CRESTA

BENCHMARKING MPI COLLECTIVES

CRESTA White Paper

Authors: Christoph Niethammer (USTUTT), Pekka Manninen (Cray),
Rupert Nash (UEDIN), Dmitry Khabi (USTUTT), Jose Gracia (USTUTT)

Editors: Lorna Smith, Catherine Inglis (UEDIN)

Collaborative Research Into Exascale Systemware, Tools and Applications (CRESTA)
ICT-2011.9.13 Exascale computing, software and simulation

SEVENTH FRAMEWORK PROGRAMME

FOREWORD

Collective operations strongly affect the performance of many MPI applications, as they involve large numbers, or frequently all, of the processes communicating with each other.

The MPI 3.0 standard introduced non-blocking collective operations that give the opportunity to speed up applications by allowing overlap of communication with computation, and reducing the synchronisation costs of delayed processes. Many MPI programs are written using non-blocking point-to-point communication operations and application developers are familiar with managing this process using request and status objects. Extending this to include collectives allows programmers to straightforwardly improve application scalability. In contrast to the blocking collectives, their non-blocking counterparts require the MPI implementations to progress the communication task in parallel to computations. This is a non-trivial task, even if the network hardware provides support for offloading network operations from the CPU, e.g., message buffers may have to be refilled for large messages or more complex collective operations need multiple communication steps. The Cray XE6 and XC30 platforms feature a special “asynchronous process engine" for this, which uses spare hyperthreads (XC30) or dedicated CPU cores (XE6) for the required operations.

ABOUT CRESTA

BY **PROFESSOR MARK PARSONS**, COORDINATOR OF THE CRESTA PROJECT AND EXECUTIVE DIRECTOR AT EPCC, THE UNIVERSITY OF EDINBURGH, UK.

The Collaborative Research into Exascale, Systemware Tools and Applications (CRESTA) project is focused on the software challenges of exascale computing, making it a unique project. While a number of projects worldwide are studying hardware aspects of the race to perform 1018 calculations per second, no other project is focusing on the exascale software stack in the way that we are.

By limiting our work to a small set of representative applications we hope to develop key insights into the necessary changes to applications and system software required to compute at this scale.

When studying how to compute at the exascale it is very easy to slip into a comfort zone where incremental improvements to applications eventually develop the necessary performance. In CRESTA, we recognise that incremental improvements are simply not enough and we need to look at disruptive changes to the HPC software stack from the operating system, through tools and libraries to the applications themselves. From the mid-1990s to the end of the last decade, HPC systems have remained remarkably similar (with performance increases being delivered largely through the increase in microprocessor speeds). Today, at the petascale, we are already in an era of massive parallelism with many systems containing several hundred thousand cores. At the exascale, HPC systems may have tens of millions of cores. We simply don't know how to compute with such a high level of parallelism.

CRESTA is studying these issues and identifying a huge range of challenges. With the first exascale system expected in the early 2020s, we need to prepare now for the software challenges we face which, we believe, greatly outnumber the corresponding hardware challenges. It is a very exciting time to be involved in such a project.

CRESTA WHITE PAPERS

BY **DR LORNA SMITH**, PROJECT MANAGER FOR THE CRESTA PROJECT AND GROUP MANAGER AT EPCC, THE UNIVERSITY OF EDINBURGH, UK.

CRESTA is preparing a series of key applications for exascale, together with building and exploring appropriate software – systemware in CRESTA terms - for exascale platforms. Associated with this is a core focus on exascale research: research aimed at guiding the HPC community through the many exascale challenges.

Key outcomes from this research are CRESTA's series of white papers. Covering important exascale topics including new models, algorithms, techniques, applications and software components for exascale, the papers will describe the challenges and current state of the art and propose solutions and strategies for each of these topics.

This white paper considers the use of non-blocking collective operations in applications and assesses the potential performance benefits resulting from the ability to overlap communication with computation. Results are presented for the use of these operations in one of our co-design applications, HemeLB. The HemeLB developers have worked closely with the software developers, making this an excellent example of the use of co-design.

CONTENTS

INDEX OF FIGURES

INDEX OF TABLES

1 INTRODUCTION

This work is structured as follows.

In **section 2** we present our common benchmarking infrastructure.

In **section 3**, we evaluate the impact of late arrivals, i.e. delay, on collective performance. Using collectives in large-scale parallel applications requires the collective operations to have some asynchronous characteristics and that the performance of the collectives be tolerant of some variation in the times when participating processes begin the operation.

In **section 4**, we evaluate an alternative DMAPP-based implementation of two selected MPI collective operations.

In **section 5**, we quantify the benefit from the overlap of computation and communication with non-blocking collectives on the Cray XC30 platform.

In **section 6**, we present a case study on the use of non-blocking collective operations within a real-world application, namely the Lattice-Boltzmann fluids solver HemeLB.

Finally, **section 7** presents our conclusions and outlook.

2 BENCHMARKING INFRASTRUCTURE

Here we describe some of the important features of the micro benchmarking suite used in the following sections. Based on the global time the benchmarks may perform the following tasks:

1. measure start and end times of an operation across all MPI processes;

2. determine the earliest start and latest end time over all involved MPI processes.

The suite is designed for extensibility and to allow the easy addition of new benchmarks. The following MPI collective operations are currently included: `MPI_Barrier, MPI_Bcast, MPI_Reduce, MPI_Allreduce and MPI_Alltoall`. This is as well as their non-blocking counterparts: `MPI_Ibarrier, MPI_Ibcast, MPI_Ireduce, MPI_Iallreduce and MPI_Ialltoall`. Each experiment is run with a different number of processes and different data sizes. Each benchmark is run 100 times initially to warm up and then the timings for 20 benchmark runs are recorded.

