LARGE SCIENTIFIC CALCULATIONS ON DEDICATED CLUSTERS OF WORKSTATIONS

LARS PAUL HUSE AND KNUT OMANG
Department of Informatics,
University of Oslo, Norway
Email: {larspaul,knuto}@ifi.uio.no

ABSTRACT

The availability of high capacity, low latency interconnects enable clusters of workstations to form powerful multicomputers with good computing and memory capabilities. To achieve high performance with modern cache hierarchies, software must be written with the architecture in mind, e.g. reducing data movement for the data intensive application increase performance. Data movement becomes even more important with the high cost of inter-node communication for clusters of workstations, but overlapping computation and communication may reduce the cost. The paper presents optimized versions of two large scientific applications on one SMP and two dedicated clusters of Sparc based workstations with up to 32 processors. The nodes in the clusters are connected using high speed SCI (Scalable Coherent Interface) based interconnects. Results show that the SCI clusters can be favorable in price and competitive in performance to large multi-processors servers, in particular when the aggregated memory demand is high.

Keyword: Dedicated Clustering, SMP, SCI, MPI, CFD

1 INTRODUCTION

This paper examines the potential of dedicated clusters of workstations[2] for solving calculation intensive scientific problems. Using off-the-shelf hardware for large calculations introduce new challenges in terms of increased potential and benefit of optimization for the particular architecture. An SMP (Symmetric multiprocessor) is a set of equal processors with private caches, sharing main memory over a crossbar or a common bus. To achieve good performance on an SMP a broad range of *intra-node* optimizations can be applied, to make optimal use of the different levels of cache and reduce access to main memory. When multiple *nodes* (each possibly with multiple processors) is clustered, the interconnect becomes yet another architecture level subject to its own *inter-node* optimizations, where each nodes memory in some sense can be viewed as a high level cache for the local processors.

Large scientific calculations has traditionally been run on huge vector machines. Code written for vector machines focus on tight loops with possibility for pipelining and vector chaining to achieve good performance[6, p. 403-446]. Due to the low number of functional units and vector registers combined with the long latencies for vector operations, the common vectorizable inner loop contain only small pieces of total calculation. Consequently, all data are washed through several times during the calculation, and the usefulness of vector caches is very limited. To keep up with the processing units without caches, a high capacity and thereby very expensive memory system is needed.

Powerful microprocessor based systems (workstation clusters[1] [16], large SMPs and MPPs) have lately become competitive in performance to the vector computers for many applications. The trend for the last decades is an annually increase in microprocessor performance of 50-100 %, while big memories (DRAM) only increase by 10-20 %. To utilize the increased processor performance, the programmer may have to adapt his algorithms to match the memory and I/O capabilities of the machine. Fortunately some of this speed gap can be filled by proper use of caches.

When running most vector codes on cache machines with work set larger than the cache size, the low level algorithm decomposition gives a high inter-nest loop cache miss rate[12] (capacity misses between different loops doing calculations on the same data). Given the relative small caches of SMPs (typical 256-1024 KB per processor) this is the case for most large numerical problems. To improve locality (temporal and spatial) the algorithm have to be rescheduled/regrouped, or totally replaced with one better suited.

The NPB (NAS parallel benchmarks)[13] has been developed for the performance evaluation of supercomputers. The benchmarks come in two flavors: a free implementation of a paper and pencil specification (NPB1) and a source code version supplied by NAS (NPB2). Together they mimic the computation and data movement characteristics of large scale CFD (computational fluid dynamics) applications. In this paper two benchmarks from the NPB1 suite are implemented and analysed as practical examples of CFD calculations. These are the embarrassingly parallel (NPB-EP) and the 3-dimensional FFT (Fast Fourier Transform) partial differential equation kernel (NPB-FT).

MPI (Message Passing Interface)[4] has become a widely used standard. MPI is used in NPB2 (and our NPB1 implementation) to allow fair comparison of performance between machine vendors and other programs implementing NPB. NPB2.x are MPI-based source-code implementations written and distributed by NAS, and is required to be run as is. Thus either MPI must be used within each node or the parallelization must be transparent through the use of a parallelizing compiler or library. Performance of an MPI and threads based implementation of NAS-FT that can benefit from the uniform shared memory available inside SMPs is presented in section 5.8.

The paper will continue by introducing three different platforms used for the experiments (section 2). Furthermore, parallelizing techniques for SMPs (*intra-node optimizations*[12]) are detailed (section 3) and applied to the two benchmarks used, NPB-EP and NPB-FT[13] in section 4 and 5. Performance results for the two benchmarks are presented in section4.3 and 6 respectively. Section 6.4

concludes with a performance and price comparison of two generations of Sparc based clusters/SMPs.

2 EXPERIMENTAL SETUP

Two different cluster solutions are investigated and compared to performance of a large multiprocessor server. For multi-node experiments SMPs are interconnected using SCI (Scalable Coherent Interface)[18]. SCI is an ANSI/IEEE standard based on point-to-point communication links connected in rings. Characteristics of the I/O adapter based SCI interconnect used in the workstation clusters described in this paper is presented in [14] and [15].

Scali-HS consists of 2, 4 or 8 AxilServer S/420 with quad 125 MHz HyperSparc processors from Ross Technologies. HyperSparc have 8 KB (all instruction, no data) L1 cache on-chip. Each processor has a 256 or 512 KB direct mapped L2 cache, and each node is equipped with 1 GB memory. The nodes are interconnected with two Sbus-2 cards from Dolphin in ring/switch configurations (2 nodes + 1 switch port per ring). Scali-HS is a trademark of Scali AS.

Ultra-2 cluster: The cluster consists of 2 Ultra-2s with dual 200 MHz UltraSparc-I processors from Sun Microsystems. The UltraSparc-I has a 32 KB (16 KB data and 16 KB instruction) L1 cache on-chip. Each processor has 1 MB direct mapped L2 cache, and each node is equipped with 512-1024 MB memory. The nodes were interconnected with one Sbus-2 card from Dolphin in a back-to-back configuration.

