Insomnia

Robert Boyer
Patrick Chase

Abstract

The performance of parallel programs is significantly affected by the organization of communications and the direct affects of hardware architecture and constraints. Use of the standard MPI libraries often fails to give the best performance.

This work describes and evaluates a variety of performance enhancements that are generally applicable to any parallel programming project. Manually tuning communications and packing/unpacking data with regard to specific hardware characteristics is shown to be superior to default MPI performance within certain constraints. Overlapping communications with processing is demonstrated and the application of threading to consume idle time is described. Memory contention and its effects on overlapping communications and processing is evaluated.

1. Introduction

A problem common to all parallel programming tasks are understanding and making effective use of the hardware and available communications libraries. This project explores several techniques that are generally applicable to parallel efforts, and measures their performance impact on the NPACI Blue Horizon parallel computer.

The paper is organized as follows. Section 2 identifies the hardware / software / operating systems employed. Section 3 describes optimization of low-level copy performance, which is integal to all further testing. Section 4 discusses performance of MPI pack/unpack versus application solutions. Section 5 evaluates intra-node memory bandwidth contention for a transpose problem, while section 6 evaluates static versus dynamic work allocation within the same problem. Section 7 discusses intentional underutilization of Blue Horizon processing nodes to mitigate the impact of preemption by system daemons on the scalability of MPI collective communication. Section 8 discusses the design and performance of a 2D FFT implementation which uses many of the techniques discussed in previous sections. Finally, observations, future research possibilities, and the conclusions are presented.

2. Project Hardware / Software / Operating Systems

The system performance was specifically evaluated for the Blue Horizon parallel cluster.

2.1 Cluster organization
The cluster is composed of 128 nodes. Each node has 8 processors.

2.2 Memory hierarchy
Each processor has two levels of data cache:

L1 Cache: 64 KB with 128 byte lines, 128-way associative
L2 Cache: 8 MB unified with 128-byte lines, 1-way associative

Each processor has a TLB (translation lookaside buffer) with 256 4KB entries, 2-way associative

2.3 Operating System
Blue Horizon runs AIX 4.3.3

3. Copy Performance

Copying is a low-level operation that is heavily used in message-based systems (for example, for moving data from a sender's buffer to the recipient's buffer). As described in a separate section, a shared-memory-based messaging system was developed to evaluate whether the intra-node performance of IBM's MPI implementation was optimal. Early testing with this shared-memory scheme indicated much lower messaging bandwidth than expected. Further investigation revealed that bandwidth was principally limited by the performance of the `memcpy()` library call, which was attaining long-buffer copy bandwidth of only 184 MB/sec (for repetitive copies between a pair of 40 MB buffers).

As a first effort to analyze the source this shortcoming, `memcpy()` was first replaced by the following naïve copy function:

```
int idx, *src, *dest

for (idx=0;idx<length;idx++)
       dest[idx] = src[idx];
```

This hand-coded copy routine performed identically to `memcpy()`, which leads us to conclude that the library `memcpy()` on Blue Horizon is similar or identical to our naïve copy function, and is not highly (or even at all) tuned.

Based on information in [IBM98], the naïve copy described above was replaced with the following tuned version:

```
volatile int ZERO;
int idx, zero = ZERO, *src, *dest;

for (idx=0;idx<length;idx++)
       dest[idx] = src[idx] + zero*dest[idx];
```

The L1 cache in Horizon's POWER3 uses an allocate-on-read-miss policy to decide which lines should be moved into cache. Because the naïve code never reads from `dest[]`, the output array elements are never cached and excessive memory write traffic is generated. Adding the multiply by zero and add induces read misses in the destination array, and thereby causes lines to be allocated for the destination buffer in cache. Though the modified copy routine does add a multiply per iteration of the innermost loop, this does not degrade performance because the cost of the multiply is very small on a modern superscalar CPU such as POWER3, relative to the memory latencies which limit the performance of the copy operation. The use of the volatile integer variable `ZERO` is necessary to prevent the compiler from optimizing away the multiply-by-zero (though this requirement is not mentioned in IBM's tuning guides), while the nonvolatile variable `zero` is needed for the inner loop because volatile variables cannot be stored in registers, and use of such a variable in an inner loop would therefore reduce performance by causing needless read accesses to memory. The result of this change was that long-copy bandwidth was improved by 54%, to 283 MB/sec.

The POWER3 CPU used in Horizon supports 4 hardware prefetch streams, and it was hypothesized that large-buffer copy performance could be further improved by modifying the optimized copy routine to exploit all 4 streams, instead of only 2 streams in the existing optimized copy (one each for the destination and source buffers). The resulting copy kernel is:

```
volatile int ZERO;
int idx, i, zero = ZERO, *src, *dest;

for (idx=0;idx<length/2;idx++)
{
```

```
        i = idx;
        dest[i] = src[i] + zero*dest[i]; i += length/2;
        dest[i] = src[i] + zero*dest[i];
    }
    // Epilogue code to handle odd lengths omitted for brevity
```

This routine effectively makes two copies in parallel, from regions spaced by half of the buffer size (i.e. 20 MB given the 40MB buffer for which results are reported here) to prevent unintended sharing. This implementation delivered copy performance of 203 MB/sec, which is slightly higher than the naïve copy kernel but much slower than the 2-stream optimized kernel. It is hypothesized that this performance shortfall is caused by the fact that copy now accesses 4 disjoint regions of memory instead of 2 on each iteration; Typical SDRAM memories can hold two pages open at a time, so the 4-stream implementation's usage pattern may prevent reuse of opened pages, and thereby increase mean latency (and reduce bandwidth) to memory.

There are three principal conclusions that may be drawn from these results:

1. Implementers of applications which are sensitive to copy bandwidth should consider using hand-coding, optimized copy routines for the Blue Horizon platform
2. The memory bandwidth which may be realized from a single CPU in a Blue Horizon node is (a somewhat limited compared to the nominal total node bandwidth of 1.6 GB/sec; More than one CPU per node must be used to exploit the full bandwidth of the memory subsystem.
3. The benefits of the POWER3 CPU's multiple prefetch streams are not always easily realized.

Using the tuned copy routine described in this section, performance slightly better than that attained by MPI was observed for intra-node message passing.

4. MPI Pack / Unpack Performance

In most parallel computations, slices of the data are passed between different nodes. In many cases, the appropriate data to send from one node to another consists of a series of non-contiguous regions in memory. In these cases, it is desirable to pack the data into a contiguous buffer before sending and the choice remains with the programmer to use either an MPI type that will cause the MPI to pack the data, or else to manually pack the data and hand a contiguous block to MPI.

The tradeoffs of manually packing the data versus allowing MPI to pack the data were evaluated by developing test cases and studying the empirical test results. The test suite was based on the Ring test normally used to evaluated message passing performance and bandwidth and identifies the ***baseline*** performance. The test tool was expanded to include MPI packing (on send, ***MPI Send Pack***) and MPI unpacking (on receive, ***MPI Receive Pack***) both individually and combined, ***MPI Send/Recv Pack***.

The basic kernel used to evaluate performance is as follows:

```
startTime = MPI_Wtime() ;
for( i = 0 ;   i < 100 ;   i++ )
{
   MPI_Irecv [ optional data type requiring packing (num elements, length,
   stride) ]
   MPI_Send [ optional data type requiring packing (num elements, length,
   stride)  ]
   MPI_wait()
}
endTime = MPI_Wtime() ;
```

The Direct Memory Assignment (***DMA Send Pack***, ***DMA Receive Pack***, ***DMA Send/Recv Pack***) packs(unpacks) the data into a temporary buffer by directly assigning memory (as opposed to relying on

whatever method the MPI internals may use). The routine first checks the size of the element to be packed and then selects the assignment style. To take maximum advantage of Blue Horizon's POWER3 CPU, the data was aligned to 8-byte boundaries by declaring the original data type as double. Typecasts are performed as required to implement small element sizes. Manual loop unrolling is implemented.

Three different copy kernels were used for copy. These were

1. Explicit assignment with cache prefectch
2. Memcpy()
3. Memcpy() with first-element prefetch

The Explicit assignment kernel is as described in section 3. Memcpy (***Memcpy Send Pack***, ***Memcpy Receive Pack***, ***Memcpy Send/Recv Pack***) packs the data using the memcpy() system calls for each element to be packed or unpacked. The Cache Prefetch Memcpy (***CPM Send Pack***, ***CPM Receive Pack***, ***CPM Send/Recv Pack***) packs the data using memcpy() as before, but forces a cache prefetch on the first output element. (The memcpy system routine is responsible for prefetching the subsequent cache lines).

These three methods were compared against the MPI pack/unpack using two different techniques

1. Constant data volume composed by varying the number of elements to pack with varying sizes
2. Preset number of elements to pack with varying sizes of data to be packed

The results for constant data volume (i.e. number of elements is related inversely to element size) are shown in figures 1.*. The 'A' series shows a comparison of packed sends, the 'B; series shows a comparison of receives that are unpacked; the 'C' series shows comparisons where the sent data is packed and the received data is unpacked. The ".1" suffix is the full data range in a logarithmic scale and the ".2" and ".3" suffixes shows a shorter data range at normal scaling. The results for a fixed number of elements are shown in figures 2.A through 2.C. In all cases, the 'baseline' data series represents the cost of performing the corresponding MPI operations without any packing on the same total volume of data.

