency of the program developed above. One line of improvement would try to reduce the cost of moving the twiddle factors. Another would return to the Four-Step program and try to incorporate the recursive transposes into the FFT steps.

8 Related Work

The UMH model is not the first to attempt to capture the cost of moving data within the memory hierarchy. Numerous papers (for example [F72, HK81, AV88, LTT89]) consider a two-level memory hierarchy. The fact that the UMH model has more than two levels has several consequences. First, it is more natural for asymptotic analysis of algorithms, since a single

[26]Carlson [C90a] reports improvement obtained by carefully exploiting the local memory attached to each processor of the Cray-2. Section 9 discusses modeling such computers.

machine can handle problems of all sizes. More significantly, the UMH focuses attention on the intermediate levels of storage. Thus, it raises questions such as what "shape" subproblems are suitable for further subdividing, and how should the reception of a problem be coordinated with the dispatching of subproblems.

The models of Vitter and Shriver [VS90] focus on an orthogonal aspect of some memories — particularly disk storage — that simultaneous data transfers may be possible between separate memory modules and a single lower module. It might be possible to incorporate this feature into the UMH model.

Our work is closely related to, and heavily influenced by, the Hierarchical Memory Model (HMM) [AACS87] and the Block Transfer model (BT) [ACS87], both of which have multiple levels. This section explores the relationship between the UMH model and the HMM and BT models. Each model is a family of machines parameterized by some function that determines the cost of accessing data. An $HMM_{f(x)}$ is a RAM machine that is charged $f(a)$ to reference the a-th location in memory. A $BT_{f(x)}$ is charged $f(a)$ to access address a but a block of length l ending at a may be moved at cost $f(a)+l$. Very roughly, the cost of touching each element in a block of length l ending at address a can be approximated as $lf(a)$ on an $HMM_{f(x)}$, as $\max(l, f(a))$ on a $BT_{f(x)}$, and as $\max(l, \sqrt{a})f(\log_{\rho^2} a)$ on a $UMH_{\alpha,\rho,f(x)}$[27]. Figure 5 directly compares access costs of different size blocks in various UMH, HMM, and BT models. High-bandwidth (low transfer cost) models $UMH_{1,2,u}$, $HMM_{\log x}$, and $BT_{\sqrt{x}}$ are shown on the left while low-bandwidth models $UMH_{1,2,\rho^u}$, $HMM_{\sqrt{x}}$, and $BT_{x^{1.5}}$ are on the right[28]. We chose to compare these models because they are, in some sense, "threshold models" for FFT (on the left) and matrix multiplication (on the right). Substantially more points have been plotted for the UMH's to convey a sense of the discon-

[27]Addresses are assigned to the locations in a UMH linearly from the lowest module M_0 up. Comparing the UMH to these models is complicated by the fact that the transfer cost is a function of level number rather than address. It can be shown that the transfer cost of moving a value at address a down one level on a $UMH_{\alpha,\rho,f(u)}$ is bounded between $f(\log_{\rho^2} \frac{a}{\alpha} - 1)$ and $f(\log_{\rho^2} \frac{a}{\alpha} + 1)$ cycles per item provided $f(x)$ is monotonic and a is not very small. Thus in this case, it is legitimate to compare apples (access cost functions of the HMM and BT models) and oranges (transfer cost functions of the UMH model).

[28]Cost for another high bandwidth model, the unit bandwidth UMH of Section 5, could have been plotted on the left as well. The values for this model would lie below the values for the $UMH_{1,2,u}$ by a small amount.

