

# Programming FFT on DSM Multiprocessors

Hongzhang Shan  
Department of Computer Science  
Princeton University  
shz@cs.princeton.edu

Jianhua Feng  
Dept. of Computer Science  
Tsinghua University  
Beijing, China  
fengjh@tsinghua.edu.cn

Hongzhong Shan  
Dept. of International Trading  
Beijing Institute of Clothing  
Beijing, China

## Abstract

*The performance of the shared address space programming model for the kinds of coarse-grained communicating programs, which have traditionally been common in scientific computing, is not clear today. In this paper, we use the challenging 1-dimensional FFT, a regular coarse-grained program, as our driving application to study how to get high performance for such kind of applications under the shared address space programming model on a hardware supported cache-coherent distributed memory machine. We find that its performance is highly affected by the data placement. Proper data placement will be critical to the success of this kind of applications. Prefetching could further improve the performance to a degree of 10 percent to 50 percent for the data sets we studied. Naive programming will easily cause the performance bottleneck by introducing much more contention and lead to great performance loss. If the shared address space programs are properly programmed, it will deliver much better performance than the other popular programming models, such as MPI and SHMEM.*

## 1 introduction

Architectural convergence has made it common for different programming models to be supported on the same platform, either directly in hardware or via software. The two most common programming models in use today are (i) explicit message passing and (ii) a cache-coherent shared address space (CC-SAS). After many vendor specific models, message passing programming has been standardized around the Message Passing Interface (MPI) designed by

the Message Passing Forum [3], which is widely used for high-performance computing in practice. Compared with message passing, shared address space programming model is much less popular for scalable high-performance computing today though it has been argued to provide programming ease to programmers. One reason for this is that scalable machines did not provide hardware support for a shared address space until quite recently (though this trend has reversed in tightly-coupled multiprocessors) and many people are not familiar with this model yet. Another reason is that the CC-SAS model was unproven for the kinds of coarse-grained communicating programs that have traditionally been common in scientific computing.

In many ways, writing parallel programs under the shared address space programming model is similar to sequential programs because of its implicit communication and data replication. The main difference is that shared memory programs need to handle synchronization among multiprocessors. However, parallel programs, which are converted from sequential programs by simply adding synchronization operations, usually could not deliver high performance on the distributed shared memory machines. Dongming [6, 4] has studied the scalability of shared memory programs on a hardware supported cache-coherent multiprocessor machine and found that the scalability does not come easily, often substantial effort will be needed.

In this paper we are going to use the challenging 1-dimensional FFT as our driving application to study how to structure the program under the CC-SAS programming model for this kind of regular coarse-grained application, the quantitative performance effect of data placement and prefetching, and the effect of naive programming. Finally we are going to compare the performance between the CC-SAS model and the more popular message passing models.

Unlike MPI, currently there is no standardized interface for shared memory programming model yet. OpenMP has been advocated by several industrial companies, but it is far from becoming a standard and its performance effect has not been explored much. We structure our shared memory programs using the MACRO interface used by SPLASH2, which provides programmers the required multiprocessor functions.

The platform we used is an SGI Origin2000, a cache-coherent distributed shared memory machine that is the most aggressive such platform today and is very widely used in high-performance computing. The CC-SAS model is directly supported in hardware on this machine. It is well known that data placement will be an important factor for high performance on this kind of machine [2]. But how critical it will be is still unclear. In this paper, we quantitatively compare the performance difference between proper and improper data placement and found that proper data placement is critical to the success of such kinds of shared memory programs on our distributed shared memory machine. For some data sets, the performance can be improved several times. The proper data placement can be implemented either by choosing the right data placement policy or changing the program code. In order to be consistent to our macro interface, we extend the macro interface by adding a keyword DISTRIBUTE before the data which need to be properly placed. It will automatically allocate the data in a round-robin fashion among all the nodes the program runs on. Prefetching is another important technology, though it's not so critical as proper data placement. By using prefetching, the performance of our program could be further improved to a degree of 10 percent to 50 percent for different data sets on different number of processors.

