Cellular automata beyond 100k cores: MPI vs Fortran coarrays

Anton Shterenlikht
Mech Eng Dept, The University of Bristol
Bristol, UK
mexas@bristol.ac.uk

Luis Cebamanos
EPCC, The University of Edinburgh
Edinburgh, UK
l.cebamanos@epcc.ed.ac.uk

ABSTRACT

Fortran coarrays are an attractive alternative to MPI due to a familiar Fortran syntax, single sided communications and implementation in the compiler. Scaling of coarrays is compared in this work to MPI, using cellular automata (CA) 3D Ising magnetisation miniapps, built with the CASUP CA library, https://cgpack.sourceforge.io, developed by the authors. Ising energy and magnetisation were calculated with `MPI_ALLREDUCE` and Fortran 2018 `co_sum` collectives. The work was done on ARCHER (Cray XC30) up to the full machine capacity: 109,056 cores. Ping-pong latency and bandwidth results are very similar with MPI and with coarrays for message sizes from 1B to several MB. MPI halo exchange (HX) scaled better than coarray HX, which is surprising because both algorithms use pair-wise communications: MPI `IRECV/ISEND/WAITALL` vs Fortran `sync images`. Adding OpenMP to MPI or to coarrays resulted in worse L2 cache hit ratio, and lower performance in all cases, even though the NUMA effects were ruled out. This is likely because the CA algorithm is memory and network bound. The sampling and tracing analysis shows good load balancing in compute in all miniapps, but imbalance in communication, indicating that the difference in performance between MPI and coarrays is likely due to parallel libraries (MPICH2 vs libpgas) and the Cray hardware specific libraries (uGNI vs DMAPP). Overall, the results look promising for coarray use beyond 100k cores. However, further coarray optimisation is needed to narrow the performance gap between coarrays and MPI.

CCS CONCEPTS

• **Software and its engineering → Parallel programming languages**; **Massively parallel systems**; **Software libraries and repositories**; • **Applied computing → Physical sciences and engineering**; *Physics*;

KEYWORDS

Fortran, coarrays, MPI, miniapps, Cray, cellular automata

ACM Reference Format:
Anton Shterenlikht and Luis Cebamanos. 2018. Cellular automata beyond 100k cores: MPI vs Fortran coarrays. In *25th European MPI Users' Group Meeting (EuroMPI '18), September 23–26, 2018, Barcelona, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3236367.3236384

1 INTRODUCTION

Cellular automata (CA) are a popular modelling tool for discrete systems on regular grids. The early use of CA is associated with the work of John Von Neumann and Stanislaw Ulam in 1940s on self-reproducing systems and theory of computation [5]. By the end of the 20th century CA found applications in very diverse areas of science and technology, including gas dynamics, diffusion, phase transitions, multiphase flow and magnetisation [7]. Recent uses of CA included epidemiology [16], biology [27], fire spread [20], dynamic recrystallisation [28], fracture [8, 22] and geology [1]. Finally, we note that lattice Boltzmann (LB) methods, popular in CFD, are a variant of CA [26].

At the heart of any CA is an idea that a physical space is split into a regular grid of identical cells, each with a discrete value from a pre-determined fixed set. An initial CA state must be chosen, from which CA can evolve iteratively. In each CA iteration the state of every cell is updated based on its state and the states of the cells in some neighbourhood of this cell. Thus an important property of a CA is *locality*, i.e. the state of a cell depends only on a local neighbourhood of this cells and any process propagates across a CA model at a rate of the width of the neighbourhood per iteration. Popular neighbourhoods are Moore's (8 cells in 2D, 26 cells in 3D) and Von Neumann's (4 cells in 2D and 6 cells in 3D) which have width of a single cell. The reader is referred to a recent CA review paper for more details [29].

The simplicity of CA formulation and update rules makes it relatively very cheap computationally because (1) cells require very little data, i.e. CA can be implemented as a multi-dimensional array of an intrinsic type; (2) no equilibrium checks, or other whole model operations are necessary; (3) the update rules (actual calculation) are typically very simple and fast. This means that much larger models can be analysed in a given time with CA than with other, more dominant numerical methods such as finite elements.

Moreover, there are cases where there is no good PDE description of the problem, or the boundary is too complex (fractal in the limit), requiring too many elements for adequate discretisation. One such case is a problem of transgranular cleavage fracture propagation in polycrystals [22, 23],

As a regular grids method, CA lends itself naturally to parallel implementation, e.g. via partitioned multi-dimensional arrays with halos. We note here some recent performance studies of CA. In 2004 [17] studied performance of CA of Fortran and C LB codes on RISC and vector architectures with OpenMP and COMPAS intra-node comms and MPI for inter-node comms. They achieved scaling of 512 CPUs with parallel efficiency of over 70%. [12] showed strong scaling of a LB dendritic growth model in Fortran90 with MPI to 3k cores on Kraken (ONRL, Cray XT5 system). [19] implemented CA in object-oriented C99 and achieved less than $\times 10$

speed-up on 15 SGI Altix ICE 8200 nodes. [14] used hardware counters for an analysis of a game of life CA code implemented in C and MPI (GCC + OpenMPI) on modern x86-64 CPUs up to 64 MPI processes. Perhaps unsurprisingly they conclude that performance is dependent on efficient use of cache.

In this work we describe how CA can be implemented in Fortran coarrays and compare performance of Ising magnetisation CA miniapps with coarrays and MPI on Cray XC30.

2 FORTRAN COARRAYS

Coarrays have been available on Cray for over 20 years, as an extension to the Fortran standard. They were standardised in Fortran 2008 [11], which is a PGAS (Partititioned Global Address Space) language. Further coarray features, such as collectives, teams, events and facilities for dealing with failures appear in the Fortran 2018 standard.

Coarrays are a native Fortran means for SPMD programming. Square bracket syntax is used to define or refer to a coarray object:

```
integer :: i, ic[*], k(10,10), kc(10,10)[*]
real, allocatable :: r(:,:,:), rc(:,:,:)[:,:,:]
```

all variables declared with [ ] are coarray variables, and : for allocatable variables means that dimensions and codimensions are chosen at run time, when a certain number of identical copies of the executable (*images*) are created by the operating system, which are executing asynchronously [25]. Each image has read/write access to coarray variables on all other images:

```
ic[5]=i ! The invoking image copies its value of i
        ! to ic on image 5 (remote write)
! allocate rc on all images - implicit sync all
allocate( rc(3,3,3) [5,5,*] )
! The invoking image copies whole array rc from
! image with coindex set [1,2,3] to its own copy
! of r (remote read)
r(:,:,:) = rc(:,:,:) [1,2,3]
```

The standard defines *execution segments* in a Fortran coarray program, which are separated by *image control* statements, such as `sync all` or `sync images`. `sync all` is a global barrier, similar to `MPI_barrier`.

Coarrays can interoperate with MPI or OpenMP, although to date there are only a few examples of such hybrid codes. The European Centre for Medium-range Weather Forecasts (ECMWF) has used coarrays in combination with MPI and OpenMP to achieve moderate scaling improvements [15]. Coarrays also have been used together with OpenMP in plasma codes [18].

3 CASUP LIBRARY FOR CA ON HPC

The authors have previously demonstrated that scalable CA miniapps can be build with their BSD licensed library CASUP (CA for SUPercomputers) [6, 21–25], https://cgpack.sourceforge.io. Microstructure solidification miniapps scaled to 32k cores on HECToR (Cray XE6) [21, 22], Fig. 1, and multiscale cellular automata finite element (CAFE) simulations of fracture in polycrystals implemented with CASUP and with an MPI Fortran FE library ParaFEM, http://parafem.org.uk, scaled to 8k cores on ARCHER (Cray XC30) [6, 23, 24], Fig. 2.

