Cost Functions for the FFT on an SGI Origin 2000

by: Michael Catchpole

Jason Cernansky

Scott Murawski

David Whittaker

Faculty Advisor: Professor Lenore Mullin

The Fast Fourier Transform or FFT is one of the most prevalent and important algorithms used in modern computing, with applications ranging from waveform analysis to signal processing and the manipulation of graphics.

Our research focuses mostly on generating cost functions for different aspects of the FFT. We look into cache misses, page faults, and message passing, as well as different methods of cache and memory management. Our goals are as follows:

- generate a function to determine the number of cache misses for a given problem size
- generate a function to determine the number of page faults for a given problem size
- generate a function to determine the cost of message passing for a given problem size
- create a function that will determine the total cost of running the FFT for a given problem

All the aforementioned goals rely on the fact that we know the size of the problem, the number and size of the cache lines, and the number and size of the pages. We realized, while doing this research, that there are too many variables involved when focusing on all possible FFT implementations, so we decided to limit the scope of our research to problem sizes that are powers of two, where the cache and memory sizes evenly fit these groupings.

For benchmarking and running the experiments, we employed the Origin 2000 cluster at the National Center for Supercomputing Applications[1]. The Origin 2000 is a non-uniform memory architecture machine, developed jointly by Silicon Graphics and Cray, which utilizes SGI's IRIX OS. Based on MIPS R10000 processors, the machine offers peak processor performance at 390 MFLOPS with each processor utilizing 32k of instruction and data cache. The processors are connected by a distributed shared memory system, which has 4mb of secondary unified cache and up to 160Gb of physical memory over the entire cluster.

These nodes are interconnected with HIPPI, a 200mb/s switching architecture, that enables the entire system to be broken into smaller segments, which appear as separate machines. NCSA has segmented their machine into 9 segments. Modi1-4 are 32 processor clusters with varying amounts of physical memory. Jord1-2 and Saga1-2 both have 64 processors and 16Gb of physical memory. The remaining processors have been partitioned off as a test system. The majority of our experiments were run using SGI's load sharing batch system, or lsbatch, as batch jobs on the Modi or Jord segments depending on the system requirements of each job and the batch queue to which the job was submitted,

The Origin 2000 provides compilers for most major programming languages, including Fortran 77, Fortran 90, C, C++ and High Performance Fortran (HPF). The algorithm that we used for benchmarking the FFT was written in HPF, which we used due to the ease with which HPF programs can be distributed over multiple processors.

Before explaining the results of our experiments it might be helpful to give a short explanation of the FFT, along with some of its' history and applications.[2] The Fast Fourier Transform is named for Jean Baptiste Joseph Fourier a mathematician of the early 1800's. He is best known for the Fourier series, a mathematical series of sine and cosine terms that he devised to analyze heat conduction in solid bodies. Fourier spent the remaining years of his life working on this series and devised the Fourier integral before his death in 1830. Both the Fourier series and the Fourier integral transform time domain waveforms to the frequency domain and vice versa. The frequency domain of the Fourier transform contains the same information as the original function, although often in a more easily understood and manipulated form.

Originally, in computer science, the Fourier transform algorithm was written using the discrete Fourier transform (DFT), which is extremely expensive in regards to computation time, requiring N^2 iterations per transform where n is the problem size. This was a major setback until 1965 when a new Fourier transform algorithm was developed at IBM research. Richard L. Garwin, in need for a fast means of performing Fourier transfers for his studies, took an outline of John W. Turkey's work on the transform algorithm to IBM to have it programmed. A young programmer at IBM by the name of James W. Cooley quickly worked out the program, writing a $\log_2 N$ algorithm and thinking that he would hear no more of it. This algorithm is often referred to as the Cooley-Turkey Fast Fourier Transform, or FFT. Contrary to Cooley's belief, the FFT has come to be an extremely important algorithm, with a wide range of applications in scientific as well as industrial sectors.

The FFT can be applied to any data that has been numerically sampled. When data is sampled it is recorded at regular intervals. Once the numbers are found, the FFT can be applied in order to obtain the complex frequency domain for many different reasons. Two different uses of Fourier transforms are mechanical signature analysis, and filtering.

