Scheduling FFT Computation on SMP and Multicore Systems

Ayaz Ali
Dept. of Computer Science
University of Houston
Houston, TX 77204, USA
ayaz@cs.uh.edu

Lennart Johnsson
Dept. of Computer Science
University of Houston
Houston, TX 77204, USA
johnsson@tlc2.uh.edu

Jaspal Subhlok
Dept. of Computer Science
University of Houston
Houston, TX 77204, USA
jaspal@uh.edu

ABSTRACT

Increased complexity of memory systems to ameliorate the gap between the speed of processors and memory has made it increasingly harder for compilers to optimize an arbitrary code within a palatable amount of time. With the emergence of multicore (CMP), multiprocessor (SMP) and hybrid shared memory multiprocessor architectures, achieving high efficiency is becoming even more challenging. To address the challenge to achieve high efficiency in performance critical applications, domain specific frameworks have been developed that aid the compilers in scheduling the computations. We have developed a portable framework for the Fast Fourier Transform (FFT) that achieves high efficiency by automatically adapting to various architectural features. Adapting to parallel architectures by searching through all the combinations of schedules (plans) is an expensive task, even when the search is conducted in parallel. In this paper, we develop heuristics to simplify the generation of better schedules for parallel FFT computations on CMP/SMP systems. We evaluate the performance of OpenMP and PThreads implementations of FFT on a number of latest architectures. The performance of parallel FFT schedules is compared with that of the best plan generated for sequential FFT and the speedup for different number of processors is reported. In the end, we also present a performance comparison between the UHFFT and FFTW implementations.

Categories and Subject Descriptors

F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems; D.1.3 [**Programming Techniques**]: Concurrent Programming; G.4 [**Mathematical Software**]: *Parallel and vector implementations*

General Terms

Performance, Design, Algorithms

Keywords

Automatic Performance Tuning, Fast Fourier Transform, Automatic Parallelization, Multicore, Shared memory

1. INTRODUCTION

The large gap in the speed of processors and main memory that has developed over the last decade and the resulting increased complexity of memory systems introduced to ameliorate this gap has made it increasingly harder for compilers to optimize an arbitrary code within palatable amount of time. Though issues of heat removal recently have considerably slowed the rate of increase in processor clock frequencies, the industry response to seek continued exponential performance growth in the spirit of "Moore's law" is multicore chips, or heterogeneous architectures, such as the Cell Broadband Engine, which further adds to the complexity of code optimization. To address the challenge to achieve high efficiency in performance critical functions, domain specific tools and compilers have been developed, which aid the compilers in scheduling the computations to adapt to underlying architecture.

The Fast Fourier Transform (FFT) is one of the most widely used algorithms in many fields of science and engineering, especially in the field of signal processing. Since 1965, various algorithms have been proposed for computing DFTs efficiently. However, the FFT is only a good starting point if an efficient implementation exists for the architecture at hand. Scheduling operations and memory accesses for the FFT for modern platforms, given their complex architectures, is a serious challenge compared to BLAS-like functions. It continues to present serious challenges to compiler writers and high performance library developers for every new generation of computer architectures due to its relatively low ratio of computation per memory access and non-sequential memory access pattern. FFTW[7, 6], SPIRAL[12, 5] and UHFFT[10, 9, 2] are three current efforts addressing machine optimization of algorithm selection and code optimization for FFTs.

In UHFFT, run-time optimization is performed by searching the best schedule (plan) from among an exponential number of combinations of factorizations and algorithms. For a given FFT problem, search adds some cost to the total execution time. However the performance gain could be significant, especially when the same size problem is repeatedly used, as in multidimensional FFTs.

In this paper, we have extended our adaptive approach for single processor to shared memory multiprocessor architectures. On parallel architectures (CMP/SMP), additional parameters such as processor count and their layout need to be considered if the absolute best performance is desired. However, additional parameters could potentially multiply the cost of search for the optimal parallel plan. Intelligent heuristics need to be developed that could make the selection process of the optimal execution strategy more efficient. We evaluate the performance of two parallel FFT implementations using OpenMP and PThreads on a number of latest CMP/SMP architectures and discuss the differences between the two programming models. The performance of parallel FFT schedules is compared to that of the best plan generated for sequential FFT and the speedup for different number of processors is reported. For moderately large problem sizes, we were able to achieve super linear speedup on all the architectures.

2. BACKGROUND

2.1 FFT

The Fast Fourier Transform (FFT) is a divide and conquer algorithm for quick evaluation of the Discrete Fourier Transform (DFT). For completeness, we briefly discuss the famous Cooley Tukey algorithm, which is one of the main algorithms used in the UHFFT library. We refer the reader to [8, 4, 14, 15] for the detailed description of the algorithms. In particular, the notation we use here mostly coincides with the notation in [8].

