Evaluation of the Coarray Fortran Programming Model on the Example of a Lattice Boltzmann Code

Klaus Sembritzki
Friedrich-Alexander University Erlangen-Nuremberg
Erlangen Regional Computing Center (RRZE)
Martensstrasse 1
91058 Erlangen, Germany
klausem@gmail.com

Georg Hager
Friedrich-Alexander University Erlangen-Nuremberg
Erlangen Regional Computing Center (RRZE)
Martensstrasse 1
91058 Erlangen, Germany
georg.hager@rrze.fau.de

Bettina Krammer
Université de Versailles St-Quentin-en-Yvelines
Exascale Computing Research (ECR)
45 Avenue des Etats-Unis
78000 Versailles, France
bettina.krammer@uvsq.fr

Jan Treibig
Friedrich-Alexander University Erlangen-Nuremberg
Erlangen Regional Computing Center (RRZE)
Martensstrasse 1
91058 Erlangen, Germany
jan.treibig@rrze.fau.de

Gerhard Wellein
Friedrich-Alexander University Erlangen-Nuremberg
Erlangen Regional Computing Center (RRZE)
Martensstrasse 1
91058 Erlangen, Germany
gerhard.wellein@rrze.fau.de

ABSTRACT

The Lattice Boltzmann method is an explicit time-stepping scheme for the numerical simulation of fluids. In recent years it has gained popularity since it is straightforward to parallelize and well suited for modeling complex boundary conditions and multiphase flows. Starting from an MPI/OpenMP-parallel 3D prototype implementation of the algorithm in Fortran90, we construct several coarray-based versions and compare their performance and required programming effort to the original code, demonstrating the performance tradeoffs that come with a high-level programming style.

In order to understand the properties of the different implementations we establish a performance model based on microbenchmarks, which is able to describe the node-level performance and the qualitative multi-node scaling behavior. The microbenchmarks also provide valuable low-level information about the underlying communication mechanisms in the CAF implementations used.

1. INTRODUCTION

In contrast to the popular Message Passing Interface (MPI), PGAS (Partitioned Global Address Space) languages like UPC (Unified parallel C), the Coarray Fortran (CAF) programming language defined in 1998 by Robert Numrich and John Reid [1] [2], and the SHMEM library [3] have not (yet) managed to gain greater acceptance outside academia. However, advocates of PGAS languages claim that PGAS languages can and should replace MPI and OpenMP, typical arguments being that they are more readable and easier to learn.

With the Fortran 2008 standard, coarrays became a native Fortran feature [2]. The "hybrid" nature of modern massively parallel systems, which comprise shared-memory nodes built from multicore processor chips, can be addressed by the compiler and runtime system by using shared memory for intra-node access to codimensions. Inter-node access may either by conducted using an existing messaging layer such as MPI or SHMEM, or carried out natively using the primitives provided by the network infrastructure.

Cray has a long tradition with PGAS languages, starting with the support for coarrays on the Cray T3E in 1998 [4]. UPC is also supported by the Cray Programming Environments, the claim being that Cray Computers can efficiently map PGAS high-level constructs to network calls. One compute platform used throughout this work is therefore a Cray XE6 machine with *Gemini* routers [5] [6].

In order to compare with a commodity-type environment we used a standard *Infiniband* Intel Westmere Cluster. On this hardware, the Intel Fortran Compiler 12.0 (update 4) was chosen, which can be considered a very common choice among software developers. The Rice University CAF 2.0 compiler and a development version of the Open64 compiler 4.2 with CAF support were evaluated, but found to be too immature for productive use.

As application testcase, we chose a solver implementing the Lattice Boltzmann method, an area of research pursued at the *University of Erlangen Nürnberg* for the past few years. In particular, two large scale production codes have been developed at the *RRZE (Erlangen Regional Computing Center)* [7] and the *LSS (Chair for System Simulation)* [8]. The Lattice Boltzmann method is an explicit time-stepping scheme for the numerical simulation of fluids, straightforward to parallelize and well suited for modelling complex boundary conditions and multiphase flows. In addition to its production code, the RRZE maintains a small prototype 3D Lattice Boltzmann code [9, 10], which is written in Fortran and is already single core optimized and parallelized with MPI and OpenMP. This work extends this code by parallelization with coarrays and compares performance and programming effort to the MPI implementati-

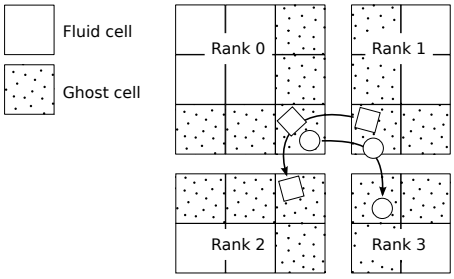


Figure 1: Exchange of ghost cells in vertical and horizontal direction. Diagonal neighbours are exchanged indirectly.

on. The measurements are supplemented by low-level benchmarks to evaluate the quality of different hardware-compiler combinations and to be able to draw conclusions about the applicability of the coarray programming paradigm for other algorithms.

This paper is organized as follows. Section 2 describes the implementation of the Lattice Boltzmann use case, both for MPI and CAF. Section 3 gives an overview of the hardware and software environments that were used and describes the performance characteristics of the different hardware-software combinations by means of low-level benchmarks. These results are then used to establish performance models for the MPI and CAF implementations of the Lattice Boltzmann solver. In section 4 we present the Lattice Boltzmann benchmark results and validate our performance models. Finally, section 5 gives an overview on related work, and section 6 contains a short summary of lessons learnt and some conclusions.

2. IMPLEMENTATION OF THE LATTICE BOLTZMANN SOLVER

Lattice Boltzmann methods (LBM) were derived from lattice gas automata by regarding the interaction not of single particles but of particle clusters with discrete velocities. The typical implementation of the algorithm discretizes the space with a Cartesian lattice. Each lattice cell stores a discrete particle distribution function (PDF), a set of positive scalar values f_i giving the probability of finding a particle with the discrete velocity of index i in the cell. The discrete velocities are chosen such that a particle, which is in the center of one cell and has one of these discrete velocities, moves exactly into the center of one adjacent cell in one time step. So, in each time step, the values of f_i are changed according to the velocities of the other particles in the cell, and each particle conglomerate represented by the f_i values moves to one adjacent lattice cell thereafter.

For correct boundary handling a "bounce-back" step is required, which implements no-slip boundary conditions. The computational effort for this step is usually small compared to the fluid cell updates (N^2 vs. N^3, where N is the number of fluid cells), so we ignore it for the performance analysis below.

