SMP vs Vector: a Head-to-head comparison

Russell K. Standish

High Performance Computing Support Unit
University of New South Wales
http://parallel.hpc.unsw.edu.au/rks

May 9, 2001

Abstract

The last decade has seen the growth in the fortunes of parallel computers constructed from commodity microprocessors, and the corresponding decline of traditional vector supercomputers. A lot of hype surrounding how high performance computers should be used, and what they're capable of, has made it difficult for the HPC customer to answer the question "Given a particular application, and a particular budget, what is the best technology for achieving the goals?". Focus on peak theoretical speeds (eg TOP500) or on extrapolations from certain benchmarks (Ahrendt's list) tends to distract from what the customer really needs to know.

In this paper, I compare the performance of a single NEC SX5 Vector CPU versus an SGI O2000 system with as many CPUs as one can buy for the same price as the SX5 system (40 CPUs at the prices paid for by the *Australian Centre for Advanced Computing and Communications*). The benchmark codes consist of small pedagogical examples written for an SMP programming course by the author, and later adapted as a benchmark suite for SMP computers. Whilst not comprehensive, this benchmark suite does illustrate the sorts of problems where one architecture has an advantage over the other.

1 Introduction

The last decade has seen the growth in the fortunes of parallel computers constructed from commodity microprocessors, and the corresponding decline of traditional vector supercomputers. As part of a strategy to regain market share, vector supercomputer manufacturers lump their competitors under the heading *scalar parallel*, in spite of the obvious difference between a between (lets say) a

64 processor SMP computer like the Origin 2400[1] and 32 node Beowulf system constructed from dual CPU Pentium 4 PCs.

Typically comparisons are made between single CPU performances in either case, and throughput comparisons made — eg. if you have 10 simultaneous users, running an application that is 30 times faster on a vector CPU than on the commodity processor, then the throughput should be 3 times faster on the vector computer than the scalar machine (these are not unrealistic figures, by the way). However, while many parallel computers are used to run single CPU jobs (ie throughput optimised), this is not necessarily the best way of utilising these machines. It is very clear, that if throughput is your primary requirement, the most cost effective way of achieving this goal is to invest in a Beowulf cluster, whose price-performance is hovering around the $1 per Mflops.

However, if turnaround times are more important (as they are for many HPC applications), then the only reasonable way to use parallel computers is to run parallel applications on them. Unfortunately, programming parallel applications in MPI (or even HPF) to run on clusters is difficult, and apart from a number of special cases, requires large inputs of programming expertise and time. Some vector vendors point to this difficulty in parallel programming as further justification of their paradigm. This argument is specious, as programming an SMP computer using OpenMP and an automatically parallelising compiler is typically no more difficult than programming a vector computer[2]. However, with the present state of technology, use of OpenMP rules out the use of Beowulf clusters.

So how to get a meaningful comparison between SMP and Vector computers, taking into account economic factors? Simply comparing peak GFlops doesn't help, as it is well known that vector supercomputers often perform at greater efficiencies than commodity processors due to the superior memory architecture employed. At ac3[3], a 2 processor NEC SX5 with 16GB of memory and a 64 processor SGI Origin O2400 with 32GB of memory were purchased at the same time, each with 3 years of maintenance included. Apportioning the total machine costs evenly over the processors results in each vector CPU costing about the same as 40 CPUs in the Origin[4]. Whilst many discounts were offered by the vendors to ac3 to come to a final figure, at all stages of the tendering process, and also with the similar process at APAC, the ratio was somewhere between 40 and 50 CPUs, at least between the most competitive solutions from either camp.

[1] Technically, the Origin is a cc:NUMA system, however this fact has little relevance to the programmer of the system

[2] In the authors experience, most codes can be handled in similar ways in both paradigms, with some codes being easier in one environment than the other (but not obvious at the start which environment is better suited).

[3] Australian Centre for Advanced Computing and Communications

[4] Including administration costs changes the calculations very little, as each machine requires a dedicated system administrator. In fact the hardware costs dominate the total cost of ownership

```
      do i=1,n
         do j=1,n
            do k=1,maxiter
               if (real(z(i,j))*real(z(i,j))+
     $              aimag(z(i,j))*aimag(z(i,j)).gt.4.0) then
                  T(i,j)=k
                  goto 1
               endif
               z(i,j)=z(i,j)*z(i,j)+kappa(i,j)
            enddo
 1          continue
         enddo
      enddo
```

Figure 1: Kernel of Mandelbrot problem. It is parallelised over the outer two loops, with an early termination of the inner loop providing a load balance problem