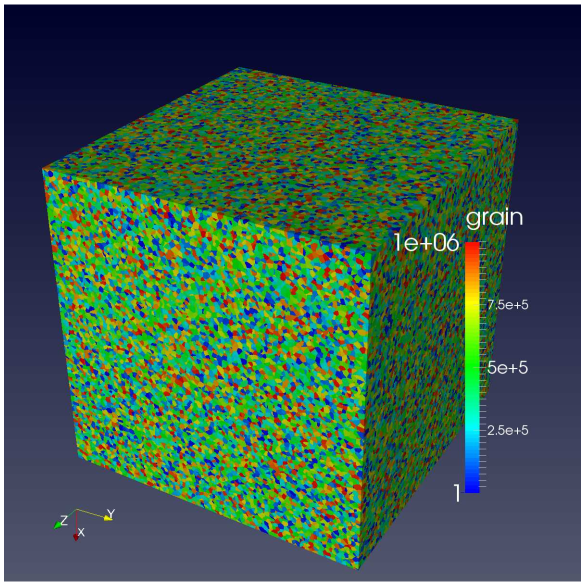

Figure 1: Equiaxed microstructure with 10^6 grains (10^{11} CA cells), from [23].

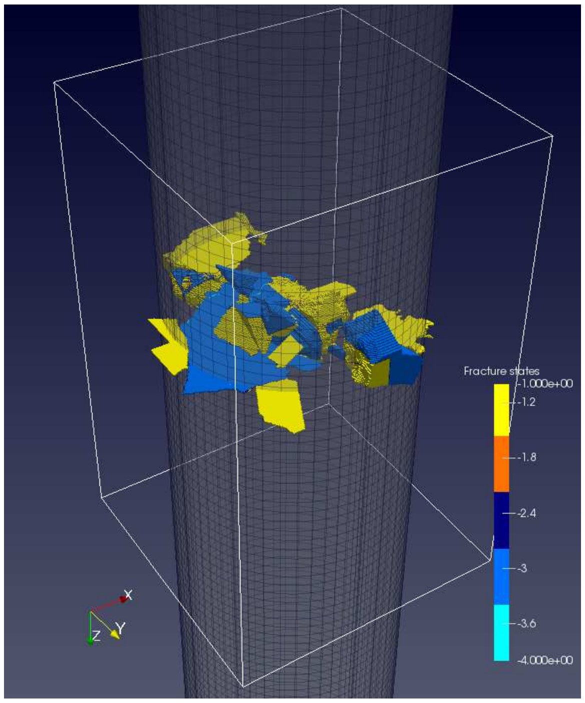

Figure 2: Fracture in a circular cylinder under uniaxial tension, from [23].

In this work the CASUP library has been completely re-designed to achieve 2 objectives:

(1) All synchronisation is now completely hidden from the user. CASUP automatically inserts the minimum required sync

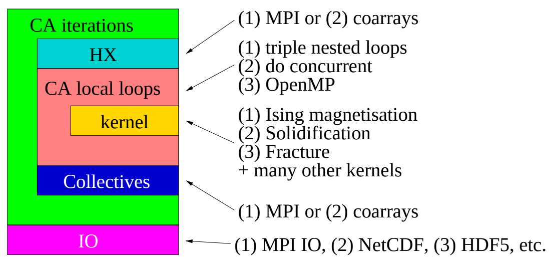

Figure 3: Modular structure of CASUP library.

calls to ensure data integrity. This removes the responsibility from the end user, who might be a domain scientist with little knowledge of HPC, to maintain data integrity when using CASUP routines. In practical terms this means there no sync calls in CASUP miniapps.

(2) CASUP is now completely modular, i.e. the CA kernels, the halo exchange or collective routines can be swapped with ease, allowing performance analysis of multiple combinations of Fortran coarray, MPI, OpenMP and do concurrent, Fig. 3.

While reducing synchronisation requirements should lead to better performance, the opposite is true for modularisation. Compilers often cannot vectorise loops with subroutine of function calls, even with inlining and whole program optimisation [13]. However, this issue will affect both the MPI and the coarray versions of CA, and hence is immaterial for performance comparisons of MPI vs coarrays, which is the aim of this work.

4 HALO EXCHANGE (HX)

In this work we use 3D domain decomposition of the CA space and Von Neumann 6-neighbourhood.

While MPI calls can be used for sending *any* data between images/processes, in Fortran 2008 and 2018 only coarray data can be used in remote calls. Therefore two possibilities for HX exist when using coarrays: (1) creating the whole of the CA model of coarrays (hereafter WCA), and (2) using coarrays only for CA halos (hereafter HCA). HX in HCA involves an extra step, copy of the boundary CA data into coarray arrays on the same image, see Fig. 4.

For comparison, the key code fragments for HX, along spatial dimension 1, are show below for HCA, WCA and MPI.

In HCA, space is a non-coarray array. Therefore the first step in HX is to copy the halo cells from space to coarray arrays h1minu and h1plus. This is a local operation:

```
if ( ci(1) .ne. 1        ) h1minu(:,:,:) =    &
 space(1:hdepth          , 1:sub(2), 1:sub(3) )
if ( ci(1) .ne. ucob(1) ) h1plus(:,:,:) =    &
 space(ihsta(1):sub(1), 1:sub(2), 1:sub(3) )
```

The second step is the actual HX:

```
if ( ci(1) .ne. 1 ) then    ! all but leftmost img
 sync images(nei_img_L(1)) ! sync with left image
 ! HX, remote op
 space(lhsta(1):0, 1:sub(2), 1:sub(3)) =        &
  h1plus(:,:,:)                                  &
   [nei_ci_L1(1), nei_ci_L1(2), nei_ci_L1(3)]
```

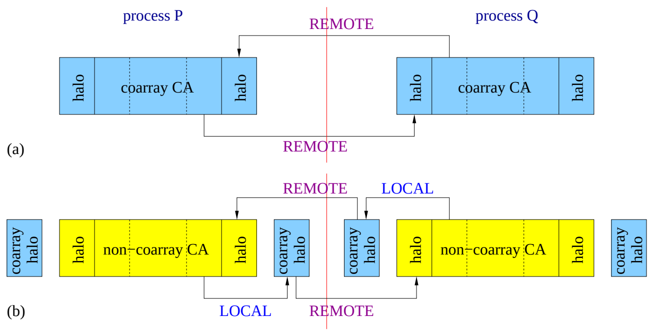

Figure 4: Schematics of HX in 1D, showing (a) a one-step algorithm (MPI and whole CA model (WCA) coarrays) and (b) a two-step algorithm used when coarrays are used only for CA halos (HCA). Local copy and remote comms are shown with arrows.

```
end if
! all but the rightmost image
if ( ci(1) .ne. ucob(1) ) then
 sync images(nei_img_R(1)) ! sync with right img
 ! HX, remote op
 space(rhsta(1):rhend(1), 1:sub(2), 1:sub(3)) = &
  h1minu(:,:,:)                                  &
   [nei_ci_R1(1), nei_ci_R1(2), nei_ci_R1(3)]
end if
```

Note that Cray 8.6.5 compiler used a non-blocking libpgas routine for the above remote calls:

```
ftn-6077: An implicit non-blocking operation
          was used for this statement.
```

In WCA, space is an array coarray and HX can be done in a single statement:

```
if (ci(1) .ne. 1) then      ! all but leftmost img
 sync images(nei_img_L(1)) ! sync with left image
 ! HX, remote op
 space(lhsta(1):0, 1:sub(2), 1:sub(3) ) =        &
  space(ihsta(1):sub(1), 1:sub(2), 1:sub(3) )    &
   [ nei_ci_L1(1), nei_ci_L1(2), nei_ci_L1(3) ]
end if
! all but the rightmost image
if ( ci(1) .ne. ucob(1) ) then
 sync images(nei_img_R(1)) ! sync with right img
 ! HX, remote op
 space(rhsta(1):rhend(1), 1:sub(2), 1:sub(3) )= &
  space(1:hdepth, 1:sub(2), 1:sub(3) )           &
   [ nei_ci_R1(1), nei_ci_R1(2), nei_ci_R1(3) ]
end if
```