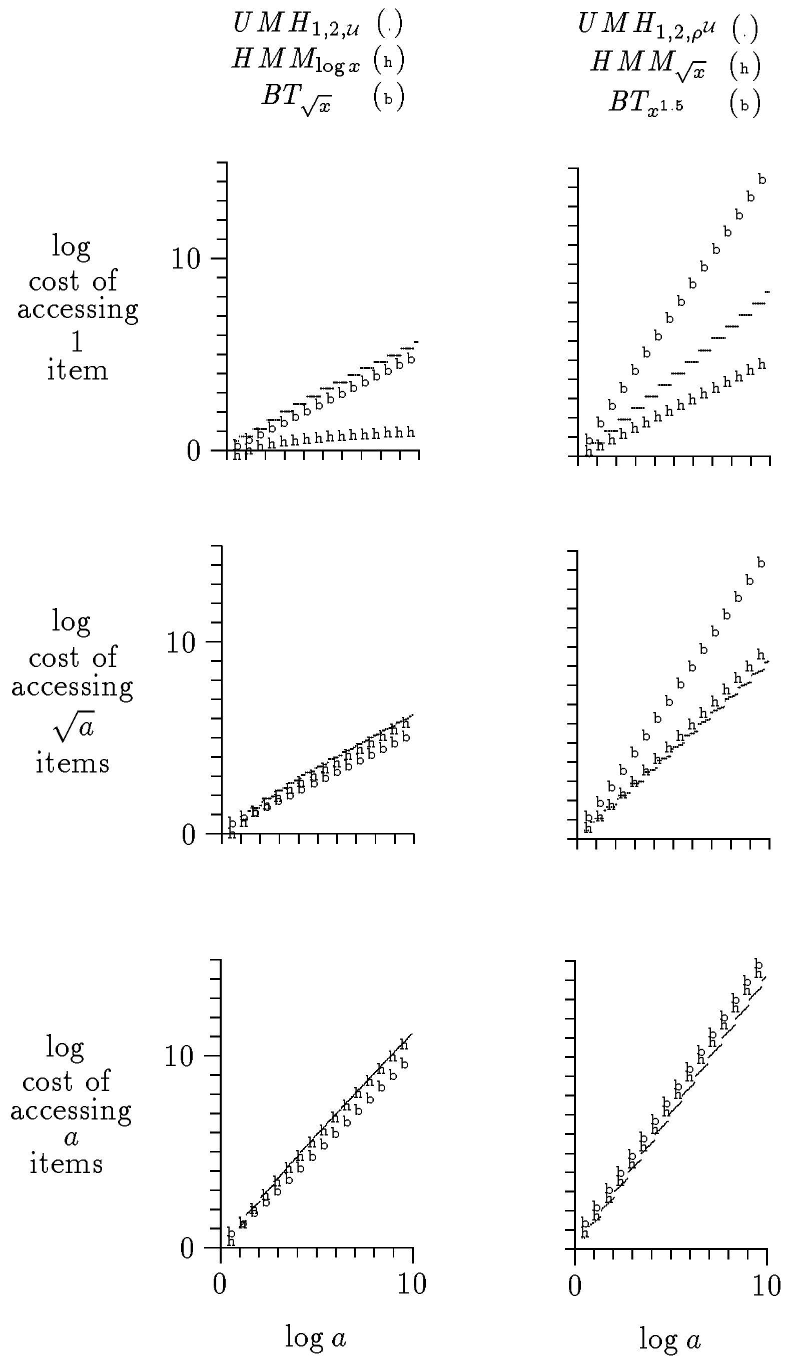


Figure 5: Gross Comparison of Similar UMH, HMM, and BT Models.

tinuities associated with crossing level boundries. The top graphs compare the cost of accessing a single data item at address a. The middle graphs compare the cost of accessing $\sqrt{a}$ items beginning at address a. And, the bottom graphs compare the costs of accessing a items beginning at address a.

These graphs show how similar the compared models really are when block sizes are large. The $HMM_{\log x}$ probably underestimates, and the $BT_{x^{1.5}}$ undoubtably overestimates, the cost of accessing a single data item. (After all, you can't bring a bit in from disk without bringing a page into main memory, but only a single page containing the bit gets brought in.) There are also more subtle, but significant, differences between the models. The HMM model provides no incentive for spacial locality whatsoever. The HMM and BT models allow only one transfer at a time, while the UMH model allows separate blocks to be transfered simultaneously on different buses. The BT model is more flexible with respect to the sizes of blocks to be transfered. Latency in the BT model and transfer cost in the UMH model play complementary roles; each models both the latency and bandwidth of a real computer with one (functional) parameter. The UMH model is more flexible with respect to bandwidth (sufficiently large blocks always have nearly unit bandwidth on the BT model).

The larger the communication cost function is on any model, the larger the complexity of an algorithm. For very large cost functions, the complexity is usually dominated by the time to touch all the input values. For the lowest communication cost functions, the complexity of the algorithm studied is a linear or nearly linear function of the RAM time. In between, there are communication cost functions for which the computation and communication are roughly in balance; this is the case for the threshold functions on the UMH. Similar cusps occur for the HMM and BT models. The HMM or BT complexities at the cusp are a factor of $\log N$ or $\log\log N$ greater than either the time to touch the input or the computation time. In the UMH, this factor is only a constant. This difference is primarily due to the ability of a UMH to use its buses simultaneously.

