Mixed OpenMP and MPI for Parallel Fortran Applications

I.J. Bush, C.J. Noble and R.J. Allan
Daresbury Laboratory, Daresbury, Warrington WA4 4AD

Abstract

In the five years since the inception of the High Performance Computing Initiative (HPCI) in 1994, the support Centres at Daresbury, Edinburgh and Southampton (now Daresbury and Edinburgh working with the Manchester CSAR service) first ported and optimised codes, and more recently contributed to the longer-term development of new applications on MPP platforms. To develop optimal codes new parallel algorithms must be tailored to the principal target hardware, currently the Cray T3E-1200E at Manchester, but all codes provided by the Daresbury Centre are designed to be portable and use current standards such as MPI, OpenMP [1] and Fortran. With available hardware now embracing NUMA systems from, for instance, IBM, Compaq and SGI, we have begun to develop and optimise applications using a mixture of MPI and OpenMP.

OMP, MPI and Mixed Kernels on IBM Power3 Winterhawk-2

We first consider OpenMP, MPI and mixed OpenMP+MPI versions of kernel algorithms. These include hand-tuned multi-threaded DDOT and DGEMM. We then report first results of examples of full-scale applications which have been ported to NUMA systems. Results of future tests will be made available via the Web site *www.ukhec.ac.uk*.

Architecture

8 SMP nodes connected by IBM's TB3-MX switch. The Winterhawk-2 SMP nodes each comprise 4 IBM Power3 375 MHz processors. Each processor has a 64 kByte primary cache, and a 4 Mbyte secondary cache.

Dot product

A simple parallel dot product was coded. OMP directives were used to parallelise the dot product within an SMP node, whilst MPI is used between nodes to compute the final results using a sum reduction. A number of different scaling behaviours have been investigated with variations of the code fragment shownbelow.

Within one SMP node the scaling with the number of threads was tested by setting the AIX environment variable XLSMPOPTS=parthds=n. Vectors of length 10^6 were used, and each calculation was repeated 100 times. Performance increased from 140 Mflop/s to 211 Mflop/s using 4 threads, but decreased down to only 30 Mflop/s with 8 threads. No attempt was made to flush the cache between repetitions. The single node scaling with thread number is poor.

Running over more than one node the performance of the pure MPI code and a hybrid MPI/OMP code was compared. For the hybrid approach four threads per node were used, as this appears to give the best SMP single-node performance. The test was with vectors of length $3*10^6$, and each calculation was repeated 3000 times. With a single MPI process per node and 4 threads performance varied from 50 Mflop/s with only one node to 2.9 Gflop/s on 8 (32 threads). However with 4 MPI processes per node and no threads we achieve 195 Mflop/s on 1 node and 4.3 Gflop/s on 4 – some 4 times improvement overall.

The single node performance is much better at 10^6 than $3*10^6$. This is probably because at 10^6 the two vectors just fit in the 4 Mbyte secondary cache, while at $3*10^6$ they can not and have to be read in for each repeat.

Comments

a) Both scalings for multiple nodes were markedly super-linear in the original code tried. This was because for larger numbers of nodes the local length of the vector is short enough for cache residency. Repeating the operations inside the main loop for several different vectors meant that this effect could be avoided, as shown in the figure.

b) Pure MPI is *much* better than OMP+MPI for this case. This is even more significant on the SGI Origin2000 where very little performance im-

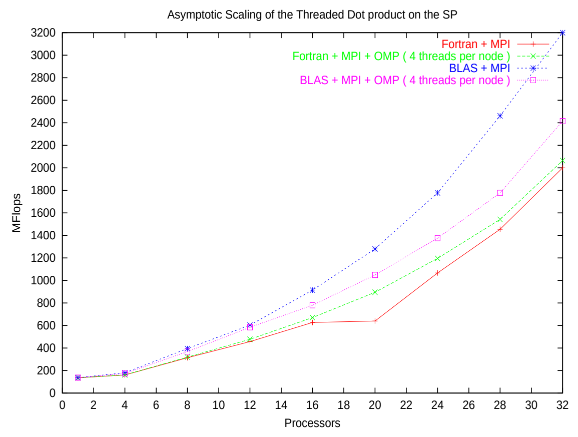

Figure 1: Dot Product

provement was seen with threads but the MPI code behaved predictably and gave speedup even on a multi-user machine.

c) A brief investigation of using the multi-threaded library DDOT function from ESSLSMP was carried out. This seemed to behave similarly to the simple Fortran loop.

