Improving Algorithm Performance Through Monolithic Array Analysis and High Performance Scientific Computing Experiments*

Lenore R. Mullin and Sharon G. Small
lenore@cs.albany.edu, small@cs.albany.edu
Department of CS, University at Albany, SUNY
Albany, New York 12222

September 13, 2000

Abstract

High Performance Scientific computing depends on using the fastest algorithms given any architectural platform. The complexity of advanced architectures and networks makes an experiment only search impractical due to numerous variants in both the environment and algorithm. We propose the use of monolithic array analysis as a way to decrease the search space for the fastest algorithm by analyzing an algorithm's memory access patterns and parameterizing as much of the environment as possible. The FFT is the focus of our initial investigations.

Introduction

High Performance Scientific Computing research addresses compute and time intensive scientific algorithms running on advanced multiprocessor architectures. The complexity of advanced architectures makes a theoretical analysis virtually impossible without complementing the analysis with experimentation. Experiments in this type of research involve precise tuning of systems, and an in depth evaluation of an algorithm's performance as it relates to a system's given configuration. Our research is guided by a formal theoretical framework for monolithic array analysis[6, 4]. It is used to define, build, and analyze an algorithm's software. In contrast, an experimental only framework may lead to a combinatorial search explosion given many algorithms, problem sizes, and software/hardware environments.

Advanced architectures pose unique and challenging problems for algorithm designers and implementors, due to their non-uniform memories, multiple processors and diverse interconnection networks. Through monolithic array analysis we hope to identify memory access patterns, determine how many variables are needed to build the algorithm, and identify

*This work was partially supported by NSF CCR–9596184.

message size and their mapping to processors[7]. Furthermore, we can show how algorithms scale to shared or distributed memory machines.

In order to identify the best algorithm for a particular platform many variants must be identified. Specifically, the environment, i.e. a system's hardware and software, may vary in many ways. Hardware variations can include the manufacturer of the system, number of processors and their speed, size and speed of cache(s), size and speed of real, virtual, and shared memory or disk space, type of I/O bus and interconnection network(s). Software variations can include the operating system, it's version and configuration, page size, message passing paradigm(MPI,PVM), compilers, etc. The algorithm may also vary. The same algorithm may be implemented many different ways, and of course the language itself and the compiler can also vary. Experiments may validate theoretical predictions or they may indicate ways to adjust an environment, e.g. change the page size, given the memory access patterns of the algorithm.

This paper will first present the radix two Fast Fourier Transform and discuss implementation issues. We will describe our experimental environment, and the scripts used to run these experiments. Then, we give a preliminary analysis of the algorithm and its performance using local, shared, and distributed memory. Finally, we discuss how to generalize the radix two algorithm to any radix, and indicate how this generalization can enhance performance of FFTs.

The Fourier Transform

The Fourier Transform is based on Gauss' theory that any periodic wave may be represented as a summation of a number of sine and cosine waves of different frequencies. The Fourier Transform computes the Fourier coefficients needed for this summation. This transform takes samples of a signal over time and produces the Fourier coefficients, allowing us to create a signal's representation over a frequency variant. In its original formulation the Fourier Transform has $\mathcal{O}(n^2)$ complexity. Our monolithic analysis begins with this. By analyzing the sparse matrix multiplication and realizing that only the diagonals of the blocks are used to decompose the matrices, a faster $\mathcal{O}(n\ log\ n)$ formulation is possible. Characterizing the generalization of the decompositions for other radices allows us to formulate fast transforms for any radix. The radix two Cooley-Tookey was the first fast formulation using this approach.

The FFT is so important that it was named one of the Top Ten Algorithms of the 20th century[2]. Minimizing space and time is a requirement of the FFT and other transforms used in real-time signal and image processing. The search for the best FFT algorithm requires that each algorithm be broken down into its constituent sub-algorithms: i.e. the bit reversal and butterfly in a sequential environment. The butterfly can be implemented simply as two scalars in and two scalars out, or as a vector butterfly in and a vector butterfly out, as illustrated in Figure 1.

```
      do i_=0,n-1,Lnew
c          do j_=0,L-1
c                   c = ww(j_)*z(i_+j_+L)
!
! What follows is the typical butterfly operation.
c          z((/ i_+j_ , i_+j_+L /)) = (/ z(i_+j_)+c , z(i_+j_)-c /)
! A possible optimization could occur if the butterfly were
! vectorized. The following fragment illustrates this
! Note that optimization through monolithic analysis
! will remove the temp below.
         temp = z
         z(i_:L-1) = temp(i_:L-1)+(ww*temp(i_+L:L-1))
         z(i_+L:i_+L+(L-1)) = temp(i_:L-1)-(ww*temp(i_+L:L-1))
!
c       end do
         end do
```

