Profiling hybrid MPI+coarray miniapps

Anton Shterenlikht*, Sameer Shende†, Luis Cebamanos‡, Lee Margetts§ and Jose D. Arregui-Mena§
*Mech Eng Dept, The University of Bristol, Bristol BS8 1TR, UK, Email: mexas@bris.ac.uk
†Performance Research Lab, 5294 University of Oregon, Eugene, OR 97403-5294, USA, Email: sameer@cs.uoregon.edu
‡Edinburgh Parallel Computing Centre (EPCC), The University of Edinburgh, King's Buildings, Edinburgh EH9 3FD, UK,
Email: l.cebamanos@epcc.ed.ac.uk
§School of Mechanical, Aero and Civil Engineering, The University of Manchester, Manchester M13 9PL, UK,
Email: lee.margetts@manchester.ac.uk, jose.arregui-mena@manchester.ac.uk

***Abstract*—Coarrays are a native Fortran means for SPMD parallel programming, implementing a single-sided communications model. Coarray Fortran belongs to the PGAS class of parallel languages. Relatively few tools support profiling of coarray programs. Profiling and tracing of pure coarray programs and mixed coarray+MPI miniapps was done in this work on 2 platforms. TAU (Tuning and Analysis Utilities) toolkit 2.25.2 was used with the Intel compiler 16.0.2 and the Intel MPI libraries 4.1.0 on Red Hat Enterprise Linux 6 cluster. CrayPAT (Performance Analysis Tool) was used on Cray XC30 system with CCE (Cray Compiler Environment) 8.4.1. Four coarray programs, of progressively increasing complexity, are studied. TAU profiling results on 32 cores and CrayPAT profiling data on 7k cores show that our hybrid MPI+coarray codes are relatively well balanced. We show that the Intel compiler maps coarray remote operations and synchronisation routines onto MPI-2 RMA calls. For the Intel implementation we show that in iterative programs, where remote data access and image synchronisation are performed each iteration, performance is heavily dominated by `MPI_Win_unlock` routine. These results might open a possibility for Intel developers to improve and optimise their MPI based coarray implementation. Cray implementation of coarrays uses their proprietary communications library DMAPP. CrayPAT results for an MPI+coarray program on 7k cores show that runtime is heavily dominated by the user routines, with MPI routines in the second place. DMAPP coarray calls consume an insignificant fraction of runtime.**

I. Introduction

Coarrays are a native Fortran means for SPMD parallel programming [1]. Although coarrays (or co-arrrays, Co-Array Fortran (CAF), as they were originally known) have been used, particularly on Cray systems, as an extension, for nearly 20 years [2], they became part of the Fortran standard only in 2010 [3]. Coarray capabilities will be substantially expanded in the Fortran 2015 standard [4]. Coarrays offer simple syntax and portability for SPMD standard conforming Fortran programs. Coarrays can be added gradually to existing Fortran projects and can co-exist with other popular parallel technologies, primarily OpenMP and MPI [5]. At runtime a coarray program is replicated a certain number of times, and each copy of the executable (called an *image*) is executing asynchronously. The standard [3] uses very strict image ordering and synchronisation rules to ensure coarray integrity. As a result, a standard conforming coarray program should not deadlock or suffer from races.

However, coarray performance can vary significantly. This is partly because coarrays represent a very high level of abstraction, and Fortran compilers can use different transport libraries to map coarrays data and communications to hardware. Cray systems use DMAPP, the Intel implementation uses MPI and the OpenCoarrays implementation is designed to support MPI and GASNet [6].

In our previous work we successfully used proprietary CrayPAT tools to optimise performance of MPI+coarray miniapps on Cray systems [7]. Here we show the latest load balancing, profiling and tracing results for MPI+coarray miniapps on ARCHER, Cray XC30, the UK national supercomputer.

Recently TAU (Tuning and Analysis Utilities) was shown to support coarray programs [8], [9]. In this work we use TAU to profile and trace several coarray and MPI+coarray programs using the Intel Fortran compiler and the Intel MPI library.

II. Profiling and tracing with TAU

TAU[1] [10], is a popular open source set of tools for performance analysis, particularly on HPC systems. In this work TAU 2.25.2 was used.

TAU is often used together with the Program Database Toolkit (PDT), [2], which is a framework for analysing source code. However, PDT 3.22 does not yet support coarrays. Hence compiler based instrumentation was used in this work, which is enabled with TAU flag `-optCompInst`.

TAU can internally use external packages such as PAPI[3] and Score-P[4]. PAPI provides TAU access to low-level hardware performance counters to track events such as instructions executed, or level 1 and 2 data cache misses. Score-P provides an efficient implementation of callpath profiling and event tracing. TAU can generate profiles in the CUBEX format using Score-P's measurement substrate and use the CUBE or ParaProf tools to view the profiles. Using Score-P, TAU can also generate native OTF2 traces that may be visualized in the Vampir[5] commercial trace visualization tool. To use Score-P, TAU can be configured with the `-scorep=download` flag.

[1]https://www.cs.uoregon.edu/research/tau
[2]https://www.cs.uoregon.edu/research/pdt
[3]http://icl.cs.utk.edu/papi
[4]http://www.score-p.org
[5]http://www.vampir.eu

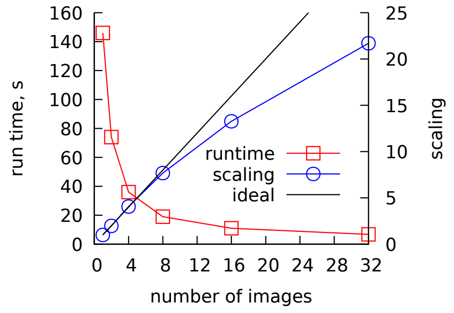


Fig. 1. Calculating π on 2 16-core nodes - scaling.

In this paper, we use TAU's native measurement library for profiling and tracing.

The University of Bristol BlueCrystal phase 3 system was used for this work[6]. Each node has a single 16-core 2.6 GHz SandyBridge CPU and 64GB RAM. Intel Cluster Studio XE, version 16.0.2, was used which uses the Intel MPI library 4.1.0. TAU was configured with

```
-mpi -c++=mpiicpc -cc=mpiicc \
-fortran=mpiifort
```

Exclusive TAU times are measured and reported in all cases.

Jumpshot-4, developed by the Argonne National Lab (ANL)[7], was used to view TAU traces.

A. Case studies

1) Calculation of π using Gregory-Leibniz series: This example is taken from the University of Bristol coarrays course[8], folder `examples/prof/tau/5pi`.