For extra flexibility, the CASUP library provides for multiple kinds of MPI integers, matching the kind of CA data:

```
integer, parameter :: ca_range = 8
integer, parameter ::                            &
  iarr = selected_int_kind( ca_range )
integer(kind=iarr), allocatable :: space(:,:,:)
call MPI_TYPE_CREATE_F90_INTEGER( ca_range,     &
  mpi_ca_integer, ierr )
```

MPI HX requires that derived types be created for halos first, e.g. for the left halo along dimension 1, mpi_h1_LV.

```
call MPI_TYPE_CREATE_SUBARRAY( 3, sizes,        &
 subsizes, starts, MPI_ORDER_FORTRAN,           &
  mpi_ca_integer, mpi_h1_LV, ierr )
```

Then point-to-point MPI calls are used for HX. Note that space can be a coarray or a non-coarray array:

```
if ( ci(1) .ne. 1 ) then
 call MPI_IRECV(space,1,mpi_h1_LV,nei_img_L(1)-1,&
  TAG1R, MPI_COMM_WORLD, reqs1m(1), ierr )
 call MPI_ISEND(space,1,mpi_h1_LR,nei_img_L(1)-1,&
  TAG1L, MPI_COMM_WORLD, reqs1m(2), ierr )
  call MPI_WAITALL( 2, reqs1m, stats, ierr )
end if
if ( ci(1) .ne. ucob(1) ) then
 call MPI_IRECV(space,1,mpi_h1_RV,nei_img_R(1)-1,&
  TAG1L, MPI_COMM_WORLD, reqs1p(1), ierr )
 call MPI_ISEND(space,1,mpi_h1_RR,nei_img_R(1)-1,&
  TAG1R, MPI_COMM_WORLD, reqs1p(2), ierr )
 call MPI_WAITALL( 2, reqs1p, stats, ierr )
end if
```

These code fragments clearly show that coarray remote comms are *single sided*, i.e. no co-operation with the remote image is needed. This, and the fact that coarrays on Cray are implemented using symmetric memory, leads to the expectation that coarray implementation should perform better than MPI. Moreover, we expect that WCA should outperform HCA, because it avoids the extra local copy step. However, the scaling results show otherwise (see Sec. 7).

5 CA ITERATIONS

At the heart of any CA are loops over all cells with some kernel. CASUP provides routines with simple triple nested loops, with nested loops wrapped into OpenMP and do concurrent, which is a Fortran 2008 do loop designed for loops where the order of loop iterations does not affect the results. The idea is that a good optimising compiler can exploit this information for some sort of auto-threading of do concurrent loops, perhaps again via OpenMP. The OpenMP version looks like this:

```
!$omp parallel do default( none )                 &
!$omp private( i, j, k )                          &
!$omp shared( sub, space, hdepth, tmp_space )
do k = 1, sub(3)
do j = 1, sub(2)
do i = 1, sub(1)
 tmp_space( i,j,k ) =                             &
    kernel( space, hdepth, (/ i , j , k /) )
end do
end do
end do
!$omp end parallel do
```

from which the simple triple nested loop can be obtained by compiling with no OpenMP or by removing the $omp tokens.

The do concurrent version looks like this:

```
do concurrent(k=1:sub(3),j=1:sub(2), i=1:sub(1))
 tmp_space( i,j,k ) =                             &
    kernel( space, hdepth, (/ i , j , k /) )
end do
```

Unfortunately, Cray 8.6.5 Fortran compiler was unable to exploit the do concurrent parallelism: The compiler diagnostic for this is not specific:

```
ftn-6910: A loop was not multi-threaded
          for an unspecified reason.
```

It is well known that it is typically impossible to vectorise loops with function/subroutine calls [13]. Therefore it is not surprising that the compiler was not able to vectorise any of the loops:

```
ftn-6287: A loop was not vectorized because
          it contains a call to function "kernel".
```

6 3D ISING MAGNETISATION

We study the performance of CASUP via miniapps made with the 3D Ising magnetisation kernel.

A 3D extension of the Q2R Vichniac's 2D rule for Ising magnetisation [7] was implemented in this work. Each CA cell represents a single magnetic spin, which can take one of two values: 0 (down) or 1 (up). Energy conservation requires that CA cells are split into 2 groups according to a 3D chess-like pattern, with either all 'white' or all 'black' cells updated in a single CA iteration. This is an extension of the 2D chess-like pattern [7], achieved using a mask array:

```
ci = this_image( halo_array )
 c = ucobound( space ) - halo_depth
do concurrent( i=1:c(1), j=1:c(2), k=1:c(3) )
 mask_array(i,j,k)=mod( (i+j+k + (ci(1)-1)*c(1)+ &
  (ci(2)-1)*c(2) + (ci(3)-1)*c(3) ) , 2 )
end do
```

The energy of a CA cell is defined in the following way. Every neighbour of the same spin as the central cell adds −1 to its energy, and every neighbour of the opposite spin adds +1 to its energy. Energy conservation means that a spin will flip (change sign) only if this does not change its energy, i.e. it has 3 neighbours with spin 0 and 3 neighbours with spin 1:

```
! sum of the spins of the 6 neighbours
n = s(i-1,j,k) + s(i+1,j,k) + s(i,j-1,k) +       &
    s(i,j+1,k) + s(i,j,k-1) + s(i,j,k+1)
if (n.eq.3 .and. mask_array(i,j,k).eq.1 ) then
 ! If the sum of 6 neighbours is exactly 3
 ! and the mask value is 1 then flip the state.
  ca_kernel_ising = 1 - s(i,j,k)
else
  ca_kernel_ising = s(i,j,k) ! Otherwise no change
end if
```

Finally, a two-step CA iteration, with energy conservation, looks as follows, where hx_sub is the desired HX routine, with necessary data synchronisation, and iter_sub is the desired kernel routine:

```
tmp_space = space
do i = 1, 2*niter
 call hx_sub( space )        ! HX, space updated
 ! tmp_space updated, local op
 call iter_sub( space, hdepth, kernel )
 space = tmp_space           ! Local op
 mask_array = 1 - mask_array ! Flip the mask array
```

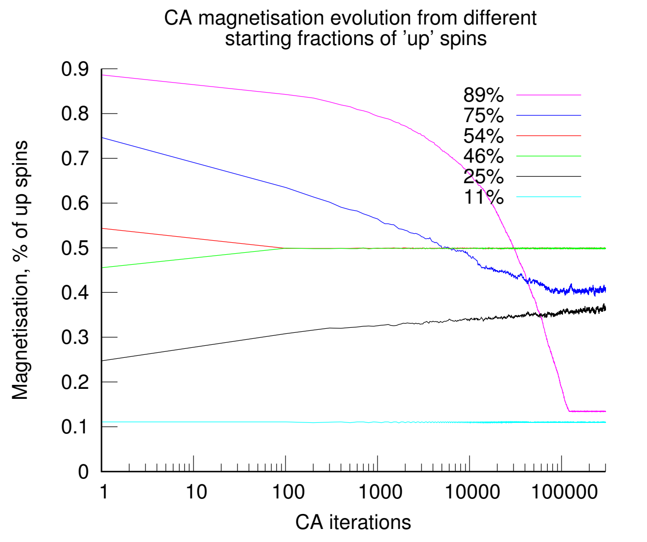

Figure 5: CASUP magnetisation evolution in the 2.7×10^{10} cell CA model.

```
end do
```

To verify that the energy is conserved in the model, it was calculated every CA iteration. Whole model magnetisation is the main model output. Accordingly it was also calculated every iteration. Collectives are used in both cases, either coarray `co_sum` or `MPI_ALLREDUCE` with `MPI_SUM`, after the local sums were calculated on each image.

