Experimental Construction of a Fine–Grained Polyalgorithm for the FFT

Harry B. Hunt III
Computer Science Department
University at Albany
State University of New York
Albany, NY 12222

Lenore R. Mullin
Computer Science Department
University at Albany
State University of New York
Albany, NY 12222

Daniel J. Rosenkrantz
Computer Science Department
University at Albany
State University of New York
Albany, NY 12222

Abstract *We describe our recent research on the design and development of fine–grained polyalgorithms for FFTs. Fine–grained polyalgorithms, as introduced here, are an extension of the concept of polyalgorithms. They provide an efficient and uniform mechanism for generating a large number of alternative instantiations of an algorithm. This facilitates matching instantiations to particular problem sizes and computing environments. Algorithm efficiency and scalability in distributed/parallel computing environments require efficient use of the memory hierarchy at all levels, from cache through the interconnection network(s). The efficient solution of particular problems in scientific computing in different distributed/parallel computing environments may require several algorithms and algorithm variants, fine tuned to particular architectures and problem sizes.*

We identified various components of a single high level algorithm for one–dimensional radix 2 FFT, and constructed alternative implementations of each component. By combining an alternative for each component, we constructed a number of alternative distributed polyalgorithmic variants for the FFT itself. We then experimentally evaluated the performance of these variants to determine performance thresholds, based on problem size and number of processors. These experiments indicate which variant to use in which situations.

Keywords: polyalgorithm, FFT, scientific programming, distributed computing

1 Introduction

Algorithm efficiency and scalability in distributed/parallel computing environments requires efficient use of the memory hierarchy at all levels, from cache through the interconnection network(s). Consequently, the efficient solution of particular problems in scientific computing in different distributed/parallel computing environments may require several algorithms, each tuned to particular architectures and problem sizes. Skjellum, et. al. [4, 3] propose using a collection of algorithms, each of which is appropriate for a particular range of problem sizes. They use the term *polyalgorithm* to refer to such a collection of algorithms.

In addition, we perceive a need for more generality and flexibility in constructing polyalgorithms. Such generality and flexibility is needed to match implementations to *both* problem size and architecture. Not only can a given problem be solved by multiple algorithms, but in addition, a given high level algorithm can be implemented in many ways, especially in a distributed computing environment. We regard such a collection of implementations to be a *fine–grained* polyalgorithm.

To facilitate constructing such a fine–grained collection of implementations, a given algorithm can be envisioned as a simple and regular combination of high level operations on various algebraic structures. From this perspective, fine–grained polyalgorithms for such problems can be constructed by separately developing alternative implementations of high level operations and combining appropriate implementations together. This yields a large number of algorithmic variants that can be matched to problem size and computing environment. These variants can be compared experimentally and/or theoretically to obtain algorithms customized for problem, range of problem sizes, and computing environment.

A second possible advantage of fine–grained polyalgorithms is that classes of problems with suitably similar abstract algebraic specifications often share high level operations. Additionally, the problems in a class may have similar data flow and data access patterns. This makes it unnecessary to optimize the implementation of these operations separately for each problem in the class. In addition, a good polyalgorithm for one problem in the class can provide a good starting point for developing good polyalgorithms for other problems in the class. This should yield similar performance on various memory hierarchies.

Constructing a fine–grained polyalgorithm requires an evaluation of the performance of thepolyalgorithmic variants. In particular, during the development of each polyalgorithmic variant, experiments can be used to guide design decisions occurring during the implementation of the variant. These design decisions involve granularity of decomposition, implementation of basic high level algebraic operations, array distribution, array alignment, communications costs, cache utilization, etc.

In this paper, we outline our recent research on the design, development, and experimental evaluation of fine–grained polyalgorithms for the one–dimensional radix 2 FFT. The methodology, experiments, and decisions used are applicable to polyalgorithms for arbitrary fixed radix FFTs. In the research presented here, we discuss a number of fine–grained polyalgorithm variants. Each of these variant implementations was derived from the single basic *high–level* algorithm for the radix 2 FFT in [5]. (See Figure 1.) These variant implementations were experimentally compared by running a consistent set of experiments on the SGI Cray Origin 2000. [1]

We systematically explored design variants of *bit–reversal*, *weight* computations, *butterfly* computations, *processor allocation strategies*, *message generation* mechanisms, programming language, and coding styles. The coding styles included a *monolithic–vector* style (i.e., a style where statements operate on entire vectors) and a *scalarized* style.

