Parallelization and Performance Analysis of the Cooley–Tukey FFT Algorithm for Shared-Memory Architectures

ALAN NORTON AND ALLAN J. SILBERGER

Abstract—**We present here a study of parallelization of the Cooley–Tukey radix two FFT algorithm for MIMD (nonvector) architectures. Parallel algorithms are presented for one and multidimensional Fourier transforms. From instruction traces obtained by executing Fortran kernels derived from our algorithms, we determined the precise instructions to be executed by each processor in the parallel system. We used these instruction traces to predict the performance of the IBM Research Parallel Processing Prototype, RP3, as a computer of FFT's. Our performance results are depicted in graphs included in this paper.**

Index Terms—**Cooley–Tukey, fast Fourier transform, MIMD, parallelism, performance analysis, shared memory.**

I. Introduction

THE fast Fourier transform [3] has become a basic tool in various scientific and engineering disciplines, ranging from oil exploration to artificial intelligence. Efficient FFT computation is a requirement of any engineering/scientific computer. Because of the regular patterns of data access of the FFT, one can optimize FFT performance by designing special-purpose hardware or by careful pipelining and scheduling of the stages of the computation.

In this paper we instead ask what performance one might expect from FFT programs written in a high-level source language not specifically tailored for one particular computer architecture. We expect that studying the implementation of an FFT algorithm, because of the generality and usefulness of the algorithm and its extensive memory and floating point demands, will help us to understand related questions for other algorithms, too.

In order to evaluate the FFT algorithm in a general context, we derive cost functions (i.e., execution time as a function of problem size) for each of several basic kernels which are repeatedly executed in the various algorithms considered. We thereby construct cost functions for various parallel FFT algorithms. We vary the architecture-dependent parameters of these cost functions to compare various algorithmic and architectural optimizations. The performance estimates which we derive are based on idealized models of architectures, which would not be precisely realized by any one machine design, but provide a basis for comparison of the basic features of many different architectures.

We consider parallelization of both one-dimensional and multidimensional FFT's. The multidimensional algorithm is easily parallelized, by assigning different one-dimensional problems to different processors. We also present a parallelization of the one-dimensional FFT. Because of the additional communication requirements, the one-dimensional parallelization has more overhead than multidimensional problems. This overhead is measured, in order to determine when it is suitable to parallelize one-dimensional FFT's. Fortran codes for these programs can be found in [13].

The performance evaluation methods used here are not directly applicable to all numeric algorithms, primarily because of one special property of FFT's: the computation is completely data independent. In other words, there are no data-dependent branches in the code. These methods can be applied to other problems if one is able to determine average cost functions (as a function of problem size) for such code.

Pease's pioneering paper [15], taken up and reformulated by [20] and later by [19], appears to have been the first paper specifically devoted to the problem of creating parallel methods for computing the finite Fourier transform. The decade of the 1970's witnessed a widespread development of methods for computing Fourier transforms either on conventional serial processing systems or on array processors. We shall not review the serial algorithm literature; we shall mention only the paper [21] as a compendium of results concerning vector algorithms and as a guide to that literature. Some algorithms suitable for array processors are also useful as parallel algorithms too. For instance, the paper [8] describes a class of parallel FFT algorithms. Among papers which study parallel, nonvector implementations of the FFT, we note [1] and [7]. The algorithm of [1], although not presented in detail, appears to be a variant of the Pease algorithm tailored to the HEP architecture. Jeon and Reeves have studied implementations of the same generalization of Pease's work which we describe and study in this paper.

The structure of this paper is as follows: first, we present both the software and hardware options which are assumed available to the programmer. Then the performance prediction methodology is outlined. We describe several parallel FFT

Manuscript received July 15, 1985; revised August 12, 1986.

A. Norton is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

A. J. Silberger is with the Department of Mathematics, Cleveland State University, Cleveland, OH 44115.

IEEE Log Number 8713938.

0018-9340/87/0500-0581$01.00 © 1987 IEEE

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 02,2026 at 17:17:35 UTC from IEEE Xplore. Restrictions apply.

algorithms and discuss in some detail a parallel version of the radix two one-dimensional FFT. Parallel multidimensional FFT's are described briefly. The various cost functions are computed for one machine, the 512-way IBM RP3 system, using projected performance parameters.

Assumptions Concerning the Implementation of Parallel Algorithms

The parallel programs evaluated here derive from existing serial Fortran programs. We have designated various DO loops in the programs as parallel, in the sense that different iterations of these loops are independent. All parallelism in the FFT algorithms is obtained from parallel loops. This is not a hypothetical software environment; currently various parallel codes are being written for execution on multiprocessor IBM S/370 systems running the VM/SP operating system. We are primarily interested in predicting parallel performance on a much wider scale, such as the RP3 machine [16].

There are several ways of scheduling the parallel DO loops in these algorithms: the separate iterations can be allocated to the processors dynamically (i.e., by self-scheduling [5]) or the work can be statically preallocated. However, because the FFT algorithms have no data-dependent branches, there is no advantage to dynamic scheduling in these FFT problems, and such scheduling was not modeled.

Correct specification of these parallel algorithms requires us to distinguish between *shared* and *private* storage. We assume that each processor has exclusive access to its own private data, but all processors are able to read or write any shared data. All communication and synchronization between different processors is performed by references to shared memory.

We are also interested in the performance impact of *locality,* or the physical closeness of data. It may for example be faster for a processor to access data which are located physically near the processor than to access data which are at the opposite end of an interconnection network. We therefore required data of the FFT to be specified as either *local* or *global.* We measured performance of these algorithms as a function of the access times of local and global memory. To simplify these studies we assume private data to be allocated in local memory, and shared data to be allocated in global memory, except in one algorithm (the one-dimensional FFT) we permit one processor to access another's local memory.

Generic Architecture Description

To support the above style of programming, we assume a generic form of parallel architecture as pictured in Fig. 1. This generic model contains three levels of memory hierarchy: a private cache, a local memory, and a global memory accessed through a network. With this model one can approximate various existing or planned parallel architectures, as for instance the NYU Ultracomputer (which has no local memory), the Caltech Cosmic Cube [2] (which has no cache or shared memory), or the BBN Butterfly, which has no cache. We simplify the network by assuming that all global memory accesses take the same mean time, regardless of origin or destination, and that any access by one processor to another's local memory is equivalent to a global memory access. This

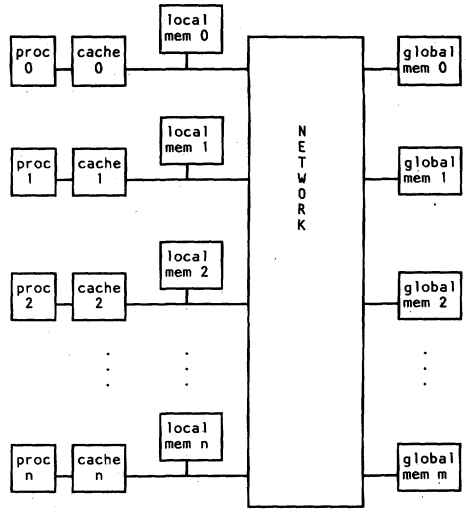


Fig. 1. Generic architecture of a shared memory system.

model accurately represents the RP3 design, and it also can be used to compare other architectures.