Evaluation of the performance under these conditions shows that manually packing the data via direct memory assignment with cache prefetching into a temporary buffer and then making the call to MPI_Send usually performs better for element sizes smaller than 2K bytes. For larger elements, the MPI calls that invoke MPI's packing routines perform better. A noticable exception is that MPI packs better for element sizes of 4 bytes. There is also a correlation to the number of elements to be packed. The implication of this behavior is that the MPI performs its packing by directly writing each element onto the communications socket except for some special tuning for 4-byte elements. Working from this hypothesis, it is observed that at the 2K byte range for 1024 elements, the cost of `write()` system calls for directly writing elements to the communications layer are amortized and MPI packing begins to outperform the copy-based strategy. The result is particularly significant, because it suggests that for datatypes with large element sizes MPI packing may be intrinsically superior to any possible application alternative, because it avoids a copy.

5. Memory Contention Evaluation

One reported problem with overlapping communications and processing is contention for the memory ports. It has previously been reported by [Baden01] that this phenomenon can be observed in the non-linear speedup of tasks when a dedicated communications processor increases its quantity of communications. Increased communications results in more frequent accesses to memory, which then causes contention with the other processors that are also using the memory subsystem for their data access.

This problem was analyzed directly by creating a threaded application across a node and evaluating performance using an increasing number of processors on the node. The application does a matrix transpose and can block for cache. Running different numbers of processors gives clear evidence

about memory contention problems as can be seen in figures 3.A and 3.B. Figure 3.A shows memory performance when the threads are bound alternating processor banks using the formula

```
for( I = 0 ;   I < numProcessors ;   I++ )
        p = ( I / 2 ) + ( I % 2 ) * 4
```

Figure 3.B shows memory performance when the threads are bound incrementally to processors using the simple formula

```
for( I = 0 ;   I < numProcessors ;   I++ )
        p = I
```

Comparing the two sets of graphs shows that (at least for the given test case) allocating processors by bank has no effect on the performance. Evaluation of the data in the graphs demonstrates that adding more processors the simple problem that is highly memory intensive does reduce overall performance due to memory contention.

6. Static vs. Dynamic Workload Allocation with Threads

One choice that confronts the developer of a parallel application is whether to allocate work to individual processes/threads statically or dynamically. In a static scheme, each process/thread is given a single fixed-size block of work at inception, while in a dynamic scheme processes take smaller granules of work as they become idle. The dynamic scheme offer the promise of accommodating non-uniform CPU processing speeds and/or workloads with non-uniform per-element processing demands.

Dynamic allocation is particularly tempting for threaded applications, because the option of shared memory makes workload partitioning inexpensive. In order to evaluate the applicability of this technique, the performance of a threaded matrix transpose was evaluated with both static and dynamic workload allocation as a function of number of threads (up to the maximum of 8 on a Blue Horizon processing node). Performance was also evaluated as a function of processing granule size (i.e. the amount of work fetched from the work queue each time a processor is idle) to investigate the performance impacts of synchronization. The test was run both using batch submission, and directly on a lightly-loaded interactive node of Blue Horizon (tf173i.sdsc.edu). For all test runs, threads were bound to CPUs, and CPUs were assigned using the memory-port-interleaved ordering described in section 5.

The results are presented in figure 4. Static work allocation was consistently faster than dynamic allocation, though the amount of difference varied quite widely as a function of process count and whether the test machine was a batch node or the interactive node. In particular, dynamic allocation was consistently slower by a wide margin at intermediate process counts (more than 1, less than 8). On the interactive node, dynamic allocation was approximately 8% slower at 8 processes, while on the batch node it was approximately 35% slower. These differences between batch and interactive node performance were extremely consistent from run-to-run. All known thread-related environment variables were set identically on both systems. It is therefore hypothesized that the difference is caused either by some unknown thread-related environment variable, or by different tuning of the OS scheduler (which interacts heavily with thread synchronization primitives) on batch nodes.

To attempt to understand this performance shortfall as a function of dynamic versus static scheduling, the experiment was re-run with the loops of transpose kernel #define'd out, but all allocation and synchronization logic left intact (i.e. the number of mutex transactions is unchanged). The resulting runtimes were on the order of 0.005 seconds or less with all evaluated thread counts (i.e. much smaller than the observed performance shortfall). From this it is hypothesized that fixed mutex overhead is not the cause of the observed performance shortfalls. This hypothesis is further supported by a sensitivity analysis of performance versus the dynamically allocated granule size, which revealed performance changes of less than 1% while changing the granule size by 2X in either direction from the tested baseline.

One other hypothesized cause of the observed performance shortfall is false sharing of cache lines. This is unlikely, because the size of the work granules allocated to each thread is a multiple of the cache line size on both axes, and so no sharing should be possible in either the input or output buffers.

7. Intentional Node Underutilization

It has been reported that Blue Horizon's MPI collective communication primitives do not scale well to large process counts (on the order of 128 or greater). One hypothesis that has been advanced to explain this lack of scalability is that of preemption by system processes: On any given node, there exist certain system processes that run periodically at high priority. As the number of participants in an all-to-all exchange is increased, the likelihood that one or more such participants will be interrupted by a system processes increases, and so the performance of the collective communication routines does not scale.

In order to explore this hypothesis, a parallel 2D FFT code that relies on MPI collective communications for the transpose was run with both 8 MPI processes per node and 7 MPI processes per node. The latter count was chosen to leave 1 CPU per node idle at all times, in the hopes that the system scheduler would run periodic system processes on that CPU rather than interrupting communicating MPI processes on the others.

The results from this experiment are presented in figure 5. Runtimes are on the order of 10% faster for the 8 CPU/node case at small node counts, where calculation is significant relative to communication. At larger node counts, the data are difficult to interpret; The 8 process/node configuration is significantly faster with 8 and 32 nodes (56/64 and 224/256 MPI processes at 7/8 processes/node, respectively), while the 7 process/node configuration is significantly faster with 16 nodes (112/128 MPI processes). Interestingly, inspection of the per-process runtimes indicate that in the 32-node, 7 CPU/node case between half and 99% of the per-process runtimes were less than 0.45 seconds (depending on run), which suggests that for this grid size, the 7 process/node version very nearly bested the performance of the 8 process/node version. Given the ambiguous nature of this data, no conclusion regarding the hypothesis of system task preemption may be drawn, although the fact that 7 MPI processes/node was faster with 16 nodes and would have been faster at 32 nodes had process runtimes been more uniform suggests that this remains a promising direction.

As a direction for future investigation, the root cause of the wide variability in the results of this experiment should be better understood. It appears likely that there exists some confounding factor that is not presently understood. One possible candidate is the internal organization of the all-to-all primitives in IBM's MPI library. In particular, the authors of the MPI library may have made assumptions regarding the distribution of processes across nodes, which might in turn have influenced their choice of communication topologies. If so, this would tend to penalize the 7 process/node case, and might explain the unusual amount of non-uniformity observed in the resulting runtimes.

8. Parallel FFT With Overlapped Communication

One opportunity for improved performance in parallel computations is to create overlap between communications and computation. In a traditional MPI program, one process handles both chores. In such an implementation, the computation is therefore serialized with communication. Unfortunately, communication often does not make full use of the CPU, and so cycles are wasted. The solution to this problem is to overlap communications with computation.

In order to enable such overlap, two issues must be addressed: First, a mechanism for non-blocking I/O must be created; This refers to the low-level approach (i.e. non-blocking system calls, thread-level parallelism, etc) used to implement the overlap of communication and computation. Second, and perhaps more importantly (and labor-intensively), the computation itself must be structured such that useful parallelism exists. As an example of how the stucture of a computation may act as an impediment to overlapped I/O and computation, consider a naïve 2D parallel FFT implementation:

1. All rows are FFT'd.
2. An all-to-all exchange is used to move data between processes
3. All data received in (2) is locally transposed
4. All rows are FFT'd again

In such an approach, there exist no computations that may be overlapped with communication, because the communication phase depends upon the completion of the first FFT phase, while all subsequent computations depend upon the completion of the all-to-all exchange. The implementation must therefore be restructured if overlap is to be achieved

8.1 Overlap Mechanisms

One common approach to achieve overlap is to use non-blocking I/O within a single thread of control, and then execute computations while the non-blocking I/O call is serviced in the background. While this strategy is often successful, in the specific case of parallel computations using MPI the value of this approach is limited by two factors:

1. Many MPI primitives, including most collective calls, exist only in blocking variants.
2. Where nominally non-blocking MPI calls do exist, the extent to which the MPI implementation is able to actually overlap the resulting I/O work with computation is often limited. For example, on the IBM Blue Horizon there is strong evidence from other investigators (cite Baden here) that use of non-blocking MPI send and receive calls is not sufficient to achieve overlap.

Based on these two factors, this approach was deemed undesirable for this investigation.

A second approach is to employ threads. Threads are "lightweight processes", each of which possesses a unique process context (program counter, registers, etc) and stack, but shares code and non-stack data with all other, peer threads within the same process. Overlap can be achieved with threads by dedicating one thread per process to communication (via MPI, for example), and the remainder to calculation. This approach is particularly well matched to parallel computer systems that consist of a cluster of SMP nodes (Blue Horizon is one such system); One process is started per node with N threads (where N is typically the number of CPUs in each node) with one thread dedicated to handling MPI traffic for inter-node communication, and the remainder performing calculation. Explicit intra-node communications are replaced by shared memory accesses among the threads. This was the approach chosen for this investigation.