2 The Benchmark Suite

So the simplest test is to run the same code optimised for the respective machines, using up to 40 threads on the SMP system. An OpenMP benchmark suite was developed from a set of small pedagogical examples written for an SMP programming course by the author. This consists of kernel codes for dense matrix-vector multiply, solving a heat equation (Laplacian) on a 2D grid, sparse matrix-vector multiply, force routine for a 2-D $n-$body calculation and a Mandelbrot set calculation. Prior to the ac3's Origin's commissioning as a production system, this OpenMP benchmark suite was run on the Origin system exhaustively at for all values of `OMP_NUM_THREADS` from 1 to 64, and for a large range of problem sizes, something that can be difficult to do with a production system. The same set of codes was ported to the vector computer. A further code benchmarking the *dgemm* level 3 BLAS routine (matrix-matrix multiplication) was added, as this sort of operation gains substantial benefits from cache reuse on SMP system. The two matrix-vector routines, and the heat routine all autovectorised, just as they autoparallelise. With matmul, the inidices of the array were reversed, as the compiler automatically swapped which loop vectorised, and the resulting vector add operation had better performance than the original vector sum. The $n-$body calculation required a simple insertion of a compiler directive, whereas the OpenMP version of the coded need some explicit coding changes using a thread private local variable.

Mandelbrot proved more difficult. This problem is a classic example of load imbalance, and OpenMP handles it simply through the `SCHEDULE(DYNAMIC)`

```
 do k=0,maxiter
    do i=1,n*n
       if (T(i,1).eq.0) then
          if (real(z(i,1))*real(z(i,1))+
$              aimag(z(i,1))*aimag(z(i,1)).gt.4.0) then
           T(i,1)=k
          else
           z(i,1)=z(i,1)*z(i,1)+kappa(i,1)
          endif
      endif
    enddo
 enddo
```

Figure 2: Vectorised kernel of the Mandelbrot problem. This code vectorises nicely, but at the cost of extra work computed and thrown away

directive. However, the vectorising compiler could not vectorise the original serial code at all, due to the presence of the early termination in the inner loop (see Fig 1). By moving the innermost loop to the outside, and replacing the early return by a masking operation as in Fig 2, the code vectorised nicely, but at the cost of extra operations that are masked away by the inner `IF` statement. In this example, `maxiter=1000`, but the average value of `k=2.7`, consequently for most iterations, the inner loops are very sparse indeed. However, the neither the use of compiler directives (`!CDIR SPARSE`) to indicate this sparsity nor restructuring the code to explicitly perform the gather and scatter operations improved the performance, so the final results used the code in Fig 2.

The results of the performance are reported in MFlops. This was measured by estimating the number of Flops performed by the calculation, and dividing by the CPU time. In the OpenMP case, the CPU time of the master thread (thread 0) was used, as this corresponded quite accurately with the wall clock time when run on an unloaded machine. The benchmark suite and results are available from ftp://parallel.hpc.unsw.edu.au/software/smp-vs-vector.tar.gz.

3 Results

The results are shown in Figs 3–8.

The matrix vector multiplication example is perhaps the most generic. Vector processors have a definite advantage over SMP machines at small vector lengths, due to the overhead of thread creation. As vector lengths are increased, the thread creation costs are amortised over more computation, and SMP systems become more competitive for vector lengths greater than 1000 or so. In

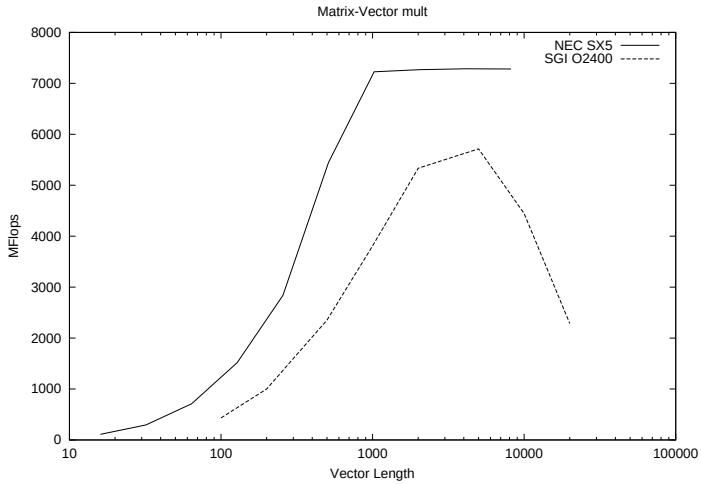

Figure 3: Performance of matrix-vector multiplication as a function of vector length

