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

Masters Thesis in Computational Engineering

submitted
by

Klaus Sembritzki

born 16.05.1986 in Erlangen

Written at the

Friedrich-Alexander-University of Erlangen-Nürnberg

Advisor: Prof. Dr. Gerhard Wellein, Dr.-Ing. Jan Treibig

Started: 1.8.2011

Finished: 31.1.2012

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 31. Januar 2012

Übersicht

Die Suche nach Programmierparadigmen, welche den Quelltext paralleler Programme robuster, leserlicher und wartbarer gestalten, hat zur Entwicklung sogenannter PGAS (Partitioned Global Address Space) Sprachen geführt. Typische Beispiele sind UPC (Unified Parallel C) und CAF (Coarray Fortran). Bisher haben diese Sprachen jedoch noch nicht den Weg aus dem akademischen Umfeld in Produktionsumgebungen gefunden.

Mit dem Fortran 2008 Standard sind Coarrays ein nativer Bestandteil von Fortran geworden. Die Popularität von Fortran im wissenschaftlichen Rechnen und die große Fortran Code Basis lässt darauf hoffen, dass für die Parallelisierung von bereits existierendem Fortran Code und für das Schreiben von neuem Quelltext von Entwicklern die native Sprachvariante einer MPI/OpenMP Parallelisierung vorgezogen wird. Dies ist jedoch nur möglich, wenn sowohl der Sprachstandard für das wissenschaftliche Anwendungsfeld ausreichend ist als auch die zur Verfügung stehenden Compiler und Entwicklungsumgebungen den Anforderungen des wissenschaftlichen Rechnens nach Performance und Zuverlässigkeit gerecht werden. Die vorliegende Master Arbeit untersucht anhand der Portierung eines prototypischen, MPI basierten Lattice Boltzmann Lösers nach Coarray Fortran und anhand von „low-level“ Benchmarks für unterschiedliche Kombinationen aus gängiger HPC Hardware und Coarray Fortran Compilern, inwieweit diese Kriterien zur Zeit der Arbeit erfüllt sind.

Abstract

The search for programming paradigms that promise to make the source code of parallel programs more robust, readable and maintainable has led to the development of so-called PGAS (Partitioned Global Address Space) languages, typical represantatives being UPC (Unified Parallel C) and CAF (Coarray Fortran). The languages have, however, not yet made their way from research into production codes.

Coarray Fortran may be able to gain greater popularity since it is, starting with the Fortran 2008 language standard, natively embedded into the Fortran programming language. This gives rise to the hope that decent coarray compiler implementations will be available and make it feasible to actually use the language features instead of MPI/OpenMP. This is, however, only possible if the language standard is sufficient for the scientific application field and if the available compilers and development environments fulfill the requirements for scientific computing, which are primarily performance and reliability. This master thesis checks those criteria for different combinations of popular HPC hardware and Coarray Fortran compilers by porting a prototype, MPI-based Lattice Boltzmann solver to coarray Fortran and by evaluating the results of different low-level benchmarks.

Acknowledgments

The thesis was partly written at the *Exascale Computing Research Center* of the *Université de Versailles-St Quentin*. I would like to thank my advisors, which are *Dr. Jan Treibig* from *RRZE (Erlangen Regional Computing Center)* and *Bettina Krammer* (Exascale Computing Research Center) for giving me valuable information to define the scope of the thesis, develop the thesis and for establishing the collaboration between both institutions. Thanks go to *Prof. Gerhard Wellein* from the RRZE for offering me the thesis and to *Prof. William Jalby* from the Exascale Lab for his support in France. Finally I would like to thank *Dr. Georg Hager* from the RRZE for his administrative, technical and editorial support.

Contents

Chapter 1

Introduction

The wide availability of parallel compute resources in the 1990s motivated the development of different programming paradigms to efficiently run programs on these architectures. In 1994, the first standard for *MPI (Message Passing Interface)* [MPI 09], a programming library providing support for efficient message passing, was released by a joint effort of researchers and vendors for parallel computers and, in 1997, a consortium of hardware and software vendors released the specification for OpenMP [Open 11], a compiler extension providing threading support by compiler directives. Those two parallelization methods have become the de facto standards for parallel programming in numerical simulation and the development of both specifications is an ongoing process. In contrast, *PGAS (Partitioned Global Address Space)* languages like *UPC (Unified parallel C)*, the Coarray Fortran programming language defined in 1998 by Robert Numrich and John Reid [Numr 98] [ISO 10], and the SHMEM library [Ltd 01] have not managed to gain greater acceptance outside academia. Still, 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.

Like all the other languages and libraries mentioned, Coarray Fortran uses an *SPMD (Single Program, Multiple Data)* approach. Multiple instances of the same program are executed on (possibly) distributed processors, each of the instances being called an "image". All images run asynchronously and work on their private part of the data. Synchronization and data transfer is specified explicitly in the source code. With the Fortran 2008 standard, coarrays became a native Fortran feature [ISO 10]. Data is distributed among the images by adding so-called "codimensions" to variables, where each element in a codimension is stored by exactly one image. An image can access data of another image by specifying the codimension of the variable. On shared memory systems, a compiler can treat codimensions like ordinary array dimensions,

thus store coarrays consecutively in the shared memory and access them very efficiently. On distributed memory architectures, a protocol like MPI or the SHMEM library can be used to perform the communication.

In the last few years, research regarding the Lattice Boltzmann method has been conducted at the *University of Erlangen Nürnberg*. In particular, two large scale production codes have been developed at the *RRZE (Erlangen Regional Computing Center)* [Zeis 09] and the *LSS (Chair for System Simulation)* [Feic 11]. 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 [Dona 04, Well 06], 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 implementation. 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.

Lattice Boltzmann methods 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. In the D3Q19 model implemented in the 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.

Cray has a long tradition with PGAS languages, starting with the support for coarrays on the Cray T3E in 1998 [CF90 98]. 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 the work is therefore a Cray XE6 machine with *Gemini* routers [Alve 10] [Baw 99].

The second hardware used is a standard *Infiniband* Intel Westmere Cluster. On this hardware,

the Intel Fortran Compiler 12, update 4, was chosen as one of the compilers, which can be considered a very common choice among software developers. Also tested were the Rice University CAF 2.0 compiler and a development version of the Open64 compiler 4.2 with CAF support. A production quality Open64 version with CAF support was not yet released at the time of writing.

This thesis is organized as follows. Chapter 2 introduces CAF in the context of existing parallel programming approaches. Chapter 3 describes the hardware and software environments that were used. Chapter 4 shows performance characteristics of the different hardware/software combinations introduced in chapter 3 by means of low-level benchmarks. In chapter 5, the Lattice Boltzmann Method and the implementation used for the thesis are introduced. The performance of the Lattice Boltzmann algorithm is reported in chapter 6. Finally, chapter 7 draws conclusions about the experiences made and the applicability of the CAF programming model to other algorithms.

Chapter 2

Parallel Programming Models

This chapter introduces MPI and OpenMP and compares them to Coarray Fortran. The serial code shown in listings 2.1 and 2.2 is parallelized with all of the programming models to show some substantial differences. The code computes the sum of two double precision vectors, a and b, and stores the result in another vector, c. This shows differences in the data distribution of the different parallelization approaches. A subsequent reduction operation is performed on c to illustrate the usage of special purpose functions for this task.

2.1 OpenMP

OpenMP introduces compiler directives to add multiprocessor support to existing serial programs running on shared memory systems. Outside the regions marked by OpenMP directives, the program is executed like a serial program by exactly one master thread (see figure 2.1a). On entering an *!$OMP PARALLEL* block, worker threads are activated and execute the code segment surrounded by the directives concurrently. All worker threads operate on the same address space

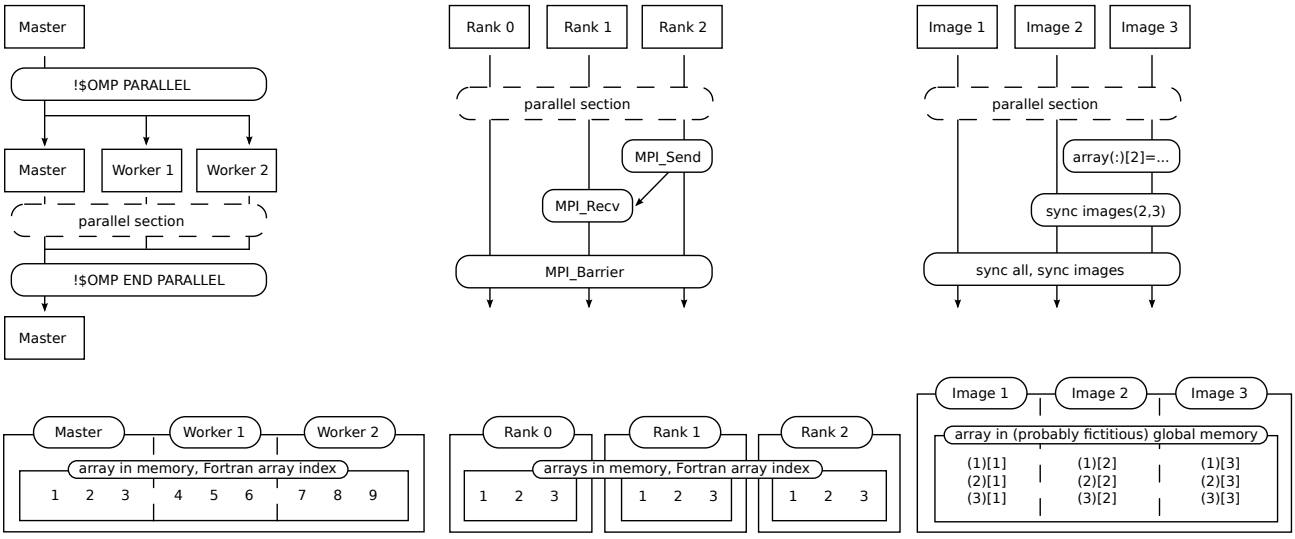
```
double precision, dimension(N) :: a,b,c
double precision :: sum_c

! initialize vectors a and b
call initialize_a_b(a,b)

c(:) = a(:) + b(:)

sum_c = sum(c(:))
```

Listing 2.1: Example serial program in vector notation



(a) OpenMP (b) MPI (c) Coarray Fortran

Figure 2.1: Control flow and memory layout

with the same start address, array indices are indices into the global array. Implicit synchronization is performed at the end of the `!$OMP PARALLEL` block, explicit synchronization directives are also available. Code example 2.3 shows the original code example enhanced by OpenMP pragmas.

As non-local data is accessed transparently, the programming model is best suited for shared memory systems having *Uniform Memory Access (UMA)*. While still providing cache coherence, modern shared memory architectures assign each memory page to one specific processor. Accesses to each memory page are therefore non-uniformly fast among the different processors, an architecture which is referred to as *cache-coherent Non-Uniform Memory Access (ccNUMA)* [Hage 10]. NUMA penalties are therefore not clearly visible in the source code.

Also, the OpenMP programming model can not be efficiently extended to situations where non-local reads/writes can not be performed with the same assembly instructions that are used for local reads and writes, which is the case when different computers are connected through network hardware like Infiniband or the Gemini Interconnect.

2.2 MPI

MPI is a software library that provides message passing routines for shared and distributed memory systems and is the de facto standard for distributed memory parallelization. All workers run

```
double precision, dimension(N) :: a,b,c
double precision :: sum_c
integer :: i

! initialize vectors a and b
call initialize_a_b(a,b)

do i = 1, n
    c(i) = a(i) + b(i)
end do

sum_c = 0.0

do i = 1, n
    sum_c = sum_c + c(i)
end do
```

Listing 2.2: Example serial program in index notation

```
double precision, dimension(N) :: a,b,c
double precision :: sum_c
integer :: i

call initialize_a_b(a,b)

!$OMP PARALLEL DO PRIVATE(i)
    do i = 1, N
        c(i) = a(i) + b(i)
    end do
!$OMP END PARALLEL DO

! method 1: use built-in OpenMP reduction operation
sum_c = 0.0

!$OMP PARALLEL PRIVATE(i) REDUCTION(+:sum_c)
    do i = 1, N
        sum_c = sum_c + c(i)
    end do
!$OMP END PARALLEL REDUCTION

! method 2: nonparallel manual reduction
sum_c = 0.0

do i = 1, N
    sum_c = sum_c + c(i)
end do
```

Listing 2.3: Example OpenMP program

independently of each other from the start of the program and can have an address space of their own. The developer manually takes care of non-local data access by inserting send/receive pairs or special purpose routines like broadcasts or reductions (see figure 2.1b). Providing greater flexibility, parallelizing a code with MPI typically requires writing many more lines of code than for an equivalent OpenMP parallelization because

- the domain decomposition has to be done by hand,
- communication calls have to be inserted,
- compound data types have to be registered and
- manual buffering or registration of strided vectors has to be implemented for many domain decomposition algorithms.

NUMA effects and the fact that non-local data accesses are slow are naturally regarded by MPI code. In theory, MPI provides means for overlapping communication and computation with non-blocking send operations. Most MPI implementations do, however, start the actual communication only when the receiver calls `MPI_Wait` to wait for the data to arrive. Listing 2.4 shows different variants of an MPI parallelization for the example code.

```
integer :: isize, rank, ierr, recv_rank, receive_status(MPI_STATUS_SIZE)
double precision, dimension(:), allocatable :: a,b,c
double precision :: sum_c_local, sum_c, sums_c_local(:)
integer :: i, indexReady, receiveRequests(:), sendRequest

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, isize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

! allocate local parts of the arrays, take care of the
! sizes of the last rank if modulo(N, isize) .ne. 0
allocate( a(1+min(((N-1)/isize+1)*(rank+1), N) - ((N-1)/isize+1)*rank) )
allocate( b(1+min(((N-1)/isize+1)*(rank+1), N) - ((N-1)/isize+1)*rank) )
allocate( c(1+min(((N-1)/isize+1)*(rank+1), N) - ((N-1)/isize+1)*rank) )

call initialize_a_b(a,b)

do i = 1, size(c)
    c(i) = a(i) + b(i)
```

```
end do

sum_c_local = 0.0

do i = 1, size(c)
    sum_c_local = sum_c_local + c(i)
end do

! sum up the local contributions and store the result in rank 0

! method 1: use built-in reduction operation
call MPI_Reduce(sum_c_local, sum_c, 1, MPI_DOUBLE, &&
     MPI_SUM, 0, MPI_COMM_WORLD, ierr)

! method 2: synchronous, manual reduction without reduction tree
sum_c = sum_c_local
if (rank == 0) then
    sum_c = sum_c_local

    do recv_rank = 1,isize-1
        call MPI_Recv( sum_c_local, 1, MPI_DOUBLE, &&
             recv_rank, 42, MPI_COMM_WORLD, receive_status )
        sum_c = sum_c + sum_c_local
    end do
else
    call MPI_Send( sum_c_local, 1, MPI_DOUBLE, 0, 42, MPI_COMM_WORLD )
end if

! method 3: asynchronous, manual reduction without reduction tree
sum_c = sum_c_local
if (rank == 0) then
    allocate(sums_c_local(0:isize-1))
    allocate(receiveRequests(0:isize-1))
    do i = 0,isize-1
        call mpi_irecv(sums_c_local(i), 1, MpiDataTypeReal, i+1, 42, &&
             MPI_COMM_WORLD, receiveRequests(i))
    end do
    do i = 0, isize-1
        call MPI_Waitany(isize, receiveRequests, indexReady, MPI_STATUS_IGNORE)
        sum_c = sum_c + sums_c_local(indexReady)
    end do
```

```
else
    call mpi_isend(sum_c_local, 1, MpiDataTypeReal, 0, 42, MPI_COMM_WORLD, sendRequest)
    call MPI_Wait(sendRequest, MPI_STATUS_IGNORE)
end if
```

Listing 2.4: Example MPI program, without error handling

2.3 Coarray Fortran

Coarray Fortran gives the programmer the illusion of having a global address space with NUMA behaviour by adding so-called codimensions to variables. Each element of a codimension is stored by the image with the corresponding `image_index()`. The image index is similar to the MPI rank. Note that the first CAF image has index 1, while the first MPI process has rank 0. When the local part of a coarray is referenced, the codimension can be omitted (see listing 2.5). Remote accesses require usage of the special brackets for the codimensions indicating possibly slow accesses. Like in MPI programs, all workers run independently from each other and are only synchronized by explicit synchronization constructs (or coarray allocations and deallocations). If the underlying hardware is a true shared memory system that could also be used to run OpenMP parallel programs, coarrays can be mapped to true arrays inside the shared memory, regarding the codimension as a true array dimension (see figure 2.1c). As all access to remote data is explicitly specified in the code with the brackets identifying the codimensions, distributed memory hardware can emulate the global address space by issuing library calls behind the scenes.

For the programmer, the biggest differences to MPI are

- the native support for compound types and multidimensional arrays,
- the inability to do unequal work distribution among different ranks with standard coarrays (unequal work distribution is only possible by using compound types with allocatable or pointer components, more in section 2.3.2),
- the fact that remote write operations do not require a corresponding receive operation from the remote image and that remote read operations do not require a corresponding send operation,
- the lack of communicators (image lists can be used as an alternative).

```
integer, dimension(3), codimension[*] :: array

array = ...

if (this_image() == 1) then
    all(array == array[1])
end if
```

Listing 2.5: Optional codimensions on local image

All information about the features of Coarray Fortran can be found in the Fortran 2008 Language Draft [ISO 10]. [Reid 10] and [Done 07] contain considerations that led to the design of the language features.

2.3.1 Segments and Synchronization Statements

Three synchronization statements are provided, being **sync** memory, **sync all** and **sync images**. **sync all** is equivalent to an MPI_Barrier(MPI_COMM_WORLD), **sync images** is equivalent to an MPI_Barrier(userdefined_communicator) where userdefined_communicator is a communicator consisting of all the images specified in the image list that is passed as an argument to **sync images**.