8.2 Algorithm Modifications for Overlap

As noted above, the most straightforward implementation of the 2D parallel FFT makes overlap impossible, due to dependency of the all-to-all exchange upon the results of the immediately preceding computation step, and dependency of the following computation step upon the results of the all-to-all exchange. A transformation of the algorithm is therefore required.

In many ways, this dependency problem resembles the problem of scheduling instructions in a modern CPU, and indeed similar approaches can be employed across both problem domains. The most direct algorithmic approach to generating overlap is very much analogous to software pipelining or loop unrolling: Each of the computation and communication operations are first separated into two or more phases, such that each such phase is dependent only upon preceding phases of its own operation, and the corresponding and preceding phases of the previous step (so, for example, the second phase of the transpose operation may depend on the result of the first phase of the transpose operation and the first or second phases of the all-to-all). This transformation is depicted in the "Serial Implementation" and "Granularized Steps" sections of Figure 6, in which arrows represent dependency relationships. Given such a division of the processing steps, it becomes possible to overlap communication and computation, as shown in the "Overlapped" section of Figure 6. Though the example used is specific to the 2D FFT problem with 1D decompositions, it can also be employed in many other problem domains: For example, the strategy described by [Baden01] of computing interior data values first in the Red-Black PDE solving problem so that overlapped values from other processes will not be used until late in the computation is an instance of this technique.

The principal advantage of this technique is that it is relatively simple to implement, particularly in problems such as the FFT where there are few true dependencies within a given operation (for example, all rows are independent in the FFT, and all pixels are independent in the transpose). Much like loop unrolling and software pipelining, its limitation is that it is a static approach: Just as static instruction scheduling requires assumptions about instruction and memory latencies, static overlap scheduling requires that assumptions be made at the time the code is written regarding the relative speeds of the processors on which it will be run; When these assumptions are violated, performance is degraded. It has been established by other investigators that the Blue Horizon system exhibits non-uniform CPU performance, and so any problem that is to be run on this system must be hand-tuned to use non-uniform partitions if maximum performance is desired. This, in turn, implies that any such code will be rendered non-optimal on machines that do have uniform CPU speeds.

The limitations of static scheduling described above may be overcome using dataflow-driven scheduling, in a manner analogous to Tomasulo's algorithm for CPU instruction scheduling. Because it is ultimately desirable to create a "generic" implementation of a problem which can be run on multiple parallel machines without having to hand-tune the data partitioning for each, such an approach was chosen in this investigation. Figure 7 depicts the data flow and producer-consumer relationships employed. Idle worker threads fetch groups of rows from the input buffer, using a shared counter protected by a mutex to govern allocation and guarantee that no two workers attempt to process the same rows. As worker threads release FFT'ed rows to the FFT output buffer, they update status variables (one per thread) to notify the communicator thread of their progress. The communicator thread waits for contiguous groups of processed rows of sufficient size of become available, and engages in all-to-all communications with the other communicators to execute the inter-node portion of the transpose for each such group. As the communicator thread does so, it updates a status variable that indicates how many rows have been received from each of the other communicator threads. The worker threads maintain a mutex-protected array of counters, one per MPI process (*not* per worker thread!), which are used to keep track of how many of the rows received from each process have been claimed by worker threads for transposition. A value in one of these counters that is less than the number of rows received represents a group of rows that are available to be transposed. This approach allows the FFT, all-to-all communication, and transpose to all occur simultaneously, subject only to dependency constraints in the calculations. Because worker threads grab groups of rows for FFT and transpose only when they are idle, non-uniformity in CPU performance is accommodated.

8.3 Miscellaneous Implementation Notes

Several other factors of this implementation are worthy of note. First, based on the outcome of the copy bandwidth experiments described elsewhere in this report, the packing operation of both the pure-MPI and mixed-mode implementations has been coded to use the fast copy routines. The transpose is cache-blocked but does not use the fast-copy prefetch technique, because a separate experiment indicated that doing so would reduce transpose performance by approximately 20%. The investigated implementation packs data for send in custom code rather than using MPI datatypes, for two reasons:

1. Doing so allows maximum flexibility to pack either on the worker threads, or on the communicator thread depending on how we wish to adjust workload balance between the two; Use of MPI packing would require that packing be performed solely on the communicator thread.
2. The results of the packing performance investigation described elsewhere in this report indicate that hand-coded packing routines perform nearly equivalently to MPI packing for the range of element lengths (i.e. the length of the portion of a row which goes to another processor) that are used in the transpose.

By default, the implementation considered in this investigation locks threads to processors, though this behavior can be adjusted at compile-time via a C #define.

8.4 Performance

Performance of the threaded, dataflow-driven 2D FFT implementation was disappointing. Runtimes for various computation grid sizes as a function of node count are shown along with 8 process/node MPI runtimes in figure 7. Though scalability is clearly better than the pure-MPI implementation, absolute runtimes are consistently worse at less than 512 CPUs, at which point the threaded implementation finally wins.

One hypothesis that might this performance shortfall is that synchronization associated with the various work queues is harming performance. In order to test this hypothesis, the granularity in which work is allocated was adjusted; because worker threads access the work queues and their associated mutexes once per granule, one would expect to see a decrease in runtime as granularity is increased if mutex contention were a limiting factor. The result of this experiment was that runtime was constant to within +/- 5% as work granularities were adjusted a factor of 2 in each direction from the default settings. This suggests that contention for work-queue synchronization objects is not the root cause of the observed performance shortfall. Using completely static allocations could further test this hypothesis, but that was beyond the scope of this investigation.

Another hypothesis to explain the shortfall is that coalescing what was formerly 8 processes' communication workload into a single thread left that thread overburdened relative to its worker counterparts, and thereby harmed total runtime. In order to test this hypothesis, the FFT implementation was recompiled with the actual communication calls removed. The resulting runtime was identical to that with communication enabled, and it is therefore concluded that the hypothesis is false, and the worker threads rather than the communicator are limiting throughput.

A third hypothesis is that false sharing of cache lines is leading to unnecessary invalidations, and thereby degrading performance. This hypothesis is disproven by noting that the sizes of the data parcels allocated to individual threads are always multiples of the cache line size, and that no two threads ever share the same data except for read-only state associated with the FFT (the pre-computed twiddle-factors) and masking operations (a 1D LUT describing intensity as a function of radial position). Because the only shared state (false or otherwise) is read-only, cache line invalidation due to the coherency protocol cannot be a factor.

As noted in a previous section, performance shortfalls in threaded applications with dynamic workload allocation have been observed on certain Blue Horizon processing nodes. It is suspected that the throughput shortfall observed for the threaded 2D FFT is a consequence of the same, as yet unexplained phenomenon.

8.5 Opportunities for Improvement

Three principal directions of future research have been identified, by which the threaded 2D FFT implementation could be enhanced or better understood:

1. Recode it to use static data allocation among worker threads, to determine whether the cause of the observed performance shortfall is indeed related to dynamic workload allocation (this is not believed to be a desirable final solution, for reasons previously outlined).
2. Exploit idle cycles on the communicator thread. The test with communications disabled suggested that the communicator thread may have been underutilized. If this is the case, then the communicator could be recoded to grab and transpose bundles of work when idle. Likewise, if non-blocking point-to-point routines were used instead of MPI collective calls, and if the MPI implementation could truly overlap communication and computation, then the communicator could process bundles of work when background communications were underway. Such scheme was coded for an alternate, point-to-point based FFT implementation during this investigation, but the scope of the project did not allow time to fully evaluate it.
3. Investigate scalability to larger process counts. Prohibitive queue times for jobs with high node counts (>= 64 nodes) prevented such an experiment from being conducted within the timeframe of this investigation.

4. Investigate hand-coded replacements for the pthreads synchronization primitives; This was not done for this experiment because experiments with work allocation granularity suggested that synchronization overhead was not a non-issue, but it is a possibly worthwhile direction for future research nonetheless.

9. Conclusion and Future Directions

While the results of this investigation were quite unambiguous and positive regarding the optimization of low-level copying and packing functions, the expected benefits of dataflow-driven threaded execution with dynamic workload allocation were not realized. The principal thrust of immediate follow-up work should therefore be to understand the performance shortfalls observed in both the dynamic-allocation versions of both the transpose and 2D FFT codes. Similarly, the results of the node-underutilization experiment were encouraging but not conclusive, and would therefore benefit from further investigation and more conclusive results.

As a longer-term project, one might investigate more generic and flexible approaches to dataflow-driven work allocation than were used in the 2D FFT implementation. These might include:

1. Generic, library-level support for interthread work allocation and non-blocking I/O.
2. More flexible partitioning of work between the “communicator” thread and the worker threads, as described in section 8.5 above.

In addition, it is clear that the Blue Horizon system in general and specifically the POWER3 processor have complex and often counterintuitive performance characteristics. Further investigation and understanding of hardware phenomena may lead to further improvements to low-level kernels for common operations, and may also aid in understanding some of the unexplained performance phenomena noted in this investigation.