π can be calculated using the Gregory-Leibniz as follows:

$$\pi = 4 \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{2n-1} \qquad (1)$$

This is a classical parallel problem - all workers calculate their own partial sums in parallel. The total sum is calculated via a collective routine or by a master process. The series limit was set arbitrarily at 2^{35}. The key fragment is shown below.

```
real :: pi[*]
do i = this_image(), 2**35, num_images()
 pi = pi + (-1)**(i+1) / real( 2*i-1 )
end do
sync all ! all images synchronise here
if ( this_image() .eq. 1 ) then
  do i = 2, num_images()
    pi = pi + pi[i]
  end do
  pi = pi * 4.0
end if
```

[6]http://www.acrc.bris.ac.uk

[7]http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/index.htm

[8]http://coarrays.sourceforge.net

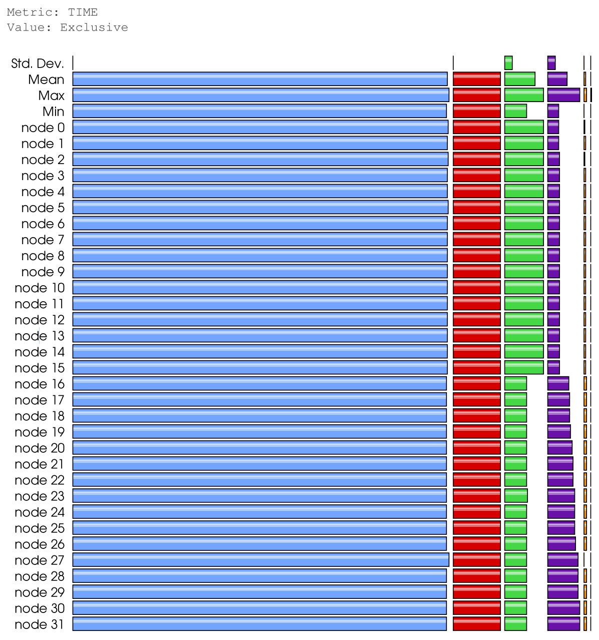
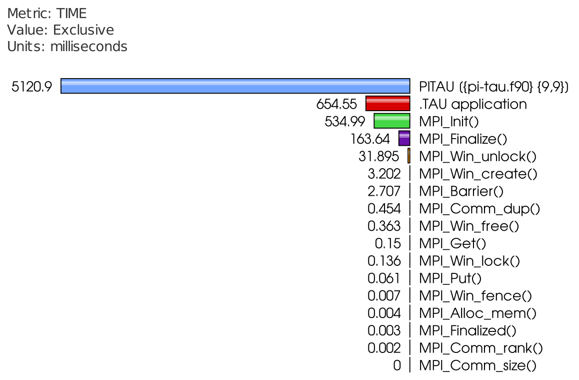


Fig. 2. Calculating π with 32 images - profiling. TAU 'node' means MPI process.



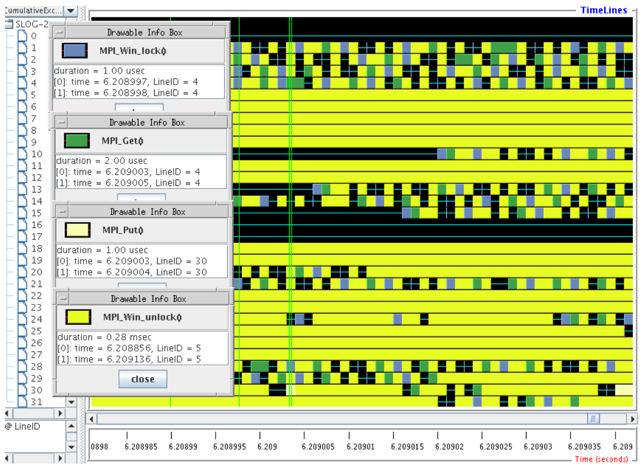
```
Metric: TIME
Value: Exclusive
Units: milliseconds

5120.9   PITAU ({pi-tau.f90} {9,9})
654.55   .TAU application
534.99   MPI_Init()
163.64   MPI_Finalize()
31.895   MPI_Win_unlock()
3.202    MPI_Win_create()
2.707    MPI_Barrier()
0.454    MPI_Comm_dup()
0.363    MPI_Win_free()
0.15     MPI_Get()
0.136    MPI_Win_lock()
0.061    MPI_Put()
0.007    MPI_Win_fence()
0.004    MPI_Alloc_mem()
0.003    MPI_Finalized()
0.002    MPI_Comm_rank()
0        MPI_Comm_size()
```

Fig. 3. Calculating π - single image profile.

Coarray collectives are not yet in the Fortran standard. They are described in [4] and will become a part of Fortran 2015 standard. Cray and OpenCoarrays already support coarray collectives. Intel Fortran 16 does not. Hence image 1 is calculating the total sum by pulling partial sums from all other images with `pi = pi + pi[i]`.

Fig. 1 shows scaling of this code on two nodes ($2 \times 16 = 32$ cores) with the Intel `-fast` optimisation flag. In all profiling and scaling runs we used a single MPI process or a single coarray image per core. So in Fig. 1 a maximum of 32 images was used on 32 cores.

Fig. 2 shows an even spread of load over all nodes, sorted by exclusive time. Note that TAU shows MPI processes starting from 0 (nodes in TAU terminology), whereas Fortran images always start from 1. So MPI process n corresponds



Drawable Info Box
MPI_Win_lock()
duration = 1.00 usec
[0]: time = 6.208997, LineID = 4
[1]: time = 6.208998, LineID = 4

Drawable Info Box
MPI_Get()
duration = 2.00 usec
[0]: time = 6.209003, LineID = 4
[1]: time = 6.209005, LineID = 4

Drawable Info Box
MPI_Put()
duration = 1.00 usec
[0]: time = 6.209003, LineID = 30
[1]: time = 6.209004, LineID = 30

Drawable Info Box
MPI_Win_unlock()
duration = 0.28 msec
[0]: time = 6.208856, LineID = 5
[1]: time = 6.209136, LineID = 5

Fig. 4. Calculating π with 32 images - a trace fragment.

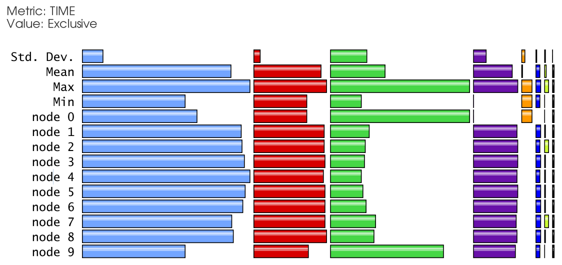


Fig. 6. Profile of `coback1-tau` on 10 images.

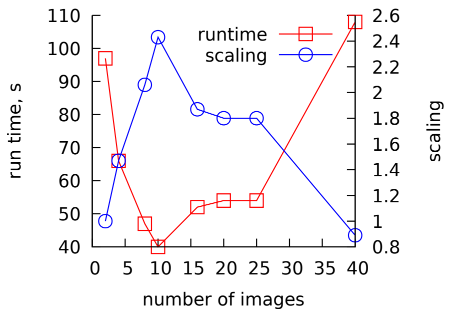


Fig. 5. Scaling of the Laplacian code `coback1-tau` on 2 16-core nodes.