Signature analysis utilizes the fact that all rotating machinery vibrates in a relatively constant manner. The detection of abnormalities in this vibration signature is an excellent indication of a problem with the machinery. By applying an FFT to the waveform of the vibration these abnormalities can be more easily recognized. This allows the machinery to be repaired before the device fails, saving large amounts of time and money in industrial settings.

The FFT is also employed in the filtering of waveforms. There are times, such as when measuring friction in small samples, where the natural frequency of the object being studied can interfere with the data being collected. By transforming this data using an FFT it becomes easily possible to filter out the data at this unwanted frequency thereby purifying the experimental results.

To further study the FFT on the SGI Origin 2000 at NCSA experiments were conducted on the algorithm using High Performance Fortran (HPF). By utilizing a few of the abilities of the HPF language, the array being transformed was distributed equally over a number of processors in a block distribution. This means that equal size contiguous blocks of the array were distributed, one to each processor, as the experiment ran. The FFT was then run on problem sizes of $16(2^4)$, $256(2^8)$, $4096(2^{12})$ and $65536(2^{16})$ elements. Each of these problem sizes was in addition run over 2, 4, 8 and 16 processors and the relevant data gathered from these experiments was plotted. As can be seen from the plot of messages passed (figure 1.1), this value grows linearly with both problem size and number of processors. This activity conforms to the requirements that

- As the size of the problem grows, more data must be sent back and forth between each processor and
- As the amount of processors increases communication cost increases in order for the processors to coordinate their activities.

Both system time (figure 1.2) and real time (figure 1.3) however do not conform as readily to this pattern. Both of the times show a recognizable spike between 4 and 8 processors, while the grade levels off from 8 to 16 processors. This is an excellent indication of how a specific hardware configuration can effect the performance of the algorithm. The NCSA's Origin 2000 system possesses 2 processors per node and 2 nodes per card. The peak seen between 4 and 8 processors is most likely the result of the cost for the new card involved to communicate with the original one. The grade slopes down as the experiment moves on to 16 processors because the cost of using additional cards is offset by the greater processing power gained.

Since the memory access patterns of the FFT algorithm are very regular, it is possible to create a function for the total cost of all the memory accesses.

This total cost is:

cost of all cache misses + cost of all page faults + cost of all messages

The cost of the cache misses includes the bus speed and the ram speed. Also included is the time it takes to figure out there is in fact a cache miss which depends on the algorithm used.

The cost of the page faults includes the disk access time and time it takes to load the page into memory from the disk. The operating system incurs costs by putting the current process on hold and calling the device driver to manage the disk.

The cost of the messages includes many things depending on the situation. In the Origin 2000 there are two processors on each node, so sending a message to the other processor on the same node doesn't incur much cost. The time it takes to pack and unpack the message and the bus speed are the main factors. The cost of sending a message on another level such as sending it to another node involves more cost. Included in this cost is the time it takes for the message to reach the switch, the time it takes the switch to direct the message to the appropriate node and the time it takes the message to reach the destination node. The Origin 2000 has 4 nodes per card, so sending a message from one card to another could possibly involve many switching stages.

Probably the worst possible situation is sending a message to another computer on the network. More costs are incurred based on the network protocol used and the speed of the network. If the network is not in dedicated mode there might even be message collisions involved if Ethernet is used.

All of these costs which are encapsulated within the cost of a cache miss, the cost of a page fault and the cost of a message have their own cost functions, that are not discussed here. The functions discussed here are the first step in figuring out the total cost function of the FFT under any condition and as a result, deal with several assumptions.

One assumption is the algorithm used. In this case, an LRU (least recently used) cache is assumed to be implemented. Because the FFT memory access patterns do not exhibit the locality that LRU is designed around, data left in the cache from a previous stage is not useful to the beginning of a new stage (except in boundary conditions). This has the same effect as flushing any usable data before a new stage begins. Another assumption is that all the data elements that need to be sent to another processor are packaged into one message.

The information used in order to determine the formulas are as follows:

N = Number of elements in the data set being transformed.
P = Number of processors over which the data set is broken up.
LS = Number of data elements that fit into an individual cache line.
PS = Number of data elements that fit into a page.