For each measurement, the process ID (integer), start time, and end time (both double precision floating point) are stored, requiring S = 20N Bytes of storage. The stored times are times corrected on the basis of the clock skew determined (see below). If not mentioned explicitly, global times for the collective operations are reported, which is the time between the start time of the first process entering and the end time of the last process finishing the collective.

2.1 Global clock

Studying, e.g., the effect of different entry times to a collective's performance requires the use of a global clock. We arbitrarily chose this to be the system clock of the process with rank zero. For this purpose, the micro benchmark suite determines the clock skews between process zero and all other processes.

The local clocks of different processors may differ in the reported times as they are not perfectly synchronised. They may even run at slightly different speeds [4, 7], which we do not take into account here due to the short duration of the benchmarking runs. To compare the times measured, the error between the clocks has to be taken into account. For the collective benchmarks, we consider the clock skew σ that we define as the constant difference between the time measured locally t and the time measured at the remote processor t' at the time point when the benchmark is started.

$$t' = t + \sigma \ . \qquad (1)$$

A modifed ping-pong experiment is used to determine σ, see Figure 1. From this experiment the ping-pong latency λ_p and timer delay Δ, which is the time required to obtain the current time itself, are obtained.

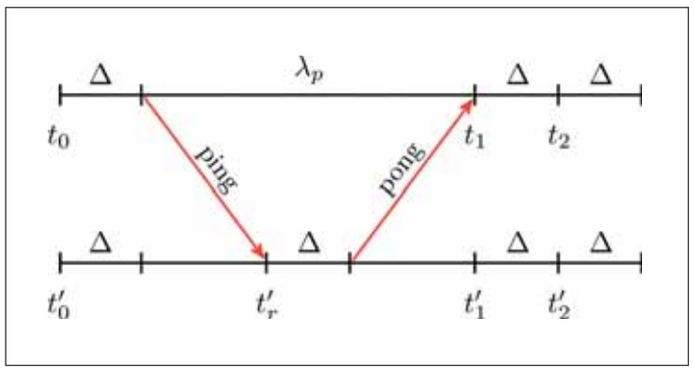

Figure 1: Modified experiment to determine ping pong latency λp, timer delay Δ and clock skew on the basis of the remote time t'_r, which is assumed to be taken at the mid-point of the ping-pong.

By measuring t'_r, which is assumed to be at the mid point of the ping-pong[1], we can compute the clock skew by:

$$\sigma = t'_r - t_0 - (\lambda_p + \Delta)/2 \,. \qquad (2)$$

To verify the correctness and to obtain an estimate for the error of the measured clock skews, intra-node times can be compared. The clock skews between rank zero and all processes residing on one node are nearly the same with a standard deviation of *±2 µs* in 100 measurements, see Table 1.

Rank	$\bar{\sigma}$/s	σ_o/s
0	+0.000000	0.000000
1	+0.000000	0.000000
2	+0.000000	0.000000
3	+0.000000	0.000000
4	-0.017258	0.000002
5	-0.017258	0.000001
6	-0.017258	0.000001
7	-0.017258	0.000002
8	-0.011140	0.000002
9	-0.011140	0.000002
10	-0.011140	0.000002
11	-0.011140	0.000002

Table 1: Determined average clock skew and standard deviation for a benchmark run with twelve processes and four processes per node based on a set of 100 measurements. Results obtained on the Hermit Cray XE6 system at HLRS, see section 3.2 for full details.

2.2 Initial synchronization

To synchronise all the processes involved before each benchmarking run, we use the `MPI_Barrier` routine. The synchronization is not perfect, as can be seen in Figure 2, but currently there is no better way. The time difference at the exit of the barrier is in the order of 4 µs for 32 processes. Measuring the time differences and trying to improve the sync using delays for faster processes results in even poorer synchronization[2].

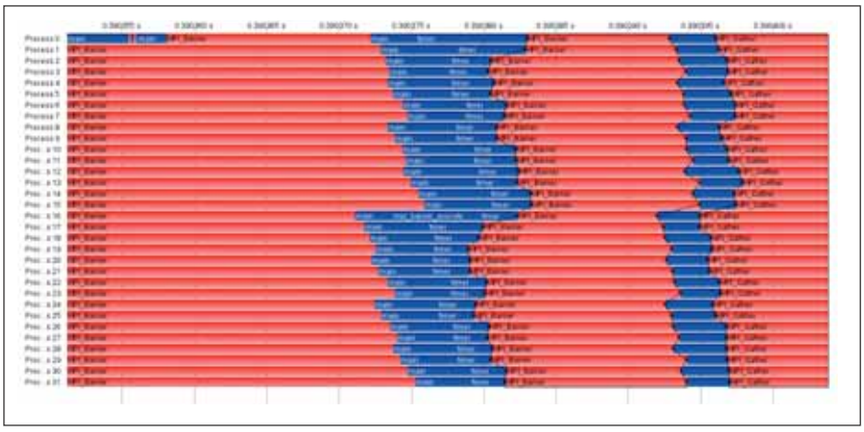

Figure 2: VampirTrace image of the synchronization barrier (first red stripe) before the benchmark (second red stripe) is executed. Benchmarks are run on the Hermit Cray XE6 system at HLRS. The processes enter the benchmark in a time shifted front which appears to come from a binary tree. The blue stripes are timer calls and benchmarking loops. (32 processes on 2 nodes)

[1] This may not be the case for e.g. a ring topology with only one communication direction.

[2] The accuracy of the used delay function was *1 µs*.