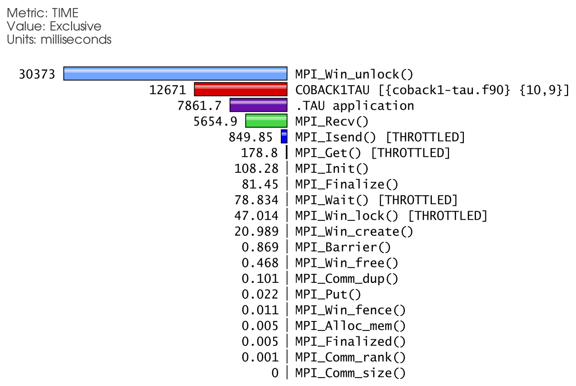


Fig. 7. Profile of `coback1-tau` on a single image.

to image $n + 1$. `MPI_Init` (green) and `MPI_Finalize` (purple) times differ between MPI processes 0-15 and 16-31, highlighting the boundary between the physical computers in the cluster. Time spent in other MPI routines is negligible.

Fig. 3 shows exclusive time profile on image 14. It is clear that Intel implementation of coarrays uses remote memory access (RMA) one-sided communications of MPI-2, where a typical sequence of calls for a lock/unlock synchronisation is `MPI_Win_create`, `MPI_Win_lock`, `MPI_Put`, `MPI_Get`, `MPI_Win_unlock` and `MPI_Win_free` [11]. Note that Fig. 3 shows that of these calls, nearly 90% of time is spent in `MPI_Win_unlock`. A fragment of the trace for this program, taken after `sync all`, is shown in Fig. 4. It highlights the dominance of `MPI_Win_unlock` calls.

2) A Laplacian solver: This example is taken from the University of Bristol coarrays course[9], folder `examples/prof/tau/9laplace`.

Program `coback1-tau.f90` iteratively reconstructs a 2D array `p` from previously calculated 2D edge array `edge`. Halo exchange and `sync images` synchronisation is used every iteration. The key fragment is show below.

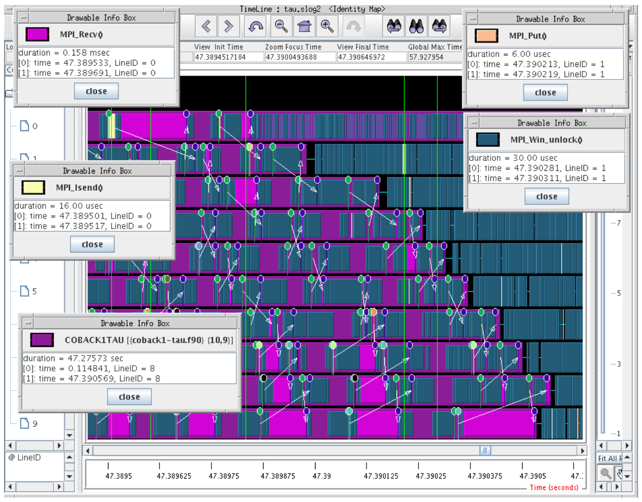
```
img = this_image() ; nimgs = num_images()
outer: do iter = 1, niter
 if (img.ne.1) op(:,0)=op(:,size2)[img-1]
 if (img .ne. nimgs) op(:, size2+1) =  &
      op(:, 1)[img+1]
 do j = 1, size2
 do i = 1, width
   p(i,j) =  &
     0.25 * (op(i-1,j) + op(i+1,j) + &
       op(i,j-1) + op(i,j+1) - edge(i,j))
 end do
 end do
 op = p
 if ( img .ne. 1 ) sync images(img-1)
 if ( img .ne. nimgs ) sync images(img+1)
end do outer
```

[9]http://coarrays.sourceforge.net

Fig. 5 shows scaling of program `coback1-tau` on two 16-core nodes. Best performance is achieved on 10 images. Profiling results with 10 images are shown in Figs. 6 and 7.

`MPI_Win_unlock` now dominates the run time, and the program itself, `cobacktau`, accounts for less than half of time. Significant load imbalance is seen in `MPI_Recv`.

Fig. 8 shows the data transfer pattern and dominance of `MPI_Win_unlock` calls in the program at the exit from the `outer` loop. Note also an emerging pattern in remote calls, when no pattern is present in the source code. The pair-wise synchronisation between the neighbouring images in the source code can be done in any order, i.e. staring at any arbitrary image. It seems the implementation chose to process all images in order, starting from image 1.


TimeLine : tau.slog2 <Identity Map>

View Init Time	Zoom Focus Time	View Final Time	Global Max Time
47.3894517184	47.3900493688	47.390646972	57.927954

Drawable Info Box
MPI_Recv()
duration = 0.158 msec
[0]: time = 47.389533, LineID = 0
[1]: time = 47.389691, LineID = 0

Drawable Info Box
MPI_Put()
duration = 6.00 usec
[0]: time = 47.390213, LineID = 1
[1]: time = 47.390219, LineID = 1

Drawable Info Box
MPI_Win_unlock()
duration = 30.00 usec
[0]: time = 47.390281, LineID = 1
[1]: time = 47.390311, LineID = 1

Drawable Info Box
MPI_Isend()
duration = 16.00 usec
[0]: time = 47.389501, LineID = 0
[1]: time = 47.389517, LineID = 0

Drawable Info Box
COBACK1TAU [{coback1-tau.f90} {10,9}]
duration = 47.27573 sec
[0]: time = 0.114841, LineID = 8
[1]: time = 47.390569, LineID = 8

Fig. 8. A fragment of the trace of `coback1-tau` on 10 images.

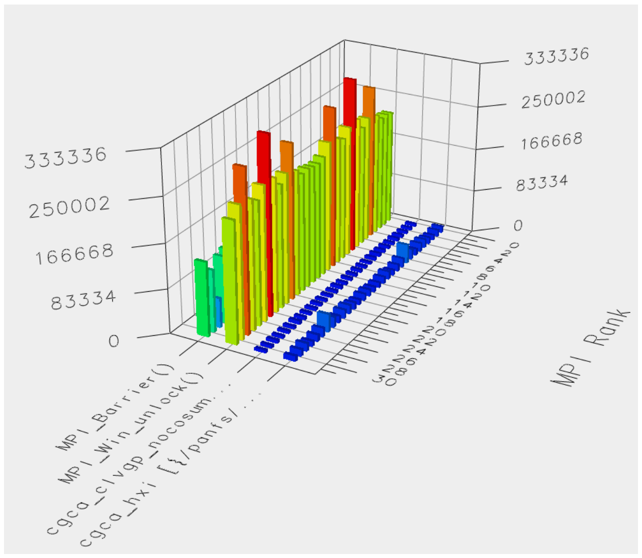


Fig. 9. Profile of CGPACK program `testABW`, showing only functions accounting for more than 1% of total exclusive time.

3) Cellular automata microstructure simulation code: CGPACK,[10] is a BSD licensed Fortran coarray library for microstructure simulation [12], [13]. It uses a cellular automata approach, where a 3D space is represented as a structured grid of cells. Cell states are updated iteratively. At every iteration the state of each cell is determined by the states of its neighbouring cells and, possibly, by a superimposed field, such as temperature or strain.