Discrete Fourier Transform (DFT) of a complex vector is a matrix vector product defined by:

$$X_l = \sum_{j=0}^{N-1} \omega_N^{lj} x_j \tag{1}$$

$$X_l = W_N . x \tag{2}$$

where ω_N is an Nth root of unity: $e^{-\frac{2\pi i}{N}}$. The periodicity of ω_N, introduces an intricate structure into the W_N DFT matrix , which makes possible the factorization of W_N into a small number of sparse factors. For example, it can be shown that when $N = r \times m$, W_N can be written as:

$$\mathbf{X} = (W_r \otimes I_m) T_m^N (I_r \otimes W_m) \Pi_{N,r} \mathbf{x} \tag{3}$$

where $T_m^N$is a diagonal "twiddle factor" matrix and $\Pi_{N,r}$is a mod-r sort permutation matrix. This is the heart of the FFT idea, and the formulation in Eq.3 is the well known Cooley Tukey Mixed Radix splitting algorithm[4]. The algorithm not only reduces the complexity from $O(n^2)$ to $O(n \log n)$, but offers a recursive divide and conquer structure that is most suited to uniprocessor as well as multiprocessor architectures with deep memory hierarchy.

Parallel FFT

The "Kronecker product" formulation of FFT is an efficient way to express sparse matrix operations. It also exposes the data parallelism that is inherent in the Mixed Radix FFT algorithm. In the above formulation, r (called radix) subproblems of size m FFT $(I_r \otimes W_m)$ can be executed independently followed by "twiddle" multiplication and m independent FFTs of size r, i.e., $(W_r \otimes I_m)$. Both, the permutation step and the "twiddle" multiplication step can be fused inside the first FFT subproblem, while still maintaining the data parallelism. Loan[8] provides a good discussion of parallel FFT algorithms including the implementations for shared memory parallel machines. Parallel algorithms for FFT can be broadly divided in two categories, i.e., the algorithms that perform explicit reordering of data using transposes[1, 3] and the algorithms that do not perform any movement of data. Computing a FFT on distributed data is not possible without movement of data. But, parallel FFT can be computed without remapping of data on architectures with shared address space. In general, explicit scheduling may perform better depending on the cost of data communication among processors. Recently, the SPIRAL team has presented an approach to automatic generation of parallel FFT code for SMP and multicore architectures[5]. In their paper, they evaluate the performance of parallel FFT code using OpenMP as well as PThreads on a number of CMP/SMP architectures. They also compare the performance of their implementations with that of FFTW.

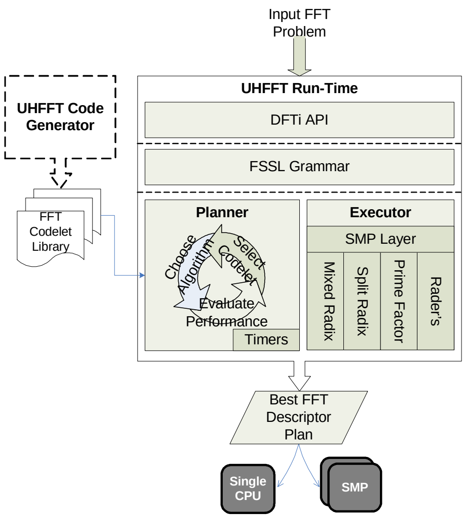


Figure 1: Block Diagram of UHFFT Run-time

2.2 UHFFT

The UHFFT system comprises of two layers: the code generator (FFTGEN) and the run-time framework. The code generator generates highly optimized small DFT, straight line "C" code blocks called *codelets* at installation time. These *codelets* are combined by the run-time framework to solve large FFT problems on Real and Complex data. Block diagram of UHFFT run-time is given in Figure 1.

2.2.1 Code Generator

The FFT library contains a number of *codelets*, which are generated at installation time. Each *codelet* sequentially computes a part of the transform and overall efficiency of the code depends strongly on the efficiency of these *codelets*. Therefore, it is essential to have a highly optimized set of DFT *codelets* in the library. The Code generator adapts to the platform, i.e., compiler and hardware architecture by empirically tuning the *codelets* using iterative compilation and feedback [2].

2.2.2 Plan Specification

An execution plan determines the *codelets* that will be

Table 1: Subset of FSSL grammar

#	CFG Rules
1-3	ROOT$\longrightarrow$FFT | SMPFFT | MODULE
4	FFT$\longrightarrow$(**outplace**$\mathbb{Z}$, FFT **mr** MODULE)
5	SMPFFT$\longrightarrow$(**mr p**$\mathbb{Z}$ BLOCK , FFT)$\mathbb{Z}$
6-7	MODULE$\longrightarrow$CODELET | (**rader**$\mathbb{Z}$, FFT)
8	BLOCK$\longrightarrow$**b** $\mathbb{Z}$: $\mathbb{Z}$
9	CODELET$\longrightarrow$$n$ $\in$generated set of codelets

used for that FFT size and also the order (schedule) in which they will be executed. An FFT plan is described in concise FFT Schedule Specification Language (FSSL), which is generated from a set of context free grammar productions given in Table 1. It allows different algorithms to be mixed together to generate a high performance execution plan based on properties of the input vector and its factors. Additionally, parallel execution plan provides an option of explicitly specifying the data distribution scheme on multiple processes. Indeed, by implementing a minimal set of rules, adaptive schedules could be constructed that suit different types of architectures.