3 PERFORMANCE IMPACT OF DELAYED COLLECTIVE START

3.1 Model

One critical issue for the performance of collective operations involving a very large number of MPI tasks is load imbalance, which causes processes to begin collective operations at different times. The influence of delayed processes on this is not currently well understood. In this section we describe a simplified model for this phenomenon and present the results.

Because of the many possible distributions of load imbalance, we select the simplest possible as our starting point. We choose to synchronise all processes as well as possible (see the section below) immediately before delaying a *single* process by a given time δ, as illustrated in Figure 3.

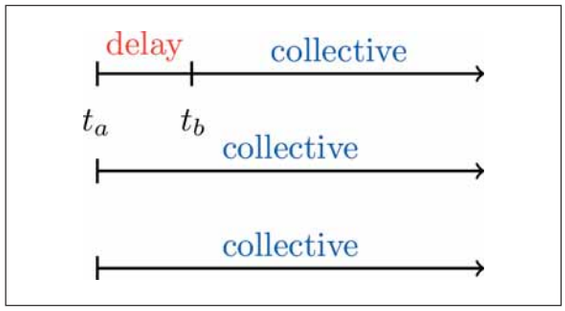

Figure 3: Processes are synchronized at time t_a and enter the collective except one. The delayed process enters the collective at time $t_b = t_a + \delta$.

3.2 Results

Here we report the performance of the collectives with different delay times and numbers of processes. The blocking and non-blocking versions of each operation are compared side by side. Beside the actual collective times, the benefit b of internal overlap of the delay with communication within the collectives itself will be examined:

$$b = \frac{t_0 + \delta - t_\delta}{t_\delta}, \quad (3)$$

with t_0 being the collective time with no delay and t_δ the collective time for delay δ.
In the following sections, results for different collective operations on the Hermit system at HLRS are reported. Hermit is a Cray XE6 system with 3,552 dual-socket compute nodes and a total of 113,664 cores which are connected via the Gemini 3D Torus network. All benchmarks were run during normal operation mode of the system so that other jobs on the system influenced the process placement and network usage. This is responsible for some outlying data points, even if multiple measurements were performed to reduce this effect.

3.2.1 Barrier

The first collective studied is the barrier: `MPI_Barrier` and `MPI_Ibarrier`. As the barrier is used for synchronization within the benchmark suite the understanding of this operation is essential.

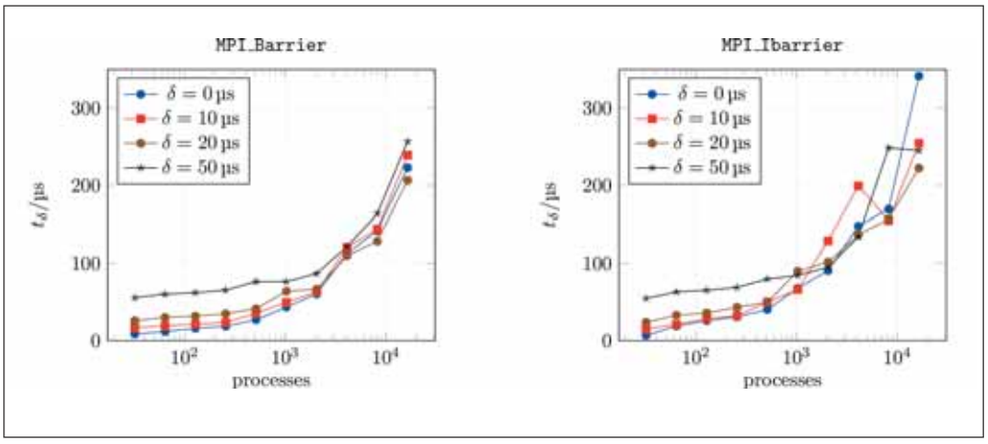

Figure 4: Collective time $t\delta$ for `MPI_Barrier` and `MPI_Ibarrier` versus number of processes, for different delay times.

The results in Figure 4 show a nearly logarithmic scaling of the blocking and non-blocking barrier operation up to approximately 2,048 processes. We note that a single cabinet of the Hermit system has 96 nodes with a total of 3,072 cores. Jobs exceeding this number of processes are more likely to be spread around the system and therefore affected by network contention caused by other applications.

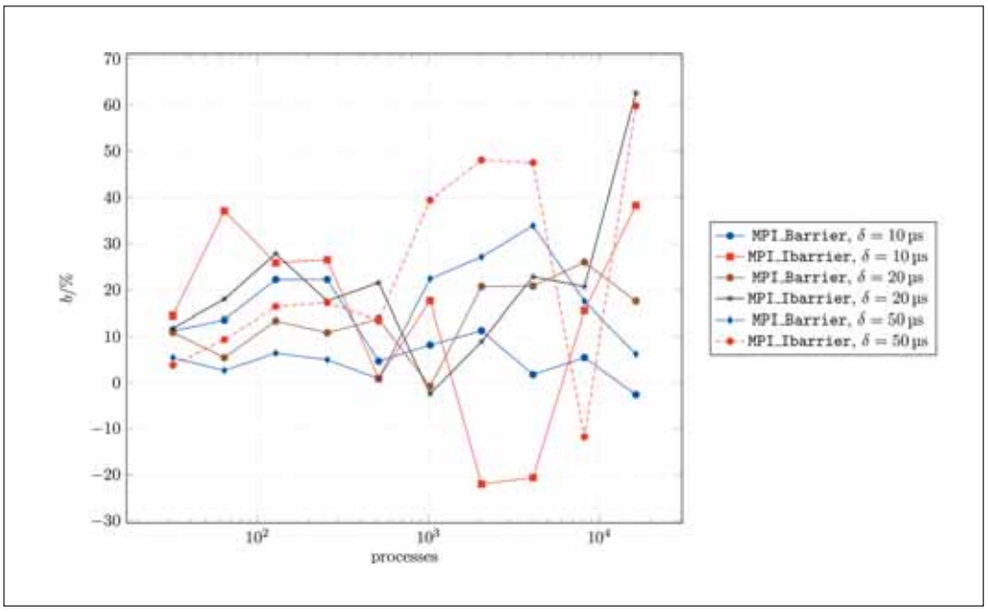

Figure 5: Delay benefit b (3) of the `MPI_Barrier` and `MPI_Ibarrier` collectives versus number of processes for different delay times δ.