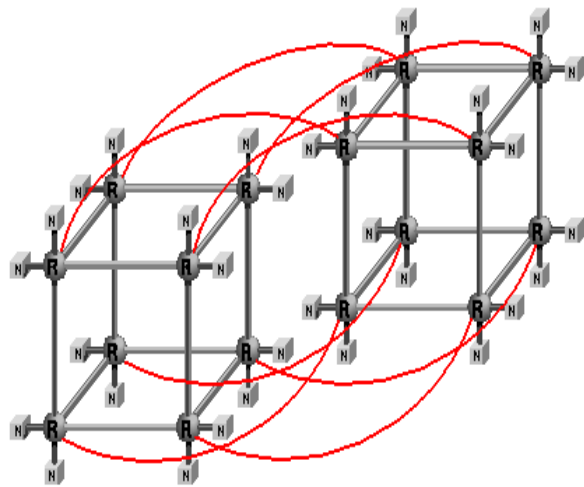
The original program we borrowed from SPLASH2 is already an optimized shared memory program. In order to avoid contention in the transpose stage, the communication is structured in a staggered way, that means processor  $i$  first fetches data from processor  $i+1, i+2, \dots, P$ , and then from  $0, \dots, i-1$ . If we use a more natural way (naive way) to implement the transpose instead of the optimized one, processor  $i$  first fetches data from processor  $0$  and then from  $1, 2, \dots, P$ , the contention will be introduced. Compared with the optimized program, the performance of the naive program dropped dramatically. This problem does not only exist for shared memory programming model, it also exists in other popular programming models, such as MPI. How to automatically reduce contention in the parallel programs is an important research issue.

One interesting question is how about the performance comparison among different programming models. Shared memory programming models has the ease of programming. Does it also deliver the better performance on this kind of hardware supported distributed shared memory machine or it has to sacrifice its performance for ease of pro-

gramming? By comparing the performance of shared memory programming model with other popular programming models (MPI, SHMEM), we found that the best shared memory programs perform much better than those best MPI and SHMEM programs. On this platform, the shared memory programming model is directly supported in hardware. The MPI and SHMEM is built in software but leverage the hardware support for a shared address space and efficient communication for both ease of implementation and performance, as is increasingly the case in high-end tightly-coupled multiprocessors. The performance difference is mainly due to the efficient hardware supported fine-grain communication. In the CC-SAS model, the transfers of cache blocks triggered by loads and stores are very efficient. In MPI or SHMEM, their message overhead could not be well amortized for smaller data sets and large number of processors, thus their performance suffers. With the increase of data set sizes, their performance becomes better, but still falls behind.

The rest of the paper is organized as follows. Section 2 describes the architecture of the Origin 2000. Section 3 describe the CC-SAS programming model and how our FFT program is structured. The performance is analyzed in Section 4, including the effect of data placement and prefetching, the effect of naive programming and the performance comparison under different programming models. Finally we summarize our key conclusion and discuss the future work in Section 5.

## 2 Platform



**Figure 1.** A 64-processor SGI Origin2000 connected in a full hyper-cube topology

The SGI Origin 2000 is a highly scalable hardware-supported cache-coherent, non-uniform memory access machine, with the most aggressive communication architecture among such machines today. It is perhaps the most widely used shared memory platforms for supercomputing today. Our machine has 64 processors, organized in 32 nodes with two 300MHZ MIPS12000 microprocessors each. Each processor has separate 32 KB first-level instruction and data caches, and a unified 8 MB second-level cache with 2-way associativity and a 128-byte block size. The machine has 16 GB of main memory (512 MB per node) with a page size of 16 Kbytes. Each pair of nodes is connected to a network router. The interconnect topology across the 16 node pairs (routers) is a hypercube as shown in the Figure 1. The peak point to point bandwidth between nodes is 1.6 GB/sec (total in both directions), and the bisection bandwidth (without the Xpress links that the machine also provides) is 10 GB/sec, And the average read latency is 796nsecs and maximum is 1010nsecs, depending on the state of the memory block in the caches. Compared with other existing machines, the SGI Origin 2000 has a very aggressive communication architecture, with a low ratio of remote to local miss penalty and a high node to network and bisection bandwidth.

### 3 Algorithm

In this section, we first describe the CC-SAS programming model and then discuss how our FFT program is structured.