2.2.3 *Planner*

Each FFT problem is identified by a DFTi descriptor [13], which describes various characteristics of the problem including size, precision, input and output data type and number of threads. Once the descriptor is submitted, planner selects the best plan, which may be used repeatedly to compute the transforms of same size. Our current implementation supports two strategies for searching the best plan. Both strategies use *dynamic programming* to search the space of possible factorizations and algorithms, given by a tree as shown in Figure 2. The first approach called *Context Sensitive Empirical Search,* empirically evaluates the sub-plans. To avoid re-evaluation of identical sub-plans, a lookup table is maintained to store their performance. In the second approach, called *Context Free Hybrid Search,* the cost of search is significantly reduced at the expense of plan quality. In this scheme, the cost of a subplan is estimated by empirically evaluating only the *codelet* that is encountered in a bottom up traversal. In this paper, we use the first approach because it generates good quality plans at reasonable cost of search.

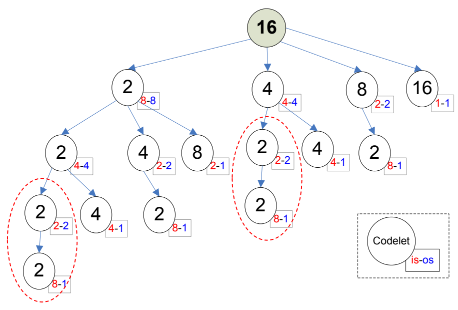


Figure 2: Search Space for Size 16 FFT. There are a total of 8 possible plans. *Codelets* are called with different input and output strides at different levels of tree (recursion) also known as ranks.

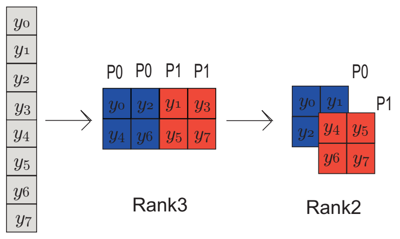


Figure 3: Parallel FFT plan for $N = 8$ and $P = 2$: $(\mathbf{mrp}2, (\mathbf{outplace}8, 4\mathbf{mr}2))8$. Parallelization is performed on the two dimensional formulation of 1D FFT.

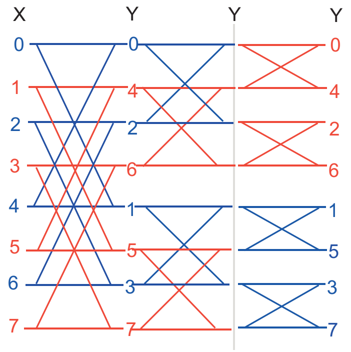


Figure 4: Butterfly representation of Parallel FFT plan for $N = 8$ and $P = 2$: $(\mathbf{mrp}2, (\mathbf{outplace}8, 4\mathbf{mr}2))8$. The computation is distributed at the first level of recursion, i.e., between $\log n$th and the previous rank.

3. PARALLEL FFT PLAN

Performance of a multithreaded execution depends on that of the serial code. Once a FFT plan is divided among threads, each of them executes part of the serial plan in parallel. In a straightforward implementation using Cooley Tukey factorization, the problem is recursively broken into smaller FFT sub-problems in a multidimensional formulation[1]. An example of the multidimensional representation of size 8 FFT plan is given in Figure 3. In general, a FFT (when $N = r^i$) can be formulated as a $\log_r N$ dimensional FFT problem using radix r factorizations. The butterfly representation of same parallel FFT plan is given in Figure 4. These representations are particularly useful in understanding the distribution of data and access pattern that is vital in parallelization of any algorithm.

Notice that the parallelization is performed at the first level of recursion, i.e., on the two dimensional formulation of FFT. Contrary to nested parallelism, this ensures there is only one point (barrier) where all threads have to synchronize before executing the deeper ranks or stages of recursion. Synchronizing threads adds to the overhead of multithreaded execution and causes major performance degradation if proper load-balancing is not employed. In general, even distribution of work is possible when P divides $\sqrt{N}$, i.e., $(P|\sqrt{N})$.

3.1 Row/Column Blocking

The communication cost on shared memory multiproces-

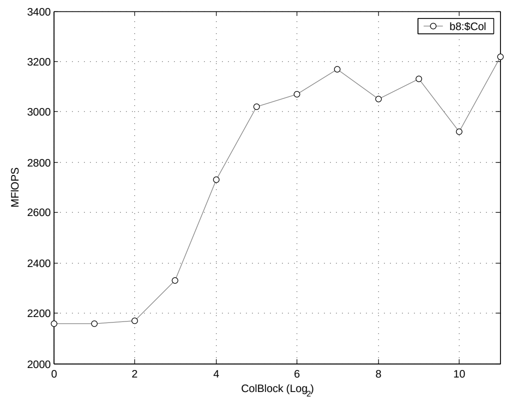


Figure 5: Performance trend (on Itanium 2 Quad SMP) for plan:$(\mathbf{mrp}2, (\mathbf{outplace}65536, 16\mathbf{mr}16\mathbf{mr}16\mathbf{mr}16))65536$, due to varying sizes of blocks along columns . Performance is measured using mflops or Million Floating Point Operations Per Second and is calculated from the execution time and standard algorithm complexity for FFT i.e. $5N \log N$.