Transpose requires very low communication costs to approach the RAM complexity. Transposing a $N \times N$ matrix takes $\Theta(N^2)$ time on a $UMH\alpha, \rho, \mathbf{1}$ model. It takes $\Theta(N^2 \log\log N)$ time in the $BT_{\sqrt{x}}$ model [ACS87], and

$\Theta(N^2 \log N)$ time in the $HMM_{\log x}$ model[29] [AACS87]. The $BT_{\sqrt{x}}$ achieves its bound by moving each data item $O(\log\log N)$ time at constant cost per item. In contrast, the $UMH_{\alpha,\rho,1}$ moves each data item twice on each of $\log_{\rho^2} N$ buses. It achieves its bound by overlapping these transfers. Whether $\Theta(N^2)$ or $\Theta(N^2 \log\log N)$ is a more realistic complexity for transpose probably depends on whether data movement on the different buses of the machine in question can be overlapped effectively.

Matrix multiplication has a high computation-to-data ratio. All three models achieve $\Theta(N^3)$ even with high communication costs; all three use essentially the same order of computation to do so. The UMH and HMM programs use similar data movement strategies. The BT model achieves $\Theta(N^3)$ even on machines with extremely high access costs by using a somewhat unnatural data movement strategy. When decomposing a problem of size N into subproblems, the natural unit of transfer is the column length of the subproblem — $O(N)$. The BT program transfers data in units that are much larger — $O(N^2/\log\log N)$ — in order to take advantage of its unit bandwidth assumption. In our experience, the strategy of the UMH and HMM programs is closer to that used to achieve high performance in practice.

Essentially the same Four-Step FFT program is analyzed in each model. This program has a $\Theta(N \log N \log\log N)$ running-time on both the $UMH_{\alpha,4,u}$ and the $HMM_{\log x}$ models. It has complexity $\Theta(N \log N)$ on the $BT_{\sqrt{x}}$ model. For the HMM and BT models, these complexities are optimal; hence, there is no motivation to find a better program. However, in the UMH model, the program can be rechoreographed to achieve an asymptotic speedup.

9 Parallelism

This section generalizes the MH and UMH models to handle parallelism and establishes threshold functions for matrix multiplication on a range of processors. A module of the *Parallel Memory Hierarchy* (PMH) model can be connected to more than one module at the level beneath it in the hierarchy,

[29]Floyd's two-level memory model [F72] also requires $\Theta(N^2 \log N)$ time because the lower level of this model only holds a constant number (3) of size N blocks.

giving rise to a tree of modules with processors at the leaves[30].

A vast collection of different architectures for machines with more than one processor currently exists. The memory and communication structure of such parallel and distributed machines can often be modelled with a tree structure. For instance, the root module can represent the global memory shared by a collection of processors, and smaller modules attached below it can represent the local caches of the individual processors. The non-uniform communication costs of various interconnection topologies can be modelled, though somewhat roughly, by using several levels of a PMH. For instance, a mesh of 64 processors could be represented by a root module (representing the total semiconductor memory of the machine) with four child modules (one for each quadrant of the mesh), each parenting 4 subquadrant modules, each of which has 4 processor modules as leaves. Further study is needed to determine how to choose blocksizes and latencies to best reflect the actual communication capabilities of the mesh.

Different classes of architectures are distinguished by how much branching there is at each level. This is illustrated in Figure 6. The left-hand figure was drawn[31] using parameters representative of traditional supercomputers, such as Cray's C90, Fujitsu's VP2600, and NEC's SX-3. Such machines have their branching near the leaves of the memory hierarchy tree. The middle figure is representative of scalable multicomputers and multiprocessors that have their semiconductor memory physically partitioned among the processors. Such machines, which include Kendall Square Research's KSR 1, Thinking Machine's CM5, and hypercube machines, have their branching in the middle of the memory hierarchy. The final figure of a cluster of workstations shows branching near the root. Thus, the "high-end" versus "low-end" spectrum is reflected by the location of branching in the PMH model.

To complete a PMH model, one must specify the communication mode between a module and its children. Clearly, the items being communicated

[30]A related extension allows multiple modules to be attached *above* a module. This corresponds to a system with a single processor and a tree of memory modules growing out of it. This has proved a useful tool for analyzing the behavior of multiple disks [VS90], and might also be useful in understanding multiple banks of semiconductor memory.