The CGPACK distribution includes a number of test programs, e.g. `tests/testABW.f90`, studied here. This program simulates formation and cleavage fracture of polycrystalline microstructure. It calls multiple routines to manipulate an allocatable integer coarray:

```
integer,allocatable :: space(:,:,:,:)[:,:,:]
```

[10]http://cgpack.sourceforge.net

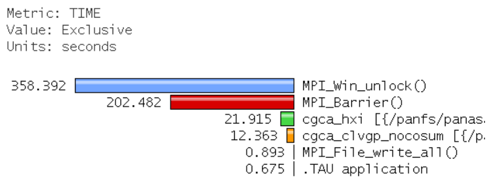
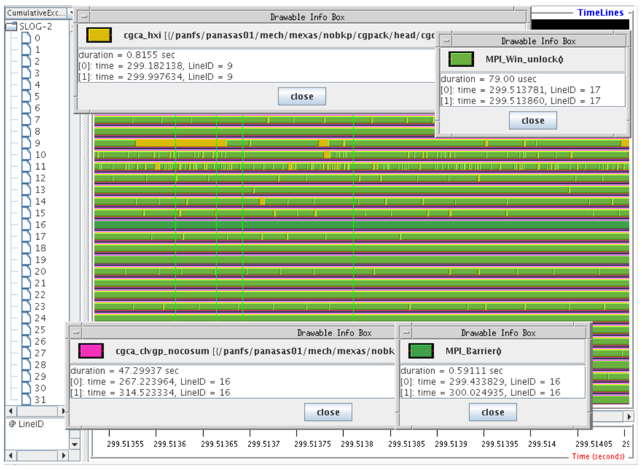

```
Metric: TIME
Value: Exclusive
Units: seconds
```

Fig. 10. Profile of `testABW` on a single image.


Drawable Info Box
cgca_hxi [{/panfs/panasas01/mech/mexas/nobkp/cgpack/head/cgc
duration = 0.8155 sec
[0]: time = 299.182138, LineID = 9
[1]: time = 299.997634, LineID = 9

Drawable Info Box
MPI_Win_unlock()
duration = 79.00 usec
[0]: time = 299.513781, LineID = 17
[1]: time = 299.513860, LineID = 17

Drawable Info Box
cgca_clvgp_nocosum [{/panfs/panasas01/mech/mexas/nobk
duration = 47.29937 sec
[0]: time = 267.223964, LineID = 16
[1]: time = 314.523334, LineID = 16

Drawable Info Box
MPI_Barrier()
duration = 0.59111 sec
[0]: time = 299.433829, LineID = 16
[1]: time = 300.024935, LineID = 16

Fig. 11. A fragment of trace of `testABW`.

The `-O2` optimisation flag was used. Figs. 9 and 10 show that the time is dominated by `MPI_Win_unlock` even more than in the previous examples. `MPI_Barrier` is in the second place. Only 2 CGPACK routines, the halo exchange, `cgca_hxi`, and the cleavage fracture propagation, `cgca_clvgp_nocosum`, exceed the threshold of 1% of the total time. Fig. 9 also shows good load balance in the user routines and some imbalance in `MPI_Win_unlock`.

Fig. 11 shows the trace of `testABW`, while the program is inside the halo exchange routine `cgca_hxi`. `MPI_Win_unlock` is run on all images except 17, which is in `MPI_Barrier`. The source code points to no obvious explanation of why only a single image would call a global barrier at this point.

4) Multiscale coarray+MPI fracture model: ParaFEM,[11] is a BSD licensed[12] highly scalable Fortran MPI finite element library [14]. It has recently been used in nuclear fusion research [15] and biomechanics [16].

By linking ParaFEM with CGPACK a multi-scale cellular automata finite element (CAFE) framework was created [12]. In the CAFE approach the structural scale is represented with FE and material microstructure evolution is modelled with CA. A concurrent two-way information transfer is established between the FE and the CA layers [17], [18].

ParaFEM includes several CAFE miniapps, as a set of developer programs, under `src/programs/dev/xx14`. In this work we profile `xx14std.f90`. Both ParaFEM and

[11]http://parafem.org.uk
[12]https://sourceforge.net/projects/parafem

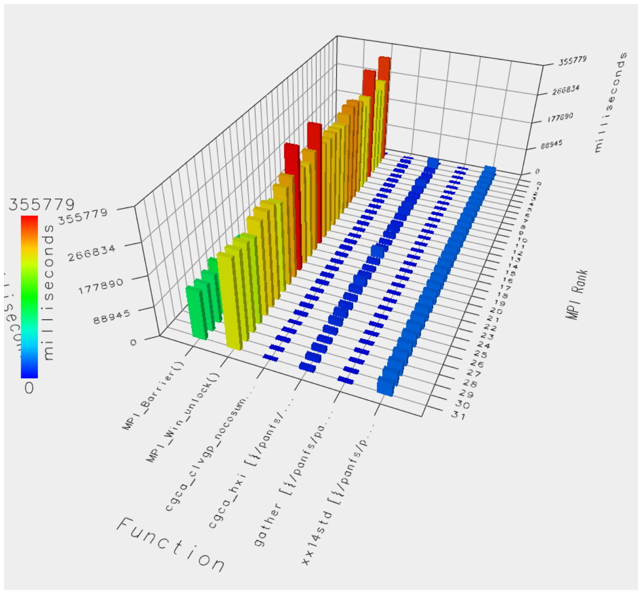


Fig. 12. 3D profiling bar chart of program `xx14std`, coarray+MPI, on 2 nodes with 32 images.

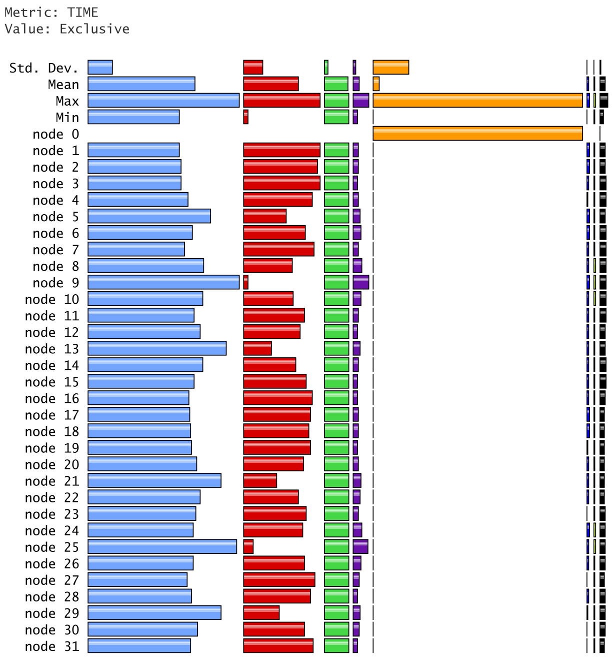
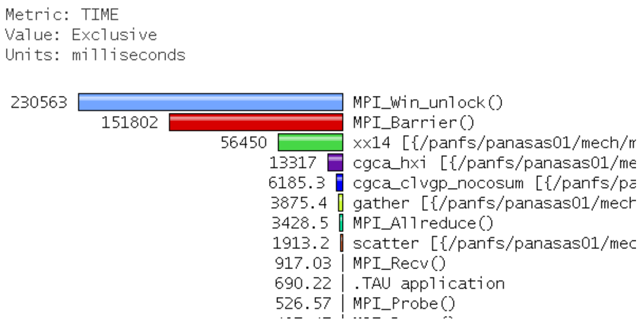


Fig. 13. Profile of program `xx14std`, coarray+MPI, on 2 nodes with 32 images.