Dot Product Code

```
PROGRAM test_omp_dot_product
! declarations
  ...
  CALL mpi_init(error)
  CALL mpi_comm_rank(mpi_comm_world, me, error)
  CALL mpi_comm_size(mpi_comm_world, my_mates, &
                     error)
  repeats = 10
  DO n = 50000, 2000000,12500
    local_n = n / my_mates
    threads = num_parthds()
! allocates
    ...
    CALL Random_number(a)
    CALL Random_number(b)
    CALL system_clock(count_start, count_rate)
    CALL cpu_time(start)
    DO  i = 1, repeats
      local_result = omp_dot_product(a, b)
      Call mpi_allreduce(local_result, result, &
                        1, mpi_double_precision, &
               mpi_sum, mpi_comm_world, error)
! use many vectors to avoid cache residency
    ...
    END DO
    CALL cpu_time(finish)
    CALL system_clock(count_finish, count_rate)
! write out results
    ...
    repeats = repeats / 8
! de-allocates
  ...
  END DO
  CALL mpi_finalize(error)
END PROGRAM test_omp_dot_product

PURE FUNCTION omp_dot_product(a, b)
! declarations
  ...
  n = Size(a)
  result = 0.0_float
!$OMP Parallel default(none), &
!$OMP Shared(a, b, n),        &
!$OMP Private(i),             &
!$OMP Reduction(+:result),   &
!$OMP Do
  DO i = 1, n
    result = result + a(i) * b(i)
  END DO
!$OMP End Do
!$OMP End Parallel
  omp_dot_product = result
END FUNCTION omp_dot_product
```

A dot product may not be a very good example of where OMP should be used as the loop distributed to the threads has relatively little work in it when compared to, say, a matrix multiply.

Matrix Multiply using Cannon's Algorithm

A blocked multithreaded matrix multiply routine for $C = \alpha * A * B + \beta * C$, i.e. no transposes, was implemented based on Cannon's algorithm [3]. It uses a dot product formulation. By using OpenMP directives it is possible to organise the code so that the various threads update independent blocks of $C(i, j)$, while blocks of A and B are kept in cache and reused as much as possible. To achieve this, the blocks of A and B are stored in thread private temporary arrays temp_B and temp_A. The latter is also transposed to ensure good strides in the inner loops. This process is either load or store bound, though loop unrolling can help. The "store bound" method was used in the final version as this appears to be faster on Power3 CPUs.

The code was optimised for good performance on the IBM Power3 architecture. For other architectures the following may have to be changed:

i) the blocking factors used. The cache size should be roughly $3*n*n$ if n is the blocking factor, where cache refers to whichever level of cache it is better to use on the target architecture;

ii) 4x4 unrolling has been used in the main multiply kernel loops. Consideration should be given to reducing this if the target architecture has less than 32 floating point registers, otherwise register spilling may occur.

Through careful use of cache the serial code achieved a performance for large matrices of nearly 1 Gflop/s, which is a 2/3 of the peak processor speed of 1,5 Gflop/s (using the 2 fused multiply-add pipes on the Power3). By varying the number of threads per process and number of MPI processes per node it was possible to compare flat MPI and multi-threaded versions of this code. Results are shown in Figure 2. It is clear from this figure that both pure MPI and threaded implementations are giving good speedup with the latter achieving 3.9 Gflop/s on four threads. The MPI implementation was optimised using asynchronous message passing, but only achieved 3.4 Gflop/s at best.

Finally using 4 threads on 4 MPI processes spread across 4 nodes we were able to achieve a peak of 14 Gflop/s, whereas the flat MPI version with 16 processes was only able to give 8.8 Gflop/s at best. Note that we used 16 nodes for this test to avoid any problems of memory bandwidth.

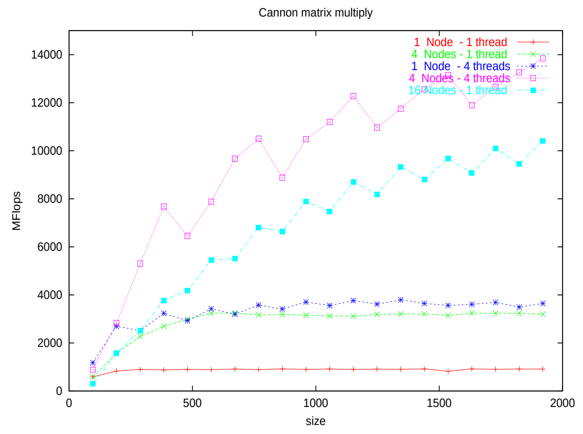

Figure 2: Cannon's Algorithm

ANGUS – a Regular-grid Engineering Code

ANGUS performs direct numerical simulations (DNS) of turbulent pre-mixed combustion to generate statistical data in support of combustion modelling. The equations to be solved are the Navier-Stokes equations for fluid flow, augmented by two additional equations each describing the transport of a single scalar variable, together specifying the thermochemical state of the system in the presence of differential diffusion effects. In total there are six partial differential equations to be solved. Discretisation of the equations is carried out using standard second-order central differences on a three-dimensional grid. The velocity nodes are located at the face-centres of each cell, giving a staggered-grid arrangement that conserves kinetic energy as well as mass and momentum. The MPI part of the code is parallelised using regular domain decomposition with "halo" data added to the domains to cache data from neighbouring domains required for the central difference updates.