[31]As before, the dimensions of the rectangles are the logarithms of the blockcount and blocksize. Additionally, the number P of modules at a given level is depicted by $1+\lg(P)$ rectangles. Thus, for instance, the middle figure represents 256 processors arranged in 8 clusters.

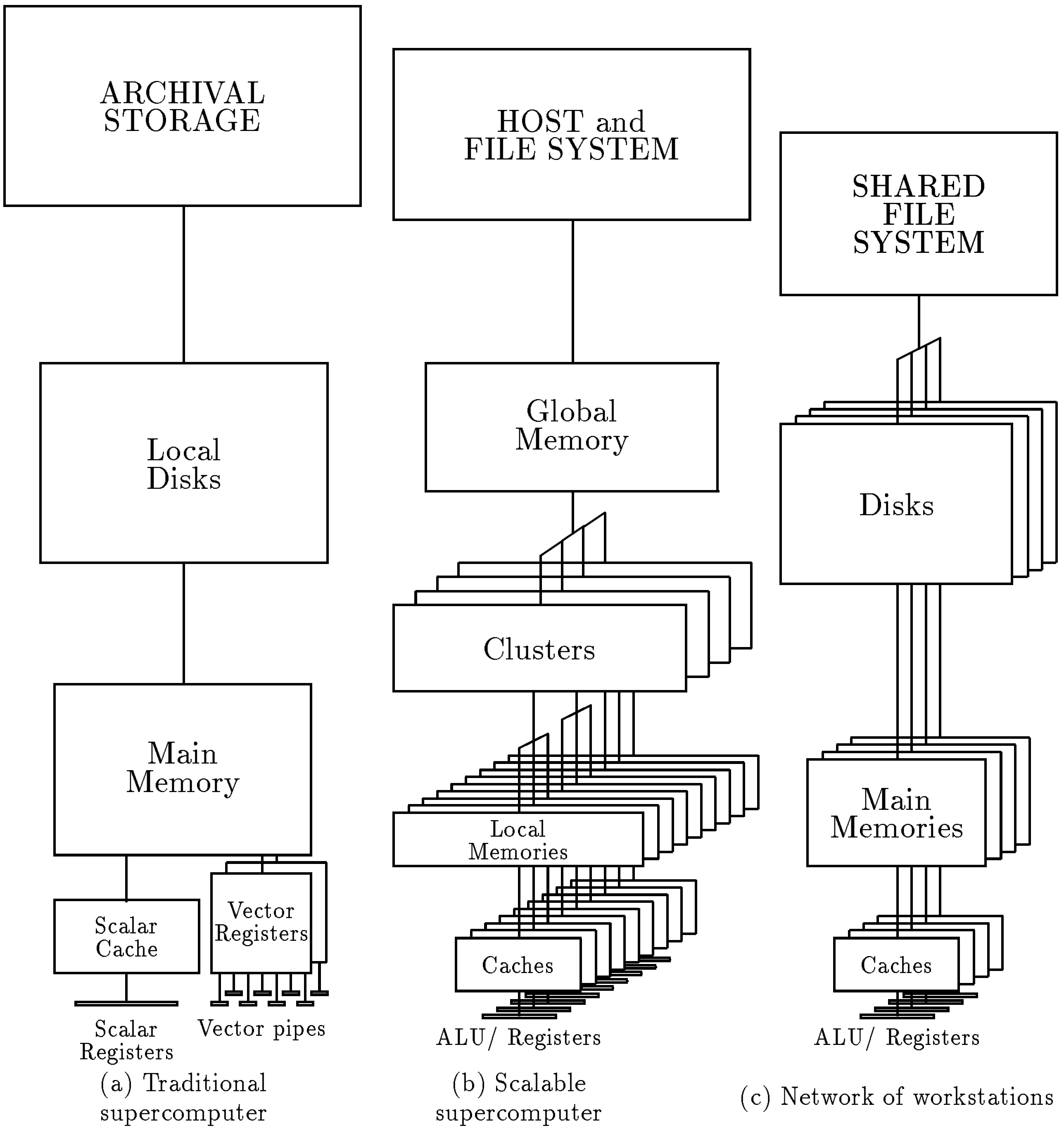


Figure 6: Some Parallel Memory Hierarchies.

will be subblocks of the parent (or, equivalently, blocks of the children). One might postulate a single logical bus connecting parent with children and allow either point-to-point or broadcast communication on it. We do not explore this alternative. The other alternative (a distinct bus connects each child to the parent) can be further refined based on the kind of simultaneous access permitted to a given location of a subblock in the parent. CRCW (Concurrent Read, Concurrent Write), CREW ("E" representing "Exclusive"), and EREW are possibilities. The results of this section hold for all three choices.

