Optimizing FFT for HPCC

Mark P. Sears
Courtenay T. Vaughan
Sandia National Laboratories
Presented at the Cray User Group Conference
Helsinki, Finland, May 2008

May 27, 2008

Abstract

One component of the HPCC benchmark for high performance parallel computing platforms is a very large one dimensional FFT. This benchmark stresses the memory and communications bandwidth at all levels of the memory hierarchy. In this paper we discuss some of the algorithmic, implementation, and performance issues for the FFT HPCC benchmark that we developed for the Sandia-Cray Red Storm architecture.

1 Introduction.

Suppose we want to do a large one dimensional discrete Fourier transform (DFT), say of length N. If N can be factored (say the factors are n, m so that $N = nm$) then we can use a factorization theorem which allows the DFT of a vector of length N to be written as n DFT operations of length m each followed by a so-called twiddle operation and then m DFT operations of length n. This theorem is the basis for the standard FFT algorithms which

are just applications of the divide and conquer method. The other tricks in high performance serial FFT implementations (the well known FFTW package is a good example) are first the use of carefully written short length DFT codelets and second the tuning of the choice and order of which codelets to use. For example, if $N = 16$ then one could use a codelet for length 2 followed by a codelet of length 8, or the reverse order, or two codelets of length 4 each, or a codelet written specifically for length 16. The more factorable N is the more possible choices there are and choosing the correct factors and order can have a big effect on performance. One can expect that tuned FFT algorithms of this kind will achieve a significant fraction of the available performance on modern CPU architectures, on the order of 50 percent. The limiting factor is memory bandwidth, including effects of cache.

The FFT problem is thus pretty much solved for serial architectures with modest N and the resulting library can be applied to serial one, two, and three dimensional applications. For multidimensional parallel applications there is still some room for better algorithms although the basic approaches are well known. The usual method to do a 3d parallel FFT for example is to apply a serial FFT algorithm to one axis which is in memory on each process, the other two axes being distributed in some way. Then a transpose operation brings the second axis into memory and serial FFTs are applied to that axis. A second transpose operation brings the third axis into memory for the third FFT. Generally an attempt to parallelize the FFT will end up using the same message passing operations as the transposes and the code is merely harder to write with no performance benefit.

For the HPCC benchmark FFT however we have to do a very large one dimensional FFT, so large that it must occupy a significant fraction of the memory on each process. The details of the implementation are open, and although a standard implementation is available any particular benchmark run can use code written and tuned for that particular architecture. In order to see exactly how such an implementation can work we go through the development of the factorization theorem to see how it can be used for a very large distributed one dimensional FFT.

2 The DFT factorization theorem.

Suppose we have a complex vector u of length N where N is factorable as described above. Let $\omega_N = e^{\frac{2\pi i}{N}}$. Then the DFT is given by the transform

$$\tilde{u}(K) = \sum_{J=0}^{N-1} \omega_N^{JK} u(J) \tag{1}$$

The basic trick behind the factorization theorem is to note that a vector of length N can be thought of as a matrix which is $m \times n$ or as a matrix which is $n \times m$. In fact we use one of these decompositions for u and the other for $\tilde{u}$. We can break up the index K into $K = p + qm$ say where the digit p is in the range $[0 : m-1]$ and the digit q is in the range $[0 : n-1]$. Similarly $J = s + nt$ where the digit s is in the range $[0 : n-1]$ and the digit t is in the range $[0 : m-1]$. Rewriting the transform we get

$$\tilde{u}(p,q) = \sum_{s=0}^{n-1} \sum_{t=0}^{m-1} \omega_N^{(p+qm)(s+tn)} u(s,t) \tag{2}$$

and using $mn = N$, $\omega_N^N = 1$, $\omega_N^m = \omega_n$, $\omega_N^n = \omega_m$ and expanding the product $(p+qm)(s+tn)$ we can break up the DFT into two steps (using an intermediate complex array v)

$$v(s,p) = \omega_N^{sp} \sum_{t=0}^{m-1} \omega_m^{pt} u(s,t) \tag{3}$$

followed by

$$\tilde{u}(p,q) = \sum_{s=0}^{n-1} \omega_n^{qs} v(s,p) \tag{4}$$

Here the factor ω_N^{sp} is the twiddle factor. The first step is a DFT of length m on the second index of the array u, followed by multiplication by the twiddle factor. The second step is a DFT of length n on the first index of the intermediate array v, but the result is out of order and must be transposed. This is the source of the shuffling or digit-reversal requirement for ordinary serial FFT algorithms, and in our large parallel one dimensional FFT this will be the source of the transpose requirement.