Ultra Enterprise ES-6000 The used Ultra Enterprise 6000 is configured with 12×200 MHz UltraSparc-I processors. Each processor has 512 KB direct mapped L2 cache, and main memory is 2 GB. The processors are connected with the Gigaplane system bus offering up to 2.5 GB/s sustained throughput.

3 CACHE OPTIMIZATIONS

One key factor to achieve good performance on microprocessor based machines is high utilization of the cache. Basic cache optimization techniques include array merging, padding and alignment, packing, loop fusion and blocking[9].

Array merging : By grouping related data arrays into one compound array, programs may increase spatial locality.

Padding and alignment : The smallest data transfer between the cache and main memory is one cache line. By aligning related data or arrays at cache line or page boundaries, the transfer of data between the cache and main memory is minimized. Padding and alignment can, when used right, reduce false sharing of data, improve calculation performance and communication time. Padding multidimensional arrays may increase communication time due to the increased data volume.

Packing : By using the smallest relevant data type (e.g. byte instead of integer) data volume decrease and spatial locality increase.

Loop fusion : If the dataset is larger than the cache and all data are involved in every step of the calculation, all data are stored in main memory between each step. By merging successive or non related steps of the calculations on the same dataset the traffic between cache and main memory are reduced (less capability misses).

Blocking (Tiling) : Blocking[8] is a technique for restructuring of the program loops to reuse chunks of data that fit in the cache and thereby reduce capacity misses. Choosing workset size according to cache parameters is often a nontrivial task.

Colouring is a special case of blocking. In calculations where a minor (constant) vector is combined with a major array or series of vectors (e.g. FFT or vector matrix multiplication) conflict misses in the cache can be reduced by duplicating the vector to a different internal offset in a direct mapped cache. The calculation can then either loop through all data and choose vector according to data address or divide the data in two non-overlapping sets and compute in two separate loops associated with one of the fixed vectors.

Cache line filling is an alternative way of looking at alignment. Traversing your dataset with non-unity stride (e.g. column access on a matrix) when datum is smaller than one cache line, can lead to false sharing with the other parallel processes doing the same calculation on neighboring datum. By grouping the calculations to match cache line boundaries, false sharing is avoided.

4 THE NPB-EP BENCHMARK

NPB-EP[13] provides an estimate of the upper achievable limits for floating point performance, i.e. the performance without significant inter-node communication. NPB-EP come in 4 sizes denoted S, A, B and C, with 2^{24}, 2^{28}, 2^{30} and 2^{32} random numbers. All sizes have a memory requirement which can be fitted into most workstation caches. Consequently, performance of the application is computing bound for almost all architectures.

4.1 Workload balancing

The workload is equally divided between nodes at compile time. Since all machines are dedicated, no hot-spots should appear between nodes. The intra-node workload is dynamically allocated at runtime time. Overhead due to dynamic scheduling (associated with a critical region), is small.

4.2 Adaption for SMP

Loop combining (unrolling + fusion) eliminates the buffering/memory access for the main loop in NPB-EP. In the original code random numbers are generated and stored in memory in one loop for further calculation in the second separate loop. By merging the two loops (generating 2 random numbers with post processing in the inner loop) memory/cache traffic is eliminated in the inner loop.

4.3 Results for Scali-HS

Results for the Scali-HS are presented in tables 1 and 2. All numbers are made relative to the performance for size A on one node using one computing thread(processor), compiled with Apogee's C compiler ($622.14s$ corresponding to 42.78 MFlops on Cray Y-MP/1[17]). The apf77 entry in table 1 is

Config. and compiler	Processes/speedup 1	2	3	4
apf77 MPI	0,82	1,60	2,37	3,08
apcc MPI	1,00	2,01	—	3,96
apcc SMP	1,00	2,02	2,70	4,03
cc SMP	1,08	2,13	2,84	4,24

TABLE 1: NPB-EP SIZE A SMP PERFORMANCE

Config.	Processors/speedup 4	8	16	32
Size S	3,92	7,51	12,48	24,50
Size A	3,95	7,98	15,95	31,01
Size B	4,00	7,93	15,54	31,86
Size C	3,98	8,07	15,81	31,52

TABLE 2: NPB-EP MULTINODE PERFORMANCE

the standard NPB-EP 2.2 code compiled on Apogee's Fortran compiler. The rest is our NPB-EP 1 code compiled with Apogee's (apcc) or SunWorks (cc) C compiler. The MPI entries are run with 1 to 4 MPI ('heavy weight') processes, while the SMP entries are run with 1 to 4 (light weight) posix threads directly. Table 2 shows results for NPB-EP 1 compiled on Apogee's C compiler. These tests are run with threads internally and MPI between nodes. For 8 nodes with 4 processors each NPB-EP used $20.11s$ for size A, $78.30s$ for size B and $316.54s$ for size C. For comparison a 16 node Cray T3D use $22.74s$ for size A and $91.83s$ for size B[17].

Performance scales almost linearly with the number of processors, due to the limited communication in NPB-EP. The largest performance gain for the NPB-EP 1 program on Scali-HS comes from inlining the $Log()$ math function. Unrolling the calculation loop gives no performance gain due to the already relative large calculation in the inner loop.

5 OPTIMIZATION OF NPB-FT

NPB-FT[13] performs the essence of many spectral codes, and is a rigorous test of long-distance communication performance. NPB-FT also come in 4 sizes denoted S, A, B and C with workset of $64 \times 64 \times 64$, $256 \times 256 \times 128$, $512 \times 256 \times 256$ and $512 \times 512 \times 512$ complex numbers (16 Bytes each) respectively. The memory requirement is dominated by 2 to 3.5 work buffers (a total of 4-7 GB for the C size). For all 1D FFTs and temporal buffering colouring as described in section 3 is applied. Simple pseudo code for the original FT benchmark is presented in figure 1.

5.1 Workload balancing

Inter-node dynamic load balancing of NAS-FT would involve moving the associated data over the network. The network is considered a limiting factor, thus an increase in inter-node traffic would slow down the calculation. Since the machines are dedicated and workload is (almost) equally divided between the nodes, no hot spots should appear.