In the D3Q19 model implemented in our prototype LBM code, 19 scalar values are stored for each cell. Parallelization is done similarly to other stencil codes, via a Cartesian distribution of the lattice cells among processes and by exchange of halo cells as depicted in Fig. 1. Our LBM code allows for a 1D, 2D or 3D domain decomposition, though, in this paper, we focus on the 2D and 3D cases only. As access to main memory is fastest when the accesses have stride one, there exist slow and fast axes for traversing a multi-dimensional array. The traversal is therefore fastest if slices that cut the 3D array along its slowest axis are exchanged. For Fortran, the last and therefore third axis is the slowest axis.

```
subroutine exchange_ghost_cells(A, B)
  call mpi_irecv(recvBufA, A, recvReqstA)
  call mpi_irecv(recvBufB, B, recvReqstB)

  if (A>=0) call copy_cells(A, sendBufA)
  call mpi_isend(sendBufA, A, sendReqstA)

  if (B>=0) call copy_cells(B, sendBufB)
  call mpi_isend(sendBufB, B, sendReqstB)

  ! missing code: wait for recvReqstA and recvReqstB

  if (A>=0) paste_cells(recvBufA)
  if (B>=0) paste_cells(recvBufB)

  ! missing code: wait for sendReqstA and sendReqstB
end subroutine
```

Listing 1: Simplified version of the MPI communication code for exchanging ghost cells with direct neighbours (rank `A` and `B`) in one direction (dummy neighbour `MPI_PROC_NULL` at the borders of compute domain), with manual buffering.

2.1 MPI Implementation

The MPI implementation uses non-blocking send/receive pairs and manual buffering. The `copy_cells` subroutine copies the data from the array storing the ghost parts of the PDF to the send buffer, while the `paste_cells` function copies the data from the receive buffer into the ghost parts of the PDF.

For the sake of simplicity, the code shown in listing 1 assumes that the send and receive buffers have the same size in each direction, which means that the ghost cells in x, y and z direction must have equal size. The actual application code does not have this restriction.

2.2 MPI-like, Buffered Coarray Fortran Implementation

The MPI-like, buffered CAF version was created by converting the receive buffers of the MPI implementation into coarrays. The corresponding code is shown in listing 2. While MPI ranks are numbered starting from 0, CAF images are numbered starting from 1, hence `images_index()=rank+1`.

The line `recvBufA[B]=sendBufB` is discussed in more detail. By definition as in the original MPI code in listing 1, `recvBufA` contains the ghost cells received from `A`. Using the original MPI rank notation, suppose that there exist only two processes, rank 0 being western of rank 1. Then, rank 0 calls `exchange_ghost_cells(A=-1, B=1)` and rank 1 calls `exchange_ghost_cells(A=0, B=-1)`, both using the dummy neighbour `-1` at their outer borders. This means that rank 1 expects the data from rank 0 in `recvBufA` and rank 0 has to execute `recvBufA[B]=sendBufB`. This is the so-called *push* mode, as data are pushed from `A` to `B`, in *pull* mode the data transfer would have to be initiated by `B`. Before and after pushing data, synchronization is needed to ensure consistency of data between `A` and `B`.

Again, the simplified code shown here expects the ghost cells in x, y and z direction to be of equal size, the actual application code does not have this restriction.

2.3 Unbuffered Coarray Fortran Implementation

The main part of the unbuffered CAF implementation is hidden in the subroutine `transmit_slice` (listing 3). In the buffered, MPI-like CAF implementation, it was relatively easy for the sender to determine the place where the receiver would expect the data from its neighbour to be stored: `B` expected the data to be in `recvBufA`. In

```
subroutine exchange_ghost_cells(A, B)
  if (A/=-1) call copy_cells(A, sendBufA)
  if (B/=-1) call copy_cells(B, sendBufB)

  if (A/=-1 .and. B/=-1) sync images([A, B]+1)
  if (A/=-1 .and. B==-1) sync images([A   ]+1)
  if (A==-1 .and. B/=-1) sync images([   B]+1)

  if (A/=-1) recvBufB[A] = sendBufA
  if (B/=-1) recvBufA[B] = sendBufB

  if (A/=-1 .and. B/=-1) sync images([A, B]+1)
  if (A/=-1 .and. B==-1) sync images([A   ]+1)
  if (A==-1 .and. B/=-1) sync images([   B]+1)

  if (A/=-1) call paste_cells(A, recvBufA)
  if (B/=-1) call paste_cells(B, recvBufB)
end subroutine
```

Listing 2: Simplified, MPI-like, buffered CAF code for exchanging ghost cells with direct neighbours `A` and `B` (`images_index()=rank+1`) in one direction (dummy neighbour `-1` at the borders of compute domain), with manual buffering.

```
subroutine exchange_ghost_cells(A, B)
  if (A/=-1 .and. B/=-1) sync images([A, B]+1)
  if (A/=-1 .and. B==-1) sync images([A   ]+1)
  if (A==-1 .and. B/=-1) sync images([   B]+1)

  if (A/=-1) call transmit_slice(A)
  if (B/=-1) call transmit_slice(B)

  if (A/=-1 .and. B/=-1) sync images([A, B]+1)
  if (A/=-1 .and. B==-1) sync images([A   ]+1)
  if (A==-1 .and. B/=-1) sync images([   B]+1)
end subroutine

subroutine transmit_slice(destRank)
  integer :: srcMin(3),srcMax(3),dstMin(3),dstMax(3)
  integer :: links(5)

  ! missing code: from destRank,
  ! compute srcMin, srcMax, dstMin, dstMax and links

  pdf(dstMin(1) : dstMax(1),
      dstMin(2) : dstMax(2),
      dstMin(3) : dstMax(3), links)[destRank] = &
  pdf(srcMin(1) : srcMax(1),
      srcMin(2) : srcMax(2),
      srcMin(3) : srcMax(3), links)
end subroutine
```

Listing 3: Unbuffered CAF communication for exchanging data with direct neighbours `A` and `B` in one direction (dummy neighbour `-1` at the borders of compute domain), with calculation of slice boundaries.

contrast, in the case of unbufferd CAF communication the sender has to calculate the boundaries of the corresponding halo slice on the receiver process (see the variables `dstMin` and `dstMax`).

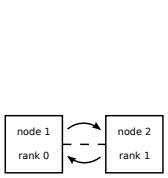
3. LOW-LEVEL BENCHMARKS AND PERFORMANCE MODEL

3.1 Experimental Set-up

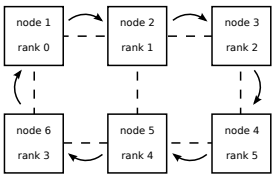
The LBM code was run on two different machines, the Cray XE6 of the Swiss National Supercomputing Center (CSCS) [11] and the "Lima" cluster of the Erlangen Regional Computing Center (RRZE) [12]. Table 1 summarizes the hardware and software characteristics of both systems.