Using this generic structure, we measure the tradeoffs of local versus global memory use, but do not distinguish performance effects obtained by locating related tasks in nearby processors. Such use of locality is an important optimization in network-based parallelizations of the FFT (cf. [20]) which we intentionally forego in this study.

The particulars of the performance of any such parallel machine are determined by the values of a number of parameters, presented below. We have limited the number of such parameters in order to simplify the model while retaining essential understanding of performance-related issues.

The Processor Model

The processor performance is specified by the instruction set and the specific times for each instruction. We have divided the instructions into several basic categories, such as integer multiply, various floating point operations, loads, stores, and branches taken. By categorizing the instructions in this fashion we obtain a simplified model of modern processors such as the 801 [18] which separate internal operations from memory-referencing instructions. All register-to-register operations except multiply, divide, and floating point operations are assumed to take place in one processor cycle, and memory-referencing operations take place in a specified number of cycles, depending on the system configuration.

To completely describe the processor performance, it is necessary to specify the execution time of each such operation as well as the rate of occurrence of each instruction type. We assume that all operations in any one category execute in the same time. Only the average rates of occurrence of these instructions are used, so that no attention is paid to sequences of operations.

The Memory Network Model

The various memory-referencing instructions do not have the same execution time, but depend on whether the intended memory reference is in cache, or in local or global memory. To determine the performance of the memory/network hierar-

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 02,2026 at 17:17:35 UTC from IEEE Xplore. Restrictions apply.

chy we specify the various memory access times, including cache, local, and global memory, measured in processor cycles. Variance of global memory access time is also specified.

In actual hardware the memory delay parameters will not be constant, because memory and network contention can make a substantial difference in access time. We, however, represent them as constants, including an allowance for expected contention delays. Simulation and other modeling techniques (as in [14]) were used to provide realistic estimates of memory access time.

For these evaluations we do not model cache coherence, but make the following assumptions: the cache attains 100 percent hit ratios on cacheable data. All object code and private data are cacheable, but shared memory, when used for storing data of the FFT and for storing synchronization variables, is not cacheable. Sine tables are cacheable only if stored in private memory.

While these cacheability assumptions may seem oversimplified, we note that experiments have shown that a two-way set-associative cache will develop hit ratios in excess of 99.9 percent on instructions and local data in the FFT code. On the other hand, if the FFT data are made cacheable, a moderate to large problem could repeatedly overrun any feasible cache size without making effective use of cache-line locality.

The input and output architecture is not explicitly defined in this model. We specify two parameters, the I/O fixed delay-time and data transmission rate. We assume that each I/O operation is initiated by a processor's request, which requires a fixed I/O delay. The data transfer then occurs at the specified data transmission rate. Input and output are allowed to be performed to or from any series of sequential addresses in the system; to obtain input or output to nonconsecutive addresses, we perform an additional memory-to-memory transfer. To estimate the cost of a synchronization which follows input or output, we also specify the variance of the I/O latency.

The complete list of architecture-dependent parameters includes the number of processors, the times for each type of instruction, variances of global memory access time and I/O latency, input/output initiation time, and I/O sustained rate.

Performance Prediction Strategy

The FFT algorithms evaluated in this study all possess a very regular structure in terms of instruction counts, memory references, etc. This regularity enables us to predict performance for problems which are too large to execute on existing hardware. We do not claim this method will produce completely accurate performance predictions; in particular, we ignore many dynamic aspects of parallel execution. We are interested in making apparent the performance impact of basic algorithmic and architectural changes, without providing more detail than can be easily assimilated. Our method consists of the several steps described below.

Algorithm Decomposition

First, we express each FFT algorithm in terms of various basic subcomponents or kernels. These kernels include:

1) computing and storing in memory the sine table for a one-dimensional FFT,
2) performing a one-dimensional FFT on data in memory, with the sine table already residing in memory,
3) performing a portion of a bit-reverse permutation on data in memory,
4) performing a portion of a shuffle permutation of data in memory,
5) moving data from local to global memory, or vice versa,
6) moving data from disk to memory or vice versa,
7) waiting for all processors to finish one task before proceeding to the next (barrier synchronization).

All of the algorithms considered here can be expressed in terms of various combinations of the above kernels. Note that, except for the last, all the kernels are executed by one processor.

Measurement of Kernels from Instruction Traces

Next, we determine the instruction counts for executing each kernel as a function of the problem size. For the first five of the above kernels we traced execution of actual Fortran programs, and obtained instruction and memory reference counts of such programs compiled either by the *H*-extended (FORTHQ) or VS/Fortran optimizing compilers. The resulting counts did not vary much between compilers, however; substantially higher counts were obtained when the compiler optimization was not used.

The instruction counts obtained from IBM S/370 code must be altered because of differences between S/370 and 801 architectures. One major difference is that some S/370 instructions perform operations on memory locations, while 801 instructions first load memory contents into registers before acting on them. We compensate for these differences by replacing the 370 operation with a memory load in addition to the appropriate register–register operation. Some 370 instructions, such as integer multiply, require multiple 801 instructions, and we provide an appropriate adjustment of the counts. This translation from 370 to 801 instructions does not take into account the architecture dependent optimizations which an 801 compiler could perform.

Determination of Instruction Count Functions

We use the measured counts to predict the precise instruction counts of the given program for any problem size. Due to the simple structure of the FFT algorithms, each such count can be expressed precisely as a linear function of N, log (N), N log (N), and sqt (N). (Here log (N) refers to the base-2 logarithm, which is an integer. By sqt (N) we denote a modification of the real square root function: if N is an even power of two, sqt (N) is the actual square root. If N is an odd power of two, sqt (N) is the square root of $2N$. This function results from the bit-reversal permutation of an FFT.) By curve fitting, we obtain the coefficients precisely, thereby giving memory reference and instruction counts for each kernel as a function of N.

In [13] we tabulated the coefficients determined by this method. Using these coefficients we express the precise kernel instruction counts as functions of the form $A + BN + C \log$

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 02,2026 at 17:17:35 UTC from IEEE Xplore. Restrictions apply.

$(N) + D\,\mathrm{sqt}(N) + EN\log(N)$. The dominant term, $EN\log(N)$, provides the asymptotic complexity of the kernel. By including lower order terms, we determine the exact operation count.

Computation of Other Kernels

The instruction counts computed for kernels 1–5 are used directly to estimate execution times of those kernels. However, the time to execute kernels 6 and 7 cannot be estimated easily from analysis of traces.

I/O Cost: We estimate input and output time by a simple model: one I/O operation results in a fixed latency for initiation, and continues at a constant data rate in bytes/cycle. In order to obtain very large I/O rates, it is essential to divide the I/O process over many parallel units. We, therefore, assume that input and output data are spread across different physical devices in appropriately sized chunks. When the effect of I/O pipelining is modeled, we assume that a processor cannot overlap its first read and last write operation with computation; however, successive I/O requests are overlapped, so the startup latency is a one-time overhead for any series of I/O operations from one processor.

