OpenMP in the Field: Anecdotes from Practice

Russell K. Standish[1,2], Clinton Chee[2], and Nils Smeds[3]

[1] School of Mathematics
[2] High Performance Computing Support Unit
University of New South Wales,Sydney, 2052,Australia
R.Standish@unsw.edu.au, chee@hpc.unsw.edu.au
http://www.hpc.unsw.edu.au
[3] Center for Parallel Computers, Kungl Tekniska Högskolan, 10044 Stockholm, Sweden
smeds@pdc.kth.se, http://www.pdc.kth.se/

Abstract. The High Performance Computing Support Unit at UNSW has a mission to support and encourage the scaling of computationally intensive applications from existing desktop implementations. OpenMP is a good match for this task. This paper reports on several projects in which OpenMP was used to parallelise an application, sometimes successfully, sometimes not so. The interest in these cases is that they are not the usual run of the mill applications that can be parallelised by simply adding a few OpenMP compiler directives, but required some lateral thinking.

1 Introduction

The High Performance Computing Support Unit at UNSW has a mission to support and encourage the scaling of computationally intensive applications from existing desktop implementations. Whilst some of our applications require the extreme performance that can only be achieved through distributed memory programming with MPI[7], for the most part our applications just need to run up to an order of magnitude faster than the desktop implementation. For this reason, we will often employ OpenMP[2] implementations with limits to scalability, but is an order of magnitude or so easier to program than an MPI implementation.

The HPCSU has a long history with OpenMP, in that an SMP parallel programming course developed during 1998 used the just released Fortran specification as a *lingua franca* to teach SMP programming. A table of equivalent compiler directives was provided for students to be able to program on their platform of choice, which at that stage did not support OpenMP. Nowadays, OpenMP support is ubiquitous, and this is no longer necessary. The course materials and tutorial examples are available from http://nswcpc.pvl.edu.au/SMP-course.html. It should be noted that this is not a self-study course, and needs to be presented by an experienced practitioner of HPC techniques.

The HPCSU supports customers of the *Australian Centre for Advanced Computing and Communications* (ac3), which has a 64 processor SGI Origin, a 68 processor IBM SP2 and a 2 processor NEC. As well as this, ac3 users may also use the *Australian Partnership for Advanced Computing*'s Compaq SC system. Both the SP2 and the SC are cluster systems based around quad processor nodes, and are generally not targets for OpenMP optimisation. An exception to this might occur if the application does not scale past 3-4 threads, in which case the Compaq SC with it's fast individual processors is an attractive option for running an OpenMP application within a node.

Since we have SGI Irix systems, we use an all in one development tool called ProDev Workshop. This is a GUI tool for debugging, profiling and analysing the results of parallelisation, included as part of the standard Irix development environment. We trialled a copy of KAPPro toolset, which is probably its main competitor, but ProDev Workshop appeared to provide superior information, and didn't require us to pay an expensive software license.

The 5 case studies presented here are not the usual sort of case studies one sees on OpenMP. After all, OpenMP's biggest attraction is to gain the advantages of parallel processing at minimal programming effort. These case studies involved considerable intellectual effort, and hopefully the lessons learnt are of interest to OpenMP practitioners.

2 NMG — Spectral Magnetohydrodynamics Code

The NMG code was originally written for the CM5 by Olga Podvigina. It solves 3D Navier-Stokes equations for an electrically conducting fluid. Similar work modeling neutral fluids was published in [5]. It is a spectral code, computing a quantity like $\mathbf{v}\times\nabla\times\mathbf{v}$. This can be expressed using the Fourier transform $\tilde{\ }$ as $\mathbf{v}\times\widetilde{i\mathbf{k}\times\tilde{\mathbf{v}}}$. So we have 3 3D FFTs, followed by a cross product, followed by 3 inverse 3D FFTs followed by another cross product.

Initially the code was run on the 128 node CM5 "colossus", which was still in operation at the time at the University of Adelaide, to obtain a baseline result for the porting and optimisation efforts. A 64^3 element volume was simulated for 50 timesteps, which took 230s to run on the CM5. This corresponds to about 130MFlops of sustained performance.