	Cray XE6	Lima cluster
Processor	AMD 6172	Intel X5650
Clock frequency	2.20 GHz	2.67 GHz
DP peak per node	211 GFLOP/s	128 GFLOP/s
#Physical cores per node	24	12
#Virtual cores per node	N/A	24
#Sockets per node	2	2
#NUMA domains per socket	2	1
L3 cache size per NUMA domain	5 MB	12 MB
Measured memory bandwidth/node	50 GB/s	40 GB/s
Network topology	2D torus	Fat tree
Measured network BW per connection	10 GB/s	6 GB/s
#Nodes	176	500
Compiler	Cray Fortran 7.4.2	Intel Fortran 12.0 update 4
MPI	Cray Programming Environment 3.1.61	Intel MPI 4.0.3

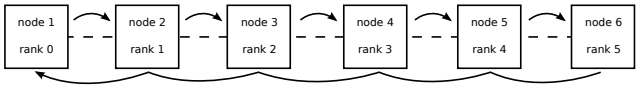
Table 1: Compute hardware and software data sheet



(a) Ringshift with 2 ranks, bidirectional communication



(b) Ringshift with 6 ranks on torus network: Optimal node selection results in unidirectional communication



(c) Ringshift with 6 ranks on torus network: Linear node selection results in bidirectional communication

Figure 2: Possible locations of nodes on the Cray XE6 torus network for the ringshift benchmark, which was used to mimic the halo exchange communication in the LBM implementation.

The process affinity was always chosen such that the amount of traffic that has to pass the inter-NUMA-domain connections (*Hypertransport* and *QuickPath*, respectively) was minimized.

The memory and network bandwidths in table 1 have been obtained by microbenchmarks (see sections 3.2.1 and 3.2.2). The XE6 has a torus network topology and the total network bandwidth available per node does therefore not only depend on the communication patterns but also on the selection of the nodes (see figure 2). Since there was no way to influence the node selection on the Cray machine, the network bandwidth given here refers to the bandwidth available per connection of two nodes.

The network interface of the Cray XE6 is based on the Gemini

```
double precision :: a(n), b(n)
for i=1..n
  a(i) = b(i)
end for
```

Listing 4: Copy benchmark with two memory streams

```
double precision :: a(n,19), b(n,19)
for i=1..n
  for l=1..19
    a(i,l) = b(i,l)
  end for
end for
```

Listing 5: Copy benchmark with $2 \cdot 19$ memory streams

interconnect [5] and promises to provide good hardware support for coarrays. The Lima cluster, on the other hand, can only support coarrays by appropriate software emulation.

3.2 Low-level Benchmarks

Low-level benchmarks are required to provide reasonable input data for the performance model [13] in section 3.3. This pertains to the main memory bandwidth and the latency and bandwidth for inter-node communication.

3.2.1 Memory Bandwidth

The benchmark codes used for the memory bandwidth measurements are shown in listings 4 and 5. We have used our own code instead of the popular STREAM benchmark, since the 19-way copy (listing 5) mimics the data access behavior of the LBM implementation. The code examples use the variable `n` to specify the size of the `double precision` arrays. The following sections will, however, use the variable $N = n \cdot 8\,\text{Bytes}$ to denote the array size in bytes. We have chosen the array sizes such that all data has to be streamed from and to main memory, since we are interested in memory-bound situations.

On modern cache-based architectures, a write miss in the cache hierarchy triggers a write-allocate transfer of the corresponding cache line from memory to cache. Hence, for every store to array `a` there is an implicit load, adding 50% to the overall data traffic. In principle, the write-allocate is not strictly required in the low-level benchmarks nor in the LBM implementation, since the data written to the target array will be evicted to memory anyway and is only required in the next update sweep. On x86 architectures, "non-temporal stores" can be used to bypass the cache hierarchy. However, those instructions have limitations that make them difficult to employ in the Lattice Boltzmann code [14] and were therefore not used in this work.

Table 2 summarizes the benchmark results, comparing the versions with 2 and $2 \cdot 19$ streams on both architectures. Whenever the number of processes was smaller than the number of available cores, a "scatter" strategy was used to distribute the processes across the resources, leading to an optimal use of bandwidth. Note that the 24-process version on the Lima node uses the SMT (a.k.a. Hyper-Threading) feature of the Intel Westmere processor. The bandwidth was calculated using the formula $B = \frac{3 \cdot N}{\text{runtime}}$ for two streams and according to $B = \frac{3 \cdot 19 \cdot N}{\text{runtime}}$ for $2 \cdot 19$ streams.

A general result of the low-level bandwidth measurements is that a large number of concurrent data streams reduces the available memory bandwidth by a measurable amount, especially when only half of the available cores are used on the Cray node. This justifies the decision to employ a dedicated $2 \cdot 19$-stream benchmark kernel for getting the baseline memory bandwidth of a node.

#Streams / Process	#Processes / Node	XE6, BW / Node [GB/s]	Lima, BW / Node [GB/s]
2	2	18.5	29.6
$2 \cdot 19$	2	8.4	16.1
2	12	51.9	40.1
$2 \cdot 19$	12	39.3	38.3
2	24	54.1	41.1
$2 \cdot 19$	24	51.9	38.9

Table 2: Memory bandwidth of the copy benchmarks (listings 4 and 5)

```
if (this_image() == 1) then
  sync all ! images 1 and 2 ready
  recv(:)[2] = send(:)[1]
  sync all ! recv[1] and recv[2] filled
end if
if (this_image() == 2) then
  sync all ! images 1 and 2 ready
  recv(:)[1] = send(:)[2]
  sync all ! recv[1] and recv[2] filled
end if
```

Listing 6: CAF ringshift using push strategy. The outer loop and the timing code are omitted.

3.2.2 Communication

A "ringshift" benchmark was used to model the communication characteristics of the Lattice Boltzmann implementation. The correctness of the results was validated by comparing them to the results of the Intel MPI Benchmarks [15].

The LBM requires, like other stencil codes, the exchange of boundary slices of multidimensional arrays between processes (see section 2). When the boundary slices can be accessed with unit stride, the expected communication time is predicted by the performance of the ringshift benchmark. Communication with strides greater than one does not occur for all the domain decomposition techniques and can therefore be avoided (at the cost of a higher communication volume though).

The bandwidths were measured for inter-node as well as intra-node situations. In case of inter-node measurements, there were always two distinct nodes taking part in the communication.

