Science & Technology Facilities Council

Technical Report

Profiling Tools for MPI Programming

J.V. Ashby

11 Jan 2008

RAL-TR-2007-017

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44 (0)1235 44 6403
Email: Library@rl.ac.uk

ISSN 1358-6254

Profiling Tools for MPI Programming

J.V. Ashby
Computational Science and Engineering Department
STFC Rutherford Appleton Laboratory
J.V.Ashby@rl.ac.uk

Abstract

Programming parallel applications using MPI for optimum performance is a difficult task given the multiplicity of options for communication. In this report we discuss the use of four tools which can assist in profiling and analysing the performance of MPI programs. The tools are applied to two scientific codes on small problems to illustrate their use. A brief discussion of applying the tools to larger problems requiring moderate numbers of processors is also given. The four profiling tools presented can give insight into possible performance bottlenecks in MPI applications. They each have their strengths and weaknesses, both text-based tools and graphical displays, and we recommend that application programmers familiarise themselves with several of these tools.

1. Introduction

Programming parallel applications using MPI for optimum performance is a difficult task. Often there is no single correct method, and the multiplicity of options governing the method of communication (blocking, non-blocking, one-sided, asynchronous, broadcast, point-to-point, etc.) can lead to different results depending on the combination of algorithm and the underlying hardware. Indeed, were this not the case there would be no need for so many options. The best approach could be chosen, enshrined in the standard and nobody would need to use anything else.

Given that there are many approaches which can be used, getting maximum performance from a code means, amongst other things, optimising the use of MPI. While simple overall timings can give an impression of how well different communication strategies work, in complex codes a more sophisticated approach is necessary to identify bottlenecks.

In this report we discuss the use of four tools which can assist in profiling and analysing the performance of MPI programs. This is not an exhaustive set of tools. It deliberately omits, for example, one of the best known MPI analysis tools, Vampir (also known as the Intel Trace Analyzer) since this has been discussed by Sunderland [1]. We also omit detailed discussion of the capabilities of some of the tools beyond monitoring and profiling MPI usage. Some other tools were not available to us during the period of time this evaluation was undertaken.

To look at these tools we ran two application codes with the tools applied to them on HPCx, an IBM Regatta machine. which features 96 IBM eServer 575 compute nodes, each containing 16 1.5GHz IBM Power5 64-bit RISC processors. The 16 processors in a compute node are packaged into 2 MCMs. Each processor has a private L1 instruction cache of 64KB and L1 data cache of 32KB. The L2 combined data and instruction cache of 1.9MB is shared between 2 processors. The L3 cache of 36MB is also shared between 2 processors. Within a node, MCMs communicate via shared memory and a High Performance Switch provides communication between nodes.

The problems used were relatively small and did not use large numbers of processors. Thus this report only discusses an impression of how easy to use the tools are and does not cover how well they cope with large quantities of data produced by a large problem using several hundred processes. It is by no means an exhaustive evaluation, but is intended to introduce application developers to tools which may be of use to them. We do look briefly at tool use on moderate numbers of processors in section 7.

The tools are: mpitrace [2] (which also includes mpiprof and mpihpm), a simple profiler from IBM available on HPCx; fpmpi, a similar profiler produced by Argonne National Labs [3]; KOJAK which is more sophisticated and requires instrumentation of the code [4]; and TAU [5], a general profiler which can profile much more than simply MPI, and which can be used in a number of different modes from simple profiling to detailed instrumentation of the code. Kojak and TAU share a pedigree to some extent. TAU is the result of a collaboration between the University of Oregon, Los Alamos National Laboratory and the Forschungszentrum Jülich. Jülich and the University of Tennessee together produced Kojak which uses TAU to instrument a user's code (in a way which does not require the user to know TAU itself).

2. The Applications

To look at how easy the various tools are to use we have applied them to two scientific applications, SIC-LMTO [6] and Flite3D [7].

The SIC-LMTO code of Temmerman and Szotek [6] is a self-consistent spin polarised calculation of the electronic band structure of a crystalline material. It uses the Linear Muffin-Tin Orbitals approach with a Self-Interaction Correction and is written mostly in Fortran95, although there is some legacy code still in Fortran77. We used a dataset for what constitutes a reasonably large problem for this code, the magnetic half metal $NiFe_2O_4$ in an inverse spinel structure. In this case the program treats 26 "atoms" (2 types of Fe, 1 Ni, 2 O and 4 empty spheres) with 98 bands leading to a Hamiltonian matrix of dimension 234. This is diagonalised at 512 points within the Brillouin zone.

The essence of the program is the solution of the eigenproblem, $H(k)\psi_k=E_k\psi_k$. The Hamiltonian H(k) depends on all the ψ_k through the electron density $n(r)=\Sigma_{\text{all occupied states}}|\psi_k(r)|^2$. Initially a guess is made at an electron density, the eigenproblem is solved and a new electron density generated. This is then fed back until self-consistency is reached. Within this self consistency loop each k-value can be solved for independently. The program is parallelised by farming out the k-points among the available processors and then performing a global broadcast of the results so that each processor can then calculate the electron density to use for the next iteration. The code is known to scale poorly for numbers of processors above about 16 due to its parallel strategy. The partitioning of the problem is in **k**-space, farming out points in the Brillouin Zone. There is no point in using more than ~512 points in the Brillouin Zone, and if these are distributed among too many processors the communications swamp the computation. For our experiments we used 16 processors which gave job times of the order of 20 minutes and kept the amount of MPI profiling data to a level that could be assimilated.

FLITE3D [7] is a three-dimensional CFD code which solves the Navier Stokes equations using a multigrid method on an unstructured mesh of finite elements. The space around an object in which fluid flows is divided into a set of tetrahedral cells. These cells and the points which make up their vertices form a mesh, and the continuous equations are transformed to a discrete form on this mesh so that the pressure and velocity of the fluid are found at each of the points. The Navier-Stokes equations define the density, pressure and velocity of the fluid. When discretised and linearised these equations generate a large sparse linear system in the variables $U_i(r_j)$, $P(r_j)$ noting that the pressure implies the density through the equation of state.

In the multigrid method several different grids of varying degrees of fineness are overlaid. The problem is solved approximately on one grid, then transferred to the next grid by a process known as prolongation (moving from coarse to fine) or restriction (moving from fine to coarse). At each mesh level the problem is solved (on coarse grids the problem is solved to give corrections to the solution on the finer grid). There are many different approaches to moving between grids – in FLITE3D the V-cycle is used in which the grid levels are fully traversed from fine to coarse and then back again. Parallelisation is achieved by domain decomposition. The meshes are divided into as many sections as there are processors available and the discrete problem is solved on each individual section or partition. Then the values at the interface between partitions are exchanged and provide new boundary conditions for the next stage of the solution process. This requires inter-process communication where each partition sends its boundary values to each of its neighbouring partitions

and receives boundary values from each of them.

We used a dataset for a wing-body assembly where the mesh consisted of 51737 points and 302079 tetrahedra. This is quite small for this code, and the run times were further reduced by only running for 100 timesteps instead of the 1000 used when the code is being analysed for computational performance. Since we are not interested in comparing the performance of MPI implementations one against the other, merely understanding how to use the profiling tools, the poor statistics that this reduction entails are unimportant.

3. mpitrace

mpitrace is a library of wrapper routines which instrument the actual MPI library, and against which an application can be linked. The result of running the application is then to produce a set of files summarising the MPI activity, one file for each process. There are three tools in the mpitrace suite: mpitrace itself which produces an executable which will provide low-overhead elapsed time measurements of MPI calls, mpihpm provides additional hardware performance data such as floating point operations, cache misses, etc., and mpiprof which gives more detailed call-graph data for MPI calls.

mpihpm goes beyond the scope of this study. To invoke it the documentation suggests linking the application adding in `-L/usr/local/lib -lmpitrace -lpmapi`, then setting the environment variable `HPM_GROUP` to the number of a Power-4 Hardware Performance Monitor Group [8], e.g. 5. A set of files, file, `mpi_profile_groupx.y` is produced where `x` is the group number and `y` the process rank.