The code was written in CMFortran, a forerunner of the HPF language. Being already written in a data parallel fashion, it seemed that this should be an easy code to parallelise with OpenMP. The first task was to port the FFT routines to available FFT libraries. The following libraries were used:

SGI Complib. The original mathematics library available for Irix. A parallel implementation using Irix sproc() is available.

SGI Cray Scientific Library (SCSL). A library available for Irix, originally sourced from Cray. A parallel implementation using Irix sproc() is available.

FFTW. the *Fastest Fourier Transform in the West*[4], an open source library available from http://www.fftw.org/. A parallel implementation using POSIX threads is available.

IBM ESSL. Available for IBM SP systems. Threaded versions available.

Compaq DXML. Available for Compaq systems. Threaded versions available.
JMFFT. Open source FFT library suitable for vector computers, available from http://www.idris.fr/data/publications/JMFFT[3]

The CM5 Scientific Subroutine Library (CMSSL) provided FFT routines that perform an m-dimensional FFT on n-dimensional data. Ranks and dimensions of each array can be extracted from the arguments passed, and an additional rank 1 array specifies which dimensions are to be transformed. In NMG's case, a 3D FFT is performed on 4D data (since each field has 3 components). In converting the code to using the 3D FFT routines in the above libraries, array sections need to be created:

```
  do i=1,3
    call zfft3d(-1,n,n,n,fff(i,:,:,:),n,n,coef)
  enddo
```

Creating these array sections is actually a quite significant component of the total runtime. One of the strategies employed was to have separate variables for each component of the field (eg fff1, fff2 and fff3), and unroll the above loop.

The cross products were originally written out in full using as:

```
h(1,:,:,:)=ider(2,:,:,:)*vf(3,:,:,:)-ider(3,:,:,:)*vf(2,:,:,:)
h(2,:,:,:)=ider(3,:,:,:)*vf(1,:,:,:)-ider(1,:,:,:)*vf(3,:,:,:)
h(3,:,:,:)=ider(1,:,:,:)*vf(2,:,:,:)-ider(2,:,:,:)*vf(1,:,:,:)
```

Since none of these arrays fitted into cache, and there is no cache reuse, performance of this loop limps along at the speed of the memory subsystem. A slight improvement can be had by performing loop fusion:

```
do k=1,n
  do j=1,n
    do i=1,n
      h(1,i,j,k)=ider(2,i,j,k)*vf(3,i,j,k)-ider(3,i,j,k)*vf(2,i,j,k)
      h(2,i,j,k)=ider(3,i,j,k)*vf(1,i,j,k)-ider(1,i,j,k)*vf(3,i,j,k)
      h(3,i,j,k)=ider(1,i,j,k)*vf(2,i,j,k)-ider(2,i,j,k)*vf(1,i,j,k)
    ...
```

This at least allows some cache reuse. Another improvement comes from replacing ider by its value computed on the spot:

```
h(1,i,j,k)=(0,j)*vf(3,i,j,k)-(0,k)*vf(2,i,j,k)
```

or even extracting real and imaginary components of vf.

None of the FFT library routines would execute in parallel if called from within a parallel region, so this required two parallel regions per timestep. What is even worse is that on Irix systems, POSIX threads is incompatible with the SGI sproc() call used by the OpenMP compilers. As a consequence, FFTW was not useful for parallel work on Irix.

An optimisation which we didn't end up trying due to limited time is to write our own 3D (or 4D even) specialised FFT routine, composed of 1D sequential

FFT routines (from any library), and parallelised using OpenMP, so it could be called from a parallel region. Even so, timing results indicate that the cross product code is actually the bottleneck, executing at a small fraction of peak speed.

As a result we turned to using the NEC SX5 computer, which has a superior memory subsystem. We found an FFT library written by France's *Institut du Développement et des Ressources en Informatique Scientifique* (Institute for Development and Resources in Computational Science) that gave good performance on the SX5. Profiling the code on the SX5 indicated that the FFT is the bottleneck on this system. Execution time results of the sample problem for the different systems are given in the following table:

Machine	No. CPUs	Time (s)
CM5	128	230
SGI Power Challenge (195MHz)	4	188
SGI Origin (400MHz)	1	190
Intel PIII (600MHz)	1	232
NEC SX5	1	42

One thing that is interesting to note is that the newer Origin system does not run any faster in parallel, and is no faster than the older Power Challenge system whose performance peaks at 4 processors. It is expected that this is because the code is ultimately limited by the performance of the memory subsystem. It is also interesting to note a year 2000 vintage PC performing as well as the original CM5.