In the usual derivation of the FFT algorithm we take one of the factors n or m to be very small, say 2 and then the small DFT is written out. For a factor of two the phase factor of ω_2 is just -1, so the codelet for this DFT involves only addition and subtraction and we can deduce the Cooley-Tukey algorithm for example. Repeatedly taking one or the other factor like this leads to two kinds of algorithms sometimes called frequency or time based schemes. We can also do this algorithm in reverse.

The first transform in the factorization algorithm is done on the second axis. We call this an *outer* transform and the corresponding transform on the first axis is an *inner* transform. Note that the stride between elements in the inner transform is 1 and the block size (number of transforms to do) is m. Accessing memory in this way generally performs well. On the other hand for the outer transform we have a stride of n and as noted below this can have a deleterious effect on performance.

Counting flops (floating point operations) and memory accesses for the transform as a whole we have

$$N_{\text{flops}} = 5N \log_2(N) \tag{5}$$

This is not a correct statement if we use anything other than the Cooley-Tukey algorithm, for example if codelets longer than 2 are used. However this is the standard expression for flop counts in use, so that different transform implementations can be compared on the same footing.

The number of memory accesses in bytes for the transform as a whole is

$$N_{\text{bytes}} = 16N \tag{6}$$

ignoring transfers to buffer space, instructions, constants, etc. Therefore the ratio of flops to bytes is

$$\frac{N_{\text{flops}}}{N_{\text{bytes}}} = \frac{5}{16} \log_2 N. \tag{7}$$

Suppose $N = 256$. Then this ratio is 2.5 flops/byte. If the memory bus is capable of 1GB/sec then we might expect performance on the order of 2.5Gflop/sec to be acheivable, and within a factor of 2 this is so for modern

microprocessors, especially when the transform data is in (at least second-level) cache. Out of cache performance is often much lower, as for three dimensional FFTs.

3 Parallel one dimensional FFT algorithm.

For a large parallel one dimensional FFT algorithm we can use the factorization theorem as it stands and take one factor to be the number of processes P. Then we can decompose the array u so that the first axis of the array is stored on the process and the second axis is distributed over the processes. In order to do a FFT on one of the axes we have to bring it into local memory, but then all the FFTs for that axis can be done simultaneously on all the processes. So the inherently parallel operations are just these parallel transposes.

The parallel one dimensional algorithm is thus given as follows. The comment (Parallel) means that parallel communications is required to implement that step:

- Step 1. (Parallel) Transpose $u(s,t) \rightarrow v_1(t,s)$. This brings the t axis into local memory on each process.
- Step 2. Perform length m FFTs and twiddle. All FFTs and multiplications are local to each process.

$$v_2(p,s) = \omega_N^{sp} \sum_{t=0}^{m-1} \omega_m^{pt} v_1(t,s) \tag{8}$$

- Step 3. (Parallel) Transpose $v_2(p,s) \rightarrow v_3(s,p)$. This brings the s axis into local memory on each process.
- Step 4. Perform length n FFTs. Again, all FFTs are local to each process.

$$v_4(q,p) = \sum_{s=0}^{n-1} \omega_n^{qs} v_3(s,p) \tag{9}$$

- Step 5. (Parallel) Transpose $v_4(q,p) \rightarrow \tilde{u}(p,q)$ to generate correctly ordered output.

Note that there are three transpose operations! This is unavoidable if (as required for the HPCC benchmark) the data must be stored in order on each process both at the beginning and end of the algorithm. Because of the number of transpose operations the overall performance of the parallel transform is limited by the communications bandwidth. Suppose this is 1GB/sec. Since we have three transposes and each one essentially moves all of the data we can estimate the performance in flops/byte per process as

$$\frac{5}{96}\log_2(N) \tag{10}$$

where we used a factor $96 = 6 \times 16$ because the data has to be both sent and received on every process and there are three transposes. Even though N is much larger than 256 for the benchmark we still must expect a very significant reduction in FFT performance from what is available from serial libraries operating on in-cache data.

4 Parallel transpose operations.

Suppose we have P processes and a $P \times P$ matrix, where element $M(i,j)$ is owned by process j for $i = 1:P$. In other words, the first axis of the matrix is local to the processes and the second is distributed. If we want to transpose the matrix then each process will need to send and receive some amount of data to every other process. In fact process p must send one element of the matrix to process q, for every pair p, q.

To generalize, assume the matrix has dimensions $kP, k'P$, with block decomposition of the second axis only. It is possible to further generalize to arbitrary decompositions but this will not be necessary for this paper. Then each process p must send a block of memory which is $k \times k'$ to each process q, and vice versa, for all pairs. But now assuming a simple storage scheme for the source and destination matrices the data that p sends to q is not contiguous and must be packed up before sending. Moreover the receiving process q

must take the message and unpack it to be placed in the destination matrix, and in either the packing or unpacking process an inner transpose must also occur (from a block which is $k \times k'$ to a block which is $k' \times k$).