To use mpitrace it is sufficient (on HPCx) to link the application with `-L/usr/local/lib -lmpitrace` before the MPI library is referenced, e.g. by `-lmpi`. This will then load the wrapper library as an intermediate layer between the application and the substantive MPI. Running the application produces a set of files `mpi_profile.y` where y is again the process rank. A typical file, the 0-th instance from a 4 processor run of Flite3D looks like this:

```
---------------------------------------------------------------
MPI Routine                 #calls     avg. bytes      time(sec)
---------------------------------------------------------------
MPI_Comm_size                    2            0.0          0.000
MPI_Comm_rank                    1            0.0          0.000
MPI_Send                     16827         4714.6          0.067
MPI_Recv                     16833         4810.6          0.483
MPI_Probe                        6            0.0          0.000
MPI_Bcast                      201            7.4          0.013
MPI_Barrier                      1            0.0          0.000
MPI_Gather                       2            4.0          0.000
MPI_Gatherv                      3        23240.0          0.013
MPI_Reduce                     300            9.3          0.818
MPI_Allreduce                  504           13.5          0.028
---------------------------------------------------------------
total communication time = 1.422 seconds.
total elapsed time       = 24.526 seconds.
user cpu time            = 24.408 seconds.
system time              = 0.030 seconds.
maximum memory size      = 15428 KBytes.

---------------------------------------------------------------
```

```
Message size distributions:

MPI_Send                #calls     avg. bytes      time(sec)
                           102          116.0         0.000
                           204          170.0         0.001
                           306          332.0         0.001
                          3906          768.5         0.012
                          4002         1681.3         0.013
                          2905         3250.9         0.011
                          2801         6410.5         0.012
                          1301        12819.1         0.008
                          1300        19520.0         0.010

MPI_Recv                #calls     avg. bytes      time(sec)
                           102          116.0         0.000
                           204          170.0         0.000
                           306          332.0         0.017
                          3906          768.5         0.022
                          4002         1681.3         0.114
                          2905         3250.9         0.104
                          2801         6410.5         0.075
                          1301        12819.1         0.007
                          1300        19520.0         0.143
                             6       274133.3         0.001

MPI_Bcast               #calls     avg. bytes      time(sec)
                             1           0.0         0.000
                           182           4.0         0.013
                            10          14.0         0.000
                             2          20.0         0.000
                             3          49.3         0.000
                             2          88.0         0.000
                             1         256.0         0.000

MPI_Gather              #calls     avg. bytes      time(sec)
                             2           4.0         0.000

MPI_Gatherv             #calls     avg. bytes      time(sec)
                             1           4.0         0.012
                             1          20.0         0.000
                             1       69696.0         0.001

MPI_Reduce              #calls     avg. bytes      time(sec)
                           200           4.0         0.817
                           100          20.0         0.001

MPI_Allreduce           #calls     avg. bytes      time(sec)
                             4           4.0         0.000
                           400          12.0         0.027
                           100          20.0         0.001
```

First of all comes an overall summary of activity for each MPI routine giving the number of times each was called, the average message length and the time in seconds spend in each routine. Several points emerge from this section. MPI_Send and MPI_Recv are the main workhorses of the code in terms of the number of calls. There is a large imbalance between these two routines with MPI_Recv taking 8 times as long as MPI_Send, time spent waiting for messages. A large amount of data is transferred in three calls to MPI_Gatherv. The bulk of the time is spent in the 300 calls to MPI_Reduce. This summary is followed by another of the times involved, showing that for this problem MPI calls constitute 6% of the total time.

After the summaries comes a detailed picture of the message size distributions. This can take a bit of

interpreting. For each routine which transfers data (thus routines such as MPI_Comm_size are not included) there are three columns: the number of calls, the average message size and the time taken. The data here has been binned logarithmically; the first line covers messages of 1 byte, the second of 2 bytes, the third 3-4 bytes, the fourth, 5-8 bytes and so on. Note that lines where no calls appear have been suppressed, so for example MPI_All_reduce, the final routine in the list, has entries for 3-4 bytes, 9-16 bytes and 17-32 bytes, there being no messages of 1, 2 or 5-8 bytes in length. The bins can be inferred from the average message length (which must perforce lie in the bin). The number of calls is then the frequency of calls in the bin and the time is the total time taken by messages in that bin. Of interest here is the last three lines of the MPI_Recv entry. Firstly we note that there are 6 MPI_Recvs of large amounts of data that are not matched by MPI_Sends, although the other smaller messages match up. The exact matching of these smaller messages is characteristic of the halo exchange algorithm used by the program. These larger messages do not take up much time, however, and *are* matched by MPI_Sends in the other processes. More interestingly from the point of view of performance is the two entries above where similar numbers of calls take 20 times longer to receive when they are only on average 60% larger.

The detailed statistics on MPI_Gatherv show that the three calls have very different characteristics. One call only passes 4 bytes, one passes 20 bytes and the other transfers nearly 70 kilobytes. The routine is called three times within the code, once in the main program and twice in the routine input. The main program instance collects the number of mesh points, an integer, on each process into an array on the master process; this is the 4 byte instance. Within input, MPI_Gatherv first collects the array which maps local node numbers to global ones – this is a large amount of data, nearly 70kb per process. Then the number of nodes in each level of the multigrid mesh is collected, in this case five levels leads to messages of 20 bytes. At first sight there is an anomalously low time for the large message. This is because an optimised MPI implementation can simply use a memory copy for messages from one process to itself. The times for the other processes reflect this by being much longer (~0.05s). Looked at this way the anomaly becomes the long time taken by the main program instance. We suspect that load imbalance is the cause – the slave processes show shorter times than the master in this instance suggesting that they reach this point in the code, send their messages and proceed. The master, on the other hand, has to wait until all the messages have been received and so can only proceed when the slowest process has reported. The ability to show message traces within Vampir would confirm this.

Looking at the summary we saw that MPI_Reduce was a potential bottleneck. The detail for this routine shows 200 very small (4 byte) calls taking 800 times as much time as 100 larger (20 byte) calls. Examination of the other three processes show that process 1 and 2 have similar time profiles for MPI_Reduce, while process 3 is more balanced. Looking at the code shows MPI_Reduce is used in several places, always to find the maximum of a real field or to sum integers. Distinguishing definitively which of these is responsible for the disparity in times requires more detailed analysis.

This further detail is provided by mpiprof. For this the application must be compiled and linked either with `-g` (the debugging flag) or `-qtbtable=full`. Either of these will produce a trace-back table which mpitrace (linked as described above) can use to determine which application routine called an MPI function. The profile files produced start the same as for mpitrace, but contain an additional call graph section. For the case above this looks like this:

```
Call Graph Section:

communication time = 0.826 sec, parent = supsp
   MPI Routine                #calls        time(sec)
   MPI_Reduce                    200           0.826

communication time = 0.565 sec, parent = xchang
   MPI Routine                #calls        time(sec)
   MPI_Send                    16827           0.069
   MPI_Recv                    16827           0.496

communication time = 0.027 sec, parent = getforce
   MPI Routine                #calls        time(sec)
   MPI_Allreduce                 400           0.027

communication time = 0.016 sec, parent = flite3d_fs
   MPI Routine                #calls        time(sec)
   MPI_Gather                      2           0.000
   MPI_Gatherv                     1           0.016

communication time = 0.014 sec, parent = mg
   MPI Routine                #calls        time(sec)
   MPI_Bcast                     100           0.014

communication time = 0.002 sec, parent = rsd
   MPI Routine                #calls        time(sec)
   MPI_Reduce                    100           0.001
   MPI_Allreduce                 100           0.001

communication time = 0.002 sec, parent = gather
   MPI Routine                #calls        time(sec)
   MPI_Recv                        6           0.001
   MPI_Probe                       6           0.001

communication time = 0.001 sec, parent = input
   MPI Routine                #calls        time(sec)
   MPI_Gatherv                     2           0.001

communication time = 0.000 sec, parent = readnam
   MPI Routine                #calls        time(sec)
   MPI_Bcast                      85           0.000

communication time = 0.000 sec, parent = volcel
   MPI Routine                #calls        time(sec)
   MPI_Allreduce                   4           0.000

communication time = 0.000 sec, parent = bc_fs_command
   MPI Routine                #calls        time(sec)
   MPI_Bcast                      14           0.000

communication time = 0.000 sec, parent = get_fs_command
   MPI Routine                #calls        time(sec)
   MPI_Bcast                       1           0.000

communication time = 0.000 sec, parent = pinit
   MPI Routine                #calls        time(sec)
   MPI_Comm_rank                   1           0.000

communication time = 0.000 sec, parent = pfinish
   MPI Routine                #calls        time(sec)
   MPI_Barrier                     1           0.000

communication time = 0.000 sec, parent = mai1
   MPI Routine                #calls        time(sec)
```

```
   MPI_Bcast                        1          0.000

communication time = 0.000 sec, parent = pstart
   MPI Routine                 #calls        time(sec)
   MPI_Comm_size                    2          0.000
```

It is clear from this that the MPI_Reduce which is taking the most time is being called from `supsp`, a routine which counts the total number of nodes in the mesh where the flow is supersonic (by counting the number of supersonic nodes for each process and summing them across processes) and the maximum Mach number in the flow field (by finding the maximum on a process, then using MPI_Reduce to find the maximum over all processes). Thus `supsp` calls MPI_Reduce twice, first to sum a set of integers (the number of supersonic nodes), then to find the maximum of a set of reals, the maximum Mach number on each process, so we still cannot completely distinguish which call is giving the problem, but we have homed in on the area of code to look at in more detail. Since in each case only one number is being passed it is again a problem of load balancing. Looking across the processes we see that processes 0, 1 and 2 each take about 0.8s for these calls, whereas process 3 takes only 0.004s. It appears that the other processes are all waiting for process 3, which is computationally more heavily loaded and lags behind in reaching `supsp`. This could be verified by altering, if possible, the way in which the mesh partitions are assigned to processes. No such help is available to determine what is going on with MPI_Send and MPI_Recv. As the call graph shows, these are all in the same routine, `xchang`, where the processes exchange interface data during the iterations. All four processes display the same differential time between MPI_Send (~0.07s) and MPI_Recv (between 0.5 and 4s). This could present scope for examining the way in which the algorithm is structured in case the waits that MPI_Recv is having to perform can be avoided. The exceptions (the six calls unaccounted for above) are the MPI_Recv calls in gather. Looking at the other profiles shows that they are indeed matched by calls to MPI_Send in gather on other processors.