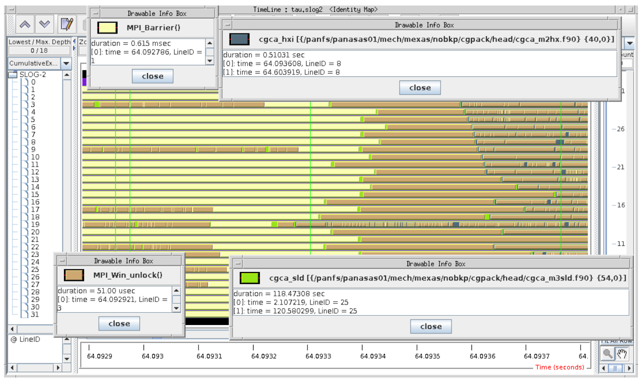
CGPACK libraries were instrumented with TAU. The `-O2` optimisation flag was used. Profiling was done on 2 16-core nodes with 32 images.

Figs. 12-14 show the profiling results for `xx14std`. The observations are consistent with the previous two examples. The load is well balanced across all images. However, `MPI_Win_unlock` dominates the run time and `MPI_Barrier` is in the second place. Only the program



```
Metric: TIME
Value: Exclusive
Units: milliseconds

230563  MPI_Win_unlock()
151802  MPI_Barrier()
 56450  xx14 [{/panfs/panasas01/mech/m
 13317  cgca_hxi [{/panfs/panasas01/me
6185.3  cgca_clvgp_nocosum [{/panfs/pa
3875.4  gather [{/panfs/panasas01/mech
3428.5  MPI_Allreduce()
1913.2  scatter [{/panfs/panasas01/mec
917.03  MPI_Recv()
690.22  .TAU application
526.57  MPI_Probe()
```

Fig. 14. Profile of program `xx14std`, coarray+MPI, on a single image.



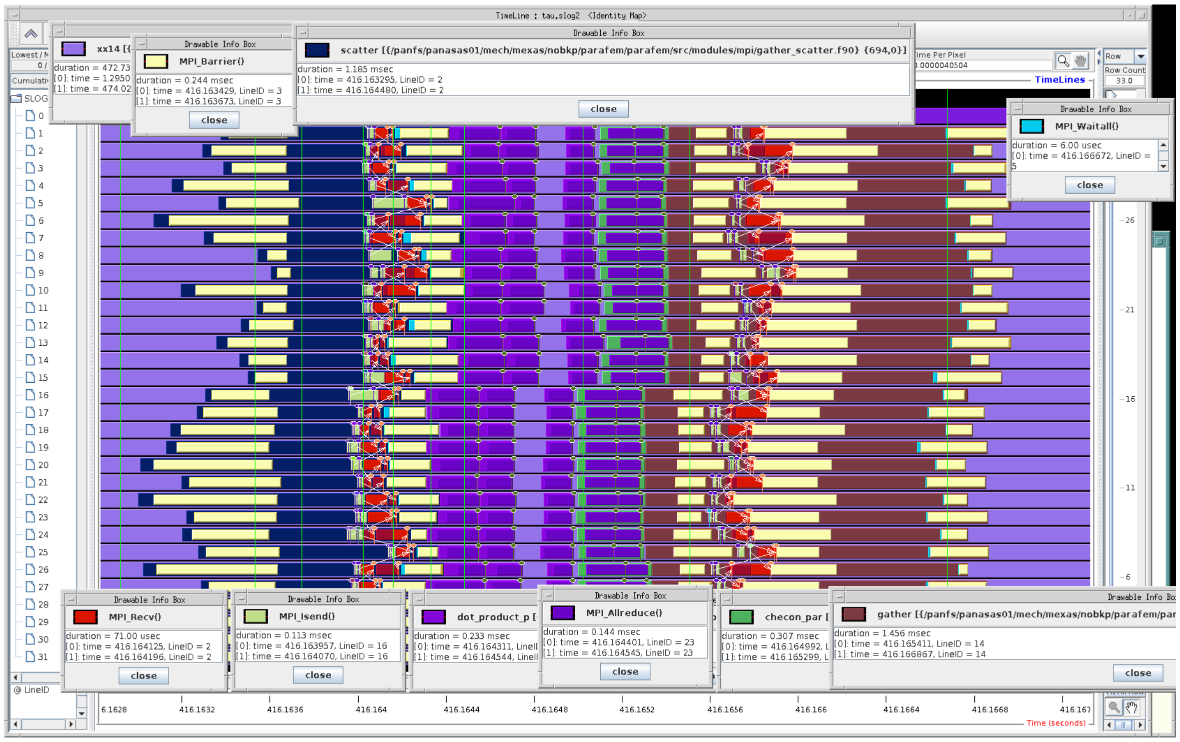
```
TimeLine : tau.slog2  <Identity Map>

Drawable Info Box
MPI_Barrier()
duration = 0.615 msec
[0]: time = 64.092786, LineID = 1
close

Drawable Info Box
cgca_hxi [{/panfs/panasas01/mech/mexas/nobkp/cgpack/head/cgca_m2hx.f90} {40,0}]
duration = 0.51031 sec
[0]: time = 64.093608, LineID = 8
[1]: time = 64.603919, LineID = 8
close

Drawable Info Box
MPI_Win_unlock()
duration = 51.00 usec
[0]: time = 64.092921, LineID = 3
close

Drawable Info Box
cgca_sld [{/panfs/panasas01/mech/mexas/nobkp/cgpack/head/cgca_m3sld.f90} {54,0}]
duration = 118.47308 sec
[0]: time = 2.107219, LineID = 25
[1]: time = 120.580299, LineID = 25
close
```

Fig. 15. A fragment of the trace of coarray+MPI program `xx14std`, inside CGPACK routine `cgca_sld`.

itself, `xx14std`, the halo exchange routine, `cgca_hxi`, the MPI FE routine, `gather`, and the fracture propagation routine, `cgca_clvgp_nocosum`, exceed the threshold of 1% of the total time.

Note that the Intel coarray implementation includes task `caflaunch`, which is assigned a rank 0 (orange bar in Fig. 13). In TAU 2.25.2, when compiler instrumentation is used, as in the this work, `caflaunch` is instrumented too. This creates a problem that both the `caflaunch` thread and the coarray program thread write to the profiling file on MPI process 0. Because `caflaunch` persists until the program exits, this process overwrites all program data on MPI process 0, so only `caflaunch` is seemingly present there, as seen in Fig. 13. The TAU team has created a fix which permits MPI ranks that have called `MPI_Init` to write profiles and traces to disk. This prevents the caf launcher task from interfering with the rest of the MPI ranks. This is key to supporting coarrays in performance evaluation tools.