Implementations were written in

- Fortran90 using sequential code (a monolithic–vector version and a scalarized version),
- Fortran90 using explicit programmer specified MPI calls for passing messages between processors (both monolithic–vector versions and scalarized versions),
- HPF (High Performance Fortran) [2] using directives to specify both initial and subsequent distributions of arrays, and letting the compiler generate necessary messages between processors (both monolithic–vector versions and scalarized versions).

The several dialects of Fortran, together with MPI, were chosen for implementations for the following reasons:

- Fortran is still the language of choice for scientific numerical computation, and Fortran90 and HPF are the main languages for such computation in distributive/parallel computing environments.

[1]This work was partially supported by NSF and NCSA under CCR–9596184 and utilized the SGI Origin 2000 at the National Center for Supercomputing Applications, University of Illinois at Urbana–Champaign.

- A number of high–level monolithic–array operations occur in Fortran90/HPF including *generalized transpose* and *reshape* (the bit reversal and distribution/redistribution occurring in the FFT are naturally and directly expressible in terms of these two monolithic array operations).

- HPF has distribution and redistribution directives for automatic array distributions among the different processors of a specified processor topology.

- As demonstrated by our experimental results, efficient message handling including *message bundling* is often not adequately handled by current state–of–the–art compilers. As a step towards ensuring efficient message handling, we investigated the use of explicit MPI calls.

2 Construction of FFT Polyalgorithmic Variants

Our design and development of a fine–grained polyalgorithm for the FFT began with Van Loan's [5] high–level Matlab program for the *radix 2* FFT, shown in Figure 1.

In Line 1, P_n is a permutation matrix that performs the bit–reversal permutation on the n elements of vector x. In Line 6, the n element vector x is regarded as being *reshaped* to be a $L \times r$ matrix consisting of L rows, each of which is a vector of r elements. Line 6 can be viewed as partitioning the L rows of this matrix into $L/2$ pairs of vectors, and doing a *butterfly* computation that combines the two vectors in each pair to produce a vector with $2r$ elements.

By direct inspection of the above program, one can identify the following five basic modules for the *radix 2* FFT, given below. These five modules are appropriate components of arbitrary radix FFTs. More generally, analogues of these modules are appropriate for the wider class of FFT–like transforms [1].

1. the computation of the bit–reversal permutation (Line 1),

Input: x in C^n and $n = 2^t$, where $t \geq 0$ is an integer.
Output: The FFT of x.

$$
\begin{array}{llr}
x \leftarrow P_n\, x & & (1) \\
\text{for } q = 1 \text{ to } t & & (2) \\
\quad \text{begin} & & (3) \\
\qquad L \leftarrow 2^q & & (4) \\
\qquad r \leftarrow n/L & & (5) \\
\qquad x_{L\times r} \leftarrow B_L\, x_{L\times r} & & (6) \\
\quad \text{end} & & (7)
\end{array}
$$

Here, P_n is a $n \times n$ permutation matrix, $B_L = \begin{bmatrix} I_{L_*} & \Omega_{L_*} \\ I_{L_*} & -\Omega_{L_*} \end{bmatrix}$, $L_* = L/2$, and Ω_{L_*} is a diagonal matrix with values $1, \omega_L, \ldots, \omega_L^{L_*-1}$ along the diagonal, where ω_L is the L'th root of unity.

Figure 1: High–level program for the radix 2 FFT

2. the computation of the complex weights occurring in the matrices Ω_{L_*},

3. the *butterfly* computation that, using the weights, combines two vectors from x, producing a vector with twice as many elements (Line 6),

4. alternative strategies for allocating the elements of the reshaped array x (line 6) to the processors for use in the distributed implementations of the computation loop (Lines 2 through 7), and

5. the generation and bundling of messages involving values from x (Lines 2 through 7).

We describe both the alternatives implemented and several of the issues that complicate these implementations.

Two variants of the *bit–reversal* module were implemented. In the first, this permutation of the input values is done at a master processor, and then the elements are distributed among all the processors. In the second, the elements are distributed among the p processors based on the low order $\log p$ bits of their indices [2], and the remaining part of the bit reversal permutation is performed in parallel with no communication of information between processors.

Two alternatives for the weight computation module were implemented. In the first, the weights are computed by a master processor, and these values accessed as needed by the other processors, with a HPF compiler determining message bundling for sending the weights. In the second, each processor computes all the weights in parallel, so there is no interprocessor communications due to the weights.