We have thus seen how with fairly simple tools and text based output it is possible to elicit useful information about the performance of an MPI application. We applied the same process to SIC_LMTO, the electronic structure code, with slightly different results. Obviously the actual analysis was different since SIC_LMTO only uses MPI_Bcast and MPI_Allreduce but the profile files for the individual processes were empty and the information was instead appended to standard output without identifying which data came from which process. We discuss this further in the next section.

4. fpmpi

fpmpi is a similar wrapper library to mpitrace. Versions are available to work with the MPI libraries from SGI, IBM and MPICH. To profile applications `-L<fpmpi directory> -lfpmpi` must be added to the link step before the MPI library is linked. For Fortran programs using either SGI or IBM MPIs, an additional object file, `farg.o`, source for which is supplied, must be linked sandwiched between two `-lfpmpi` flags.

Running the application produces in this case just one file, `fpmpi_profile.txt` which contains information about the whole run. The file name can be changed using the `MPI_PROFILE_FILE` environment variable. The profile for our 4 process Flite_3D run is:

```
MPI Routine Statistics (FPMPI2 Version 2.1d)

Explanation of data:
Times are the time to perform the operation, e.g., the time for MPI_Send
```

```
Average times are the average over all processes, e.g., sum (time on each
process) / number of processes
Min and max values are over all processes(Data is always average/min/max)
Amount of data is computed in bytes.  For point-to-point operations,it is the
data sent or received.  For collective operations, it is the data contibuted to
the operation.  E.g., for an MPI_Bcast, the amount of data is the number of
bytes provided by the root, counted only at the root. For synchronizing
collective operations, the average, min, and max time spent synchronizing is
shown next.
Calls by message size shows the fraction of calls that sent messages of a
particular size.  The bins are 0 bytes, 1-4 bytes, 5-8 bytes, 9-16, 17-32, 33-
64, -128, -256, -512, -1024, -4K, -8K, -16K, -32K, -64K, -128K, -256K, -512K,
-1M, -4M, -8M, -16M, -32M, -64M, -128M, -256M, -512M, -1GB, >1GB.
Each bin is represented by a single digit, representing the 10's of percent of
messages within this bin.  A 0 represents precisely 0, a . (period) represents
more than 0 but less than 10%.  A * represents 100%.
Messages by message size shows similar information, but for the total message
size.

The experimental topology information shows the 1-norm distance that the longest
point-to-point message travelled, by process.

MPI_Pcontrol may be used to control the collection of data.  Use the values
defined in fpmpi.h, such as FPMPI_PROF_COLLSYNC, to control what data is
collected or reported by FPMPI2.

Date:           Tue Sep 11 09:58:56 2007
Processes:      4
Execute time:   5.694
Timing Stats: [seconds] [min/max]       [min rank/max rank]
  wall-clock: 5.694 sec 0.000000 / 22.775240    1 / 0
        user: 22.83 sec 22.776878 / 22.851238   0 / 1
         sys: 0.04978 sec       0.045403 / 0.061962     1 / 0

Memory Usage Stats (RSS) [min/max KB]:  13204/14932

                  Average of sums over all processes
Routine                 Calls       Time Msg Length    %Time by message length
                                                      0.........1.........1........
                                                                K         M
MPI_Allreduce       :     504    0.00843   6.82e+03 0.08200000000000000000000000
MPI_Bcast           :     201     0.0249        372 .*0.....00000000000000000000
MPI_Gather          :       2   3.81e-05          8 0*00000000000000000000000000
MPI_Gatherv         :       3     0.0404    6.8e+04 0.00.0000000037000000000000
MPI_Reduce          :     300      0.316    2.8e+03 0*00.0000000000000000000000
MPI_Recv            :   16828      0.563   7.74e+07 000000...14112000.0000000000
MPI_Send            :   16828       1.83   7.74e+07 000000...24121000.0000000000
MPI_Probe           :       1   1.17e-05
MPI_Barrier         :       1      0.182

Details for each MPI routine
                  Average of sums over all processes
                                                  % by message length
                                (max over         0.........1.........1........
                                 processes [rank])          K         M
MPI_Allreduce:
        Calls     :        504          504 [   0] 0.08200000000000000000000000
        Time      :    0.00843       0.0086 [   3] 0.08200000000000000000000000
        Data Sent :   6.82e+03         6816 [   0]
        SyncTime  :      0.022       0.0405 [   2] 0.08200000000000000000000000
MPI_Bcast:
        Calls     :        201          201 [   0] .90.....00000000000000000000
        Time      :     0.0249        0.037 [   1] .*0.....00000000000000000000
        Data Sent :        372         1488 [   0]
```

```
MPI_Gather:
        Calls     :          2            2 [   0] 0*00000000000000000000000000
        Time      :   3.81e-05     6.41e-05 [   0] 0*00000000000000000000000000
        Data Sent :          8            8 [   0]
MPI_Gatherv:
        Calls     :          3            3 [   0] 03003000000000130000000000000
        Time      :     0.0404       0.0847 [   3] 0.00.000000000370000000000000
        Data Sent :    6.8e+04        70424 [   1]
MPI_Reduce:
        Calls     :        300          300 [   0] 07003000000000000000000000000
        Time      :      0.316        0.631 [   2] 0*00.000000000000000000000000
        Data Sent :    2.8e+03         2800 [   0]
MPI_Recv:
        Calls     :      16828        16833 [   0] 000000...24111000.0000000000
        Time      :      0.563        0.756 [   0] 000000...14112000.0000000000
        Data Sent :   7.74e+07     98199240 [   3]
        SyncTime  :       0.33        0.515 [   0] 000000...14112000.0000000000
MPI_Send:
        Calls     :      16828        16829 [   1] 000000...24111000.0000000000
        Time      :       1.83         3.76 [   3] 000000...24121000.0000000000
        Data Sent :   7.74e+07     98749040 [   3]
        SyncTime  :       1.71         3.64 [   3] 000000...23121000.0000000000
        Partners  :          3 max 3(at 0) min 3(at 0)
MPI_Probe:
        Calls     :          1
        Time      :   1.17e-05
        SyncTime  :          0
MPI_Barrier:
        Calls     :          1
        Time      :      0.182
MPI_Comm_split(all communicators):
        Calls     :          1 max 1(at 0) min 1(at 0)
MPI_Comm_split(MPI_COMM_WORLD):
        Calls     :          1 max 1(at 0) min 1(at 0)

Summary of target processes for point-to-point communication:
1-norm distance of point-to-point with an assumed 2-d topology
(Maximum distance for point-to-point communication from each process)
  2  2
  2  2

Detailed partner data: source: dest1 dest2 ...
0:1 2 3
1:0 2 3
2:0 1 3
3:0 1 2
```

The preamble explains how the data is presented and indicates that more control over what data is collected and presented is possible. Then come some overall statistics for the run, then data for the MPI calls. In this case we see that the average time for MPI_Send (over processes) is three times that for MPI_Recv. At first sight this contradicts what we learnt using mpitrace but looking at the detailed data for individual routines we see that there is a much larger maximum time for MPI_Send from process 3 than the maximum for MPI_Recv, located on process 0. and it suggests that the problem lies in synchronisation, given the large value of the maximum SyncTime, also located on process 3.

Looking at the data for MPI_Reduce confirms what we learnt from mpitrace, that virtually 100% of the time this routine uses is in sending short 4 byte messages.

The profile finishes with an analysis of the process topology. In this case it is uninteresting, all processes communicate with every other process, but in more complex cases, in particular with much

larger numbers of processes, this could provide useful information to improve performance by locating processes which need to communicate "close" to one another.

Again, we also applied fpmpi to SIC_LMTO, but in this case were unable to find any sign of the profile output. We surmise that in both cases something in the code was interfering with the I/O system used by the profiling libraries. If this is the case, the usefulness of the tools is limited, though as we have seen, when they do work they can provide important information.

5. KOJAK

KOJAK (which stands for Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks) represents a different approach to profiling MPI applications. It is also capable of profiling OpenMP and SHMEM applications, as well as combinations of the three programming models.

