CSE Case Study: Optimising the CFD code DG-DES

CSE Team
NAG Ltd., support@hector.ac.uk

Naveed Durrani
University of Sheffield

June 2008

Introduction

One of the activities of the NAG CSE (Computational Science and Engineering) team is to provide computational support to scientists using the HECToR service, which includes helping users get the most out of the machine by providing short periods of support to optimise code performance. This case study describes one such effort to improve the performance of a Computational Fluid Dynamics (CFD) code, DG-DES (Dynamic Grid - Detached Eddy Simulation), developed by a group from the Department of Mechanical Engineering at the University of Sheffield[1].

Some members of the group attended NAG HECToR training (which is freely available to all HECToR users) and the CSE team were able to let them know how to get access to HECToR and help port DG-DES on to the machine. After porting, the group found that the code was crashing and submitted a query requesting help with debugging. The CSE team spotted and fixed two minor errors in the code (a mis-declared array resulting in a segmentation fault and a divide by zero). With the code working properly the next step was to improve its performance with specific production runs in mind (the users wanted to perform a series of 12 hour runs using around 256 PEs), so a request for a short period of help with optimisation was sent to SAFE.

DG-DES

DG-DES is a Fortran 90 MPI code which models turbulent fluid flows over time in a 3-dimensional spatial domain. The finite element spatial domain is decomposed using the Metis graph partitioning software[2] and time-stepping follows a multi-stage Runge-Kutta scheme. DES is a hybrid numerical approach that combines the strengths of two other numerical schemes called Reynolds Averaged Navier Stokes (RANS) and Large Eddy Simulation (LES). DG-DES is also capable of simulations that require a dynamic grid, where the locations of mesh points are altered at regular intervals according to the behaviour of the fluid.

The test case supplied with the source code was for a comparatively small fixed mesh of around 35,000 nodes; production run meshes consist of more than 1.5 million nodes.

[1]http://www.shef.ac.uk/mecheng/staff/qin.html
[2]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

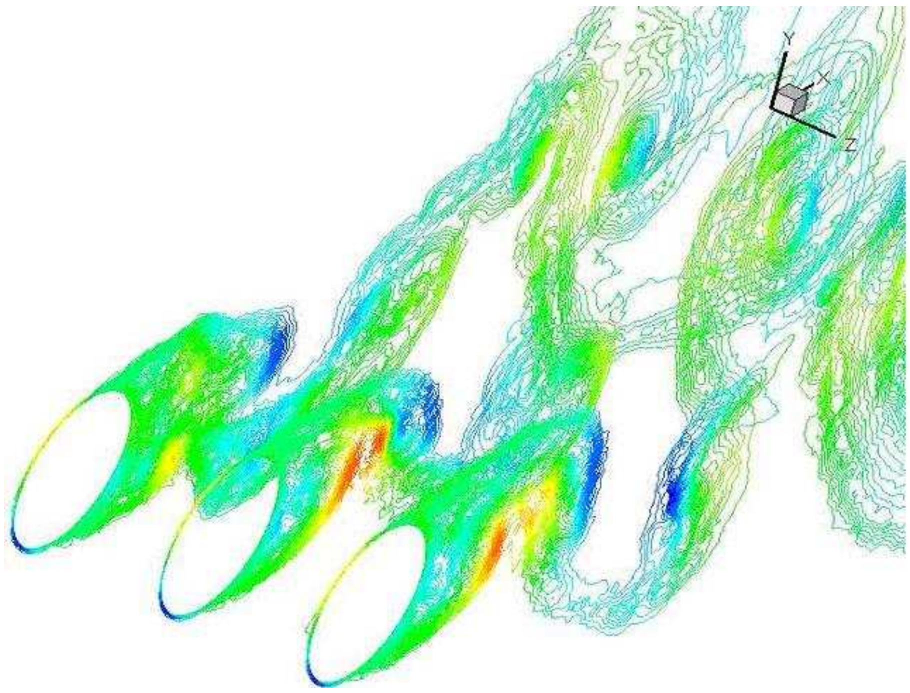

Figure 1: Plot of vorticity magnitude at different spanwise locations around cylinder for a Reynolds Number of $3.6 \times 10^{-}6$

Initial Performance Tests

In porting their code to HECToR the group employed the default options for compiling - simply replacing in their Makefile the old compiler command with the ftn command and using the PGI compiler without additional options.