Because of the algorithm used, it makes no difference how many cache lines there are in the cache, or how many pages there are in real memory, as long as there are at least two of each and few enough that they cannot hold the entire problem.

The formulas for the cost of cache misses, page faults and messages passed follows:

Cost of Cache Misses

$$TotalCost = (CostPerCacheMiss) \cdot \log_2 N \cdot \frac{N}{LS}$$

Cost of Page Faults

$$TotalCost = (CostPerPageFault) \cdot \log_2 N \cdot \frac{N}{PS}$$

Cost of Messages

$$TotalCost = (CostPerMessage) \cdot \log_2 P \cdot \frac{N}{2}$$

As mentioned above, the formulas presented make certain assumptions about how the system manages the cache and the virtual memory pool:

- We have assumed that the cache has at least two blocks in it, but also that the problem is large enough that none of it is left in the cache when a new stage begins.
- We have assumed that the real memory contains at least two pages, but that the problem is large enough that none of it is left in memory when a new stage begins.

If the problem does in fact fit into real memory (or the cache) then the cost function for that portion of the memory hierarchy can be adjusted by removing the $\log_2 n$ multiplier next to the appropriate term. In this case, the only faults or cache misses that occur are those needed to get the problem loaded in the first stage. During all subsequent stages everything that is needed is already available.

As long as LRU is being used to handle the cache, increasing the number of available blocks in the cache will have no effect on the number of cache misses until the cache actually becomes big enough to hold the entire problem.

Similarly, for the virtual memory system, adding physical memory will not affect the number of page faults until enough memory is present to hold the entire problem.

This situation occurs because the access pattern is such that LRU will always throw out entries before they are needed when the FFT is being run. As each new stage begins, with the exception of very small problems, the first cache block [0-xx] from the previous stage will not be available because it was ejected in order to load new data. Based on the LRU algorithm the more relevant the data is in the cache, the more chance it stands of being in the most stale line of the cache, therefore being evicted moments before it is needed so that a block containing new data can be loaded.

Experimentation with problem sizes 1024, 2048, and 8192 (figure 2) has shown that reusing cache data on a subsequent stage is usually impossible. Cache sizes that are not powers of 2 and are almost, but not quite big enough to hold the problem sometimes save a few misses, but the savings are usually several orders of magnitude smaller than the overall number of misses one would expect from the problem.

This means that a system designer wishing to reduce the number of faults resulting from data not being preloaded, should concentrate on widening one level of the memory as much as possible instead of marginally increasing every level.

Another design issue involves division of various levels of memory into pages or blocks. For example, if the cache memory is sufficient to hold 64 numbers, it can be divided a variety of ways, ranging from two entries of 32 numbers, 4 entries of 16 numbers, and so on, all the way up to 64 entries of two numbers. The same amount of data will need to be accessed on each stage of the problem. By increasing the size of the blocks, fewer faults / misses will occur per stage. If decreasing the system overhead spent handling these situations is an issue, then the blocks should be made as large as they can without making all of the memory into one single block. Creating a single block breaks the low-end boundary condition. This happens because the first and second cells needed in each pair may lie on separate pages. For all but the first few stages, the algorithm would then generate a number of faults equal to the size of the problem.

The cost of a 1 dimensional FFT can be accurately predicted using this system. Multi-dimensional FFTs depend on the one dimensional FFT, so these formulas could be expanded to further define the behavior in higher order FFTs. Additionally, the formula demonstrates that adding memory to a system will usually fail to generate significant increases in the performance of this algorithm unless the high-end boundary condition is reached. Indeed, to fully optimize the usage of memory when running FFT would require either a memory replacement algorithm designed around the FFT, or a platform that would allow running algorithms to specify their memory usage patterns in advance to improve memory performance.