The code between two **sync** memory statements is called a segment, the behaviour of the **sync** memory statement is implicitly contained in the other synchronization statements. The **sync** memory statement assures that all modifications initiated inside a segment on image A are seen inside succeeding segments on other images. More precisely, if other segments shall see the modifications done in a segment on image A, they have to wait for the segment on image A to end before they can start. This ordering can either be enforced manually by the programmer by using semaphores, mutexes or spin waiting loops, for instance, or by just using the **sync all** or **sync images** statement.

Instructions may be reordered inside segments as if no other images would exist. This means that it is possible for the compiler to implement the optimization of postponing push communication until the next synchronization statement (see section 2.3.3).

Also, the code example 2.6 is valid because a single image should see a coarray as if it was truly an array.

```
if (this_image() == 1) then
    ar[2] = 3
    ar[2] = ar[2] + 3
end if
```

Listing 2.6: Successive remote write accesses

```
type Array
    double precision, dimension(:), allocatable :: data
end type

type(Array), codimension[*] :: array
```

Listing 2.7: Nonsymmetric work distribution

2.3.2 Nonsymmetric coarrays

If a problem domain is to be distributed among different images but not all subdomains are of the same size, this requires using coarrays of derived types with pointer or allocatable components. Code fragment 2.7 illustrates the technique.

With a reasonably good compiler, access to such "pseudo" arrays can be expected to have the same costs as access to true arrays because, on shared memory architectures, each rank can cache the base pointers and the dimensions of the arrays on other images and, on distributed memory architectures, the memory access has to emulated by the sender anyway.

Nonsymmetric coarrays were not used in the LBM implementation, varying subdomain sizes are currently not supported.

2.3.3 Push vs. Pull Communication

If the brackets indicating the codimension are on the left hand side of an assignment equation, the communication type is called `push` communication, whereas it is called `pull` communication if the brackets are on the right hand side (see code example 2.8).

Push/pull communication can be implemented naturally using the `shmem_put`/`shmem_get`

```
! push communication
array(:)[this_image()+1] = array(:)
! pull communication
array(:) = array(:)[this_image()-1]
```

Listing 2.8: Push and pull communication

functions from the SHMEM library [Ltd 01] or `gasnet_put`/`gasnet_get` from the GASNet communication library [Bona 02]. The Rice and the Open64 compiler use GASNet, GASNet in turn can use different conduits to perform the communication (see chapter 3, figure 3.4).

In case of `push`, the compiler is free to postpone the assignment and overlap the communication with the computation until the next synchronization statement (see section 2.3.1). This requires to perform buffering or to check whether the variable being copied is changed in later places inside the code segment. `Pull` communication does not need to be finished until the first usage the gathered remote data.

2.3.4 Image Order

The draft explicitly specifies that the order of images in multidimensional coarrays is the same as the subscript order value of array elements, which is column-major. This means that the image order is exactly the opposite of what it is like in the MPI implementations used in this thesis. As depicted in figure 2.2, this results in different communication characteristics for MPI compared to CAF and has to be corrected for to not confound performance comparisons.

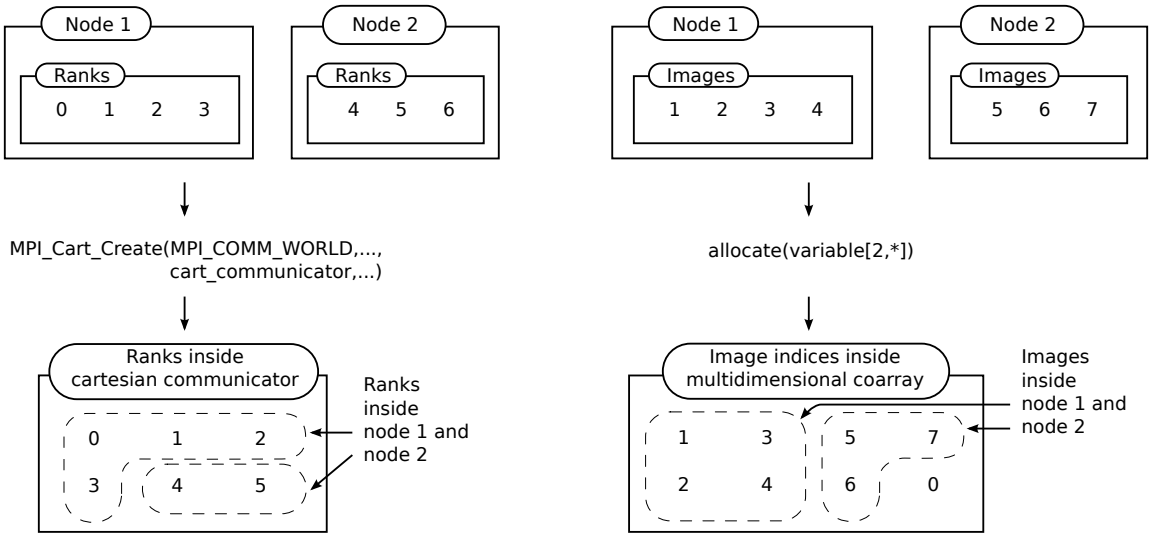
Figure 2.2 also shows the different behaviour of MPI and CAF in cases where not all ranks/images can take place in a multidimensional domain partitioning. This is the case when the number of images (here: 7) is larger than the product of the sizes of the codimensions (here: $2 \cdot 3 = 6$). Whereas MPI will create a communicator in which some of the ranks are not contained (here: rank 6), CAF creates a coarray that is larger than required by the user and assigns all elements where no rank is available to the invalid zero rank (here: array element `variable[2,4]` is assigned to no rank at all and attempts to access it will result in undefined behaviour).

2.3.5 Subarrays

It is possible to pass a subarray of a coarray to subroutines and functions and access remote parts of the subarray inside the procedure. The subarray may not have vector subscripts because this would require copy-in and and copy-out, which would cause problems when other images attempt to modify the coarray while the subroutine has not yet finished. Code example 2.9 shows valid and invalid calls to subroutines with dummy coarray variables.

2.3.6 Reductions

The proposal in [Reid 10] made it possible to write code like this.



(a) MPI rank order

(b) Coarray Fortran image order

Figure 2.2: Rank/image order

```
integer, dimension(7,8,9), codimension[*] :: array

call sub( ar(:,2,2) ) ! ok
call sub( ar(2,:,2) ) ! ok
call sub( ar(2,2,:) ) ! ok

call sub( ar([1,2,3],2,2) ) ! not ok, array subscripts are not allowed

subroutine sub(ar)
    integer, dimension(:), codimension[*] :: array
end subroutine
```

Listing 2.9: Valid and invalid usage of dummy coarray variables

```
double precision, codimension[*] :: array
double precision :: array_sum
array = ....
array_sum = sum(array[:])
```

This means that a coarray can, under special circumstances, be converted into a regular array, thus making it possible to easily code reductions with standard coarray syntax. The feature is, however, not contained in Fortran 2008. Also, in contrast to MPI and OpenMP, no special purpose routines exist for performing reductions concurrently.

```
double precision, dimension(:), codimension[:], allocatable :: a,b,c
double precision :: sum_c
integer :: i, isize

isize = 1+min(((N-1)/isize+1)*(rank+1), N) - ((N-1)/isize+1)*rank
! allocate local parts of the arrays, allocate more space
! than required for the last image if modulo(N, isize) /= 0
allocate(a((N-1)/isize+1)[*])
allocate(b((N-1)/isize+1)[*])
allocate(c((N-1)/isize+1)[*])

call initialize_a_b(a,b)

c(:) = a(:) + b(:)

! make sure all images finished the computation
sync all

! CAF does not provide a built-in reduction operation
! we use an unoptimized global gather
if (this_image() == 1) then
    sum_c = 0.0
    do i=1, num_image()
        sum_c = sum_c + sum(c[i])
    end do
end if
```

Listing 2.10: Example CAF program

2.3.7 Code Example

Listing 2.10 contains a coarray version of the example code that was already shown for MPI and OpenMP.

Chapter 3

Test Machines

3.1 Hardware

The LBM code was run on two different machines, the Cray XE6 of the Swiss National Compute Center [Palu] and the "Lima" cluster of RRZE (Erlangen Regional Computing Center) [Lima]. Table 3.1 summarizes the hardware characteristics of both systems, figure 3.1 visualizes the internal hardware configuration of a compute node of each machine.

When all processors (and virtual processors on the Westmere) were used on a compute node, the process affinity was chosen according to the CPU numbering depicted in figure 3.2. The purpose of this numbering is to minimize the amount of traffic that has to pass the *Hypertransport* and *QuickPath* interconnects respectively. When less processes were used per node, the ordering was again chosen such that the processes were equally distributed among the different NUMA domains and the traffic through the interconnects was minimized.

The memory bandwidths of table 3.1 are taken from the measurements in section 4.1, the network bandwidth is said to be the highest value of the ringshift bandwidth from section 4.2.1. The XE6 has a torus network topology and the total bandwidth available per node does therefore not only depend on the algorithmic memory access patterns but also on the selection of the nodes, see figure 3.3. Resulting from the decision to define the network bandwidth to be the highest bandwidth measured in the ringshift benchmark using two nodes, the network bandwidth given here refers to the bandwidth available per connection of two nodes.

The network interface of the Cray XE6 is provided by the Gemini System Interconnect [Alve 10] and promises to provide good hardware support for coarrays. The Lima cluster, on the other hand, can only provide coarray support by appropriate software emulation.

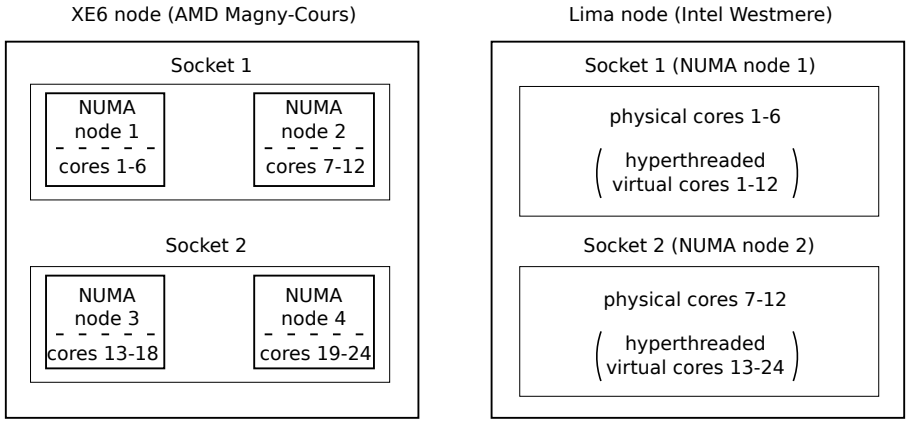


Figure 3.1: NUMA configuration

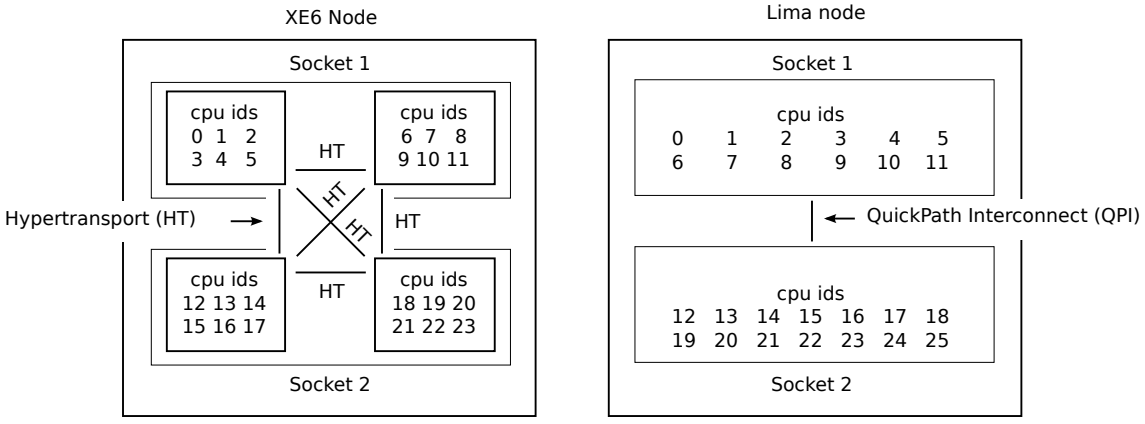


Figure 3.2: Process affinity (system dependent)

	Cray XE6	Lima cluster
Processor	AMD Magny-Cours	Intel Westmere
Clock frequency	2.20 GHz	2.67 GHz
Nominal performance/node (double precision)	211 GFLOP/s	128 GFLOP/s
#Physical cores/node	24	12
#Virtual cores/node	24	24
#Sockets/node	2	2
#NUMA domains/socket	2	1
L3 cache size/NUMA domain	5 MB	12 MB
Measured memory bandwidth/node	50 GB/s	40 GB/s
Network topology	3D torus	Fat tree
Measured network bandwidth/connection	10 GB/s	6 GB/s
#Nodes	176	500

Table 3.1: Compute hardware data sheet

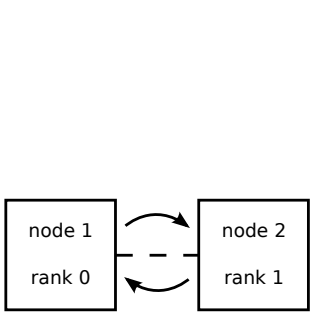
3.2 Software Environment

Table 3.2 contains a "✓" for each compiler/hardware combination that was tested. Table 3.3 lists the software/versions that were used.

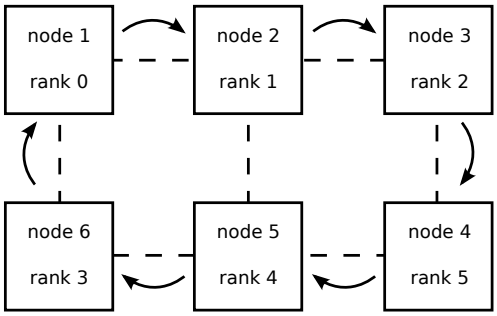
On the Cray XE6, only the Cray Programming Environment was taken into account because it is readily shipped with the hardware and contains a Fortran compiler with coarray support that promises to have good support for the Gemini chipset.

On the Lima cluster, the standard InfiniBand network does not make the choice of the compiler that obvious. While the Intel compiler is only able to use MPI as a conduit, Open64 and the Rice University CAF 2.0 compiler were configured to use the GASNet communication library which in turn was set up to use MPI and the IB Verbs library shipped with the OpenFabrics Enterprise Distribution (OFED) [Open]. Figure 3.4 depicts the dependency trees of Rice and Open64. The attempts to produce working executables failed for the Rice compiler and partially for the Open64, more details are given in section 3.2.4.

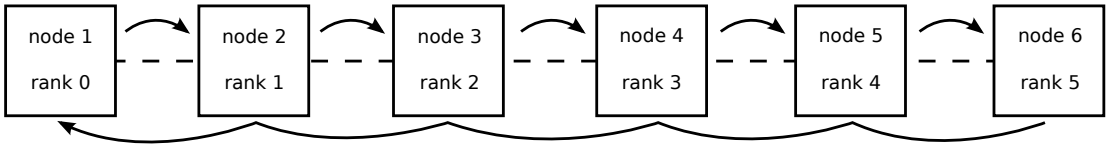
The performance tool collection likwid was installed on both systems [Likw]. Likwid supports program optimization and performance analysis by, amongst others, providing tools for gathering hardware topology information, running processes with a user-defined process affinity mask and reading hardware performance counters. In particular, likwid-topology was used to get information about the node topology and the cache sizes reported in figure 3.1 and table 3.1.



(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 3.3: Ringshift on torus network

3.2.1 Cray Fortran

The Cray Fortran compiler, which was already pre-installed on the Cray XE6 machine, supports coarrays and is able to efficiently utilize the Gemini System Interconnect [Alve 10] [Baw 99] of the XE6. Sufficient documentation is provided and coarray performance is comparable to MPI. The only bug found during the work on the thesis, a bug related to memory allocation, is already fixed in newer versions of the compiler. As for MPI programs, executing a coarray program is done with 'aprun'.

For coarrays larger than a few megabytes, an environment variable has to be set to specify the size per image required for coarrays. The choice of the value has an influence on the performance, but this effect is not investigated in the thesis.

When coarray support is enabled during compilation and large non-coarray variables are to be allocated, it is necessary to instruct the system to use huge memory pages or the allocation will fail. Huge pages can, however, have a negative impact on the performance.

3.2.2 Intel Fortran

Intel's Fortran Compiler is able to compile coarray code since version 12.0. It is as easy to install as the previous versions of the compiler but little documentation is available for the coarray

	Cray Fortran	Intel Fortran	Rice CAF 2.0	Open64
Used on XE6	✓			
Used on Lima		✓	✓	✓

Table 3.2: Hardware/compiler combinations

Software	Version
Cray Programming Environment	Version 4.0.36
Cray Fortran	Version 7.4.4
Intel Fortran	Version 12.0 update 4
Rice University CAF 2.0	Revision 2789 from http://svn.rice.edu/r/caf/caf-compiler
Open64	An extension based on version 4.2
GASNet	Version 1.16.1

Table 3.3: Software version numbers

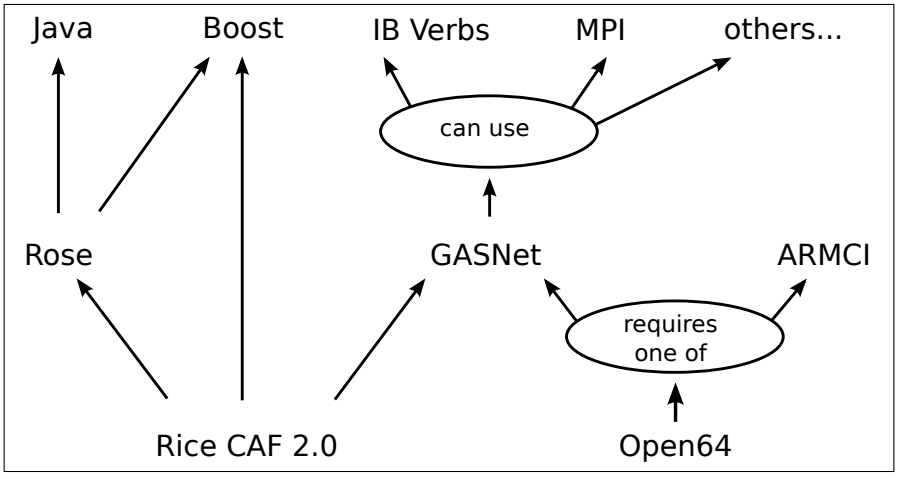


Figure 3.4: Rice CAF 2.0 compiler and Open64 dependency tree

```
if (this_image() == 1) then
    sync images( 2 )
end if
if (this_image() == 2) then
    sync images( 1 )
end if
```

Listing 3.1: Code that works

```
integer :: one
one = 1
if (this_image() == 1) then
    sync images( 2 )
end if
if (this_image() == 2) then
    sync images( one )
end if
```

Listing 3.2: Code that dead-locks

support. The performance is so poor that the compiler must be considered a proof of concept. Measurements indicate that each array element is transferred separately during coarray assignments, even elements that are stored contiguously in memory. The stability has improved a lot from version 12.0 to 12.0 update 4, which was the latest version available during the making of the thesis. In version 12.0 update 3, for instance, it was not yet possible to pass a part of a coarray to a function and modify remote elements inside that function. Working around this issue required extensive code changes in the Lattice Boltzmann code, but the bug is fixed with the compiler update 4. Another important issue has, however, not yet been fixed: `sync images(1)` behaves different to `sync images(one)` with `one` being an integer variable and `one == 1`. The code examples 3.1 and 3.2 describe the compiler bug.