Therefore, the first experiment was to find out which compiler and combination of optimisation options to use. Figure 2 illustrates the results of tests performed on the Test and Development System (TDS)[3] for the PGI and Pathscale compilers with and without the vendor-suggested optimisation options.

Figure 2 shows that the Pathscale (3.0) compiler produces the best-performing executable: compared with the case where the PGI (7.0.4) compiler is used without specifying optimisation options (top line, i.e. how the code was originally being compiled), simply switching to the Pathscale compiler and using `-Ofast` produces a significant speedup.

The tests in Figure 2 were in dual core mode; the results in Figure 3 show clearly that there is no benefit to running DG-DES in single core mode.

From browsing the source code it is apparent that DG-DES performs many subroutine and function calls within the main computational loops, which suggests the importance of inlining. Pathscale's `-Ofast` option expands to `-O3 -ipa -OPT:ro=2:Olimit=0:div_split=ON:alias=typed -fno-math-errno -ffast-math`, and although inlining is included as part of the Inter-Procedural Analysis option, `-ipa`, there are further, more aggressive options that control inlining. The following were tried in addition to `-Ofast`: `-IPA:plimit=20000:callee_limit=20000 -INLINE:aggressive=on`. Using `-INLINE:list=on` shows that more of the larger subroutines called within loops are inlined, but Figure 4 shows this has no effect on performance; the routines inlilned by `-ipa` are sufficient.

[3]A test machine available to the CSE team for supporting users.

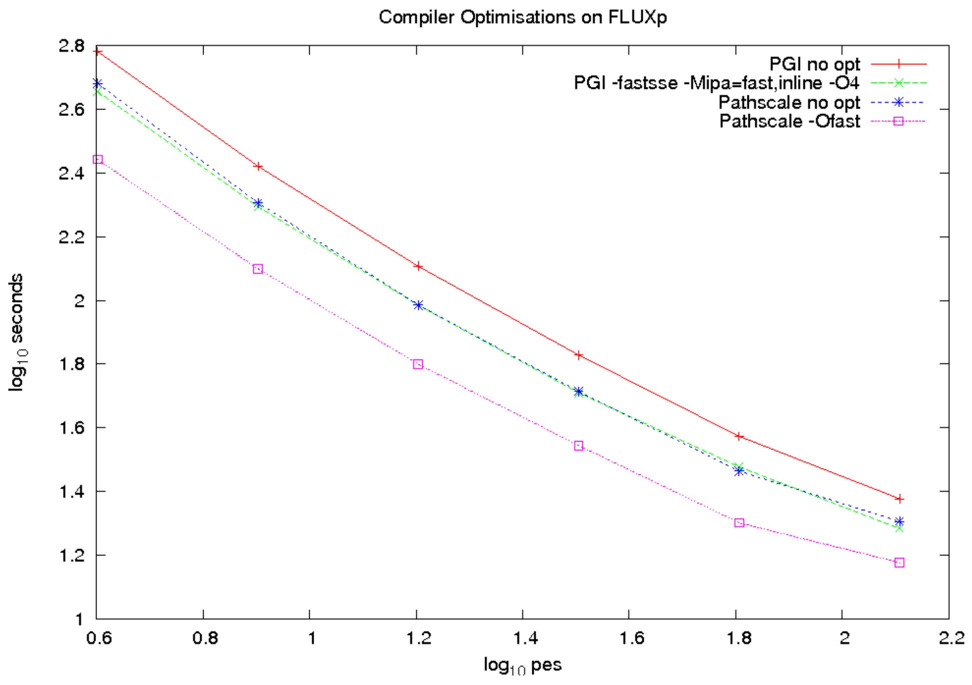

Figure 2: Performance of DG-DES for 4, 8, 16, 32, 64 and 128 Processing Elements (PEs) for the PGI and Pathscale compilers with and without optimisation flags.