Listing 6 shows a simplified version of the benchmark code in CAF using a "push" strategy, i.e., remote transfers are writes to the distant image. The actual code takes care that no spurious intra-cache copy effects occur if the communicating processes reside on the same node (this corresponds to the `-off_cache` option in the Intel MPI Benchmarks [13]), and contains an outer loop that makes sure that the runtime is large enough for accurate measurements.

We assume a simple model for the message transfer time that has the latency L and the asymptotic bandwidth B as parameters:

$$T = L + \frac{2 \cdot N}{B}$$

Here N is the message length in bytes. L and B were determined by solving the following minimization problem:

$$L,B = \arg\min_{L,B} \left\| \begin{pmatrix} 1/T_1 & N_1/T_1 \\ 1/T_2 & N_2/T_2 \\ \vdots & \vdots \end{pmatrix} \cdot \begin{pmatrix} L \\ 2/B \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \\ \vdots \end{pmatrix} \right\|_2$$

The results show that while the model is suitable for two communication partners, it is only approximate for larger numbers. As a general result, Cray CAF is about as fast as MPI (figure 3). Inter-node CAF has a higher bandwidth for small numbers of processes, but the latency is worse than for MPI. Figure 4 reveals that the communication bandwidth for the Intel CAF implementation is extremely low; further tests suggested that all message transfers occur element-wise, and hence communication is always strictly latency-dominated (see below).

The ringshift benchmarks use the process affinity described in section 3.1. On Lima with MPI and on the XE6 with MPI and CAF the intra-node ringshift saturates at about half of the main memory bandwidth with 24 processes per node. The inter-node benchmarks already saturate the network with one process per node.

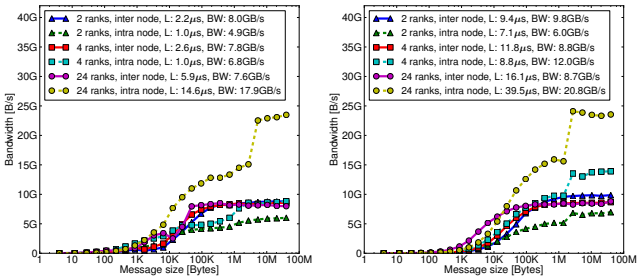
One might expect "pull" communication (i.e., remote transfers are reads from the distant image) to be slower than push communication because of the inherent synchronization between the communication partners; the initiator of the "pull" has to wait until the other image has sent the data. In contrast, push communication might be implemented in a non-blocking way by using buffering, thus decreasing the bandwidth. Measurements of the ringshift benchmark using pull communication showed that the performance characteristics of push and pull differed neither on the XE6 with Cray CAF nor on the Intel cluster with Intel CAF.

Figure 5 compares contiguous and element-wise ping-pong (i.e., unidirectional) communication for the two environments. The ping-pong benchmark was modified such that contiguous copy statements like `dst(:)[2] = src(:)[1]` were converted into loops that perform element-wise copy operations. No synchronization statements were inserted into the inner copy loop. The inner loop was obfuscated to make it difficult for the compiler to convert the loop into one contiguous copy statement at compile time.

If acceptable performance was still obtained in such a situation, this would mean that either the compiler does not perform communication instantaneously or that the network hardware aggregates multiple successive data transfers. As measurements for both compilers showed bad performance, none of this takes place. The XE6 inter-node communication bandwidth drops by a factor of 3000, while the intra-node bandwidth drops by a factor of 30. As expected, the Intel CAF results do not change, which substantiates our conjecture that contiguous communication is not supported.

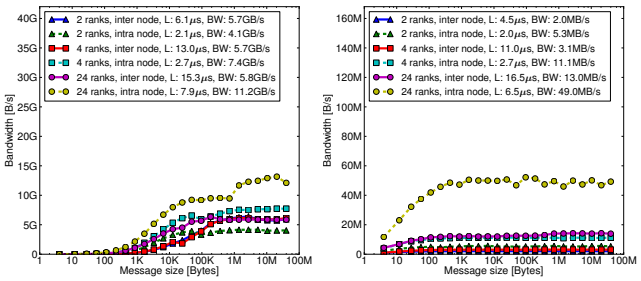
To check whether it is possible to overlap computation and communication with the available compilers, the ringshift benchmark was modified as shown in listings 7 and 8.

The benchmark was run for all possible combinations of `comm` and `comp`, except for `comm = comp = false`. Since the chosen computational kernel does not require any memory bandwidth, the process placement does not influence the results if `comm = false`. A message size of 4 MB was chosen, and the runtime of the `do_calculation()` subroutine was set such that it takes roughly the same time as the (bidirectional) transmission of a 4 MB message.



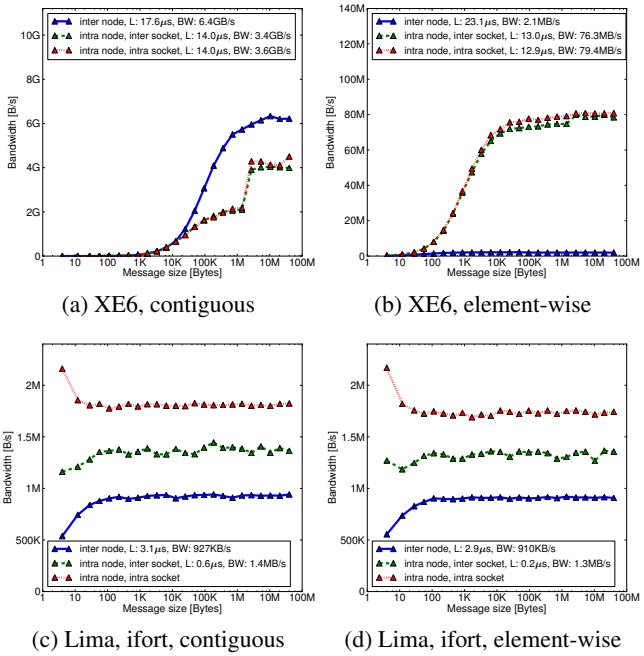
(a) MPI, ringshift (b) CAF, push ringshift

Figure 3: Results for the ringshift benchmark with Cray Fortran on the Cray XE6



(a) MPI, ringshift (b) CAF, push ringshift

Figure 4: Results for the ringshift benchmark with Intel Fortran on Lima



(a) XE6, contiguous (b) XE6, element-wise

(c) Lima, ifort, contiguous (d) Lima, ifort, element-wise

Figure 5: Explicitly contiguous vs. element-wise ping-pong communication

```
sync all
if (comm) array(:,2)[modulo(this_image(), &
  & num_images())+1] = array(:,1)
if (comp) call do_calculation()
sync all
```

Listing 7: "Push-style" ringshift benchmark with overlapping computation. The "sync all' is a global synchronization and requires all images to complete all operations before execution can proceed.

```
sync all
if (comm) array(:,2) = array(:,1) &
  & [modulo(this_image()-2, num_images())+1]
if (comp) call do_calculation()
sync all
```

Listing 8: "Pull-style" ringshift benchmark with overlapping computation

The results are as listed in table 3: Communication and computation overlap neither for push nor for pull communication, no matter if the processes are on the same node or not. Comparing the results when the time is measured inside or outside the code segment that is surrounded by `sync all` statements further reveals that the communication is performed instantaneously and is not scheduled and shifted to the next synchronization statement. Only the results on the XE6 are shown here because Intel CAF showed exactly the same behaviour.