Two alternative coding styles for the butterfly computation module were implemented. In the first, the computation is scalarized. In the second, the computation is expressed in terms of (monolithic) vector operations.

Two alternative processor allocation strategies were used. In both strategies, the initial iterations of the loop (Lines 2 through 7) used a mapping of the reshaped rows of x to the processors, such that each pair of vectors to be combined in a butterfly computation are co–located on the same processor. However, at a certain crucial iteration, each processor has only a single large vector, so the two vectors to be combined in each butterfly computation are on different processors. The two allocation strategies differ in how they proceed at this iteration. The first strategy ensures that whenever two vectors are combined, they are co–located on the same processor; as a consequence, the number of active processors is cut in half in each of the last few iterations. The second strategy completely redistributes the data among all the processors at the crucial iteration, and then subsequently combines values co–located on the same processor, thus requiring no interprocessor communication for subsequent iterations. We call the first strategy *decreasing–processors*, and the second *all–processors*.

[2]In terms of HPF directives, this can be specified using a *cyclic* distribution directive.

Two alternatives for message generation were implemented. In the first, the HPF *redistribute* directive is applied to x whenever the distribution of x should change. The use of the *redistribute* directive causes the compiler to automatically generate code for the appropriate messages, to move the data. The compiler determines how the data is bundled into messages. In the second, messages involving subarrays of x are explicitly constructed and sent using MPI. Each data transfer between a given pair of processors is contained in a single message that bundles *all* the data that needs to be sent from the first processor to the second.

3 Experimental Results

Experiments were run on an SGI Origin 2000 multiprocessor, using the SGI Fortran90 compiler and the PGI (Portland Group International) HPF and Fortran90 compilers. (These were the only Fortran90 and HPF compilers that we had access to at NCSA.) These experiments revealed that there are both compile–time and run–time restrictions on the maximum problem size (i.e. the number of elements n in vector x) that can be handled (without resorting to "tricky" coding). The compile–time restrictions are on the maximum allowable range of values of a given dimension in an array declaration. On a single processor, PGI's HPF compiler could not handle programs with vectors of size exceeding 2^{16}. In contrast, the PGI and SGI Fortran90 compilers can handle programs with vectors of sizes up to 2^{30}. Attempts to declare vectors with lengths greater than these limits generated compiler errors. In addition to the above compile–time restrictions, we encountered a run–time limit at NCSA of at most 1,024 messages per processor in some queue. Consequently, in our experimental environment, the maximum problem size n for which we could systematically obtain performance results for 8 processors was 8,192. Furthermore, experiments showed that given this limitation on problem size, there was no performance advantage in using more than

8 processors.

Our Fortran90 with MPI programs are independent of the problem size n (for $n \leq 2^{30}$, the compiler limit) and number of processors, without the need for special case code.

We conducted a consistent set of experiments on sixteen variant FFT programs. Each set of experiments involved seven values of problem size n (powers of 2 from 128 to 8,192) and three values for the number of processors p (2, 4, and 8). For each combination of n and p, we executed the program three times, and averaged the performance results.

The performance of certain of the programs were essentially identical over this range of experiments. On the basis of these performance evaluations, the sixteen programs fell into nine equivalence classes. These classes are the following:

1. Fortran90 *scalarized* code with *explicit control of parallelization and messages via MPI*, the *all–processors* allocation strategy, and *parallel bit reversal.*

2. Fortran90 *monolithic* code with *explicit control of parallelization and messages via MPI*, the *all–processors* strategy, and *parallel bit reversal.*

3. Fortran90 *monolithic* code with *explicit control of parallelization and messages via MPI*, the *all–processors* strategy, and *non–parallel bit reversal.*

4. Fortran90 *scalarized* code with *explicit control of parallelization and messages via MPI*, the *all–processors* strategy, and *non–parallel bit reversal.*

5. (a) Fortran90 sequential *monolithic* code, parallelized by the SGI compiler [3].
   (b) Fortran90 sequential *scalarized* code, parallelized by the SGI compiler.
   (c) Fortran90 *monolithic* code with *explicit control of parallelization and messages via MPI*, the *decreasing–processors* strategy, and *parallel bit reversal.* (The performance of this code was the same using the SGI and PGI compilers.)
   (d) Fortran90 *monolithic* code with *explicit control of parallelization and messages via MPI*, *decreasing–processors*, and *non–parallel bit reversal.* (The performance of this code was the same using the SGI and PGI compilers.)
   (e) Fortran90 *scalarized* code with *explicit control of parallelization and messages via MPI*, *decreasing–processors*, and *parallel bit reversal.* (The performance of this code was the same using the SGI and PGI compilers.)
   (f) Fortran90 *scalarized* code with *explicit control of parallelization and messages via MPI*, *decreasing–processors*, and *non–parallel bit reversal.* (The performance of this code was the same using the SGI and PGI compilers.)