Figure 1: Scalar butterfly and vectorized butterfly of the FFT

Related Work

The FFTW[3] group at MIT, the ATLAS[13] experiments led by Dongarra, and the TPL compiler[5] are driven only by experiments. Their *experiment only* approach may lead to a combinatorial explosion of experiments. In a sequential environment the search space is large but not so large as to prohibit the outcome of the search. However, when a shared or distributed memory environment is explored, the number of variants and subsequent complexity increases. Furthermore, there is no concept of using a relational database to store previous experiments. Consequently, new experiments are required even for the same problem size, algorithm, and environment. While their results are beneficial and meet the immediate needs of computational science now, we hope our combined theoretical and experimental approach will contribute to ways compilers and operating systems can automatically optimize a given algorithm on diverse multiprocessor architectures.

Monolithic Array Analysis of the FFT

Our analysis begins with the radix two Cooley-Tookey FFT which theoretically generalizes to any radix[11, 12, 6]. By generalizing we can investigate if radix four or eight performs better than radix two. Programs are built using a hand derived radix two only design(*fftf90*) and a general radix design(*fftgen*) with radix equal to two. The differences between *fftf90* and *fftgen* are illustrated in Figure 2. Initially, our experiments use local memory on a single processor. We do this as a control for our shared and distributed memory experiments.

```
!! Note how the code changes to support
!! a radix other than two. Initially c was a scalar,
!! for radix two. For radix equal to n c is a vector
!! with length equal to radix-1.
!!                  c = ww(j_)*z(i_+j_+L)
!!         z((/ i_+j_ , i_+j_+L /)) = (/ z(i_+j_)+c , z(i_+j_)-c /)
              c(start) = z(i_+j_)
              ebase(start) = \begin{figure}[t]
              do k_=start+1,base-1
                 c(k_) = ww(j_*k_)*z(i_+j_+(k_*L))
                 z(i_+j_) = z(i_+j_)+c(k_)
                 ebase(k_)=EXP((-2*pi*i)/base)**k_
              end do
!              print *,c
! What was the butterfly is now a loop that iterates base-1 times
!
              do k_=start+1,base-1
                 temp = c(start)
                 do l_=start+1,base-1
                    temp = temp+c(l_)*ebase(modulo((k_*l_),base))
                 end do
                 z(i_+j_+(k_*L))=temp
              end do
! The loop ends here
!
```

Figure 2: Fragment from derived radix two and generalized radix FFT

The algebra used to formulate the sequential design is also used to formulate the distributions and subsequent mapping to processors[8]. This formulation provides a way to algebraically transform the high level discrete design to a *normal form* that describes how to build the sequential, shared, and distributed memory programs. The *normal form* denotes a generic form independent of the programming language or message passing paradigm(MPI). The *normal form* also characterizes message size, number of variables, and communications patterns. Furthermore, the regularity of the memory access patterns coupled with problem size and memory size allow us to attempt an analysis of a fast transform's behavior as it moves through a particular memory hierarchy.

Experimental Environment

Hardware

We custom built two separate systems, Einstein and Hawking. Einstein is a SUN1000, with two 60 MHz processors, 128MB of shared memory and two internal one gigabyte drives. It also has one external nine gigabyte drive connected by an SBUS. There is no local memory. Einstein has 384 MB of disk space dedicated to swap, and its pagesize is 4096 bytes. Hawking is a SUN20 with a single 50 MHz processor and 64 MB of memory. It has a single internal one gigabyte drive and one external one gigabyte drive connected by an SBUS. Hawking has 256MB of disk space dedicated to swap, and its pagesize is 4096 bytes. Our communication network between Einstein and Hawking consisted of a 155Mbs FORE ATM and a 100Mbs Fast Ethernet.

Software

Both machines were completely cleared of any pre-existing software and all files prior to OS installations. Although we used a general purpose operating system, Solaris(2.7), we did not install software that might use machine cycles, e.g. sendmail, tcpwrappers or netscape. On the external drives we loaded Sun's Fortran 90 compiler, Perl 5.004, MPI 1.2.0, and FORE 4.1.1. There were no network mounted files.