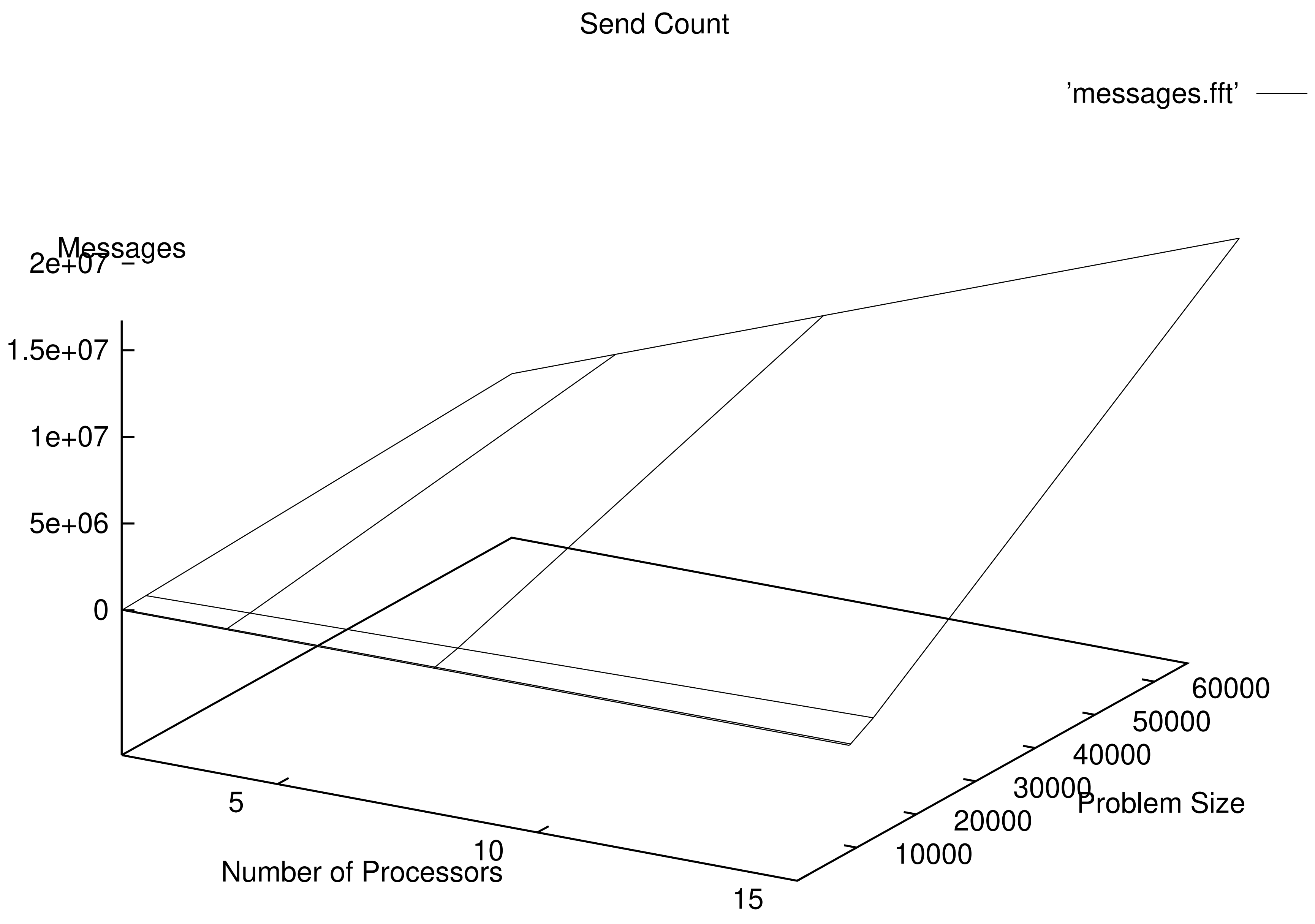
Send Count
'messages.fft'
Messages
2e+07
1.5e+07
1e+07
5e+06
0
5
10
15
Number of Processors
10000
20000
30000
40000
50000
60000
Problem Size