#### 3.1 CC-SAS Programming Model

The cache-coherent SAS model is supported in hardware, with hardware support for a shared address space and implicit local replication in the cache at the granularity of a cache block. The data structures used can be the same as in a sequential program; only those that are accessed by more than one process are declared in the shared address space. Remotely allocated data are accessed just like locally allocated data or data in a sequential program, using ordinary loads and stores. A load or store that misses in the cache and must be satisfied remotely communicates the data in hardware at cache block granularity, and automatically brings it into the local cache. The transparent naming and replication provides programming simplicity, especially for dynamic, fine-grained applications. In our implementation, the initial or parent process spawns off a number of child processes, one for each additional processor. These cooperating processes are then assigned chunks of work using static assignment. The synchronization structures used are locks and barriers. Processes are spawned once near the beginning of

the program, do their work, and structuring, and application restructuring are used to improve performance [8, 5].

The most often used macro interfaces are :

- `GMALLOC(size)` : to malloc shared data
- `LOCK(lock-name)` : to enter the LOCK
- `UNLOCK(lock-name)`: to exit the LOCK
- `BARRIER(bar-name, P)` : to synchronize P processes
- `DISTRIBUTE var` : to properly distribute the data *var*, processor *i* will get range  $\text{size}(var) \cdot i/P$  to  $\text{size}(var) \cdot (i+1)/P$

#### 3.2 FFT Program

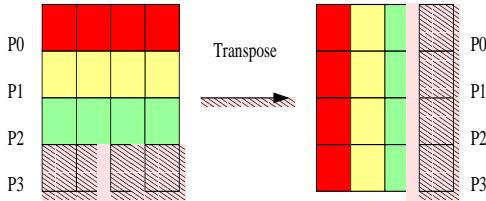
FFT is a coarse-grained regular application with personalized all-to-all communication. The original code directly borrowed from SPLASH2 parallel application suite, is a double-precision complex 1-D version of the radix- $\sqrt{r}$  six-step FFT algorithm described in [1], which is optimized to minimize the interprocess communication. 1-D FFTs are more challenging than higher-dimensional FFTs, since there is more communication relative to computation. The data set consists of  $n$  double-precision complex data points to be transformed (in fact, two arrays of these points), and another array of double-precision complex data to be used as the roots of unity. The  $n$ -point data set is arranged in the form of a  $\sqrt{n} * \sqrt{n}$  matrix for this high-radix implementation, and the matrix is partitioned among the processors in blocks of  $\sqrt{n}/p$  contiguous rows each. Each processor stores its  $\sqrt{n}/p$  rows in its local memory ( $p$  is the total number of processors), as shown in figure 2. The matrices are transposed three times, alternating which matrix is the input to the transpose and which is the output.

The whole FFT structure is as follows:

- (i) transpose matrix,
- (ii) perform 1-D FFTs individually on local rows of size  $\sqrt{n}$  each,
- (iii) multiply the elements of the resulting complex matrix by the corresponded roots of unity,
- (iv) transpose matrix,
- (v) perform 1-D FFTs individually on local rows,
- (vi) transpose matrix.

In each transpose stage, each processor communicates a sub-matrix of size  $\sqrt{n}/P * \sqrt{n}/P$  to every other processors, resulting in coarse-grained and regular (value-independent and completely predictable a priori) all-to-all personalized

communication. An example is shown in 2. A blocked transpose is used to exploit cache line reuse. To avoid memory hot-spotting, the sub-matrix is transmitted in a staggered way, That means processor  $i$  first fetch data from processor  $i+1, i+2, \dots, P$ , and then from  $0, \dots, i-1$ .



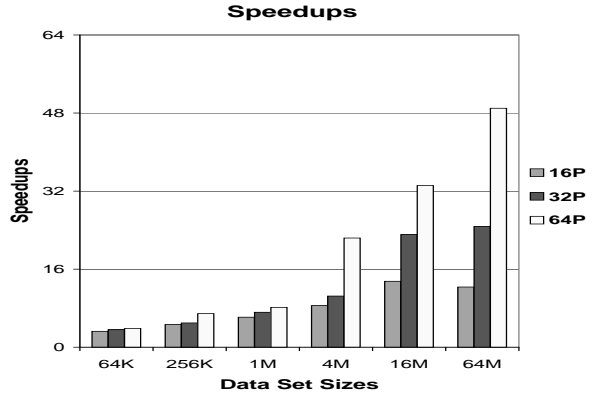
**Figure 2.** The transpose of FFT with 4-processor case