Barrier Synchronization: Even though all of the tasks in a parallel DO loop may be of the same size, they will not always finish at the same time because of the random effects of network contention. At the end of such a loop it is necessary for all processors to wait for the last one to complete before proceeding to the next stage of the computation. The average wait per processor, called barrier synchronization time, must be included in determining execution time.

We used an estimate given by Kruskal and Weiss [11], to determine the barrier wait in terms of the variance of task size. This variance was estimated as the number of memory accesses times the variance of memory access time, since all other times involved in these task executions are constant, depending only on the processor execution rate. The Kruskal–Weiss estimate gives $\sqrt{2MV\log(P)}$ as the barrier delay, where V is the variance of memory access time, P is the number of processors, and M is the (expected) number of global memory reads per task. We estimate the variance of global memory access time from the second model in [10], which provides the variance in terms of the network utilization.

Limitations of the Method

The method described above ignores certain aspects of program behavior which affect the accuracy of our results. We list some of the notable omissions:

Operating System Overhead: The operating system can substantially reduce performance, depending on its specific characteristics. Our model does not include the system overhead for managing the page/segment memory mapping. If, for example, an FFT code is traversing a large fraction of global memory, one can expect frequent misses in the TLB (translation lookaside buffer), to result in a substantial overhead for such global access. The precise cost is difficult to estimate because it depends on many specific features of the mapping scheme, such as page size, TLB size, whether TLB entries are cacheable, etc.

Details of Cache Performance: We have intentionally included a simplified model of cache performance for the evaluation. Cacheability of shared data was not modeled here, and such cacheing can improve performance.

I/O Modeling: Our simplified I/O structure assumes equal access from any processor to any I/O device. In practice that requires either an I/O network or additional manipulation of data in shared memory. In modeling a specific machine, the actual layout of data on disk is relevant to estimates of I/O cost.

Contention: We have treated memory, synchronization, and I/O delays as constants, whereas these delays will depend on the utilization of the various resources. A static model such as employed here cannot precisely predict the dynamic effects of contention. To evaluate contention without relying on complete simulation, we have estimated the constant delays from network simulation of random traffic. The dynamics of processor/network interaction can be better estimated from the methods in [14].

Access Patterns: FFT algorithms have specific access patterns to the data which can be used to optimize synchronized execution. When different processors exercise such memory accesses asynchronously (as in our model of parallel execution), additional contention may result due to conflicting communication patterns in the network. This contention results not from different processors accessing the same memory address simultaneously, but from simultaneous accesses to different addresses which happen to reside at the same physical memory module. We have simulated such conflicting communication patterns and found them to decrease network throughput by as much as 20 percent. Address skewing hardware (resulting in a permutation of the physical address space) is incorporated in RP3 to eliminate such patterned contention. In modeling the FFT algorithm on a system with no address skewing, one should allow for additional network and memory delays.

Network Locality: With our model we cannot directly model nearest neighbor communication versus distant communication, except by modifying the overall average of global memory access. FFT algorithms which utilize locality require different modeling techniques.

Block Data Transfer: All data transfers in these algorithms are assumed to take place as loads and stores by the various processors and I/O processors. In message-based systems, however, a more appropriate model would include transfers of large amounts of data as being more efficient than individual word transfers.

The Parallel FFT Algorithms

We outline in this section the various algorithms which we examine. In the case of one-dimensional FFT's, some analysis was necessary to determine the appropriate kernels for parallelization. Detailed descriptions of the kernels are pre-

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 02,2026 at 17:17:35 UTC from IEEE Xplore. Restrictions apply.

sented in the Appendix, and actual Fortran code is presented in [13]. Here we briefly indicate the structure of the algorithms.

Suppose the number P of processors is a power of 2, $P = 2^{(M-K)}$, less than the number of points to be transformed, $N = 2^M$. The entire FFT consists of a bit-reversal permutation followed by M computational stages. After the bit reversal, the first K stages can be subdivided into P independent computations, each having the dataflow of an FFT (after bit reversal) on 2^K points. The remaining $M - K$ stages of the computation can be similarly subdivided, after an inverse shuffle permutation of the data, and the process may be repeated until all M stages have been computed. If K does not divide M, the final stage of the computation is shorter than the others.

For this algorithm, the FFT data to be transformed are placed in the union of the local memories of the system, rather than in global memory. Global memory is used to store the sine table. We outline the algorithm as follows.

1) Each processor computes a part ($N/4P$ sines) of the table, storing values in global memory.

2) Each processor performs the part of a bit-reversal permutation on the segment (N/P complex numbers) of the input data which is located in its own memory. For each datum, a processor computes the location of the datum with which it is to be exchanged; when this address is larger than the location of the given datum, it exchanges the two complex numbers.

3) Barrier synchronization (wait for all processors to finish).

4) Repeat the following until all stages have been completed:

a) Each processor performs a computation like an N/P-point FFT on data in local memory (without bit reversal), except that the values used from the sine table are indexed via a shuffle permutation.

b) Each processor performs a portion of an unshuffle permutation, transferring data from its local memory to other local memories.

c) Barrier synchronization.

The parallel algorithm is illustrated in the "butterfly diagram" Fig. 2, which shows the dataflow for a 16-point FFT, to be executed on four processors. After the first two stages of the computation are performed locally, an inverse shuffle permutation rearranges the data so that the remaining two stages can again be performed in local memory.

Two and higher dimensional FFT's admit a very straightforward parallelization algorithm. Such FFT's can be realized as multiple one-dimensional FFT's applied to subarrays of the given higher dimensional data array. These one-dimensional FFT's can be performed in parallel, as large parallel DO loops. For example, a two-dimensional FFT algorithm can be expressed in terms of one-dimensional FFT's as follows:

```
do_parallel: for i = 1 to number_of_rows
  perform 1-dim FFT in place on row i;
  end;
barrier synchronization (wait for all processors to complete);
do_parallel: for i = 1 to number_of_columns
  perform 1-dim FFT in place on column i;
  end;
```

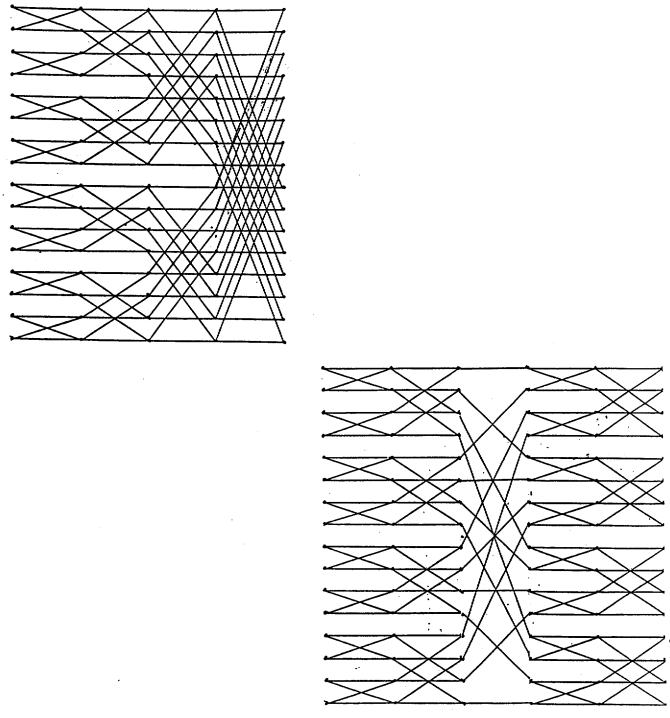


Fig. 2. Butterfly diagrams for a parallel one-dimensional problem. The upper diagram shows the dataflow for a 16-point FFT. An inverse shuffle is inserted after the second stage of the computation, resulting in the lower diagram. If this is executed on four processors, the first two stages and the last two stages will be performed on data in local memory. The intermediate inverse shuffle transfers data between the local memories of different processors.