KOJAK is built of five components, EPILOG [9], OPARI [10], EARL [11], EXPERT [12] and CUBE [13]. EPILOG is a binary event trace format, combined with an API and run-time library for generating trace files. The first stage in using KOJAK is to pass the application code through the OPARI instrumenter. This is a source-to-source translator which adds calls to the EPILOG and POMP profiling libraries automatically. A fully automatic method of doing this is provided, but the semi-automatic method is recommended. Routines are identified for profiling by adding directives to the code: to profile a procedure in C the following must be inserted before the first executable statement in the procedure:

```
#pragma pomp inst begin(name)
```

and before the procedure finally returns:

```
#pragma pomp end(name)
```

If there are alternative returns, then

```
#pragma pomp altend(name)
```

should be inserted. In Fortran the equivalents are

```
!POMP$ INST BEGIN(name)
!POMP$ INST END(name)
!POMP$ INST ALTEND(name)
```

Additionally the main program must have

```
#pragma pomp inst init
```

or

```
!POMP$ INST INIT
```

as the first executable statement.

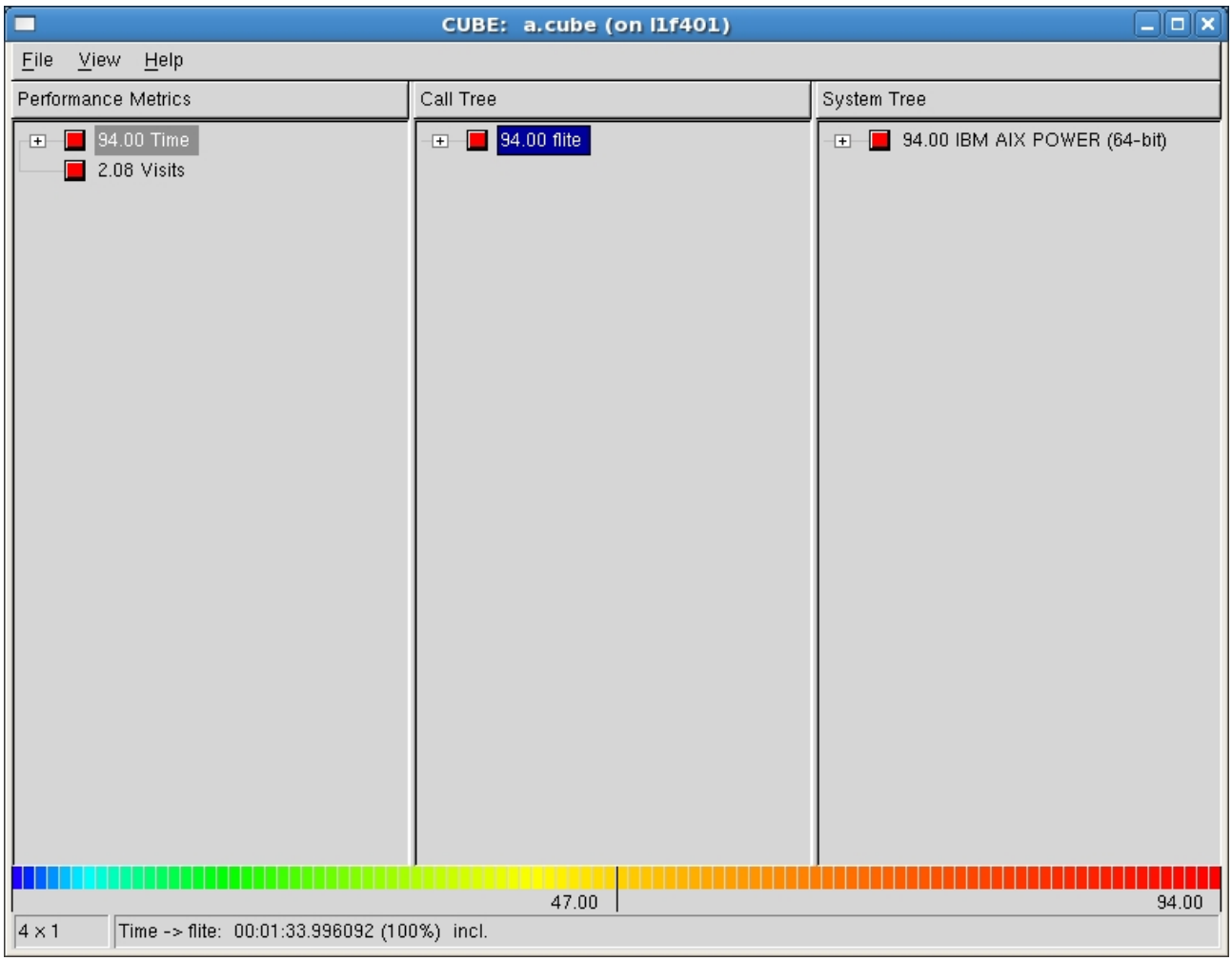
The application is built in the usual way, except that the compiler command with all its arguments must be preceded by `kinst-pomp`. The usual way to do this is to redefine the compiler in the Makefile, e.g. `F90=kinst-pomp mpxlf90_r`. When the application is run, a file `a.elg` is produced. This is an EPILOG trace file. Some environment variables can be used to control this file,

see [14] for details. The analysis of the data within this file is done by the three remaining components using the command `kanal a.elg`. This uses EARL, a high level interface to the EPILOG event traces and the EXPERT component which searches the event trace for patterns which indicate low performance. The results of this analysis are translated into a CUBE file and the final component, CUBE, is used to present and browse it. If desired the analysis and presentation can be split up by using EXPERT and CUBE separately. This might be appropriate where presentation is to be done remotely from the application machine.

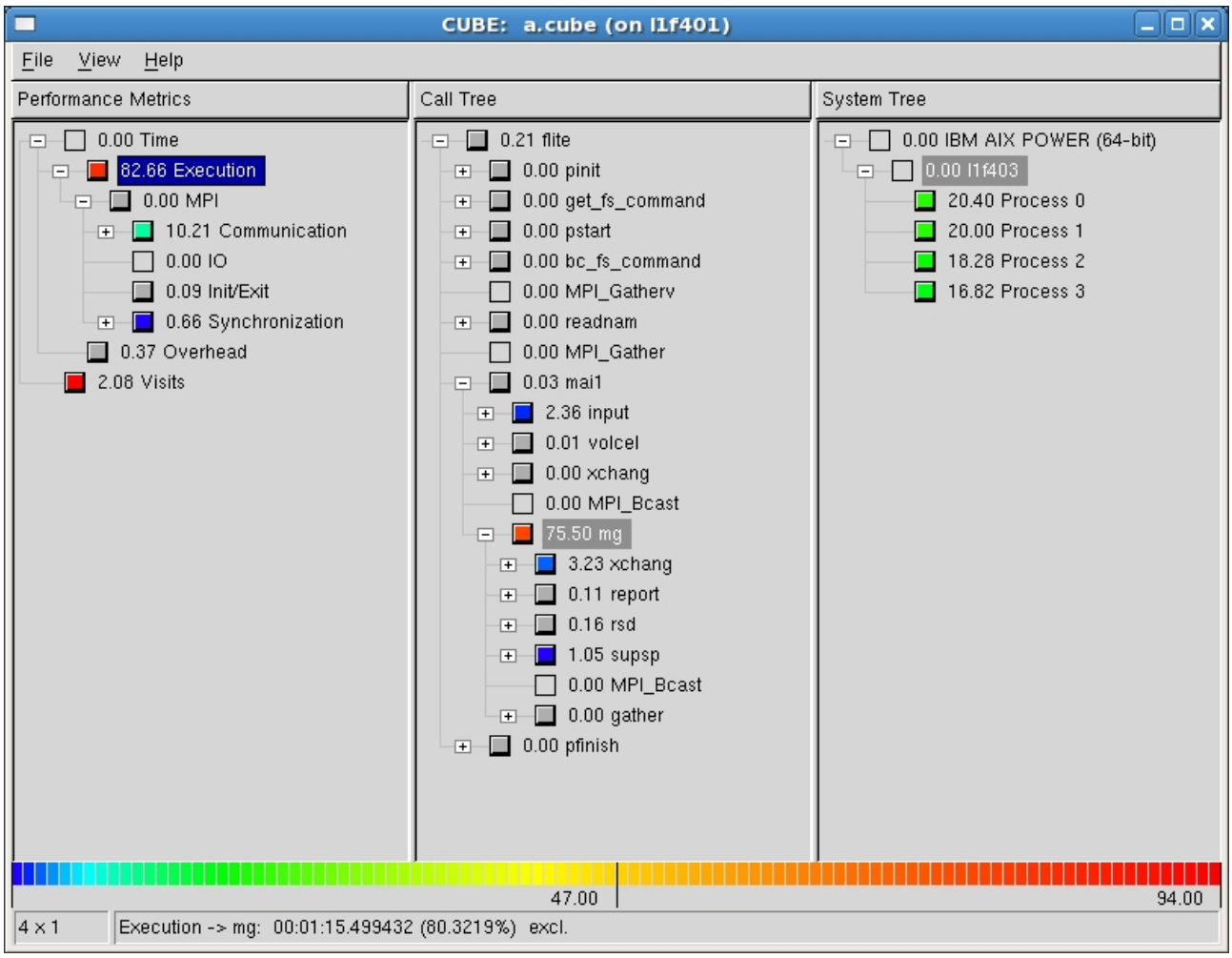
If greater control is desired, the code can be completely manually instrumented. For this work we used the semi-automatic approach and only profiled those routines with MPI calls.