Also, in the LBM, a global gather operation in the time measurement code equivalent to code

```
integer :: array(100)[*]
integer :: i
array = ...
if (this_image() == 1) then
    do i=1,num_images()
        array = array + array(:)[i]
    end do
end if
```

Listing 3.3: Slow gather operation

example 3.3 slowed down the program in such a way that the coarray bandwidth dropped to a few bytes per second and let this part of the code run for minutes. Also, the ping-pong benchmark crashed for messages larger than 500MB.

To execute distributed coarray programs, an MPI configuration file has to be specified, the writing of which is very tedious and eventually made it necessary to write a configuration file generator.

3.2.3 Rice

The Rice University provides a work in progress Coarray Fortran source-to-source compiler for a language syntax that is different from the coarrays introduced in the Fortran 2008 language standard. The language is called Coarray Fortran 2.0 and is, according to [Mell 09], superior to the syntax introduced by Fortran 2008. As can be seen in figure 3.4, the installation requires resolving many dependencies. The documentation is still sparse and the compiler is not yet very stable. The underlying network conduit is provided by GASNet [Bona 02].

No benchmarking was performed because it does, with the Rice compiler in its current state, seem very hard to compile a coarray program unless one has information about which features have already been implemented, and which have not. For instance, "`while`" loops are not supported and source code errors typically cause failing assertions inside the compiler without information about the line number and the kind of error. Static coarrays can only be implemented in modules, not in program units. There are further bugs in program units, and it seems more safe to declare all coarray variables inside modules. Filenames containing non-lowercase characters result in a failing assertion and all files containing coarrays must have the file extension ".caf". Usage of interfaces also causes a failing assertion, which is extremely problematic because the time measurement that is used inside the benchmarks is implemented in the C programming language and its binding to coarray Fortran was done via an interface. In the Fortran statement `read(unit=rdbuf, iostat=rdstat, fmt=*)`, the part `fmt=*` vanishes during the source-to-source translation. Currently, only one co-dimension is supported which is problematic for multidimensional domain partitioning. Also, using a preprocessor does not work out of the box.

3.2.4 Open64

Compiling Open64 requires resolving either the dependency on GASNet or on ARMCI. As the ARMCI website was down during the writing of the thesis and as GASNet is required for Rice anyway, GASNet was chosen. There does currently not exist any documentation about the coar-

ray features of Open64 because the coarray support is not yet publicly available. In contrast to Rice, Open64 is a real compiler, not a source-to-source compiler and it is Fortran 2008 compliant. It seems to be much more stable than the Rice compiler. As for the Rice compiler, coarray programs are executed with the tools provided by GASNet.

Open64 and GASNet were compiled in two versions, one with Intel MPI support and one with native InfiniBand support provided by IB Verbs [Open]. The executables produced with MPI support worked, if not more than one process ran per compute node. With more than one executable running on one node, segmentation faults occurred in **`sync all`** statements. The InfiniBand executables crashed with a segmentation fault inside a GASNet call during the program startup. One benchmark from section 4.2.6 failed with a segmentation fault because of a bug in the intrinsic `reshape` function.

Benchmarking was performed with the MPI version for inter node communication only. During the start of executables using the MPI conduit, a warning message informs about the lack of speed of the MPI implementation of GASNet and proposes to recompile GASNet with InfiniBand support. As already mentioned, the InfiniBand version crashed during program startup.

3.3 Pinning configuration

All program and benchmark runs were performed with a pinning configuration that maximizes the available memory bandwidth. This is achieved by placing the same amount of processes on each NUMA domain of a processor. On the XE6 with a Magny Cours processor, 4 NUMA domains are available per node, one Lima node contains 2 NUMA domains (see figure 3.1).

Chapter 4

Low-level Benchmarks

The purpose of this chapter is to provide reasonable input data for the performance model [Hage 10] in section 5.7, which requires knowledge about the bandwidth of the memory system and the latency and bandwidth during communication.

4.1 Memory Bandwidth

4.1.1 Benchmark Implementation

The benchmark codes used for the memory bandwidth measurements are shown in listings 4.1 and 4.2. Suitable timing code was omitted. Table 4.1 summarizes the results. The code examples use the variable `n` to specify the size of the **`double precision`** arrays. The following sections will, however, use the variable $N = n \cdot 8$ Bytes, which stores the array size in bytes. As the benchmark results are to be used in the performance model of the Lattice Boltzmann algorithm and as the benchmark runs used for the Lattice Boltzmann algorithm are designed such that the problem domain does not fit into the cache, the array size `n` in the benchmarks is also chosen such that the arrays do not fit into the processor cache.

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

Listing 4.1: 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 4.2: Copy benchmark with $2 \cdot 19$ memory streams

$\frac{\text{\#streams}}{\text{process}}$	$\frac{\text{\#processes}}{\text{node}}$	$\frac{\text{XE6, bandwidth}}{\text{node [GB/s]}}$	$\frac{\text{Lima, bandwidth}}{\text{node [GB/s]}}$
2, non temporal	2	19.9	19.8
2	2	18.5	29.6
$2 \cdot 19$	2	8.4	16.1
2, non temporal	12	50.5	40.4
2	12	51.9	40.1
$2 \cdot 19$	12	39.3	38.3
2, non temporal	24	49.8	40.8
2	24	54.1	41.1
$2 \cdot 19$	24	51.9	38.9

Table 4.1: Memory bandwidth of copy benchmarks 4.1 and 4.2

4.1.2 Non Temporal Stores

On modern cache based architectures, writing to the variable `a` (see listings 4.1 and 4.2) does first require loading `a` from main memory to the cache ("write allocate") before it can be modified and evicted to main memory later on. Loading `a` into the cache is not necessary in the considered copy benchmarks because no subsequent read operations on `a` are performed while it is in the cache. Optimally, loading `a` should therefore be omitted by using "non temporal stores". Non temporal stores are assembler instructions that write to a memory location and bypass the cache.

The benchmark using two memory streams was implemented with and without non temporal stores, the benchmark with $2 \cdot 19$ memory streams was only implemented without non temporal stores. The LBM implementation that is used throughout the thesis does not use non temporal stores. When the working set of the LBM method does not fit into the cache and a sufficiently large number of processes runs on each NUMA domain, the performance is memory bound and the performance characteristics are resembled by the copy benchmark using $2 \cdot 19$ streams.

4.1.3 Results

The bandwidth was calculated using the formula $B = \frac{2 \cdot N}{\text{runtime}}$ for two streams when non temporal stores were used, and according to $B = \frac{3 \cdot N}{\text{runtime}}$ and $B = \frac{3 \cdot 19 \cdot N}{\text{runtime}}$ otherwise. It can be seen that the bandwidths for $2 \cdot 19$ memory streams do only approach the bandwidths for two streams for sufficiently large numbers of processes per NUMA domain. Analysing the assembly code revealed that the loop with 2 memory streams was vectorized, while the code with $2 \cdot 19$ memory streams was not.

4.2 Communication

As explained in section 5.3, the Lattice Boltzmann Method does, like other stencil codes, require the exchange of boundary slices of multidimensional arrays between processes. The communication time expected for the transfer of those slices is predicted by the performance of the ringshift and the strided ping-pong benchmark. The ringshift benchmark is used to predict the communication time for stride 1 communication, while the strided ping-pong benchmark is used to check how the communication bandwidth is affected by communication types other than stride 1. In the LBM algorithm, strided communication does not occur for all domain decomposition techniques and can therefore be avoided (at the cost of a higher communication volume though). This section contains the benchmarking results for those two low-level benchmarks and also the results for a ping-pong benchmark with stride 1. The correctness of the results obtained for the ping-pong and ringshift benchmark was validated by comparing them to the results of the Intel MPI Benchmarks [Inte].

The benchmarks worked on either `integer` or `double precision` arrays. If not specified otherwise, the performance graphs show results for benchmarks that used `integer` arrays.

The bandwidths were measured either inter node oder intra node. In case of inter node measurements, there were always two nodes taking part in the communication.

4.2.1 Benchmark Implementation

Listings 4.3, 4.4, 4.5 and 4.6 show the benchmark codes used for the ping-pong and ringshift measurements. The code actually used for the measurements is more complicated than shown here. More precisely, the real code has a loop around the lines of code that perform the network communication. It is ensured that the loop terminates only if more than one second has passed since loop entry. Also, when intra node communication is taking place, the inner loop does not

only operate on a buffer of size `n`, but on a buffer that is guaranteed to be larger than the L3 cache. Chunks of size `n` within this buffer are transferred, ensuring that no cache reuse can occur between successive loop iterations. The strategy of using a large buffer and sending chunks was not used for inter node transfers because this resulted in strong bandwidth drops for medium sized messages. We were not able to explain those bandwidth drops.

MPI Implementation

To simplify the code snippets, the `MPI_COMM_WORLD`, `ierr` and `istatus` function parameters were removed (see listings 4.3 and 4.4)

```
integer, dimension(n) :: buffer

rank = ...
buffer = rank
call MPI_Barrier()

if (rank == 0) then
    call MPI_Send(array, n, MPI_INTEGER, 1, 29)
    call MPI_Recv(array, n, MPI_INTEGER, 1, 31)
end if

if (rank == 1) then
    call MPI_Recv(array, n, MPI_INTEGER, 0, 29)
    call MPI_Send(array, n, MPI_INTEGER, 0, 31)
end if
```

Listing 4.3: MPI ping-pong

```
integer, dimension(n) :: send_buffer, recv_buffer

rank = ...
destRank = ...
destRank = ...

buffer = rank
call MPI_Barrier()

call MPI_SendRecv(send_buffer, n, MPI_INTEGER, destRank, 1, &&
                  recv_buffer, n, MPI_INTEGER, srcRank, 1)
```

Listing 4.4: MPI ringshift

CAF Implementation

The code examples 4.5 and 4.6 illustrate the CAF implementations of the benchmarks. A push implementation was used in both cases.

```
integer, dimension(n), codimension[*] :: buffer

buffer = this_image()
sync all

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

Listing 4.5: CAF ping-pong using push strategy

```
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 4.6: CAF ringshift using push strategy

4.2.2 Bandwidth Model

The bandwidth model for the ping-pong benchmark reads as follows. Let

N be the message size,

T the time spent for the completion of the benchmark,

L the latency and

B be the bandwidth, then

$$T = 2 \cdot \left(L + \frac{N}{B} \right)$$

For the ringshift, the following model is used.

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

The estimation of L and B is shown for the ping-pong benchmark. The values for the ringshift were estimated in the same way. In a first attempt, the following minimization problem was

solved. Let

N_i be the message size in the i-th benchmark run,

T_i the time spent for the completion of the i-th benchmark run,

L the latency and

B the bandwidth, then

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

This problem formulation did, however, result in poor results for the latency because of low absolute errors for small message sizes. To increase the weighting of small message sizes, the following minimization problem, which minimizes the squared sums of the relative residuals, was solved instead.

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

4.2.3 Non-strided Communication using 4 Byte Array Elements

The Cray CAF implementation is about as fast as MPI (figure 4.1). Inter node CAF has a higher bandwidth, but the latency is worse than for MPI, which results in higher transfer times for small message sizes.

Figure 4.2 shows that the communication bandwidth for the ifort coarray implementation is too slow to be suitable for most applications. Open64 with GASNet and MPI (figure 4.3) already performs reasonably well and there is hope that a native InfiniBand version will be faster.

In figures 4.1 and 4.2, the term inter/intra socket refers to inter/intra NUMA domain communication. The ping-pong benchmark was run with two different intra node process pinnings, one involving only communication inside a NUMA node and one involving only communication between NUMA domains. Communication is typically faster inside NUMA domains. However, the results only have limited relevance for practical applications because the ping-pong benchmark only used two processes, whereas real world applications would use many more processes, which might result in different ping-pong curves.

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 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.

4.2.4 Push vs. Pull Communication

One might expect pull communication to be slower than push communication because more network traffic to set up the network communication might be required. On the other hand, push communication might be implemented in a non-blocking way by using buffering, thus decreasing the bandwidth. The measurements of the ping-pong benchmark using pull communication show that the performance characteristics of push and pull do not differ on the XE6 (figure 4.1) but on the Lima with Intel CAF and Open64 (figures 4.3 and 4.2). The XE6 shows this behaviour because all communication is performed instantaneously (see section 4.2.8), no further investigations were made to find the reasons for the differences with Intel CAF and Open64.

4.2.5 4 Byte vs. 8 Byte Array Elements

Figure 4.4 compares the ringshift benchmark with double precision (64 Bit) buffers to the ringshift benchmark with integer (32 Bit) buffers. Only the Cray compiler performs the same for both, **`integer`** and **`double precision`** arrays. Open64 and the Intel compiler are faster with **`double precision`** arrays. The Intel compiler even doubles in bandwidth when **`double precision`** arrays are used, which suggests that element-wise communication is taking place.

4.2.6 Strided Ping-pong

Figure 4.5 shows strided ping-pong communication results for two code variants: `pingStrd` refers to native, strided coarray communication, while `pingStrb` refers to strided coarray communication with previous manual buffering.

Strided inter node communication is poorly supported on the XE6 and manual buffering is required if strided accesses cannot be circumvented. Strided intra node communication does not need to be buffered.

For parallelized stencil codes this means that the computational domain should not be decomposed along the fast axis unless manual buffering is implemented. This effect can also be seen in the LBM benchmarking results in section 6.2.

The performance of the Intel CAF implementation is not affected by the choice of the stride. This result can be expected if element-wise communication is used underneath.

The inter node performance of Open64 shows a strong penalty for strides greater than one. Due to a bug in the reshape function of Open64, no results are available for the buffered and strided ping-pong benchmark.

4.2.7 Explicitly Element-wise Communication

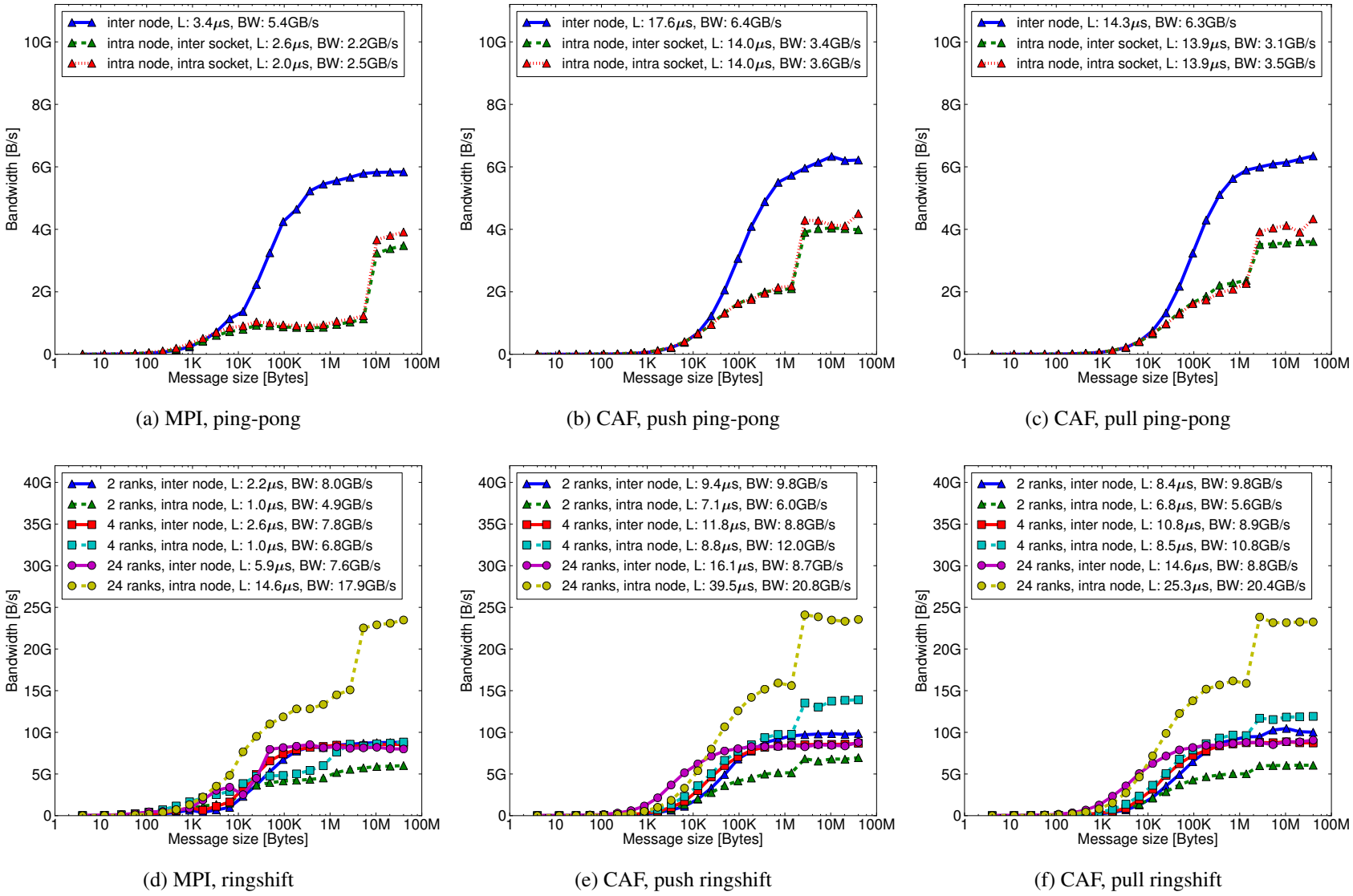
Figure 4.6 compares contiguous and element-wise ping-pong communication for the different compilers. The ping-pong benchmark was modified such that the contiguous copy statements like `dst(:)[2] = src(:)[1]` were converted into loops that perform element-wise copy operations. No synchronization statements were inserted into the inner copy loop. The inner loop was obfuscated to make it (nearly) impossible for the compiler to convert the loop into one contiguous copy statement during 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 schedules and merges data transfers. As measurements for all the compilers showed bad performance on this task, neither of the statements seems to be true for any of the tested compilers/hardware.