## 4 Performance

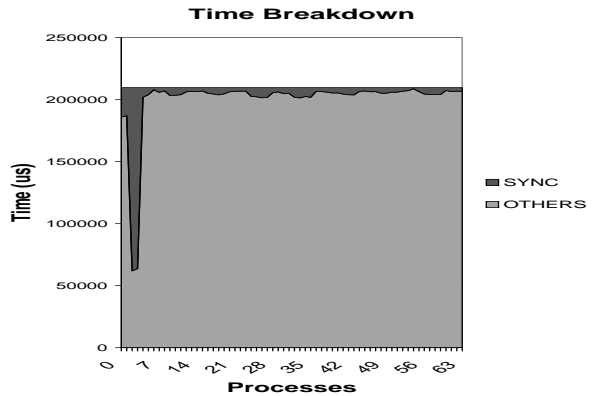
Now let’s look at the performance. We will examine the speedups on 16, 32 and 64 processors for different data set sizes. The sequential wall-clock times of our original code for different data set sizes are shown in Table 1. The data set size we selected covered a very wide range from 64K to 64M complex double data. The much smaller data set sizes are not included here since they are not so interesting to a large multiprocessor machine. Our general approach is to examine a range of potentially interesting problem sizes at each machine size.

The speedups of the original program are shown in Figure 3. We find that the speedups are very low for data sets up to 4M size, the parallel efficiency is below 50% on all number of processors. For 64K data set size, from 16 processors to 64 processors, there is almost no any improvement. Increasing problem size is helpful. The speedups on 64 processors for 64M data size approximates 50. This is because increasing data set size will generally reduce the communication to computation ratio. In FFT, the communication to computation ratio will diminish logarithmically with the problem size. However the data set size required to achieve higher parallel efficiency is relatively too large here.

In order to analyze the performance bottleneck, let’s examine the per-processor time breakdown in Figure 4. We divide the per-processor wall-clock running time into two categories: CPU time spent for synchronization events (SYNC) and others including CPU computation time and CPU stall time for cache misses. We found that the SYNC time is extremely unbalanced. For very few number of processors, the SYNC time is very high. This problem is related with data placement.



**Figure 3.** Speedups of original program on 16, 32, and 64 processors for different data set sizes



**Figure 4.** The per-processor time breakdown of 1M data set size on 64 processors for original program

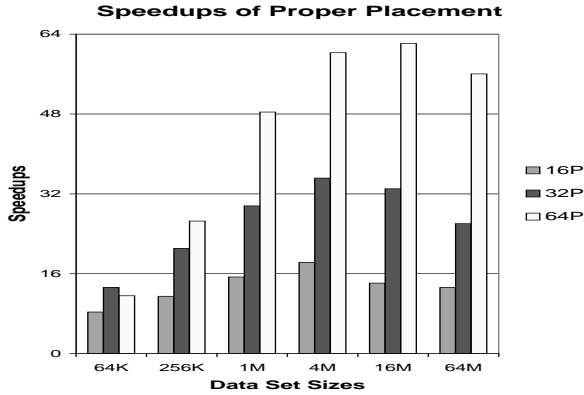
### 4.1 Effect of Data Placement

By carefully analyzing the original program code, we found that the data initialization work is all done by one processor. Since the default data placement policy on this machine is first-touch. Under this policy, the process that first touches a page of memory causes that page to be allocated in the node on which the process is running. Having all the data concentrated on one node or a small radius of it creates a bottleneck: all data accesses are satisfied by one hub, and thus limits the memory bandwidth. The initialization process (or other processes which resident on the same node with initialization process) will access all the data locally while others remotely, thus it will spend much time on synchronization.