sor systems with uniform memory access (UMA) does not play a major role in the performance. However, different data distributions have an impact on the performance of memory systems on some architectures. In Figures 3 and 4, we have shown the block distribution of workload with strided data along row dimension and contiguous data along columns. The example given in the Figures 3 and 4 has only one possible distribution along rows since there are only two rows and two processors. Based on the results of our experiments for larger data sizes on SMP/CMP machines, we found that there was no performance difference in how the rows were distributed as long as each processor worked on a full row (of strided data). However, different distributions of columns had some performance variation as shown in Figure 5. This is mainly due to the cache coherence conflicts and false sharing [11], which are caused by the elements at the column boundaries falling in same cache line but different processors. As a rule of thumb, choosing maximum blocks of columns works best as indicated in the graph in Figure 5.

3.2 Plan Selection

In general, the sequential plan generated by one of the search schemes in the planner could be selected for parallel execution. However, the plan thus selected is not guaranteed to be optimal as shown in Figure 6 . In the graph, one of the plans (annotated by an arrow) represents the parallel execution of sequential plan selected by our existing search scheme. Notice that the execution results in sub-optimal performance. A better search scheme needs to take into account a limited number of parallel plans in order to select the best parallel plan for the target architecture.

Algorithm 1, selects the best parallel plan $\rho_{parallel}$ for a FFT of size N using P cores/processors. It uses UHFFT's search scheme for sequential plan selection to find the best serial plan ρ_{serial} for size N FFT. Since UHFFT uses dynamic programming and maintains a table of performance numbers for sub-problems, all iterations inside the loop incur

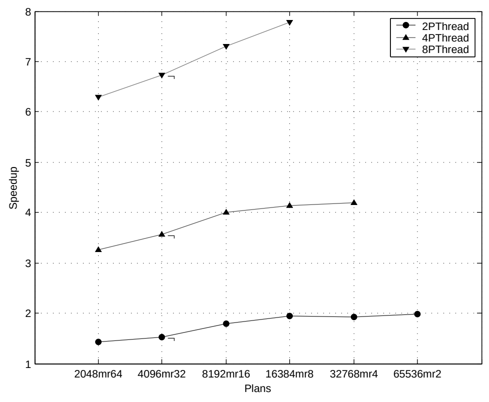


Figure 6: Speedup of different parallel plans for FFT of size 128K on Opteron 8-way SMP Machine.

Algorithm 1 Selecting Parallel FFT Plan

```
Search best sequential plan $\rho_{serial}$ for $N$
$perf_{serial}$ = Evaluate $\rho_{serial}$
$perf_{max} = 0$
while $r = \{i \times P \in codelet\, library \quad \forall i \in \{1, 2, 3 \cdots\}$
  if $r \mid N$
    Lookup best sequential plan $\rho$ for $\frac{N}{r}$
    $perf$ = Evaluate $(\mathrm{mrp}\, P, \rho\, \mathrm{mr}\, r)N$
    if $\max(perf\,, perf_{max}) \neq perf_{max}$
      $\rho_{parallel} = (\mathrm{mrp}\, P, \rho\, \mathrm{mr}\, r)N$
        $perf_{max} = perf$
if $\max(perf_{serial}\,, perf_{max}) \neq perf_{max}$
  Return $\rho_{serial}$
Return $\rho_{parallel}$
```

constant cost of table lookup and evaluation. The algorithm evaluates a limited number of parallel plans to find the best two dimensional (parallel) formulation of a given $1D$ FFT problem. Each formulation consists of rows of length $\frac{N}{r}$ and columns of length (radix) r.

4. IMPLEMENTATION

Most compilers support native multithreaded programming APIs. Posix thread (PThreads) library routines are flexible but offer limited portability across different architectures. OpenMP is a portable, scalable model. It provides a simple interface for developing multithreaded applications. Two types of multithreaded programming models are commonly used, i.e., fork/join model and thread pooling model. Posix Thread is an example of fork/join threading model. Nevertheless, thread pooling can be built on top of fork/join threading. Most OpenMP implementations use thread pools to avoid the overhead of creating and destroying threads after each parallel region. These threads exist for the duration of program execution.