Figure 5 shows the delay benefit b as defined in (3) of `MPI_Barrier` and `MPI_Ibarrier` for different delay times, where the delayed rank was always rank zero. As the benefit is mostly positive the MPI implementation's blocking and non-blocking barrier algorithms are hiding some delays in collective start.

3.2.2 All-reduce

The all-reduce collective is used to aggregate data of multiple processes into a single value and then broadcast this to all participants. It is used to determine e.g. global energies in molecular simulations, time step lengths in finite element based programs or residues in linear solvers.

Again, the influence of delaying the process with rank zero for different number of processes is studied. The results are presented in Figure 6.

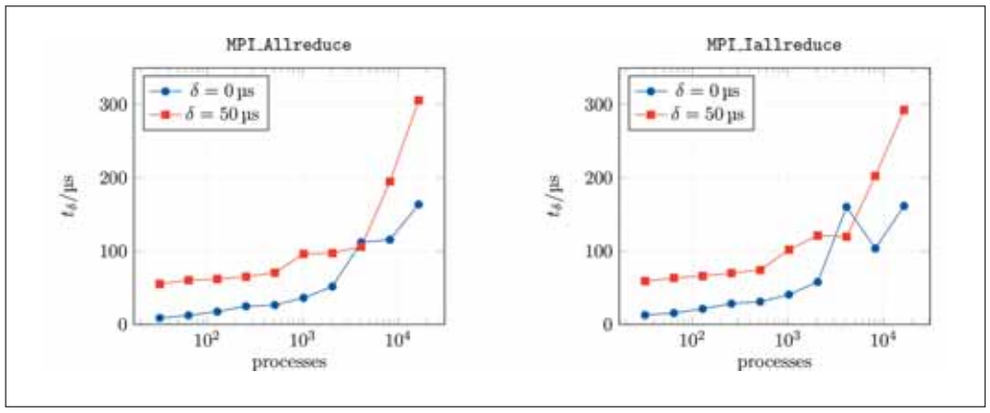

Figure 6: MPI_Allreduce and MPI_Iallreduce collective times tδ versus number of processes for different delay times (message size 8 B).

The effect of the delay on blocking and non-blocking allreduce operations presented in Figure 7 shows slight overlap for smaller number of processes. For more than 1,024 processors the delay has a negative effect on the overall performance. (The peak for 4,096 processes is caused by a too high value for the collective time t_0.)

3.2.3 All-to-all

The all-to-all operation is another important collective pattern used in many parallel codes to distribute data in an application. The same measurements as for the all-reduce operation were performed. Results show similar behaviour as for the allreduce operation, see Figure 8.

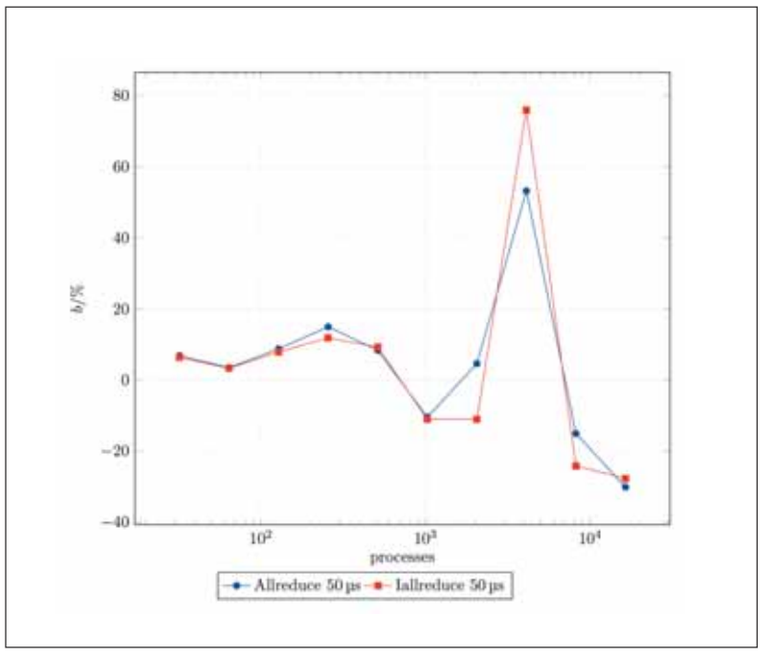

Figure 7: Delay benefit b for the MPI_Allreduce and MPI_Iallreduce collectives versus number of processes (message size 8 B).