In conclusion, even though efficient shared memory versions of FFT routines exist, their use is limited by the performance of the memory subsystem in existing SMP implementations.

3 Force Calculations in N-Body Problems

N-body simulations are dominated by the force calculation, which scales as N^2, where the rest of the computation, and memory usage scales linearly with N. One of the example problems developed for the SMP programming course features implementing a 2D N-body force calculation for an inverse square force (like gravity):

$$\mathbf{F}_j = \sum_{k\neq j} \frac{\mathbf{x}_j - \mathbf{x}_k}{|\mathbf{x}_j - \mathbf{x}_k|^3} \tag{1}$$

Fortran code for computing eq (1) is shown below:.

```
do j=2,N
   do k=1,j-1
      rx = x(j)-x(k); ry = y(j)-y(k)
      r=sqrt(rx*rx + ry*ry); ir3 = 1/(r*r*r)
      Fx(j) = Fx(j) + rx * ir3
      Fy(j) = Fy(j) + ry * ir3
      Fx(k) = Fx(k) - rx * ir3
      Fy(k) = Fy(k) - rx * ir3
   enddo
enddo
```

On traditional vector supercomputers, the inner loop can be effectively vectorised (The carried dependency on $\mathbf{F}_j$ can be simply resolved as a sum reduction, and usually compilers will do this automatically). However, the operation density of the inner loop is too low on SMP systems for practical values of N to overcome the thread creation overhead, and the flushing of the caches. So parallelising the outer loop becomes mandatory.

The first problem is to break the carried dependency on $\mathbf{F}_k$. This can be achieved by placing the last two assignment statements within a `CRITICAL` region, however the thread lock overhead absolutely kills any performance gain from parallelism. The answer to the problem comes from realising that total force computation is decomposable into independent pieces that can be summed up to give the final force. So we need to introduce a thread private force variable, which will remain in a processor's cache (bear in mind the linear dependence on N of memory requirements). Once the force has been computed between pairs of particles, for which one of the pair is located on the processor, the result can be summed into the shared force variable, within a `CRITICAL` section, as per the code snippet below.

Only one other feature needs commenting. The default STATIC scheduling of the DO workshare construct will lead to load imbalance due to the triangular work distribution. However, as is reasonably well known, OpenMP provides the GUIDED schedule which is optimised for this situation.

Performance of this particular piece of code was reported elsewhere[8], exceeding 16GFlops for 20,000 particles on a 40 processor SGI Origin. This exceeds 50% of peak floating point performance (each CPU has a peak of 800MFlops).

Exciting though this result is, it is worth noting that for more than a few thousand particles, one would employ a neighbourlist algorithm if the force was short range, or an Ewald summation for long range forces [1]. This changes the overall algorithm complexity to something more like $o(N \log N)$, and probably reduces the amount of gain from parallelism.

This technique was employed with a real world 3D *Discrete Particle* code used for Material Science research. These particles had van der Waals attraction, and the algorithm employed a neighbourlist to imporve performance. We achieved reasonable performance scaling to about 8 processors (about 4 times) for 250 particles.

```
C$OMP PARALLEL private(i,j,k,rx,ry,r,ir3,Fxk,Fyk) shared(x,y,Fx,Fy)
         do j=1,N
            Fxk(j)=0
            ...
C$OMP DO schedule(guided)
         do j=1,N
            Fx(j)=0
            Fy(j)=0
            do k=1,j-1
               ...
               Fxk(j) = Fxk(j) + rx * ir3
               ...
C$OMP CRITICAL
         do j=1,N
            Fx(j) = Fx(j) + Fxk(j)
            ...
C$OMP END CRITICAL
C$OMP END PARALLEL
```

4 HKondo — Lattice Gauge Theory Calculations

HKondo was a code presented by one of our users (who shall remain nameless). It performed some kind of lattice gauge calculation, which is not important for the present purposes. What is important is that the user desired to speed the program up, and that the code consisted of a monolithic block of 670 lines of Fortran 66 dialect and coding style. No indentation, no comments and a large number of goto statements. A large number of optimisations had already been applied to the code, which in part explained the codes crypticity. There was little to be achieved by trying standard sequential optimisation techniques.