5.2 Data driven scatter and gather

Instead of moving it to reflect their logical placement in the array or structure between each step of the computation, a mapping of data is kept by the programmer. Data then has to be gather before the calculation and scattered or remapped after. This approach spread out memory traffic, overlapping

```
nas-ft()
{
  <Initialize memory and network>;
  <Start timer>;
  <Fill fc-buffer with evolve constants>;
  <Fill pr-buffer with pseudo random numbers>;
  /* Forward FFT in 3 orthogonal dimensions */
  for (<all Y vectors>) <Forward FFT(Y,pr)>;
  for (<all X vectors>) <Forward FFT(X,pr)>;
  <Exchange data with all others nodes>;
  for (<all Z vectors>) <Forward FFT(Z,pr)>;
  for (<all time-steps>)
  {
     <Evolve the FFT converted data to ft-buffer>;
     /* Inverse FFT in 3 orthogonal dimensions */
     for (<all Z vectors>) <Inverse FFT(Z,ft)>;
     for (<all X vectors>) <Inverse FFT(X,ft)>;
     <Exchange data with all others nodes>;
     for (<all Y vectors>) <Inverse FFT(Y,ft)>;
     <Normalize numbers after FFT>;
     <Calculate part-checksum on available data>;
     <Gather and verify checksum>;
  }
  <printout timing and result>;
}
```

FIGURE 1: PSEUDO CODE FOR THE NPB-FT 2.X[13]

it with calculation, and thereby reducing the cost of data movement. This approach eliminating the explicit matrix transposing for the 3-D FFT.

5.3 Data placement

Pure data movement (transposing, copying) is limited by processor I/O or bus bandwidth[11]. Transposing the 3D matrix from NPB-FT size A on a single workstation with 100 MB/s usable memory bandwidth would take $22.5s$. Given 3 transposes (NPB2.0/2.1 makes 7 copy/transpose) per timestep (size A have the initial + 6 timesteps) the total time just moving data will add up to $53.7s$, which would be a substantial time of the total (See results in table 4). By avoiding explicit transposing/copying of the matrix between different FFT phases, some of this time can overlap with calculation and the memory pressure is eased.

Given the FFT lengths X_{len}, Y_{len} and Z_{len} over a 3D distributed matrix with dimensions X_{dim}, Y_{dim} and Z_{len} (assuming Z the highest dimension) divided equally over a number of $Nodes$, where

$$X_{dim} = X_{len} + X_{scew}\ ;\ Y_{dim} = Y_{len} + Y_{scew} \quad (1)$$

The X_{skew} and Y_{skew} are non-used memory gap and padding between vectors to get better cache performance in direct mapped caches. Assume:

$$X_{len} = 2^{m1}; Y_{len} = 2^{m2}; Z_{len} = 2^{m3} \quad (2)$$

$$min(Y_{len}, Z_{len}) > Nodes = 2^n \quad (3)$$

and let

$$Y_{div} = \frac{Y_{len}}{Nodes} = 2^{m2-n}; Z_{div} = \frac{Z_{len}}{Nodes} = 2^{m3-n} \quad (4)$$

Unless otherwise noted we use:

$$x \in [0, X_{len}\rangle, y \in [0, Y_{len}\rangle, z \in [0, Z_{len}\rangle$$

$Rank$ is the node identification starting at 0 ($Rank < Nodes$). For a single node we have that the internal offset from the start of a data buffer for the point (x,y,z) is given by:

$$\text{xyz2Offset}(x, y, z) = x + (y + z * Y_{dim})X_{dim} \quad (5)$$

which is the normal mapping for a 3D matrix. In a single node configuration no data have to be moved between the different phases of the calculation.

As the calculation is spread out on more nodes matrixes have to be divided between the nodes. If groups of Z-planes are distributed between nodes, the internal offset for the point (x,y,z) is given by:

$$\begin{aligned} &\text{xyz2Offset}(x,y,z) = x+ \\ &\quad (y + (z - Rank * Z_{div})Y_{dim})X_{dim} \\ &z \in [Rank * Z_{div}, (Rank+1)Z_{div}\rangle \end{aligned} \tag{6}$$

A 3D FFT can be viewed as 3 successive 1D FFT's of all

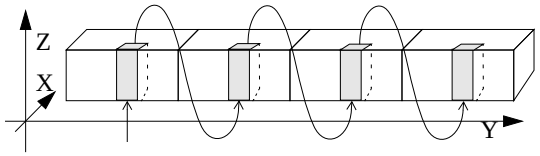

FIGURE 2: MAPPING OF Z VECTORS ON 4 NODES

vectors in all directions (X, Y & Z). To gather a set of Z vectors in a node it needs to exchange data with all other nodes. Data from equation 6 is exchanged over the network without any rearranging before or after the transfer. After gathering all data points with the same Z value in the same node (see section 5.4 and figure 2) we get:

$$\begin{aligned} &\text{xyz2Offset}(x,y,z) = x+ \\ &\quad ((y - Rank * Y_{div}) \\ &\quad +(z \text{ DIV } Z_{div})Y_{div} \\ &\quad +(z \text{ MOD } Z_{div})Y_{dim})X_{dim} \\ &y \in [Rank * Y_{div}, (Rank+1)Y_{div}\rangle \end{aligned} \tag{7}$$

Given integer arithmetic and that Y_{div} and Z_{div} are powers of two, equation 7 can be rewritten as:

$$\begin{aligned} &\text{xyz2Offset}(x,y,z) = x+ \\ &\quad ((y \text{ AND } (Y_{div}-1)) \\ &\quad +(z \text{ SHL } (m3-n))Y_{div} \\ &\quad +(z \text{ AND } (Z_{div}-1))Y_{dim})X_{dim} \end{aligned} \tag{8}$$

where $Z_{div} = 2^{m3-n}$, SHL is shift left and AND bitwise and. Multiplying with Y_{div}, Y_{dim} and X_{dim} (if $X_{scew} = Y_{scew} = 0$) can similarly be replaced by shift right (SHR).

5.4 Data movement

The communication pattern in NPB-FT is that all node exchange blocks (implicit transpose) of data ($X_{dim} \times Y_{div}$ numbers) with all other nodes as illustrated in figure 3 for one Z plane. This operation has to be repeated for all Z planes or all data have to be rearranged before (and further remapped or rearranged after) the transfer to increase the packet size.