Implementation of parallel FFT using OpenMP is relatively straightforward given an efficient parallel plan. The two loops around the row and column FFTs are decorated with OMP directives and they are distributed in contiguous chunks (blocks). In our Posix Threads implementation, we use thread pooling technique to avoid the overhead of creating threads. The pool of peer threads, as shown in Figure 7,

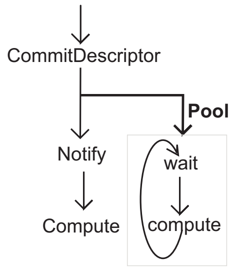


Figure 7: Thread Pool Model

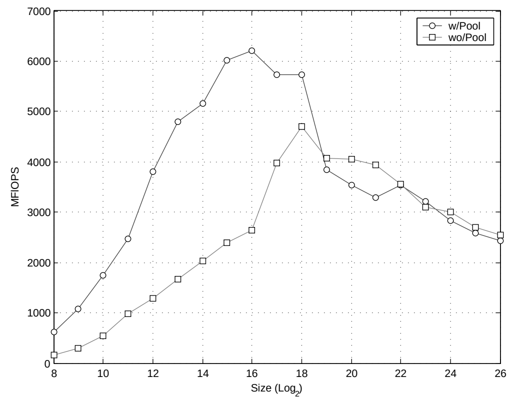


Figure 8: Performance Comparison of two multi-threaded implementations on Xeon 2×Dual (four) cores. Current implementation uses thread pools and busy wait. The old implementation creates threads as needed and uses condition variables and mutex for synchronization.

is created when the DFTi descriptor is submitted and after the best plan is selected. The pool is destroyed when the descriptor is freed. To reduce the cost of synchronization, we implemented a low latency barrier using atomic decrement instruction. In addition to that, we used *busy wait* when the plan was perfectly load balanced. Contrary to waiting on events, this technique avoids the overhead of system calls and thread scheduling. In Figure 8, we give the performance comparison of our preferred multithreaded implementation using pooling and customized barrier with our old implementation that used fork/join model and native synchronization primitives. In general, we observed that thread pooling was a major factor in the performance improvement for relatively small sizes. But, for larger sizes, using *busy wait* resulted in slightly lower performance.

5. RESULTS

Performance benchmarking of the OpenMP and PThreads parallel implementations was performed on a number of recent SMP/CMP architectures listed in Table 2. Both Itanium 2 and Opteron 846 are traditional SMP machines where each processor has separate caches and shared main memory. The Xeon Woodcrest node consists of two dual cores arranged in a SMP setting; the two cores in a dual share L2 cache. The Opteron 275 node also consists of two dual cores; each core having separate L1/L2 cache. In this paper,

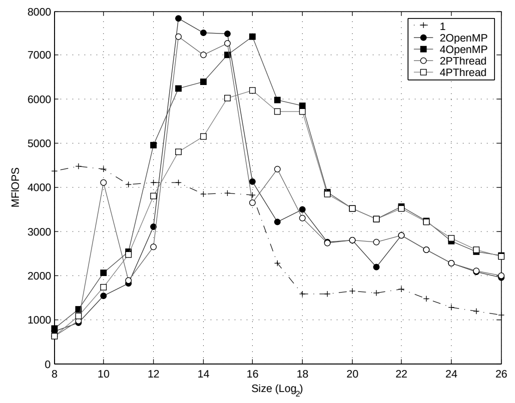


Figure 9: Performance of Complex FFT problems for powers of two sizes. The plots compare the OpenMP and PThreads implementations and speedup gained for varying number of threads on Xeon Woodcrest 2×Dual CMP/SMP Machine. For all sizes the performance of sequential FFT execution is also given.

the performance numbers are derived from the total execution time of a FFT problem. The performance measure, Million Floating Point Operations Per Second (MFLOPS), is calculated from the execution time and commonly used algorithm complexity of FFT, i.e., $5N \log N$. All the results were collected on double precision complex FFT of powers of two sizes. In each case, the benchmarking was performed using varying number (powers of two) of available CPUs.

5.1 OpenMP vs PThreads

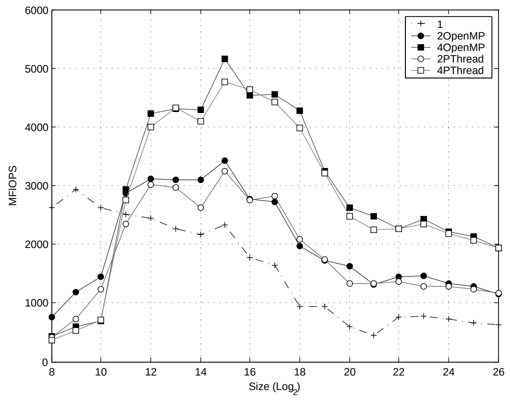
In the first set of results, we compared the performance of the two parallel FFT implementations, i.e., OpenMP and PThreads on three different architectures. Our PThreads implementation uses thread pooling technique, which is common in most OpenMP flavors. Similar to OpenMP, we also employ *busy wait* to implement barrier synchronization. We found that this worked better than using operating system assisted synchronization primitives such as semaphores or condition variables. In both implementations, we favor allocating contiguous chunks (blocks) of rows and columns to each core/processor. Considering all these similarities, it is not surprising that the performance of both the implementations is almost identical as shown in Figures 9 & 10. Although both the compilers, i.e. Intel and Pathscale, have good implementations of OpenMP, distributing loops efficiently among worker threads should be straightforward for most parallelizing compilers because the parallel plans generated by UHFFT are sufficiently load balanced.