7 RESULTS

7.1 Physics

Although the focus of this work is the performance comparisons between coarray and MPI miniapps at scale, it is important to show that 3D Ising magnetisation results achieved with CASUP are physically sound.

Fig. 5 shows magnetisation evolution for different starting values. The model shows existence of several stationary magnetisation states - around 50%, 40%, 11% and 13% of up spins. The 40% stationary state is consistent with existing literature [7], and the 50% stationary state (no overall magnetisation) is an obvious result, the other 2 stationary states, 11% and 13% of up spins, are unexpected and deserve a further investigation, particularly because these are very close together, but seem to be two distinct states. It is also worth investigating why the 75% and the 90% models continue to evolve beyond the first (or the first 2) stationary states. It is important to rule out any modelling artifacts before presenting these as true physical results.

Fig. 6 shows the starting and the final states of the CA model for the 90% initial fraction of up spins. Cell states are distributed at random at the start, with no particular spatial pattern. It is interesting to note that the final state likewise does not show any particular spatial arrangement or the up and down spin islands.

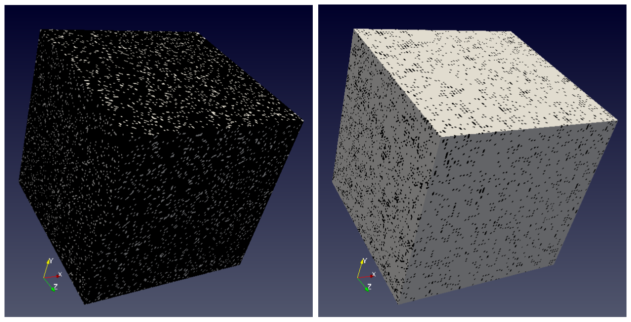

Figure 6: CA with 90% of up spins (left) evolves to 13% of 'up' spins in ~100k iterations (right). Up spin cells are black. Down spin cells are white.

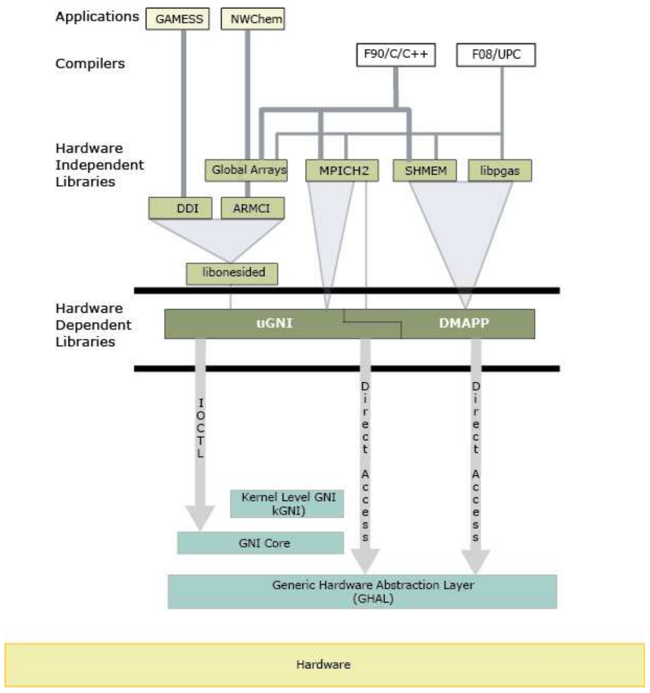

Figure 7: Details of Cray software stack, from [10].

7.2 HPC

All performance studies were done on ARCHER, the UK national HPC system, Cray XC30 with 2×12-core Ivy Bridge CPUs per node. All jobs were submitted with `aprun` as:

```
aprun -n $NP -ss -N $PN -S $PU -d $NT -T $EXE
```

where NP = PN× number of nodes; PU × NT = 12, to ensure there was always just one thread of execution per physical core, and `-ss` specifies 'strict memory containment per NUMA node', to ensure no process/thread allocates memory off local NUMA region.

It is important to consider the software stack on Cray, see Fig. 7. Cray provides two different hardware dependent libraries: nGNI (Generic Network Interface) and DMAPP (Distributed Shared Memory Application). Hardware independent portable communication

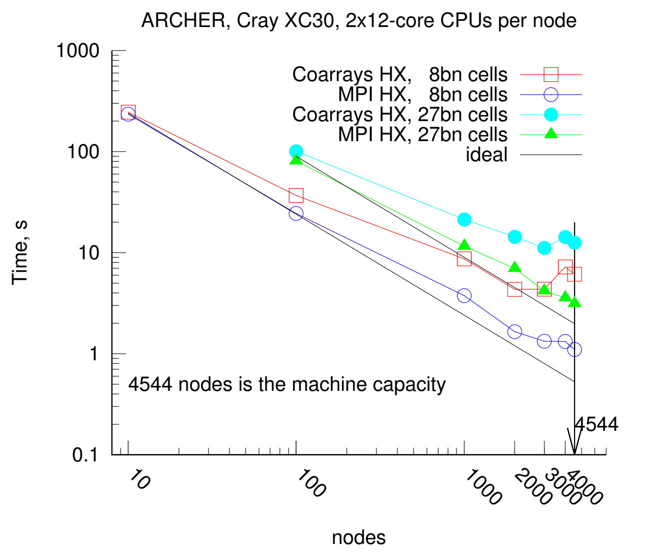

Figure 8: Scaling of a 3D Ising magnetisation miniapp with a cubic CA space.

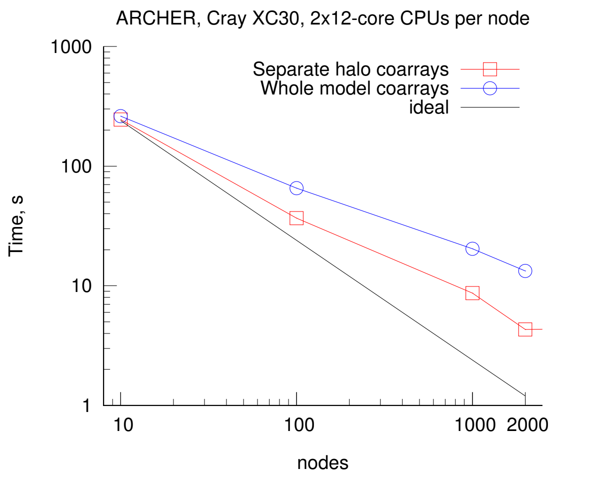

Figure 9: Scaling of a 3D Ising magnetisation coarray miniapps with 8×10^9 cells.

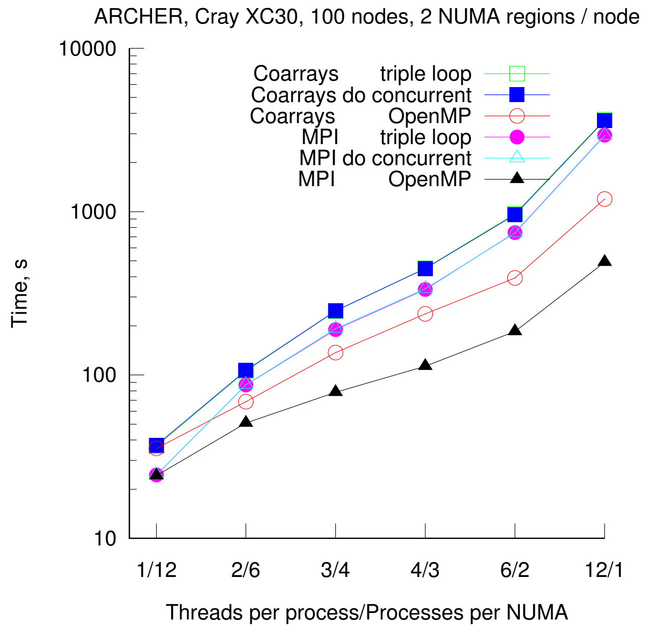

Figure 10: The influence of threading on performance for MPI and coarray 3D Ising magnetisation CA miniapps.

libraries are layered on top of these, specifically the MPI library (MPICH2) sits on top of uGNI, and the PGAS library sits on top of DMAPP. Fortran 2008 coarray miniapps are compiled with a PGAS Cray compiler (default option `-h pgas_runtime`) and linked against `libpgas`. The key observation, relevant for the understanding of the MPI and coarray CA performance, is that coarray and MPI miniapps use different hardware-specific libraries.

