Case Study – Optimising Combustion Code SoFTaR

Ning Li
HECToR CSE Team

The code "SoFTaR" (Simulations of Flames: Turbulence and Reaction) was based on the direct numerical simulation (DNS) code for jet flow turbulence and combustion simulations developed at Brunel University, which is in the process of being linked with the database of flamelet-generated manifolds for combustion chemistry developed at TU/e (Eindhoven University of Technology, The Netherlands) for DNS of flames with realistic chemistry. The SoFTaR code is used for the ongoing research of an EPSRC grant EP/G062714/1 "Clean Coal Combustion: Burning Issues of Syngas Burning" (PI: Dr Xi Jiang; PDRA: Dr George Siamas).

The research group contacted the CSE team hoping to have the code optimised before running large-scale simulations on HECToR. A site visit by the CSE team was arranged and hands-on training was given to the research staffs involved. The CSE team subsequently worked on the application code directly. Performance bottlenecks were identified and new communication routines were implemented by the CSE team, making the code **5** times faster in one realistic test case. Other suggestions regarding proper parallel I/O were given. These optimisation details are documented in this report.

Optimising Communication Code

The application is based on structured-mesh finite-difference method using a form of compact scheme. Naturally, one-dimensional decomposition is used together with global data transposition strategy so that expensive calculations can be done using established fast algorithms in local memory. Unfortunately the communication routines in this code is not in a great shape for modern supercomputers like HECToR, as demonstrated by the following Cray PAT report.

Time %	Time	Imb. Time	Imb. Time %	Calls	Group Function PE='HIDE'
100.0%	622.380772	--	--	10590767.6	Total
76.5%	**475.970683**	--	--	10213591.6	MPI
58.4%	363.761574	149.615814	29.4%	5084341.8	mpi_send_
17.6%	109.648667	283.128510	72.7%	5084341.8	mpi_recv_
13.2%	**82.433108**	324.558243	80.4%	44904.0	MPI_SYNC
					mpi_barrier_(sync)
10.3%	63.976981	--	--	332272.0	USER
2.5%	15.512290	0.743604	4.6%	1347.0	rhs_
1.7%	10.337003	1.738476	14.5%	22453.0	swapdat_
1.6%	10.099456	1.762906	15.0%	17511.0	swapinv_
1.2%	7.647495	0.311883	3.9%	38614.0	dx_

As can be seen, the communication part of the code (MPI + MPI_SYNC) takes 89.7% of the total time to run. Closer examination of the code revealed that the global transposition of data is done via explicit MPI blocking send and receive calls, rather than ALLTOALL type of communication. The application user effectively implemented MPI_ALLTOALL by himself. This is very inefficient on HECToR where highly optimised ALLTOALL routines are available. By rewriting the two communications routines (*swapdat* and *swapinv* in above table) using MPI_ALLTOALL, significant better performance was achieved as shown in the following Cray PAT report.

```
Time % |       Time |Imb. Time |   Imb. |    Calls |Group
       |            |          | Time % |          | Function
       |            |          |        |          |  PE='HIDE'

100.0% | 113.863253 |       -- |     -- | 519839.6 |Total
|-------------------------------------------------------------------
|  44.2% |  50.355772 |       -- |     -- | 102699.6 |MPI
||------------------------------------------------------------------
||  33.0% |  37.576854 | 2.345243 |   5.9% |  39964.0 |mpi_alltoall_
||   9.0% |  10.273081 | 9.488242 |  48.4% |   8913.8 |mpi_recv_
||   2.1% |   2.441814 | 0.093033 |   3.7% |  44904.0 |mpi_barrier_
||==================================================================
|  40.5% |  46.121814 |       -- |     -- | 332272.0 |USER
||------------------------------------------------------------------
||  13.5% |  15.378509 | 0.663161 |   4.2% |   1347.0 |rhs_
||   6.7% |   7.639985 | 0.350639 |   4.4% |  38614.0 |dx_
||   3.8% |   4.339009 | 4.312641 |  50.2% |      1.0 |exit
||   3.3% |   3.742230 | 0.134771 |   3.5% |   8082.0 |d2x_
||   1.4% |   1.614176 | 0.063646 |   3.8% |  40410.0 |dz_
||   1.4% |   1.608269 | 0.073636 |   4.4% |  46696.0 |thomax3d_
||   1.3% |   1.509728 | 0.035236 |   2.3% |   8082.0 |d2z_
||   1.3% |   1.456928 | 0.175116 |  10.8% |  22453.0 |swapdat_new_
||   1.2% |   1.400617 | 0.077870 |   5.3% |  38165.0 |dy_
||   1.2% |   1.309752 | 0.009497 |   0.7% |  48492.0 |thomaz3d_
||   1.1% |   1.249496 | 0.025782 |   2.0% |      1.0 |MAIN_
||   1.1% |   1.243370 | 0.013231 |   1.1% |  46247.0 |thomay3d_
||   1.0% |   1.169883 | 0.174941 |  13.1% |  17511.0 |swapinv_new_
||==================================================================
|  15.3% |  17.385667 |       -- |     -- |  84868.0 |MPI_SYNC
||------------------------------------------------------------------
||  11.6% |  13.243273 | 2.823314 |  17.7% |  44904.0 |mpi_barrier_(sync)
||   3.6% |   4.142394 | 0.951367 |  18.8% |  39964.0 |mpi_alltoall_(sync)
|==================================================================
```