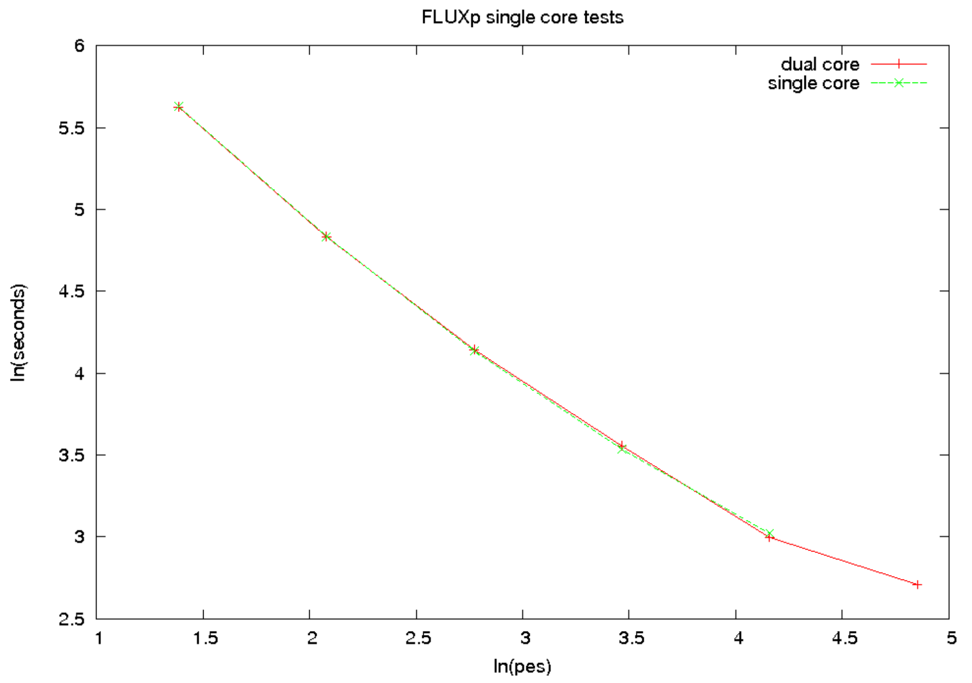

Figure 3: Performance of DG-DES running in single and dual core mode.

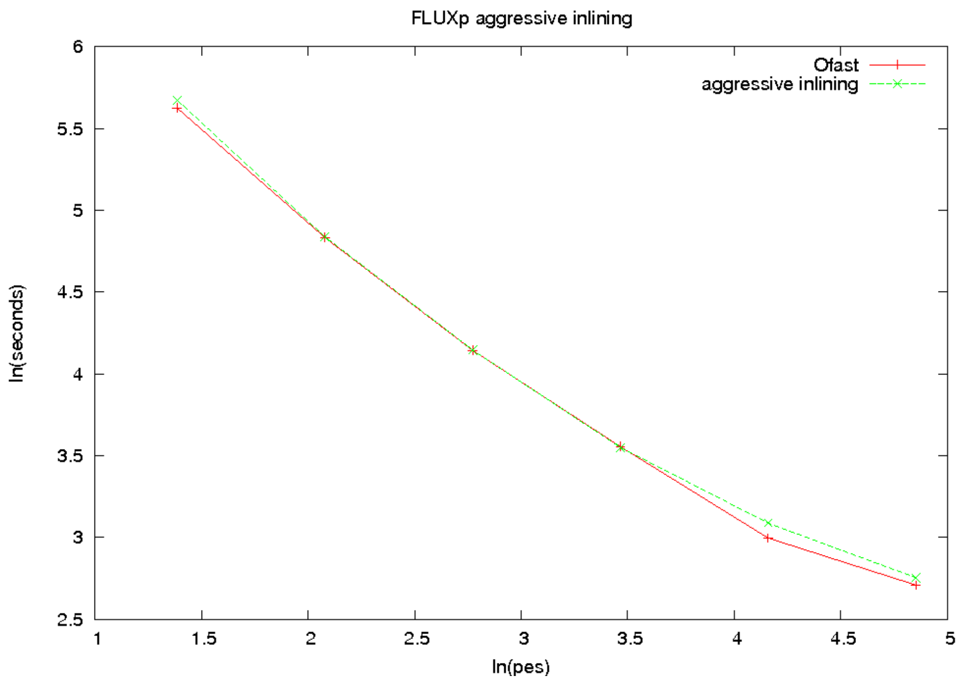

Figure 4: Performance of DG-DES with extra, aggressive inlining options.