Fig. 8 shows that contrary to our expectations, MPI miniapps scale better than coarrays. Indeed, for a sufficiently large model, with 2.7×10^{10} cells, the MPI miniapp scaling limit is at least as high as the full ARCHER capacity, 4,544 nodes. In contrast the coarray miniapp scaling limit is only about 3,000 nodes or even 2,000 nodes for a smaller model with 8×10^9 cells. Moreover, while parallel efficiency with MPI miniapps is about 60%, it is only about 30% in coarray miniapps.

A 2015 study [9] showed that coarrays slightly outperformed single sided comms of MPI-3 for several PDE problems, (including multigrids, which are reasonably similar to CA, so a comparison is meaningful) at least up to 8k cores on Cray XC30 and up to 65k cores on Cray XK. Our finding that MPI-2 outperformed coarrays, also on Cray XC30, is therefore surprising.

Fig. 9 shows that, contrary to our expectations, WCA miniapps performed significantly worse than HCA, with the parallel efficiency of only about 10%.

Finally, Fig. 10 shows that any amount of threading dramatically reduces performance. Note that the `do concurrent` results match exactly with the triple nested loops, because the compiler was unable to auto-thread `do concurrent` loops, as explained in Sec. 5. Also note that all 6 combinations of the number of threads per process and process per core result in a single tread per core. For example, the data points for '3/4' in Fig. 10 show performance with 4 processes per NUMA region and with 3 threads per process, i.e. with 12 total threads per 12-core NUMA region. All memory was allocated outside OpenMP parallel regions, i.e. by the master thread. However, this could not have caused 'NUMA effects', because there was always at least 1 image per NUMA. This and the use of aprun option `-ss` ('strict memory containment per NUMA') ensured that all threads touch only the memory in the local NUMA region.

Although in all cases of using underpopulated nodes OpenMP shows best performance, this is still significantly worse than using

100 nodes			
Group	HCA	WCA	MPI
Triple loop + Ising kernel, %	19.3	17.9	27.9
Ising energy + collectives, %	29.6	33.6	21.2
HX, %	28.3	29.1	17.0
Total, %	77.2	80.6	66.1
Total time, s	191	210	120
1000 nodes			
Group	HCA	WCA	MPI
Triple loop + Ising kernel, %	4.6	3.2	5.0
Ising energy + collectives, %	27.1	26.2	22.2
HX, %	25.0	23.3	16.2
Total, %	56.7	52.7	43.4
Total time, s	59	82	44

Table 1: Relative times spent in different parts of 3D Ising magnetisation miniapps on 100 and on 1000 nodes. The total run times are also shown.

fully populated nodes and no threading. The most likely explanation for this is that the miniapps are well balanced because the same kernel is called for all CA cells and the same number of cells are processed by each image. It is known that addition of OpenMP to MPI is typically helpful only when applications show significant load imbalance [4]. We expect this to be true also for coarrays.

In addition the miniapps are likely to be memory and (at scale) network bound, rather than compute bound. Sec. 8 will show that there is a significant imbalance in comms. Since OpenMP adds overheads and potentially suffers from 'false sharing', a reduction in performance when adding OpenMP is not that surprising. A recent survey concluded that 'performance of all-MPI (no threading) on the Xeon and now even the KNL has been surprisingly good, and the need for adding threading on those systems has become less urgent' [13].

8 PROFILING

8.1 Non-threaded miniapps

CrayPat (Cray Performance Measurement and Analysis Tools) 6.4.6 was used in this work. We started with an automatic profiling of non-threaded miniapps, with CrayPat default settings, on 100 and 1000 nodes. The summary is shown in Tab. 1

It is clear that relatively more time is spent for remote comms (HX) and on collectives in coarray miniapps compared to MPI miniapps. Accordingly, coarray miniapps spent relatively less time calculating inside the triple loop than MPI. The data also confirms that WCA coarray approach results in less time spent computing and more time in remote comms and collectives that HCA, although the difference is small. On 1000 nodes the relative time spent by WCA on remote operations is actually lower than by HCA, however overall WCA spent less time than HCA in the user code. Finally, as expected, the overheads increase significantly from 100 to 1000 nodes, which is shown by a significant reduction of time spent in user code (Total, %). Overall this data just confirms what was seen already in the scaling results and does not provide any extra insight.

```
    Samp% |      Samp |     Imb. |  Imb. | Group
          |           |     Samp | Samp% |  Function
|-----------------------------------------------------------------
|  77.7% | 18,256.5 |       -- |    -- | USER
||----------------------------------------------------------------
|| 28.3% |  6,650.9 | 4,960.1 | 42.7% | ca_hx_all$ca_hx_
|| 21.8% |  5,119.2 | 5,456.8 | 51.6% | ca_ising_energy_col$ca_hx_
|| 10.7% |  2,519.5 |   144.5 |  5.4% | ca_kernel_ising$ca_hx_
||  8.6% |  2,011.1 |   138.9 |  6.5% | ca_iter_tl$ca_hx_
||  7.8% |  1,829.4 |   122.6 |  6.3% | ca_kernel_ising_ener$ca_hx_
```

Figure 11: HCA user functions on 100 nodes.

```
    Samp% |      Samp |     Imb. |  Imb. | Group
          |           |     Samp | Samp% |  Function
|------------------------------------------------------------------
|  80.8% | 20,428.1 |       -- |    -- | USER
||-----------------------------------------------------------------
|| 29.1% |  7,353.7 | 5,991.3 | 44.9% | ca_co_hx_all$ca_hx_
|| 24.7% |  6,245.7 | 6,590.3 | 51.4% | ca_co_ising_energy$ca_hx_
||  9.9% |  2,510.6 |   131.4 |  5.0% | ca_kernel_ising$ca_hx_
||  8.0% |  2,029.1 |   133.9 |  6.2% | ca_iter_tl$ca_hx_
||  7.2% |  1,821.4 |   140.6 |  7.2% | ca_kernel_ising_ener$ca_hx_
||  1.7% |    437.3 |    40.7 |  8.5% | ca_co_run$ca_hx_
```

Figure 12: WCA user functions on 100 nodes.

```
    Samp% |      Samp |     Imb. |  Imb. | Group
          |           |     Samp | Samp% |  Function
|-------------------------------------------------------------------
|  40.5% |  6,574.7 |       -- |    -- | USER
||------------------------------------------------------------------
|| 15.4% |  2,501.5 |   152.5 |  5.7% | ca_kernel_ising$ca_hx_
|| 12.5% |  2,024.0 |   139.0 |  6.4% | ca_iter_tl$ca_hx_
|| 11.4% |  1,845.5 |   104.5 |  5.4% | ca_kernel_ising_ener$ca_hx_
||==================================================================
|  27.3% |  4,425.4 |       -- |    -- | MPI
||------------------------------------------------------------------
|| 14.9% |  2,419.9 | 1,589.1 | 39.7% | mpi_waitall
||  9.8% |  1,592.8 | 1,832.2 | 53.5% | MPI_ALLREDUCE
||  2.1% |    347.3 |   119.7 | 25.6% | mpi_isend
```

Figure 13: MPI miniapp functions on 100 nodes.