10. References

[IBM98] Stefan Andersson et al, RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide, IBM Corporation, 1998

[Baden01] Results presented to CSE 260 lecture on or around 05/01, University of California at San Diego, 2001

11. Who did what

Patrick Chase conducted the thread-based messaging performance test (including copy optimization) described in section 3

Bob Boyer conducted an MPI-based inter- vs. intra-node messaging performance experiment which led us to conclude that our copy performance was low and needed work (because the thread-based version wasn't keeping up), and that IBM's MPI implementation was exploiting shared memory for intra-node messaging as advertised. These results were reported in a previous progress report, but are omitted here.

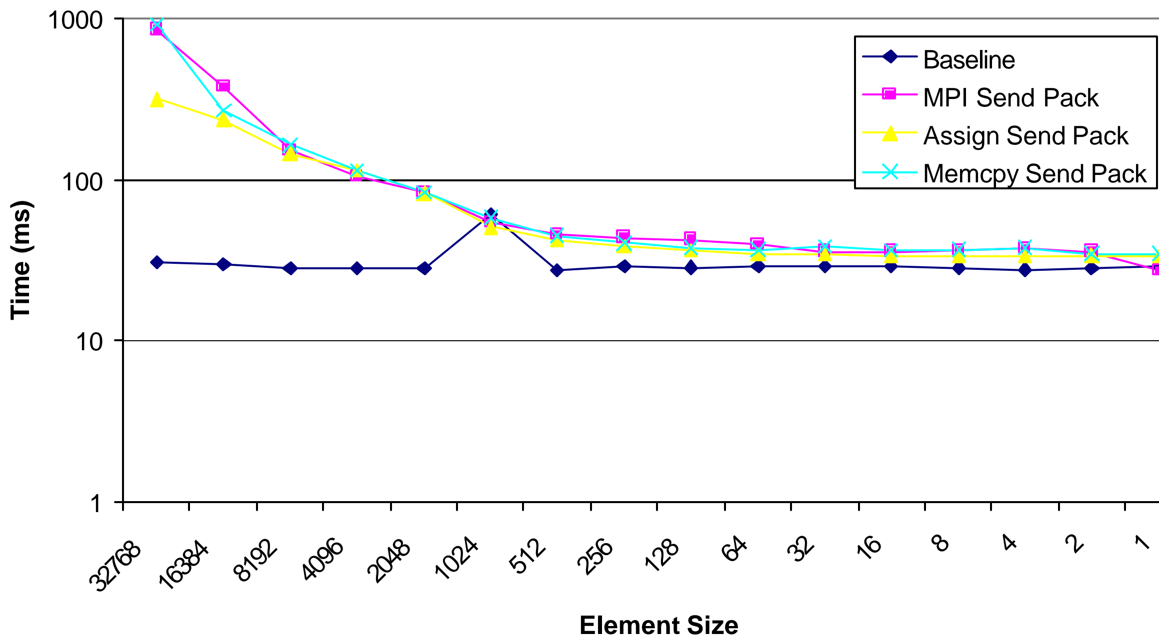
Bob Boyer conducted the packing experiment described in section 4

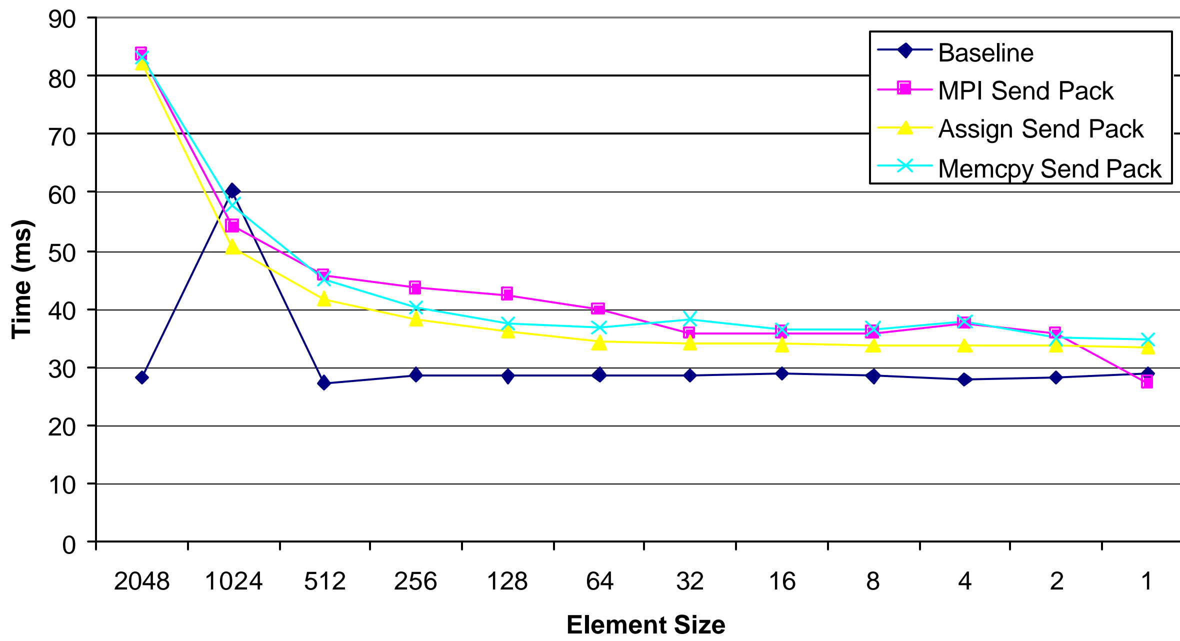
Bob Boyer conducted the memory contention experiment dscribed in section 5

Bob Boyer and Patrick Chase separately implemented and tested dynamic allocation experiments as described in section 6, because neither of us were willing to believe the results without confirmation.

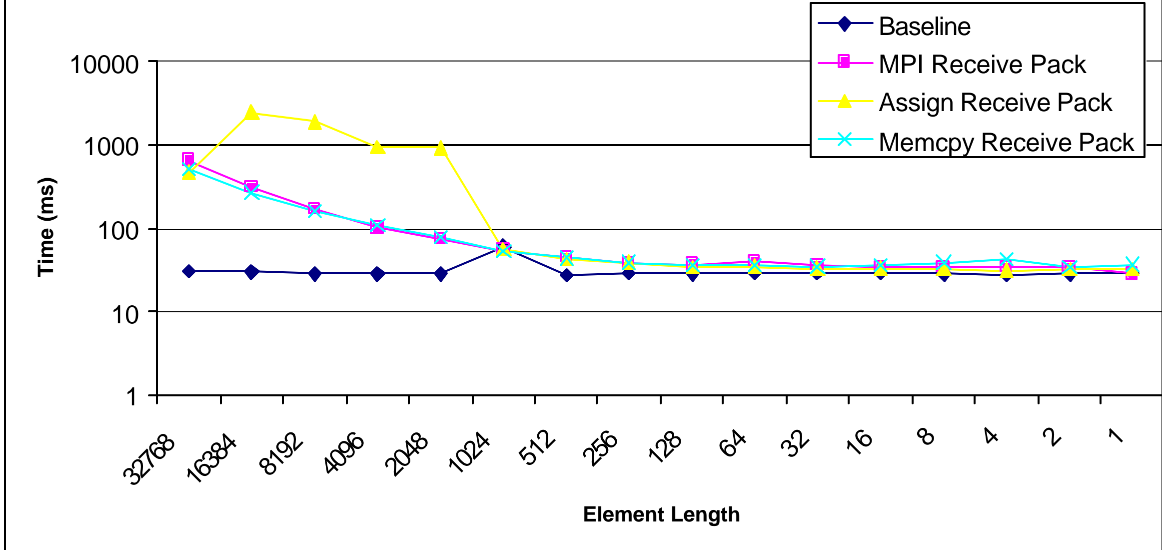
Patrick Chase conducted the 7 CPU/node vs 8 CPU/node experiment described in section 7

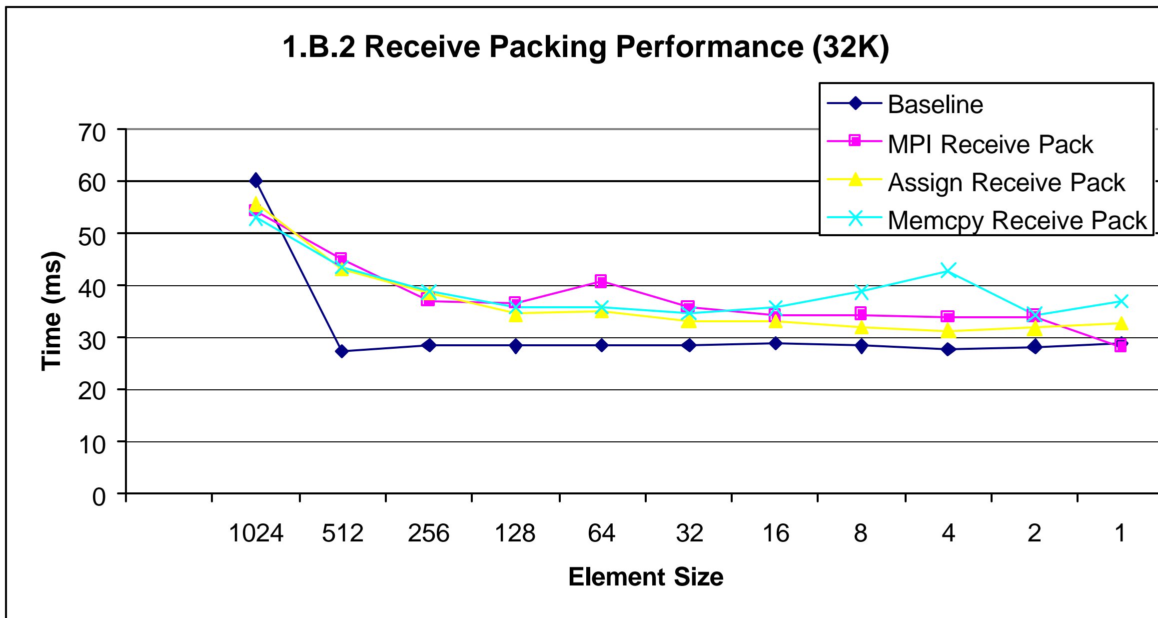
Patrick Chase developed the dataflow-driven FFT implementation and conducted the experiments described in section 8.