A *Uniform Parallel Memory Hierarchy*, $UPMH_{\alpha,\rho,f(u),\tau}$, is a PMH forming a uniform τ-ary tree of $\langle \rho^u, \alpha\rho^u, \rho^u f(u)\rangle$ memory modules[32]. The *communication efficiency* of a $UPMH$ program is the ratio of its $PRAM$ complexity to its $UPMH$ complexity.

Theorem 9.1 *If there is a communication-efficient program for multiplying* $N \times N$ *matrices at level* $w = \lceil \log_\rho N \rceil$ *on* $UPMH_{\alpha,\rho,f(u),\tau}$ *with* τ^w *processors, then* $f(u)$ *is* $O((\frac{\rho}{\tau})^u)$.

> **proof:** The $PRAM$ time for matrix multiplication is $\frac{N^3}{\tau^w}$. The $UPMH$ time of a communication-efficient program can be at most c times larger for some constant $c > 0$. Consider the communication cost. Since all of each input matrix is used in the computation, there must be some bus out of the root that has at least $\frac{N^2}{\tau}$ data items pass over it. Therefore, the item transfer time of this bus is at most $\frac{cN^3}{\tau^w}/\frac{N^2}{\tau} = \frac{cN}{\tau^{w-1}} = c\tau(\frac{\rho}{\tau})^w$ cycles. $\surd$

A corollary follows from observing that if $\rho < \tau$, then $(\frac{\rho}{\tau})^u$ approaches 0 asymptotically.

Corollary 9.2 *On a* $UPMH_{\alpha,\rho,\mathbf{c},\tau}$ *with constant transfer cost c and with* $\rho < \tau$ *(i.e., with more than N processors), multiplication of* $N \times N$ *matrices will be communication-bound.*

The following theorem shows that a program closely related to one described by Valiant [V90] achieves communication-efficiency for transfer cost functions not proscribed by the theorem above.

[32]The uniform parallel model is not realistic of today's computers. Typically, these have branching at only a few levels, and the sizes of the modules and the transfer costs are far from uniform.

Theorem 9.3 *Multiplication of nicely aligned $\rho^{w} \times \rho^{w}$ matrices at level w of $UPMH_{9,\rho,(\frac{\rho}{\tau})^{u}/4,\tau}$ can be computed with communication efficiency $1/(1+\frac{2\tau}{\rho^3})$ for $1 \leq \tau \leq \rho^2$ provided τ divides ρ^2 evenly.*

proof: The goal is to use τ^{w} processors and $(1+\frac{2\tau}{\rho^3})\rho^{3w}/\tau^{w}$ cycles to perform $\mathsf{C} = \mathsf{C} + \mathsf{AB}$. Each processor will compute $(\frac{\rho^2}{\tau})^{w}$ values of C. As with the sequential multiplication program of Section 6.1, a problem in a module is partitioned into ρ^3 subproblems, each being a $\rho^{w-1} \times \rho^{w-1}$ subcube of the Matrix Multiplication Solid of Figure 2. Each subproblem is transferred down one level and further partitioned into ρ^3 subsubproblems, and so on. In the parallel case, $\frac{\rho^3}{\tau}$ subproblems are pipelined through each of the τ buses to smaller modules. Since τ divides ρ^2, each bus handles a multiple of ρ subproblems, and so the ρ subcubes with a given submatrix of the result matrix C can all be handled by same module one level down.

Time on a level u bus is partitioned into epochs. The duration of an epoch is the time required to perform all the multiply-adds of a level u problem — $(\frac{\rho^3}{\tau})^{u}$ cycles. During the first three quarters of an epoch, the input to a level u problem (submatrices of A, B, and C) travel down the bus; during the final quarter the solution to a previous problem moves up. Notice that the time to transfer a $\rho^{u} \times \rho^{u}$ matrix on a level u bus is exactly $(\frac{\rho^3}{\tau})^{u}/4$, that is, one-quarter epoch.