A detailed load balancing data for the 3 miniapps is shown in Figs. 11-13 for 100 nodes and in Figs. 14-16 for 1000 nodes.

Figs. 11-13 show very little load imbalance (`Imb. Samp%` column) in all 3 miniapps in computation (kernels `ca_kernel_ising` and `ca_kernel_ising_energy`, and the triple loop `ca_iter_tl`) but a significant imbalance in comms. Note that in coarray miniapps (both HCA and WCA) comms are hidden inside the user functions, when sampling with the default CrayPat options. The HX functions `ca_hx_all` (HCA) and `ca_co_hx_all` (WCA) include coarray remote reads, and the energy calculation functions `ca_ising_energy_col` (HCA) and `ca_co_ising_energy` (WCA) include coarray collectives, as explained in Sec. 4.

Figs. 14-16 show a higher imbalance in computation, although these now account for only 5-7% of the total time. However, the imbalance in the comms is still about 50%, as at 100 nodes.

In an attempt to separate user functions from coarray comms we tried to instrument the HCA miniapp with `pat_build -g caf,pgas` option, which traces PGAS library calls separately from coarray user functions. Unfortunately this dramatically increased the overheads, to the extent that runtime grew by 7 times and relative timings were very different from the first sampling run. In fact, only

```
   Samp% |     Samp |    Imb. |  Imb. | Group
         |          |    Samp | Samp% |  Function
|-----------------------------------------------------------------
|  56.8% |  5,693.3 |      -- |    -- | USER
||----------------------------------------------------------------
|| 25.3% |  2,533.7 | 2,581.3 | 50.5% | ca_ising_energy_col$ca_hx_
|| 25.0% |  2,504.0 | 2,419.0 | 49.1% | ca_hx_all$ca_hx_
||  2.6% |    259.9 |    66.1 | 20.3% | ca_kernel_ising$ca_hx_
||  2.0% |    200.5 |    62.5 | 23.8% | ca_iter_tl$ca_hx_
||  1.8% |    183.9 |    50.1 | 21.4% | ca_kernel_ising_ener$ca_hx_
```

Figure 14: HCA user functions on 1000 nodes.

```
   Samp% |     Samp |     Imb. |  Imb. | Group
         |          |     Samp | Samp% |  Function
|------------------------------------------------------------------
|  53.1% |  7,610.0 |       -- |    -- | USER
||-----------------------------------------------------------------
|| 24.9% |  3,569.5 |  3,715.5 | 51.0% | ca_co_ising_energy$ca_hx_
|| 23.3% |  3,346.3 |  3,210.7 | 49.0% | ca_co_hx_all$ca_hx_
||  1.8% |    261.2 |    106.8 | 29.0% | ca_kernel_ising$ca_hx_
||  1.4% |    205.9 |     80.1 | 28.0% | ca_iter_tl$ca_hx_
||  1.3% |    187.1 |     71.9 | 27.8% | ca_kernel_ising_ener$ca_hx_
```

Figure 15: WCA user functions on 1000 nodes.

```
   Samp% |    Samp |    Imb. |  Imb. | Group
         |         |    Samp | Samp% |  Function
||===============================================================
|  37.4% | 3,436.1 |      -- |    -- | MPI
||---------------------------------------------------------------
|| 20.2% | 1,858.8 | 1,633.2 | 46.8% | MPI_ALLREDUCE
|| 16.5% | 1,516.0 | 1,604.0 | 51.4% | mpi_waitall
||===============================================================
|   7.1% |   653.3 |      -- |    -- | USER
||---------------------------------------------------------------
||  2.8% |   254.4 |    45.6 | 15.2% | ca_kernel_ising$ca_hx_
||  2.2% |   198.3 |    54.7 | 21.6% | ca_iter_tl$ca_hx_
||  2.0% |   181.5 |    32.5 | 15.2% | ca_kernel_ising_ener$ca_hx_
```

Figure 16: MPI miniapp functions on 1000 nodes.

```
  Time% |    Time |  Imb. |  Imb. |   Calls | Group
        |         |  Time | Time% |         |  Function
|------------------------------------------------------------------
| 44.4% |   671.8 |    -- |    -- |     34M | PGAS
||-----------------------------------------------------------------
||42.6% |   645.1 | 755.4 | 54.0% | 1,679.1 | __pgas_sync_with_image
|| 1.6% |    24.8 | 137.7 | 84.7% |     34M | __pgas_umove_nbi
||=================================================================
| 36.9% |   559.3 | 654.1 | 53.9% |   206.0 | CAF
||-----------------------------------------------------------------
||36.9% |   559.3 | 654.1 | 53.9% |   206.0 | __caf_cosum
||=================================================================
| 16.2% |   244.7 |    -- |    -- |   411.0 | USER
```

Figure 17: HCA miniapp functions on 100 nodes from `pat_build -O caf,pgas` tracing.

16% was spent in user routines, see Fig. 17, down from 77% with default CrayPat sampling settings. Note that CrayPat puts coarray collective `co_sum` in a different group from PGAS, which is surprising. While `pgas_umove_nbi` shows a significant load imbalance, the total time spent there is negligible. Overall, this data again does not shed any light on relatively poor performance of coarrays compared to MPI.

```
   Samp% |   Samp |    Imb. |  Imb. | Group
         |        |    Samp | Samp% |  Function
|-----------------------------------------------------------------
|  81.1% | 38,725 |      -- |    -- | USER
||----------------------------------------------------------------
|| 55.3% | 26,397 | 7,348.7 | 21.8% | ca_hx_all$ca_hx_
|| 12.0% |  5,744 | 9,613.7 | 62.7% | ca_ising_energy_col$ca_hx_
||  4.9% |  2,359 |   100.3 |  4.1% | ca_kernel_ising$ca_hx_
||  4.0% |  1,902 |   113.5 |  5.6% | ca_iter_omp$ca_hx_.LOOP@li.1130
||  3.9% |  1,874 |    72.3 |  3.7% | ca_kernel_ising_ener$ca_hx_
||================================================================
|   6.0% |  2,885 |      -- |    -- | OMP
```

Figure 18: HCA miniapp with OpenMP on 100 nodes with 4 threads per process, 3 processes per NUMA.

We can only speculate that source of the performance differences between the MPI the coarray comms lie lower down in the Cray software stack, either in the comms libraries (MPICH2 vs libpgas) or even lower, in the hardware libraries (uGNI vs DMAPP).

8.2 Threaded miniapps

Fig. 10 shows very similar trends for all miniapps. Therefore effects of threading were analysed via CrayPat sampling only on the HCA miniapp. Results on 100 nodes in Fig. 18 must be compared against those in Fig. 11. In addition Running with 4 threads per process resulted in ×2 increase in HX time, from 28% to 55%, and ×2 decrease in compute time, from 27% to 13%. In addition OpenMP overheads account for 6% of total time.

Importantly, the L2 cache utilisation is worse with OpenMP miniapps than in pure coarray or MPI miniapps. Without OpenMP it is about 75% in HCA, WCA and MPI miniapps:

```
D2 cache hit,miss ratio  76.0% hits  24.0% misses
```

With 4 OpenMP threads per process the hit rate drops to 65%:

```
D2 cache hit,miss ratio  65.4% hits  34.6% misses
```

although both values are quite low. As explained in Sec. 5, this is a direct consequence of the fact that CASUP is a modular library, with a subroutine call inside the main loops, which prohibits vectorisation.