Traces of `xx14std` show clear differences between those MPI calls which are included directly in the ParaFEM library, and the MPI calls into which the Intel compiler translated the CGPACK coarray remote operations. Fig. 15 shows a typical fragment of the trace of `xx14std`, where CGPACK coarray routines are executed. No communication pattern or structure can be seen. `MPI_Win_unlock` and `MPI_Barrier` are executed from CGPACK halo exchange routine `cgca_hxi`, which is called from `cgca_sld`. In contrast, Fig. 16 shows



```
TimeLine : tau.slog2 <Identity Map>

xx14 [{
duration = 472.73
[0]: time = 1.2950
[1]: time = 474.02

Drawable Info Box
MPI_Barrier()
duration = 0.244 msec
[0]: time = 416.163429, LineID = 3
[1]: time = 416.163673, LineID = 3

Drawable Info Box
scatter [{/panfs/panasas01/mech/mexas/nobkp/parafem/parafem/src/modules/mpi/gather_scatter.f90} {694,0}]
duration = 1.185 msec
[0]: time = 416.163295, LineID = 2
[1]: time = 416.164480, LineID = 2

Drawable Info Box
MPI_Waitall()
duration = 6.00 usec
[0]: time = 416.166672, LineID =
5

Drawable Info Box
MPI_Recv()
duration = 71.00 usec
[0]: time = 416.164125, LineID = 2
[1]: time = 416.164196, LineID = 2

Drawable Info Box
MPI_Isend()
duration = 0.113 msec
[0]: time = 416.163957, LineID = 16
[1]: time = 416.164070, LineID = 16

dot_product_p [
duration = 0.233 msec
[0]: time = 416.164311, LineI
[1]: time = 416.164544, LineI

Drawable Info Box
MPI_Allreduce()
duration = 0.144 msec
[0]: time = 416.164401, LineID = 23
[1]: time = 416.164545, LineID = 23

checon_par [
duration = 0.307 msec
[0]: time = 416.164992, L
[1]: time = 416.165299, L

Drawable Info Box
gather [{/panfs/panasas01/mech/mexas/nobkp/parafem/par
duration = 1.456 msec
[0]: time = 416.165411, LineID = 14
[1]: time = 416.166867, LineID = 14
```

Fig. 16. A fragment of the trace of program `xx14std`, showing MPI routines included directly in the ParaFEM library.

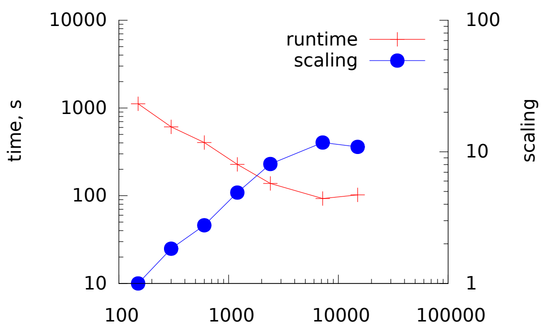


Fig. 17. Runtime and scaling of `xx14`, coarray+MPI, up to 15,000 cores on a Cray XC30.

a very well structured MPI communication pattern translated directly from the MPI calls in the ParaFEM library.

III. Cray Systems

The Cray Performance Analysis Tool (CrayPAT) is a powerful framework for analysing parallel applications' performance on Cray supercomputers. It can provide very detailed information on the timing and performance of individual application procedures, directly incorporating information from the raw hardware performance counters available on processors.

CrayPAT has two modes of operation,- sampling and tracing. Sampling takes regular snapshots of the application, recording which routine the application was in. This can provide a good overview of the important routines in an application without interfering with the run time, however it has the potential to miss smaller functions and cannot provide the more detailed information.

Tracing involves instrumenting each subroutine with additional instructions that can record this extra information when they enter and exit. This approach ensures full capture of information, but can result in high overheads, especially where individual functions and subroutines are very small. Furthermore, it can generate very large amounts of data which become difficult to process and visualise.

CrayPAT supports Fortran, C, C++, UPC, MPI, Coarray Fortran, OpenMP, Pthreads and SHMEM.

The general workflow to profile an application with CrayPAT involves 4 steps:

1) Compilation and normal execution of the application
2) Sampling (or tracing) using `pat_build`
3) Execution of the new generated executable
4) Visualization of profiling report using `pat_report`

The ParaFEM + CGPACK miniapps have been profiled and traced with CrayPAT on a Cray XC30 system (ARCHER). In this work we present results of profiling `xx14.f90`. This miniapp uses coarray collectives `CO_SUM` and `CO_MAX`, which are described in TS18508 [4] and will be included in the next revision of the Fortran standard, Fortran 2015. At the time of writing, coarray collectives are available on Cray systems as extension to the standard.

Apart from the use of coarray collectives, `xx14.f90` is identical to `xx14std.f90`, profiled in Sec. II-A4. `xx14.f90` calls routine `cgca_clvgp` that calls `CO_SUM` coarray collective, whereas `xx14std.f90` calls routine

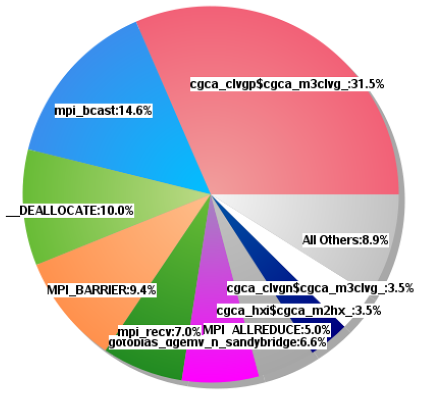


Fig. 18. Profile function distribution of program `xx14`, coarray+MPI at 7200 cores on Cray XC30.

`cgca_clvgp_nocosum` that implements a global sum in a user code.

Fig. 17 shows the scaling of `xx14.f90` on 15000 cores of the mentioned XC30 system. It is clear that this miniapp scales well to 7000 cores at which point the scalability drops dramatically.

`cgca_clvgp` and `cgca_clvgp_nocosum` are supposed to be the most computationally expensive routines in programs `xx14` and `xx14std` respectively. Both routines have a triple nested loop over the first 3 dimensions of `space` coarray.

Profiling results of the `xx14` miniapp on 7200 are shown in Figs. 18 and 19. It can clearly be seen that `cgca_clvgp` is indeed the most computationally expensive routine of the miniapp, taking over 30% of the total time. The second most time consuming routine, using 14% of the total time, is `MPI_BCAST` which belongs to the ParaFEM code.

This result is in stark contrast to `xx14std` profiling data, shown in Figs. 9 and 10, which show that `cgca_clvgp_nocosum` takes just over 1% of runtime.

CrayPAT is also able to show profiling information by separating code functions into different groups. Fig. 19 shows the results from sampling the miniapp using 7200 cores with 1 MPI process and 1 coarray image per core. `USER` functions are those defined by the miniapp, `MPI` functions contain the time spent in MPI library functions and `ETC` functions are generally library or miscellaneous functions.

One of the main obstacles to scaling applications out to large numbers of parallel tasks is load imbalance. Fig. 20 shows the whole program activity over processing elements (PEs) 6363 to 7199, where the time taken by classes of functions, e.g. User, Collectives, Synchronisation, Data transfer, etc. is shown as a percentage of the total time. This diagram is a good indicator of the load balance present in the whole application. A significant load imbalance in the user functions in `xx14` can be seen in Fig. 20.