Given any serial one-dimensional FFT program, it is easy to parallelize multidimensional FFT's in this way. Several choices which affect the performance are hidden in the above description.

In order to evaluate the usefulness of local memory, we considered two variants of the algorithms: either the various processors compute the one-dimensional FFT's on the data, in place, in global memory, or they first copy the data from global to local memory, perform the FFT on the local data, and then copy the results back to global memory.

To better understand the effect of I/O on the computation, we evaluated three different variants of the three-dimensional algorithm. First, we evaluated the performance when the data are kept in memory (using the local/global copying, as above). Second, we assumed the data initially resided on disk, with the separate one-dimensional subarrays of data independently accessible by all processors. In this case, the data are assembled in memory into two-dimensional slices, transformed there, and then copied back to disk in the appropriate order. To complete the transform requires three passes through the full three-dimensional data array. Third, we estimated the effect of pipelining disk access, by permitting computation on each two-dimensional slice of data to be overlapped with the I/O on other slices.

EXAMPLE: RP3

Using the methods of the preceding sections we have estimated the performance of a 512-way RP3 on various FFT

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 02,2026 at 17:17:35 UTC from IEEE Xplore. Restrictions apply.

algorithms. Note that the RP3 architecture [16] is of the generic form for which these modeling techniques were designed. Architecture parameters of the model, such as I/O rates, floating point operation times, etc., were set in accordance with the planned RP3 hardware and network simulation results. Global memory delay is estimated as in [14], consistent with the memory reference rates deduced in this analysis.

We considered the following algorithms, which are described above:

1) one-dimensional parallelized FFT, with data residing in local memories,

2) two-dimensional FFT performed in place on data in global memory,

3) two-dimensional FFT in memory, performed by copying data from global memory to local memory, transforming it there, and copying back,

4) three-dimensional FFT, performed on data in memory by copying from global to local and back,

5) three-dimensional FFT, with data residing on disk, using synchronous I/O.

6) three-dimensional FFT, with data residing on disk, computation overlapped with I/O.

Fig. 3 illustrates the performance of the algorithms as a function of problem size. The computation time is represented in terms of the number of processor cycles required for completion. This time includes the various instruction executions, as well as waiting time resulting from barriers. To gauge the efficiency of these computations, Fig. 4 displays the number of megaflops achieved on these problems.

For comparison we have also plotted some CPU times measured on an IBM 3081 for a serial one-dimensional FFT in memory.

The graphs show that the one-dimensional algorithm and the disk-accessing three-dimensional algorithm are more time consuming than the other algorithms, almost by a factor of two, for large problems. The additional cost of the one-dimensional algorithm is due to the additional permutations and data manipulation performed in the algorithm.

For large problems (involving over 1 GByte of data), it is inappropriate to perform the FFT in memory, and I/O overhead must be considered. Algorithms 5 and 6 show the performance when I/O overhead is included. The difference between algorithms 5 and 6 shows a significant advantage to performing I/O asynchronously, overlapping it with computation. A major component of the I/O cost is the latency required to initiate an I/O operation, and asynchronous I/O can be used to recover most of that lost time.

We measured the effect of local/global copying of data, by examining the performance of algorithms 2 and 3. Keeping local memory speed constant, global memory speed was varied, obtaining execution times indicated in Fig. 5. For the problem sizes indicated (and for larger problems as well) it is clearly beneficial to perform local/global copying, even if the global memory is only 10 percent slower than local memory. The benefit is substantial for RP3, where the local/global speed ratio is estimated to be about 2.

Rather than scaling the problem size, in Fig. 6, we show

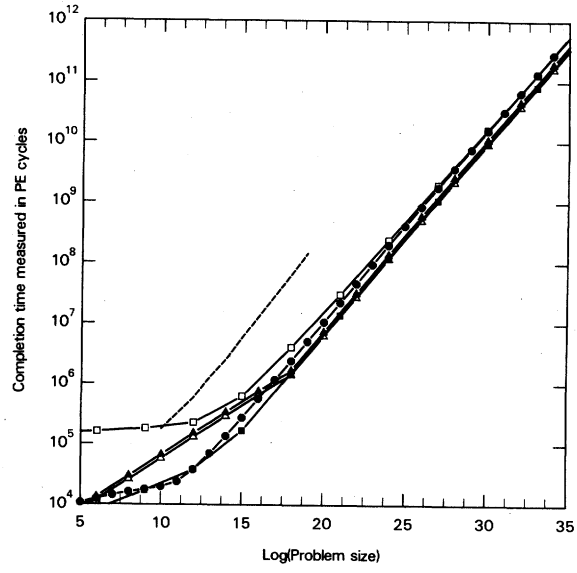


Fig. 3. Predicted performance of RP3 on parallel FFT algorithms. Techniques described in the text are used to estimate the number of processor cycles required to complete various FFT problems. Algorithms are: one-dimensional (●), two-dimensional (▲) in shared memory, two-dimensional (△) with local/global copying, three-dimensional (■) in memory, and three-dimensional (□) on disk. Dashed curve is approximate 3081 timing of serial one-dimensional FFT.

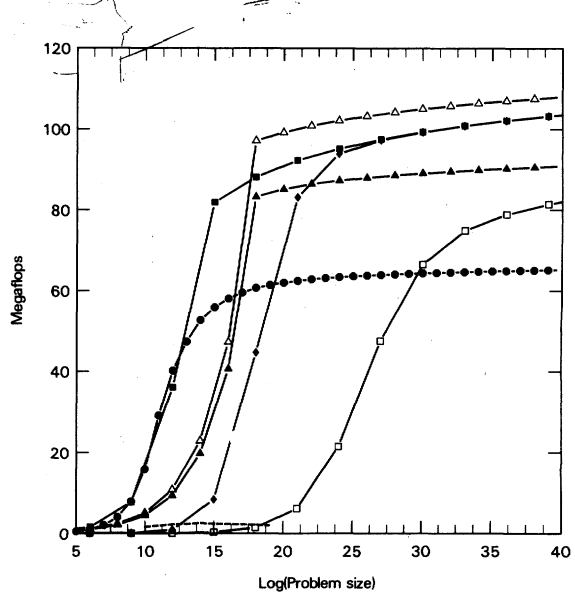


Fig. 4. Megaflops predicted for RP3 on parallel FFT algorithms: The FFT performance of Fig. 3 is presented here in terms of the megaflops achieved on the various FFT problems. Algorithms are: one-dimensional (●), two-dimensional (▲) in shared memory, two-dimensional (△) with local/global copying, three-dimensional (■) in memory, three-dimensional (□) on disk, and three-dimensional with pipelined I/O (♦). Dashed curve is approximate 3081 timing of serial one-dimensional FFT.

how the granularity in the FFT algorithms affects the efficiency. We estimate the performance of the same FFT problem (2^{30} points) executed by a variable number of processors, using one, two, or three-dimensional problems (algorithms 1, 3, and 4). In order to realistically accommodate the increased time to global memory, the global memory access time, its variance, and the queue access time, are all scaled in proportion to the logarithm of the number of processors. We see in Fig. 6 that the overhead of 1-D

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 02,2026 at 17:17:35 UTC from IEEE Xplore. Restrictions apply.