Profiling

Profiling with CrayPAT reports that the subroutine `residual_mpi` and its children account for around 75% of execution time. However, the incompatibility of the `-ipa` option and the `-g` option added by CrayPAT meant that CrayPAT was of limited further use in this exercise to, for example, compare utilisation rates between optimised and unoptimised code. CrayPAT was also found to produce very slow instrumented executables for tracing experiments, even when the Automatic Profiling Analysis (APA) option was used (tracing still fewer routines than APA also proved sluggish). A custom timing module was therefore incorporated into the code, using the `MPI_WTIME` function.

Simple Optimisations

Subroutine `residual_mpi` is called within the main computational loop and calls other routines many times per iteration, resulting in millions of calls even for short test runs. Two of the top time-consuming routines, `mu` and `sonic`, are one-line functions which can be inlined manually very easily. Two operands for the calculations performed by these routines are constants, but were not declared as such; doing so gives the compiler the opportunity to remove two loads per calculation. Inlining and changing these variables to constants produced around a 10% speedup[4] overall compared to `-Ofast` alone.

It was observed that the statement `if(teqs!=0)` is executed repeatedly because the constant `teqs` (a flag for the turbulence model) declared as a variable. Re-declaring as a `parameter` allows the compiler to remove these conditionals, producing a further 5% speedup overall.

Repeated execution of the following case statement was also redundant since the users are only interested in 'DES' experiments:

[4]Percentage speedup is measured as follows throughout: $100 \times (originaltime - newtime) \div originaltime$, i.e. percentage of the original execution time that has been saved.