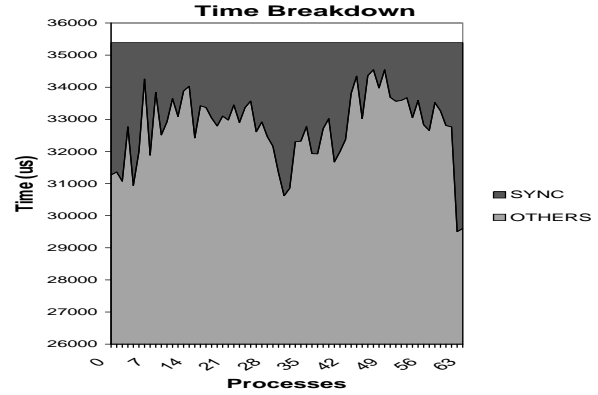
This problem can be easily solved by either changing the data placement policy or modifying the initialization process from sequential to parallel. The data placement pol-

Data Sets:	64K	256K	1M	4M	16M	64M
Time(ns):	53176	294254	1712836	8800127	42244034	198179949

**Table 1.** The sequential execution time (microseconds) of different data set sizes.



**Figure 5.** Speedups of program with proper data placement on 16, 32, and 64 processors for different data set sizes



**Figure 6.** the per-processor time breakdown of 1M data set size on 64 processors with proper data placement

icity can be changed from first-touch to round-robin, under which data are allocated in a round-robin fashion from all the nodes the program runs on. There are several ways on this platform to finish this purpose. We can use the compiler directives, use the data placement tools or set up environment variables. In order to be consistent with our programming interface, we provide a new macro “DISTRIBUTE”, which will automatically allocate those data following it in a round-robin way under proper granularity. However, the better way to solve this problem is by parallel initialization. Instead of letting one process initializing all the data, we allow all the processes share the initialization work equally. Each process is responsible for those rows from  $i * \sqrt{n}/P$  to  $(i + 1) * \sqrt{n}/P$ , where  $i$  is the process index,  $n$  is the data set size and  $P$  is the number of processes.

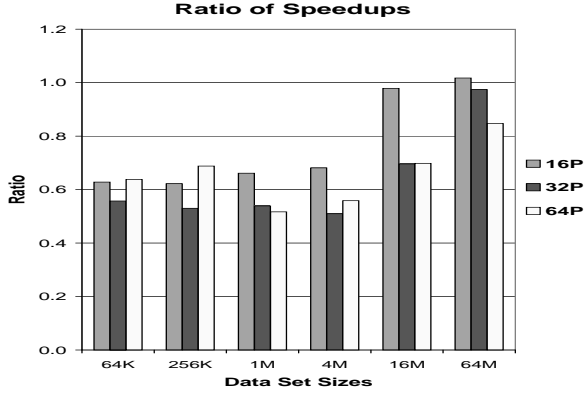
By doing the proper data placement, the speedups of FFT has been improved greatly. The new speedups are shown in Figure 5. At 1M data set size, the parallel efficiency has reached over 60 percent on all 16, 32 and 64 number of processors. The improvement for 64K data set size on 64 processors is as large as 6 times. The new per-processor time breakdown for 1M data set size on 64 processors is shown in Figure 6. The SYNC time are much more balanced now compared with that in Figure 4. (note that the Y scale is different)

## 4.2 Effect of Naive Programming

Our implementation of the transpose communication are algorithmically optimized to reduce contention (though

contention, of course, nonetheless occurs in the machine). The transpose is staggered, so process  $i$  communicates first with process  $i + 1$ , then with  $i + 2$  etc (with wraparound). This avoids all processes communicating with the same process at the same time. A more naive, but also more natural, way to structure the loop is to have all processes start communicating with process 0, then process 1, and so on, causing contention (hot-spotting) at one process at every stage. This version of FFT can be used to examine the effect of such algorithmic contention on the performance of the shared address space programming model. The speedup comparison between the naive method and the optimized method are shown in Figure 7 for the different data sets (the ratio of the speedups of naive method to speedups of optimized method).

There is a big gap between performance of the optimized method and the naive method for data sets from 64k up to 4M. In most of these cases, the performance of the naive method stays around 60 percent of the performance of the optimized method, in the worst case, only 50 percent. The performance loss of using the naive non-staggered transpose is large, since there are not only many small messages contending at the end point in this case but also protocol messages like invalidations and acknowledgments. Surprisingly we found that for larger data set sizes, such as 64M complex data set, using naive method has much less effect on the performance. One reason is that the communication to computation ratio diminishes logarithmically with the increasing of the data set size, thus the effect of the non-staggered transpose becomes less. Another reason is that