The XE6 inter node communication bandwidth drops by a factor of 3000, the intra node bandwidth drops by a factor of 30.

As expected the Intel compiler bandwidth does not change, because contiguous communication is not supported.

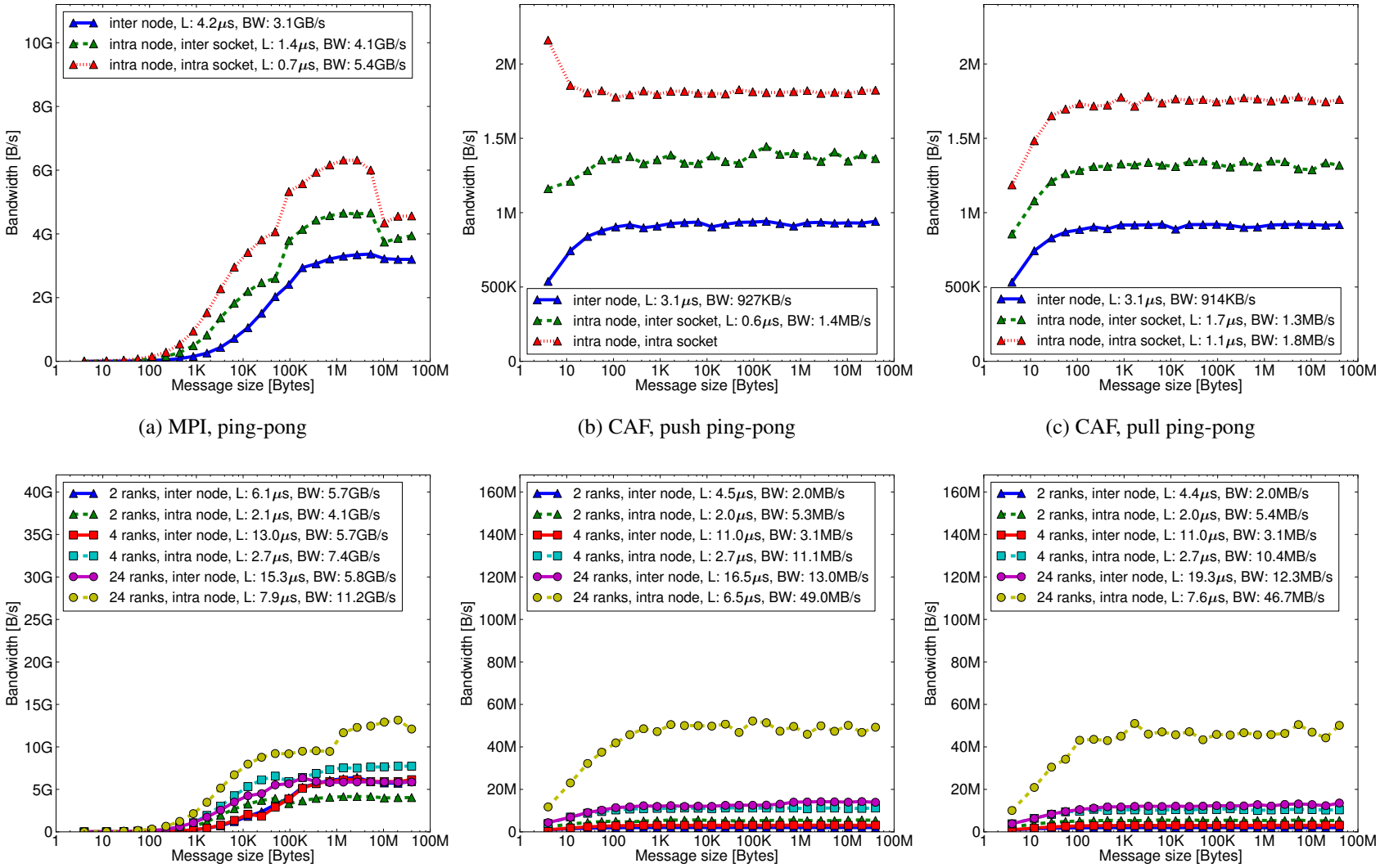
The Open64 benchmark exited with the error message "`GASNet Extended API: Ran out of explicit handles (limit=65535)`" for messages larger than 200 KB. The peak bandwidth was obtained for message sizes of about 1 KB with a bandwidth that was a factor 500 lower than the peak bandwidth of the contiguous communication.



(a) MPI, ping-pong (b) CAF, push ping-pong (c) CAF, pull ping-pong

(d) MPI, ringshift (e) CAF, push ringshift (f) CAF, pull ringshift

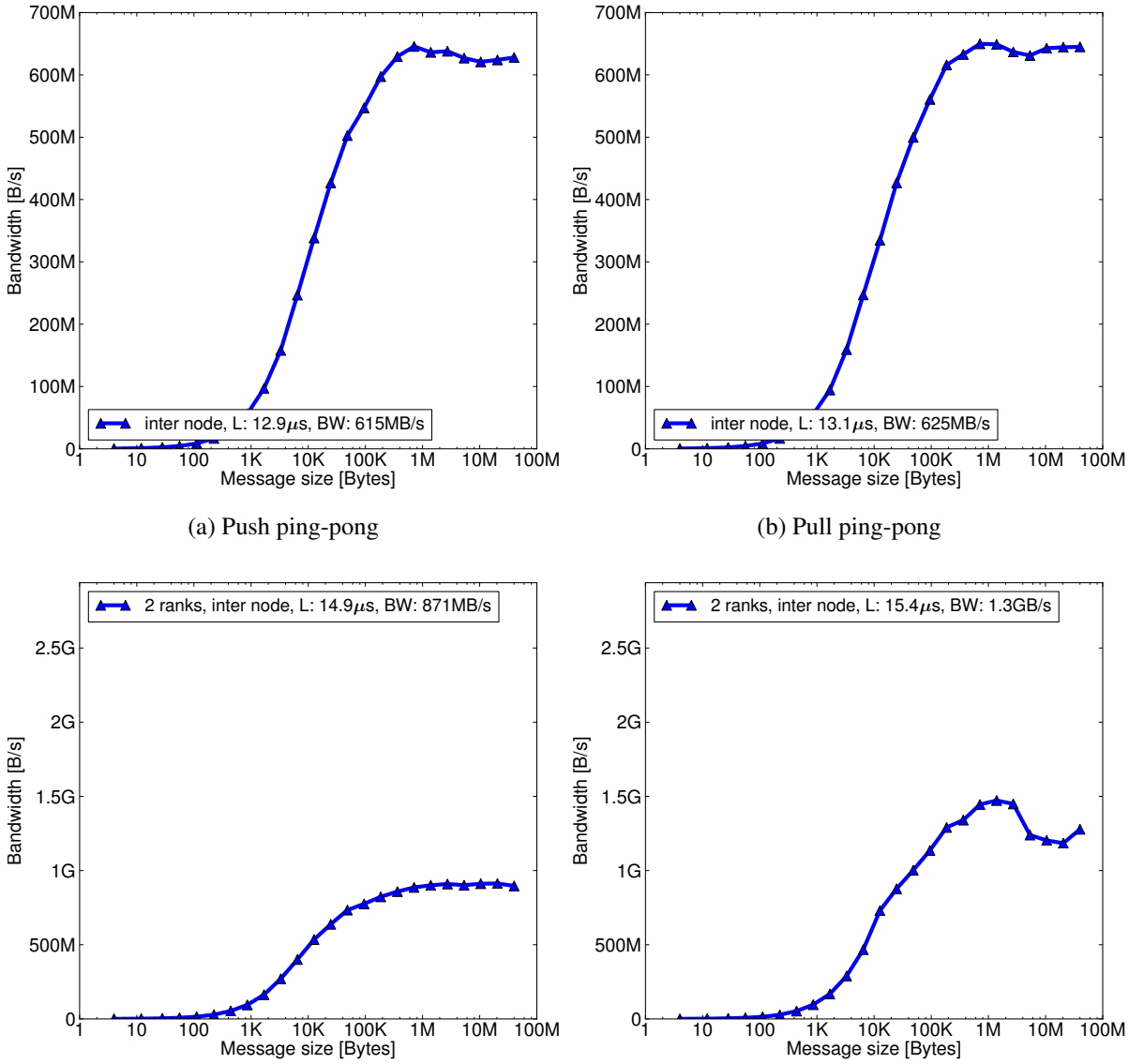
Figure 4.1: Ping-pong and ringshift with Cray Fortran on the Cray XE6



(a) MPI, ping-pong (b) CAF, push ping-pong (c) CAF, pull ping-pong

(d) MPI, ringshift (e) CAF, push ringshift (f) CAF, pull ringshift

Figure 4.2: Ping-pong and ringshift with Intel Fortran on Lima



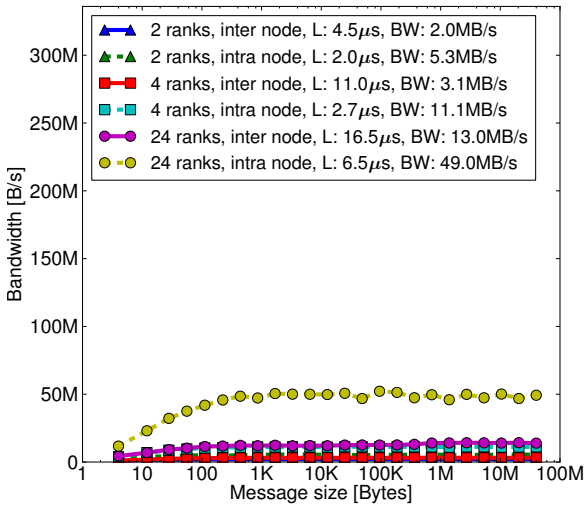
(a) Push ping-pong

(b) Pull ping-pong

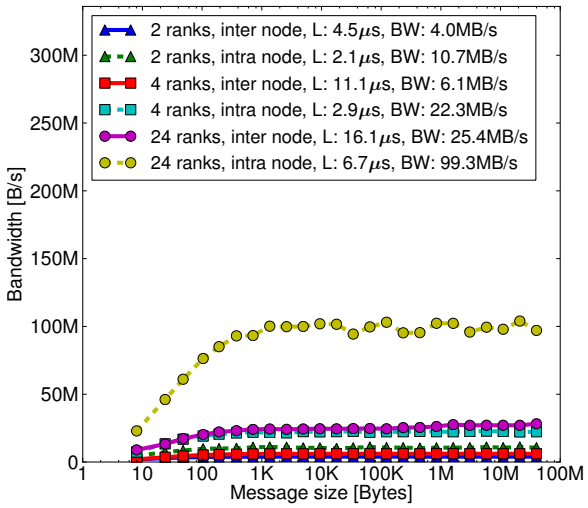
(c) Push ringshift

(d) Pull ringshift

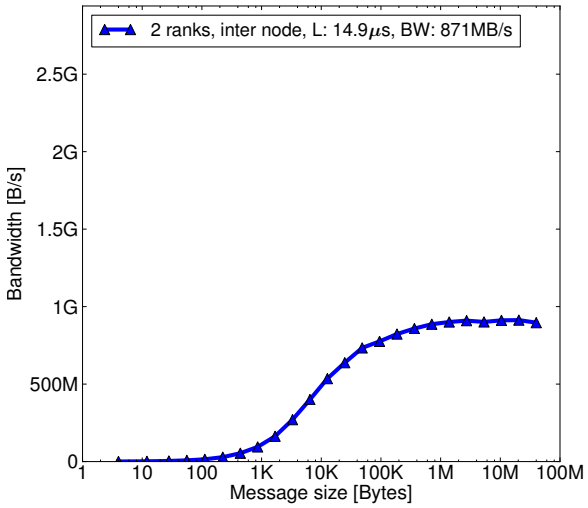
Figure 4.3: Ping-pong and ringshift with Open64 on Lima



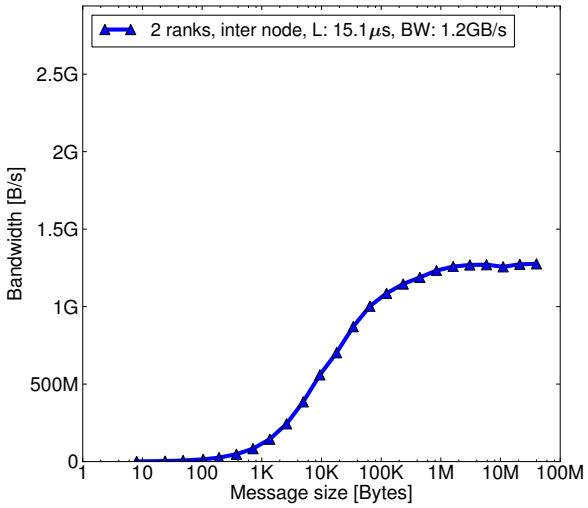
(a) Lima, ifort, integer



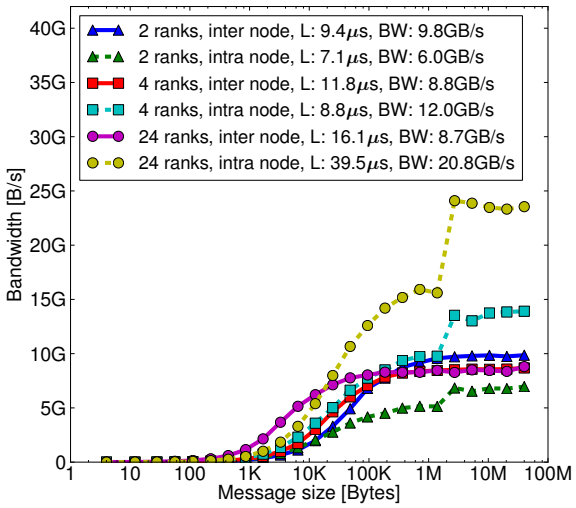
(b) Lima, ifort, double precision



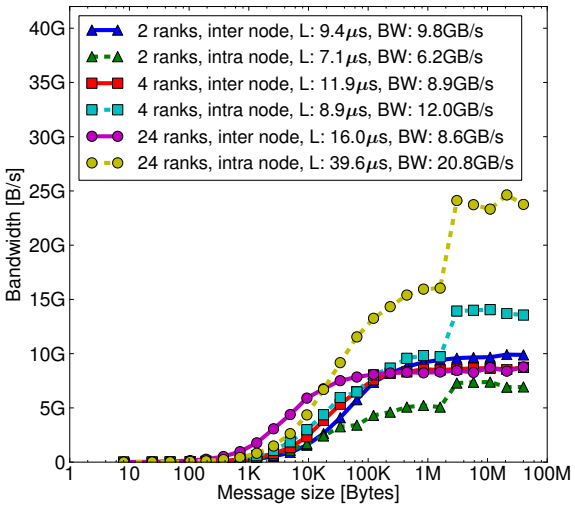
(c) Lima, open64, integer



(d) Lima, open64, double precision

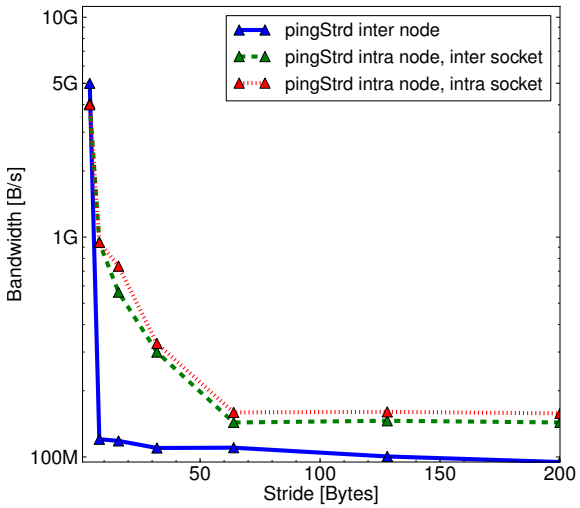


(e) XE6, crayftn, integer

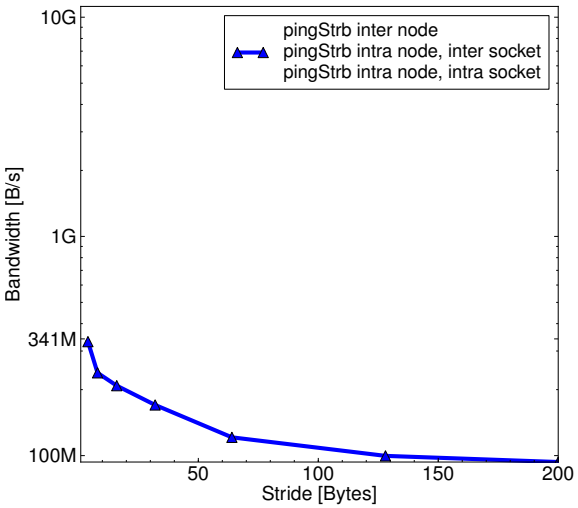


(f) XE6, crayftn, double precision

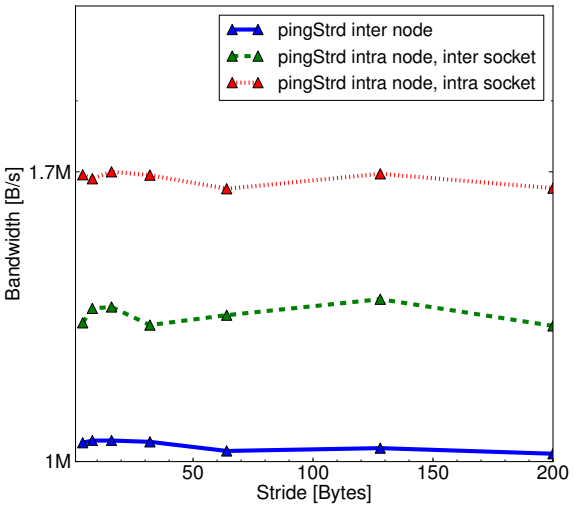
Figure 4.4: CAF ringshift with integer buffer vs. double precision buffer