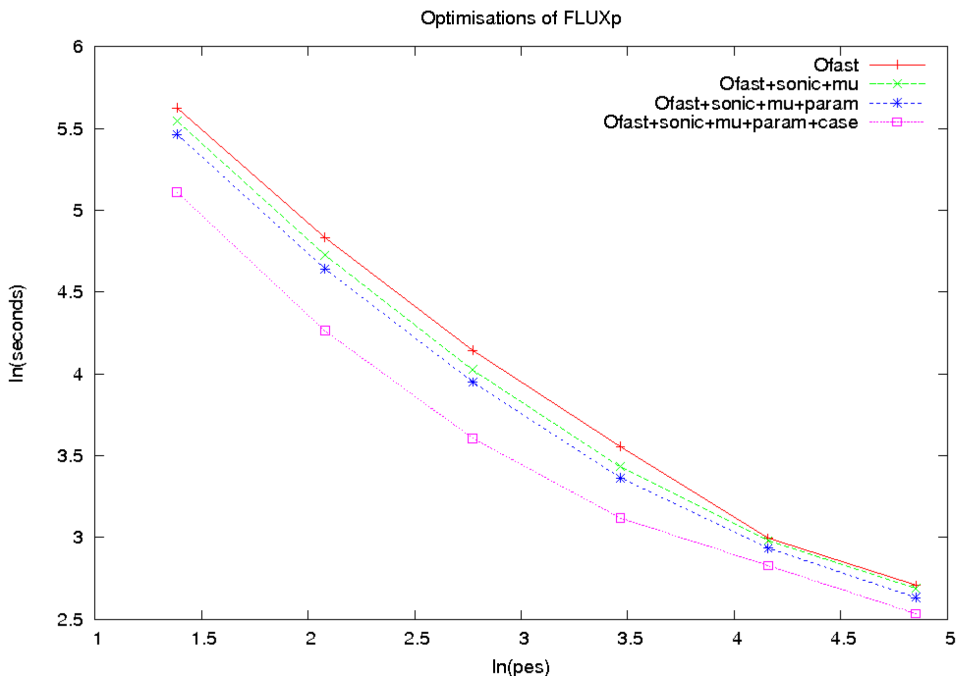

Figure 5: Performance of DG-DES with: `-Ofast` alone; after inlining `sonic` and `mu`; after declaring `teqs` as a parameter; and after eliminating the case statements. The combined performance improvement is around 40% for 16 PEs.

```
select case(TRIM(tmodel))
case('S-A', 'DES')
    tFV = Vn*tFV
    tFV = tFV*face(id)%VOL
case('k-w')
    ...
end select
```

Replacing this block with unconditional execution of the DES case produced a further 25% speedup overall. This change is controlled by a pre-processor macro, enabling users to switch back to the original code easily. Figure 5 shows the performance improvements made by each of these three changes.

Scaling

The performance curves in Figure 5 suggest that DG-DES may not scale well for the test case. This was confirmed by performing tests up to 512 PEs on HECToR: see Figure 6. Timing key sections of code, shown in Figure 7, reveals the source of the poor performance beyond 64 PEs: a broadcast in the timestepping loop, which indicates a load balancing problem. It was thought this was likely due to partitioning a relatively small, irregular test mesh among many processes; running the same code using a larger production mesh indicates that this seems to be the case (see Figure 8) and that there should be no problems scaling up to 256 PEs, as the users intend.

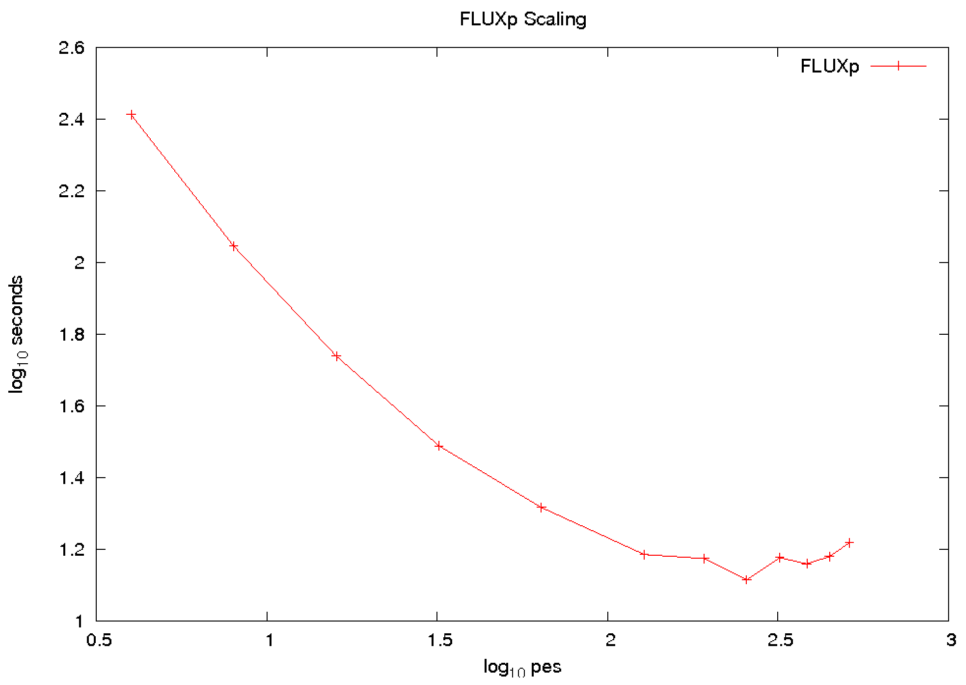

Figure 6: Performance of DG-DES in the test case up to 512 PEs: poor scaling.