9 COARRAYS VS MPI COMMUNICATIONS

In an attempt to understand better the difference in coarrays vs MPI communications, we did simple 'ping-pong' measurements of latency and bandwidth for both. We used the EPCC Fortran Coarray micro-benchmark suite [2]. As Figs. 19 and 20 show, the results are very similar, which does not explain the difference seen in CA miniapps. These figures show the time taken for messages up to 1MB using `get` with coarrays, and using `mpi_Isend` with MPI for comparison. To ensure that the Cray Aries network is used rather than shared-memory copies, the tests were done using two images (or MPI processes) placed on different compute nodes of the XC30 system. The results shown in Fig. 19 demonstrate that MPI has lower latency for messages up to 1kB. The reason for this is that the coarray time is dominated by synchronization calls (`sync all`). Above 1kB, MPI and coarray show very similar performance. Similar situation can be seen in Fig.20 with regards to the bandwidth, MPI bandwidth is about 2× higher than coarray bandwidth for small messages up to 1kB and very similar rate above 1kB.

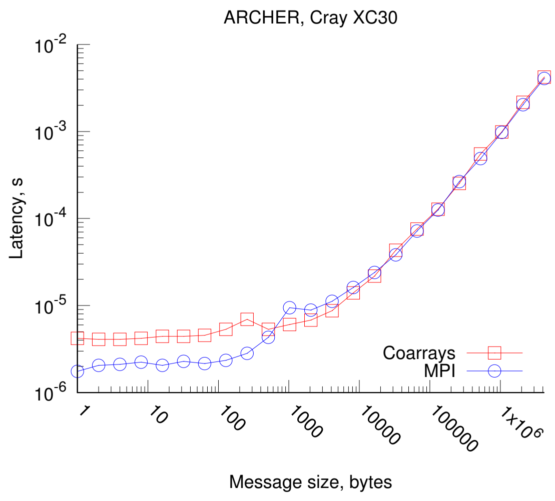

Figure 19: Coarrays vs MPI latency.

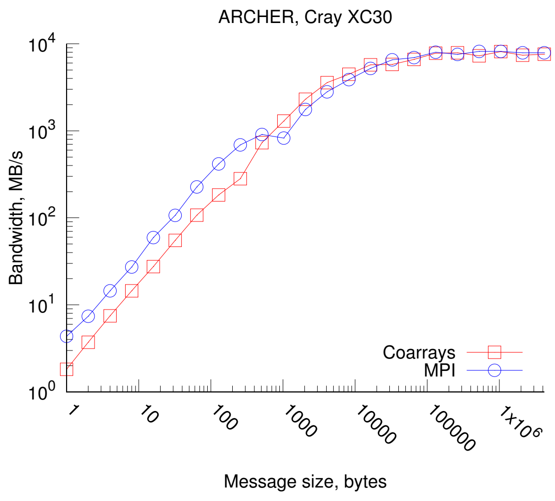

Figure 20: Coarrays vs MPI bandwidth.

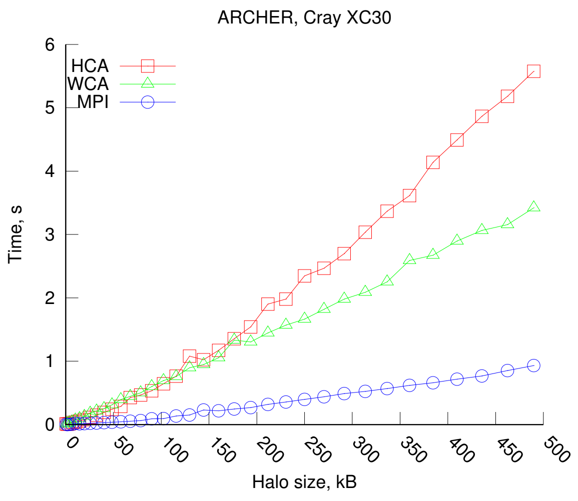

Figure 21: HX timing on 6 nodes.

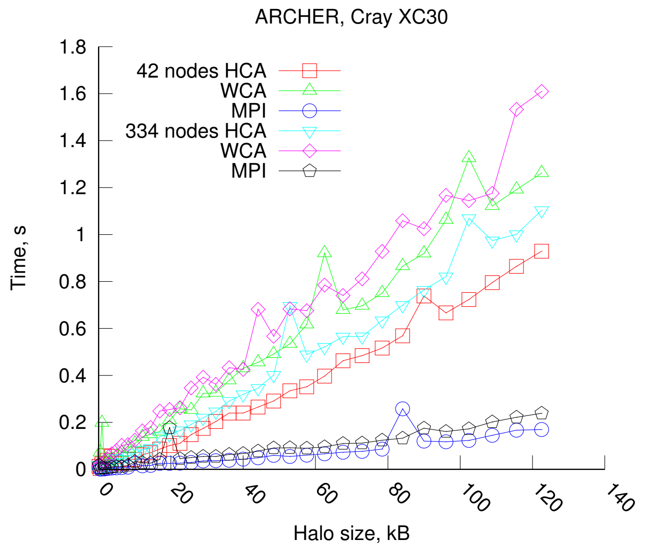

Figure 22: HX timing on 42 and 334 nodes.

Figs. 21-22 show timing for only the HX step, where each point is a maximum time, across all processes, of 10 HX calls. Fig. 21 shows that not only MPI is faster by a factor of 4-5, but also that WCA becomes beneficial only for halo sizes > 200kB on 6 nodes. At scale, when the halos are 100kB or smaller, Fig. 22 shows that MPI is 6-8 times faster, and, as expected, a higher number of nodes results in a longer run times, for the same halo size.

10 FUTURE

It might be beneficial to change CA domain decomposition from a 3D grid to a 1D linear array. A 1D decomposition reduces the number of messages by a factor of 3, but the messages are much bigger (very roughly by a factor $n^{2/3}$, assuming that a cubic CA model is decomposed into n identical cubes on n images), see Fig. 23. In addition, the amount of memory required for halo cells in 1D decomposition is higher than in 3D. Finally, a 1D decomposition might not be possible at all at scale, if the longest dimension of the CA model has fewer cells than the number of images. A 2D decomposition might be a useful compromise.

Fortran 2018 standard, expected to be published in 2018, will add events and a richer set of atomics to the language. These might be used to implement an asynchronous HX, similar to [3], which should help balance the comms load.

11 CONCLUSIONS

CASUP library proved useful for studying performance of MPI and coarray CA miniapps with optional OpenMP threading. Contrary to our expectations, and to previously published measurements [9],

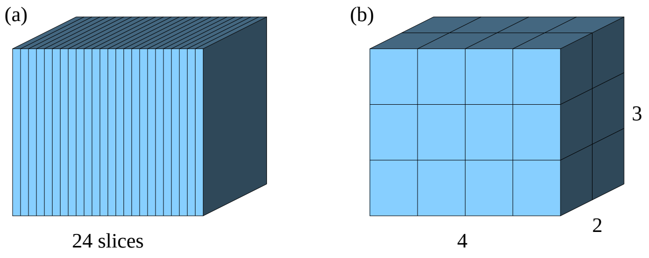

Figure 23: Schematic of a 1D (left) and a 3D (right) domain decomposition of CA on 24 images.

the results show clearly that single sided Fortran coarray communications perform worse than MPI-2 non-blocking ISEND/IRECV for a regular grid halo exchange in 3D CA simulations on Cray XC30. This is true from small scale (6 nodes) to the full machine capacity (4544 nodes). Sampling of complete CA miniapps shows that although computations are well balanced, there is significant imbalance in communications, but the level of imbalance is similar with MPI and with coarrays. Moreover, although a simple 'ping-pong' benchmark shows very similar bandwidth and latency characteristics for MPI and coarrays, timing of the halo exchange step shows that MPI is faster than coarrays by a factor of 4-8. These inconclusive results point to different levels of optimisation in MPI and PGAS libraries, or the hardware specific libraries uGNI (for MPI) and DMAPP (for coarrays). One possible explanation is that coarrays are still new and received little use. This is likely reflected in the amount of effort spent by vendors such as Cray on coarray optimisation. Wider adoption of coarrays in HPC will likely improve vendor support and therefore performance. Adding OpenMP to MPI or to coarrays was found to lower performance significantly in all cases, because the computation in CA miniapps is well balanced and the CA algorithm is memory and network bound. Although this study focused on strong scaling, CASUP is equally well suited for the study of node-level performance, i.e. loop optimisation for efficient cache usage.