6. HPF *monolithic–vector* code with *directives for initial parallelization*, *non–parallel bit reversal*, *decreasing–processors*, and *explicit redistribution.*

7. (a) HPF *scalarized* code with *directives for initial parallelization*, *non–parallel bit reversal*, and *no explicit redistribution.*
   (b) HPF *scalarized* code with *directives for initial parallelization*, *non–parallel bit reversal*, *decreasing–processors*, and *explicit redistribution.*

8. HPF *monolithic–vector* code with *directives for initial parallelization*, *parallel bit reversal*, *decreasing–processors*, and *explicit redistribution.*

9. (a) HPF *scalarized* code with *directives for initial parallelization*, *parallel bit*

[3] The PGI Fortran90 compiler does not do automatic parallelization.

reversal, and *no explicit redistribution.*

(b) HPF *scalarized* code with *directives for initial parallelization*, *parallel bit reversal*, *decreasing–processors*, and *explicit redistribution.*

There is an ordering of these classes, from most efficient to least efficient, consisting of classes 1, 2, 3, 5, 6, 7, 8, 9. Class 4 is between classes 1 and 5 in the ordering, but is incomparable with classes 2 and 3. In some experiments, class 4 was more efficient than class 2, and in other experiments it was less efficient than class 3.

Some of the experimental results comparing alternative polyalgorithmic variants are shown below. Figure 2 shows a class 5 variant using explicit parallelization via MPI outperforming a class 7 variant that instead uses HPF and directives. Figure 3 compares the performance of a class 4 and a class 5 variant that differ in the processor allocation strategy, with the all processors strategy outperforming decreasing processors. Figure 4 compares the performance of class 1 and class 4; these variants differ in bit reversal strategy, with parallel bit reversal outperforming non–parallel. Figure 5 compares the performance of class 5 and class 9, with code produced by the SGI parallizing Fortran90 compiler outperforming a HPF variant with decreasing–processors and parallel bit reversal. Figure 6 compares the performance of class 7 and class 9. These variants both use HPF and the decreasing–processors, but differ in bit reversal strategy, with non–parallel bit reversal outperforming parallel bit reversal.

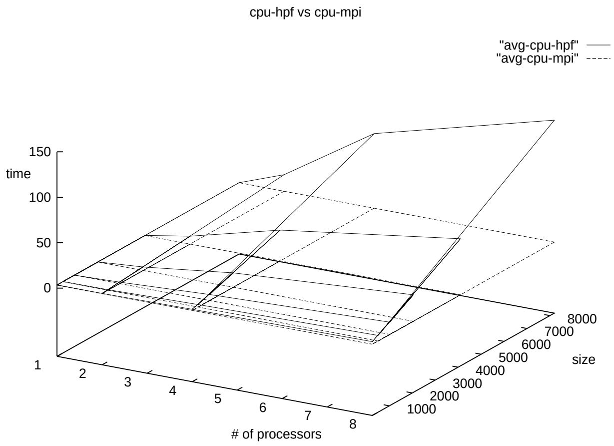

Figure 2: Class 5(c) (avg-cpu-mpi) versus Class 7(a) (avg-cpu-hpf): MPI versus HPF

References

[1] Douglas F. Elliott and K. Ramamohan Rao. *Fast Transforms: Algorithms, Analyses, Applications.* Academic Press, Inc., 1982.

[2] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[3] J. Li and A. Skjellum. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies, 1995. Mississippi State University.

[4] A. Skjellum, N. Doss, and P. Bangalore. Driving issues in scalable libraries: Poly-algorithms, data distribution independence, redistribution, local storage schemes. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing.* SIAM Press, 1996.

[5] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform.* Frontiers in Applied Mathematics. SIAM, 1992.