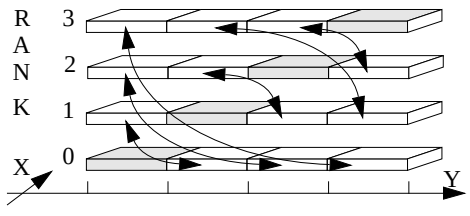

FIGURE 3: DATA EXCHANGE FOR 4 NODES (1 Z-PLANE).

The simplest communication approach is ping-pong communication between node pairs. While one node sends (`MPI_Send()`) a block of data, the other node receives (`MPI_Recv()`), before they swap roles as sender and receiver. Given that most networks have higher throughput for two-way than for one-way traffic, it is natural to choose primitives that can take advantage of this. When exchanging data with MPI the immediate send (`MPI_Isend()`), the combined send and receive (e.g. `MPI_Sendrecv()`) or one of the high-level collective primitives (e.g. `MPI_Alltoall()`) can be used as a replacement for standard blocking primitive.

It may be advantageous to reduce the number of work buffers to save memory. When sending and receiving from the same buffer, buffered send or some sort of temporal buffering of data is needed. Balancing usage of temporal buffering between nodes can easily be applied. Buffered send (`MPI_Bsend()`) implies copying of the message from user to kernel space on the sender side before starting transfers. This will usually introduce a performance penalty over standard send.

Package size in the data exchange is dependent of the number of nodes. Given that complete Z planes are stored continuously before or after the data transfer the largest package size without data reorganization for size A is 1 GB/$nodes$, i.e. given 4 nodes the maximum package size is 256 KB (the number doubles for every size increment). The part of the Z plane that needs to be moved to other nodes is given by $(nodes-1)/nodes$. Note that for two nodes the amount of copying after transfer is equal for single and dual work buffers (50 %).

5.5 Loop fusion

Each part of the NPB-FT computation wash through the complete dataset, which is larger than any reasonable cache (no inter-nest loop locality). By merging successive or non-related calculations on the same dataset these cache misses are reduced, and hopefully performance improved.

The normalization after the 3D FFT can be merged with evolve. The normalization of the final result is a multiplication with a constant. It can be done with the evolving, during or after the inverse-FFT. Since the evolve is the most data intensive part of the calculation, normalization is merged into the evolve.

Since the evolve is followed by inverse FFT in Z direction, the evolve calculation can be merged with data gathering before the inverse FFT in Z direction. This also enables the forward FFT transformed matrix to be stored with unit stride for the gathering, which usually improves the memory systems performance. In this case the matrix is divided in 'groups' of Y-planes (stored in a z,x,y manner). The internal offset for the point (x,y,z) is therefore given by:

$$\begin{aligned} &\text{xyz2Offset}(x,y,z) = z+ \\ &\quad (x + (y - Rank * Y_{div})X_{dim})Z_{dim} \\ &y \in [Rank * Y_{div}, (Rank+1)Y_{div}\rangle \end{aligned} \tag{9}$$

If one processors cache could hold one complete Z plane, data only would have to be fetched once from memory for a unified FFT in X and Y taking one Z plane at the time. This would reduce the intra-nest loop capacity misses in the cache. For size A one Z plane is 1 MB, which is quite common for relevant processors today. For the larger work sets the cache requirement doubles for each step (B: 2 MB, C: 4 MB), making this approach difficult to scale.

The two last operations in the loop are inverse FFT Y followed by the checksum calculation. By doing the check-

sum calculation directly after the the inverse FFT Y calculation before the data is scattered back into the buffer, the scatter is no longer needed and one complete data movement of the workbuffer is eliminated. Since this doesn't seem to be within the intention of the benchmark, it has not been implemented.

5.6 Loop unrolling

For the gather before and scatter after FFT Y/Z, alignment and slice sizes must match cache lines to eliminate false sharing. Since these slices are relatively small (typically no more than 16 vectors) complete unrolling can be used, thus eliminating the overhead of one loop level. This unrolling/slicing arrange the memory reference as bursts of unit strides accesses, hence usually improving memory performance. If the compiler is good in determining aliasing, automatic unrolling may give better code than programmed unrolling e.g. in the evolve and exponent calculation. The effect of loop unrolling is restricted by the number of registers available.

5.7 Serializing memory access

If the memory bus or controllers is a bottleneck, serializing the filling the cache or writing to memory may ease the problem. There are 10 cases in the NPB-FT that can be subjected to serialization: storing of pseudo-random numbers and exponential values, loading and storing evolve and FFT (X, Y and Z) data. Serialization of memory accesses imply using synchronized threads since synchronizing MPI processes on the same node would be too slow.

5.8 Parallel calculations and communication

Since NPB-FT for most workstations will be communication bound, overlapping communication with calculation seems promising. Since there are no dependencies between calculations for the different time steps, communication can completely overlap with every calculation (except the start of the first and the end of the last). Using the single workbuffer approach but allocating two buffers makes it possible to exchange data for timestep T in parallel with doing the last part of the calculation for timestep T–1 and the start for timestep T+1. The two buffers switch roles for each timestep, resulting in a neat pipelined calculation.

Another relative simple approach is to send each Z plane as they are done with FFT in X direction. Since this is relative difficult to administrate and for clusters does not have the potential of the approach above we will not discuss it further.

5.9 Alignment of the different buffers

By aligning the different buffers (2 to 4) so they map 1/4 cache size apart in the cache, the conflict misses in the evolve will be reduced (to zero if evolve constants are calculated on the fly). In multilevel caches the skew must reflect all cache levels in order to minimize cache conflicts.

5.10 Special system adaption

If the node have high calculation potential relative to memory bandwidth, calculating the exponential terms in the evolve on the fly (not storing them in memory) can pay off. Since storing the exponential terms in memory have a start-up penalty, the effect of calculating is best for few time-steps. The reduced memory requirement (0.5 work buffer) comes as a free bonus.

When the number of nodes increase beyond two (see section 5.3), less data need to be copied when using two work buffers. Assumed that the inter-node bandwidth is significantly less than double of the memory bandwidth, little is gained when going from one to two work buffers even for larger configurations. Using two workbuffers (as in NPB2) increase the memory requirement by 128 MB for the A size, and quadruples for every size increase.