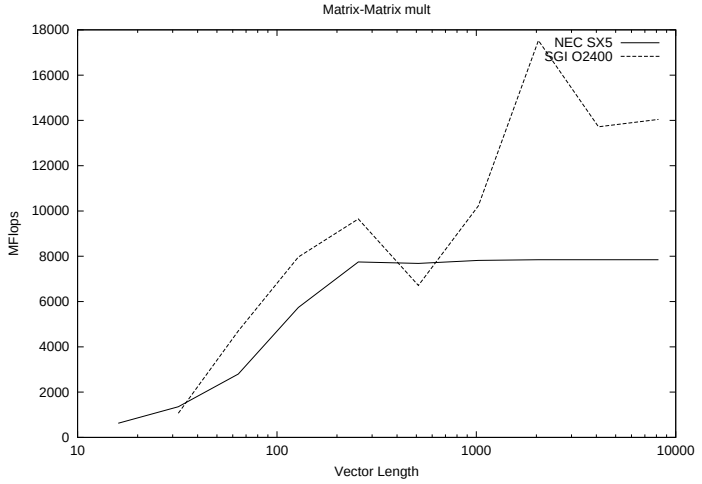

Figure 4: Performance of matrix-matrix multiplication (dgemm) as a function of vector length

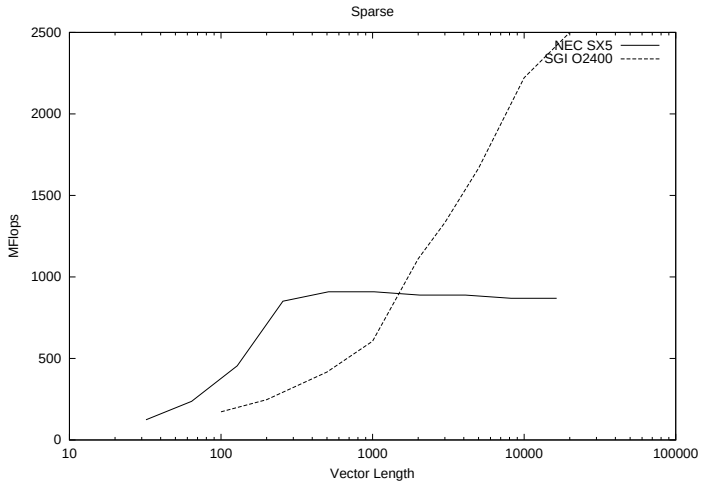

Figure 5: Performance of sparse matrix-vector multiplication as a function of vector length

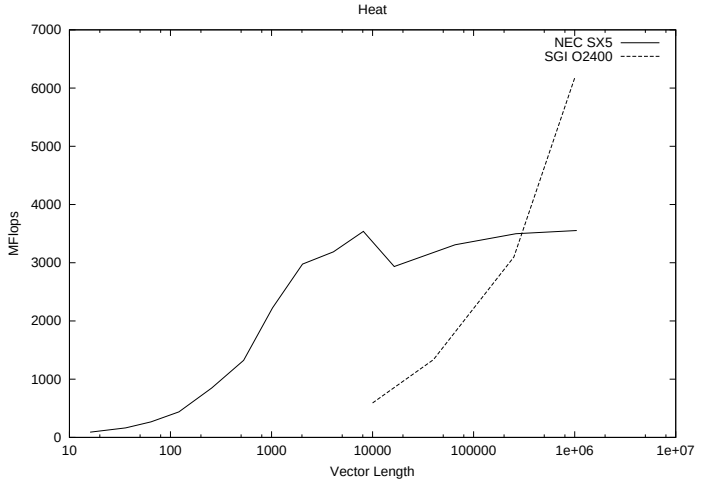

Figure 6: Performance of a heat equation solver on a square grid. Vector length is taken as the square of the grid size