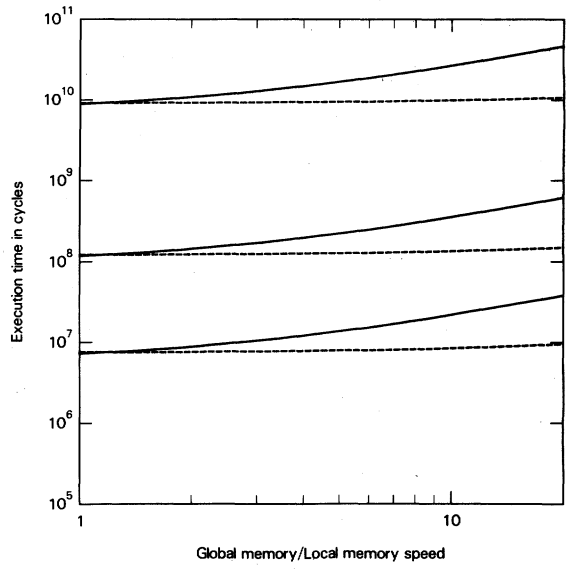


Fig. 5. Effect of local/global copying. This graph shows the difference in performance between a two-dimensional FFT performed in place in global memory versus the same FFT performed by copying to local memory, transforming locally, and copying back. Solid curves indicate time to compute in place, dashed curves describe local/global copying. Horizontal coordinate indicates the ratio of global/local access times, assuming ten processor cycles for local memory access. Three different problem sizes are represented, 2^{20}, 2^{24}, 2^{30} complex points.

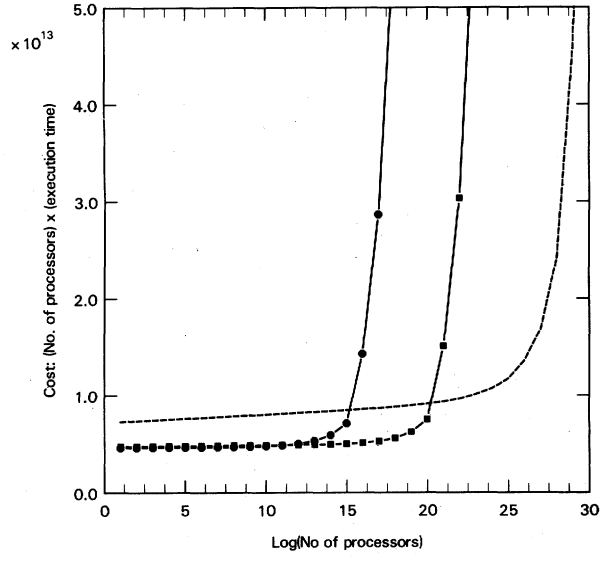


Fig. 6. Granularity of FFT algorithms. Cost of performing one, two, and three-dimensional parallel algorithms with varying number of processors, on 2^{30} complex points. Global memory access time, global memory variance, and synchronization time are increased in proportion to the logarithm of the number of processors. One-dimensional (dash), two-dimensional (●), three-dimensional (■).

parallelization is substantial, so that one should not parallelize the one-dimensional problems inside a two-dimensional computation, unless the number of processors available becomes comparable to the parallelism in the algorithm. If, however, more than 2^{15} processors are available, they can be effectively utilized on this problem by parallelizing the one-dimensional subproblems.

Analysis of FFT timing can be instructive in assessing the suitability of RP3 for scientific computations with computational demands similar to the FFT. One key issue is the balance of the machine, or whether any component will be a bottleneck in such computations. Related to that is the utilization of the memory and I/O subsystems, or the fraction of capacity at which such systems will be expected to operate. We summarize these results below; the conclusions we draw are largely independent of problem size or FFT algorithm.

- Most of the time spent on the FFT is for floating point operations, memory access, integer divides, and I/O (when data are on disk). All other operations amount to less than 10 percent of the total computation time.
- The times spent on floating point, memory access, and I/O are about the same. This implies that no one of these categories can be regarded as the performance bottleneck. By the same token, FFT performance by RP3 cannot be substantially improved without speeding up all three categories of operations. However, the FFT algorithm does not saturate either the I/O, floating point, or memory access capabilities of the RP3 machine. Therefore, much improvement is possible from pipelining these activities: overlapping memory fetching with floating point, and overlapping I/O with computation could bring as much as a threefold performance increase.
- While the cost of global memory access is substantially reduced by local/global copying of data, the local memory access cost is increased. With such copying, total memory accesses remain a major component of the computation time.
- The impact of integer division is substantial (10–30 percent) because of numerous divisions by powers of two. This cost would be made negligible if performed as right shifts rather than integer divides. Appropriate source modification or compiler support would provide this improvement.
- The cost of synchronization is not serious, both for barrier synchronization and fetch-and-add references. When the fetch-and-add operation is used for dynamic DO loop scheduling, it occurs sufficiently infrequently that it will not be a source of memory or network contention.
- The RP3 network should be adequate to handle the communication requirements of FFT's. The greatest network load occurs when the computation is done in global memory. In that case, each processor will issue a global memory reference about every 33 cycles, issuing two loads for every store. That is approximately the planned capacity of the RP3 network. If local/global copying is used, the network load will be slight, about one reference per 500 cycles.
- The RP3 memory should also be adequate. Its highest utilization will occur when the computation is performed in local memory. The average memory reference rate will be about once per 20 processor cycles, well below planned memory capacity.
- The store-through cache of RP3 results in more memory accesses than a store-in cache would cause. The difference is, however, not great, at most one extra memory reference per 100 cycles. On the whole, this analysis shows the RP3 design to be well balanced for FFT computation.

In addition, we may conclude that other algorithms with similar computation and communication requirements will perform similarly on RP3. Because the algorithms as modeled for RP3 do not take advantage of the regular structure of the

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 02,2026 at 17:17:35 UTC from IEEE Xplore. Restrictions apply.

FFT, RP3 will not perform as well as highly pipelined and vectorized architectures which have been carefully programmed to optimize FFT performance. However, the substantial computing resources of RP3 will be invoked from Fortran, without vectorization, using minimal programmer effort.