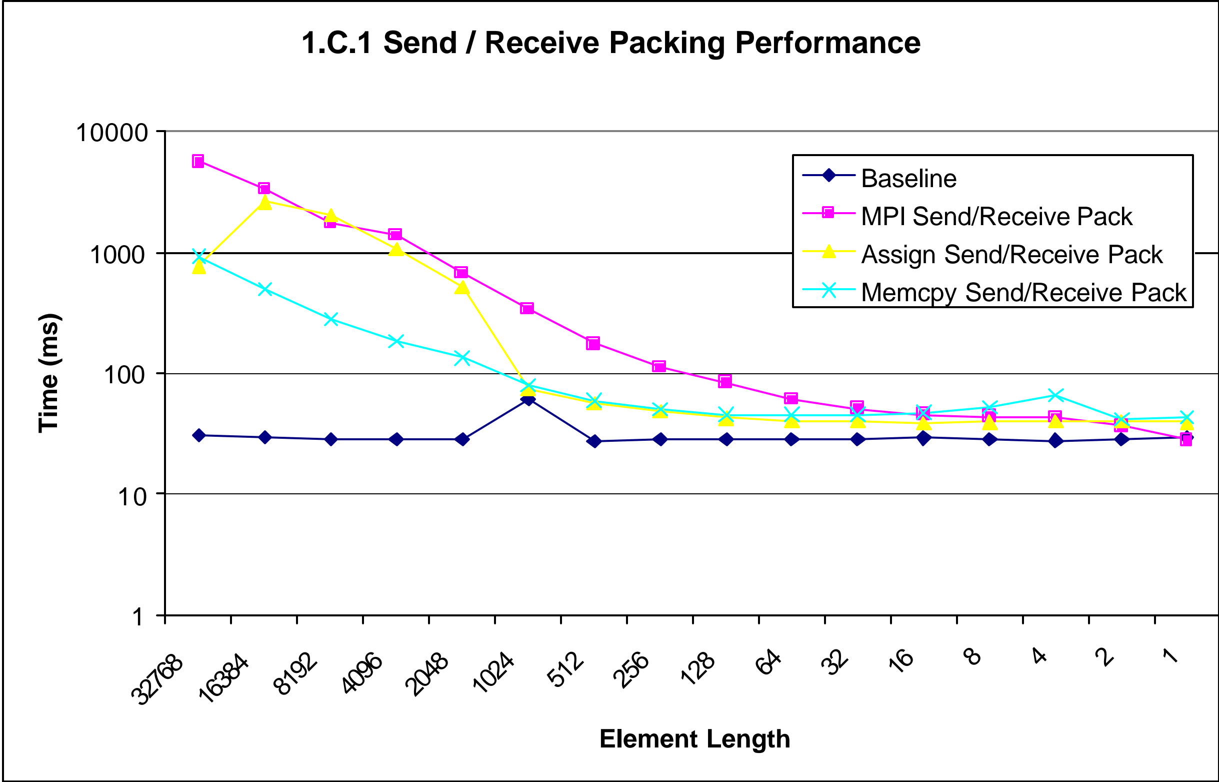

1.A.1 Send Packing Performance (Sent 32K)
Time (ms)
1000
100
10
1
Baseline
MPI Send Pack
Assign Send Pack
Memcpy Send Pack
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
Element Size

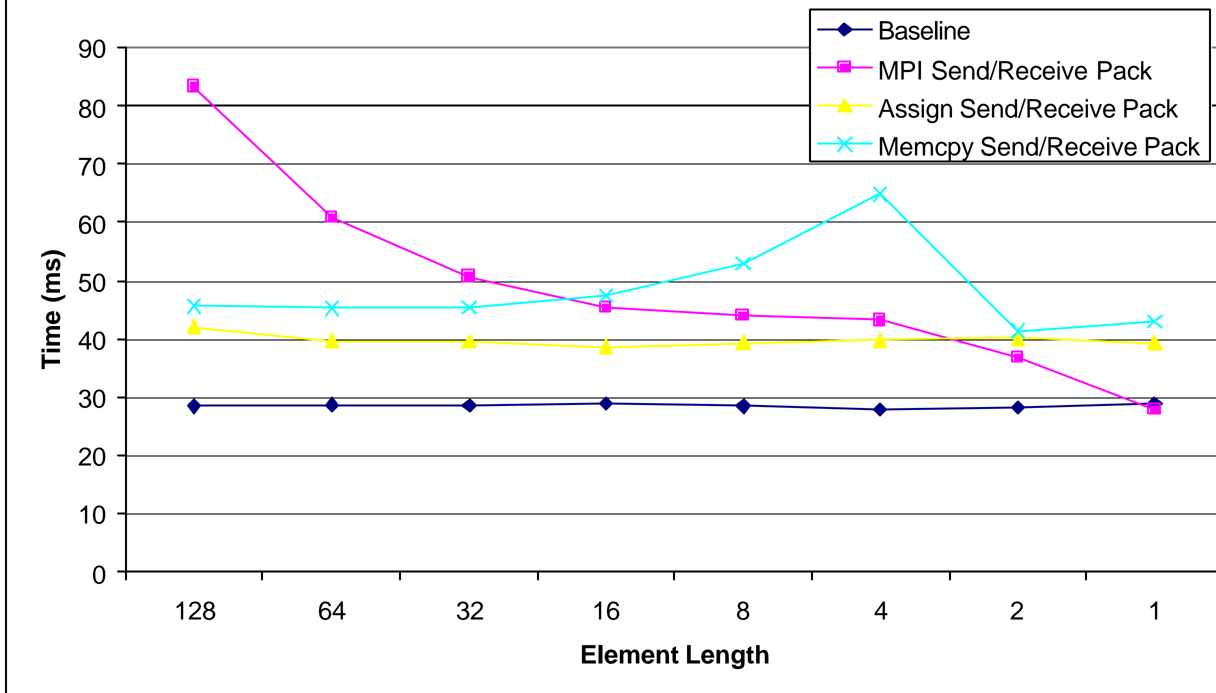


1.A.2 Send Packing Performance (Sent 32K)
Time (ms)
90
80
70
60
50
40
30
20
10
0
2048
1024
512
256
128
64
32
16
8
4
2
1
Element Size
Baseline
MPI Send Pack
Assign Send Pack
Memcpy Send Pack

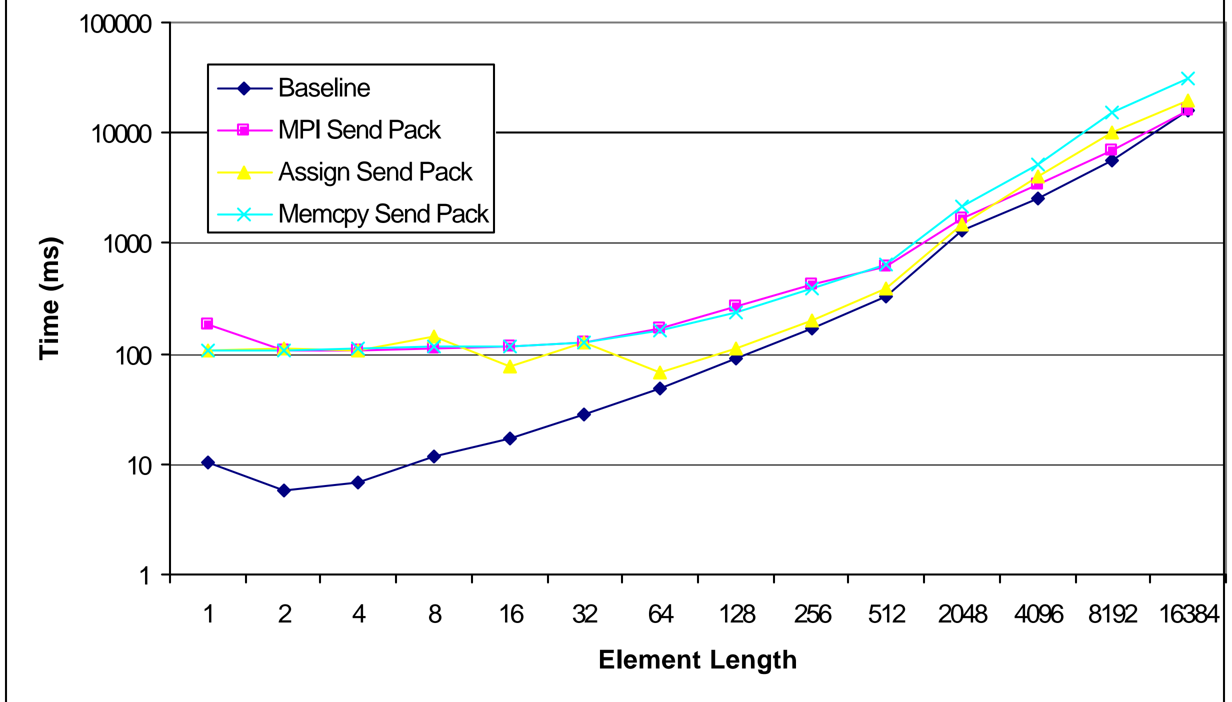

1.B.1 Receive Packing Performance (32K)
Baseline
MPI Receive Pack
Assign Receive Pack
Memcpy Receive Pack
10000
1000
100
10
1
Time (ms)
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
Element Length