The results for element-wise and overlapping communication suggest that it would not make sense to design a CAF implementation of the LBM algorithm that does not use ghost cells but accesses remote data directly. The performance of such a code would be strongly latency-bound.

3.3 Performance Model

For the considerations already shown for the ringshift in figure 2 and for the LBM method in figures 6b and 6c, the selection of the nodes has a big influence on the communication times of the LBM on the XE6. Therefore the performance model created here can only be applied strictly to fully non-blocking fat tree networks. In case of a torus network, the predicted communication overhead can only be a rough estimate.

Figure 6a is used to estimate the number of neighbours of a node. This means that the following assumptions are made.

Push/pull	Comm. enabled	Comp. enabled	Inter/Intra Node	Comm. BW [GB/s]	Runtime [μs]
don't care	no	yes	don't care		1278
push	yes	no	inter	9.6	806
push	yes	yes	inter	3.8	2080
push	yes	no	intra	6.2	1254
push	yes	yes	intra	3.0	2553
pull	yes	no	inter	9.8	805
pull	yes	yes	inter	3.6	2163
pull	yes	no	intra	6.2	1273
pull	yes	yes	intra	3.0	2551

Table 3: Test results for overlap of communication and computation on the Cray XE6. If overlap takes place, the effective bandwidths for the cases with and without computation should be similar.

- There exist exactly two process subdomains inside a node that share exactly one face with another process subdomain inside that node (ranks 0 and 23 in figure 6a).
- All other subdomains share exactly two faces with other process subdomains inside that node (ranks 1,...,22 in figure 6a).

This means that the subdomains on all processes inside a node are lined up along one coordinate direction (with the exception of boundary effects). While this is not optimal from the perspective of communication overhead, it is how MPI (and CAF) implementations usually map ranks to Cartesian topologies.

Together with the additional assumption that

- at least one node is fully surrounded by other nodes,

the two previous assumptions require that such a fully surrounded node communicates with $6 \cdot p - 2 \cdot (p-2) - 1 \cdot 2 = 4 \cdot p + 2$ inter-node neighbours if p is the number of processes per node (compare also to figure 6a). Also, inside each node, $2 \cdot (p-1)$ intra node communications take place. The following additional assumptions are made.

- The LBM kernel is memory bound, thus a simple bandwidth model [13] will suffice to predict the node-level performance. The saturation properties also make it mandatory to report scaled performance with respect to the number of nodes or sockets (as opposed to cores).
- The network is bidirectional with a bandwidth of $B_e/2$ in each direction and is fully non-blocking.
- The subdomain of every process is a box of the same size, meaning each process stores the same number of cells N^3 and that each array dimension is of the same size N.
- All nodes contain the same number of processes.
- Double precision numbers are used.

To summarize, let

P be the total number of processes

p be the number of processes per node

$L_{e/a}$ be the inter/intra node communication latency, measured with the ringshift benchmark in section 3.2.2,

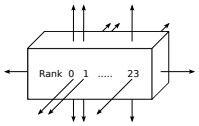
$B_{e/a}$ be the inter/intra node communication bandwidth, measured with the ringshift benchmark in section 3.2.2,

M be the memory bandwidth, measured with the copy benchmark with 19 streams in section 3.2.1, and

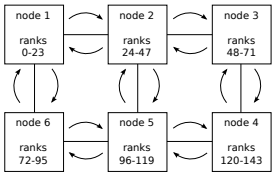
N be the number of lattice cells in each dimension of the subdomain stored by a process.

Then, the time required for one time step is composed of intra- and inter-node communication overhead and the time for the fluid cell updates:

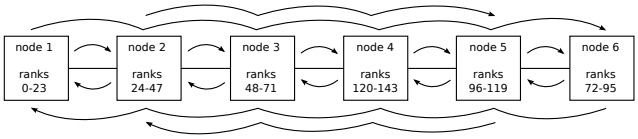
$$\begin{aligned} T_{\text{step}} &= T_{\text{comm}}^{\text{inter}} + T_{\text{comm}}^{\text{intra}} + T_{\text{calc}} \\ &= 3 \cdot L_e + \frac{5 \cdot 8\,\text{Bytes} \cdot N^2}{(B_e/2)/(4 \cdot p + 2)} \\ &\quad + 3 \cdot L_a + \frac{5 \cdot 8\,\text{Bytes} \cdot N^2}{(B_a/2)/(2 \cdot (p-1))} \\ &\quad + \frac{3 \cdot 19 \cdot 8\,\text{Bytes} \cdot N^3}{M} \end{aligned}$$



(a) Algorithmic neighbours of one compute node



(b) Optimal node placement for the LBM with a 2D domain decomposition on a torus network



(c) Suboptimal node placement for the LBM with a 2D domain decomposition on a torus network

Figure 6: Parallel LBM implementation

The performance S, i.e., the number of lattice site updates per second, is

$$S[\text{LUPS/s}] = P \cdot \frac{N^3}{T_{\text{step}}}$$

4. APPLICATION BENCHMARKS AND VALIDATION OF PERFORMANCE MODEL

The analysis of the performance of the LBM implementations starts with determining the number of processes required per node to saturate the memory system. The optimal "node filling" factor is then used in the subsequent scaling runs. We compare CAF and MPI, using the Cray compiler on the XE6 and the Intel compiler on the Westmere Cluster. The measured performance is compared to the prediction generated by the performance model from section 3.3.

4.1 Optimal Single Node Performance Evaluation

To find out how many processes are required per node to saturate the memory system, figure 7 shows the LUPS/s achieved by one compute node for different numbers of processes per node. Each process was assigned a compute domain of 400 MB, no matter how many processes were running on each node. Intra-node MPI communication was taking place, but the communication time was subtracted from the total runtime before the performance metric was calculated.