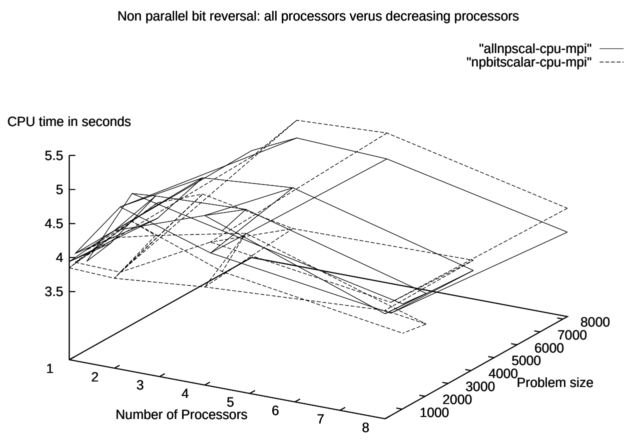

Figure 3: Class 4 (allnpscal-cpu-mpi) versus Class 5(f) (npbitscalar-cpu-mpi): All Processors versus Decreasing Processors

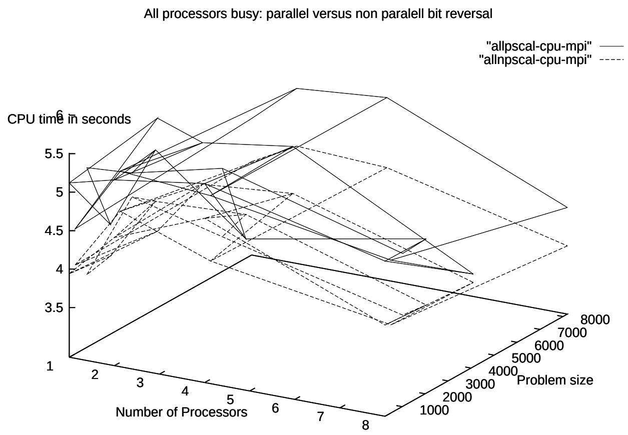

Figure 4: Class 1 (allpscal-cpu-mpi) versus Class 4 (allnpscal-cpu-mpi): All Processors: Parallel versus Non–Parallel Bit Reversal

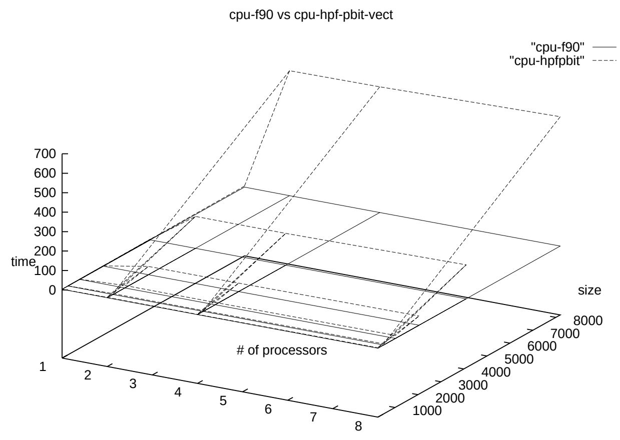

Figure 5: Class 5(b) (cpu-f90) versus Class 9(b) (cpu-hpfpbit): Parallized Fortran90 code versus HPF with Decreasing–Processors and Parallel Bit Reversal

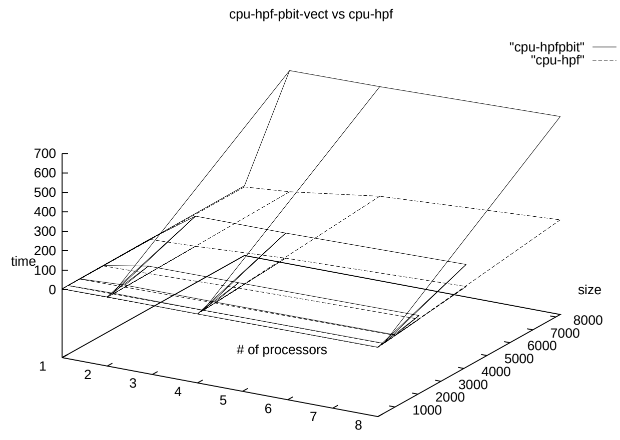

Figure 6: Class 7(b) (cpu-hpf) versus Class 9(b) (cpu-hpfpbit): Scalarized HPF with Decreasing–Processors: Non–Parallel versus Parallel Bit Reversal