KOJAK is freely downloadable, but we did not find installation completely straightforward. It was complicated by the need to build two versions, one for 32 bit and one for 64 bit operation on the IBM, and also by the number of subsystems that need to be correctly built. When it was eventually installed, however, with the help of the HPCx administrators, the process worked smoothly.

Once the program has been instrumented and executed, the bulk of a user's interaction with KOJAK is through the CUBE interface. CUBE presents the results of the EXPERT analysis in three panes each of which shows a hierarchical tree view. The leftmost shows performance metrics, the central pane locates the problem in the call tree and the rightmost displays process information. To illustrate the use of these we consider the use of CUBE on the data from an instrumented Flite3D run. The initial window is

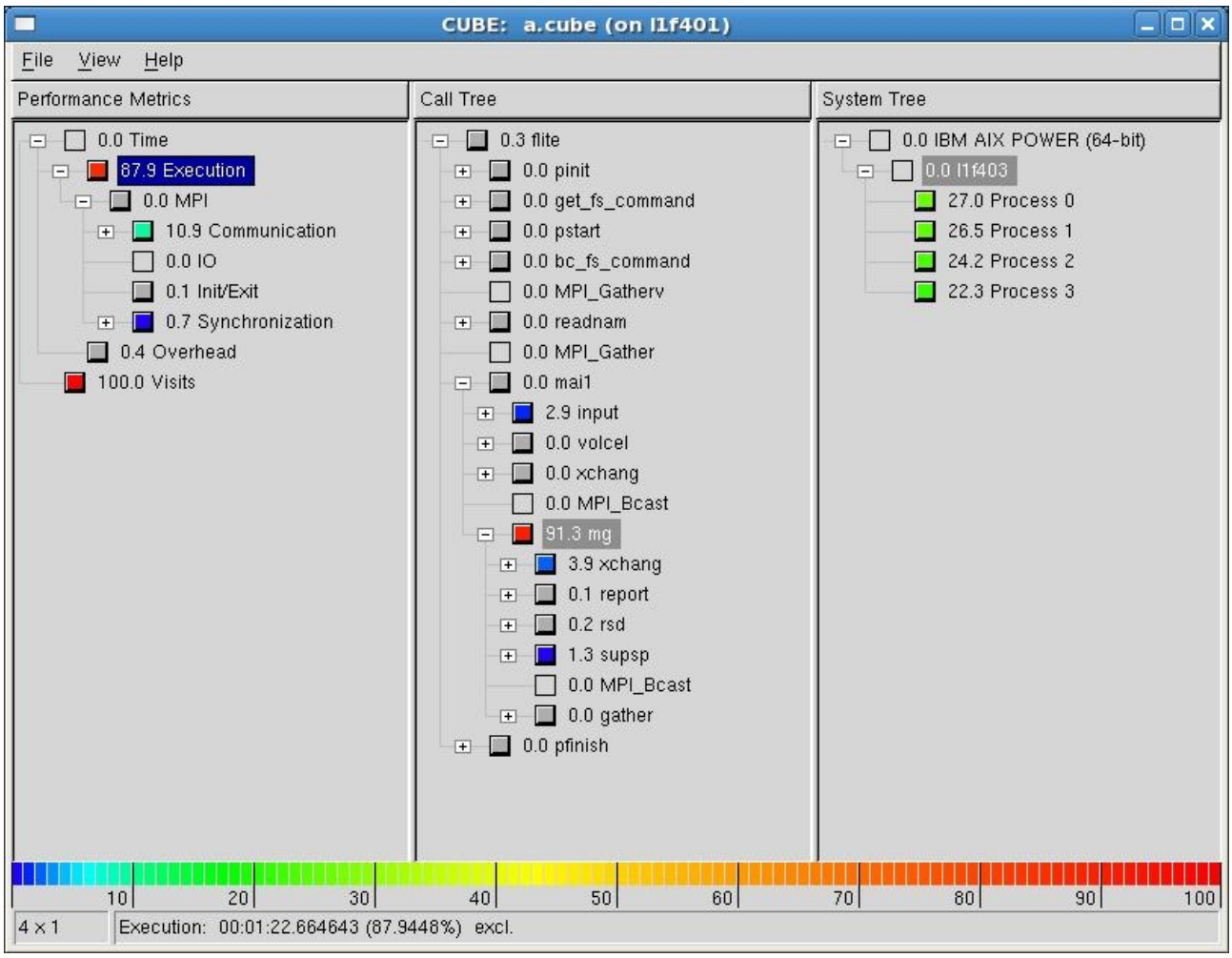


The three panes can be seen, and the overall numbers show that the execution took 94 seconds and that 208000 events were registered (the number of visits is normalised between 1 and 10, clicking on Visits displays the actual number on the colour bar at the bottom of the screen. As can be seen, the entries are expandable and a few levels of expansion gives the following screen:

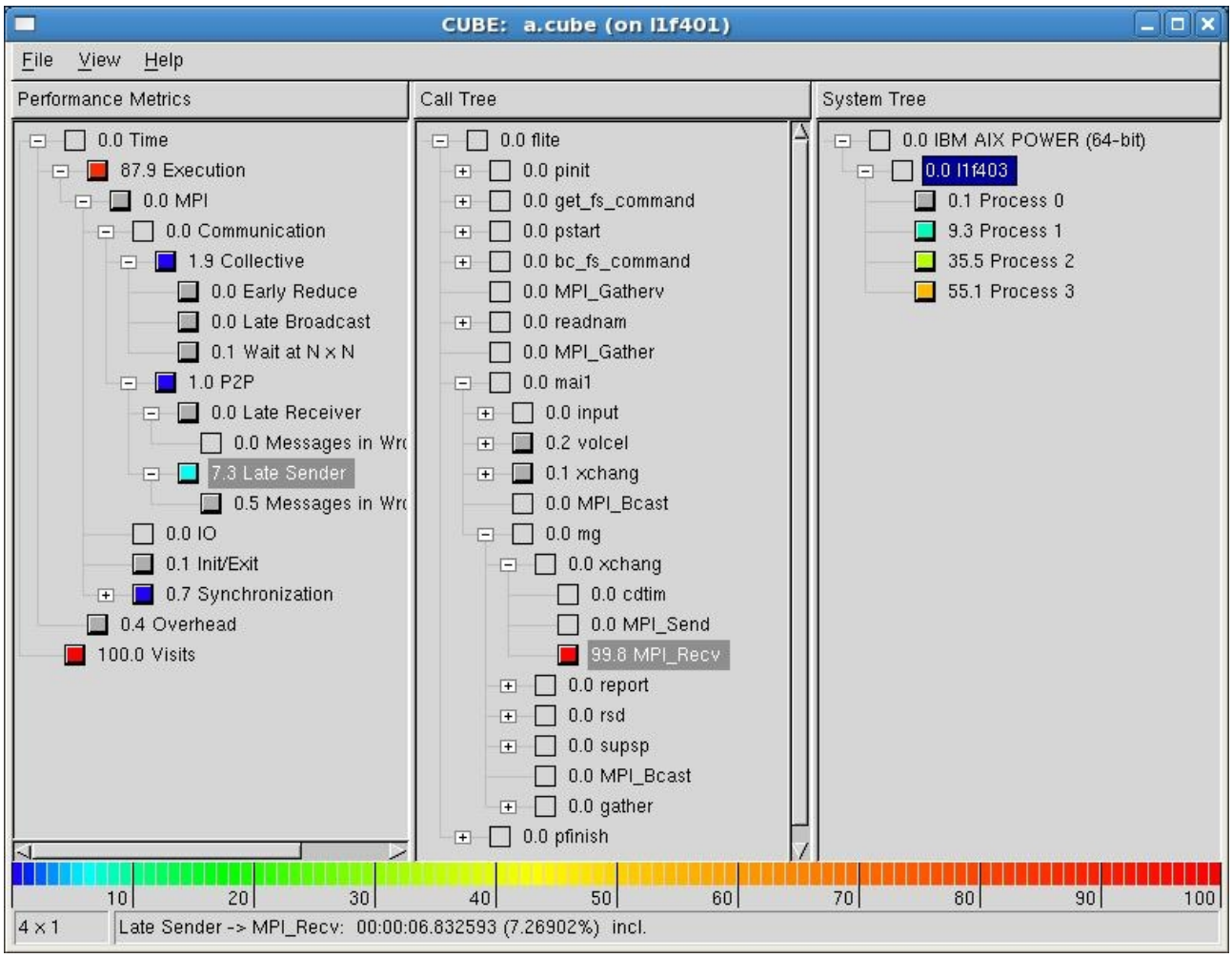


Because the Execution time entry is highlighted the numbers in the other panes relate to that – thus subroutine `mg` took the major share of the execution time – 75.5 seconds out of 82.66. We can instantly see by looking at the system tree that there is some load imbalance in the `mg` routine with process 3 taking over three seconds less than processes 0 and 1. Looking back to the Performance Metrics, the time in MPI calls is split between Communications and Synchronization with the former clearly the larger. Highlighting this shows that as expected the two major consumers of MPI time were `xchang` and `supsp`. It also shows one of the deficiencies of CUBE since the colours displayed still relate to the overall time, and since numbers are only displayed to 2 decimal places, the point is rapidly reached where all colouration is in dark blue and the numbers are all close to zero.