For sequential and shared memory multiprocessing, we chose F90 for its monolithic expressiveness and our ability to optimize monolithic array expressions[9]. We are aware that Fortran 77 outperforms F90 in some situations. But, with continued compiler research and development[10], monolithic optimizations will be the norm. For shared memory experiments we used Sun's F90 parallelizing compiler. For distributed processing we used F90 with MPI.

Experiments

Programs used in experiments resided on and were executed from the external drives. Experimental controls were essential in order to guarantee reproducible results[1]. Our experiments assumed a dedicated environment because issues such as user load or quantum size could interfere with our analysis. Each time we ran an experiment, we recorded environmental information, such as the operating system and its version, compiler used, page size, the number of processors and their speed, the size of memory, cache, real and virtual, and whether there was shared memory and how much.

Experiments Driven by Scripts

In order to understand how an algorithm performs on a particular platform it is essential to vary the problem size using small enough increments to observe memory behavior. Size should start very small, to observe cache behavior, and grow to very large, to observe paged virtual memory behavior. This can only happen if the real memory is small.

Different compilers and flags may also be required. The variants could be enormous. Consequently, editing and running each experiment could take an enormous amount of time if done manually. Hence, mechanization through scripts, to modify, run and record performance was required. Since we were not using network mounted files, MPI programs had to be compiled separately on each machine. The scripts had to accommodate this. We also felt that experiments should be guided by the user not by scripts only. Hence, interactability is desired.

Scripts were written in Perl. They query the user to enter all variant information: size of problem, amount of shared memory, compiler, compiler flags, time command and path, and number of processors. The vector size must be specified by indicating a start size, stop size, as well as an increment, the radix, program name, and how many times(recommended minimum of three) to execute each instance before taking an average. New variants can easily be added to the script as needed. Following this input, the script would create a parameter file instantiating all input variables. Then, it would compile and run the chosen program(s) on designated machines. When complete the averaged times would be outputted to a file in a format compatible with gnuplot graphing. Performance is then visually analyzed to detect anomalous results. In these cases we rerun the experiments, to verify that the anamolies were reproducible.

Algorithm and Environment Analysis

We ran our programs for a wide range of vector sizes on single and multiple processor shared and distributed memory machines. Vector sizes are powers of the radix. We ran the experiments for vector sizes from 2^2 through 2^{24}. We ran each of the programs, *fftf90* and *fftgen*, for each of these vector sizes a minimum of three times on Einstein using a single processor, and again using two processors. We repeated the entire procedure on Hawking using it's single processor. When we utilized two processors on Einstein this was done during compilation by supplying the -parallel flag to the F90 compiler. This is combined with the environment variable PARALLEL set to two. The PARALLEL variable tells the compiler the number of processors and -parallel flag compiles with optimizations for shared memory multi-processing.

We made performance predictions prior to running the experiments, based on our understanding of the algorithms and our analysis of the given machines memory hierarchy. We predicted that Einstein's 60 MHz single processor experiments should run slightly faster than Hawking's 50MHz experiments, up to a threshold vector size. We predicted that there would be a point where the vector's size would be large enough to affect machine performance based on the limitations of its swap space size dedicated to virtual memory. A threshold point, where the vector would become so large, virtual memory would be unable to handle it, causing numerous page faults and severe performance issues. We predicted that Hawking with 64 MB of main memory and 256 MB of swap would reach this point sooner than Einstein with its 128MB of main memory and 384MB of swap space. Finally, we predicted that Einstein's dual processor experiments should show significant improvement over it's single processor, with some limitations. We also believed that distributed computation would outperform virtual memory use.

Evaluation of Results

Number of Processors

The results of the experiments solidly confirmed our predictions. On Einstein, the *fftf90* and *fftgen* two processor experiments both showed significant improvement over the one processor times, up until the threshold vector size 2^{24}. Figure 3 shows the predictable behavior of the programs up until this threshold.

Notice the exponential behavior of every experiment for 2^{24} vector size; 2^{23} is the last size where a time can be seen plotted on the graph. As the vector size increased the memory access patterns of the bit reversal caused continuous cache misses and subsequent page faults.