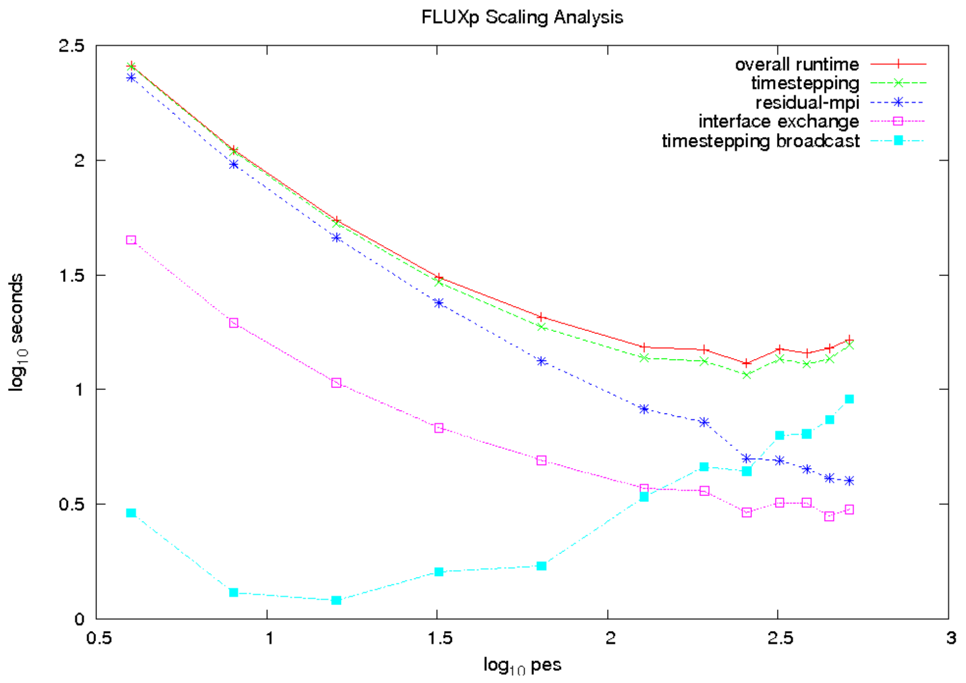

Figure 7: Timing key sections of code show that poor scaling is mainly due to the cost of a broadcast in the timestepping loop.

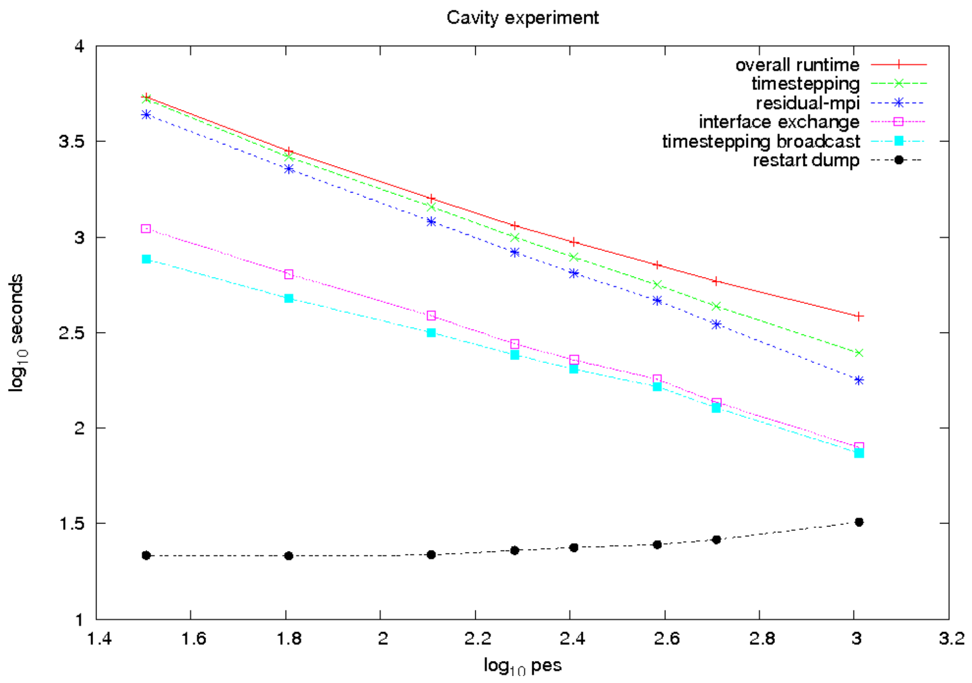

Figure 8: Timing the same sections of code as in Figure 7 for a production mesh of 1.8 million nodes shows that DG-DES does scale well up to 256 PEs, as required.