Several different pressure solvers are available in the code. We have concentrated effort initially on one that utilises a pre-conditioned Conjugate Gradient (CG) method, with level-1 BLAS (DSCAL, DAXPY and DDOT) used heavily throughout. Alternative pressure solvers use a multigrid method or an FFT method based on a special approach for Poisson solvers [4].

The computational work is roughly proportional to the number of grid points n^3, and n=96, a rather small case, was used for the tests reported. In the CG solver the matrix to be solved is of size n^3*7 for a first-order FD or n^3*13 for a second order FD. The number of iterations depends on the size of the matrix n, typically around 280 with n=96. Parallelisation using OpenMP was introduced by re-writing the BLAS as explicit merged Fortran loops and introducing PARALLEL DO directives on the outer loop. Comparing the Fortran and library versions indicated that, in fact, performance of the Fortran was somewhat better on most machines. Using the threaded ESSLSMP library on the IBM with the original level-1 BLAS code gave only a factor of 2 speedup going from 1 to 4 threads on a single Winterhawk-2 node.

Pre-conditioning can be applied using a special case of an ILU pre-conditioner which takes account of the known structure of the matrix (-1,-1,6,-1,-1) in bands. This is applied to each block, i.e. the data belonging to each MPI process. The number of iterations is greatly reduced if pre-conditioning is used, in the case of n=96 it is halved, but the ILU step takes a significant time and we found that running with 4 MPI processes on one Winterhawk node there was no significant gain. A further complication is that the code for the preconditioner contains loop iteration dependencies as it is applying the 3D point-wise update and there is no simple way to parallelise it using OpenMP. The IBM Red Book [2] does indicate how a similar dependency in a 1-dimensional loop might be re-written by analysing the recursion algebra, but this is more complicated in 3D and would probably introduce a significantly larger operation count than in the present code.

Preconditioner Code Fragment

```
 FUNCTION precon(nx,ny,nz,i5,i6,i3,i4,i1,i2, &
                d,dinv,res,pres,nop)
! declarations
 ...
! non-local part contains recursion
 DO j=i3, i4
   DO i=i5, i6
     DO k=i1, i2
       term=pres(i-1,j,k)+pres(i,j-1,k)
       pres(i,j,k)=(res(i,j,k)+term)*d(i,j,k)
       pres(i,j,k)=pres(i,j,k)+pres(i,j,k-1)* &
                   d(i,j,k)
     END DO
   END DO
 END DO
! local part can be parallelised
!$OMP parallel shared(pres,dinv) private(k,j,i)
!$OMP do
 DO k=i1, i2
   DO j=i3, i4
     DO i=i5, i6
       pres(i,j,k)=pres(i,j,k)*dinv(i,j,k)
     END DO
   END DO
 END DO
!$OMP end do
!$OMP end parallel
! non local inverse part
 ...
 END
```

Total execution times for the "optimised" code as a function of the number of nodes, processes per node and threads per process are reported below without pre-conditioning. The time taken by the OMP-parallel and BLAS sections of the CG pressure solver are also reported per iteration. Only two iterations were performed and remaining time is spent in copying arrays, setting up the problem and message passing between halo regions.

As can be seen from the Table 1, the best performance is obtained by using a larger number of MPI processes if this can be done explicitly. For a given number of MPI processes increasing the number of threads per process does however have a benefit.

Table 1: ANGUS performance on Winterhawk-2

a	b	c	d	e	f	g
4	2	2	1	6.0	0.56	20.9
4	4	1	1	4.2	0.53	16.6
8	2	2	2	2.5	0.74	15.1
8	4	1	2	2.0	0.52	12.8
16	4	1	4	1.1	0.5	12.3
8	2	4	1	1.85	0.25	9.1
8	4	2	1	1.6	0.32	9.0
16	4	2	2	0.95	0.38	8.0
16	4	4	1	0.64	0.1	5.0

a – total parallelism
b – number of nodes
c – number of MPI tasks per node
d – number of threads per task
e – time in OMP regions [s]
f – time in remaining BLAS [s]
g – total elapsed job time

A Case Study in Atomic and Molecular Collision Physics

We are in the process of developing a number of full-scale scientific application codes using both MPI and OpenMP. We describe one such case in some detail to illustrate the advantage of this mixed-mode style and the importance of selecting the computational algorithm to obtain the best parallel decomposition.