5.2 Number of Processing Units (Threads)

Due to the synchronization overhead of multithreaded execution, small size FFT problems show a slowdown in performance as shown in graphs in Figures 9 & 10. An intelligent plan selection scheme, as given in Algorithm 1, would ensure that a parallel plan is selected only when it is effective. For all the architectures, the break even points occur after sizes $2^{11} - 2^{12}$, which is equal to the first level cache capacity. Due to low computation to I/O ratio, scalability

Table 2: Architecture Specifications

	Itanium 2	Opteron 846	Xeon Woodcrest	Opteron 275
Configuration	4 Processor SMP	8 Processor SMP	2×Dual Core CMP/SMP	2×Dual Core CMP/SMP
CPU Speed	1.5 GHz	2.0 GHz	2.66 GHz	2.2 GHz
Data Cache	16K,256K,6M	64K,1M	32K/Core,4MB/Duo	64K/Core,1MB/Core
Line Size	64B,128B,128B	64B,64B	64B,64B	64B,64B
Associativity	4,8,12 way	2,16 way	8,16 way	2,16 way
Theoretical Peak	24 GFLOPS	32 GFLOPS	42.56 GFLOPS	17.6 GFLOPS
Compilers	icc9.1	pathcc2.5	icc9.1	gcc4.1



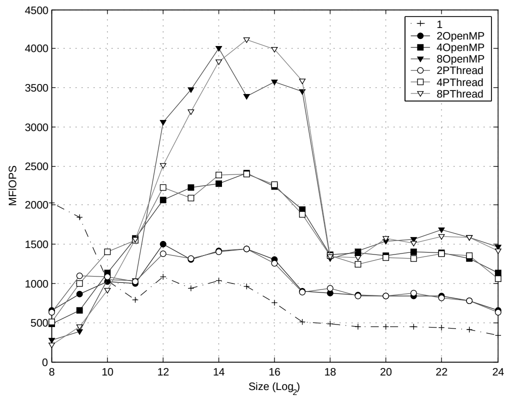


Figure 10: Performance of Complex FFT problems for powers of two sizes. The plots compare the OpenMP and PThreads implementations and speedup gained for varying number of threads on Itanium 2 Quad SMP Machine (left) and Opteron 8-way SMP Machine (right). For both machines, the performance of sequential FFT execution is also shown.

of the FFT algorithm depends on the memory bandwidth for significantly large sizes. However, for moderately large sizes, we were able to achieve super linear speedup on SMP machines due to the increase in effective size of cache. This pattern is evident in Figures 9 & 10.

5.3 Cache Architecture

Both Xeon and Opteron 275 architectures have identical number of cores but their cache configuration is quite different. The two dual cores on Xeon have shared L2 cache while the cores on Opteron have private caches similar to a conventional SMP machine. Although the shared cache configuration has its benefits in certain scenarios, the non-uniformity can pose some scheduling problems. On Xeon, we observed inconsistent performance for small sizes (that fit in cache) when only two threads were spawned. This phenomenon is shown in Figure 11; notice the extent of performance variation on the Xeon compared to the Opteron. Although the scheduler in linux kernel 2.6 tries to schedule tasks on different physical packages (sockets) when the system is lightly loaded, there is no guarantee that the scheduler will get it right the first time. The performance drops when the two threads are scheduled on the same dual, which may result in evictions due to conflicts in shared cache. This problem can be resolved by setting the affinity of CPUs to place the threads properly. In UHFFT the planner schedules the threads to different sockets (duals) if only two threads are spawned.

5.4 UHFFT vs FFTW

FFTW recently released a new version with improved multithreaded FFT implementation that uses thread pooling. In the final set of results, given in Figures 12 & 13, we compare the multithreaded performance of UHFFT with that of FFTW on three architectures. UHFFT employs the search algorithm presented earlier in the paper, to select the best plan with appropriate number of threads. In case of FFTW, it appears that the multithreaded execution is employed for sizes 256 and larger, as shown in Figures 12 & 13. The performance of UHFFT is quite competitive with that of FFTW for most sizes on all the architectures. On the two multicore machines (Xeon Woodcrest and Opteron 275), FFTW performs significantly better for very large sizes. We believe that it could be due to the fact that FFTW generates architecture specific *codelets* that exploit SIMD instructions on those two architectures. Unfortunately, UHFFT currently does not support SIMD enabled *codelets*, which is why performing a comparison strictly on the basis of parallel implementation of the two libraries is not possible. Nevertheless, these plots establish the parallel FFT performance benchmarks on the latest CMP/SMP systems.