By inlining math and small leaf routines called in tight loops a performance can be improved due to the reduced overhead and better register allocation. By inlining (e.g. by using macros in C) the pseudo random routine the pseudo-random calculation time is reduced with ≈10 % on Scali-HS. If a compiler has problems utilizing all floating point registers in leaf procedures without math functions (e.g. gcc on SPARC), manual register allocation may improve performance. Since this require assembly inlining witch is against the NPB policy, this is not applied here.

6 NPB-FT: PERFORMANCE RESULTS

Padding the X dimension ($X_{scew} \neq 0$) introduce a different access pattern for data scatter and gathering in the FFT Y and Z. Given the relatively short vectors for the NPB-FT ($X_{len} = 256$ for size A) the increase in data volume is substantial even for relatively small X_{scew}. A small performance improvement was observed when padding X for a single node system ($X_{scew} \leq 4$). A small performance degradation was observed for multinode systems due to the relatively slow interconnect and forced alignment for data transfer. No performance changes were observed for padding Y.

For all platforms no difference in timing between intra-node work allocation at compile- and run-time was observed. Since time-slices on Solaris are relatively long, no performance improvement was achieved when fixing threads to processors.

3 FFT implementations were used for comparison: Stockhams FFT used in NPB2.x[13], Ron Mayer's FFT wrapper round the Hartley transform, Swarztrauber's FFT with and without colouring (section 3) from the calling function, later denoted as NPB, Mayer, Colour and Swartz.

As the number of nodes increases transfered data increases in volume and the maximum package size decreases (section 5.4), thus resulting in reduced communication performance. Running with more than one MPI process per node (e.g. one per processor) have the same negative effect due to the more fine grained data distribution.

A Scali-HS node or an Ultra-2 can hold up to 1 GB memory. Consequently, only sizes S and A can be run on a single node. The times and figures in the rest of the section, unless otherwise stated, is taken for size A.

6.1 Results for Scali-HS with MPI.

The processors in Scali-HS delivers 62.5 MFlops peak at double precision. All the data to and from the four processors caches are transferred over a 45 MHz MBus. Keeping all processors supplied with sufficient data therefore require special attention from software. For the sizes with few iterations (S and A) it therefore pays off to calculate the ex-

ponential terms in the evolve function on the fly instead of storing them in memory.

6.1.1 Single Node 256 KB cache

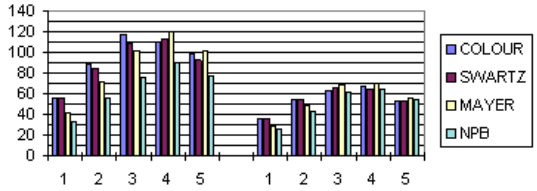

FIGURE 4: FFT X AND Y PERFORMANCE[MFLOPS] FOR 1-5 THREADS, SCALI-HS WITH 256 KB CACHE

A serialized filling of the cache before the calculation decreases the FFT X time from 20.6 to $19.2s$ for Mayer and increases from 20.9 to $21.3s$ for Colour. The results are similar for FFT Y and Z, but with a performance degradation when both scatter and gather are serialized. The reference calculation for the rest of the subsection is therefore with serialization for the cache fetch in FFT X and the scatter after FFT Y and Z at $127.27s$ (59.25 MFlops). Figure 4 shows the FFT X and Y (FFT X to the left) performance as a function of the number of threads for the reference measurement. Up to the number of processors, the number of threads is equal to the number of processors used.

The rest of the numbers in this subsection are given using the Mayer FFT and 4 computing threads. Doing normalization and evolve in separate loops takes $44.4s$ compared to $27.6s$ when normalization is done inside the evolve loop, which is a 13 % increase in the total time. Calculating the evolve constants on the fly reduces the evolve and exponentiation time from 31.1 to $28.7s$.

Mixing the normalization, evolve and (inverse) FFT Z in one loop reduces the time in this pass from 68.1 to $53.7s$. Mixing the FFT X and Y for one Z plane increases this time from 53.1 to $53.9s$.

Slice	1	2	4	8	16
X	19.48	19.43	19.35	19.32	19.39
Y	93.52	48.48	37.86	33.79	35.26
Z	102.2	66.04	58.23	53.47	44.08

TABLE 3: FFT TIMING [S] VS FFT SLICE SIZE

Table 3 shows that while FFT X timing is little affected by slice size, FFT Y and Z have massive false sharing for slice = 1 and generally improve performance with larger slice size.

The best run for size A gave $98.08s$ (76.89 MFlops) using the Mayer FFT, slice size $X = 4$, $Y = 8$ and $Z = 16$, mixing the normalization, evolve and (inverse) FFT Z, calculating the exponents to the evolve and serialization for the cache fetch in FFT X and the scatter after FFT Y and Z.

6.1.2 Single Node 512 KB cache

Figure 5 shows the FFT X and Y (FFT X to the left) performance on a Scali-HS node with 512 KB cache per processor as a function of the number of threads. The parameters are the same as in figure 4.

Doubling the cache size should ideally result in 50 % less conflicts between data and constant vectors. For size A this reduces the memory traffic by a factor 1/128 in FFT

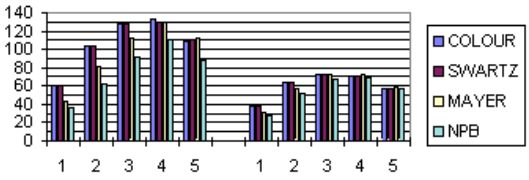

FIGURE 5: FFT X AND Y PERFORMANCE[MFLOPS] FOR 1-5 THREADS, SCALI-HS WITH 512 KB CACHE

X in all FFTs except Colour. Interestingly the Colour algorithm is the FFT that benefits the most from the larger cache (20 %), relative to the others (≈8 %) for FFT X. For FFT Y the trend is the same, but the relative improvement are half due to the more complex gather and scatter.

The relative performance response to algorithmic changes are very similar to the 256 KB cache case, but while with 512 KB cache the Mayer outperformed Colour with typical 5 % they are similar in the 512 KB cache case - giving Colour a 9 % performance increase and Mayer 4 %.