An initial profile of the code indicated that most time was spent within two inner loops, however the loop count on these were depressingly small. The next loop out was then the loop to attempt parallelisation. However this loop of about 200 lines has dozens of array references and assorted other control logic. The compiler's automatic paralleliser was of no assistance here, the code was simply too complex.

In order to determine the data flow through this block, an unusual trick was resorted to. The entire loop was moved into its own subroutine, and used the `IMPLICIT NONE` directive. The first pass of the compiler gave me a list of variables that needed to be declared. The compiler listing were pasted into the code, and reformatted as declarations. Next, the parameter list for the subroutine needed to be determined. This involved a manual check as to whether a variable name had been referenced outside the subroutine using the text editor's search feature. It soon became clear whether a variable was imported into, exported from, or simply ignored by the code block. The conclusions could be tested by using Fortran's `INTENT` statement, and leaving the variable out of the parameter list. If a conclusion was wrong, the program would break. This led to a small list of 35 variables that needed to be declared as SHARED, with the remainder declared as thread private. This was the most labour intensive part of the whole process, and took perhaps about 2 hours to perform.

After this was done, the compiler's paralleliser indicated 3 carried dependencies. The first two were of the nature of tracking maximum values, eg

```
      if (nst2.gt.nlmax) nlmax=nst2
```

which can be easily handled by placing these in a critical region. The third carried dependency was a little more difficult:

```
       afr(1,k1,k2,k3)=afr(1,k1,k2,k3)+ax
```

These might be handled via a critical section, but unfortunately were inside the innermost loop, and consequently the most expensive operation. The critical section idea lost in lock overheads, all the gains made by parallelisation.

A closer inspection of the code revealed that k1 was constant within the inner loop, and depended in a deterministic way on the loop index of the outer loop:

```
      do ist=1,nst
       k1=v(ist); nt=...
       do j=1,nt
         k2=...; k3=...; ax=...
         afr(1,k1,k2,k3)=afr(1,k1,k2,k3)+ax
         ...
```

What needed to be done, therefore, was to gather all loop iterations with the same value of k1, and execute this on the same processor. Then the loop dependence carried by afr is simply broken without needing a critical section. To do this required a small precomputation:

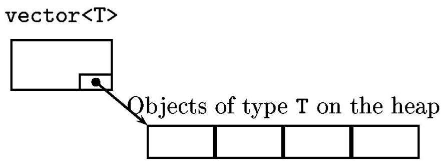
```
      integer istvec(maxk1,nst), nist(maxk1)
      do ist=1,nst
        k1=v(ist)
        nist(k1)=nist(k1)+1
        istvec(k1,nist(k1))=ist
      enddo
C$OMP PARALLEL DO SCHEDULE(DYNAMIC)
      do k1=1,maxk1
        do i=1,nist(k1)
          ist=istvec(k1,i); nt=...
          do j=1,nt
            k2=...; k3=...; ax=...
            afr(1,k2,k3,k1)=afr(1,k2,k3,k1)+ax
          ...
```

Note that the indices of `afr` needed to be reordered to make the memory references more cache friendly. DYNAMIC scheduling was used to improve load balancing. The resultant code achieved a speedup of approximately 3 on 4 threads, which was about the limit of its scalability. The quad processor APAC nodes proved the most effective platform in this case.

5 C++ Implementation of Ray Tracing

As a student project for an honours course taught by one of us, a student Tom Edlund decided to implement a parallel Monte Carlo ray tracing algorithm, choosing C++ for its object oriented nature, and used the VTK toolkit[6] for handling scene geometries.