The schedule of Theorem 6.1 is very aggressive; a problem occupies a module for only two epochs, necessitating that a subproblem be dispatched before the problem has fully arrived. A less aggressive schedule is easier to find in the parallel case. The first τ subproblems of the problem P arriving at level u in epoch i are dispatched at the beginning of the fourth quarter of i (immediately *after* the entire problem has been received). Inductively, it can be shown that processing P after it arrives at level u requires a full level u epoch plus two subepochs (that is, epochs at level u-1) of $(\frac{\rho^3}{\tau})^{u-1}$ cycles. There are ρ^3/τ subepochs in an epoch. Since $\rho \geq 2$ and $\rho^2 \geq \tau$, there are at least two subepochs in an epoch. Thus, the result matrix of P can be transfered up from level u during the fourth quarter of epoch $i+2$. Notice that problem P occupies the module for exactly three epochs. Thus, $\alpha = 9$ (that is, room for three problems in each module) suffices.

At the processor level, the duration of an epoch is a single cycle. As soon as data starts arriving at a processor, the processor is kept busy until it finishes its final multiply-add, $(\frac{\rho^3}{\tau})^{w} + 2$ cycles later. At level w-1, the entire computation takes $\frac{\rho^3}{\tau}+2$ epochs of $(\frac{\rho^3}{\tau})^{w-1}$ cycles each. Dividing the *PRAM* time ρ^{3w}/τ^w by $(\frac{\rho^3}{\tau}+2)(\frac{\rho^3}{\tau})^{w-1}$, gives the desired communication efficiency.

All that remains is to consider the complications posed by restrictive communication modes. Since all the modules at a given level compute different subblocks of the C matrix, "exclusive write" does not pose a problem. The same submatrix of A (and similarly, of B), might travel down ρ buses during the same epoch since each submatrix is involved in ρ subproblems. However, each submatrix has at least ρ columns. By staggering the order in which columns are read, any restrictions on "concurrent reads" may be finessed. ∞

As with sequential matrix multiplication, the unaligned case can be made communication-efficient by increasing α. The communication efficiency will be reduced since a column of a submatrix may span two subblocks, and further reduced in the exclusive read or exclusive write cases since a subblock may be shared by up to three submatrices. Still, these considerations only affect the running time by a constant factor. Similarly, the case where τ does not divide ρ^2 can be handled by increasing both α and the length of the pipeline at each module; the uneven load entailed by splitting up each problem over τ buses can be evened out over τ problems. Thus, $(\frac{\rho}{\tau})^u$ is a threshold function for any τ between 1 and ρ^2.

10 Conclusion

The purpose of this paper is both to present a model of computation that captures memory architecture and to show that the model is useful for designing high-performance programs. Many performance-tuning problems that arise after the algorithm and data structures have been chosen come down to data movement problems. The *UMH* model is a tool for quantifying the efficiency of data movement, thus providing a bridge between algorithm analysis and performance programming practice.

The Uniform Memory Hierarchy model is an abstraction of the Memory Hierarchy model, which in turn grew out of work on a high-performance implementation of LAPACK for the RS/6000. Both models reflect characteristics of the various levels within semiconductor memory — registers, cache, address translation mechanisms, main memory, and semiconductor backing store. They may also prove useful in modeling magnetic storage devices and communication channels between processors. The UMH model is intended to focus attention on aspects of algorithm and program design that are relevant to a wide class of machines.

Analyses are give for several problems that have been studied using other models [F72, HK81, ACS87, AACS87, LTT89] that consider the cost of moving data — matrix transpose and other rational permutations, matrix multiplication, and Fast Fourier Transforms — to facilitate comparison of results. We leave open the question of how to extend our model for nonoblivious problems such as sorting.

The analyses in this paper proceed in a similar fashion. A model is chosen for which the time to move just the input and output data on the topmost bus is equal to the computation time. A pipelined program is exhibited that is communication-efficient on that model. For matrix transpose and multiplication, explicit "two-epoch" pipelines were presented that have very tight timing constraints. For parallel matrix multiplication, a looser three-epoch pipeline was used. The advantage of the shorter pipeline is smaller memory requirements (as measured by the minimum α needed) and a slightly reduced startup latency. The paper establishes threshold functions that separate transfer cost functions for which each algorithm is communication-efficient from those for which it is communication-bound. The constant function $\mathbf{1}$ is a threshold function for matrix transpose, the identity function is one for FFT, and ρ^u is for matrix multiplication. A threshold function for parallel matrix multiplication on a $UPMH$ is $(\frac{\rho}{\tau})^u$.