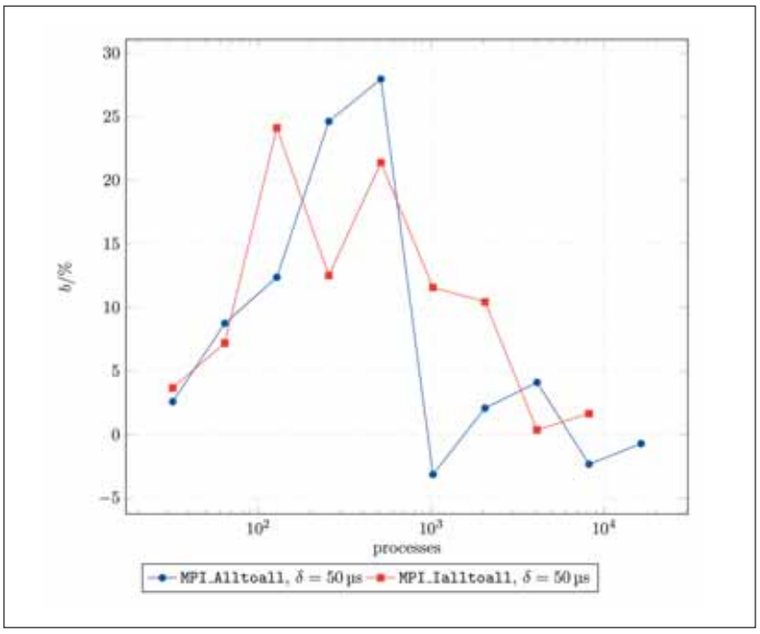

Figure 8: Delay benefit b of the MPI_Alltoall and MPI_Ialltoall collectives versus number of processes (message size 8 B).

4 PERFORMANCE OF DMAPP-BASED COLLECTIVES

DMAPP is a communication library that supports a logically shared, distributed memory programming model. DMAPP provides remote memory access (RMA) between processes within a job in a one-sided manner. One-sided remote memory access requests require no active participation by the process at the remote node; synchronization functions may be used to determine when side effects of locally initiated requests are available. DMAPP is typically not used directly within user application software. The DMAPP API allows one-sided communication libraries and partitioned global address-space (PGAS) compilers, implemented on top of DMAPP, to realize much of the hardware performance of the interconnect.

The MPI library in the Cray XE6 and XC30 platforms features alternative DMAPP-based implementations of selected MPI collective operations. The DMAPP collectives are not enabled by default, but the user of an XE/XC system may link the DMAPP library to his or her application and enable the collectives via an environment variable. The collectives have certain limitations (e.g. not working with MPMD programs and the transaction size being limited). Here we examine the performance of DMAPP collectives compared to the default implementation.

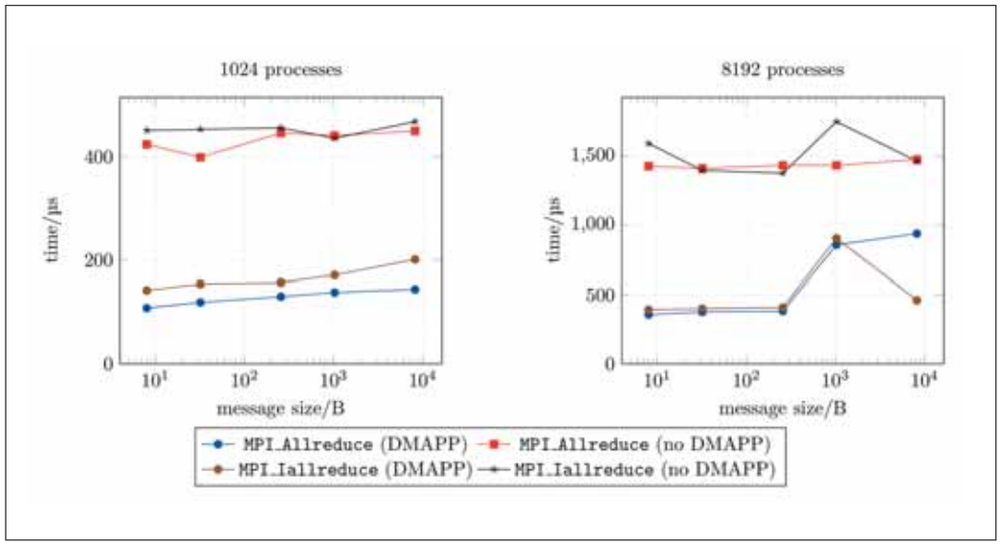

Figure 9: Performance of the DMAPP-based collectives on the Cray XC30: `MPI_Allreduce` and `MPI_Iallreduce`. Left: with 1,024 MPI processes. Right: with 8,192 MPI processes.

In Figure 9 we show measurements, using the benchmark suite described in section 2, of the duration of `MPI_Allreduce` and `MPI_Iallreduce`. On the left, the measurements are being carried out with 1,024 MPI processes, and on the right with 8,192 processes, for varying sized reductions. The benefit of the DMAPP collectives is very significant: for 1,024 cores, a speed-up of more than a factor of three (up to 4x) in `MPI_Allreduce` for all of the measured message sizes. The DMAPP `MPI_Iallreduce` is slightly less optimized than its blocking counterpart, but still outperforms the default implementation by a factor ranging from 2 to 3. With 8,192 cores, the benefit from DMAPP implementation is less pronounced but still significant; again a factor of three for small messages but only some 1.5 for larger `MPI_Allreduce` reductions. Roughly the same applies to `MPI_Iallreduce`.