First of all the MPI calls in the new code is **8.2** times faster than the old code, down from **558** seconds to only **68** seconds. Most computational intensive routines (shown in USER section of the above tables), such as *rhs* (for assembling the right hand side of equations being solved) and *dx* (for evaluating derivatives), spent nearly same amount of time to run. However additional savings were obtained from the non-MPI part of the communication routines (*swapdat_new* and *swapinv_new*) because of much better memory access patterns are used when packing and unpacking the MPI_ALLTOALL send and receive buffers – a further saving of **18** seconds was achieved. Here is an example of the old code in which the send buffer *fsend* is filled using big memory stride (fastest changing index *k* being the 2nd dimension) therefore poor cache usage:

```
        do i=1,nx
          do j=1,nyblk
            jj=j+jshift
```

```
        do k=1,nzblk
          fsend(i,k,j)=f(i,k,jj)
        end do
      end do
    end do
```

Here is an example of the new code where the MPI_ALLTOALL send buffer *send_buf* is assembled in a better manner (continuous memory space being updated):

```
do j=1,nyblk
  do k=j1,j2
    do i=1,nx+2
      send_buf(pos)=f(i,k,j)
      pos=pos+1
    enddo
  enddo
enddo
```

As shown by these examples, coding details can be very important in HPC term but indeed not directly relevant to the scientific study. Luckily it was possible to make these changes in a black-box manner so that the user could simply replace two communication subroutines with new ones having identical user interface.

Timing Result and Quad-core Performance

The timing results shown in this section is based on two medium-size simulations: one using 128^3 mesh and running on 128 cores; the other using 128*256^2 mesh and running on 256 cores. Although these two cases take only minutes to run on HECToR, they can represent a typical production run of the code as the per-core workload is very similar to that of a production run.

The following table[1] shows that the new code, with optimised communication routines, is more than 5 times faster than the old code. As the number of cores increase, the benefit becomes more obvious as the ALLTOALL communication is more dominant. It worthy mentioning that at the time of this writing (October 2009), the quadcore programming environment is not loaded by default on HECToR. It is therefore important to 'module load xtpe-barcelona' before compiling any applications to get the optimisation for the quadcore hardware and slightly better performance.

No. core	Old code	New code	New code (xtpe-barcelona module)	speed-up
128	622	112	105	**5.9**
256	4301		662	**6.5**

As the application is communication intensive, it is also possible to further improve the performance by using only 2 cores from each quad-core node, reducing the wall-clock-time by another 20% or so. However this is not recommended due to the current AU-charging policy.

1 Problem size of the two test cases arbitrary so only the speed-up values meaningful.

Other Suggestions

- As can be seen from the second Cray PAT report, the new code spends 9% of time in MPI_RECV. The data exchange there is not for computation but for collecting data onto a sub-set of processors to perform I/O. This has been poorly implemented. As the data structure of this application is extremely simple (3D Cartesian mesh), it would be very easy to implement a MPI-IO solution that both improves the I/O performance and clean up the I/O logic.

- The constraint posed by the existing communication algorithm is that the total number of cores that can be used in any simulation must be equal to or smaller than the number of mesh points in both y and z direction. If the group is to perform large-scale Direct Numerical Simulations in the future, it may be necessary to update the communication algorithm to use 2D decomposition. One such library is being developed by the author of this report in an ongoing Distributed CSE project to support another CFD application. So this group is very likely to benefit from the dCSE work in the near future.