The best run for size A was observed with $105.11s$ (71.74 MFlops) using the same parameters as section 6.1.1.

6.1.3 Multinode

Multinode results are obtained using dual SCI interfaces in each node. The nominal inter-node capacity over MPI is 27 MB/s for one-way and 14 MB/s in each direction for two-way traffic for 512 KB messages with 256 KB cache per processor (≈15 % higher for 512 KB cache). This is limited by the memory bandwidth on the MBus. Furthermore, the MPI implementation used on Scali-HS makes an extra copy of data to/from kernel space. This comes in addition to a single copying on the receiver side in the SCI driver. The

Size	Nodes	Time(s)	MFlops	MFlops/ node
S[1]	1	2.55	74.39	74.39
A	1	105.11	71.74	71.74
A	2	77.12	97.78	48.89
A	4	44.74	168.56	42.14
A	8	25.73	292.99	36.62
B	2	860.90	109.25	54.63
B	4	466.93	201.43	50.36
B	8	257.74	364.90	45.61
C[2]	8	1231.11	328.48	41.06

TABLE 4: NPB-FT RESULTS WITH 512 KB CACHE

results in table 4 are achieved using the same parameters as section 6.1.1 with dual work-buffers (except single node) and storing the exponents to evolve for size B and C.

Reduced packet size and increase in transferred data volume is the dominant factor for node performance degradation when going to larger configurations. The time spent (explicit) moving data on a dual node configuration for a size A computation is $\approx 17s$ (10.6 MB/s) for two and $\approx 21s$ (8.9 MB/s) for one SCI interfaces per node, i.e. 'adding' 29 or 35 % to the pure computation time. For larger configuration this becomes even worse (46 or 54 % in a 4 nodes configuration). Transfer data one way at the time (ping-pong

[1] $2.41s$ at 78.44 MFlops for Colour

[2] Run with 3 threads and Colour. The result should improve 10 - 17 % if 4 threads and Mayer were used (360 - 400 MFlops i.e. 45 % - 50 MFlops/node).

communication) between two nodes increased data transfer time with $\approx 3s$ for one and $\approx 1s$ for two SCI interfaces.

As the number of iterations increase (size B and C) storing the evolve constants increase performance, but also increase the memory requirement. Duplicating the work-buffer have a positive effect on configurations larger than 2 ($\approx 3s$ for 4 nodes size A) due to the reduced copying. The extra buffer can alternatively be used for overlapping calculation and data transfer. Unfortunately on Scali-HS this slows down the calculation so much that the total time increase ($0.5s$ for dual node, $2s$ for quad node size A).

6.2 Results for Ultra-2 using raw SCI

The Ultra-2's were back-to-back connected with a single SCI interface each similar as in the experiments in [14]. Since MPI for the Ultra-2's wasn't available at the time of the experiments, the SCI driver interface for DMA transfers was used directly. Due to the high memory bandwidth, the extra copying not taken should have minor impact on performance.

6.2.1 Single Node

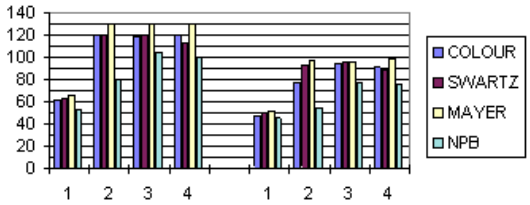

FIGURE 6: FFT X AND Y PERFORMANCE[MFLOPS] FOR 1-4 THREADS, ULTRA-2 WITH 1 MB CACHE

A serialized filling of the cache before the calculation increase the FFT X time from 18.0 to $18.4s$ (Mayer). Serializing the scatter and/or gathering FFT Y increased the time from $24.0s$ to between 25.0 and $30.5s$. For FFT Z the difference were minor ($23.1 \pm 0.2s$). Thus the reference calculation for the rest of the subsection is with no serialization, at $78.6s$ (95.8 MFlops). Figure 6 shows the FFT X and Y (FFT X to the left) performance as a function of the number of threads. The performance is little affected by the number of threads when the number of threads is larger or equal to the number of processors.

Doing normalization and evolve in separate loops takes $12.7s$ compared to $7.9s$ when it is done in the same loop. This is a 6.1 % increase in the total time. Calculating the evolve constants on the fly increases the evolve and exponentiation time from 10.8 to $16.9s$. Mixing the normalization, evolve and (inverse) FFT Z in one loop reduces the time in this pass from 30.7 to $29.2s$. Mixing the FFT X and Y for one Z plane reduces time from 42.0 to $40.0s$. Using dual workbuffers reduced the FFT timing with $0.6s$. Increasing the FFT slicing beyond 4 gives only minor changes.

The best run for size A gave $76.12s$ (99.0 MFlops) with dual workbuffers and normalization, evolve and FFT Z in one loop. This figure should be reduced 1-$2s$ by removing debug information and mixing the FFT X and Y.

6.2.2 Dual Node

With a single SCI interface an inter-node bandwidth of 14.6 $\pm$ 0.2 MB/s in each direction during two way traffic was achieved. This is comparable to the 18 MB/s achieved in [14] with similar transfer sizes (64KB). These results can probably be improved by running using more threads than processors and possibly by giving higher priority to threads doing communication. The UltraSparc experiments were done within a very limited timeframe. Due to the small number of processes and few (two) threads used in the experiment no performance gain was observed when overlapping computation and data transfer. The best run gave $52.85s$ (142.6 MFlops) with normalization, evolve and FFT Z in one loop. The times can be significantly improved by using two SCI interfaces. This should cut the communication time by around 50 % according to bandwidth results in [14]. Also using more than two threads with overlapping computation and data transfer has a potential for improvements.

6.3 Results for Enterprise with threads

The Enterprise-6000 server consists dual processor boards with private memory connected with the Gigaplane backplane bus. A serialized filling of the cache before the calculation increases the FFT X time from 3.1 to $5.2s$. Serializing the scatter and/or gathering in FFT Y increases the time from $4.0s$ to between 6.9 and $8.6s$. For FFT Z it is even worse. The reference calculation for the rest of the subsection is with 12 threads and no serialization at $14.52s$ (519.2 MFlops). The single thread performance is $151.2s$ (49.8 MFlops). Figure 7 shows the FFT X and figure 8 the FFT Y performance as a function of the number of threads (processors).