1.B.2 Receive Packing Performance (32K)
Baseline
MPI Receive Pack
Assign Receive Pack
Memcpy Receive Pack
Time (ms)
70
60
50
40
30
20
10
0
1024
512
256
128
64
32
16
8
4
2
1
Element Size

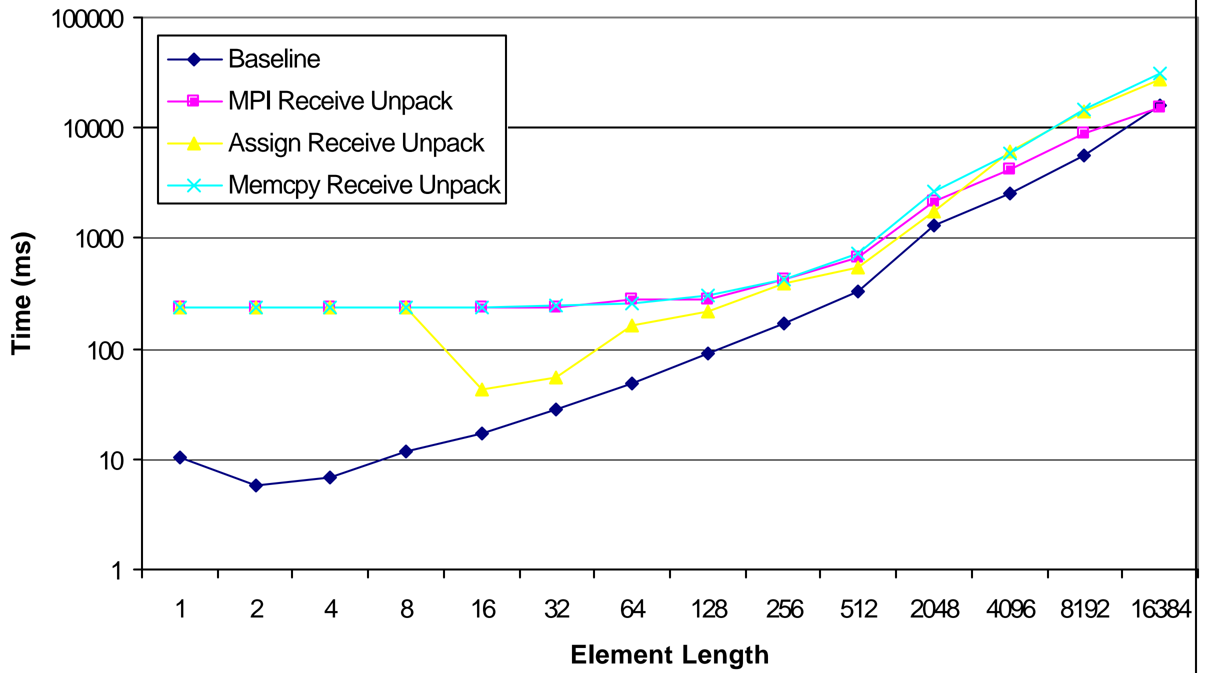

1.C.1 Send / Receive Packing Performance
Time (ms)
10000
1000
100
10
1
Baseline
MPI Send/Receive Pack
Assign Send/Receive Pack
Memcpy Send/Receive Pack
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
Element Length

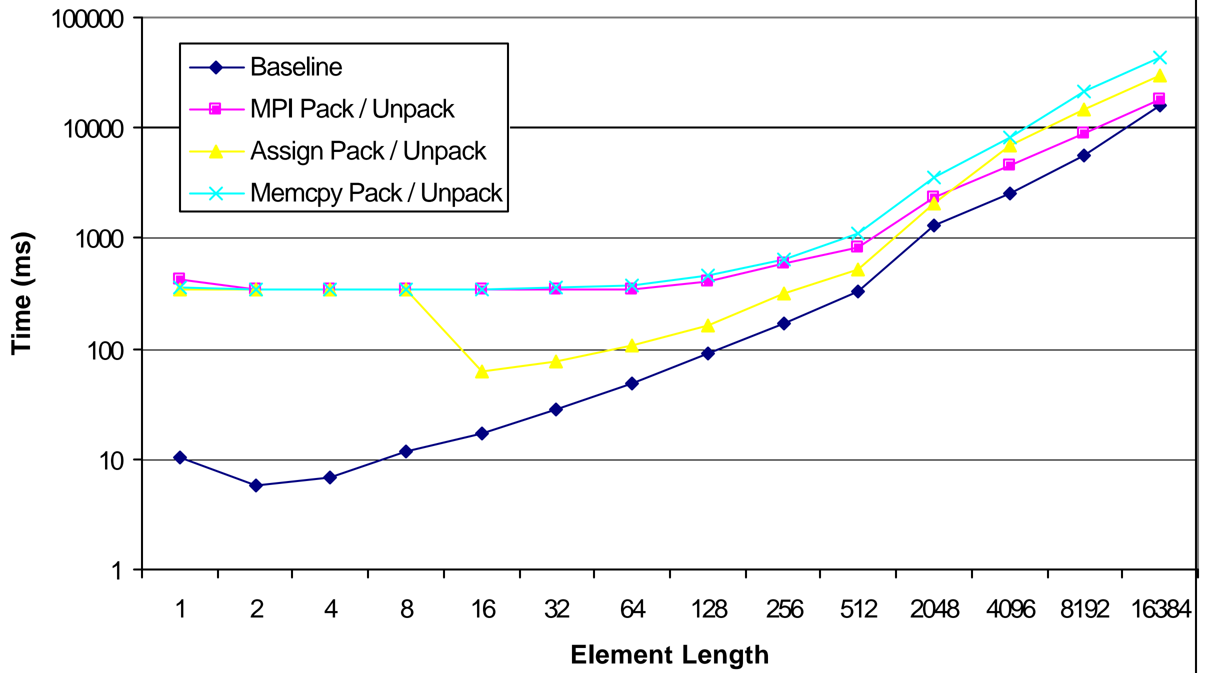


1.C.2 Send / Receive Packing Performance
Baseline
MPI Send/Receive Pack
Assign Send/Receive Pack
Memcpy Send/Receive Pack
Time (ms)
0
10
20
30
40
50
60
70
80
90
128
64
32
16
8
4
2
1
Element Length

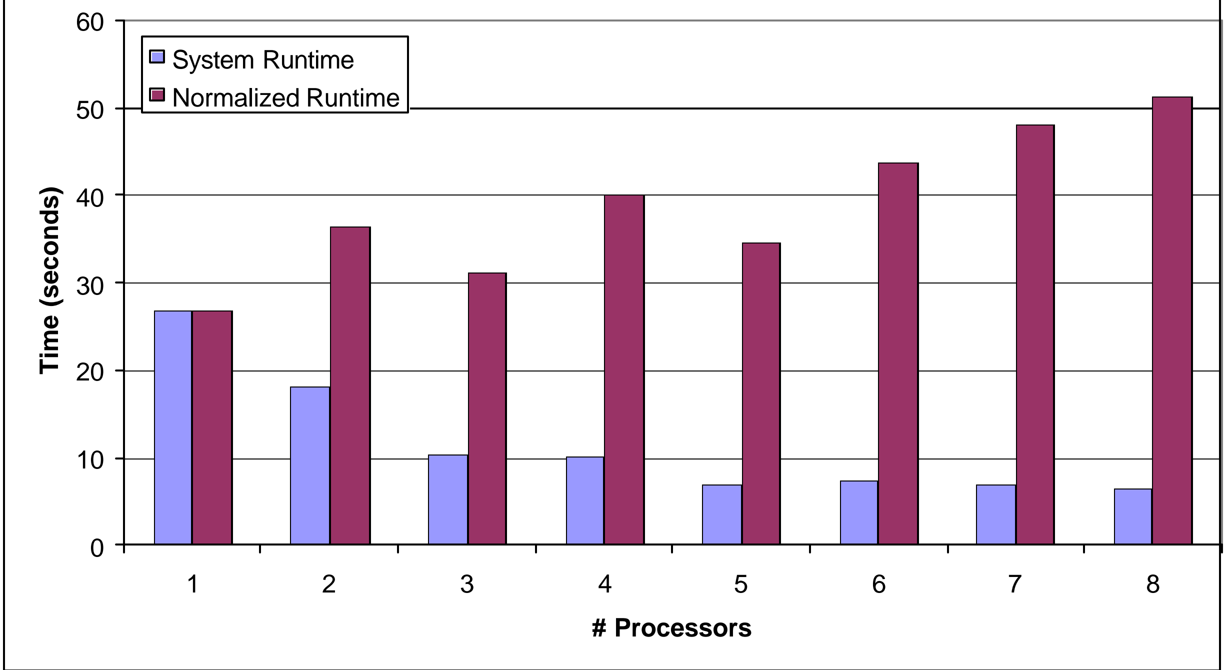


2.A Pack Performance 1024 Elements
100000
10000
1000
100
10
1
Time (ms)
Baseline
MPI Send Pack
Assign Send Pack
Memcpy Send Pack
1
2
4
8
16
32
64
128
256
512
2048
4096
8192
16384
Element Length