Although CrayPAT supports coarrays, a small number of issues were found when profiling the MPI/coarray miniapps. These issues have been already reported to Cray developers for further investigation in upcoming releases. It was

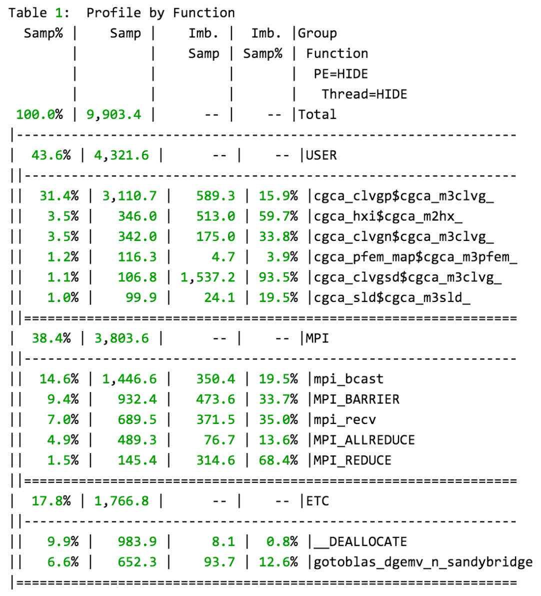

```
Table 1:  Profile by Function
  Samp% |    Samp |    Imb. |  Imb. |Group
        |         |    Samp | Samp% | Function
        |         |         |       |  PE=HIDE
        |         |         |       |   Thread=HIDE
 100.0% | 9,903.4 |      -- |    -- |Total
|-----------------------------------------------------------------
|  43.6% | 4,321.6 |      -- |    -- |USER
||----------------------------------------------------------------
||  31.4% | 3,110.7 |   589.3 | 15.9% |cgca_clvgp$cgca_m3clvg_
||   3.5% |   346.0 |   513.0 | 59.7% |cgca_hxi$cgca_m2hx_
||   3.5% |   342.0 |   175.0 | 33.8% |cgca_clvgn$cgca_m3clvg_
||   1.2% |   116.3 |     4.7 |  3.9% |cgca_pfem_map$cgca_m3pfem_
||   1.1% |   106.8 | 1,537.2 | 93.5% |cgca_clvgsd$cgca_m3clvg_
||   1.0% |    99.9 |    24.1 | 19.5% |cgca_sld$cgca_m3sld_
||================================================================
|  38.4% | 3,803.6 |      -- |    -- |MPI
||----------------------------------------------------------------
||  14.6% | 1,446.6 |   350.4 | 19.5% |mpi_bcast
||   9.4% |   932.4 |   473.6 | 33.7% |MPI_BARRIER
||   7.0% |   689.5 |   371.5 | 35.0% |mpi_recv
||   4.9% |   489.3 |    76.7 | 13.6% |MPI_ALLREDUCE
||   1.5% |   145.4 |   314.6 | 68.4% |MPI_REDUCE
||================================================================
|  17.8% | 1,766.8 |      -- |    -- |ETC
||----------------------------------------------------------------
||   9.9% |   983.9 |     8.1 |  0.8% |__DEALLOCATE
||   6.6% |   652.3 |    93.7 | 12.6% |gotoblas_dgemv_n_sandybridge
|=================================================================
```

Fig. 19. Raw profiling data of program `xx14`, coarray+MPI on 7200 cores on Cray XC30.

noticed that the tracing experiments of ParaFEM/GCPACK miniapps were reporting an inconsistent percentage of time for some `USER` functions which were previously highlighted in sampling experiments. Figs. 21 and 22 illustrate this effect on subroutine `cgca_gcupda`. Although the sampling report indicates that this subroutine is the most time consuming user function, `cgca_gcupda` is not present at all in the tracing report, even when `cgca_gcupda` was specifically traced. However, in the miniapp, `cgca_gcupda` and `cgca_hxi` are called exactly the same number of times. Indeed a call to `cgca_gcupd` is immediately followed by a call to `cgca_hxi` in `cgca_clvgp`.

IV. Conclusions

TAU (Tuning and Analysis Utilities) toolkit is well suited for profiling and tracing analysis of pure coarray and mixed coarray+MPI programs, where coarray operations are ultimately mapped onto MPI calls. The Intel implementation of coarrays seems to rely on MPI-2 RMA, with `MPI_Win_unlock` heavily dominating run times in three out of four analysed programs. Although MPI-2 RMA is supposed to be an optimal mapping of PGAS coarray communications, these results show that performance critically depends on implementation, thus indicating. a potential for optimisation of (MPI based) coarray implementations, such as that from Intel. CrayPAT (Cray Performance Analysis Tool) has been proved useful in highlighting load imbalance and performance hotspots in coarray+MPI programs on Cray XC30. Both TAU and CrayPAT are powerful and flexible tracing and sampling tools suitable for pure coarray and mixed MPI+coarray programs. The major differences from the user's prospective are that TAU is a free open source highly portable tool, whereas CrayPAT is a proprietary tool available only on Cray

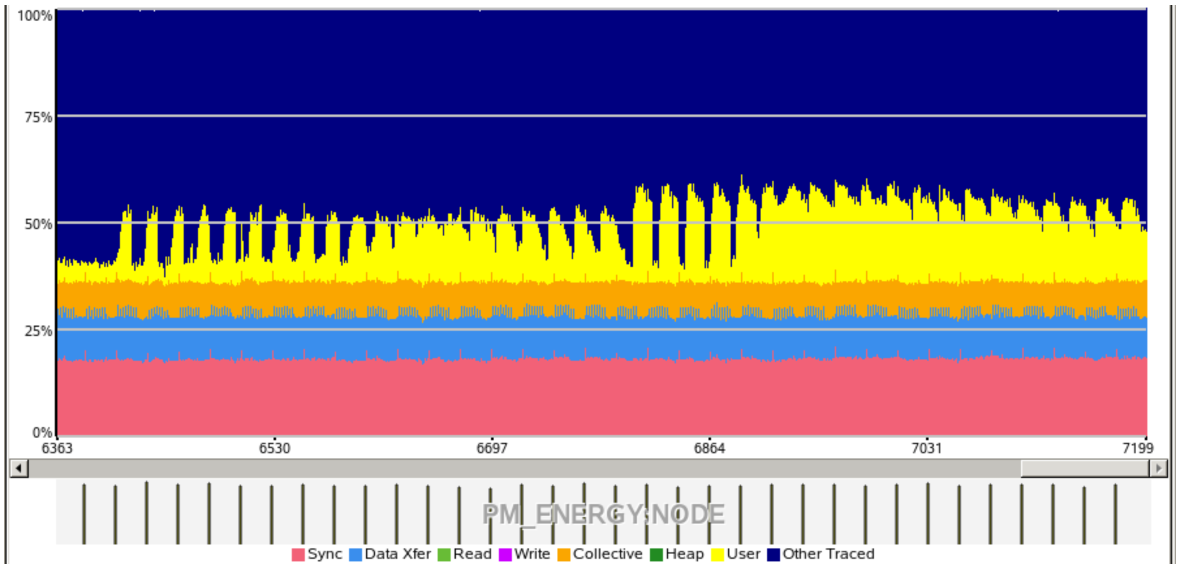
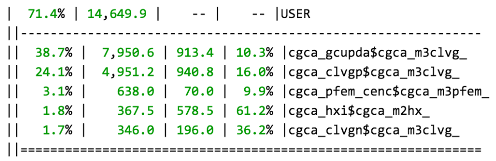
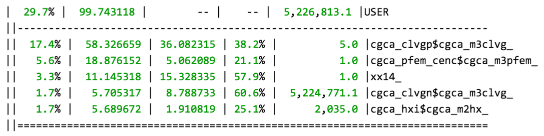