Whilst the exact details of the algorithm are not germane here, the main issues with OpenMP related to the use of standard containers, in particular `vector`. A `vector` is a dynamically resizable array of some type of object. Clearly, it must consist of some kind of header for managing the object, with the actual storage taking place on the heap:



Firstly, lets consider what happens when the vector is a shared object. Vectors are very flexible objects that can be built by adding objects to their end with `push_back()`, resized, or have elements deleted from with their range. Any of these operations can caused a new space to be allocated for the vector, and the contained objects copied into the new space. If another thread is trying to access the vector, even just read only access, while the vector is being copied to a new location, disaster strikes. Even though the Standard Template Library claims its containers to be thread safe, they can only be used safely if treated like static arrays — namely their size remains fixed throughout a parallel region, and they are either readonly, or written to from within a critical region.

Now consider thread private vectors. As soon as a vector is resized to contain some data, it must call `new` to obtain some memory. The problem is that `new` must update some shared internal heap management variables, as the heap is shared between all threads. This problem is fixed by providing a thread safe version of `new`, which is easy enough to do in C++:

```
void *operator new(size_t s)
{
#pragma omp critical
  return malloc(s);
}

void operator delete(void *p)
{
#pragma omp critical
  free(p);
}
```

Obviously these critical statements can be a serious performance bottleneck, but by judiciously using `vector`'s `reserve()` method, one can minimise the number of calls to `new` and `delete`. Alternatively, one could set up private heaps for each thread, and supply appropriate `new` and `delete` operators to manage them.

Tom found that even with these thread safe memory allocation operators, his code did not run correctly in parallel. In the end, he had to write his own threadsafe vector class, which did work. This could be simply a problem with the SGI implementation of the standard template library, as we did not try this out on any other C++ implementation.

6 Finite Difference Time Domain

The FDTD code uses a Finite Difference Time Domain technique to find the solution of a vector field problem.

The porting and optimisation work was performed on the IBM SP2, then laterported to the NEC SX5 vector machine. The same OpenMP parallel techniques used on the IBM SP2 version, was employed the dual processor SX5 using the native NEC SMP parallellising compiler directives.

The first of the two main sections to be parallelized is shown below; the second section being similar in structure to the first but with slightly more complex terms. For relatively simple constructs as above, the FORALL multi-loop structure is fairly efficient for it is an optimized command. Initial testing which replaced the FORALL with a triple do-loop, slowed down this section by 3.5 times.

```
      FORALL(I=0:IM,J=0:JM,K=0:KM)
Gxy(I,J,K) = Cay(J)*Gxy(I,J,K) - Cby(J)*
( Fzx(I,J+1,K) - Fzx(I,J,K) +  Fzy(I,J+1,K) - Fzy(I,J,K) )
      FORALL(I=0:IM,J=0:JM,K=0:KM)
Gxz(I,J,K) = Caz(K)*Gxz(I,J,K) + Cbz(K)*
( Fyx(I,J,K+1) - Fyx(I,J,K) +  Fyz(I,J,K+1) - Fyz(I,J,K) )
      FORALL(I=0:IM,J=0:JM,K=0:KM)
Gyx(I,J,K) = Cax(I)*Gyx(I,J,K) + Cbx(I)*
( Fzx(I+1,J,K) - Fzx(I,J,K) +  Fzy(I+1,J,K) - Fzy(I,J,K) )
      FORALL(I=0:IM,J=0:JM,K=0:KM)
Gyz(I,J,K) = Caz(K)*Gyz(I,J,K) - Cbz(K)*
( Fxy(I,J,K+1) - Fxy(I,J,K) +  Fxz(I,J,K+1) - Fxz(I,J,K) )
      FORALL(I=0:IM,J=0:JM,K=0:KM)
Gzx(I,J,K) = Cax(I)*Gzx(I,J,K) - Cbx(I)*
( Fyx(I+1,J,K) - Fyx(I,J,K) +  Fyz(I+1,J,K) - Fyz(I,J,K) )
      FORALL(I=0:IM,J=0:JM,K=0:KM)
Gzy(I,J,K) = Cay(J)*Gzy(I,J,K) + Cby(J)*
( Fxy(I,J+1,K) - Fxy(I,J,K) +  Fxz(I,J+1,K) - Fxz(I,J,K) )
```

Various features of the code snippet above indicate that each of the six loops are fairly independent and there is no data dependency. There is a possibility of cache reuse which can be accomplished by loop fusion. Looking at the big picture, it appears that this code would benefit from coarse-grain parallelism, rather than fine-grain / loop level parallelism.