To overcome this we can switch to display relative percentages using the View menu. The display above then becomes



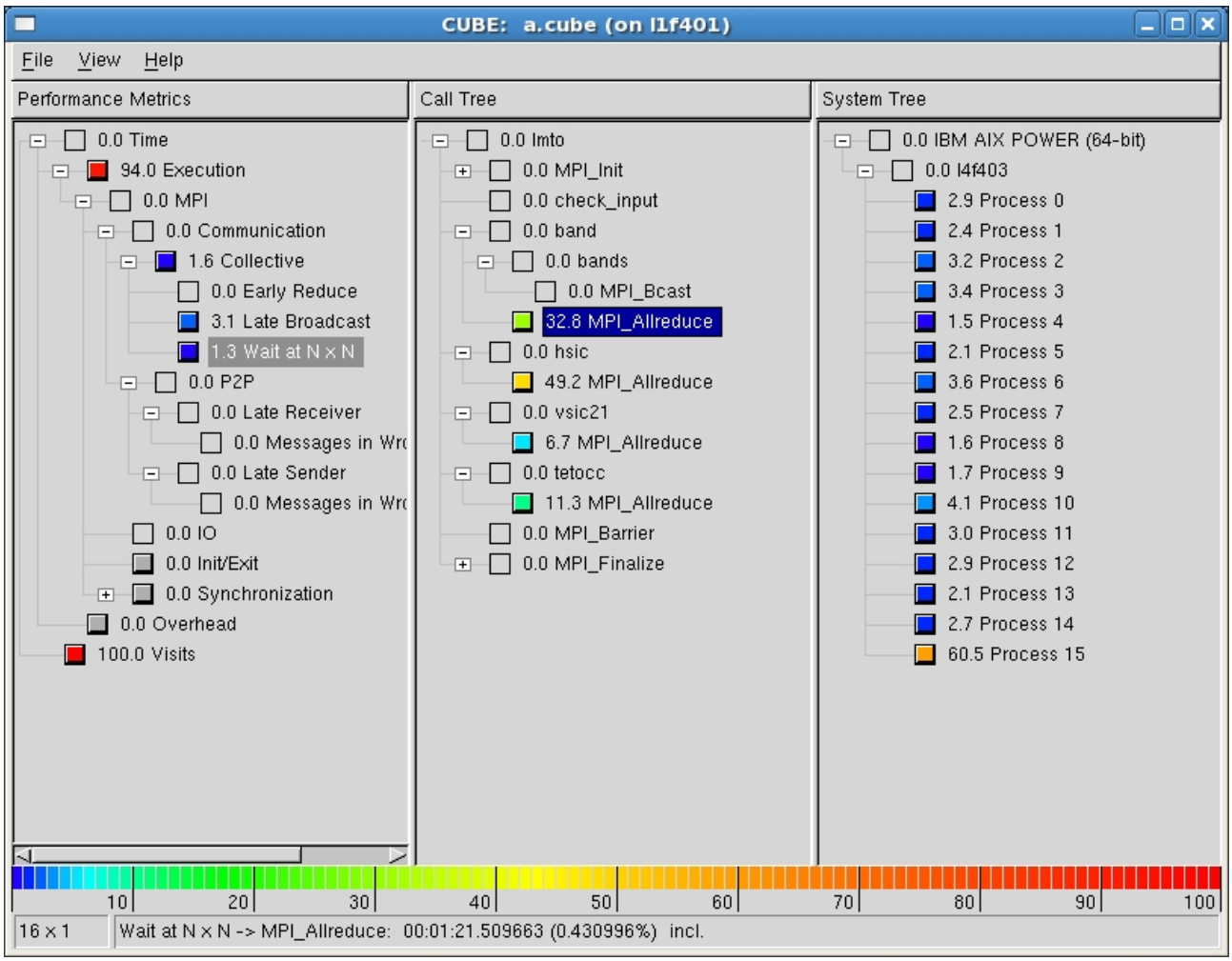
Note that in this mode the numbers displayed in each pane sum to 100% and are percentages of the highlighted entry to the left. Finally we look down at the MPI communications and see that in the point-to-point section there is a large amount of time caused by Late Senders.



These are all in MPI_Recv, as we know from earlier sections, but in this case we are easily able to identify the processes responsible as process 3 and, to a lesser extent, process 2.

Missing from this analysis, of course, is the message length information that both mpitrace, and fpmpi were able to provide.

A screendump from SIC_LMTO's data is interesting, in that it shows a large imbalance in one particular area. As was said in section 3, SIC-LMTO only uses MPI_Bcast and MPI_Allreduce, so all its MPI time is spent in collective operations. The performance problems arise from Late Broadcasts and from non-synchronisation waiting. This latter shows a major contribution to this from just one process, process 15 when called from subroutine `band`, and almost as much from `hsic`. In `band`, there are three calls to MPI_Allreduce, two of which are called once for each atom in the simulation for each invocation of `band`. These all perform summations over the Brillouin Zone as part of the self-consistency program. In `hsic`, which calculates quantities related to the self-interaction correction, it is the real space wavefunctions which are accumulated. Other calls to MPI_Allreduce from `vsic21` (the spin-dependent potentials) and `tetocc` do not show similar imbalance.



6. TAU

TAU (Tuning and Analysis Utilities) is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java, Python. As with KOJAK, TAU operates at several levels of user intervention, from automatic to hand instrumented. It makes extensive use of the Program Database Toolkit (PDT) [15] and has its own profile visualisation tool, ParaProf [16]. The TAU profile datafiles can also be exported to CUBE using a utility, tau2cube.

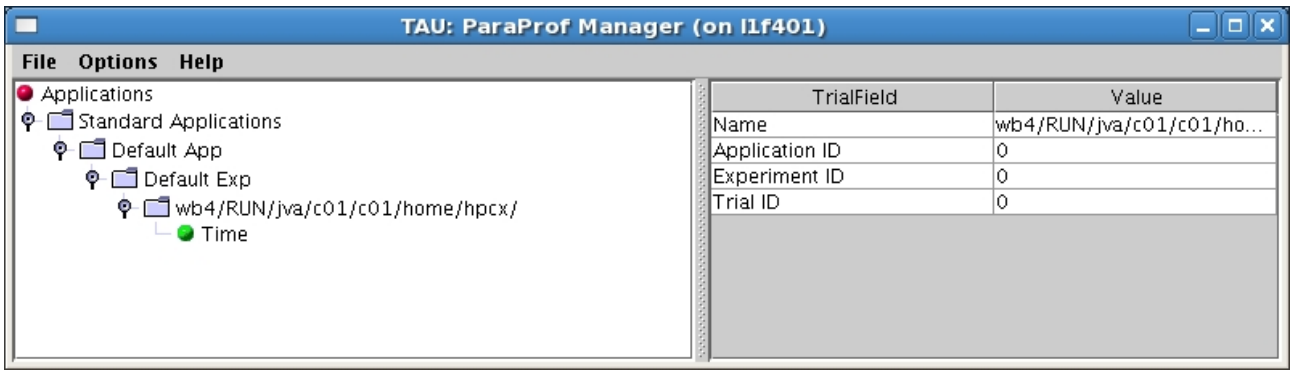
Our experience of installing TAU was that it required several components to be installed with the appropriate options in order that it would perform to what it promised. When that is sorted out, however, TAU is a powerful and flexible tool. Among these components is one that gives TAU an interface to VampirTrace [17], so that TAU traces can be displayed in VampirTrace using the Open Trace Format (OTF) and a similar interface to the Intel Trace Analyzer [18] (formerly Vampir) using the VTF3 format.