**Figure 7.** Ratio of speedups of naive method to speedups of natural method

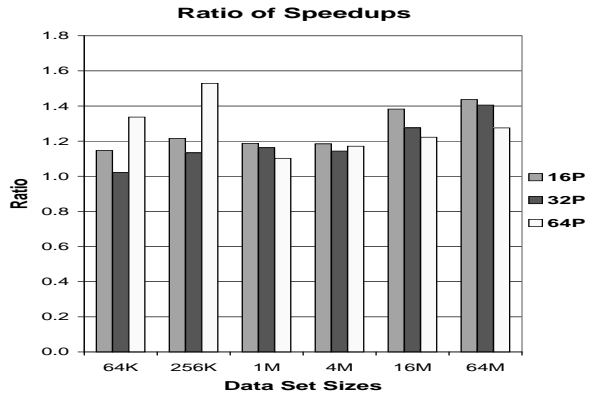
for larger data sets, the effect of cache misses and TLB misses will become more important.

### 4.3 Effect of Prefetching

In the above sections, the program is structured in six steps and there are three independent transpose steps, which are totally communication without any overlap with computation. The transpose stage occupies 50 percent of the total execution time. One interesting idea is to overlap the communication with computation by combining the transpose with local fft operations. In this way, we can take advantage of the prefetching. And the corresponding program structure will be as following:

- (i) For each local row
  - fetch the data for this row from other processes
  - perform 1-D FFTs individually on this row of size  $\sqrt{n}$
  - multiply the elements of the resulting complex matrix by the corresponded roots of unity
  - End
- (ii) For each local row
  - fetch the data for this row from other processes
  - perform 1-D FFTs individually on this row of size  $\sqrt{n}$
  - End
- (iii) transpose matrix

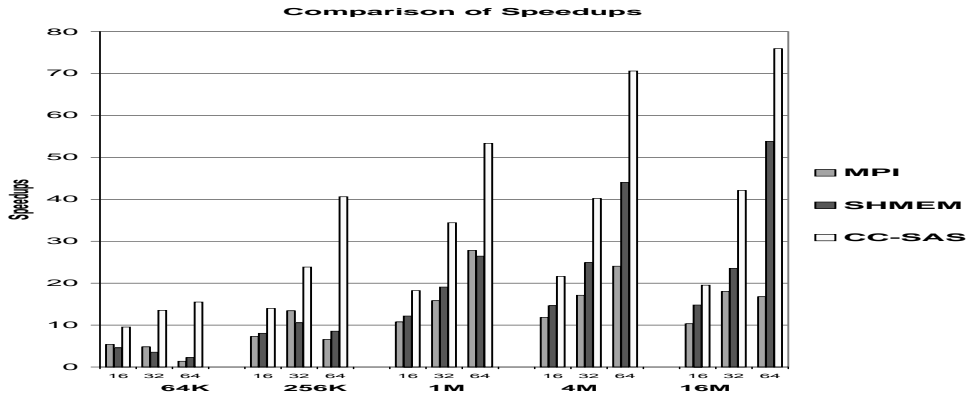
Since the cache block size on this machine is 128 bytes and each element is  $8 \times 2$  bytes (complex data), thus during the above loop (i) and (ii), when one process fetches data for one row from all others, suppose row  $i$ , it actually also gets the data for the next 7 rows from row  $i+1$  to  $i+7$ . Thus we can prefetch row  $i+8$  during the period of local FFT computation for this 8 rows. The performance effect is shown in Figure 8, which shows the ratios of the speedups of prefetching program versus the non-prefetching program. In most of the cases, the performance improvement ranges from 10 percent to 50 percent. The effect becomes larger with the increase of data set sizes.



**Figure 8.** Ratio of speedups of prefetching method to speedups of non-prefetching method

### 4.4 Performance Comparison of Different Programming Models

One interesting question is how about the performance comparison among different programming models. Shared Memory programming model is argued to provide programming ease to programmers over message passing models. Does it also delivers better performance on this hardware supported cache-coherent machine or it has to sacrifice performance for ease of programming? Our focus here is for the 1-dimensional FFT only. In order to compare their performance, we converted the shared address space program into MPI and SHMEM programs, their performance are equal or superior to other known algorithms to us [7]. SHMEM is like MPI in that communication and replication are explicit and usually made coarse-grained for good performance; however, unlike the send-receive pair in MPI, communication in SHMEM requires process involvement on only one side (using `put` or `get` primitives) and SHMEM allows a process to name or specify remote data via a local name and a process identifier. Compared with MPI, the SHMEM routines minimize the overhead associ-



**Figure 9.** Speedup comparison of different programming models on 16, 32, and 64 processors for different data set sizes

ated with data passing requests, maximize bandwidth, and minimize data latency. The best speedups for different programming models are shown in Figure 9.