The Lima cluster has only 12 physical cores per node and 12 virtual hyperthreaded cores, in contrast to the XE6 with 24 physical cores. However, both the Lima and the XE6 need 24 processes on each node (i.e., fully populated NUMA domains) to saturate the memory system. This picture may change if some effort is invested in SIMD vectorization, but this option was not investigated here (see [14] for a more in-depth analysis).

4.2 Strong Scaling

This section shows which domain partitioning approaches are best suited for strong scaling runs of each of the three different implementations (MPI, MPI-like buffered CAF and unbuffered CAF). The symbol (x,x,1) in the legends of figures 8 and 9 corresponds to a 2D domain decomposition along the first (and fast) axis and the second fastest axis of the Fortran array, while (x,x,x) corresponds to a 3D domain decomposition. All "x" have the same size, meaning the decomposition is done equally along each axis.

The problem domain contains 350^3 lattice cells, which makes $2 \cdot 8 \cdot 19 \cdot 350^3$ Bytes ≈ 13 GB in total. A 1D domain position was tested but resulted in very poor performance due to a very unfavourable fraction of required computation to communication, as expected.

Figures 8 (for Cray XE6) and 9 (for Lima) show strong scaling runs on up to 63 nodes. In order to increase the sampling rate, the domain size was increased in each dimension up to the next number divisible by the number of ranks in that dimension. If the resulting computational volume was more than 10% larger than the original computational volume, the benchmark was not run.

On the XE6 with MPI, the best 2D domain decomposition, the (1,x,x) decomposition, and the 3D decomposition are equally fast. The 2D decompositions that cut along the fast axis are slower, and equally slow.

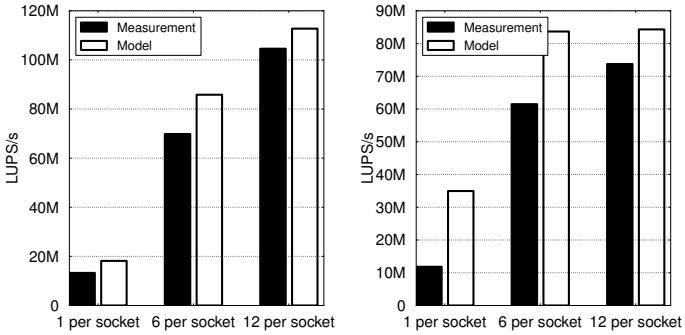
The buffered, MPI-like CAF implementation shows nearly the same performance characteristics as the MPI version and both implementations are equally fast on up to about 30 nodes. The CAF performance is slightly worse than MPI for larger node counts. 30 nodes corresponds to a message size of $5 \cdot 8\text{Bytes} \cdot 350^2/\sqrt{24 \cdot 30} \approx 180$ kB for the buffered CAF implementation, which is, according to results of the ringshift benchmark from section 3.2.2, still a message size large enough to hide CAF's higher latency. No further investigations were made to find the reason for the differences in predicted and measured performance for Cray CAF.

The unbuffered CAF implementation performs worse than MPI and buffered CAF when a true 3D domain decomposition is used, and even worse if the unfavourable 2D decompositions (x,x,1) and (x,1,x) are used. The (1,x,x) decomposition is about as fast as for the buffered CAF version.

As mentioned before, the performance model can only provide a rough estimate for the communcation time on the XE6 due to its torus network. The model works best for (1,x,x) decomposition and MPI, as shown in figure 8a. In particular, it can track the variations due to communication topology between, e.g., 20 and 27 nodes. On the other hand, the model is mediocre for the (x,x,x) decomposition; it is well known that the bandwidth performance model for the LBM update step (and stencil codes in general) does not encompass variations due to inner loop length [13]. In general, single-node performance drops when the inner loop becomes small. This explains the larger deviation from the model in the (x,x,x) case. The deviation also becomes larger for the CAF versions (figures 8b,c), but is still inside a 10%-20% range in the optimal (1,x,x) case.

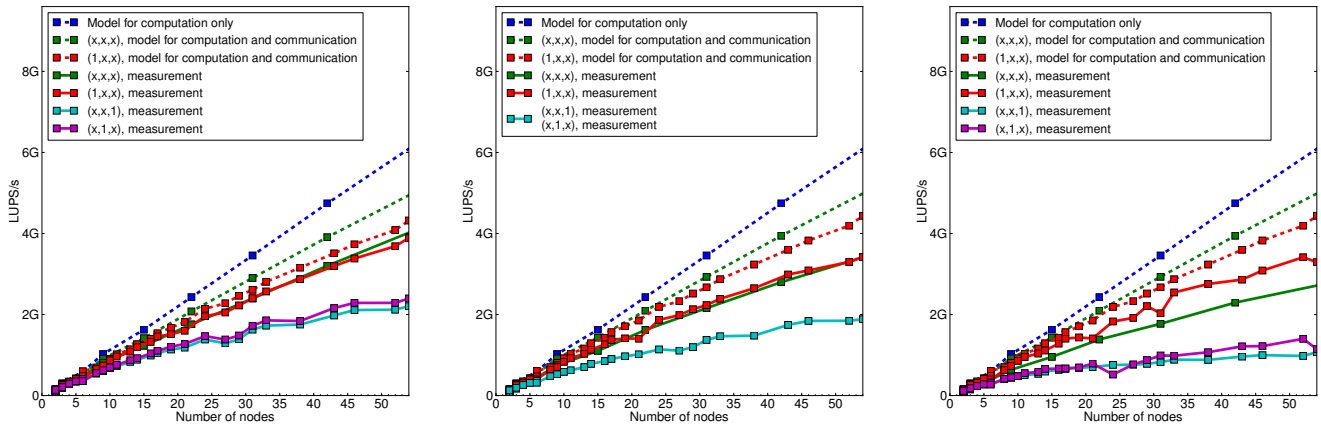
In contrast to the Cray XE6, the Intel MPI implementation performs best on the (1,x,x) domain decomposition. Due to the low CAF communication bandwidth (see section 3.2.2), the CAF parallelization is latency-bound on Lima and is 40 times slower than the reference MPI implementation for large numbers of processes. This means that the time used for manual buffering does not affect the overall performance and buffered CAF is as fast as unbuffered CAF.

As element-wise communication seems to take place (see section 3.2.2), only the dimensionality of the decomposition affects the performance. Therefore, the 3D domain decomposition shows the best performance, and all 2D domain partitioning approaches are slower and have equal performance. The model predicts a performance that is about twice as large as the true performance of the



(a) AMD Opteron 6172 (Cray XE6)(b) Intel Xeon X5650 (Lima cluster)

Figure 7: Measurements and model prediction for LBM LUPS/s per node without communication. A domain size of $110 \times 110 \times 110$ per rank was chosen (400 MB per rank).