2.B Unpack Performance 1024 Elements
Baseline
MPI Receive Unpack
Assign Receive Unpack
Memcpy Receive Unpack
Time (ms)
100000
10000
1000
100
10
1
1
2
4
8
16
32
64
128
256
512
2048
4096
8192
16384
Element Length



2.C Pack / Unpack Performance 1024 Elements
Baseline
MPI Pack / Unpack
Assign Pack / Unpack
Memcpy Pack / Unpack
Time (ms)
100000
10000
1000
100
10
1
1
2
4
8
16
32
64
128
256
512
2048
4096
8192
16384
Element Length


3.A Memory Contention - Banked Processor Binding
System Runtime
Normalized Runtime
Time (seconds)
60
50
40
30
20
10
0
1
2
3
4
5
6
7
8
Processors

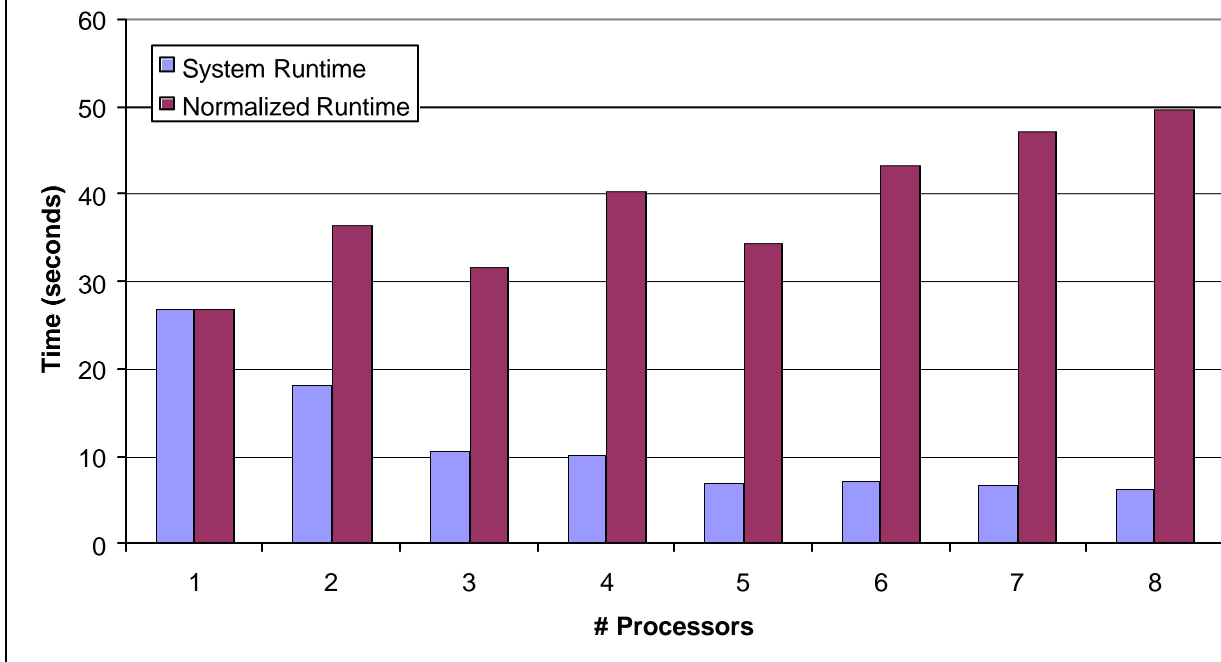
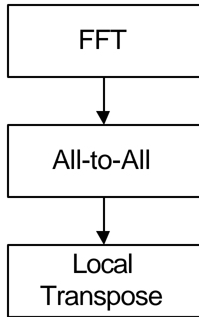
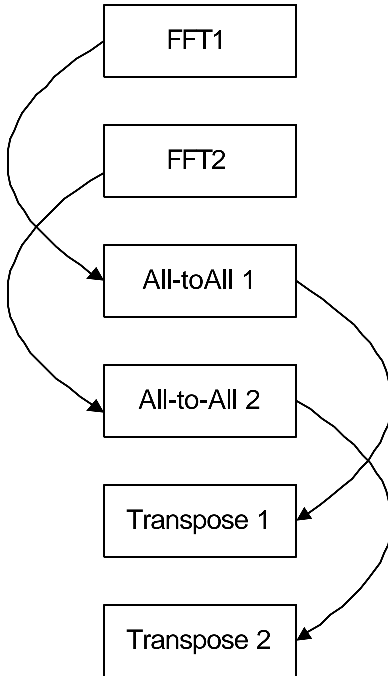

3.B Memory Contention - Unbanked Processor Binding
System Runtime
Normalized Runtime
Time (seconds)
0
10
20
30
40
50
60
1
2
3
4
5
6
7
8
Processors

Figure 6 - Static Work Allocation





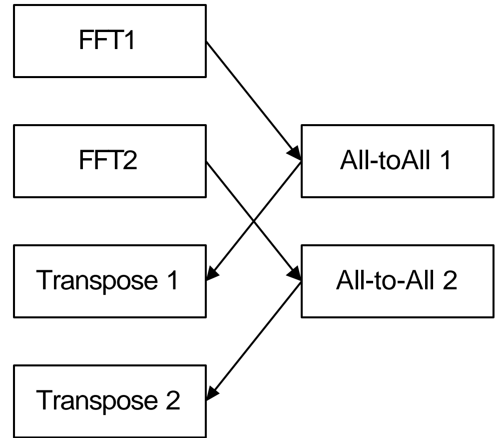
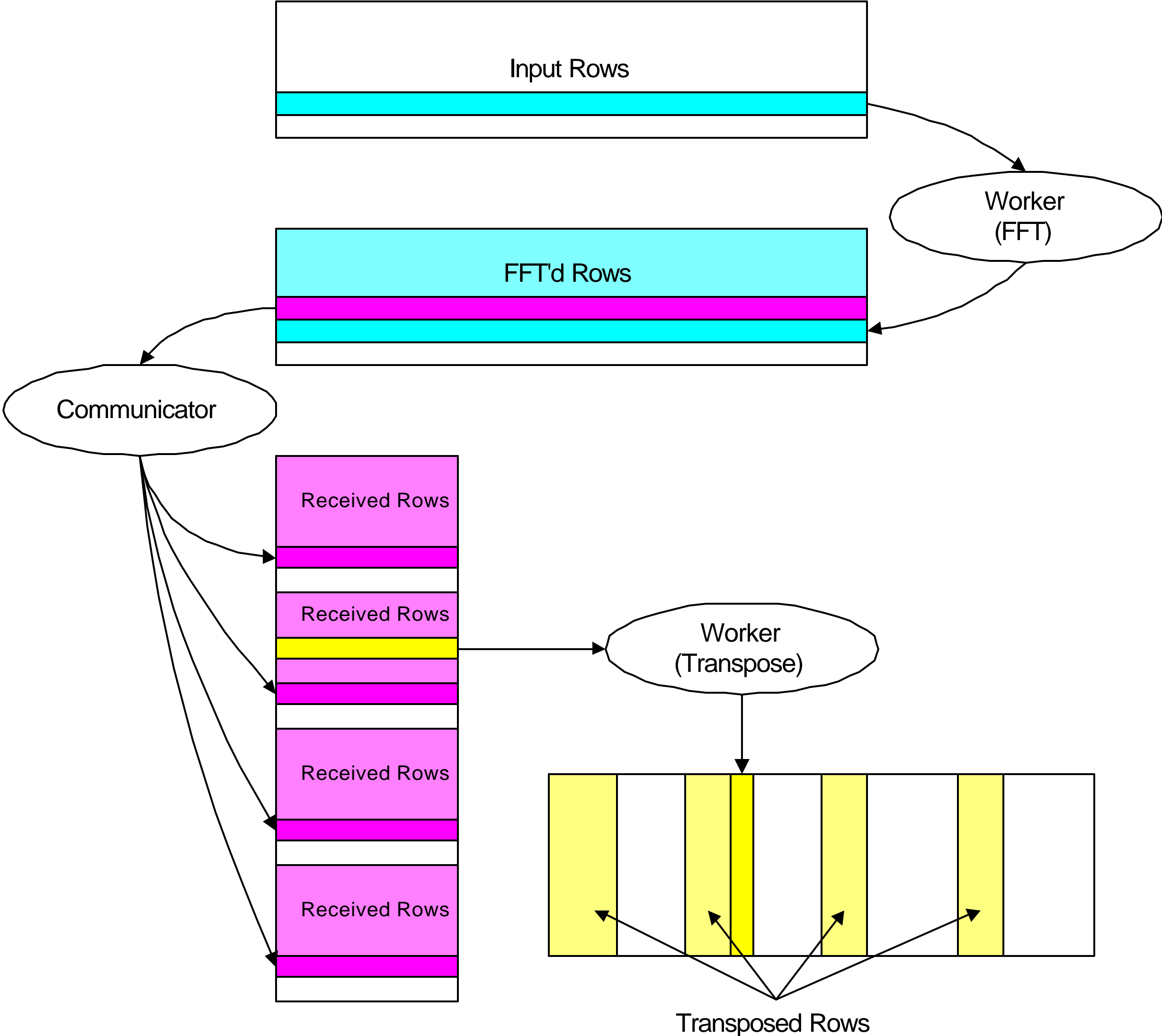