The other issue in the transfer of data is how messages are scheduled. The data transfer is an all to all transfer, that is each process will be sending data to every other process and receiving data from every other process. The MPI communications interface supports such transfers and it is possible to write code which turns the whole thing over to the MPI implementation. The possible benefit of this that the implementation can potentially use lower level operations which work faster than the method described below, but the drawback is that the developer has no control over what the implementation decides to do. Below we describe an approach that just uses point to point communications. This approach illustrates a number of the issues that any implementation (MPI internal or not) would have to consider.

5 Packing and unpacking.

The issues of packing and unpacking at first seem trivial. However, the amount of data that needs to be packed for the benchmark is a large fraction of the available memory on each process, which far exceeds the capacity of the lower levels of cache. A poorly written access pattern can interact very badly with the cache hierarchy and memory management system (page tables, TLB entries, etc.). A basic rule of thumb for dealing with cache is to try to make as many references to each cache line as possible before turning to another cache line, where a cache line is the quantum of information that gets transferred between levels in the hierarchy. Typical cache line sizes for microprocessors are on the order of 128 bytes. If we read a complex number (16 bytes) from a cache line, then go to a different cache line to read another and so forth then the effective performance of memory is reduced by a factor of eight (128/16). This could happen when a vector (stream) access pattern is used where the vector has a large stride.

Similarly if we access a small part of a page before accessing other pages then the page tables and TLB entries can be overwhelmed and again performance will suffer. The cure for both cache performance and page table/TLB per-

formance is contiguous access. Although eventually we will run out of cache entries or TLB entries and we will have to evict cache entries before bringing in new ones, still the cost is amortized over the accesses that are in-cache.

6 All to all parallel communications.

A very simple all to all scheme is to just send P messages to all P processes, then receive P messages. This is obviously bad because when a message arrives on a process there is probably no matching receive and therefore the message will go into an unexpected queue. Worse, the unexpected queue may be of limited size and chances are that if the exchange involves large amounts of memory the MPI implementation is free to crash. Even if this method worked the received data would have to be copied out of the unexpected queue buffer space. In a better algorithm we would like to ensure that when messages are received there is a receive buffer already in place to avoid the unexpected queue and the extra buffer copy.

A first alternative is for each process to prepost all the messages that it will receive, then send all the messages it is going to send, then wait for all the messages to complete. This is not a bad approach for parallel machines or applications involving modest numbers of processes. But it is not scalable: when P is on the order of 1000 then we have to prepost 1000 messages and the MPI layer must check to see if an incoming message matches any of these 1000 requests. Moreover, the amount of message space that we have to allocate is also proportional to P.

A much better approach is to arrange the sending and receiving so that processes are paired. That is, we want to guarantee that when process q is supposed to be exchanging with process q' then process q' is also exchanging with process q. It is not obvious that this is possible, but there is a simple and elegant way to guarantee that the exchanges occur pairwise with no loss of time. This is shown in the following algorithm:

```
// on process q, loop over P exchanges:
for(s=0;s<P;s++)
```

```
{
 qp = (P - q + s) % P;
 exchange(q,qp)
}
```

It is easy to see that as the loop variable `s` sweeps over the numbers $[0 : P-1]$ that the expression for the dual process `qp` sweeps over these numbers as well, so the exchange function is executed with all the other processes (including process `q` itself). Consider a fixed step and compute `qp` for a given `q`. Now on process `qp` we can compute the dual process that it computes, and this turns out to be `q`. So at each step all of the processes are paired and as the loop proceeds all processes pair with all other processes, with no interference. There is no wasted time in the loop.

With this approach the pressure on matching and queue space is vastly reduced. In fact the number of messages outstanding on each process at any time is reduced from $O(P)$ to $O(1)$, independent of the size of P. Thus this approach is perfectly scalable from this point of view. In the communications fabric there are $O(P)$ messages running around at any time. The fabric may have limitations on bandwidth due to topology that will cause more limited performance for large P. This will show up as messages or parts of messages try to occupy the same links and collide, and the effective bandwidth will thus be reduced. In fact a simple version of this algorithm can be used to detect collisions and is an excellent test of the fabric reliability. We note also that the fabric may allocate virtual channels for messages that exclude other messages from sharing the bandwidth, and this may cause the algorithm to behave poorly. Other fabric implementations may quantize the messages into packets and thus effectively share the bandwidth, which is better. If the fabric allocates channels and this becomes a problem then the algorithm can easily be rewritten to packetize messages at the application level. Finally, note that the paired nature of the messages being sent can also be a problem if the messages in both directions use the same path through the network.