ACKNOWLEDGMENTS

This work was supported by EPSRC grant EP/R013047/1 and used the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk), project e560.

REFERENCES

[1] G. A. Albertao, R. Eschard, T. Mulder, V. Teles, B. Chauveau, and P. Joseph. 2015. Modeling the deposition of turbidite systems with Cellular Automata numerical simulations: A case study in the Brazilian offshore. *Marine Petrol. Geol.* 59 (2015), 166–186. https://doi.org/10.1016/j.marpetgeo.2014.07.010
[2] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham. 2012. A Parallel Benchmark Suite for Fortran Coarrays. *Applications, Tools and Techniques on the Road to Exascale Computing, IOS Press* 22 (2012), 281 – 288. https://doi.org/10.3233/978-1-61499-041-3-281
[3] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham. 2014. Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP. *Int. J. HPC Appl.* 28 (2014), 97–111. https://doi.org/10.1177/1094342013493123
[4] J. M. Bull, J. Enright, Xu Guo, C. Maynard, and F. Reid. 2010. Performance evaluation of mixed-mode OpenMP/MPI implementations. *International Journal of Parallel Programming* 38 (2010), 396–417. https://doi.org/10.1007/s10766-010-0137-2
[5] A. W. Burks (Ed.). 1970. *Essays on Cellular Automata.* University of Illinois Press.
[6] L. Cebamanos, A. Shterenlikht, D. Arregui-Mena, and L. Margetts. 2016. Scaling hybid coarray/MPI miniapps on Archer. In *Cray User Group 2016 meeting (CUG2016), London.* https://cug.org/proceedings/cug2016_proceedings/includes/files/pap120s2-file1.pdf.
[7] B. Chopard and M. Droz. 1998. *Cellular Automata Modelling of Physical Systems.* Cambridge.
[8] S. Das, A. Shterenlikht, I. C. Howard, and E. J. Palmiere. 2006. A general method for coupling microstructural response with structural performance. *Proc. Roy. Soc. A* 462 (2006), 2085–2096. https://doi.org/10.1098/rspa.2006.1681
[9] S. Garain, D. S Balsara, and J. Reid. 2015. Comparing Coarray Fortran (CAF) with MPI for several structured mesh PDE applications. *J. Comp. Phys.* 297 (2015), 237–253. https://doi.org/10.1016/j.jcp.2015.05.020
[10] Cray Inc. 2018. *XC Series GNI and DMAPP API User Guide (CLE 6.0.UP06) S-2446.*
[11] ISO/IEC 1539-1. 2010. *Fortran – Part 1: Base language, International Standard.*
[12] B. Jelinek, M. Eshraghi, C. Felicelli, and J. F. Peters. 2014. Large-scale parallel lattice Boltzmann-cellular automaton model of two-dimensional dendritic growth. *Comp. Phys. Comms* 185 (2014), 939–947. https://doi.org/10.1016/j.cpc.2013.09.013
[13] J. Levesque and A. Vose. 2018. *Programming for Hybrid Multi/Manycore MPP Systems.* CRC Press.
[14] E. N. Millan, C. S. Bederian, M. F. Piccoli, C. G. Garino, and E. M. Bringa. 2015. Performance analysis of Cellular Automata HPC implementations. *Computers Electr. Engng* 48 (2015), 12–24. https://doi.org/10.1016/j.compeleceng.2015.09.015
[15] G. Mozdzynski, M. Hamrud, and N. Wedi. 2015. A Partitioned Global Address Space implementation of the European Centre for Medium Range Weather Forecasts Integrated Forecasting System. *Int. J. High Perf. Comp. Appl.* 29 (2015), 261–273. https://doi.org/10.1177/1094342015576773
[16] B. Pfeifer, K. Kugler, M. M. Tejada, C. Baumgartner, M. Seger, M. Osl, M. Netzer, M. Handler, A. Dander, M. Wurz, A. Graber, and B. Tilg. 2008. A cellular automaton framework for infectious disease spread simulation. *Open Med. Inform. J.* 2 (2008), 70–81. https://doi.org/10.2174/1874431100802010070
[17] T. Pohl, F. Deserno, N. Thurey, U. Rude, P. Lammers, G. Wellein, and T. Zeiser. 2004. Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures. In *SC2004.* http://supercomputing.org/sc2004/schedule/pdfs/pap173.pdf.
[18] R. Preissl, N. Wichmann, B. Long, J. Shalf, S. Ethier, and A. Koniges. 2011. Multithreaded Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms. In *SC11, USA.* http://upc.lbl.gov/publications/Preissl_SC2011.pdf.
[19] L. Rauch, L. Madej, P. Spytkowski, and R. Golab. 2015. Development of the cellular automata framework dedicated for metallic materials microstructure evolution models. *Arch. Civil Mech. Eng.* 15 (2015), 48–61. https://doi.org/10.1016/j.acme.2014.06.006
[20] X. P. Rui, S. Hui, X. T. Yu, G. Y. Zhang, and B. Wu. 2018. Forest fire spread simulation algorithm based on cellular automata. *Natural Hazards* 91 (2018), 309–319. https://doi.org/10.1007/s11069-017-3127-5
[21] A. Shterenlikht. 2013. Fortran coarray library for 3D cellular automata microstructure simulation. In *7th International Conference on PGAS Programming Models.* Edinburgh, Scotland, UK, ISBN 978-0-9926615-0-2.
[22] A. Shterenlikht and L. Margetts. 2015. Three-dimensional cellular automata modelling of cleavage propagation across crystal boundaries in polycrystalline microstructures. *Proc. Roy. Soc. A* 471 (2015), 20150039. https://doi.org/10.1098/rspa.2015.0039
[23] A. Shterenlikht, L. Margetts, and L. Cebamanos. 2017. Fortran coarray/MPI Multi-Scale CAFE for Fracture in Heterogeneous Materials. In *Fifth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering.* Civil-Comp Press, Stirlingshire, UK. https://doi.org/10.4203/ccp.111.40
[24] A. Shterenlikht, L. Margetts, L. Cebamanos, and J.D. Arregui-Mena. 2016. Multi-scale CAFE framework for simulating fracture in heterogeneous materials implemented in Fortran coarrays and MPI. In *PGAS Application Workshop (PAW), held in conjunction with SC16.* https://doi.org/10.1109/PAW.2016.006
[25] A. Shterenlikht, L. Margetts, L. Cebamanos, and D. Henty. 2015. Fortran 2008 coarrays. *ACM SIGPLAN Fortran Forum* 34 (2015), 10–30. https://doi.org/10.1145/2754942.2754944
[26] G. Vahala, L. Vahala, and M. Soe. 2013. Lattice Boltzmann algorithms for plasma physics. *Radiation Effects Defects Solids* 168 (2013), 735–758. https://doi.org/10.1080/10420150.2013.831856
[27] A. Yoshimoto, P. Asante, M. Konoshima, and P. Surovy. 2017. Integer programming approach to control invasive species spread based on cellular automaton model. *Natural Resource Model.* 30 (2017), 1–42. https://doi.org/10.1111/nrm.12101
[28] C. W. Zheng and D. Raabe. 2013. Interaction between recrystallization and phase transformation during intercritical annealing in a cold-rolled dual-phase steel: A cellular automaton model. *Acta Mat.* 61 (2013), 5504–5517. https://doi.org/10.1016/j.actamat.2013.05.040
[29] Y. H. Zhuang, W. Y. Li, H. H. Wang, S. Hong, and H. J. Wang. 2017. A Bibliographic Review of Cellular Automaton Publications in the Last 50 Years. *J Cellular Automata* 12 (2017), 475–492.