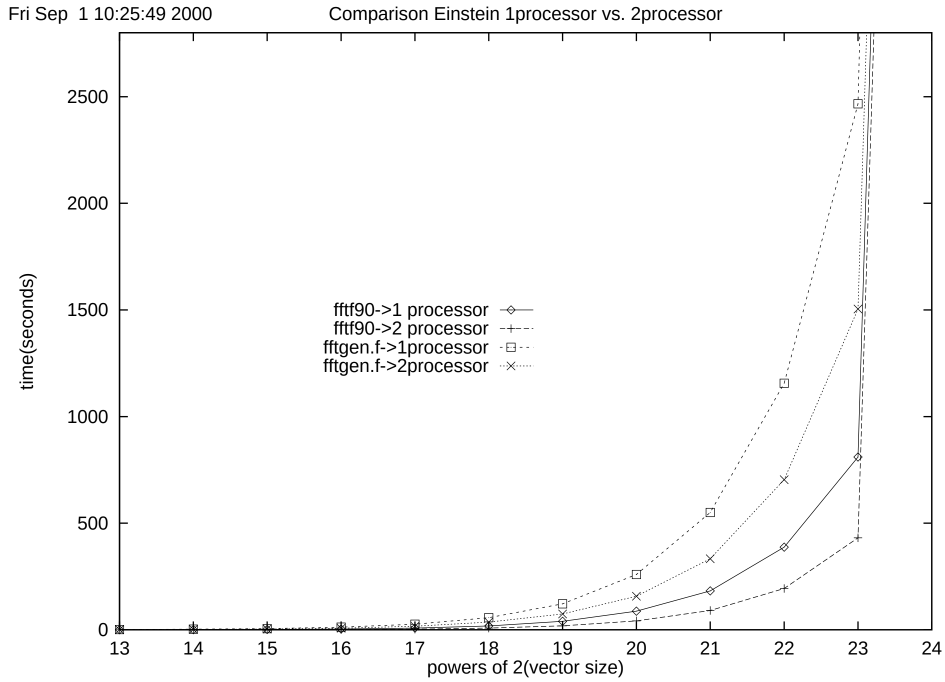

Figure 3: Comparison of Einstein 1 processor vs. 2 processor performance

Beyond the obvious exponential increase in time for all experiments, it is important to note that the percent of improvement of time from one processor to two processors diminishes at this larger size. Table 1 shows this by providing the exact times in seconds, given the vector sizes 2^{18} through 2^{24}, for the *fftf90* experiments. The average decrease in execution time when a second processor was added is 51.2% for vector sizes 2^{18} through 2^{23}. The results show that for vector size 2^{24} the cost of adding a second processor is not justifiable with only a 6.5% decrease in execution time, since execution went virtual.

Table 1: Improvement of Run Times for 1 processor vs. 2 processor

Vector size	One Processor	Two Processors	Percent Decrease in time
2^{18}	18.2	8.2	54.9
2^{19}	40.1	18.9	52.9
2^{20}	87.6	41.7	52.4
2^{21}	182.5	90.4	50.5
2^{22}	387.9	193.7	50.1
2^{23}	810.1	431.3	46.8
2^{24}	12235.1	11439.2	6.5

Speed of Processors

We compared the results of the experiments between Hawking's 50MHz processor and Einstein's 60 MHz processor. Recall that Einstein also had a larger virtual memory, with its 128MB of sharable main memory and 384MB of dedicated swap space, whereas Hawking was limited with its 64MB of main memory and 256MB of dedicated swap space. Figure 4 confirms our predictions of Einstein's slight performance improvement over Hawking up to a threshold vector size. As predicted, Hawking reached it's threshold,2^{23} sooner than Einstein because of Einstein's larger real memory.