Figure 7 - Threaded FFT Dataflow



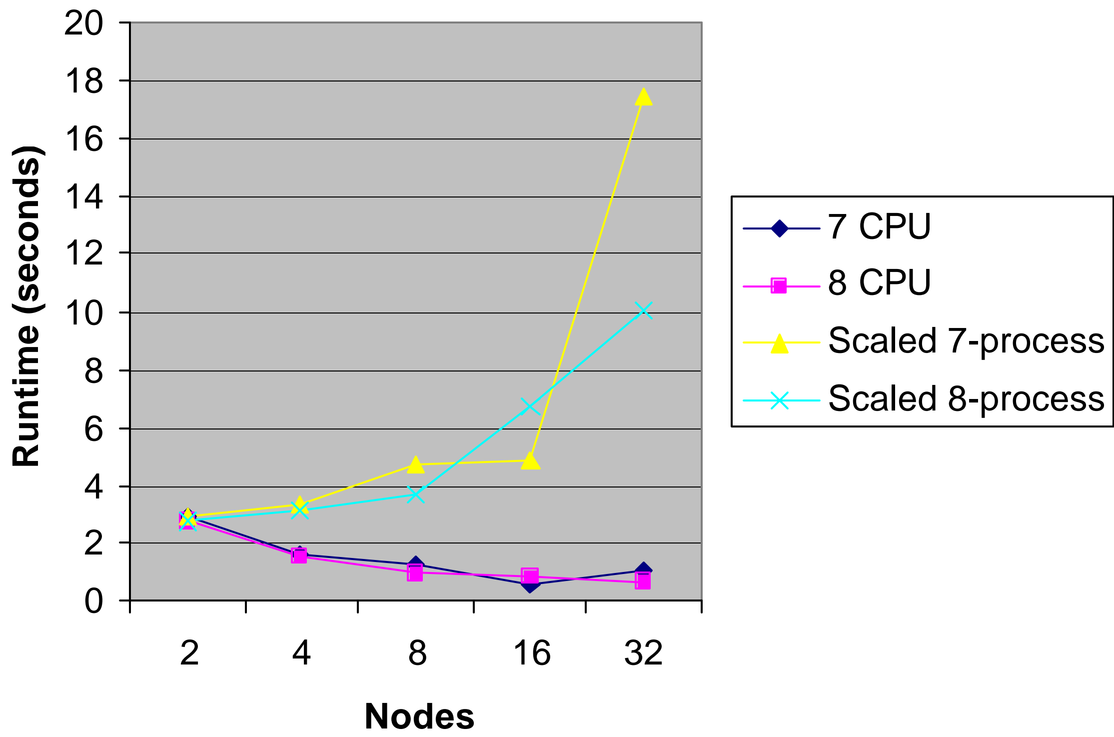


FIgure 4 - 2D FFT Performance
7 process/node vs 8 process/node
Runtime (seconds)
20
18
16
14
12
10
8
6
4
2
0
2
4
8
16
32
Nodes
7 CPU
8 CPU
Scaled 7-process
Scaled 8-process

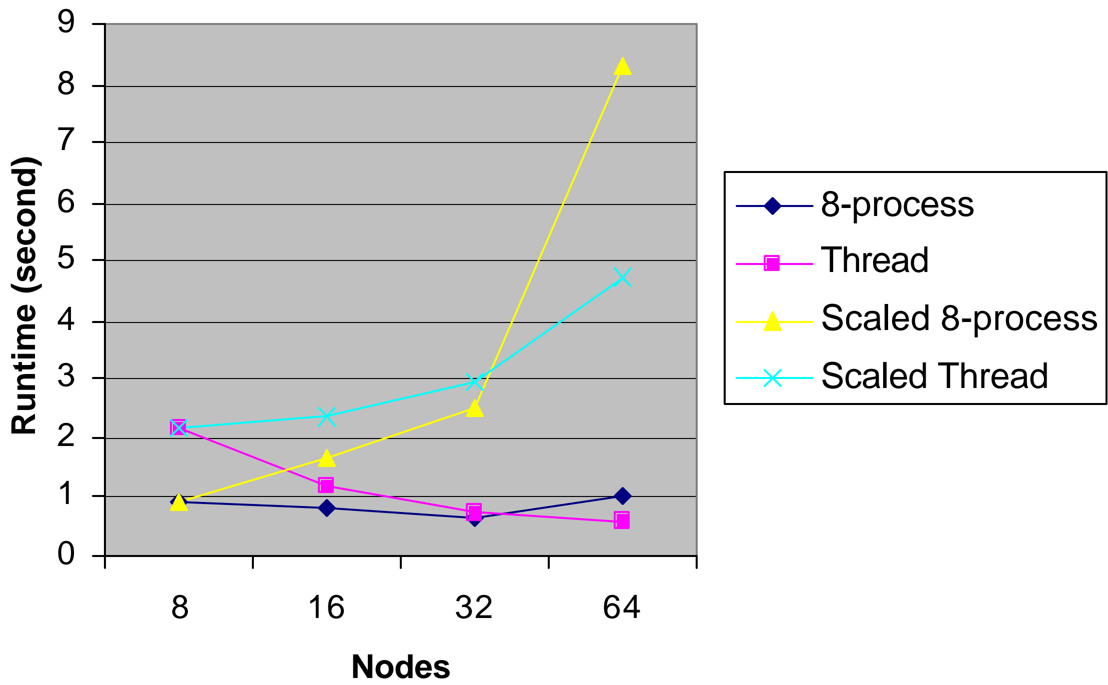


FIgure 8 - 2D FFT Performance
Threaded vs. 8 processes/node
Runtime (second)
9
8
7
6
5
4
3
2
1
0
8
16
32
64
Nodes
8-process
Thread
Scaled 8-process
Scaled Thread

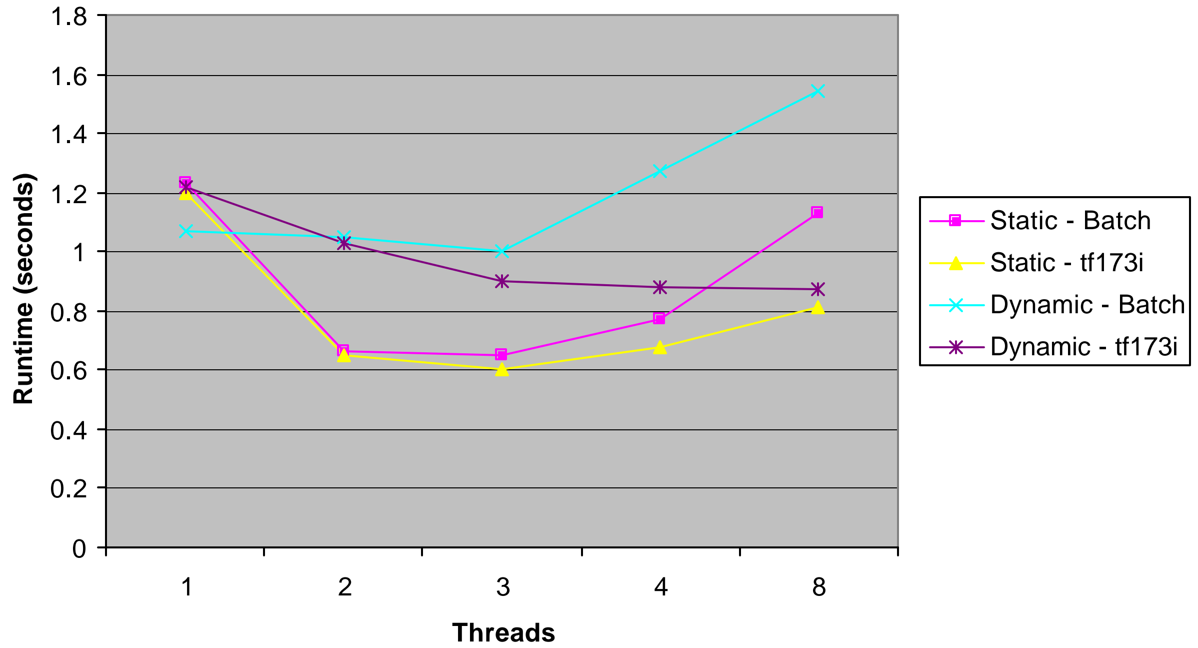


Figure 4 - Threaded Transpose Performance
Static vs Dynamic Workload Allocation
Runtime (seconds)
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
8
Threads
Static - Batch
Static - tf173i
Dynamic - Batch
Dynamic - tf173i