If one were to tailor the RP3 design for FFT computation, the major improvement to seek would be efficient vector floating point with scatter–gather memory accessing capability, to permit the computation to be fully pipelined.

Conclusion

We have presented here parallelizations of the Cooley–Tukey radix two FFT algorithm, measured the execution of these algorithms, and demonstrated a methodology for predicting performance of such algorithms on shared-memory architectures.

It is perhaps not surprising that FFT algorithms contain substantial parallelism; parallel and vector algorithms for the FFT are widely known. We have, however, demonstrated that this parallelism can be made available in a high-level scientific programming language, and can with a minimum of effort be implemented in a suitable shared-memory computer architecture, like RP3.

By using these techniques we are able to size various parallel architectures for FFT performance, or adapt the algorithm to fit the architecture. We have applied the analysis to the RP3 architecture, predicting how a 512-way machine will perform.

The FFT is a special kind of parallel algorithm, in that the total computation can be evenly distributed across a large number of processors, with very little overhead for synchronization. To apply these performance analysis methods to other algorithms, careful attention must be paid to serialization, for the distribution of work to processors may be less efficient than with FFT's. The approach we presented here can be applied to problems other than the FFT, provided they admit a static description of the tasks to be performed by each processor.

Appendix

The Mathematics of FFT Parallelization

In this Appendix we develop and describe the algorithms which are studied in the main part of the paper. We regard the problem of computing the Fourier transform on N points as the problem of computing the product of a (row) N-vector (of indeterminates) and the N-by-N Fourier transform matrix. We derive the radix two Cooley–Tukey FFT algorithm as a factorization of the Fourier transform matrix into a product of sparse matrices. For our purposes, a parallel algorithm is represented as a multiplication by a block-diagonal matrix. The flow of data among processors, between the stages of a computation, is represented by a permutation matrix. As the main goal of this Appendix, we show how the matrix factorization associated to the Cooley–Tukey algorithm may be transformed into a product of block-diagonal matrices and permutation matrices.

The Serial One-Dimensional FFT Algorithm

The serial FFT algorithm which we parallelize in this paper is essentially just the Cooley–Tukey algorithm. In the first part of the Appendix we recall the factorization of the Fourier transform matrix which underlies the Cooley–Tukey algorithm. We shall use this factorization in constructing our parallel algorithm; in essence the parallel algorithm will implement a slightly more complicated factorization in which the factors will be block-diagonal matrices and permutation matrices.

For the following discussion we shall employ notations for the direct sum and for the tensor product of square matrices. Let $X = (x_{i,j})$ and $Y = (y_{k,l})$, with $1 \le i, j \le P$ and $1 \le k, l \le Q$, be square matrices. Then the *direct sum* of X and Y is the $(P + Q)$-by-$(P + Q)$ block-diagonal matrix

$$X \oplus Y = \begin{bmatrix} X & 0 \\ 0 & Y \end{bmatrix},$$

with X and Y as main diagonal blocks and 0 entries outside of these blocks. The *tensor product* of X and Y is the PQ-by-PQ matrix

$$X \otimes Y = \begin{bmatrix} x_{1,1}Y & \cdots & x_{1,P}Y \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ x_{P,1}Y & \cdots & x_{P,P}Y \end{bmatrix},$$

which may be regarded as the P-by-P array of submatrices $x_{i,j}Y$. We extend the definitions of the direct sum and of the tensor product of matrices to any number of summands or factors, respectively, in such a way as to produce associative operations. In the particular case that X is the P-by-P identity matrix $X \otimes Y = Y \oplus \cdots \oplus Y$, P summands, while, in the case that Y is the Q-by-Q identity matrix, one sees that $X \otimes Y$ is a P-by-P array of Q-by-Q scalar matrices, the i, jth scalar matrix having the diagonal entries $x_{i,j}$.

To construct a power of two FFT algorithm consider the following factorization of the Fourier transformation matrix. For M and L positive integers let $N = 2^M$ and assume that $0 \le L \le M$. Let $W = W_M = e^{2\pi i/N}$ $(i = \sqrt{-1})$ and, more generally, let W_L denote the power 2^{M-L} of W. Let I_L denote the $2^L \times 2^L$ identity matrix, 0_L the $2^L \times 2^L$ zero matrix, and Λ_L the $2^L \times 2^L$ diagonal matrix with entries the first 2^L powers of W_{L+1} enumerated down the diagonal in natural order from the 0th power to the power $2^L - 1$. For instance, if $L = 2$, then

$$\Lambda_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \omega & 0 & 0 \\ 0 & 0 & \omega^2 & 0 \\ 0 & 0 & 0 & \omega^3 \end{bmatrix},$$

where $\omega = e^{2\pi i/8} = (1 + i)/\sqrt{2}$.

Let F_L denote the $2^L \times 2^L$ Fourier transform matrix (i.e., $F_L = (W_L^{jk})$, $0 \le j, k \le 2^L - 1$) and let B_L denote the matrix which represents the bit-reversal permutation for a vector space of dimension 2^L with basis indexed by the ordered set of

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 02,2026 at 17:17:35 UTC from IEEE Xplore. Restrictions apply.

binary numbers k, $0 \le k < 2^L$. Let

$$J_{L+1} = \begin{bmatrix} I_L & I_L \\ I_L & -I_L \end{bmatrix}$$

and

$$D_{L+1} = \begin{bmatrix} I_L & 0_L \\ 0_L & \Lambda_L \end{bmatrix}.$$

For instance,

$$J_2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

and

$$D_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{bmatrix}.$$

Clearly, $F_1 = J_1$. For $M > 1$ we have

$$\begin{aligned} B_M F_M &= (B_{M-1}F_{M-1} \oplus B_{M-1}F_{M-1})D_M J_M \\ &= \begin{bmatrix} B_{M-1}F_{M-1} & 0_{M-1} \\ 0_{M-1} & B_{M-1}F_{M-1} \end{bmatrix} \begin{bmatrix} I_{M-1} & I_{M-1} \\ \Lambda_{M-1} & -\Lambda_{M-1} \end{bmatrix} \\ &= (J_1 \oplus \cdots \oplus J_1)(D_2 J_2 \oplus \cdots \oplus D_2 J_2) \\ &\quad \cdots (D_{M-1}J_{M-1} \oplus D_{M-1}J_{M-1})D_M J_M \\ &= (I_{M-1} \otimes J_1)(I_{M-2} \otimes D_2 J_2) \\ &\quad \cdots (I_1 \otimes D_{M-1}J_{M-1})D_M J_M. \end{aligned}$$

Notice that the final product consists of M factors, each of which is an $N \times N$ block diagonal matrix with exactly two nonzero entries in each row and column. The number of blocks increases from 1 to $N/2$ from right to left, while the size of the blocks increases from 2×2 to $N \times N$ in the opposite direction. As an example, observe that

$$B_2 F_2 = (I_1 \otimes J_1)(I_0 \otimes D_2 J_2),$$

where

$$I_1 \otimes J_1 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

and

$$I_0 \otimes D_2 J_2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & i & 0 & -i \end{bmatrix}.$$

We convert the above matrix product into a serial algorithm for computing the Fourier transform on N points as follows. Let $f = f(i)$ denote a complex valued function defined on the finite set of integers i, $0 \le i < N$. Write $\mathfrak{F}_M$ for the Fourier transform operator on N points. We wish to compute the sum

$$(\mathfrak{F}_M f)(m) = \sum_{i=0}^{N-1} f(i) W^{mi}.$$

To represent this computation as a procedure we first write it as a matrix product. For this represent f as the row vector

$$f = [f(0), \cdots, f(N-1)].$$

To compute the Fourier transform $\mathfrak{F}_M(f)$, we compute the matrix product fF_M i.e., $(\mathfrak{F}_M f)(m)$ is the mth component of the row vector fF_M.