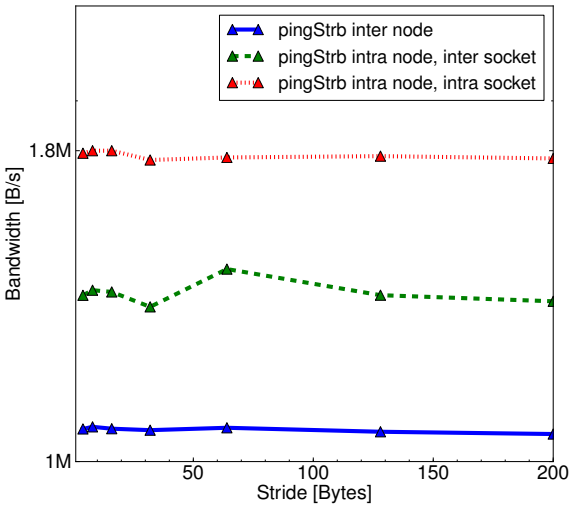
(a) XE6, unbuffered



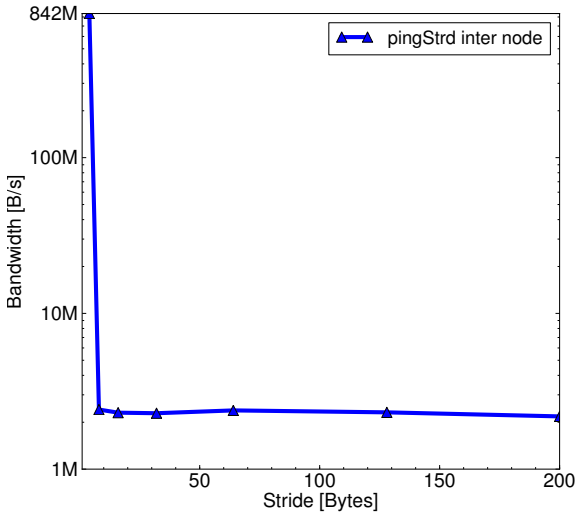
(b) XE6, buffered



(c) Lima, ifort, unbuffered

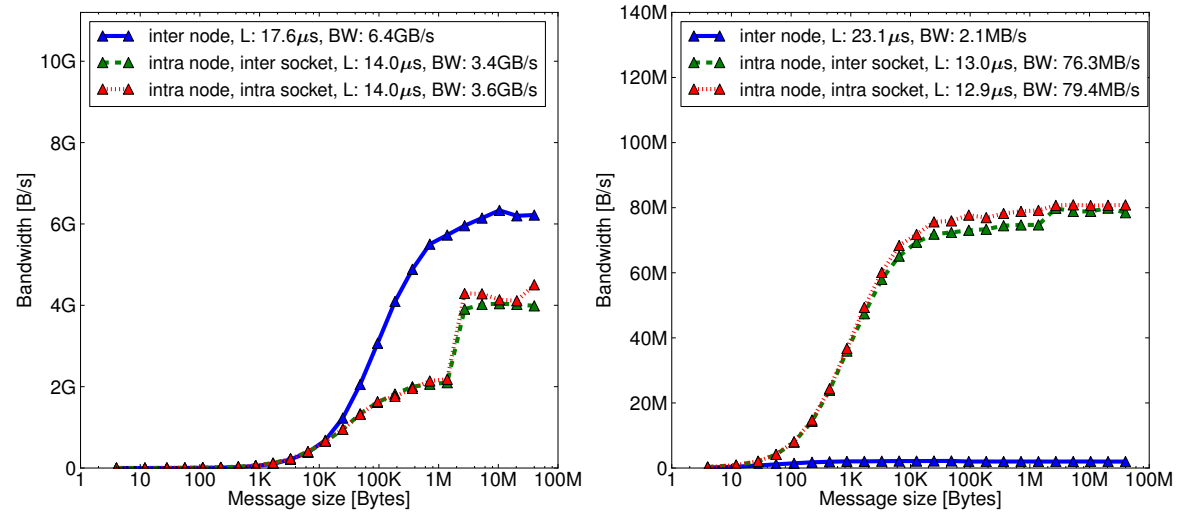


(d) Lima, ifort, buffered



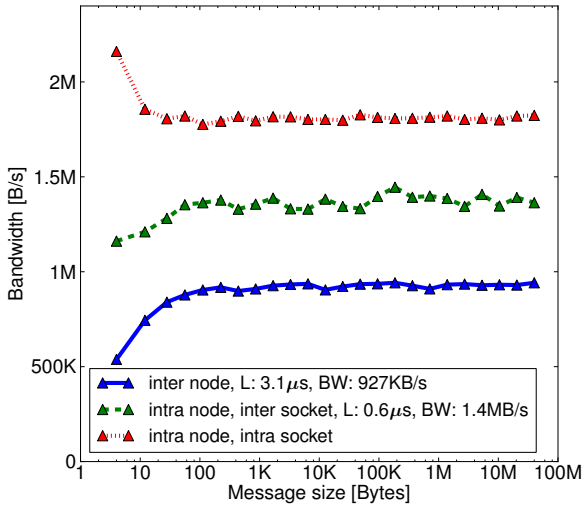
(e) Lima, open64, unbuffered

Figure 4.5: Strided ping-pong, CAF, message size of 8.5 MB

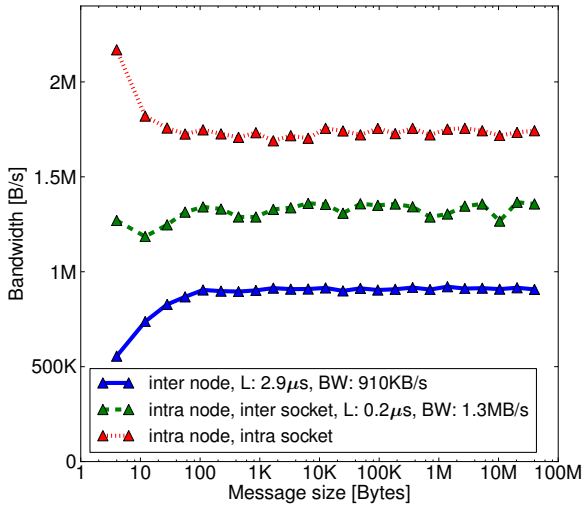


(a) XE6, contiguous

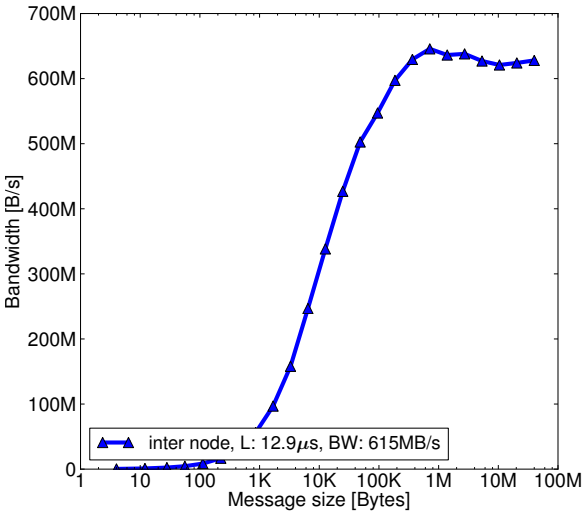
(b) XE6, element-wise



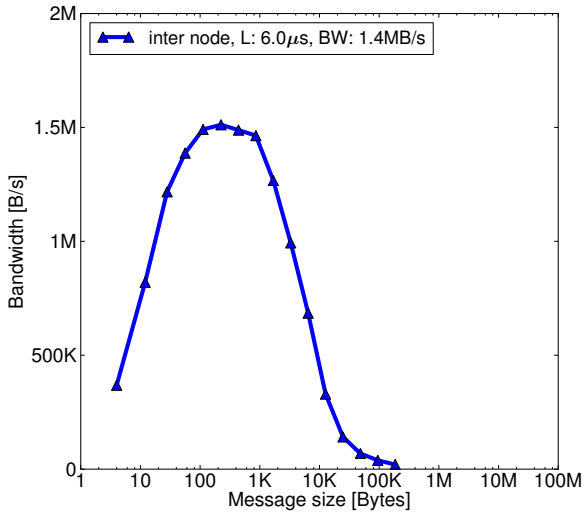
(c) Lima, ifort, contiguous



(d) Lima, ifort, element-wise



(e) Lima, open64, contiguous



(f) Lima, open64, element-wise

Figure 4.6: Explicitly contiguous vs. element-wise ping-pong communication

Push/pull	Communication enabled	Computation enabled	Inter/Intra Node	Communication Bandwidth [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 4.2: Test for overlap of communication and computation on the XE6

4.2.8 Overlap of Computation and Communication

To check whether it is possible to overlap computation and communication with the available compilers, the ringshift benchmark was modified as shown in listings 4.7 and 4.8.

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

Listing 4.7: Push ringshift with overlapping computation

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

Listing 4.8: Pull ringshift with overlapping computation

The benchmark was run for all possible combinations of `comm` and `comp`, except for `comm = comp = false` on all compilers. As the computational kernel does not require any memory bandwidth, the process placement does not influence the results if `comm = false`, thus the "don't care" in the corresponding cells.

A message size of 4 MB was chosen, the measurement results are as listed in table 4.2. It shows that communication and computation do neither overlap for push nor for pull communication, no matter if the processes are on the same node or not.

```
# create an executable file "program_name.exe+pat"
# from the original program "program_name"
pat_build -f -u -g caf program_name
# run the instrumented program,
# this creates a trace file program_name.exe+pat*.xf
aprun -n 2 -N 2 -S 1 ./program_name.exe+pat
# visualize the trace file
pat_report -T program_name.exe+pat*.xf
```

Listing 4.9: CrayPat command line

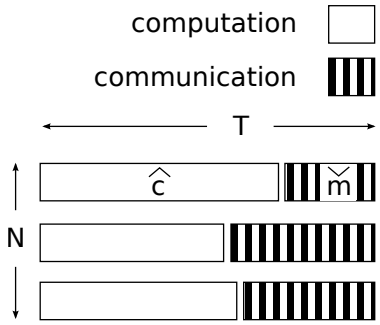


Figure 4.7: CrayPat performance model

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 the other compilers showed exactly the same behaviour.

4.2.9 CrayPat

The performance measurement tool CrayPat is available on the XE6 system and was applied to a strided/non-strided, inter/intra node ringshift using push/pull communication with two coarray images. It is important to note that the coarray was not allocatable. 500 data transmissions were performed. The -g compiler switch must not be enabled if the Cray compiler shall not use element wise communication. The CrayPat command line and the results look as depicted in listings 4.9 and 4.10.

The number of calls has obviously been divided by the number of images by CrayPat. The columns `Imb. Time` and `Imb. Time%` provide information about the imbalance time and imbalance% [DeRo]. Each of the metrics asks a different question. For imbalance time it is “what is the upper bound for the program runtime that could be saved if the load balance was perfect?”. For imbalance% it is “what fraction of the workers could be saved if the load balance was per-

```
 Time%  |    Time  |    Imb.  |  Imb.  |  Calls  |Group
        |          |    Time  | Time%  |         | Function
        |          |          |        |         |  PE=HIDE
        |          |          |        |         |   Thread=HIDE

 100.0% | 0.115166 |       -- |     -- | 27831.0 |Total
|-------------------------------------------------------------------------
|  92.9% | 0.106970 |       -- |     -- | 27729.0 |PGAS
||------------------------------------------------------------------------
||  64.4% | 0.074125 | 0.000969 |   2.6% |   500.0 |__pgas_sync_nb
||  14.5% | 0.016669 | 0.000726 |   8.3% | 17229.0 |__pgas_poll
||   3.3% | 0.003801 | 0.003149 |  90.6% |  1000.0 |__pgas_barrier_wait
||   2.3% | 0.002664 | 0.000203 |  14.2% |  1000.0 |__pgas_aor
||   2.2% | 0.002539 | 0.000172 |  12.7% |  1000.0 |__pgas_aadd
||   1.9% | 0.002222 | 0.001252 |  72.1% |  1000.0 |__pgas_barrier_notify
||   1.4% | 0.001654 | 0.000747 |  62.2% |  2000.0 |__pgas_fence
||   1.0% | 0.001170 | 0.001170 | 100.0% |   500.0 |__pgas_aand
||   0.9% | 0.001081 | 0.000000 |   0.0% |   500.0 |__pgas_put_nb
||   0.5% | 0.000633 | 0.000007 |   2.3% |  1000.0 |__pgas_sync_all
||   0.2% | 0.000194 | 0.000005 |   5.5% |   500.0 |__pgas_memput_nb
||   0.1% | 0.000157 | 0.000004 |   4.8% |   500.0 |__pgas_link_handle_to_syncid
||   0.0% | 0.000044 | 0.000006 |  24.7% |   500.0 |__pgas_sync_nb_adaptive
||   0.0% | 0.000017 | 0.000004 |  37.5% |   500.0 |__pgas_reserve_handle
||=========================================================================
```

Listing 4.10: CrayPat results for non-strided, inter node, push communication with a static coarray

fect and the program should run for the same time as the original program with imperfect load balance?". The number of workers is defined as the total number of processes minus one.

CrayPat's performance model assumes that the program consists of computation and communication calls, and that a global barrier exists at the end of the communication calls (see figure 4.7). Under these assumptions the imbalance time can be calculated as $imb = \hat{c} - \bar{c}$ for computation calls and, with $T = \bar{c} + \bar{m} = \hat{c} + \check{m}$, as $imb = \bar{m} - \check{m}$ for communication calls where $\hat{c}$ is the maximum computation runtime, $\check{m}$ is the minimum communication time and $\bar{c}$ and $\bar{m}$ denote the mean times.

Regarding calculation of imbalance%, if N_0 denotes the number processes, N_1 denotes the number of processes required in case of a perfect load balance and $\hat{c_0}$, $\hat{c_1}$ and all other variables are defined accordingly, the imbalance% can be calculated as follows.

Because the total amount of computation has to remain constant,

$$\bar{c}_0 \cdot N_0 = \bar{c}_1 \cdot N_1$$

The total runtime has to remain constant, so

$$\bar{c}_1 = \hat{c}_0$$

$$\Rightarrow imb\% = \frac{N_0 - N_1}{N_0 - 1} = ... = \frac{N_0}{N_0 - 1} \cdot \frac{imb}{\hat{c_0}}$$

When calculating $imb\%$ for communication calls, $\hat{m_0}$ is used instead of $\hat{c_0}$ in the formula above. This makes an interpretation of the $imb\%$ value for communication calls problematic. The benchmarked code consists solely of communication calls. As the model also assumes that all asynchrony comes from computation calls, the interpretation of the imb values that appeared in the analysed benchmark is problematic es well.

Attempts were made to

- deduce the type of communication (strided/non-strided, inter/intra node, push/pull) from the name of the function that requires the most communication time in table 4.10 or from some other function that only shows up for a specific kind of communication,
- find performance differences between the different function calls,
- check whether some of the communication types overlap communication and computation, while others do not.

All function calls were found to show the same performance characteristics and none of the function calls showed overlap of communication and computation. Looking at the summary table 4.3 we came to the conclusion that the neither the strided-ness nor whether inter or intra node communication is taking place can be deduced from the CrayPat results for application codes, because application code will typically use allocatable coarrays.

allocatable	strided	inter/ intra	push/ pull	specific function	topmost function	runtime contribution of topmost function
n	n	inter	push	__pgas_put_nb	__pgas_sync_nb	64.4%
n	n	inter	pull	__pgas_get_nb	__pgas_sync_nb	61.7%
n	n	intra	push	__pgas_put_nb	__pgas_put_nb	70.9%
n	n	intra	pull	__pgas_get_nb	__pgas_get_nb	78.6%
n	y	inter	push	__pgas_put_strided	__pgas_put_strided	98.1%
n	y	inter	pull	__pgas_get_strided	__pgas_get_strided	98.8%
n	y	intra	push	__pgas_put_strided	__pgas_put_strided	93.8%
n	y	intra	pull	__pgas_get_strided	__pgas_get_strided	97.5%
y	n	inter	push	__pgas_put_strided	__pgas_put_strided	63.7%
y	n	inter	pull	__pgas_get_strided	__pgas_get_strided	68.6%
y	n	intra	push	__pgas_put_strided	__pgas_put_strided	47.9%
y	n	intra	pull	__pgas_get_strided	__pgas_get_strided	62.9%
y	y	inter	push	__pgas_put_strided	__pgas_put_strided	98.0%
y	y	inter	pull	__pgas_get_strided	__pgas_get_strided	98.9%
y	y	intra	push	__pgas_put_strided	__pgas_put_strided	88.0%
y	y	intra	pull	__pgas_get_strided	__pgas_get_strided	95.9%

Table 4.3: CrayPat results for different benchmark configurations

Chapter 5

Lattice Boltzmann Algorithm

5.1 Lattice Boltzmann Theory

The Lattice Boltzmann Method (LBM) [Succ 01] is an explicit time stepping scheme for the numerical simulation of fluids. The fluid is modelled as conglomerates of particles which collide in each time step and move to adjacent cells afterwards. The viscosity of the fluid and the fact that fluid particles move according to their velocity are modelled in two separate steps, the *collide* and the *stream* step (see figures 5.1a and 5.1b). The particle conglomerates, also known as *particle distribution functions* (PDFs) f_l, are only stored for the centres of lattice cells and can only have discrete velocities. If a particle is located in the centre of a lattice cell and has such a discrete velocity, it will move exactly into the centre of one adjacent cell in one time step. This lets the stream step become a simple memory copy operation.