Collisions between electrons, photons or atoms and atomic or molecular targets are central, for example, in the physics and chemistry of the upper atmosphere and in the interaction of lasers with matter. The investigation of these problems, as well as many others, requires the solution of large coupled sets of radial Schrödinger equations of the form

$$\left(\frac{d^2}{dr^2} + \mathbf{W}(r)\right)\mathbf{F}(r) = 0 \qquad (1)$$

where the wave vector matrix, $\mathbf{W}(r)$ is related to the local potential coupling matrix $\mathbf{V}$ by

$$\mathbf{W}(r) = \mathbf{k}^2 - \mathbf{V}(r). \qquad (2)$$

The diagonal matrix of channel energies, $\mathbf{k}^2$, depends on the scattering energy E. The dimensions of these matrices range from a few hundred to a few thousand and the coupled equations may have to be solved for many values of the scattering energy E. Solutions are required to be regular at the origin

$$\mathbf{F}(r) \xrightarrow[r\to 0]{} 0 \qquad (3)$$

or are matched at some inner boundary, $r = r_a$, to the solutions of a separate boundary condition model. Equation (1) is integrated radially outwards to a point $r = r_b$ where the solution is matched to some asymptotic boundary condition. Each of the coupled equations represents one asymptotic channel, i, and may be either open or closed according to whether $k_i^2 > 0$ or $k_i^2 < 0$. Closed channels are introduced in order to describe the distortion of the target by the projectile. In this situation very stable integration methods such as the R-Matrix or log-derivative (LD) propagator methods are particularly efficient.

Propagator methods solve the equations by dividing the the radial interval $[r_a, r_b]$ into subintervals, $[r_i, r_{i+1}]$, for $i = 1, \ldots n$ where $r_1 = r_a$ and $r_n = r_b$. Each of these subintervals, or sectors, are chosen to be sufficiently small that the Greens function corresponding to equation (1) may be determined within the sector either by introducing a basis set expansion or by approximating the potential $\mathbf{V}(r)$. Given the Greens function $\mathcal{Y}$, the logarithmic derivative of the solution at the inner boundary of a sector i,

$$\mathbf{Y}(r_i) \equiv \mathbf{F}'(r_i)\mathbf{F}^{-1}(r_i) \qquad (4)$$

(the prime denotes differentiation with respect to r) may be propagated to the outer boundary r_{i+1} by the equation

$$\mathbf{Y}(r_{i+1}) = \mathcal{Y}_4 - \mathcal{Y}_3 \left[\mathcal{Y}_1 + \mathbf{Y}(r_i)\right]^{-1} \mathcal{Y}_2 \qquad (5)$$

The diagonal matrices, $\mathcal{Y}_i, i = 1 \ldots 4$ are the sector Greens function evaluated on the boundaries. The recursive application of equation (5) allows the solution corresponding to a particular boundary condition at $r = r_a$ to be obtained at $r = r_b$.

We have studied the parallelisation of the propagation method [5] on distributed memory parallel computers using MPI. When electrons are scattered by atomic ions very large numbers of scattering energies have to be considered in order to map out the rapidly varying structures associated with the formation of Rydberg resonances below each scattering threshold. In this case an effective strategy is to set up systolic pipes of processors in which each processor node performs a single sector calculation. Results for each scattering energy are propagated to successively larger radial distances as they are passed along the pipeline. This approach is very efficient as communication costs may be hidden and for sufficiently large numbers of scattering energies the start-up and wind-down costs are negligible. Multiple processor pipes may be set up and controlled to ensure load-balancing and the efficient use of large numbers of processors.