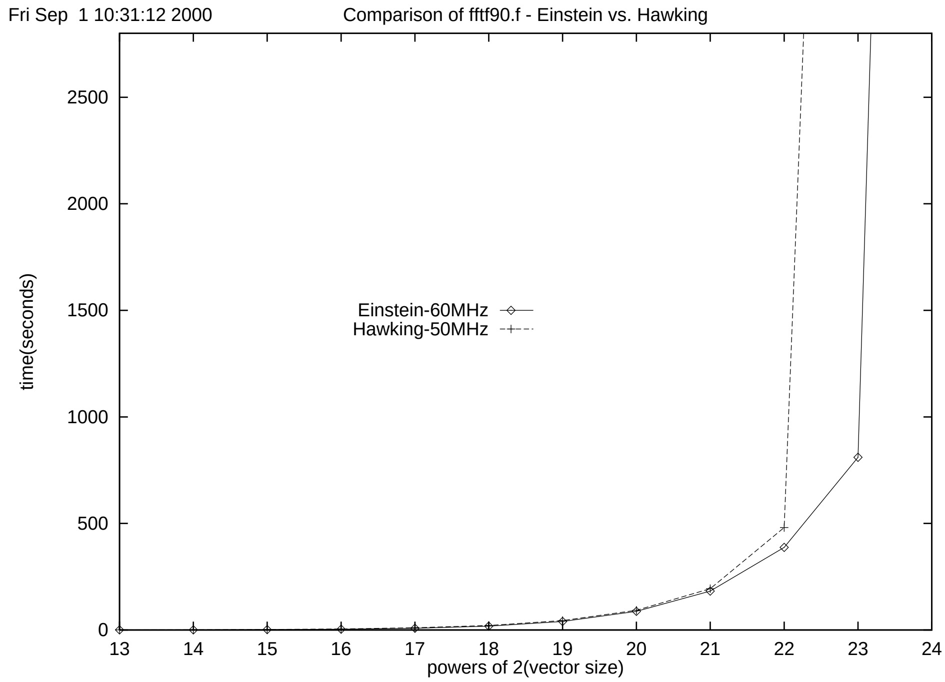

Figure 4: Comparison of Einstein vs. Hawking performance

It is important to note that the percent improvement in processor speed slowly diminishes until they begin to reach Hawking's threshold, as shown in Table 2. This is due to the fact that they have both begun to use virtual memory. As they both begin to use virtual memory the difference in processor speed becomes less of a factor, as more of the total execution time is composed of virtual memory processing, up until the point where Hawking runs out of virtual memory, 2^{23}.

Table 2: Comparison of run times for 60MHz vs. 50MHz processor			
Vector size	60MHz Processor	50MHz Processor	Percent Increase in Time
2^{14}	0.4	0.6	33.3
2^{14}	0.8	1.19	32.8
2^{15}	1.7	2.43	30.0
2^{16}	3.8	4.96	23.4
2^{17}	8.2	10.4	21.2
2^{18}	18.2	21.5	15.3
2^{19}	40.1	44.7	10.3
2^{20}	87.6	96.7	9.4
2^{21}	182.5	193.5	5.7
2^{22}	387.9	480.9	19.3
2^{23}	810.1	9246.6	91.2

Optimization

In Figure 5 we observe *fftgen's* optimized performance. By analyzing away the temps, performance for the general radix instantiated for radix two is competitive with the radix two only design, *fftf90*. We also note that monolithic analysis can optimize away the large static arrays used within our programs. These are further reductions based on index permutations and correlations of local addresses relative to their original global address. That is, we continue to refine our designs based on the same algebraic approach as before.