Since we were focussing on profiling MPI applications we took the path of installing a version configured to use MPI and PDT. This produced a makefile which can be included in the application makefile to define various entities such as the Fortran or C compiler TAU expects to use. As with KOJAK, TAU passes the code through an instrumentation phase, and this is automated by using the variable `$(TAU_COMPILER)` in the makefile. For example:

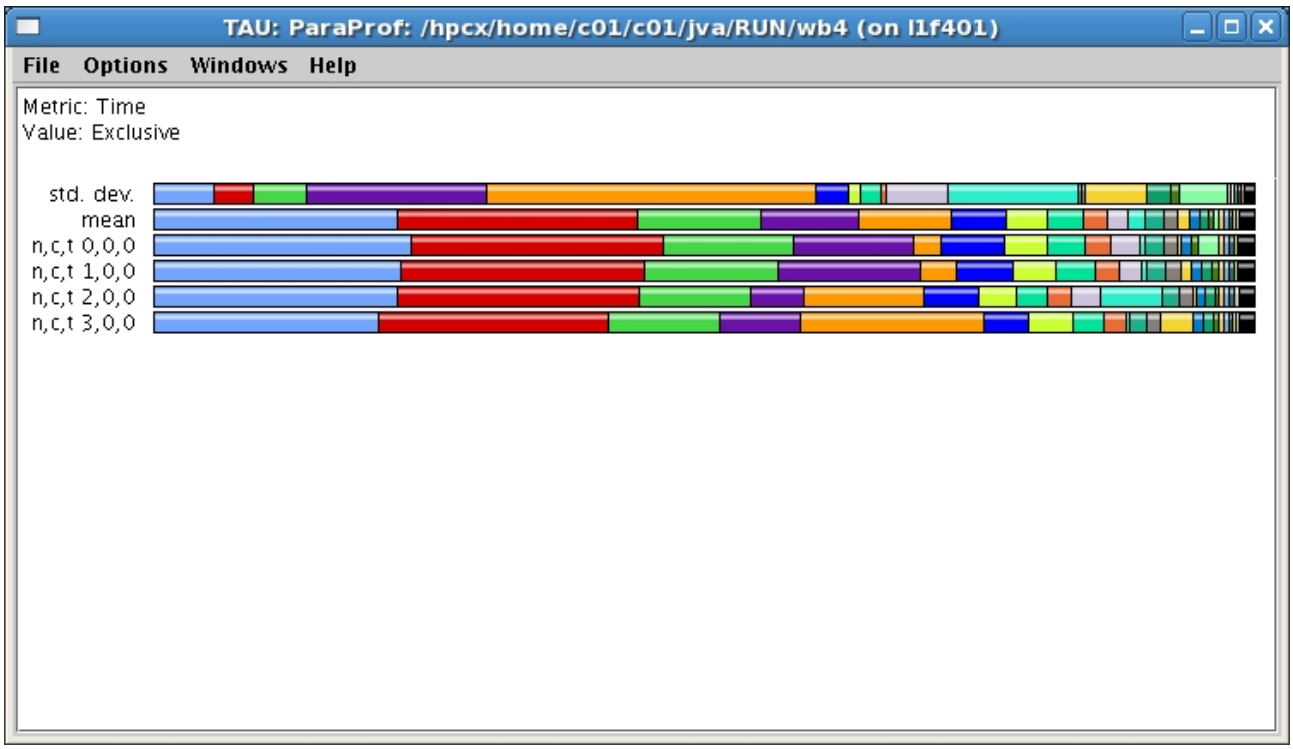
```
F90=$(TAU_COMPILER) $(OPTS) $(TAU_F90)
```

This also takes care of loading the correct MPI libraries and any other libraries TAU might need. A large set of examples are provided with the TAU distribution, the one called `taucompiler` illustrates this way of using TAU. When the code is run a set of files `profile.x.0.0` are produced where `x` is the process ID (the other suffices label the context and the thread respectively). These may be converted for viewing by CUBE or other systems, including Vampir, or viewed in ParaProf, TAU's own visual profile presenter.

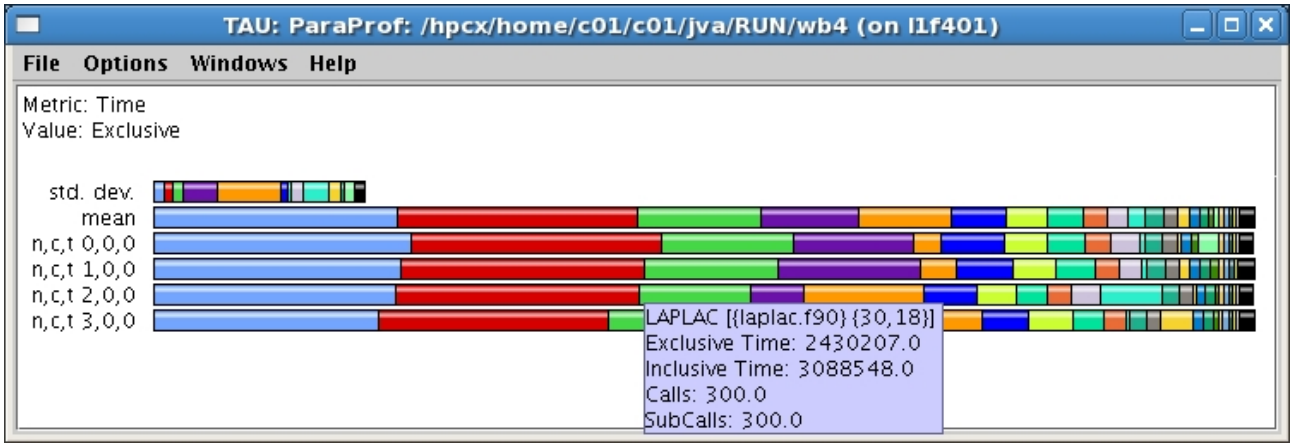
Launching ParaProf in the Flite3D directory finds the 4 profile files and opens two windows. The main window is the ParaProf manager which presents a tree view of the data (a very simple one in this case) which can be of use when more experiments have been run and one wishes to compare results.



The other window shows the time taken in all routines of the application. Hovering over a coloured bar will pop up a tooltip with details about which routine it refers to, where the source code can be found, the exclusive and inclusive time spent in the routine (exclusive time is the time spent in the routine itself, excluding time spent in subroutines called by it, inclusive time includes this), the number of times this routine is called and the number of calls to other routines that it makes.



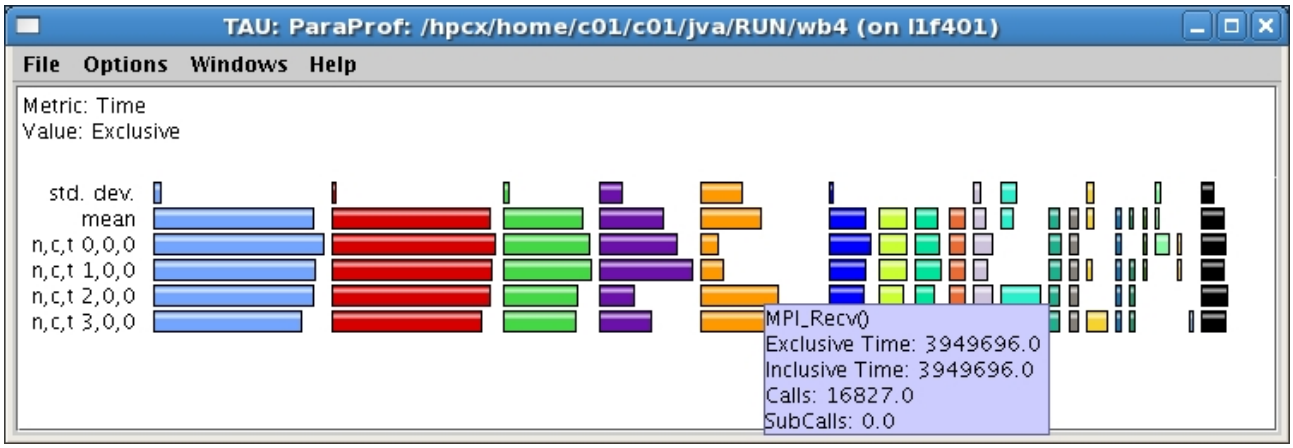
The time in this display is normalised on a per process basis, so all the bars are the same length. Switching to unnormalised display via the options menu is a good way of spotting load imbalancing. Here the overall computational load appears well balanced.