Figure 1.1

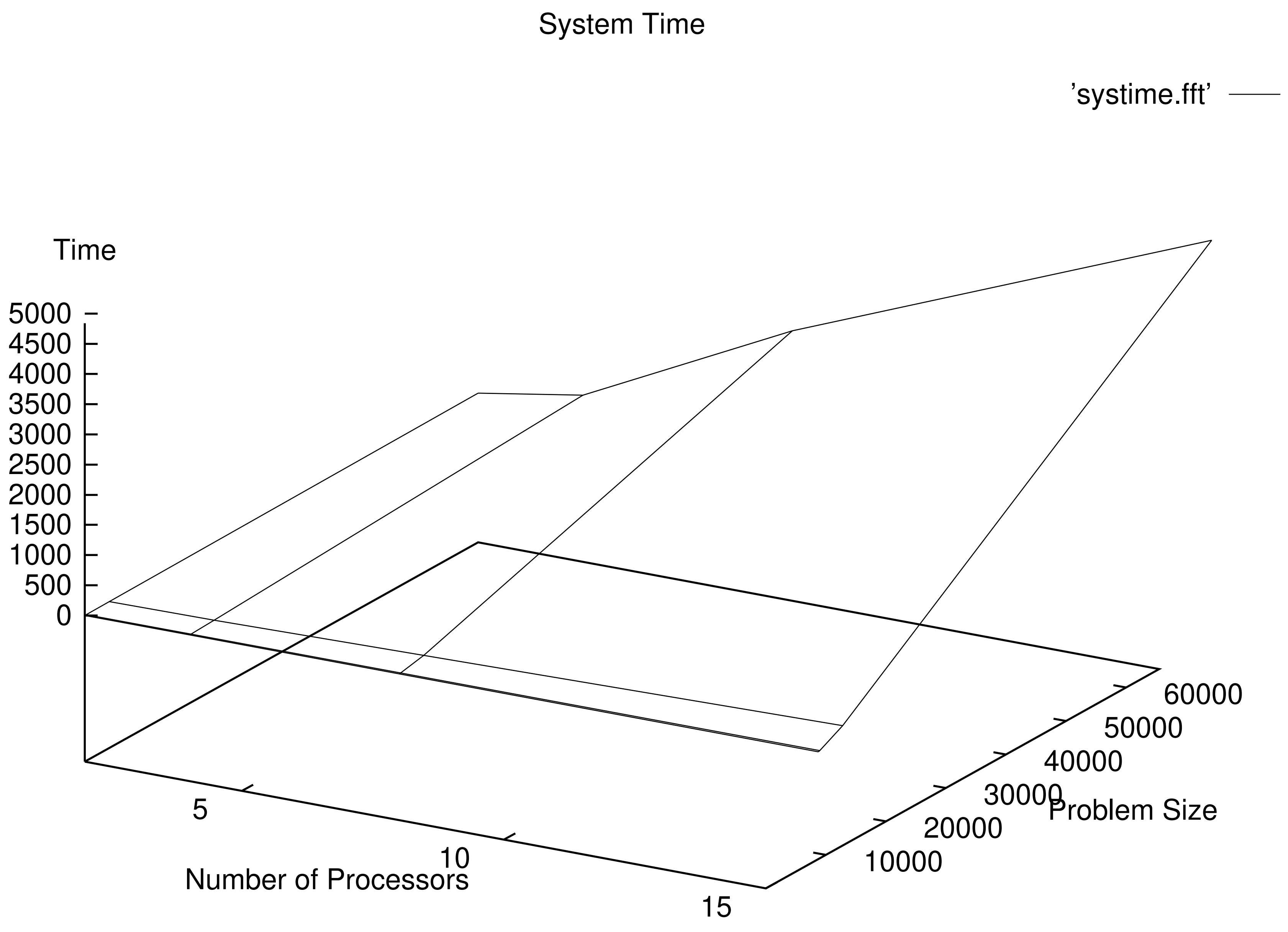
System Time
'systime.fft'
Time
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
5
10
15
Number of Processors
10000
20000
30000
40000
50000
60000
Problem Size