The code used for the thesis assumes the fluid to be incompressible, the physical quantities time step size and density are normalized to $\delta t = \rho = 1$. All other quantities are normalized as well and gaining the true physical quantities back does therefore require denormalization.

1. Collide step

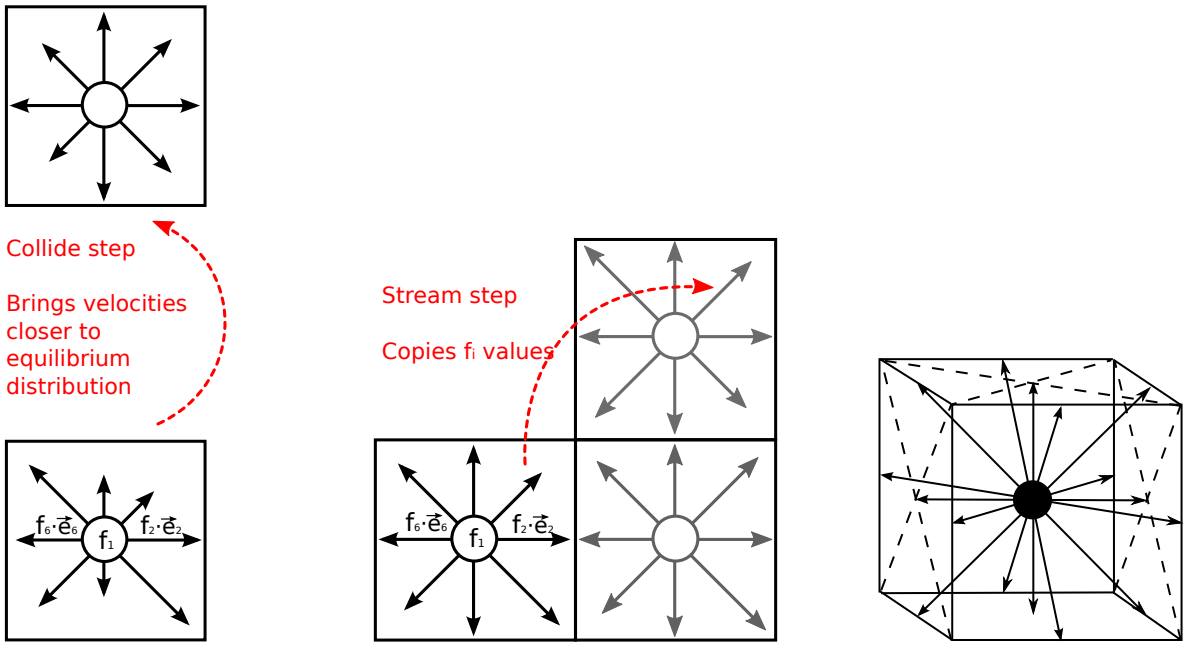
$$\tilde{f}_l(\boldsymbol{x}, t+1) = f_l(\boldsymbol{x}) + \frac{1}{\tau}(f_l^{eq} - f_l(\boldsymbol{x}, t))$$

2. Stream step

$$f_l(\boldsymbol{x} + \boldsymbol{e}_l, t+1) = \tilde{f}_l(\boldsymbol{x}, t+1)$$

With the second order Taylor expansion of the Maxwell equilibrium distribution function

$$f_l^{eq} = \omega_l \rho \left(1 + \frac{3\boldsymbol{e}_l \boldsymbol{u}}{c^2} + \frac{9(\boldsymbol{e}_l \boldsymbol{u})^2}{2c^4} - \frac{3(\boldsymbol{u})^2}{2c^2}\right)$$



(a) Collide step (b) Stream step (c) D3Q19 Model

Figure 5.1: LBM Algorithm

and

$$\boldsymbol{u} = \sum_{l} f_l \cdot \boldsymbol{e}_l$$

For a 3D code, $\boldsymbol{x}$ is the cell coordinate in the 3D arrays f_l and $\tilde{f}_l$ (t is ignored, no time history is stored), $\boldsymbol{e}_l$ are discrete lattice velocities and are specified, for example, by the D3Q19 model visualized in figure 5.1c. The D3Q19 model does also specify the parameters ω_l and c, with $c = 1$. The D3Q19 model gets its name from the fact that it operates in 3 dimensions and incorporates 19 particle distribution functions. There exist other models, differing in the dimensionality and in the number of distribution functions.

Simple boundary conditions can be implemented easily, no-slip boundary conditions, for instance, are implemented by reflecting all PDFs penetrating obstacle cells.

5.2 Compute Kernel

The algorithm used for the thesis fuses the collide and the stream step into one combined collide-stream step, in this order, meaning it performs the collision first and propagates the particles thereafter. The propagation optimized memory layout [Well 06] is used, which means that each

```
do k=1,Nz
    do j=1,Ny
        do i=1,Nx
            do l=1,19
                load source_pdf(i,j,k,l)
            end do
            perform computations using pdf(i,j,k,:)
            do l=1,19
                store destination_pdf(i+e1(l),j+e2(l),k+e3(l),l)
            end do
        end do
    end do
end do
```

Listing 5.1: Simplified LBM compute kernel

particle distribution function is stored in a separate array and the array index denoting the index of the velocity direction is the slowest index. Two grids are stored, one storing the PDFs of the previous time step and one storing the PDFs of the current time step. A simplified version of the inner loop of the algorithm reads as listing 5.1.

When the algorithm is memory-bound, which is the case for all measurements in the thesis, its performance characteristics can be approximated by the stream benchmark using 19 read and 19 write streams, the results of which can be found in section 4.1.

5.3 Parallelization

The parallelization is done by a Cartesian, equidistant domain decomposition and exchange of ghost layers between the different parts of the domain. Each Cartesian cell corresponds to one MPI rank (or coarray image) and has 18 neighbouring cells in the D3Q19 model. It is, however, sufficient to communicate with six neighbours if the communication is scheduled as shown in code example 5.2, where `exchange_ghost_cells(East, West)` means that the ghost layers with the neighbours in eastern and western direction are exchanged. East, North and Top denote the ranks of the neighbours in direction of the three Cartesian basis vectors that define the grid of lattice cells. The indirect ghost layer exchange for the diagonal neighbours is visualized in figure 5.2b.

It should be noted that the ghost cell transmission is performed before the boundary condition handling. Figure 5.2a shows, for a 1D example with one obstacle cell at the interface between to process subdomains, why it is necessary to order the function calls in this way. One stream-communicate-boundary handling cycle of the algorithm is shown. The memory layouts

```
collide_stream()

exchange_ghost_cells(East, West)
exchange_ghost_cells(North, South)
exchange_ghost_cells(Top, Bottom)

handle_boundary_conditions()
```

Listing 5.2: Ghost layer exchange in pseudo code

before/after the algorithmic steps are divided by horizontal lines. Inside the regions divided by horizontal lines, the upper lattice cells belong to process 1, the lower cells belong to process 2. The interface between the two processes is visualized by the vertical line. The lattice cell of process 1 which is on the right of the vertical line is a ghost cell of process 1, the cell of process 2 which is left of the boundary is a ghost cell of process 2. Each arrow corresponds to a particle conglomerate. The crucial point is that only complete boundary slices (slices without holes) are transmitted. This means that, after the "communicate" step, the second cell of process 1 is filled with bogus values from process 2. Placing the boundary treatment after the "communicate" step overwrites those bogus values.

Resulting from the structure of the D3Q19 model, each process has to send and receive 5 distribution functions during every ghost layer exchange. The ringshift in section 4.2.1 is used to model the performance characteristics.

The following code examples use the convention that, if a process does not have a neighbour in one direction, this neighbour rank is set to the invalid value "`-1`".

5.4 MPI Implementation

The MPI implementation uses non-blocking send/receive pairs (see listing 5.3). 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.

As access to main memory is fastest when the accesses have stride 1, there exist slow and fast axes for traversing an array (see figure 5.2c). The traversal for the buffering 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.

For simplicity the code shown 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.

```
subroutine exchange_ghost_cells(rankA, rankB)
    call mpi_irecv(receiveBufferA, receiveBufferSizeA, rankA, receiveRequestA)
    call mpi_irecv(receiveBufferB, receiveBufferSizeB, rankB, receiveRequestB)

    if (rankA .ne. -1) call copy_cells(directionA, sendBufferA)
    if (rankA .ne. -1) call mpi_isend(sendBufferA, sendBufferSizeA, rankA, &
&
                                      sendRequestA)

    if (rankB .ne. -1) call copy_cells(directionB, sendBufferB)
    if (rankB .ne. -1) call mpi_isend(sendBufferB, sendBufferSizeB, rankB, &
&
                                      sendRequestB)

    ! missing code: wait for receiveRequestA and receiveRequestB

    if (rankA .ne. -1) paste_cells(receiveBufferA)
    if (rankB .ne. -1) paste_cells(receiveBufferB)

    ! missing code: wait for sendRequestA and sendRequestB
end subroutine
```

Listing 5.3: Simplified version of the MPI communication code

The actual application code does not have this restriction.

5.5 MPI-like CAF Implementation

The MPI-like, buffered CAF version was created by converting the receive buffers of the MPI implementation into coarrays. The send/receive code is shown in listing 5.4.

The line `receiveBufferA[rankB]=sendBufferB` is discussed in more detail. By definition (compare to the original MPI code in listing 5.3), `receiveBufferA` contains the ghost cells received from `rankA` (`rank=images_index()-1`). Suppose that there exist only two processes and that rank 0 is eastern of rank 1. Then, rank 0 calls `exchange_ghost_cells(rankA=-1, rankB=1)` and rank 1 calls `exchange_ghost_cells(rankA=0, rankB=-1)`. This means that rank 1 expects the data from rank 0 in `receiveBufferA` and rank 0 has to execute `receiveBufferA&[rankB]=sendBufferB`.

Like the MPI implementation the code shown here expects the ghost cells in x, y and z direction to be of equal size to improve readability. The actual application code does not have this restriction.

```
subroutine exchange_ghost_cells(rankA, rankB)
    if (rankA .ne. -1) call copy_cells(directionA, sendBufferA)
    if (rankB .ne. -1) call copy_cells(directionB, sendBufferB)

    if ((rankA .ne. -1) .and. (rankB .ne. -1)) sync images([rankA, rankB]+1)
    if ((rankA .ne. -1) .and. (rankB .eq. -1)) sync images([rankA       ]+1)
    if ((rankA .eq. -1) .and. (rankB .ne. -1)) sync images([       rankB]+1)

    if (rankA .ne. -1) receiveBufferB[rankA] = sendBufferA
    if (rankB .ne. -1) receiveBufferA[rankB] = sendBufferB

    if ((rankA .ne. -1) .and. (rankB .ne. -1)) sync images([rankA, rankB]+1)
    if ((rankA .ne. -1) .and. (rankB .eq. -1)) sync images([rankA       ]+1)
    if ((rankA .eq. -1) .and. (rankB .ne. -1)) sync images([       rankB]+1)

    if (rankA .ne. -1) call paste_cells(directionA, receiveBufferA)
    if (rankB .ne. -1) call paste_cells(directionB, receiveBufferB)
end subroutine
```

Listing 5.4: Simplified, buffered CAF code

5.6 CAF Implementation

The main part of the unbuffered CAF implementation is hidden in the subroutine `transmit_slice` (listing 5.5). 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: `rankB` expected the data to be in `receiveBufferA`. In 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 `pasteStart` and `pasteEnd`).

As already explained in section 5.4 and shown in figure 5.2c, the performance of the buffering before the halo exchange depends on the dimension along which the halo cuts through the 3D array. Such an effect can also be seen for unbuffered coarray accesses.

5.7 Performance Model

For the considerations already shown for the ringshift in figure 3.3 and for the LBM method in figures 5.2e and 5.2f, 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 to the fully non-blocking fat tree network of the Lima.

Figure 5.2d is used to estimate the number of neighbours of a node. This means that the following assumptions are made.

```
subroutine exchange_ghost_cells(rankA, rankB)
    if ((rankA .ne. -1) .and. (rankB .ne. -1)) sync images([rankA, rankB]+1)
    if ((rankA .ne. -1) .and. (rankB .eq. -1)) sync images([rankA       ]+1)
    if ((rankA .eq. -1) .and. (rankB .ne. -1)) sync images([       rankB]+1)

    if (rankA .ne. -1) call transmit_slice(directionA, rankA)
    if (rankB .ne. -1) call transmit_slice(directionB, rankB)

    if ((rankA .ne. -1) .and. (rankB .ne. -1)) sync images([rankA, rankB]+1)
    if ((rankA .ne. -1) .and. (rankB .eq. -1)) sync images([rankA       ]+1)
    if ((rankA .eq. -1) .and. (rankB .ne. -1)) sync images([       rankB]+1)
end subroutine

subroutine transmit_slice(direction, destRank)
    integer :: copy_Start(3), copy_End(3), pasteStart(3), pasteEnd(3)
    integer :: links(5)

    ! missing code: from direction, compute copy_Start, copy_End,
    !   pasteStart, pasteEnd and links

    pdf(pasteStart(1):pasteEnd(1),
        pasteStart(2):pasteEnd(2),
        pasteStart(3):pasteEnd(3), links)[destRank] = &
    pdf(copy_Start(1):copy_End(1),
        copy_Start(2):copy_End(2),
        copy_Start(3):copy_End(3), links)
end subroutine
```

Listing 5.5: Unbuffered CAF communication

- 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 5.2d).

- All other subdomains share exactly two faces with other process subdomains inside that node (ranks 1,...,22 in figure 5.2d).

This means that the union of the subdomains contained by all processes inside a node is topologically connected and "looks like a snake". This will not always be the case in practice because the "snake" might get cut at the array boundary if the first communication axis is not divisible by the number of processes inside a node. An underestimation of the communication time will be the result.

Secondly it means that the "snake does not touch itself". This assumption might not hold if the first communication axis contains less processes than are contained inside a node.

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 5.2d). Also, inside each node, $2 \cdot (P-1)$ intra node communications take place. The following additional assumptions are made.

- The performance of the LBM kernel is memory bound.

- 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 with a size of 8 Bytes 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 latency,
measured with the ringshift benchmark in section 4.2.1,

$B_{e/a}$ be the inter/intra node bandwidth,
measured with the ringshift benchmark in section 4.2.1,

M be the memory bandwidth,
measured with the copy benchmark with 19 streams in section 4.1 and

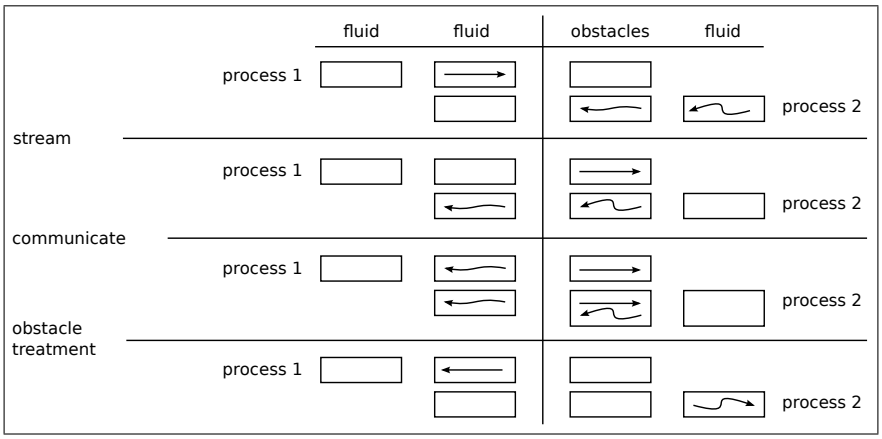
N be the number of lattice cells in each dimension of the subdomain stored by a process.