Given the above factorization of the matrix F_M, we may compute this matrix product by computing the products of the row vector f with the factors of F_M from left to right. To begin this procedure we first compute the matrix product fB_M; this amounts to reordering the values of f such that they are listed in bit-reversed order. This part of the algorithm appears in a Fortran program as a DO loop which computes the bit-reversing permutation. The remaining $2M$ matrix products from left to right are computed as a nested DO loop. A Fortran program implementing the serial algorithm appears in [13].

The Parallel One-Dimensional Algorithm

To obtain algorithms which parallelize the computation of one-dimensional FFT's we generalize a matrix factorization of F_M from the paper of Pease [15]. To explain our matrix factorization of F_M we shall have to introduce some more terminology and give still another representation of the Fourier transformation $\mathfrak{F}_M$.

For $0 < L \le M$ we write $\mathcal{V}_L$ for the complex vector space of dimension 2^L with a basis which we now describe. First, $\mathcal{V}_1$ will have basis vectors e_0 and e_1. For $L > 1$, $\mathcal{V}_L$ will be the Lth tensor power of $\mathcal{V}_1$. This means that we may give $\mathcal{V}_L$ a basis consisting of the set of L-fold tensor products of the basis elements e_0 and e_1 of $\mathcal{V}_1$. Concretely, for $0 \le i < 2^L$ we represent i as the binary number $i_{L-1} \cdots i_0$. Then we have as an ordered basis for $\mathcal{V}_L$ the set of all 2^L vectors of the form $e(i_{L-1}) \otimes \cdots \otimes e(i_0)$, where $e(i_K) = e_0$, when $i_K = 0$, and $e(i_K) = e_1$, when $i_K = 1$, $0 \le K < L$.

The two-by-two Fourier transform matrix $F_1 = J_1$ represents the linear transformation $\mathfrak{F}_1$, where

$$\mathfrak{F}_1(e_j) = e_0 + (-1)^j e_1 (j = 0, 1).$$

For any $L \ge 0$ let $\mathfrak{I}_L$ denote the $2^L \times 2^L$ identity operator on $\mathcal{V}_L$. Then one sees that, more generally, the matrix J_L represents the linear transformation $\mathfrak{J}_L = \mathfrak{F}_1 \otimes \mathfrak{I}_{L-1}$ of the vector space $\mathcal{V}_L = \mathcal{V}_1 \otimes \mathcal{V}_{L-1}$ relative to the ordered basis of $\mathcal{V}_L$ we have described above. In the same way the block diagonal $2^M \times 2^M$ matrix

$$I_{M-L} \otimes J_L = J_L \oplus \cdots \oplus J_L$$

represents the linear transformation $\mathfrak{I}_{M-L} \otimes \mathfrak{J}_L$. We will write $\mathfrak{A}_L = \mathfrak{I}_{M-L} \otimes \mathfrak{J}_L$ and $A_L = I_{M-L} \otimes J_L$.

Now let $\mathfrak{S} = \mathfrak{S}_M$ denote the linear transformation of $\mathcal{V}_M$ which sends $e(i_{M-1}) \otimes \cdots \otimes e(i_0)$ to $e(i_{M-2}) \otimes \cdots \otimes e(i_0)$

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 02,2026 at 17:17:35 UTC from IEEE Xplore. Restrictions apply.

$\otimes e(i_{M-1})$, i.e., $\mathcal{S}$ is the linear transformation which cyclically permutes the tensor factors of any basis vector of $\mathcal{V}_M$ so that the highest order factor of any basis vector becomes the lowest. Concretely, $\mathcal{S}$ effects a shuffle permutation of this ordered basis. We write S for the matrix representing $\mathcal{S}$ relative to the given basis for $\mathcal{V}_M$. We use the symbol $\cdot$ to denote composition of mappings from right to left. With this convention matrix multiplication, matrices acting by left multiplication on column coordinate vectors, corresponds to composition of linear transformations of $\mathcal{V}_M$. We prove the following assertion.

Lemma:

$$\mathcal{A}_{L-1} = \mathcal{S}^{-1} \cdot \mathcal{A}_L \cdot \mathcal{S}.$$

Proof: We have

$$\begin{aligned}
&\mathcal{S}^{-1} \cdot \mathcal{A}_L \cdot \mathcal{S}(e(i_{M-1}) \otimes \cdots \otimes e(i_0)) \\
&\quad = \mathcal{S}^{-1} \cdot (\mathcal{I}_{M-L} \otimes \mathcal{J}_L)(e(i_{M-2}) \\
&\qquad \otimes \cdots \otimes e(i_0) \otimes e(i_{M-1})) \\
&\quad = \mathcal{S}^{-1}(e(i_{M-2}) \otimes \cdots \otimes e(i_{L-1}) \otimes \mathcal{F}_1(e(i_{L-2})) \\
&\qquad \otimes \cdots \otimes e(i_0) \otimes e(i_{M-1})) \\
&\quad = e(i_{M-1}) \otimes \cdots \otimes \mathcal{F}_1(e(i_{L-2})) \otimes \cdots \otimes e(i_0) \\
&\quad = \mathcal{I}_{M-L+1} \otimes \mathcal{J}_{L-1}(e(i_{M-1}) \otimes \cdots \otimes e(i_0)).
\end{aligned}$$

■

The above Lemma implies the following statement concerning matrices:

Corollary:

$$S^{-1} A_L S = A_{L-1}.$$

From this Corollary we obtain for any positive integer $K < M$ a product representation of $B_M F_M$. For simplicity let $M = KL$, where L is an integer. Then we have for $B_M F_M$ the product representation

$$\begin{aligned}
&A_1 \cdots D'_K A_K S^K D'_{K+1} A_1 \cdots D'_{2K} A_K S^K \\
&\qquad \cdots S^K D'_{(L-1)K+1} A_1 \cdots D'_{LK} A_K S^K.
\end{aligned}$$

We must observe that, for any integer $P = QK + R$, $0 \le Q < L$ and $0 < R \le K$,

$$D'_P = S^{-QK}(I_{M-P} \otimes D_P) S^{QK}.$$

Although it is not very interesting to write down formulas for the entries of the diagonal matrices D'_P, it is easy to give programs for computing these matrices from the matrices $I_{M-P} \otimes D_P$ occurring in the original factorization of $B_M F_M$.