(a) MPI (b) CAF, MPI-like, buffered (c) CAF, unbuffered

Figure 8: Strong scaling performance results and effects of 2D/3D domain decomposition on the Cray XE6

LBM. This effect could also be seen in the weak scaling results addressed by subsection 4.3, but the weak scaling results are not discussed in detail in this paper. Due to Intel CAF's overall poor performance, no further investigations were conducted to find the reason for the differences between the modelled and the true performance.

4.3 Weak Scaling

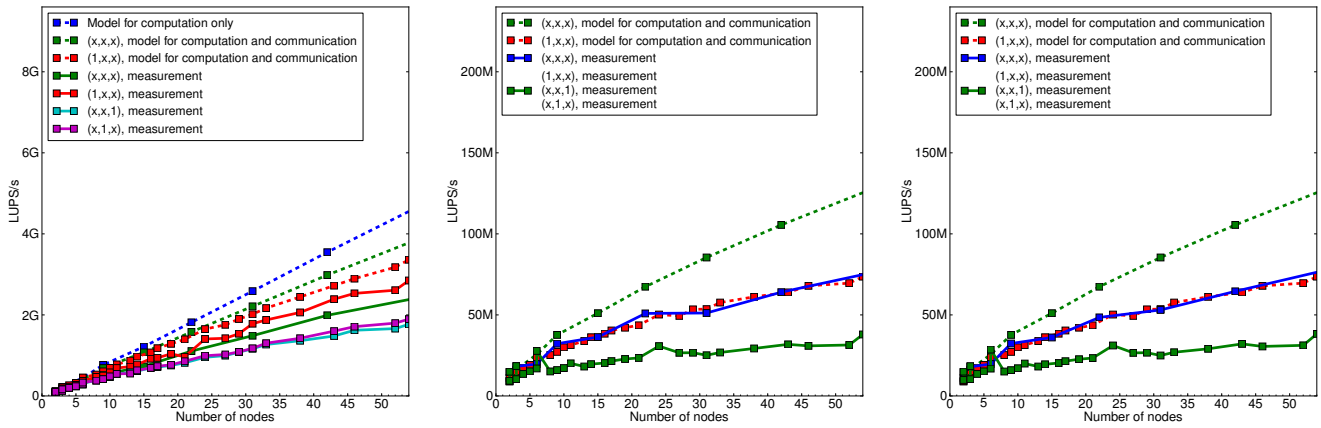
A weak scaling run using a 2D domain decomposition was benchmarked on Lima with MPI and CAF. Each process operated on a domain of size $96 \times 96 \times 96$, which corresponds to a memory requirement of 6.5 GB per full compute node. Weak scaling of LBM shows, by design, communication overhead that is solely a function of the Cartesian topology, i.e., the per-face communication volume and hence the effective bandwidth is always the same. Our results reflected this property, so we are not going into detail here.

5. RELATED WORK

Lattice Boltzmann methods for numerical simulation of fluids have been developed for more than twenty years, with growing popularity in recent years. Their parallelization with MPI or hybrid MPI and OpenMP is well understood, and extensive research has already been done on optimized implementations, e.g. in [9, 10].

There is not much work published on implementing LBM codes in Coarray Fortran and comparing it with MPI. The approach closest to our paper is described in [16], investigating different implementation schemes w.r.t. data structures and concluding, similarly to us, that the MPI-like buffered CAF implementation performs best, though not better than MPI. In contrast to our paper, experiments were only done using recent Cray machines, and no performance model was developed.

Taking a broader view, a still relatively small number of publications exist on porting diverse application codes from MPI to Coarray Fortran, though these case studies are not necessarily directly comparable to LBM. In [17], the CGPOP mini-app, a conjugate gradient method based on 2D arrays with exchange of ghost cell layers, is implemented in CAF using various strategies, e.g. with buffered or unbuffered, push or pull data transfer. Benchmarking CAF against MPI on several Cray systems, the authors could not find a performance benefit of CAF over MPI. Finite Differencing Methods are investigated in [18], where a 5-point stencil performed best on a Cray XE1 in the CAF MPI-style implementation, depending on the boundary lengths. For a 9-point stencil, the MPI implementation proved best. Other examples can be found in [19] where the CENTORI fluid simulation and a sparse matrix multiplier code are analyzed, or in [20], using the SBLI code (also known as



(a) MPI (b) CAF, MPI-like, buffered (c) CAF, unbuffered

Figure 9: Strong scaling performance results and effects of 2D/3D domain decomposition on the Intel Westmere cluster "Lima"

PDNS3D), a finite difference formulation of direct numerical simulation of turbulence.

Summarizing, in general, authors agree that porting an MPI code to CAF is fairly straightforward, especially when adopting the MPI-like buffered style, and that the CAF programming model may be easier to use than MPI. CAF has some restrictions limiting usability, though, such as requiring the coarray to be of equal length on each image, making it more suitable for regular codes. For real applications, the MPI performance seems better in most cases, though CAF can get close when the best-possible data transfer implementation is chosen. For certain low-level benchmarks CAF can perform better than MPI, regarding e.g. communication times for certain message sizes. Experiments were usually conducted on Cray systems, as they offer good hardware and software support for the CAF programming model. [21] evaluates the Cray Gemini interconnect using low-level and NAS benchmarks. To our best knowledge, the other research papers do not take into account Intel compilers, and they do not develop performance models, based on characteristics derived from low-level benchmarks, as we do in our paper.

6. CONCLUSIONS

We have investigated the performance of contemporary Coarray Fortran programming solutions by means of low-level benchmarks and evaluated their suitability for use in application codes. Our application testcase is based on replacing a former MPI parallelization of an existing prototype 3D Lattice Boltzmann code by a parallelization with coarrays. The performance of the Lattice Boltzmann MPI implementation was compared to a CAF version incorporating the manual buffering taken from the original MPI version and to a simpler, pure CAF implementation without manual buffering. We evaluated coarrays with the Cray Compiler on a Cray XE6 and the Intel Fortran Compiler on an *Infiniband* Intel Westmere Cluster.

For the task of parallelizing our prototype 3D Lattice Boltzmann code, CAF turned out to be slightly easier to program than MPI, but the Cray Compiler was the only compiler that was sufficiently stable and generated communication code that was efficient enough to be considered an alternative to the MPI parallelization. On the node level we checked the quality of the generated code by comparing with a bandwidth-based performance model, which revealed that on both systems the LBM implementation uses almost the full available memory bandwidth if all cores are utilized (including virtual cores on the Intel Westmere processor).