Figure 1.2

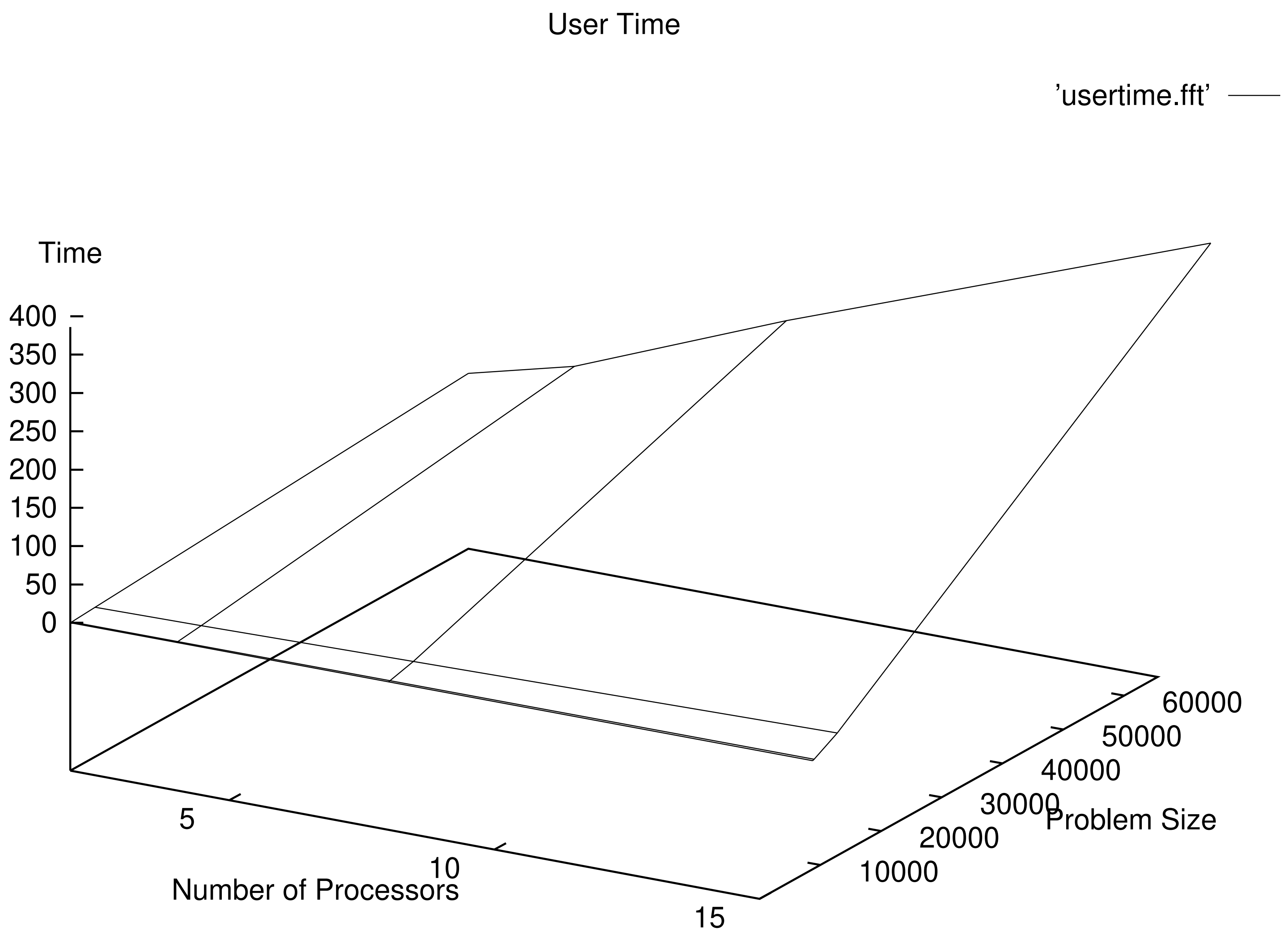
User Time
'usertime.fft'
Time
400
350
300
250
200
150
100
50
0
5
10
15
Number of Processors
10000
20000
30000
40000
50000
60000
Problem Size