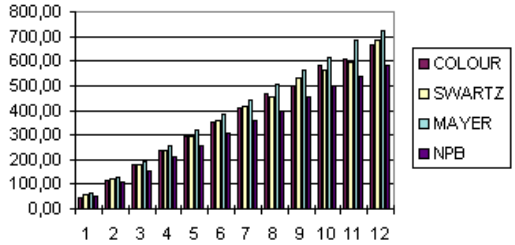

FIGURE 7: FFT X PERFORMANCE[MFLOPS] FOR ES-6000

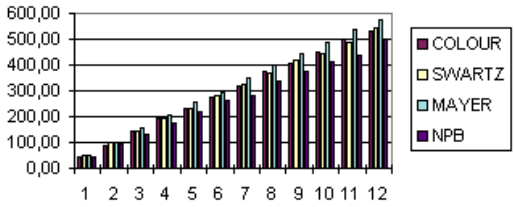

FIGURE 8: FFT Y PERFORMANCE[MFLOPS] FOR ES-6000

Doing normalization and evolve in separate loops takes $7.0s$ compared to $1.6s$ when normalization is done inside the evolve loop, which is a 37.1 % increase in the total time. Calculating the evolve constants on the fly increase the evolve and exponentiation time from 2.5 to $3.1s$.

Mixing the normalization, evolve and inverse FFT Z to one loop reduces the time in this pass from 5.8 to $5.2s$. Mixing the FFT X and Y for one Z plane reduce timing from 7.2 to $7.1s$. Increasing the FFT slicing from 4 to 8 makes a small positive effect in Y ($0.1s$), but none for X and Z (less than $0.05s$).

The best run for size A gave $13.80s$ (546.2 MFlops = 45.5 MFlops per processor) with normalization, evolve and FFT Z in one loop, FFT X and Y in another and 12 threads. Time can be reduced by 0.1-0.2s by removing debugging information and increasing the slice in FFT Y. Unfortunately we were not able to run any tests larger than size A within the timeframe of the experiments.

6.4 Summing up NPB-FT results

Running NPB-FT with one MPI process per processor in an SMP increase memory traffic (due to finer grain in the data distribution) and make intra-node synchronization more cumbersome. Consequently, intra-node parallelism should be done by a threads or a parallelizing compiler.

Cache optimization techniques have high impact on performance when the bus is limiting performance. Given the technology trend on memory relative to processor performance (see section 1) optimization on cache performance will become more importance in the future. For a single node Scali-HS application timing was reduced by 23 % for NPB-FT size A. On the better balanced Ultra-2 the single node timing was only reduced by 3.2 %, but given faster upgrades and more processors in the SMP the improvement potential from a naive approach will increase. The FFT routines were kept cache aware at all times, thus hiding some of the true potential of optimization.

In todays marketplace a workstation with dual processor UltraSparc at 200 MHz is similar in price to a workstation with quad processor HyperSparc at 125 MHz. Given that the memory pricing is the same, the two clustered machines are directly comparable in price. For the NAS-FT size A, the Ultra-2 with large cache perform 28 % better than a Scali-HS node in a single workstation environment. The price of a single workstation with only Ethernet connection are obviously cheaper per node than a cluster. Given that the user memory requirement is high (e.g. near 2 GB as for NPB-FT size B) more than one node is required to hold the memory, introducing the need for clustering. Given the 2 GB memory requirement for size B, the costup over speedup factor[20] is near one for Scali-HS configurations from 2 to 8 nodes. For size C (near 8 GB user memory requirement) the 8 node Scali-HS have 25 % better price/performance than a single workstation, given that the memory could have been fitted in one workstation.

The same is expected to apply for the planned Scali-US cluster where the HyperSparc based workstations in Scali-HS are replaced with UltraSparc based systems. Given that the pricing is similar, the UltraSparcs have a better price/performance for the NPB-FT. The better bus capacity on the Ultra's improves both communication (using two SCI interfaces) and processing speeds. Thus the UltraSparc based nodes are expected to scale better than the HyperSparc based for larger configurations, further improving price/performance. This gives an expected 50 % better price/performance from '1' to 8 workstations for size C.

The very high capacity Gigaplane bus connecting the processor-boards in the Enterprise-6000 server enables the NPB-FT to scale almost linearly from 1 up to the 12 processors available for the experiment. Since the processors are the same as in the Ultra-2 and the interprocessor bandwidth is high, the performance gained by rescheduling the highlevel code is limited ($\approx$5 %). The pricing on large SMPs (memory, processors and cabinet) are relative high compared to the smaller workstations. This gives the workstation clusters an price advantage per processor, even with the cost of cluster infrastructure and function duplication. Given the NPB-FT size B an 8 node Scali-US would have slightly (15 %) better price/performance than an 12 processor Enterprise-6000, but would outperform it by 30 % when going to size C. The performance for the two machines are similar.

In this subsection list prices have been used and no adjustments for actual memory usage has been performed. Adjusting the numbers for actual memory usage may change the absolute values of numbers, and improve the position to the HyperSparc based solutions slightly.

7 RELATED WORK

Todays compilers are still limited in their ability to generate optimal code for scientific applications. Extensive research is being performed on parallelizing and optimizing compilers (e.g. the preprocessors KAP from Kuck and Associates, Inc. and VAST from Pacific Sierra Research, Inc.).