6. DISCUSSION

One of the key objectives of this paper has been to evaluate the performance of parallel FFT implementations on the emerging multicore/manycore architectures. There are many factors affecting the speedup and efficiency of a parallel FFT besides load balancing. FFT is a highly data parallel algorithm due to its divide and conquer structure. The-

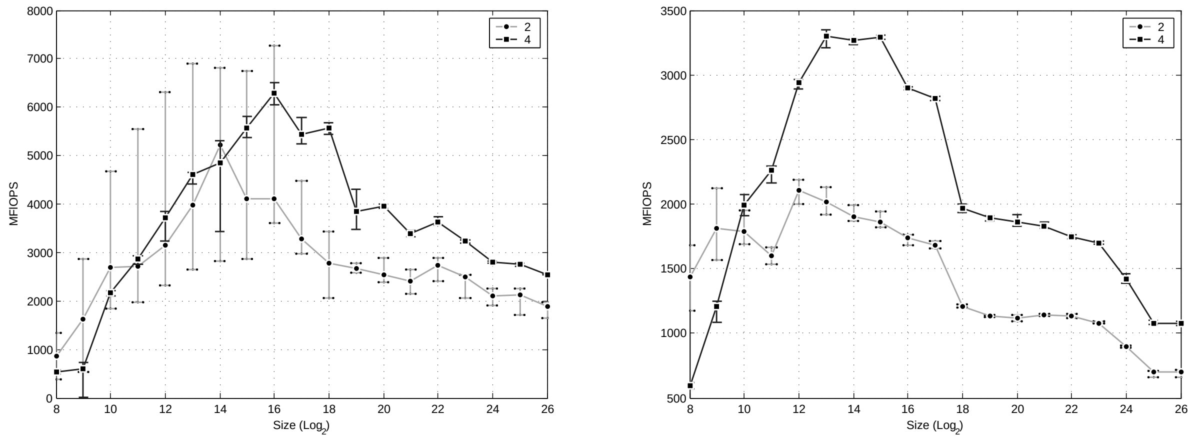


Figure 11: Performance variation on the two multicores with different cache configurations (Shared L2 Cache vs Separate L2 Cache). On Xeon Woodcrest (left), the two cores per dual share L2 cache. On Opteron 275 (right), the cores have private L2 cache. For each size, the same plan was executed 10 times and the mean performance was plotted along with the variation (given by error bars).

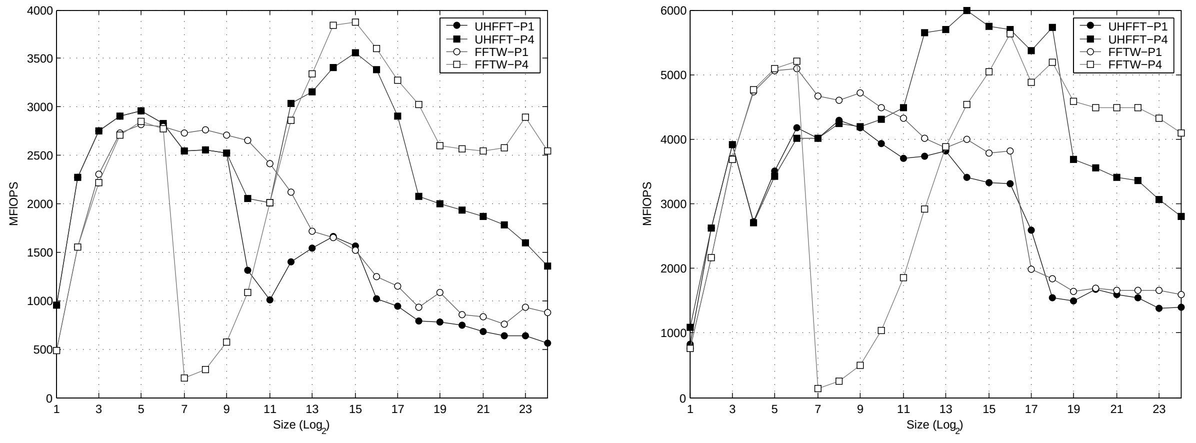


Figure 13: Performance comparison of UHFFT-2.0.1beta and FFTW-3.2alpha on Opteron 275, 2×Dual CMP/SMP (left) and Xeon Woodcrest, 2×Dual CMP/SMP (right).

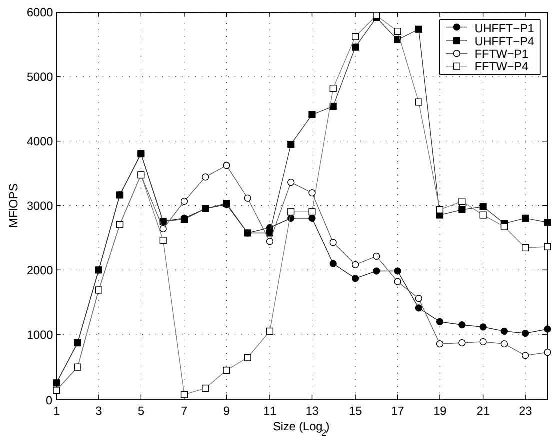


Figure 12: Performance comparison of UHFFT-2.0.1beta and FFTW-3.2alpha on Itanium 2 Quad SMP