Figure 1.3

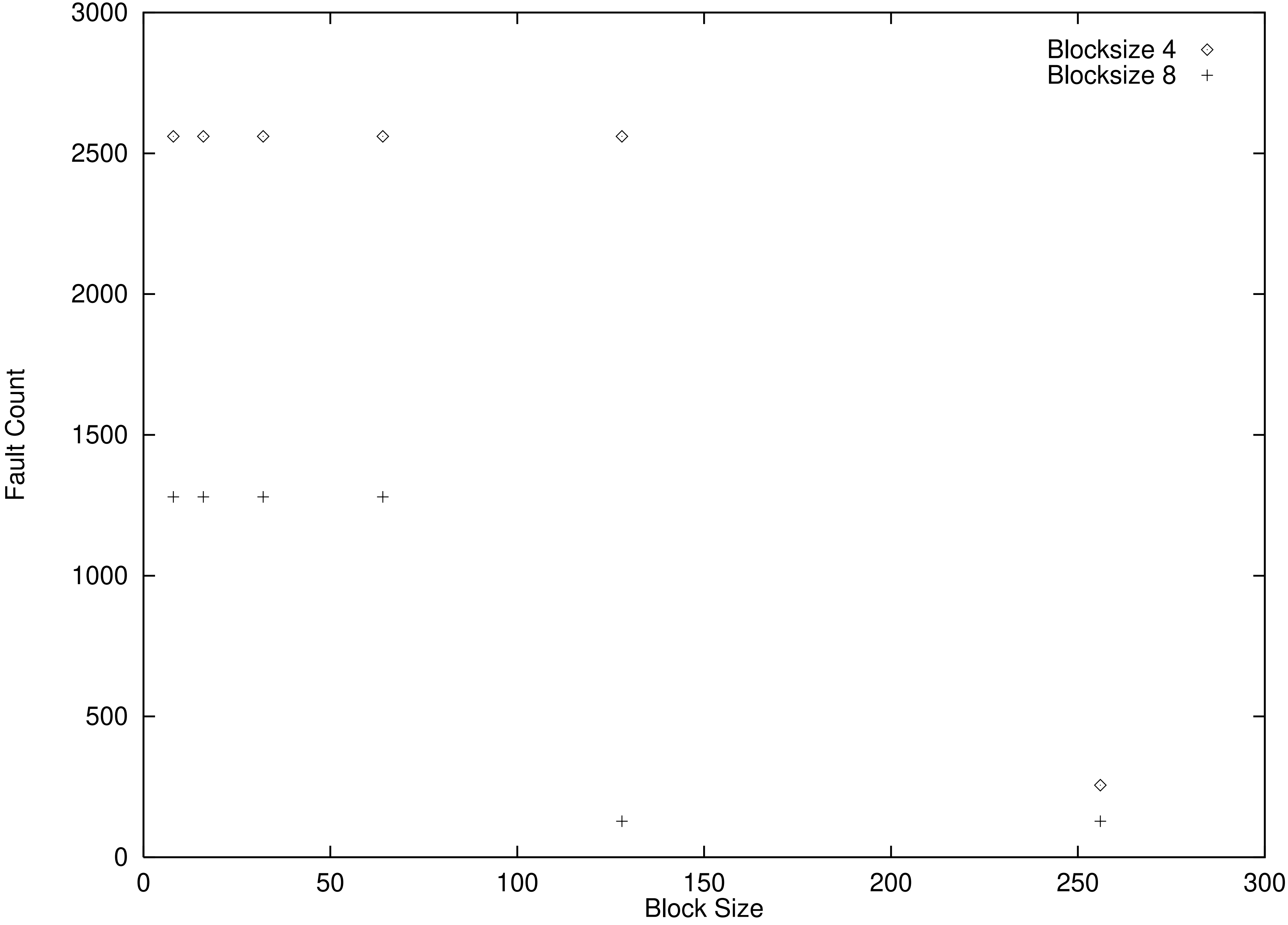
Blocksize 4
Blocksize 8
Fault Count
Block Size
3000
2500
2000
1500
1000
500
0
0
50
100
150
200
250
300

Figure 2

[1] Various specifications pertaining to the SGI-Cray Origin 2000 were obtained electronically via the World Wide Web from the NCSA: http://www.ncsa.uiuc.edu and from SGI: http://www.sgi.com.

[2] Information on the history and uses of the FFT was obtained electronically via the World Wide Web from http://www.me.orst.edu/classes/me452/winter95/ButlerKeithMurray/

We would like to thank the National Center for Supercomputing Applications (NCSA) for allowing us to use the SGI Origin 2000 system for our research.