Then, the time required for one time step is

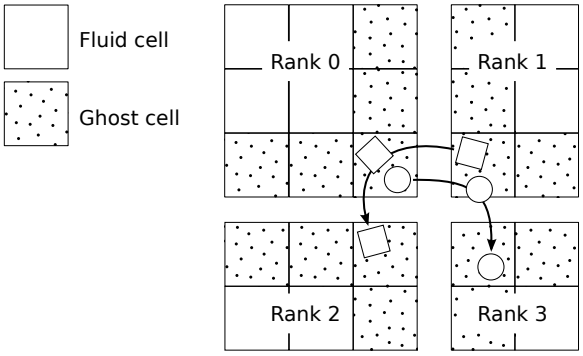
$$t = 3 \cdot L_e + \frac{5 \cdot 8\,\text{Bytes} \cdot N^2}{(B_e/2)/(4 \cdot p + 2)} + 3 \cdot L_a + \frac{5 \cdot 8\,\text{Bytes} \cdot N^2}{(B_a/2)/(2 \cdot (p-1))} + \frac{3 \cdot 19 \cdot 8\,\text{Bytes} \cdot N^3}{M}$$

The number of lattice site updates per second is

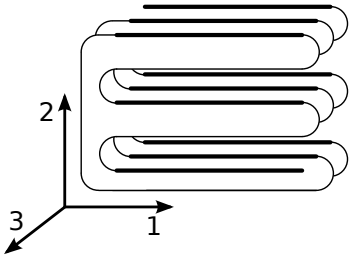
$$LUPS/s = P \cdot \frac{N^3}{t}$$



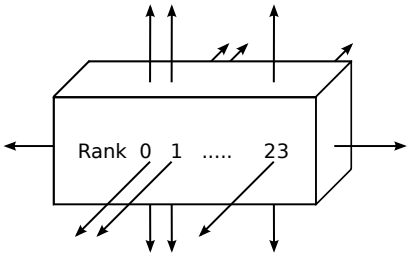
(a) Communication order



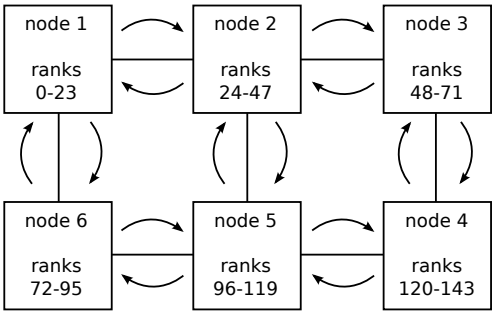
(b) Indirect ghost layer exchange for diagonal neighbours



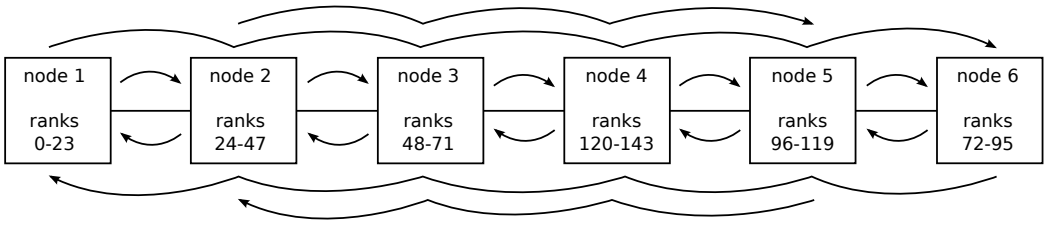
(c) 3D array, memory layout



(d) Algorithmical neighbours of one compute node



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



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

Figure 5.2: Parallel LBM implementation

Chapter 6

Performance Evaluation of the LBM Implementations

The analysis of the performance of the LBM implementations starts with determining the number of processes required per node to saturate the memory system and the optimal node filling factor is then used in the subsequent strong scaling and weak scaling runs. CAF and MPI is benchmarked, 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 5.7.

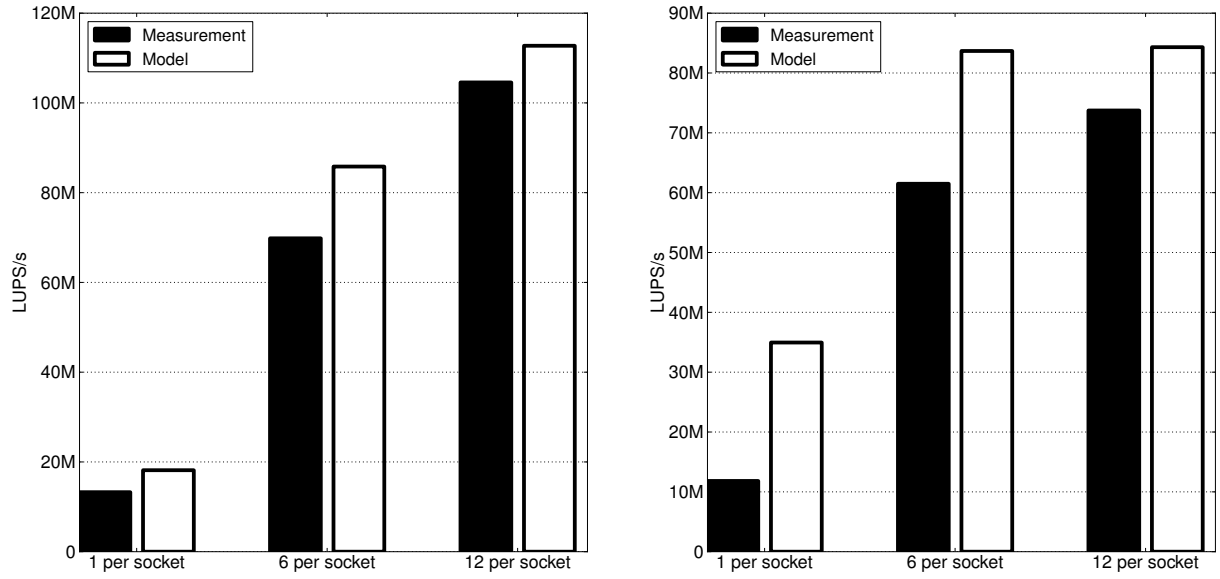
6.1 Optimal Single Node Performance Evaluation

To find out how many processes are required per node to saturate the memory system, figure 6.1 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 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 to saturate the memory system.

```
pat_build -f -u -g caf lbm
export PAT_RT_HWPC=15
aprun -n 1 -N 1 -S 1 ./lbm+pat
pat_report -T program_name.exe+pat*.xf
```

Listing 6.1: Command line for CrayPat LBM with L3 cache counters



(a) XE6

(b) Lima

Figure 6.1: 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).

To check the validity of the stream benchmark as a performance model for the LBM, CrayPat was used to monitor hardware performance counters giving information about the number of memory transfers. The program was executed as illustrated in listing 6.1, where the line `export PAT_RT_HWPC=15` enables the hardware performance group 15, which is the L3 socket level. Only a single time iteration was performed and only one process was run on a domain of size $110 \times 22 \times 110$.

As shown in listing 6.2, CrayPat measured $1\,275\,378$ L3 cache misses. With a cache line size of 64 Bytes one would expect $\frac{2 \cdot 8 \cdot 19 \cdot 110 \cdot 100 \cdot 22}{64} = 1\,264\,450$ evictions, which is in good agreement with the measurements.

```
USER / relax_
------------------------------------------------------------------------
  Time%                                            21.2%
  Time                                          0.033748 secs
  Calls                          29.6 /sec           1.0 calls
  L3_EVICTIONS:ALL              0.004M/sec           145 ops
  READ_REQUEST_TO_L3_CACHE:ALL  44.056M/sec       1487032 req
  L3_CACHE_MISSES:ALL           37.786M/sec       1275378 misses
  L3_FILLS_CAUSED_BY_L2_EVICTIONS:
    ALL                         37.335M/sec       1260163 fills
  User time (approx)             0.034 secs      70881502 cycles  100.0%Time
  Average Time per Call                           0.033748 secs
  CrayPat Overhead : Time          0.0%
  L3 cache hit,miss ratio         14.2% hits        85.8% misses
```

Listing 6.2: L3 cache counters measured with CrayPat

6.2 Strong scaling

The following 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,1,1) in the legend corresponds to a 1D domain decomposition along the first (and fast) axis of the Fortran array, (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. This means that the algorithm's memory footprint is larger than the cache size up to about 40 nodes in the strong scaling computations.

As it is not possible to use more than three compute nodes for a 1D domain decomposition of the chosen computational domain with the given number of processes per node, it is hard to compare the 1D domain decomposition to the 2D and 3D decomposition approaches and on a per-node basis. The 1D decomposition is therefore compared separately in an extra plot on a per-process basis.

Figures 6.2 and 6.3 show strong scaling runs up to 63 nodes. In order to increase the sampling rate, the domain size is increased in each dimension up to the next number divisible by the number of ranks in that dimension. If the resulting computational volume is more than 10% larger than the original computational volume, the benchmark is not run.

6.2.1 2D/3D Domain Decomposition

XE6

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 CAF implementation shows nearly the same performance characteristics as the MPI version and both implementations are equally fast up to about 30 nodes. The CAF performance is slightly worse than MPI for larger amounts of nodes. 30 nodes corresponds to a message size of $\frac{5 \cdot 8\text{Bytes} \cdot 350^2}{\sqrt{24 \cdot 30}} \approx 180\,\text{kB}$ for the buffered CAF implementation, which is, according to results of the ringshift benchmark from section 4.2.1, 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. This could already be expected from the strided pingpong measurements in section 4.2.6. The (1,x,x) decomposition is about as fast as for the buffered CAF version.

Lima

In contrast to the MPI implementation on the XE6, the Lima MPI implementation performs best if the (1,x,x) domain decomposition is used.

Due to the low CAF communication bandwidth (see section 4.2.1), the CAF parallelization is communication 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 4.2.6), 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 perdicts a performance that is about twice as large as the true performance of the LBM. This effect can also be seen in the weak scaling results of section 6.3. 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.

6.2.2 1D Domain Decomposition

XE6

On the XE6, the performance penalties for the 1D domain decomposition are equal for MPI, CAF and buffered CAF (figure 6.3). This leads to the conclusion that the domain is so large that the communication does hardly affect the runtime. The differences in the performance are therefore due to different performances of the computational kernel for the different decomposition directions. It is not possible to use many more processes than shown here for the 1D domain decomposition because the slices for 60 ranks already have a thickness of 6 lattices.

Lima

On the Lima with Intel CAF the algorithm performance is limited by communication due to the low bandwidth of the Intel coarray implementation. As the communication volume per node does not shrink for increasing numbers of processes, no speedup can be seen.

6.3 Weak Scaling

A weak scaling run using a 2D domain decomposition was benchmarked on Lima with MPI and CAF. Each process operates on a domain of size $96 \times 96 \times 96$, which corresponds to a memory requirement of 6.5 GB per full compute node. The placement of the MPI ranks for 56 nodes is shown in figure 6.4a, the topology of the runs with fewer nodes can be constructed from the placement of figure 6.4a by just removing all nodes with higher numbers. The results are shown in figures 6.4c to 6.4f. No results for the XE6 are shown, because its torus network topology makes the resulting performance graphs hard to interpret.

6.3.1 Assumptions

The following assumptions are made in the analysis of the results.

- The computation of one process can overlap with the communcation of its horizontal and vertical neighbours.
- If a process has to wait for communication of its neighbours to start, and another process inside that node can already start its LBM kernel, this LBM kernel run will require less time than in the case where all processes run synchronously.

So, to oversimplify: If the processes run asynchronously in the horizontal direction, the LBM kernel will require less runtime and the communication will take longer than in the synchronous case.

This statement is problematic because running intra node communication also requires memory bandwidth, only the waiting times make resources available.

- Adding more processes in vertical direction will, except for the step from 2 to 3 horizontal lines, let the processes run more synchronously in horizontal direction due to a higher coupling in vertical direction.

To clarify the last point, suppose that only one horizontal line of nodes exists in figure 6.4a, consisting of nodes 1 and 2. Further suppose that the processes in node 2 run slower than the processes in node 1. This will result in a decreased overall computation time and an increased communication time.

Now suppose that a second line of nodes is added in vertical direction, with node 3 above node 1 and node 4 above node 2. Again node 2 runs slower than node 1. Node 2 will now additionally slow down node 1 through the connection node 2 $\rightarrow$ node 4 $\rightarrow$ node 3 $\rightarrow$ node 1. Nodes 2 and node 1 will therefore run more synchronously which results in an increased computation time.

Now suppose that the communication time (including waiting times) between two vertical neighbours is a random variable with a variance of σ^2. This variance will add up to the time difference between horizontal neighbours inside the nodes. If a third line of nodes is added in vertical direction, the overall vertical communication time of one process in the vertically central line of nodes will be a random variable of variance $2 \cdot \sigma^2$ under the assumption of statistical independence of the times needed to communicate with the upper and the lower neighbour. The horizontal synchrony of the processes in the vertically central line of nodes will therefore be lower than the synchrony of the processes in the case of only two lines of nodes and the computation time will therefore be lowered.

Adding additional nodes in vertical direction will, like the step from one to two vertical lines, result in a stronger coupling of the different processes and thus slightly increase the horizontal synchrony again.

6.3.2 Application

The following analysis explains the MPI performance graphs.

The base runtime for one node is increased when a second node is added, because inter node communication is now taking place. More time is required for the communication, and as the processes run more asynchronously in horizontal direction now, the kernel runtime decreases.

Starting with three nodes, there exists one node that has two neighbours and its runtime determines the runtime of the total algorithm. As more processes are added there exist more nodes with two neighbours, but the total runtime does hardly change because the slowest node determines the overall runtime.

From 7 to 14 nodes, one line of nodes is added in vertical direction which results in a higher horizontal coupling of the nodes and thus a higher LBM kernel runtime. As more communication is done per node, the communication time increases. This effect is, however, partly compensated by the fact that the higher coupling decreases the communication time a little bit.

From 14 to 21 nodes, one line of nodes is added that requires much more inter node communication time than all other nodes. This results in a great increase of the communication time and, due to the higher communication time, in a higher horizontal asynchrony and thus a decrease of the kernel runtime.

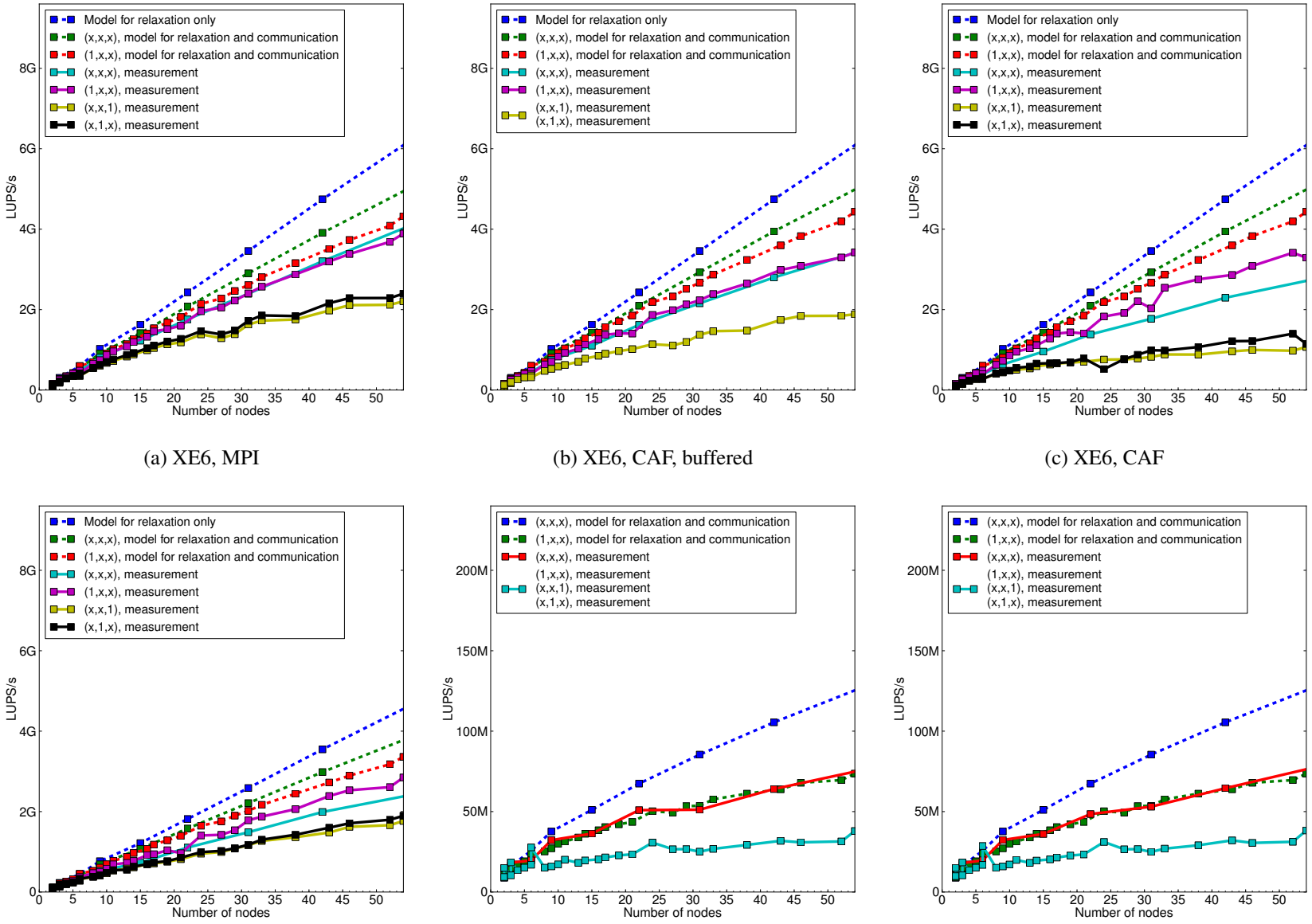
As more nodes are added, the communication time does hardly change for the same reasons that applied for the graph segment between 3 and 7 nodes. However, due to the higher number of vertical nodes, the horizontal synchrony increases and the communication time does therefore slightly decrease while the kernel runtime increases.

Figure 6.4f reveals that the performance model does not work well for Intel CAF communication. First of all, the communication time is strongly underestimated as soon as inter node communication takes place. Secondly, the increase in communication time from 3 to 4 horizontal lines is not explained by the performance model. Due to Intel CAF's poor overall performance no further inverstigations were made to find the reasons for the mismatches between model prediction and measurement results.

6.4 CrayPat PGAS Communication Calls

CrayPat was used to monitor the internals of an LBM run. As already seen in section 4.2.9, it is hard to draw conclusions about the communication characteristics of a program from the monitored PGAS calls. In contrast to the ringshift code benchmarked in section 4.2.9, which used `sync all` statements, the LBM uses `sync images` statements to finalize communication calls. The observations from section 4.2.9 can therefore not be directly applied to the LBM CrayPat measurements.

Runs with different domain partitionings were performed, but the CrayPat outputs looked similar in all runs and no decisive differences could be found. Listing 6.3 shows the output of the CrayPat run with a (1,5,1) domain partitioning, a domain size of $110 \times 110 \times 110$ and 17 time iterations. The program was executed like already explained in section 4.2.9.