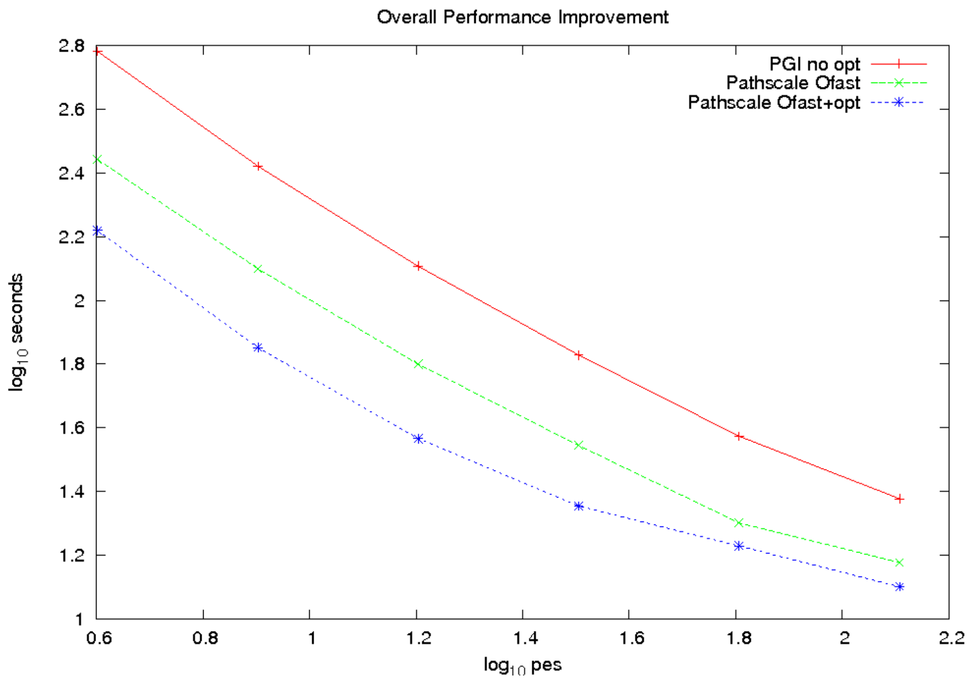

Figure 9: Overall performance improvement of DG-DES: before CSE help the code was being compiled using PGI with no optimisation options; with CSE help, the code runs 70% faster due to switching to the Pathscale compiler with `-Ofast` and making simple code modifications.

Summary

Figure 9 shows the performance of the code before and after CSE assistance – around a 70% performance improvement gained by making simple, changes such as switching compiler, enabling optimisation options and removing redundant code.

The NAG CSE team were able to support the DG-DES users by advising them how to gain access to HECToR, helping to port and debug the code, profiling and optimising for specific production runs, and providing advice in planning their experiments. The users have also been supplied with timing routines to help monitor the performance of their code in future. For example, the output below gives the cumulative time spent in key sections of code. In particular the problematic broadcast of Figure 7 is timed; if the difference between the maximum and minimum values is large then the problem is poorly load-balanced.

```
-----------------------------
Overall run time
max time = 70.801842
min time = 70.624277
-----------------------------
-Cumulative time for global loop
-max time = 69.031489
-min time = 68.958275
-----------------------------
---bcast with slamon
---max time = 2.804001
---min time = 0.064152
-----------------------------
```

This case study shows that it is expedient to allow the compiler to optimise your code, giving it as much help as possible. It also demonstrates how eliminating small sections of redundant code can have a large performance impact – it is not always necessary to make drastic changes. These changes may reduce the flexibility of your code, but this may be worthwhile for production runs. Also make sure you take into account the test case you are using for benchmarking; it is important to check that changes you make scale up to the large cases you intend to use for your experiments.