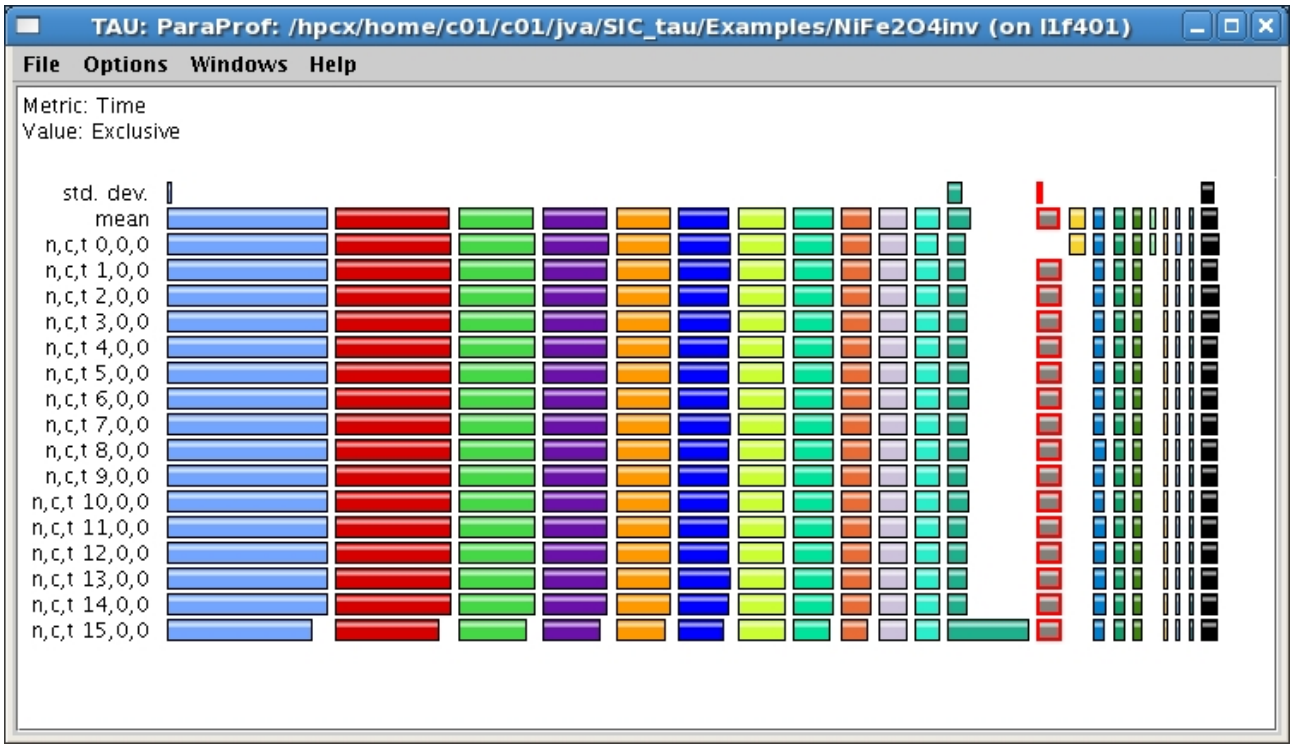
However, individual routines may still exhibit an imbalance, and turning off the stacking of bars can identify this.

As the tooltip shows, there is a large imbalance in MPI_Recv with process 3 spending roughly 7 times as long in the routine as process 0. This further localises the difference between MPI_Send and MPI_Recv we noted with mpitrace. Various other displays are available by right clicking on either the coloured bars or the process identifiers (n,c,t 0,0,0 etc.). This provides, for example, access to per thread or per routine barcharts, text or table displays of statistics.

A clear disadvantage with ParaProf is that TAU has profiled all routines in this case, so the MPI calls are to some extent swamped by the more time-consuming computationally intensive routines. Using a profiling method which allowed more control over which parts of the code were profiled would help this problem.



Running ParaProf on the data from SIC-LMTO clearly identifies the same imbalance in MPI_Allreduce with process 15 taking much longer than the others (this is the green bar twelfth from the left). From this screen it appears that the problem is one of load imbalance, with process 15 having less computational work to do (fewer **k**-points) as shown by the shorter bars for other routines, and then having to wait in MPI_Allreduce for the other processes to catch up. The routines with less work are all called approximately 10% fewer times in process 15 than in the other processes (this information is available from the tooltip by hovering the mouse over a bar). Again, a full message trace analysis of the sort provided by Vampir would confirm this.



7. Larger Numbers of processors

While some useful information can be gathered about MPI performance at the low number of processors used here, many programs will not exhibit communications problems worth investigating until they are run on much larger numbers of processors. With this in mind we tried the tools on Flite3d running on 64 processors. Since we used the same small grid, this is an artificial problem where communication swamps computation and is only of interest to indicate the usability of the

tools.

Using mpitrace the most obvious difference is that instead of producing 4 profiles, the code now produces 64, with an accompanying increase in the difficulty of interpreting the output across processors. The higher the processor count, the more relevant a graphical interface to the data becomes, though the detail mpitrace provides can be useful once a pattern or deviations from the pattern have been spotted. One useful detail is that in this implementation the logfiles can distinguish between MPI_Send and MPI_Recv which communicate within a node and MPI_Send_Ext (MPI_Recv_Ext) communicating between nodes.

The fpmpi version failed to run, citing insufficient memory during message passing initialization as a reason. This is particularly unfortunate as the process topology could well have been of interest in this case. The KOJAK version, on the other hand, ran to completion and produced an `a.elg` file, but `kanal` ground to a halt and eventually was timed out when trying to read it. It is worth commenting on the relative sizes of the files: each of the 64 `mpi_profile` files produced by mpitrace is about 5.5kb, a total of approximately 350 kb, while the `a.elg` file is 87Mb. Clearly `a.elg` contains more information when it is accessible, but how useful this is for our purposes can be judged by the 4-processor case where `a.elg` is 10Mb and the `a.cube` file which `kanal` derives from it to display in CUBE is 0.1Mb.

Finally, we were also able to run the TAU version on 64 processors which produced files of a similar size to mpitrace. As the number of processors increases the TAU windows become scrollable and the same information is available. In a very large number of processors, spotting anomalies would be difficult, but not impossible. TAU does not distinguish between intra- and extranodal communication.

We also tried to use the tools on another CFD code which, it was hoped, would exercise them at ~1000 processors. In this case the results were disappointing across the board. The mpitrace version failed to run with an invalid communicator. fpmpi ran but produced no output. KOJAK also ran, but the `kanal` stage aborted saying that messages had inconsistent timestamps. Surprisingly the most difficult of all was TAU, whose model of a Makefile did not fit with the Makefile this code used. It is probable that one of the other approaches possible with TAU would have yielded more positive results.

8. Conclusions

We have seen in this report how the four profiling tools presented can give insight into possible performance bottlenecks in MPI applications. They each have their strengths and weaknesses. mpitrace and fpmpi are text based while KOJAK and TAU use visual presentation. Each of these has its place; visual presentation can give rapid information on where in a code to look for problems, while the textual data may give a more quantitative feel for what is going on. TAU and mpitrace present their results process by process, though it is much easier to compare processes one against the other in TAU. The mpiprof facility of the mpitrace suite allows still further localisation by distinguishing between calls to the same MPI routine from different application routines.

By using its three parameter space of Performance Metrics, program location and process, KOJAK is potentially capable of presenting a large amount of useful information to the user. As with all of these systems, though, there is a steep learning curve to discover how best to elicit that information. In the

long run, there is no easy way, other than to sit with the profiler, display some data and look at what the presentation tells you about the code as you explore different options.

None of the tools are easy to use or foolproof enough to be applied automatically. As we saw, some tools failed on some codes or problems, and none of them was universally successful.

All four tools discovered much the same things about the applications we profiled, though interpreting them took different amounts of effort and some provided different details. The Intel Trace Analyzer and VampirTrace remain the gold standard for profiling MPI applications (though TAU should not be dismissed given its capacity to profile more widely). These four tools, though, are useful where they are available and where access to the Intel Trace Analyzer or Vampir may be restricted. Application programmers should consider familiarising themselves with several of these tools.

References

1. Sunderland A. Profiling Parallel Performance using Vampir and Paraver, HPCx Technical Report: http://www.hpcx.ac.uk/research/publications/HPCxTR0704.pdf
2. mpitrace libraries: http://www.hpcx.ac.uk/support/documentation/IBMdocuments/mpitrace
3. FPMPI-2 Fast Profiling library for MPI: http://www-unix.mcs.anl.gov/fpmpi/WWW/
4. KOJAK – Automatic Performance Analysis Toolset: http://www.fz-juelich.de/zam/kojak/
5. TAU Tuning and Analysis Utilities: http://www.cs.uoregon.edu/research/tau/home.php
6. Ashby J.V., SIC-LMTO – Benchmarking an Electronic Structure Code: http://www.cse.scitech.ac.uk/arc/sic-lmto.shtml
7. Emerson D.R. and Ashworth M., Parallelisation of Flite3D: http://www.cse.scitech.ac.uk/ceg/flite3d/flite3d.shtml
8. Hein J., Using the Hardware Performance Monitor Toolkit on HPCx, HPCx Technical Report: http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0307
9. EPILOG Event Processing, Investigating and LOGing: http://www.fz-juelich.de/zam/kojak/components/epilog/
10. OPARI OpenMP Pragma And Region Instrumentor: http://www.fz-juelich.de/zam/kojak/components/opari/
11. EARL Event Analysis and Recognition Library: http://www.fz-juelich.de/zam/kojak/components/earl/
12. EXPERT Extensible Performance Tool: http://www.fz-juelich.de/zam/kojak/components/expert/
13. CUBE Cube Uniform Behavioral Encoding: http://www.fz-juelich.de/zam/kojak/components/cube/
14. KOJAK usage document: http://www.fz-juelich.de/jsc/datapool/Kojak/USAGE.txt
15. PDT – Program Database Toolkit: http://www.cs.uoregon.edu/research/pdt/home.php
16. ParaProf – User's Manual: http://www.cs.uoregon.edu/research/tau/docs/paraprof/index.html

17. Vampir – Performance Optimization: http://www.vampir.eu/

18. Intel Trace Analyzer and Collector 7.0 for Linux:
http://www.intel.com/cd/software/products/asmo-na/eng/306321.htm