Low-level communication benchmarks revealed that the Cray CAF compiler was slower than MPI for small messages due to a higher communication latency, but was faster than MPI for large messages. In practical applications, the very large message sizes where CAF is faster than MPI is only seen when each process' subproblem is so large that the runtime is clearly dominated by computation. The current Intel Compiler was not able to produce competitive communication code. Up to now, its executables show effective bandwidths that are three orders of magnitude below native MPI due to element-wise data transfers. For application developers this means that productive use of CAF applications is currently limited to Cray hardware unless they have communication requirements that are so low that the Intel compiler becomes an option. However, in terms of correctness and features, the Intel CAF implementation is currently the only alternative to Cray CAF that we know of.

On the Cray XE6, the optimal kind of domain decomposition for parallelizing the Lattice Boltzmann algorithm using unbuffered coarray communication was found to be a 2D domain decomposition along the slowest and the second-slowest array dimension. The domain decomposition could also be replaced by a 3D domain decomposition if manual buffering was used to collect the data before communication. Those findings certainly hold also for other stencil codes operating on regular grids. Via a comprehensive performance model, using the measured ring-shift communication performance, the communication topology, and the bandwidth-based node-level model as inputs, we were able to predict the qualitative scaling behavior of the CAF and MPI implementations on the Cray system.

Finally, we have confirmed the expected result that achieving performance levels similar to a well-optimized MPI code requires the CAF programming style to closely match the MPI style. Expecting the compiler to sort out optimal strategies for message aggregation and buffering is too optimistic, at least with currently existing implementations.

Acknowledgements

The work presented in this paper was partly conducted at the *Exascale Computing Research Center*, with support provided by CEA, GENCI, Intel, and UVSQ. Any opinions, findings, and conclusions

or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, GENCI, Intel or UVSQ. The work was also supported by KONWIHR, the Competence Network for Scientific High Performance Computing in Bavaria, within the OMI4papps project. We are indebted to the Swiss National Supercomputing Center (CSCS) in Manno for granting access to their Cray XE6 system.

7. REFERENCES

[1] R. W. Numrich and J. Reid. *Co-Array Fortran for parallel programming*. ACM FORTRAN FORUM **17(2)**, (1998) 1–31.

[2] ISO. *Fortran 2008 Language Draft, ISO/IEC JTC 1/SC 22/WG 5/N1826*. `ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1826.pdf`, 2010.

[3] Q. S. W. Ltd. *Shmem Programming Manual*. `http://staff.psc.edu/oneal/compaq/ShmemMan.pdf`, 2001.

[4] *CF90 Co-array Programming Manual*. `http://docs.cray.com/books/004-3908-001/004-3908-001-manual.pdf`, 1998.

[5] R. Alverson, D. Roweth and L. Kaplan. *The Gemini System Interconnect*. In: *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)* (ACM). ISBN 978-1-4244-8547-5, 83–87, (2010).

[6] C. S. Baw, R. D. Chamberlain, M. A. Franklin and M. G. Wrighton. *The Gemini Interconnect: Data Path Measurements and Performance Analysis*. In: *Proceedings of the The 6th International Conference on Parallel Interconnects*, PI '99 (IEEE Computer Society, Washington, DC, USA). ISBN 0-7695-0440-X, 21–30, (1999). `http://portal.acm.org/citation.cfm?id=826032.826711`

[7] T. Zeiser, G. Hager and G. Wellein. *Benchmark Analysis and Application Results for Lattice Boltzmann Simulations on NEC SX Vector and Intel Nehalem Systems.* Parallel Processing Letters , (2009) 491–511.

[8] C. Feichtinger, S. Donath, H. Köstler, J. Götz and U. Rüde. *WaLBerla: HPC software design for computational engineering simulations*. Journal of Computational Science **2(2)**, (2011) 105–112. ISSN 18777503. `http://dx.doi.org/10.1016/j.jocs.2011.01.004`

[9] S. Donath. *On Optimized Implementations of the Lattice Boltzmann Method on Contemporary High Performance Architectures*. Master's thesis at the University of Erlangen-Nuremberg, 2004.

[10] G. Wellein, T. Zeiser, G. Hager and S. Donath. *On the single processor performance of simple lattice Boltzmann kernels*. Computers & Fluids **35(8-9)**, (2006) 910–919. ISSN 0045-7930. Proceedings of the First International Conference for Mesoscopic Methods in Engineering and Science. `http://www.sciencedirect.com/science/article/pii/S0045793005001532`

[11] *Palu, CSCS Website*. `http://user.cscs.ch/hardware/palu_cray_xe6/index.html`.

[12] *Lima, RRZE Website*. `http://www.rrze.uni-erlangen.de/dienste/arbeiten-rechnen/hpc/systeme/lima-cluster.shtml`.

[13] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers* (CRC Press, Inc., Boca Raton, FL, USA), 1st ed., 2010. ISBN 978-1439811924.

[14] M. Wittmann, T. Zeiser, G. Hager and G. Wellein. *Comparison of Different Propagation Steps for Lattice Boltzmann Methods*. Computers & Mathematics with Applications (Proc. ICMMES 2011) . `http://dx.doi.org/10.1016/j.camwa.2012.05.002`

[15] *Intel MPI Benchmarks*. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`.

[16] M. Hasert, H. Klimach and S. Roller. *CAF versus MPI - applicability of coarray fortran to a flow solver*. In: *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11 (Springer-Verlag, Berlin, Heidelberg). ISBN 978-3-642-24448-3, 228–236, (2011). `http://dl.acm.org/citation.cfm?id=2042476.2042502`

[17] A. Stone, J. M. Dennis and M. Strout. *Evaluating Co-Array Fortran with the CGPOP Miniapp*. In: *Fifth Partitioned Global Address Space Conference* (Galveston, TX), (2011).

[18] R. Barrett. *Co-Array Fortran Experiences with Finite Differencing Methods*. In: *Cray User Group 2006*. (2006).

[19] A. Myers. *Coarray Fortran in CENTORI and a Sparse Matrix Multiplier Code*. Master's thesis, University of Edinburgh, 2008.

[20] J. V. Ashby and J. K. Reid. *Migrating a Scientific Application from MPI to Coarrays*. Tech. rep., STFC, 2008.

[21] H. Shan, N. J. Wright, J. Shalf, K. Yelick, M. Wagner and N. Wichmann. *A preliminary evaluation of the hardware acceleration of the cray gemini interconnect for PGAS languages and comparison with MPI*. In: *Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems*, PMBS '11 (ACM, New York, NY, USA). ISBN 978-1-4503-1102-1, 13–14, (2011). `http://doi.acm.org/10.1145/2088457.2088467`