Another advantage of this algorithm is that the temporary buffer space needed is now much smaller, also by a factor of P. In terms of the block sizes above we only need two buffers of size $k \times k'$, one for sending and one for receiving.

The exchange operation must be carefully written. The simplest approach is to post a receive, pack the data to be sent, send the data, then wait for the receive to complete, then unpack the data. The MPI progress rules guarantee that this must eventually complete successfully. But there is no guarantee with this approach that when a process sends data that the dual process has posted its receive, and there are a number of scenarios that can take place. Possibly the other process does have its receive posted and all goes well. Otherwise part or all of the message will go into the unexpected queue before the receive begins, with the extra overhead that involves.

A better exchange algorithm will first exchange short PTS (permission to send) messages before exchanging the larger buffers. With this scheme each process posts its receives, sends a PTS message, waits for the corresponding PTS message, sends the data, then waits for the receive to complete. The message ordering guarantees that the receive buffers are posted before message data arrives and no large volume of message buffer space is needed. This is a place where an MPI implementation can potentially outperform this explicit point to point code, by using some lower level primitives to possibly synchronize the exchange more quickly. The appropriate routine is `MPI_SendReceive`. But we note that the point to point version allows for the overlapping of the packing and unpacking with the send/receive operations, which `MPI_SendReceive` does not.

7 Benchmark results.

The HPC Challenge (HPCC) benchmark suite [1] which provides a variety of benchmarks that span the space of processor and network performance for parallel computers. These benchmarks include HPL (factor a large dense matrix) which emphasizes processor performance, PTRANS (matrix transposition) which tests network bisection bandwidth, STREAMS (vector operations) which tests memory performance, RandomAccess (modify random memory locations across the entire machine) which stresses small message network performance, and FFT.

The rules for the HPCC competition state that the optimized routines have to be called from the structure present in the framework and that the for-

ward transform performed by FFT routine has to be verified by the inverse transform that is present in the framework. For each of the transforms there is a planning stage in which the N for the transform is factored and order and number of smaller transforms is planned. During this phase, memory is allocated and some of the twiddle factors are precomputed. The only phase that is timed is the forward transform.

We have tested this algorithm with two versions of the HPCC benchmark suite, which differ on how the FFT benchmark is implemented. With version 1.0, the FFT benchmark is run on the largest power of two number of processors that is contained in the total number of processors used to run HPCC. With version 2.0, this was extended to run on the largest number of processors that can be factored by the numbers 2, 3, and 5 that is contained in the total number of processors.

Red Storm is a Cray XT4 located at Sandia which has 12960 dual core AMD Opteron processors running at 2.4 GHz. We ran HPCC version 1.0 on 25920 cores on Red Storm with the baseline FFT algorithm and the optimized algorithm. The FFT with those tests ran on 16384 cores and with the allocation algorithm, those cores are on a mix with 6848 cores running in dual core mode and 9536 cores running in single core mode. With the baseline algorithm, we got 1554 GFLOPS and with the optimized algorithm, we got 2871 GFLOPS. This is almost a factor of two performance increase with the optimized algorithm. This won the FFT category at the HPCC awards at SuperComputing 2007. We have been able to run HPCC version 1.2 on 16384 cores on 8192 nodes of Red Storm and got 1234 GFLOPS with the baseline algorithm and 2272 GFLOPS with the optimized algorithm. This again is almost a factor of two better performance for the optimized algorithm. On 25920 cores, the baseline algorithm got 2755 GFLOPS and we were not able to get a result from the optimized algorithm that validated (we have since fixed the bug, but did not get a chance to rerun the algorithm).

8 Conclusions.

We have implemented an optimized FFT algorithm within the HPCC benchmark suite which resulted in almost a factor of two improvement over the

baseline algorithm on Red Storm and winning the FFT category within the HPCC competition at SuperComputing 2007. The algorithm relied on a better use of the cache and a faster algorithm to perform the all to all communication that is present in the algorithm. The algorithm performance measured by FLOPS per core shows a reduction going from serial to parallel due to the communication required, but in parallel, the algorithm shows excellent scaling with a consistent FLOPS per core as the number of cores is increased.

9 Acknowledgments.

This research was sponsored by Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

10 References.

1. P. Luszczek, J. Dongarra, D. Koester, R. Rabensiefner, R. Lucas, J. Kepner, J. McCalpin, D. Baily, and D. Takahasi, Introduction to the HPC challenge benchmark suite, March 2005, http://icl.cs.utk.edu/hpcc/pubs/index.html.