(a) XE6, MPI

(b) XE6, CAF, buffered

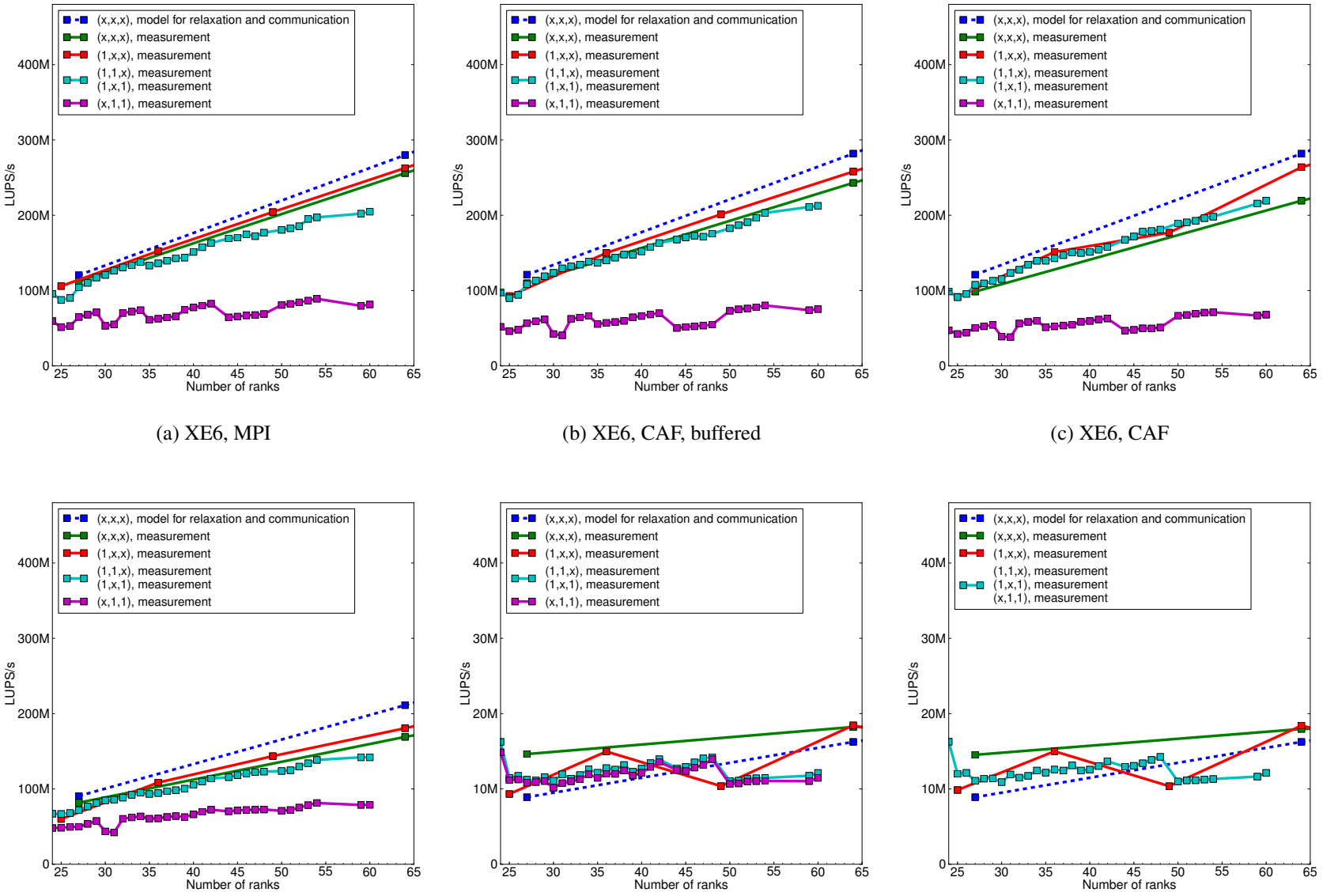
(c) XE6, CAF

(d) Lima, MPI

(e) Lima, CAF, buffered

(f) Lima, CAF

Figure 6.2: Effects of 2D/3D domain decomposition



(a) XE6, MPI

(b) XE6, CAF, buffered

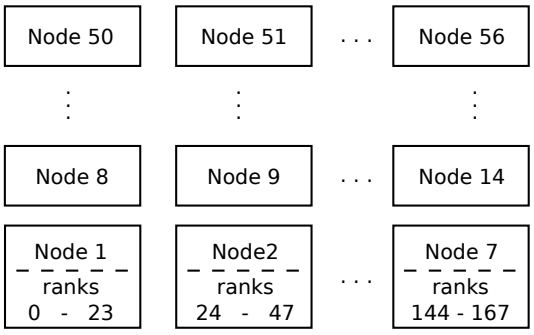
(c) XE6, CAF

(d) Lima, MPI

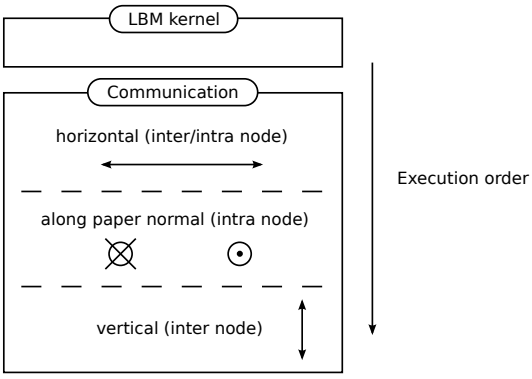
(e) Lima, CAF, buffered

(f) Lima, CAF

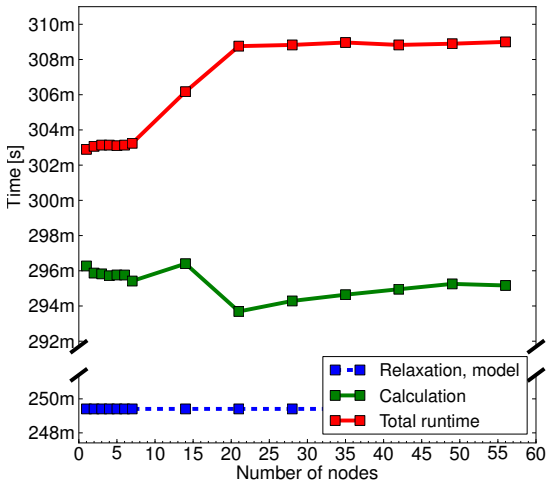
Figure 6.3: Effects of 1D domain decomposition



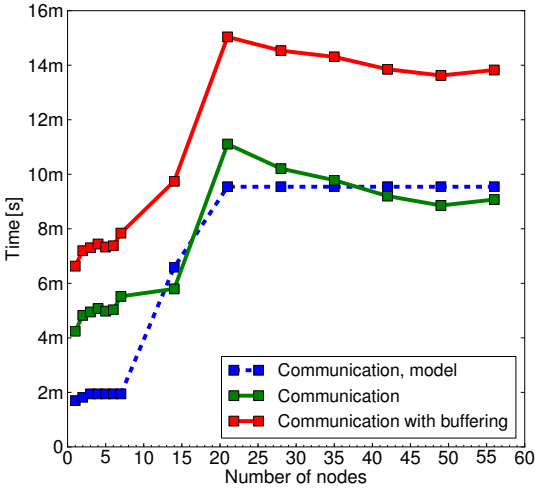
(a) Topology (2D)



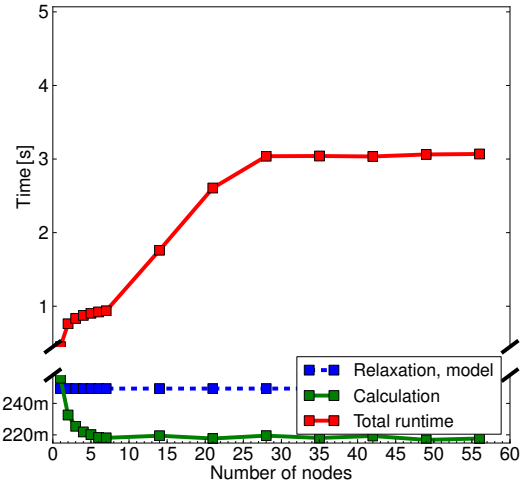
(b) Order of execution



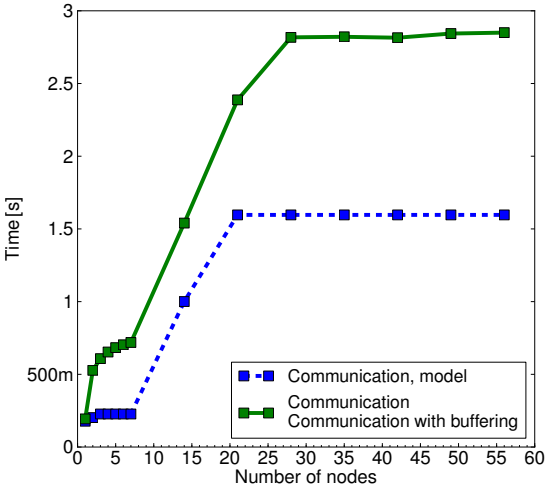
(c) MPI, total runtime and calculation time



(d) MPI, communication time



(e) CAF, total runtime and calculation time



(f) CAF, communication time

Figure 6.4: Weak scaling on Lima

```
Table 1:  Profile by Function Group and Function

 Time%  |    Time  |    Imb.  |  Imb.  |   Calls  |Group
        |          |    Time  | Time%  |          | Function
        |          |          |        |          |  PE=HIDE
        |          |          |        |          |   Thread=HIDE

 100.0% | 1.421276 |       -- |     -- | 135127.2 | Total
|-----------------------------------------------------------------
|  70.1% | 0.996689 |       -- |     -- |    201.6 |USER
|||---------------------------------------------------------------
||  49.5% | 0.703123 | 0.039322 |   6.6% |     17.0 |relax_
||   7.9% | 0.112893 | 0.028810 |  25.4% |     27.2 |copy_cells_1
||   5.5% | 0.077665 | 0.023476 |  29.0% |     17.0 |bounceback_index_
||   0.1% | 0.000734 | 0.000098 |  14.8% |     51.0 |commun...
(....)
||===============================================================
|  17.4% | 0.247249 |       -- |     -- |  15297.8 |ETC
|||---------------------------------------------------------------
||   8.7% | 0.123188 | 0.084440 |  50.8% |     35.6 |__pgas_sync_with_image_list
||   8.0% | 0.113328 | 0.028677 |  25.2% |  15232.0 |__pgas_put_nb
||   0.6% | 0.007853 | 0.001970 |  25.1% |      1.0 |exit
||   0.2% | 0.002490 | 0.001720 |  51.1% |      7.0 |__pgas_barrier_wait
||   0.0% | 0.000155 | 0.000078 |  41.7% |      7.0 |__pgas_barrier_notify
||   0.0% | 0.000072 | 0.000004 |   7.1% |      6.0 |__pgas_barrier
||   0.0% | 0.000050 | 0.000003 |   6.4% |      2.0 |__pgas_sheap_malloc
||   0.0% | 0.000033 | 0.000002 |   5.6% |      1.0 |__pgas_sheap_free
||   0.0% | 0.000032 | 0.000010 |  29.2% |      2.8 |__pgas_get
||   0.0% | 0.000024 | 0.000001 |   6.1% |      1.0 |__pgas_sync_all
||   0.0% | 0.000014 | 0.000002 |  16.6% |      1.6 |__pgas_sync_with_image
||   0.0% | 0.000008 | 0.000002 |  28.6% |      0.8 |__pgas_get_nb
||===============================================================
|  12.5% | 0.177338 |       -- |     -- | 119627.8 |PGAS
|||---------------------------------------------------------------
||   4.3% | 0.060687 | 0.015160 |  25.0% |      7.0 |__pgas_register
||   3.8% | 0.054542 | 0.013848 |  25.3% |  15232.0 |__pgas_memput_nb
||   3.8% | 0.053972 | 0.014584 |  26.6% |  15232.8 |__pgas_sync_nb
||   0.3% | 0.004612 | 0.003815 |  56.6% |  73707.8 |__pgas_poll
||   0.1% | 0.001962 | 0.000763 |  35.0% |  15232.0 |__pgas_sync_nb_adaptive
||   0.0% | 0.000694 | 0.000028 |   4.9% |     85.2 |__pgas_sync_nbi
||   0.0% | 0.000458 | 0.000064 |  15.4% |     49.6 |__pgas_fence
||   0.0% | 0.000264 | 0.000029 |  12.4% |     56.0 |__pgas_afadd_nbi
||   0.0% | 0.000067 | 0.000007 |  12.0% |      7.0 |__pgas_register_dv
||   0.0% | 0.000045 | 0.000002 |   4.7% |      7.0 |__pgas_aor
||   0.0% | 0.000026 | 0.000004 |  18.6% |      7.0 |__pgas_aadd
||   0.0% | 0.000005 | 0.000021 | 100.0% |      1.4 |__pgas_aand
||   0.0% | 0.000002 | 0.000000 |  17.5% |      1.0 |__pgas_local_sheap_free
||   0.0% | 0.000002 | 0.000000 |  12.2% |      2.0 |__pgas_local_sheap_alloc
|===============================================================
```

Listing 6.3: CrayPat profile for LBM

Chapter 7

Conclusion

This work has investigated the performance of contemporary Coarray Fortran programming solutions by means of low-level benchmarks and evaluated their suitability for use in application codes on the example of 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, on an *Infiniband* Intel Westmere Cluster, the Intel Fortran Compiler, the Rice University CAF 2.0 compiler and a development version of the Open64 compiler with CAF support. The Open64 version was not yet released at the time of writing.

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. All the other compilers that were tested were either too unstable (Open64, Rice) or too slow (Intel Compiler).

Low-level 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 was faster than MPI are only to be seen when each process's subproblem is so large that the runtime is clearly dominated by computation. Open64 showed promising performance characteristics but could not be totally evaluated because the resulting executables were not stable enough. As the performance of the MPI version of Open64 was already encouraging one might expect that the InfiniBand version is also able to compete with a native MPI version, but results for Open64 are not available for

the reasons given in section 3.2.4. The Intel Compiler was not yet able to produce competitive communication code. Up to now, its executables show bandwidths that are 1000 times lower than the bandwidths of the MPI implementation, most likely due to elementwise data transfers. The Rice University CAF 2.0 compiler was not benchmarked because it did not yet show a sufficiently large feature set and the compiler version tested was too unstable.

For application developers this means that execution of CAF applications is currently bound to Cray hardware unless they have communication requirements that are so low that the Intel compiler becomes an option. As the compilation of the benchmark codes revealed several bugs inside Open64 and the Rice compiler, they can also be expected to be too unstable to compile most other application codes.

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 should also hold for other stencil codes operating on regular grids.

Comparison of the Lattice Boltzmann single node performance to a copy benchmark showed that the algorithm uses the maximum memory bandwidth available on both the Intel Westmere, and the XE6 Cluster.

List of Figures

List of Tables

Bibliography

[Alve 10] R. Alverson, D. Roweth, and L. Kaplan. "The Gemini System Interconnect". In: *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, pp. 83–87, ACM, 2010.

[Baw 99] 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*, pp. 21–30, IEEE Computer Society, Washington, DC, USA, 1999.

[Bona 02] D. Bonachea. "GASNet Specification, v1.1". University of California, Berkeley, `http://sunsite.berkeley.edu/TechRepPages/CSD-02-1207`, 2002.

[CF90 98] "CF90 Co-array Programming Manual". `http://docs.cray.com/books/004-3908-001/004-3908-001-manual.pdf`, 1998.

[DeRo] L. DeRose. "Detecting Load Imbalance on the Cray XT". Cray Inc., `http://www.cscs.ch/fileadmin/user_upload/customers/CSCS_Application_Data/Files/Presentations/CPW09_Detecting_Load_Imbalance_____on_the_Cray_XT.pdf`.

[Dona 04] S. Donath. "On Optimized Implementations of the Lattice Boltzmann Method on Contemporary High Performance Architectures". Chair for System Simulation (Department of Computer Science 10), University Erlangen-Nuremberg, 2004.

[Done 07] A. Donev. "Rationale for Co-Arrays in Fortran 2008". ISO, `ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1826.pdf`, 2007.

[Feic 11] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde. "WaLBerla: HPC software design for computational engineering simulations". *Journal of Computational Science*, Vol. 2, No. 2, pp. 105–112, 2011.

[Hage 10] 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.

[Inte] "Intel MPI Benchmarks". Intel Corporation, `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`.

[ISO 10] ISO. "Fortran 2008 Language Draft, ISO/IEC JTC 1/SC 22/WG 5/N1826". ISO, `ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1826.pdf`, 2010.

[Likw] "Likwid Website". `http://code.google.com/p/likwid/`.

[Lima] "Lima, RRZE Website". Regional Computing Center Erlangen, `http://www.rrze.uni-erlangen.de/dienste/arbeiten-rechnen/hpc/systeme/lima-cluster.shtml`.

[Ltd 01] Q. S. W. Ltd. "Shmem Programming Manual". `http://staff.psc.edu/oneal/compaq/ShmemMan.pdf`, 2001.

[Mell 09] J. Mellor-Crummey, L. Adhianto, W. N. S. III, and G. Jin. "A New Vision for Coarray Fortran". Department of Computer Science, Rice University, `http://caf.rice.edu/publications/caf2-pgas09-revised.pdf`, 2009.

[MPI 09] "MPI-2.2 Specification". `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`, 2009.

[Numr 98] R. W. Numrich and J. Reid. "Co-Array Fortran for parallel programming". *ACM FORTRAN FORUM*, Vol. 17, No. 2, pp. 1–31, 1998.

[Open] "OpenFabrics Alliance (OFA)". OpenFabrics Alliance (OFA), `http://www.openfabrics.org`.

[Open 11] "OpenMP Specification, Version 3.1". `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, 2011.

[Palu] "Palu, CSCS Website". Swiss National Compute Center, `http://user.cscs.ch/hardware/palu_cray_xe6/index.html`.

[Reid 10] J. Reid. "Coarrays in the next Fortran Standard, ISO/IEC JTC1/SC22/WG5 N1824". JKR Associates, `ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf`, 2010.

[Succ 01] S. Succi. *The lattice Boltzmann equation for fluid dynamics and beyond. Numerical mathematics and scientific computation*, Clarendon Press, 2001, ISBN 9780198503989.

[Well 06] G. Wellein, T. Zeiser, G. Hager, and S. Donath. "On the single processor performance of simple lattice Boltzmann kernels". *Computers & Fluids*, Vol. 35, No. 8-9, pp. 910–919, 2006. Proceedings of the First International Conference for Mesoscopic Methods in Engineering and Science.

[Zeis 09] 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*, pp. 491–511, 2009.