oretically, achieving super-linear speedup is quite possible ($N = 2^{18}$ in Figure 12) because of larger effective cache size on shared memory multiprocessor and multicore systems. For example on Itanium 2 Quad SMP, we were able to get speedup of 3 for most sizes. However, the speedup on the two multicore machines, i.e., Xeon Woodcrest and Opteron 275 was not as good. On Xeon, the best speedup for both UHFFT and FFTW was between 1.5 and 2.5 using all the four cores. Similarly, the best efficiency was also observed on Itanium 2 Quad SMP, where UHFFT was able to get 25% of the peak for in-cache sizes; the efficiency drops further to 12% of the peak for very large sizes. On Opteron 275, which has private cache per core, the performance was similar, i.e., 22% and 15% of the theoretical peak for in-cache and out-of-cache sizes respectively. The efficiency on Xeon Woodcrest was the lowest: 14% for moderately large sizes and 9% for very large sizes. The performance potential of shared cache multicore architectures is realized when the two threads can potentially share some section of data. In FFTs, all the processors may share the read-only twiddle factor array. It is likely that one of the two threads sharing the cache would save twiddle array cache misses. However, the potential gain is over-shadowed by the conflicts related to cache associativity; as the stride increases, the two threads start competing for a smaller (localized) portion of the cache, resulting in lower effective bandwidth. Given that the main bottleneck in the performance of FFTs has largely been the effective memory bandwidth; the speedup and efficiency is always going to be determined by such factors as the cache associativity and cumulative cache size instead of the computation power of cores.

7. CONCLUSION

With the emergence of multicore and hybrid shared memory multiprocessor machines, performance scalability can not be guaranteed without careful partitioning of workload. In UHFFT, this is achieved by searching the plan that will yield the best parallel performance on available processors. We presented a simple algorithm based on our existing search schemes, which selects the best multithreaded plan. Different plans result in different distributions of data. The best distribution and factorization is selected through empirical evaluation of a limited number of parallel plans. We evaluated the performance of two multithreading models, i.e., thread pooling and fork/join models. In general, thread pooling performs better because it avoids the overhead of creating and destroying threads. Synchronization is another source of overhead in a load balanced FFT plan. For relatively smaller sizes, the performance of a parallel plan depends on the availability of efficient synchronization mechanisms that can bypass system calls and thread scheduler. For larger sizes, the computation to synchronization ratio minimizes, resulting in better speedup. In order to compare the efficacy of parallelizing compilers, we implemented parallel FFT using two multithreaded programming models, i.e., OpenMP and PThreads. The performance numbers indicate that if the loops are carefully structured, OpenMP can achieve the performance that is as good as that of native threads. We also presented a head-to-head performance comparison of the most recent versions of UHFFT and FFTW. To our knowledge this is the first such study comparing the performance of different parallel FFT implementations on most recent multicore/manycore architecture using different compilers.

8. REFERENCES

[1] Agarwal, R. C., Gustavson, F. G., and Zubair, M. A high performance parallel algorithm for 1-d fft. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 34–40.

[2] Ali, A., Johnsson, L., and Mirkovic, D. Empirical Auto-tuning Code Generator for FFT and Trignometric Transforms. In *ODES: 5th Workshop on Optimizations for DSP and Embedded Systems, in conjunction with International Symposium on Code Generation and Optimization (CGO)* (San Jose, CA, March 2007).

[3] Bailey, D. H. Ffts in external or hierarchical memory. *J. Supercomput. 4*, 1 (1990), 23–35.

[4] Cooley, J., and Tukey, J. An algorithm for the machine computation of complex fourier series. *Mathematics of Computation 19* (1965), 297–301.

[5] Franchetti, F., Voronenko, Y., and Püschel, M. FFT program generation for shared memory: SMP and multicore. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), ACM Press, p. 115.

[6] Frigo, M. A fast Fourier transform compiler. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation* (New York, NY, USA, 1999), ACM Press, pp. 169–180.

[7] Frigo, M., and Johnson, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231. special issue on "Program Generation, Optimization, and Platform Adaptation".

[8] Loan, C. V. *Computational frameworks for the fast Fourier transform.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[9] Mirkovic, D., and Johnsson, S. L. Automatic Performance Tuning in the UHFFT Library. In *ICCS*

'01: Proceedings of the International Conference on Computational Sciences-Part I (London, UK, 2001), Springer-Verlag, pp. 71–80.

[10] Mirkovic, D., Mahasoom, R., and Johnsson, S. L. An adaptive software library for fast Fourier transforms. In *International Conference on Supercomputing* (2000), pp. 215–224.

[11] Patterson, D. A., and Hennesy, J. L. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2002.

[12] Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B. W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93*, 2 (2005), 232–275.

[13] Tang, P. T. P. DFTI – A New Interface for Fast Fourier Transform Libraries. *ACM Transactions on Mathematical Software 31*, 4 (Dec. 2005), 475–507.

[14] Temperton, C. Self-Sorting Mixed-Radix Fast Fourier Transforms. *Journal of Computational Physics 52* (1983), 1–23.

[15] Tolimieri, R., An, M., and Lu, C. *Algorithms for discrete fourier transform and convolution*. Springer-Verlag, 1997.