The program currently being developed is required for studies of the multiphoton ionisation of atoms by intense laser beams and for the investigation of harmonic generation using Floquet methods. The LD propagator method employed assumes a linear reference potential so the sector Greens functions are represented by Airy functions [6]. In addition, the appropriate outer boundary conditions are given by the Siegert condition

$$\mathbf{F}(r) \xrightarrow[r\to\infty]{} \exp(i\mathbf{k}r) \qquad (6)$$

which corresponds to all outgoing waves. The energy in this application is complex-valued (a quasi-energy) and is normally determined iteratively. The first stage of the calculation is to determine the sector sizes and this involves the diagonalisation of the wave vector matrix at selected points within each sector. This information is saved and is used for subsequent calculations at each quasi-energy E. The core of the calculation reduces to two steps which must be repeated for each quasi-energy E and for each sector i. These are: (1) to transform the LD matrix into a local matrix representation which diagonalises the potential in sector i; and (2) propagate the solution across the sector using equation (5). These operations require two matrix-matrix multiplies, one matrix-vector multiply and the solution of one set of linear equations. The fact that the kernel of the method is expressed in terms of standard matrix operations is a key advantage of the approach. The algorithm is ideally suited to the use of OpenMP on an SMP mode and is able to take full advantage of the optimised libraries and the global memory.

In the following table we show sample timings for the computation of a problem involving 300 scattering channels in which solutions are integrated from 10 to 100 Bohr radii. The timings were carried out on a Winterhawk-2 node and show the scaling as the number of threads is increased. The use of tuned BLAS and linear algebra from the ESSLSMP library provides good scalability and high efficiency. Further, but less significant speedups are obtained in other parts of the code by the use of OpenMP directives.

In this scheme larger calculations are easily accommodated by the construction of systolic pipelines of two or more SMP nodes. The approach used previously for electron scattering may be taken over with

Table 2: Propagator performance on Winterhawk-2

Number of threads	Set up (secs)	Propagation (secs)
1	15.39	55.35
2	10.14	30.97
3	8.07	22.56
4	8.20	19.59

only minor amendments.

Further parallelisation requires an algorithmic change. The iterative method commonly used to determine the quasi-energy is inherently serial. However it is possible to determine a number of quasi-energies simultaneously using a complex-energy contour integration technique [7]. Each integrand point of the contour integral corresponds essentially to a single instance of the calculation we have described above. The evaluation of the contour integral is simply accomplished by MPI task-farming the integrand calculations to the SMP nodes.

Although the full program package is incomplete we have demonstrated each of the major elements. The linear reference potential LD propagation method, even for the case of complex energies, may be readily and efficiently parallelised using OpenMP on each processor node. The overall scheme, using both OpenMP and MPI, is flexible and capable of making efficient use of large processor arrays.

Conclusions

Whilst it has been demonstrated that mixed-mode MPI/ OpenMP codes can give significant performance on kernel algorithms such as Cannon's matrix multiply, there can be a significant amount of work involved to achieve this. On real codes using a flat MPI implementation may even give better performance. The reason is partly in the overhead of starting teams of threads at the loop level – IBM recommend keeping a team active throughout the whole run (SPMD program), but this requires a different approach to that usually adopted on shared-memory systems.

There may also be significant cache effects with cache lines being invalidated if different processors (threads) access neighbouring array elements. Bandwidth for several processors to access memory within a node is also important. Finally Amdahl's Law applies, and devising a thread-parallel implementation for all sections of a large code is difficult. We have so far had no success in using the automatic compiler options to do this. Nevertheless combining MPI and OpenMP (or using thread-parallel libraries) can give the flexibility needed to parallelise a complex code and yield good performance.

Finally we note that we have not carried out rigorous tests on all available platforms, e.g. Compaq, IBM, SGI, SUN and HP. Compilers and library software are evolving rapidly and we shall re-visit mixed-mode programming in the future.

Acknowledgements

The work presented was funded by EPSRC partly through a grants GR/K82635 to the HPCI Centre at Daresbury Laboratory and GR/N09688 to the UKHEC Collaboration and partly through its Service Level Agreement with the CSE Department.

Bibliography

[1] R.J. Allan and C.J. Müller *Shared-Memory Programming Paradigms* (CLRC Daresbury Laboratory, 1999)

[2] S. Andersson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh and J. Tuccillo *Power3 Introduction and Tuning Guide* IBM Red Book order number SG24-5155-00 (IBM, October 1998) See *www.redbooks.ibm.com*

[3] Cannon. Ph.D. Thesis (Montana State University, 1969)

[4] J. Demmel *Solving the Discrete Poisson Equation using Jacobi, SOR, Conjugate Gradients and the FFT* U.C. Berkeley Lecture Notes CS267. See *www.cs.berkeley.edu/ demmel/cs267*

[5] A.G. Sunderland, J.W. Heggarty, C.J. Noble and N.S. Scott, Comp. Phys. Comm. 114 (1998) 183

[6] M.H. Alexander and D.E. Manolopoulos, J. Chem. Phys. 86 (1987) 2044

[7] C.J. Noble, M. Dörr and P.G. Burke, J. Phys. B:At. Mol. Opt. Phys. 26 (1993) 2983