Fig. 20. `xx14` whole program activity on Cray XC30, shown in percentage of the total time per process. Processes 6363 to 7199 are shown.



```
|  71.4% | 14,649.9 |    -- |    -- |USER
||------------------------------------------------------------
||  38.7% |  7,950.6 | 913.4 | 10.3% |cgca_gcupda$cgca_m3clvg_
||  24.1% |  4,951.2 | 940.8 | 16.0% |cgca_clvgp$cgca_m3clvg_
||   3.1% |    638.0 |  70.0 |  9.9% |cgca_pfem_cenc$cgca_m3pfem_
||   1.8% |    367.5 | 578.5 | 61.2% |cgca_hxi$cgca_m2hx_
||   1.7% |    346.0 | 196.0 | 36.2% |cgca_clvgn$cgca_m3clvg_
||============================================================
```

Fig. 21. Sampling data for program `xx14`, coarray+MPI, on 7200 cores on Cray XC30, indicating that `cgca_gcupda` is the most time consuming.



```
|  29.7% |  99.743118 |        -- |    -- |  5,226,813.1 |USER
||---------------------------------------------------------------------
||  17.4% |  58.326659 | 36.082315 | 38.2% |          5.0 |cgca_clvgp$cgca_m3clvg_
||   5.6% |  18.876152 |  5.062089 | 21.1% |          1.0 |cgca_pfem_cenc$cgca_m3pfem_
||   3.3% |  11.145318 | 15.328335 | 57.9% |          1.0 |xx14_
||   1.7% |   5.705317 |  8.788733 | 60.6% |  5,224,771.1 |cgca_clvgn$cgca_m3clvg_
||   1.7% |   5.689672 |  1.910819 | 25.1% |      2,035.0 |cgca_hxi$cgca_m2hx_
||=====================================================================
```

Fig. 22. Tracing data for program `xx14`, coarray+MPI, on 7200 cores on Cray XC30.

systems. CrayPAT does not require program recompilation, whereas TAU 2.25.2 can support coarrays only using compiler instrumentation which requires a recompilation.

ACKNOWLEDGMENTS

This work was funded under the embedded CSE programme of the ARCHER UK National Supercomputing Service, http://www.archer.ac.uk. This work was carried out using the computational facilities of the Advanced Computing Research Centre, The University of Bristol, http://www.bris.ac.uk/acrc.

REFERENCES

[1] A. Shterenlikht, L. Margetts, L. Cebamanos, and D. Henty, "Fortran 2008 coarrays," *ACM Fortran Forum*, vol. 34, pp. 10–30, 2015.
[2] R. W. Numrich, J. Reid, and K. Kim, "Writing a multigrid solver using Co-Array Fortran," *Appl. Parallel. Comp.*, vol. 1541, pp. 390–399, 1998.
[3] ISO/IEC 1539-1:2010, *Fortran – Part 1: Base language, International Standard*, 2010.
[4] ISO/IEC JTC1/SC22/WG5 N2048, *TS 18508 Additional Parallel Features in Fortran*, 2015.
[5] G. Mozdzynski, M. Hamrud, and N. Wedi, "A partitioned global address space implementation of the European centre for medium range weather forecasts integrated forecasting system," *Int. J. High Perf. Comp. Appl.*, vol. 29, pp. 261–273, 2015.
[6] A. Fanfarillo, "Parallel programming techniques for heterogeneous exascale computing platforms," Ph.D. dissertation, University of Rome Tor Vergata, Italy, 2016.
[7] L. Cebamanos, A. Shterenlikht, D. Arregui, and L. Margetts, "Scaling hybid coarray/MPI miniapps on Archer," in *Cray User Group meeting (CUG2016), London, 8-12-MAY-2016*, 2016.
[8] H. Radhakrishnan, D. W. I. Rouson, K. Morris, S. Shende, and S. C. Kassinos, "Using coarrays to parallelize legacy Fortran applications: Strategy and case study," *Sci. Prog.*, vol. 2015, p. 904983, 2015.
[9] M. Haveraaen, K. Morris, D. Rouson, H. Radhakrishnan, and C. Carson, "High-performance design patterns for modern Fortran," *Sci. Prog.*, vol. 2015, p. 942059, 2015.
[10] S. Shende and A. D. Malony, "The TAU parallel performance system," *Int. J. High Perf. Comp. Appl.*, vol. 20, pp. 287–331, 2006.
[11] V. Tipparaju, W. Gropp, H. Ritzdorf, R. Thakur, and J. L. Träff, "Investigating high performance RMA interfaces for the MPI-3 standard," in *Proc. 2009 Int. Conf. Parallel Processing*, 2009. [Online]. Available: http://www.mcs.anl.gov/~thakur/papers/ICPP_09_RMA.pdf
[12] A. Shterenlikht and L. Margetts, "Three-dimensional cellular automata modelling of cleavage propagation across crystal boundaries in polycrystalline microstructures," *Proc. Roy. Soc. A*, vol. 471, p. 20150039, 2015.
[13] A. Shterenlikht, "Fortran coarray library for 3D cellular automata microstructure simulation," in *Proc. 7th PGAS Conf., Edinburgh, Scotland, UK*, 2013.
[14] I. M. Smith, D. V. Griffiths, and L. Margetts, *Programming the Finite Element Method*, 5th ed. Wiley, 2014.
[15] L. M. Evans, L. Margetts, V. Casalegno, L. M. Lever, J. Bushell, T. Lowe, A. Wallwork, P. Young, A. Lindemann, M. Schmidt, and P. M. Mummery, "Transient thermal finite element analysis of CFC-Cu ITER monoblock using X-ray tomography data," *Fusion Eng. Des.*, vol. 100, pp. 100–111, 2015.
[16] F. Levrero, L. Margetts, E. Sales, S. Xie, K. Manda, and P. Pankaj, "Evaluating the macroscopic yield behaviour of trabecular bone using a nonlinear homogenisation approach," *J. Mech. Behavior Biomed. Mater.*, vol. 61, pp. 384–396, 2016.
[17] A. Shterenlikht and I. C. Howard, "The CAFE model of fracture – application to a TMCR steel," *Fatigue Fract. Eng. Mater. Struct.*, vol. 29, pp. 770–787, 2006.
[18] S. Das, A. Shterenlikht, I. C. Howard, and E. J. Palmiere, "A general method for coupling microstructural response with structural performance," *Proc. Roy. Soc. A*, vol. 462, pp. 2085–2096, 2006.