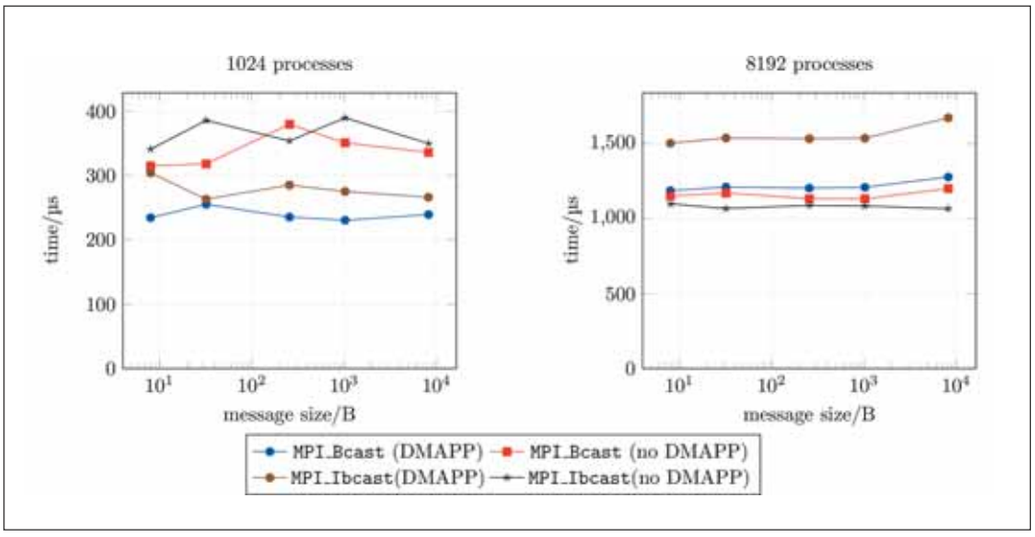

Figure 10: Performance of the DMAPP-based collectives on the Cray XC30: `MPI_Bcast` and `MPI_Ibcast` with 1,024 and 8,192 MPI processes.

In Figure 10 we present similar measurements for the broadcast operations `MPI_Bcast` and `MPI_Ibcast`. We observe that the benefit from the DMAPP implementation is much smaller than for all-reduce. While for the smaller core count the benefit is still significant, up to some tens of percent, at larger core counts both `MPI_Bcast` and `MPI_Ibcast` the DMAPP implementation is in fact slower (especially for `MPI_Ibcast`) than the default implementation.

We recommend that users experiment with the DMAPP collectives, since they can benefit the performance of some operations, while giving no improvement or even a degradation for others.

5 OVERLAP AVAILABILITY OF NON-BLOCKING COLLECTIVES

One key argument for non-blocking collectives is the possibility to hide the parallel overhead caused by the operation by overlapping the communication with computation or other work (see section 6). The usage pattern would then be:

- initialize the collective operation;
- do work that does not require the data involved in the communication;
- and, wait for the collective to finish.

Not all algorithms have this independent work available for the overlap and, even if they do, it depends on the implementation of the MPI library whether the non-blocking communication actually happens simultaneously with the overlapped work, or it occurs only in the waiting phase.

Here we assess the current situation of this "overlap availability" of non-blocking collective operations on the Cray XC30 platform. We study two typical bottleneck collectives, the all-to-all data exchange (`MPI_Alltoall` and `MPI_Ialltoall`) and the global reduction (`MPI_Allreduce` and `MPI_Iallreduce`).

The benchmark is performed as follows:

1. measure the average time over all MPI ranks needed to perform the non-blocking collective operation (T_{coll});
2. measure the average time over all MPI ranks needed to perform a matrix-vector multiplication of size equal to the number of MPI ranks (T_{comp});
3. measure the time needed for the above combined and overlapped operations ($T_{overlap}$).

These steps are repeated and timings are averaged over N_i iterations, where N_i = 100 for messages under 8 kB and N_i = 10 for larger messages. Then, the benefit time T_B from the overlap is

$$T_B = (T_{\text{coll}} + T_{\text{comp}}) - T_{\text{overlap}} \quad (4)$$

and we report the relative benefit β

$$\beta = \frac{T_B}{T_{\text{overlap}}}. \quad (5)$$

This value represents the expected speedup from overlapping. Negative values imply that the overlap in fact slows down the overall execution, and hence it would be better not to overlap at all (compare with "delay overlap benefit" introduced in 3). The benchmark can be found from the CRESTA Collective Communication Library [5].

In the left of Figure 11, we show the relative benefit for the `MPI_Ialltoall` operation measured for 64, 128, 1,024 and 4,096 MPI tasks, as a function of the individual message size (i.e. send buffer size divided by the number of tasks). We observe a general trend that the benefit reduces rapidly for messages over 512 bytes. A benefit of 5–15% can be seen for smaller messages. No clear trends as a function of number of MPI tasks can be recognized.

In the right of Figure 11, we show the results for `MPI_Iallreduce`. Here the message size is the size of the reduction (i.e. the size of the send buffer) divided by the number of tasks. It appears that this operation in practice does not allow for overlap with larger (1,024 and 4,096) core counts with any size of reductions. Within smaller communications, a very modest benefit of up to 4% can be seen with all sizes of reductions.