Parallelizing compilers for SMPs usually try to find and schedule the outermost loop without data dependencies. The rest of performance improvement come from blocking[8] and architectural based rescheduling of machine code, usually leaving the data mapping as is. Careful placement of data is another way to achieve performance. High Performance Fortran (HPF) approaches this by letting the programmer give directives about data placement, and trust the compiler to figure out the optimum movement of data. Algorithmic based data remapping e.g. the SUIF compiler[19] is another approach for addressing the cache/memory bottleneck problem. The SUIF compiler achieved a linear speedup for NAS-EP (slightly modified) but no speedup at all for NAS-FT from the NPB 2.x suite running on an 8 processor Alpha SMP[5]. Parts of the NPB1 has been implemented for distributed shared memory (ThreadMarks) and message passing (PVM) as described in [10] and [3]. On a cluster of 8 HP735 connected with 100 Mb/s FDDI network they achieved speedups of 7.9 for NPB-EP and 4.41/5.47 for NAS-FT[10]. On an 8 processor IBM SP/2 they achieved of speedups of 3.06/5.12 for NPB-FT[3]. These works mention only small data sets for NAS-FT (size S and A).

Some previous work using Dolphin SCI interfaces for 'real' applications exist. Performance of a SAR (Synthetic Aperture Radar) application on workstations clustered with the previous generation of SCI interfaces is measured in [2]. Measurements of a small FFT benchmarks using an implementation of Split-C over SCI is presented in [7].

8 CONCLUSION

Two examples of CFD kernels with different communication requirement have been studied in detail. The NPB-EP have nearly no communication and like other embarrassingly parallel problems perform well on any parallel machine. In the NPB-FT on the other hand, large data volumes are exchanged for every time step with all the other nodes taking part in the calculation. Both kernels performed well on the available clusters of workstations, and had a good price/performance even for large configurations compared to a performance compatible Enterprise-6000 server. The costup over speedup factor are kept one or below for all cluster configurations.

Making the software architecture aware (e.g. minimizing data movement when the interface to memory is the bottleneck) and using top of the line compilers has perfor-

mance gain sufficient to be justified for production code and libraries. The potential is largest for architectures with limited bus or memory systems.

This work demonstrates scalability of SCI based workstation clusters for real applications and shows that clusters of dedicated workstations with high speed interconnects can be configured to powerful systems. Our experiment is the only reported result we have found for running the NAS-FT size C on a cluster of workstations. Although a large SMP may have higher performance for a certain number of processors, clusters are justified from a price perspective for midrange supercomputing, in particular for applications with a high memory requirement. As for performance, there is always the possibility (within reasonable limits) to add more nodes to a cluster.

9 FURTHER WORK

Todays supercomputer communication performance can be illustrated by the Cray T3E interprocessor interconnect which gives 140 MB/s throughput per two processors and a 3μs minimal latency. As a comparison, a PCI based SCI board from Dolphin shows promising results with more than 85 Mbytes/s of bandwidth and 2.6μs latency. Using multiple such boards to connect medium range SMPs with multiple PCI buses, would form a very powerful cluster.

An interesting option for future work is to look at similar or perhaps even more demanding scientific applications on a cluster of quad processors UltraSparcs with multiple SCI boards connected to multiple I/O buses to circumvent bottlenecks in the I/O bus implementations. Since the time of the experiments in this paper, bandwidth for the Sbus/SCI interfaces have improved by another 20 % .

The SCI specification describe a coherent shared memory architecture. Todays implementation of SCI connect to the computers I/O bus. With implementations of SCI connected to the internal system bus, much better performance is expected. An interesting experiment would be to compare performance of shared memory for the NPB-FT to our message passing based implementation. Since SCI in fact implement true shared memory the results from [10] and [3] may be reversed.

Acknowledgments

We would like to thank Håkon Bugge, Scali AS for his patient listening and comments on cache and SMP performance, Stein Gjessing at the University of Oslo for his helpful comments and the Scali people for system support on the Scali-HS. The Ultra based experiments were done during Knut Omangs stay at Sun Microsystems in 1996.

References

[1] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, pages 54–64, Feb. 1995.

[2] H. Bugge and P. Husøy. Dedicated Clustering: A Case Study. In *Proceedings of 3rd International Workshop on SCI-based High-Performance Low-Cost Computing*, pages 1–12, 1995.

[3] A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the performance of software distributed shared memory as a target for parallizing compilers. In *Proceedings of Eleventh International Parallel Processing Symposium*, Apr. 1997.

[4] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report v1.1, University of Tennessee, Knoxville, TN, June 1995.

[5] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S.Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, pages 84–89, Dec. 1996.

[6] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill Inc., 1993. ISBN 0-07-031622-8.

[7] M. Ibel, K. E. Schauser, C. J. Scheiman, and M. Weis. High-performance cluster computing using scalable coherent interface. In *Proceedings of 7th International Workshop on SCI-based High-Performance Low-Cost Computing*, pages 45–54, Mar. 1997.

[8] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of 4th International conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[9] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, pages 15–26, Oct. 1994.

[10] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on network of workstations. In *Proceedings of Supercomputing '95*, Dec. 1995.

[11] J. D. McCalpin. STREAM: Measuring Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[12] K. S. McKinley and O. Temam. A Quantitative Analysis of Loop Nest Locality. In *Proceedings of 7th International conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–104, 1996.

[13] NPB : NAS (The Numerical Aerospace Simulation Facility at NASA Ames Research Centre) Parallel Benchmarks. http://www.nas.nasa.gov/NAS/NPB/.

[14] K. Omang and B. Parady. Performance of Low-Cost UltraSparc Multiprocessors Connected by SCI. In *Proceedings of Communication Networks and Distributed Systems Modeling and Simulation, Phoenix Arizona*, pages 109–115, Jan. 1997. Also available at http://www.ifi.uio.no/~sci/papers.html.

[15] K. Omang and B. Parady. Scalability of SCI Workstation Clusters, a Preliminary Study. In *Proceedings of 11th International Parallel Processing Symposium*, Apr. 1997.

[16] G. F. Pfister. *In Search of Clusters*. Prentice Hall PTR, 1995. ISBN 0-13-437625-0.

[17] S. Saini and D. H. Bailey. NAS Parallel Benchmark Results 12-96. Technical Report NAS-96-18, Numerical Aerospace Simulation Facility, NASA Ames Research Center, December 1996.

[18] IEEE Standard for Scalable Coherent Interface (SCI), Aug. 1993.

[19] The SUIF (Stanford University Intermediate Format) compiler system. http://suif.stanford.edu/.

[20] D. A. Wood and M. D. Hill. Cost-effective parallel computing. *IEEE Computer*, pages 69–72, Feb. 1995.