From Figure 9 we find that the performance of the shared address space model is much better than the other two programming models. From 4M data set size, it delivers super-linear speedups on 64 processors. This is mainly due to the efficient hardware supported fine-grain communication. In the CC-SAS model, the transfers of cache blocks triggered by loads and stores are very efficient for the fine-grained communication needed. For example, for the 64K data size, the size of a packed message being communicated is 256 bytes, which is two cache block size on SGI Origin 2000. While in MPI and SHMEM, the message overhead can not be amortized well for smaller data sets. This is worse in MP than in SHMEM, since an explicit send and a matching receive must be initiated by sender and receiver separately for each communication, potentially increasing not only messaging overhead but also synchronization time, since the sends and receives have to be posted in timely ways and matched. With the increase of data set size, message size increases and explicit message passing becomes more efficient, so the performance of MP and SHMEM becomes better, but still far behind.

The result of performance comparison is somewhat different from what we presented in the paper [7]. There, we found that for smaller data sets (up to 1M) shared address space model works much better than the other two programming models; for larger data sets, the three models performs similarly. One reason is that our machine has been upgraded. The CPUs have been upgraded from MIPS R10000 (195Mhz) to R12000 (300MHz). The secondary unified instruction/data cache size increased from 4M to 8Mbytes. It seems that the performance effect of system changes has larger effect on the shared memory programming models than the other two programming models. The study of the effect of the system configuration changes is

undergoing. Another reason is that we have not included the prefetching effect in the paper [7]. Without the effect of prefetching, the shared memory model still works better than the other two models, but the difference becomes smaller.

## 5 conclusion

The shared address space programming model has the ease of programming over other parallel programming models, such as MPI. However, programs converted from sequential version to parallel version by simply adding the required synchronization operations usually does not deliver high performance. In this paper, we use the challenging 1-dimensional FFT as our driving application to study how to achieve high performance on a hardware supported cache-coherent machine for such kind of regular, coarse-grained application. We first studied the program structure of the FFT, then we quantitatively studied the performance effect of data placement and prefetching. We found that the performance is highly affected by the data placement. Proper data placement will be critical to the success of this kind of applications on a distributed shared memory machine. Prefetching could further improve the performance to a degree of 10 percent to 50 percent for the data sets we studied. Naive programming will easily cause the performance bottleneck by introducing much more contention and lead to great performance loss. However, if the shared address space programs are properly programmed, it will deliver the much better performance than the other popular programming models, such as MPI.

Our future work will include modeling the performance for shared address space programs to find out how the performance will be affected by system configuration.

## References

- [1] David H. Bailey. FFTs in external or hierarchical memories. *Journal of Supercomputing*, 4:23–25, 1990.
- [2] David Cortesi. Origin 2000 and onyx2 performance tuning and optimization guide. <http://techpubs.sgi.com>, 1997.
- [3] Message Passing Interface Forum. Document for a standard message-passing interface. <http://www-c.mcs.anl.gov/mpi>, June 1993.
- [4] Dongming Jiang, Hongzhang Shan, and Jaswinder Pal Singh. Application restructuring and performance portability across shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.
- [5] Dongming Jiang and Jaswinder Pal Singh. An methodology and an evaluation of the sgi origin2000. In *Proceedings of ACM Sigmetrics98 / Performance 98*, June 1998.
- [6] Dongming Jiang and Jaswinder Pal Singh. Does application performance scale on modern cache-coherent multiprocessors: A case study of a 128-processor sgi origin2000. In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.
- [7] Hongzhang Shan and Jaswinder Pal Singh. Comparison of message passing, SHMEM and cache-coherent shared address space programming models on the SGI Origin 2000. In *International Conference on Supercomputing*, June 1999.
- [8] S.C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, June 1995.