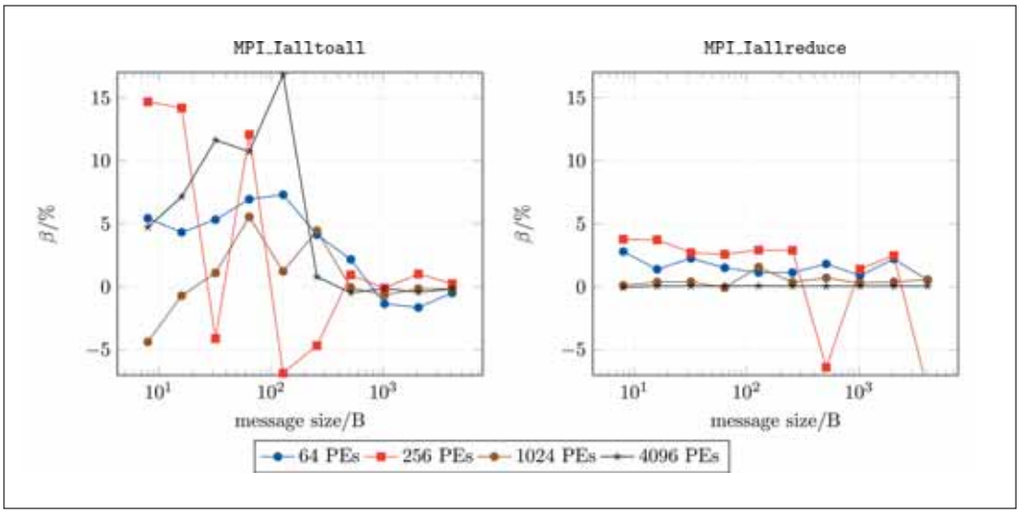

Figure 11: The relative benefit of overlapping communication and computation in the MPI_Ialltoall (left) and MPI_Iallreduce (right) operations.

In summary, the computation communication overlap is not always available, but depending on the platform, operation, the size of the communicator, and the amount of data to be communicated, some performance gains may be obtained by performing the overlap. The programmer should also verify that performing the overlap is not causing performance degradation.

6 NON-BLOCKING COLLECTIVES IN A PRODUCTION APPLICATION

HemeLB [8] is a lattice-Boltzmann based fluids solver, optimised for simulation of blood flow in domains derived from 3D angiography data. It also includes capability for in situ imaging of flow-fields and real-time steering [6]. It is a distributed memory application, parallelized with MPI and written in C++ using an object-oriented design. Previous work has shown that its computational performance scales linearly up to at least 32,768 cores [3] on HECToR, the UK's previous generation national supercomputer. The software is online[3], under the open-source GNU Lesser General Public License (LGPL).

The core lattice-Boltzmann algorithm requires data exchange between neighbouring points only, giving very high potential scalability. Further, HemeLB updates the sites on inter-rank boundaries at the start of the timestep and begins communicating the necessary data, before proceeding to update those sites that do not need data from another rank. The code then waits for communication to finish and updates the boundary sites. Combined with a good domain decomposition, provided by ParMETIS[4], the software scales to tens of thousands of cores.

However, a production application also requires monitoring for convergence and stability. These properties need to be known by all processes to allow the simulation to react appropriately. In the original code, these collective communications (effectively an `MPI_Allreduce`) are split over multiple timesteps using a phased-communication object, which performs a reduction using a tree communication pattern [1]. The phased-communicator uses non-blocking point-to-point MPI operations, which are posted and waited on at the same time as the core lattice-Boltzmann communications. This keeps the performance impact of the global monitoring very low, but comes at the price of significant software complexity and adds a multiple timestep delay until the result is known. As a proof of concept, we have replaced this phased communication with a lightweight wrapper around MPI 3.0 asynchronous collectives.

6.1 Integration of non-blocking collectives

The main loop of HemeLB is orchestrated by a StepManager that controls a number of IteratedActions. An action, such as the core lattice-Boltzmann update, the renderer, the output controller, or the global monitors, is responsible for updating one part of the application each timestep and must do so in distinct stages: before communication, begin non-blocking communication, overlappable computation, waiting for communication to finish, and after communication.

The communication stages may be delegated to a CommunicationNet object, but as this class is not aware of non-blocking collectives, we chose to take full control of the stages in a base class CollectiveAction. The pertinent parts of its class declaration are:

[3] http://github.com/UCL/hemelb

[4] http://glaros.dtc.umn.edu/gkhome/metis/parmetis

```
class CollectiveAction : public IteratedAction {
protected:
//     Constructor duplicates  the  provided
//     communicator  to  ensure
//     concurrent  collectives  do  not  interfere
//     with  each  other
CollectiveAction ( const MpiCommunicator& comm );
public:
//     Optional:  called  before  communication
//     begins;
//     intended to compute rank - local values
virtual void BeginStep () {}
//     Pure virtual ,  must  begin  non - blocking
//     collective
virtual  void  Send ()  =  0;
//     Optional:  perform  any  commputations
//     for this concern that may be overlapped .
virtual void Overlap () {}
//     Implemented here :  calls  MPI_Wait .
void  Wait ();
// Optional: act on result of monitoring
virtual void EndStep () {};
protected:
MpiCommunicator collectiveComm ;
MpiRequest collectiveReq ;
}
```

This is then subclassed by the monitoring objects. For example, one object monitors whether the simulation is in the quasi-incompressible limit as is necessary for the simulation to be considered reliable. This requires knowledge of the relative density range across the entire simulation as well as the maximum Mach number. This is tracked with a simple data type and reduction functions (all members of the IncompressibiltyCheckerClass to aid encapsulation):

```
struct IncompData {
double min;
double max;
double maxVel;
}

void UpdateData (const IncompData& in ,\
            IncompData& inout) {
inout.min = std::min(in.min, inout.min);
inout.max = std::max(in.max, inout.max);
inout.maxVel = std::max(in.maxVel, inout.maxVel);
}

void  MpiOpUpdateFunc ( void *invec, void *inoutvec,\
            int *len, MPI_Datatype *datatype) {
    const IncompData* iv = \
            static_cast<IncompData*>(inoutvec);
    for (int I = 0; I < *len; i++) {
      UpdateData(iv[i], iov[i]);
    }
}
```

Briefly, the implementation of the class's key methods perform the following tasks:

- **Constructor** Creates a user-defined `MPI_Op` (note that the parent class's constructor creates the private MPI communicator).
- **BeginStep** Calculates the maximum and minimum density of all sites updated by the current rank.
- **Send** Begins an `MPI_Iallreduce` of the data.
- **Overlap** No operation.
- **EndStep** No operation.
- **Destructor** Frees the `MPI_Op`.