UMH analysis is geared toward determining, and improving, constant factors. This activity is the theoretical analog of tuning a program for peak performance. We have presented matrix transpose and multiplication programs with communication efficiencies almost, but not quite, 1 for nicely aligned matrices. We also have shown that no program for matrix transpose or multiplication can achieve communication efficiency of 1 on models with transfer costs given by their threshold functions.

A novel feature of the MH and UMH models is that buses can be active simultaneously. Hence, one seeks a program in which the communication cost along each bus is dominated by the computation rather than one that reduces the sum of the communication costs along all buses. In complexity analyses, this can make an asymptotic improvement. An interesting line of research would be to investigate what combination of architectural, language, operating-system and compiler features would facilitate a higher degree of overlapped or pipelined data movement than is now possible.

We believe we are justified in inflicting yet another model of computation on the Computer Science community. The UMH model can express the kind of tight control over data movement that is necessary for achieving near-peak performance on important computation kernels. Designers of both software and hardware currently try to present the programmers with the illusion of a RAM. There is no question that the RAM paradigm makes achieving a moderate level of performance easier, but it is worth questioning whether the RAM model is helpful for attaining high performance [C90b]. Performance programmers spend days rewriting inner loops to trick their compiler's register allocator, analyzing how 2-way or 4-way associative caches will behave on certain programs, and learning the messy details of disk behavior and communication implementations. These burdens are imposed because there is no direct way to dictate which blocks should be moved into registers, cache or main memory. It is our hope that programming tools will be designed to support the illusion of a MH or UMH (and for parallel computers, a PMH), and that performance programming will then be easier since a single mechanism (the memory module) will suffice for all the levels of memory and communication. We emphasize that such tools should supplement, not replace, the standard tools.

Secondly, UMH analysis suggests principles for computer architecture and system design. For example, we have seen that having an aspect ratio of at least 8 or so has made designing certain programs simpler. This provides fresh insight into the question of how large cachelines and pages should be. Similarly, knowing the threshold function for various algorithms suggest how much bandwidth is needed to support efficient implementations of those algorithms. We can also quantify how much speed is gained by allowing communication along the various buses to overlap. Ultimately, just as computer hardware and software have evolved to support the RAM illusion, we hope they will evolve towards the UMH and $UPMH$ models.

A third justification pertains to parallelism. There is a recognized need for a "type architecture" [S86] or "bridging model" [V90] that makes a closer connection between parallel algorithm design and actual multiprocessors. When looking for such a model, people often start with the assumption that the RAM model is an unqualified success for sequential computing, and thus is a good starting point. But peak performance is of paramount importance in parallel processing. Thus, the UMH might be a better beginning. In the examples of this paper, the effort required to choreograph a program for a UMH model is virtually the same as needed for parallelization.

A final justification is our hope that the models introduced in this paper have pedagogic value. Caches, backing store and virtual memory management are usually considered to be so messy that they should only be mentioned in architecture or operating system courses. Similarly, performance programming is currently regarded as something of a black art. We hope that the MH and UMH models, and the visualizations presented in this paper, will suggest ways that these subjects can be integrated into in the Computer Science curriculum.

Acknowledgements

Many people have contributed to the content and exposition of this paper. They include Ramesh Agarwal, Robin Alpern, Len Berman, Alan Borodin, Ashok Chandra, Jeanne Ferrante, Bhaskar Ghosh, Matt Grayson, Fred Gustavson, Dan Prener, Gene Ressler, Larry Rudolph, Larry Ruzzo, Rafael Saavedra-Barrera, Karl Seger, Elizabeth Shriver, Jeff Vitter, and the referees.

References

[AG89] Agarwal, R. C. and F. G. Gustavson, "Vector and Parallel Algorithms for Cholesky Factorization on IBM 3090," *Proc. Supercomputing '89*, November, 1989.

[AACS87] Aggarwal, A., B. Alpern, A. K. Chandra, and M. Snir, "A Model for Hierarchical Memory," *Proc. 19th Symp. on Theory of Comp.*, May 1987, pp. 305-314.

[ACS87] Aggarwal, A., A. K. Chandra, and M. Snir, "Hierarchical Memory with Block Transfer," *Proc 28th Symp. on Foundations of Comp. Sci.*, October 1987, pp. 204-216.

[AV88] Aggarwal, A. and J. Vitter, "IO Complexity of Sorting and Related Problems," *CACM*, September 1988, pp. 305-314.

[AHU74] Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

[ACS90] Alpern, B., L. Carter, and T. Selker, "Visualizing Computer Memory Architectures," *IEEE Visualization '90 Conference*, October 1990.