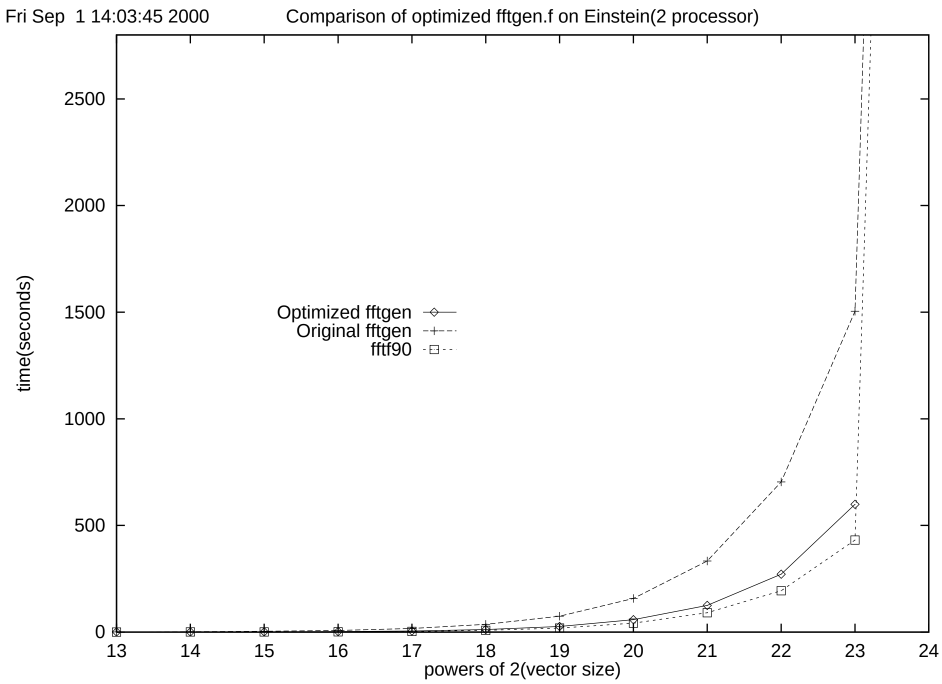

Figure 5: Comparison of Optimized fftgen.f performance

Distributed Algorithms

Figure 6 illustrates the performance of our two distributed memory designs *decproc* and *allproc*. In the distributed memory design*decproc*, the vector is *block* distributed to the two processors available, i.e. to Hawking and Einstein. For *allproc*, there is an initial cyclic distribution. Based on the vector size and subsequent partition size, communication is not necessary until the loop count of the butterfly operation reaches the logarithm of the partition size. For example, given a vector of length sixteen and two processors, the partition size is eight. In total there are four iterations of the butterfly and communication does not begin until the iteration count is three. This is also true for *allproc*. However, in *decproc* the butterfly is preceded by communications. The message size is doubled and the number of processors is halved. It was conjectured that for two processors *decproc* would outperform *allproc* because the distribution was *block* thus ensuring contiguity in the memory access patterns. For *allproc*, with an initial *cyclic* distribution, the memory access patterns are not contiguous. Furthermore, since we are only using two processors the amount of communications is the same. Upon completion of the butterfly step *allproc* must do a final *redistribution* thus performing one more communication step. Generally, *allproc* should perform better when more processors are used since a number of steps will take place without communications. Interestingly, for this one case, a different strategy for communications, i.e. *decproc*, could be used instead of *allproc*, the more traditional approach, thus confirming our initial predictions. In Figure 6 this performance is apparent when the vector size exceeds the local memory size, i.e. for vector sizes greater than 2^{20}. We do not observe much difference before that since all data is within local memory and memory accesses are referenced efficiently by cache. Because our local memory is so small MPI runs out of buffer space, thus constraining our MPI experiments to 2^{22}.