Let us give some indication as to how this matrix factorization may be implemented as a parallelized algorithm for computing the Fourier transform on N points. We regard the vector f (see the preceding section) as segmented into subvectors of length 2^K. The final factorization which we have given above represents the matrix $B_M F_M$ as a product in which the factors alternate between block-diagonal matrices, with blocks of size $2^K \times 2^K$, and permutation matrices. We program multiplication by a $2^K \times 2^K$ block of a matrix as a kernel of our algorithm. Factoring the blocks of these block-diagonal matrices into sparse matrices, which differ from the usual factors of $B_K F_K$ only in the diagonal matrices which occur, we see that our kernels are themselves very similar to FFT's (however, with the bit-reversal permutation removed and with different diagonal matrices of "twiddle factors"). The permutation matrices, which represent powers of shuffle permutations, are implemented as the passage of data between processors. Thus, our parallelized algorithm begins with a parallelized computation of the bit-reversing permutation followed by a synchronization of the processors. It continues with each of 2^{M-K} processors computing the part of an FFT on 2^K points which comes after the bit-reversing permutation. Following this first FFT program each processor sends its data points to their new locations and we synchronize the processors. We continue with $(M/K) - 1$ stages, each stage consisting of a "chunk" FFT, a permutation of the data, and a synchronization. These kernels were programmed in Fortran and included in [13].

Acknowledgment

The authors wish to acknowledge, with thanks, the assistance of the following colleagues during the preparation of this work. B. Lubachevsky described to us in general terms the parallel one-dimensional FFT algorithm which we construct. L. Auslander suggested that we study the tensor product representation of the Cooley–Tukey algorithm in order to understand its parallelizations. G. Pfister's ideas in performance evaluation were essential in this work, as well as his advice and support. K. So provided us with a VM simulator, indispensable for computing instruction counts. He also modified the simulator program to accommodate our needs. F. Darema-Rogers and E. Melton provided assistance in the testing and execution of the parallelized versions of the code.

References

[1] W. L. Briggs, L. B. Hart, A. O'Gallagher, and R. A. Sweet, "Parallel FFT methods," preprint.

[2] E. Brooks, G. Fox, R. Gupta, O. Martin, S. Otto, and E. De Benedictis, *Nearest Neighbor Concurrent Processor,* CALT-68-867, 1981.

[3] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.,* vol. 19, pp. 297–301, Apr. 1965.

[4] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU ultracomputer—Designing a MIMD shared memory parallel machine," *IEEE Trans. Comput.,* pp. 175–189, Feb. 1983.

[5] A. Gottlieb, B. Lubachevsky, and L. Rudolph, "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors," *ACM TOPLAS,* vol. 5, pp. 164–189, Apr. 1983.

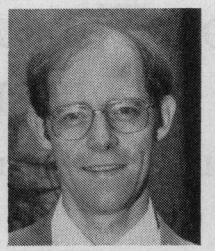
[6] A. Gottlieb and J. T. Schwartz, "Networks and algorithms for very large scale parallel computations," *Computer,* vol. 15, pp. 27–36, Jan. 1982.

[7] C. H. Jeon and A. P. Reeves, "The fast Fourier transform on a multicluster computer architecture," in *Proc. 19th Hawaiian Int. Conf. Syst. Sci.,* Jan. 1986.

[8] C. R. Jesshope, "The implementation of fast radix 2 transforms on array processors," *IEEE Trans. Comput.,* vol. C-28, pp. 20–27, Jan. 1980.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 02,2026 at 17:17:35 UTC from IEEE Xplore. Restrictions apply.

[9] D. Korn, *Timing Analysis for Scientific Codes Run Under the Washcloth Simulator,* Ultracomput. Note 24, Courant Instit. NYU, 1981.
[10] C. Kruskal, M. Snir, and A. Weiss, "On the distribution of delays in buffered multistage interconnection networks for uniform and nonuniform traffic," in *Proc. Int. Conf. Parallel Processing.* Computer Soc. Press, 1984.
[11] C. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," (extended abstr.), in *Proc. Int. Conf. Parallel Processing.* Computer Soc. Press, 1984.
[12] D. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.,* vol. C-24, pp. 1145–1155, 1975.
[13] A. Norton and A. Silberger, "Parallelization and performance prediction of the Cooley–Tukey FFT algorithm for shared-memory architectures," IBM Res. Rep. RC11237, July 1985.
[14] V. A. Norton and G. F. Pfister, "A methodology for predicting multiprocessor performance," in *Proc. Int. Conf. Parallel Processing,* 1985.
[15] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *J. Ass. Comput. Mach.,* vol. 15, pp. 252–264, Apr. 1968.
[16] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM research parallel processor prototype (RP3): Introduction and architecture," in *Proc. Int. Conf. Parallel Processing,* 1985.
[17] C. M. Rader, "FOUREA—A short demonstration version of the FFT," in *Programs for Digital Signal Processing.* New York: IEEE Press, 1979.
[18] G. Radin, "The 801 minicomputer," *IBM J. Res. Development,* vol. 27, pp. 237–246, May 1983.
[19] J. T. Schwartz, "Ultracomputers," *ACM TOPLAS,* vol. 2, pp. 484–521, Oct. 1980.
[20] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.,* vol. C-20, pp. 153–161, Feb. 1971.
[21] P. N. Swartztrauber, "Vectorizing the FFT," in *Parallel Computers,* G. Rodrigue, Ed. New York: Academic, 1982.



Alan Norton was born in Salt Lake City, UT, on August 20, 1947. He received the B.A. degree from the University of Utah, Salt Lake City, in 1968 and the Ph.D. degree from Princeton University, Princeton, NJ, in 1976, both in mathematics.

He was Instructor at the University of Utah from 1976 to 1979 and Assistant Professor at Hamilton College, Clinton, NY, from 1979 to 1980 before coming to IBM Research. Currently, he is a Research Staff Member at IBM, Yorktown Heights, NY, working on the Research Parallel Processing Prototype (RP3). His research interests include the performance analysis and architecture of parallel computer systems, parallel algorithms, fractals, and computer graphics.

Dr. Norton is a member of the Association for Computing Machinery and the American Mathematical Society.

Allan J. Silberger was born in York, PA, on August 24, 1933. He received the A.B. degree from the University of Rochester, Rochester, NY, in 1955 and the Ph.D. degree in mathematics from The Johns Hopkins University, Baltimore, MD, in 1966.

He has taught mathematics at Johns Hopkins, Bowdoin College, the University of Massachusetts, and Cleveland State University, where he is Professor of Mathematics. He has held visiting research positions at the Institute for Advanced Study, the University of Bonn, West Germany, the Thomas J. Watson Research Center of IBM, and the City University of New York Center for Large Scale Computation. During recent years he has studied digital signal processing algorithms and their implementations on modern computing systems.

Dr. Silberger is a member of the American Mathematical Society and the Mathematical Association of America.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on June 02,2026 at 17:17:35 UTC from IEEE Xplore. Restrictions apply.