Thus the major step to parallelize the code is to use the parallel sections feature of OpenMP. There are a number of ways in which to construct parallel sections over this code, some will be more efficient than others. In order to achieve high efficiency, the first criteria is maximizing cache reuse, and the second criteria is matching the number of parallel sections with a multiple of the number of processors available.

The optimized code is shown below here with a 2-2-2 configuration denoting 3 parallel sections with 2 equations each:

```
!$OMP PARALLEL SECTIONS
      FORALL(I=0:IM,J=0:JM,K=0:KM)
Gxy(I,J,K) = Cay(J)*Gxy(I,J,K) - Cby(J)*
( Fzx(I,J+1,K) - Fzx(I,J,K) +  Fzy(I,J+1,K) - Fzy(I,J,K) )
Gyx(I,J,K) = Cax(I)*Gyx(I,J,K) + Cbx(I)*
( Fzx(I+1,J,K) - Fzx(I,J,K) +  Fzy(I+1,J,K) - Fzy(I,J,K) )
      end FORALL
!$OMP SECTION
   ...
!$OMP SECTION
   ...
!$OMP END PARALLEL SECTIONS
```

The 2-2-2 configuration take advantage of the symmetry of the pairing of the 6 field equations — to allow some cache reuse. The resultant 3 parallel sections is also sufficient to be implemented on the IBM SP2 by using 1 node. Within this node, 4 processors are available in a shared memory configuration. An OpenMP thread will ideally be processed by one processor. In this case, 3 OpenMP threads are required - specified by the environment variable *OMP_NUM_THREADS=3*. In addition, experience has shown that the computing job runs better when all 4 processors of the node are reserved by the batch queue, as the 4th processor takes over some of the system processing time.

Config.	Loop Type	OMP thrd.	Real Time	User Time	Sys Time
2-2-2	FORALL	1	1m4.78s	1m3.23s	0m0.45s
2-2-2	FORALL	2	0m44.69s	1m6.22s	0m0.47s
2-2-2	FORALL	3	0m28.25s	1m11.64s	0m3.20s
2-2-2	DO	3	0m30.80s	1m19.27s	0m3.71s
1-1-1-1-1-1	DO	3	0m32.60s	1m22.41s	0m3.92s

The results when the remainder of the code was parallelized is shown in table above. The size of the matrices are IM=120, JM=80, KM=120 for 100 iterations.

There is a clear improvement as the number of OMP threads are increased from 1 to 3. As noted before, changing the FORALL loops into triple Do-loops slows down the code.

The code was finally ported over to the SX5 vector machine, and after some manual optimization, ran 10 times faster (using 2 threads) than on the SP2. However, this was not only due to the vectorization. In fact the same principles of parallel sections was applied using compiler directives (very similar to OpenMP) native to the SX compilers. Thus the techniques of OpenMP can sometimes be transferred successfully to other non-OpenMP compilers.

References

1. M. P. Allen and D. Tildesley. *Computer Simulation of Liquids.* Clarendon, Oxford, 1987.
2. Rohit Chandra. *Parallel Programming in OpenMP.* Morgan Kauffman, San Francisco, CA, 2001.
3. Jalel Chergui. Transformées de Fourier rapides monoprocesseur sur NEC SX-5. Technical report, CNRS/IDRIS, Bâtiment 506, BP 167 91403 Orsay cedex, France, 2000. http://www.idris.fr/data/publications/fft-SX5.pdf.
4. M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings ICASSP*, volume 3, pages 1381–1384, 1998. http://www.fftw.org/.
5. Olga Podvigina. Spatially-periodic steady solutions to the three-dimensional Navier-Stokes equation with the ABC force. *Physica D*, 128:250–272, 1999.
6. Will Schroeder, Ken Martin, and Bill Lorensen. *The visualization toolkit : an object-oriented approach to 3-D graphics.* Prentice Hall, Upper Saddle River, N.J., 1996.
7. Marc Snir et al. *MPI: the complete reference.* MIT Press, Cambridge, MA, 1996.
8. R.K. Standish. SMP vs Vector: a head-to-head comparison. In *Proceedings HPCAsia 2001.*, 2001. http://parallel.hpc.unsw.edu.au/rks/docs/ps/smp-vs-vector.ps.gz.