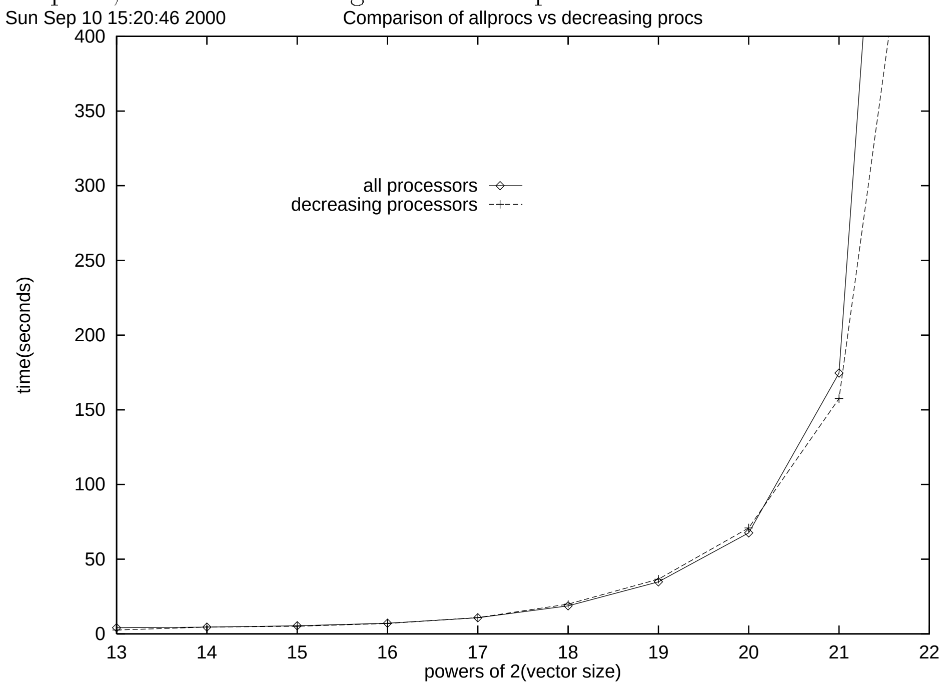

Figure 6: Comparison of all processors versus decreasing processors

Conclusions

Running sequentially on Hawking, the bit reversal and the *fftf90* program spends so much time swapping that the performance differential between 2^{22} and 2^{23} appears exponential graphically. For 2^{22}, the *fftf90* program runs for *three hours*, for 2^{23} it runs for *three days*. This is a threshold point that indicates the need to either add more real memory or move to distributed processing. As expected, doubling the memory on Einstein allowed us to increase the vector by a power of two as illustrated in Figure 4. In Figure 5 we observe how *fftgen*, the generalized radix program, performs on Einstein's two processors. Observe that the maximum vector remains the same as in Figure 4, but the time is halved when we double the processors. Optimizations possible for *fftgen* indicate a slight overhead due to generalization. Our sequential and shared memory designs scale to distributed message passing. Unfortunately our distributed memory designs were not competitive. Adding more memory is indicated to increase performance. We attribute this to the slow SBUS and network.

Presently, our research is not meant to compete with performances achieved by the FFTW and ATLAS group. It is however meant to show a systematic way of analyzing memory access patterns and subsequent performance of the FFT on one dimensional radix two arrays.

Future Research

We plan to analyze how vectors with length 2^n compare to 4^m, 8^p, such that $2^n = 4^m = 8^p$, using the the general radix sequential program, *genrad*. We will run and compare the same vector sizes using Hawking and Einstein. We will also generalize *allproc* and *decproc* to handle all radices and compare their distributed performance.

We have already begun experiments with more radices using *fftgen*. We are currently focusing on comparisons between radix two, four, eight and sixteen. Presently, radix eight performs best in our environment. We want to understand why and build this into our monolithic model.

We hope to investigate ways to store experimental results in a relational database to eliminate the need to re-run experiments. Also, we plan to install Solaris(2.7) on two Intel PCs each with a single 200 MHz processor. Our reason for this is to keep the operating system the same, and vary the architectural platform. By doing this we hope to observe performance, then parameterize that architecture. These and current results will bring us closer to a general monolithic model of performance.

References

[1] D. Baird. *Experimentation: An Introduction to Measurement Theory and Experiment Design.* Prentice-Hall, 1962.

[2] C. M. H. Center. Top ten algorithms of the 20th century. *Computing Science and Engineering Magazine*, 1999.

[3] M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of ICAASP 98*, 1998.

[4] H. B. Hunt III, L. Mullin, and D. J. Rosenkrantz. Towards an indexing calculus for efficient distributed array computation. Technical Report 97–4, University at Albany, Department of Computer Science, 1997.

[5] J. Johnson, R. Johnson, D. Padua, and J. Xiong. Searching for the best fft formulas with the tpl compiler. In *13th International Workshop on Languages and Compilers for Parallel Computing 2000*, 2000.

[6] L. M. R. Mullin. *A Mathematics of Arrays.* PhD thesis, Syracuse University, Dec. 1988.

[7] L. R. Mullin, D. Dooling, E. Sandberg, and S. Thibault. Formal methods for scheduling, routing and communication protocol. In *Proceedings of the Second International Symposium on High Performance Distributed Computing (HPDC-2).* IEEE Computer Society, July 1993.

[8] L. R. Mullin, D. Eggleston, L. J. Woodrum, and W. Rennie. The PGI–PSI project: Preprocessing optimizations for existing and new F90 intrinsics in HPF using compositional symmetric indexing of the Psi calculus. In M. Gerndt, editor, *Proceedings of the 6th Workshop on Compilers for Parallel Computers*, pages 345–355, Aachen, Germany, Dec. 1996. Forschungszentrum Jülich GmbH.

[9] D. Rosenkrantz, L. Mullin, and H. B. H. III. On materialization of array valued temporaries. In *13th International Workshop on Languages and Compilers for Parallel Computing 2000*, 2000.

[10] G. Roth. *Optimizing Fortran90D/HPF for Distributed–Memory Computers.* PhD thesis, Dept. of Computer Science, Rice University, Apr. 1997.

[11] R. Tolimieri, M. An, and C. Lu. *Mathematics of Multidimensional Fourier Transform Algorithms.* Springer-Verlag, 1993.

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

[13] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. Technical Report UT-CS-97-366, Department of Computer Science, University of Tennessee, Dec. 1997.