[ABetc92] Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A.McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.

[B90] Bailey, D. H., "FFTs in External or Hierarchical Memory," *The Journal of Supercomputing*, v. 4, pp. 23-35, 1990.

[BW90] Bakoglu, H. B. and T. Whiteside, "RISC System/6000 Hardware Overview," *IBM RISC System/6000 Technology*, IBM Corp. SA23-2619, pp. 8-15 (1990).

[C90a] Carlson, D. A., "Using Local Memory to Boost the Performance of FFT Algorithms on the CRAY-2 Supercomputer," *The Journal of Supercomputing*, v. 4, pp. 345-356, 1990.

[CK89] Carr, S. and K. Kennedy, "Blocking Linear Algebra Codes for Memory Hierarchies," *Fourth SIAM Conference on Parallel Processing for Scientific Computing*, December 1989.

[C90b] Carter, L., "The RAM Model and the Performance Programmer," IBM Research Report RC16319, November, 1990.

[CSB86] Cheriton, D. R., G. A. Slavenburg, and P. D. Boyle, "Software-Controlled Caches in the VMP Multiprocessor," *International Symposium on Computer Architecture*, June, 1986, pp. 366-374.

[C92] Corman, T. H., "Fast Permuting on Disk Arrays," *Advanced Research in VLSI: Proceedings of the 1992 Brown/MIT Conference*, 1992, pp. 58-76.

[DDDH88] Dongarra, J., J. Du Croz, I. Duff, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," AERE R 13297, Harwell Laboratory, Oxon, October 1988.

[F72] Floyd, R. W., "Permuting Information in Idealized Two-Level Storage," *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 105-109.

[FOW87] Ferrante, J., K. Ottenstein and J. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. on Programming Languages and Systems*, Vol. 9, No. 3, July, 1987, pp. 319-349.

[GJMS88] Gallivan, K., W. Jalby, U. Meier, and A. H. Sameh, "Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design," *Int. J Supercomputer Appl.*, Vol. 2, No. 1, Spring 1988, pp. 12-48.

[GJ87] Gannon, D., and W. Jalby, "The Influence of Memory Hierarchy on Algorithm Organization: Programming FFTs on a Vector Multiprocessor," *The Characteristics of Parallel Algorithms*, Jamieson, Gannon, and Douglass, ed., MIT Press, 1987.

[H92] Hoskins, J., *IBM RISC System/6000: A Business Perspective*, Second Edition, John Wiley and Sons, Inc., New York, 1992.

[HK81] Hong, J-W. and H. T. Kung, "I/O Complexity: The Red-Blue Pebble Game," *Proc. 13th. Symp. on Theory of Comp.*, May 1981.

[IBM86] "ESSL Guide and Reference," order number SC23-0184-0, IBM Corporation, February 1986.

[IT88] Irigoin, F. and R. Triolet, "Supernode Partitioning," *Proc. 15th ACM Symp. on Principles of Programming Languages*, January 1988, pp. 319-328.

[LTT89] Lam, T., P, Tiwari, and M. Tompa, "Tradeoffs Between Communication and Space," *Proc. 21th Symp. on Theory of Comp.*, May 1989, pp. 217-226.

[MK69] McKellar, A.C. and E.G. Coffman, Jr., "Organizing Matrices and Matrix Operations for Paged Memory Systems," *CACM*, March, 1969, pp. 153-165.

[OS75] Oppenheim, A. V. and R. W. Schafer, *Digital Signal Processing,* Prentice-Hall, Englewood Cliffs, N.J. (1975).

[PW86] Padua, D. A. and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *CACM*, December, 1986, pp. 1184-1201.

[RR51] Rutledge, J.D., and H. Rubinstein, "Matrix Algebra Programs for the UNIVAC," Presented at the *Wayne Conference on Automatic Computing Machinery and Applications*, March, 1951.

[S86] Snyder, L., "Type Architectures, Shared Memories, and the Corollary of Modest Potential," *Annu. Rev. Comput. Sci. 1*, 1986, pp. 289-317.

[V90] Valiant, L. G., "A Bridging Model for Parallel Computation," *CACM*, August 1990, pp. 103-111.

[V-L92] Van Loan, C., *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philidelphia, 1992.

[VS90] Vitter, J.S. and E.A.M. Shriver, "Optimal Disk I/O with Parallel Block Transfer" *Proc. 22th Symp. on Theory of Comp.*, May 1990.