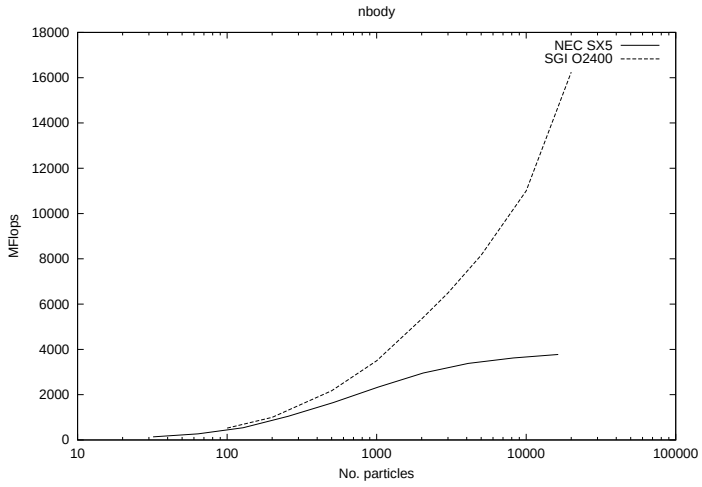

Figure 7: Performance of a 2D $n-$body force routine

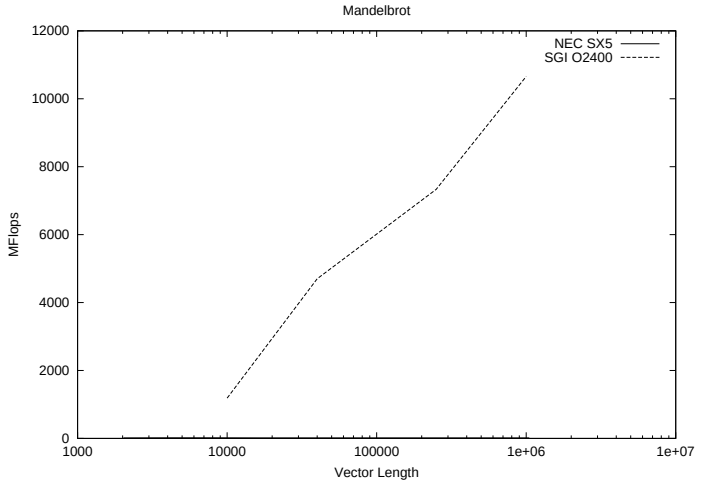

Figure 8: Performance of a computation of the Mandelbrot set. Vector length is taken as the square of the grid size

this instance (basically a BLAS level 2 operation), cache reuse is nonexistent, so the vector machine outperforms the SMP system at all vector lengths.[5] Finally, the problem gets too large to fit in the cache, and proves devastating to the SMP machine's performance. At large vector lengths, the superior memory architecture of the vector machine shows its strength.

Matrix matrix multiplication demonstrates that it is possible to exploit the cache more efficiently for some problems. Matrix-matrix multiplication can be written in a block fashion the exploits cache reuse, yielding performances on the SMP system, far in excess of the theoretical peak speed of the vector system.

Sparse matrix vector multiplication illustrates a type of problem that hurts cache-based architectures particularly badly. Interestingly, the NEC's memory banked architecture also did not perform that well, leading to a vector advantage only at small vector lengths.

The heat equation example again shows the effect of thread creation overhead, and the SMP system only becomes competitive for problems sizes exceeding 500×500.

The $n-$body force calculation is interesting, as the computational cost grows as the square of the storage, so typically this type of problem easily fits into cache on SMP based systems. For this problem, the SMP computer easily outperforms the vector computer at all reasonable problem sizes, however care must be taken when parallelising the code to use thread private update variables to avoid different CPU caches from interfering with each other. For problem sizes greater than ≈ 1000 particles, a slightly different algorithm called *neighbourlists* is employed. Recent work done in my group indicates the neighbourlist algorithms can be effectively scaled on SMP computers, but the difference between vector and SMP systems may not be quite as marked.

The Mandelbrot problem, as mentioned before, is an example of a poorly load-balanced problem. The traditional approach to this type of problem is to code a *master-slave* type solution, using a message passing environment like MPI. OpenMP offers the possibility of applying this sort this sort of algorithm to ordinary do loops, using the `SCHEDULE(DYNAMIC)` workshare construct. As can be seen from the results, it performs very well. Considerable effort was lavished on attempting to optimise this problem for the NEC vector processor. In the end, the algorithm that performed best (but still badly), computed every point on the grid out to `maxiter`, throwing away the computations not needed. As this resulted in 97.3% of computations being wasted, a processing rate of 5.1 GFlops translated into only 14MFlops of useful work. One would have thought that applying a gather on the inner loop, followed by a scatter of the results should have improved the performance, but in this case the overheads introduced proved too much.

[5] During the course of preparing this paper, an upgrade to the cross-compiler used for the NEC computer improved the maximum speed achieved on this problem to near the machine's theoretical peak. Previously, the machine was topping out at about half the theoretical peak speed.

This benchmark suite doesn't aim to be comprehensive. A number of users might be interested to see how *Fast Fourier Transforms* fare (FFT's have been known to cause problems with parallel computers), and how eigenvalue calculations might perform (one suspects these might do well on SMPs). This is an exercise for the future.

4 Acknowledgements

I would like to thank Paul Ryan from NEC for his keen critical eye on my methods.