The other monitors, for stability and the entropy monitoring (to ensure that Boltzmann's H-theorem is obeyed), are implemented similarly.

This implementation process was relatively straightforward, but with much of the time spent investigating problems due to a bug in the OpenMPI implementation of non-blocking user-defined reduction operations. Using this simplified approach allowed us to remove a net 1,117 lines from the source code of HemeLB (2,747 lines removed, 1,630 new lines added) as well as harder-to-quantify simplifications of code design.

6.2 Benchmarking on the Cray XC30

We used the UK's national supercomputer ARCHER, a Cray XC30 with 3,008 nodes, linked with an Aries interconnect. Each node contains two 2.7 GHz, 12-core Intel Ivy Bridge processors with 64 GB of RAM. Simulations were run on a minimum of one node, increasing by a factor of two until the domain decomposition failed to give a valid partition (this occurs when the number of ranks exceeds the number of spatial blocks in the input data; a block contains at most 512 sites).

We selected three problems for these tests: a small section surrounding the bifurcation of the internal carotid artery, discretised with a spatial resolution of 50 μm and 40 μm, and the full circle of Willis discretised at 33 μm. These have, respectively, 650,492, 3,164,555, and 73,039,365 fluid sites and sparsity fractions of 10%, 1%, and 2% (where the sparsity fraction is defined as the volume of fluid divided by the volume of the axis-aligned bounding box of the fluid points). In what follows, these are termed small, medium and large. Each simulation was run for 1,000 time steps.

We compiled three versions of HemeLB for these tests: "Default": the unmodified version of the software with its standard options for Archer (GCC 4.8.2, Cray Message Passing Toolkit 6.3.1); "NBC": the adapted version of HemeLB, using non-blocking collectives for global monitoring, using the default implementations for collectives; "DMAPP": the NBC-enabled version of HemeLB compiled and run with the DMAPP implementations of the collectives.

HemeLB contains a number of timers that measure the total wall clock time spent executing different parts of the simulation. The most relevant for our purposes here are: T_{LB}, the time spent doing lattice-Boltzmann calculations; T_{sim}, the total time from beginning the first time step until ending the last step (i.e. it excludes simulation start up and shut down time); T_{wait}, the time spent waiting for MPI operations to complete; and, T_{mon}, the time spent performing monitoring calculations.

For lattice-Boltzmann codes, the most widely used measure of application is SUPS: the number of site-updates per second. We define this as $S = N_{steps}N_{sites}/T_{sim}$. On the left of Figure 12 we show the scaling of S. In each case, HemeLB shows good strong scaling when core counts increase by a factor of around 100, before performance saturates and begins to decrease at the largest scales. Replotting the perfomance per core against the number of sites per core on the right of Figure 12 shows that this decrease in performance occurs once the average per-core problem size decreases to approximately 2,000 sites. The implementations show small differences in performance: at large core counts, the two NBC ones slightly outperform the default, phased-communication one. However at smaller scales the reverse is true.

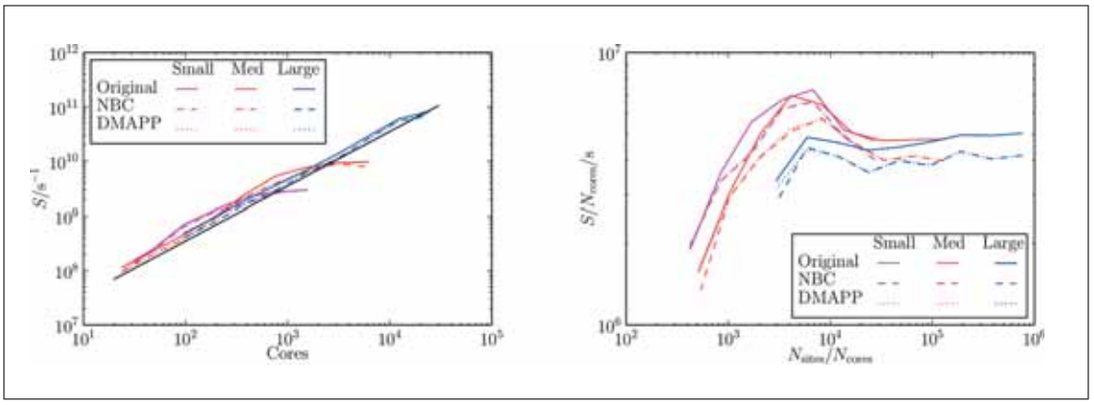

Figure 12: Scaling of HemeLB. Left: total performance in SUPS. Right: per-core performance. The line colour indicates which size of problem was used (small: magenta; medium: red; large: blue) and the style of line indicates which type of collectives were used (default: solid; NBC: dashed; NBC with DMAPP: dotted). The solid black line is a guide to the eye showing perfect linear scaling.

In Figure 13 we show T_{wait} the average time per time step spent waiting for MPI operations to complete (left) and T_{mon} the average time per time step spent on monitoring calculations. It is important to note here that in the default case, the application only performs the monitoring calculations once during the entire time spent in performing the reduction. For p ranks, this takes $2\ ceil(log_{10}p)$ time steps. In the two non-blocking collective implementations, the monitoring is done every time step, which accounts for most of the difference between the monitoring time curves. Despite the much more frequent reductions, the waiting times do appear to be generally lower with non-blocking collectives.

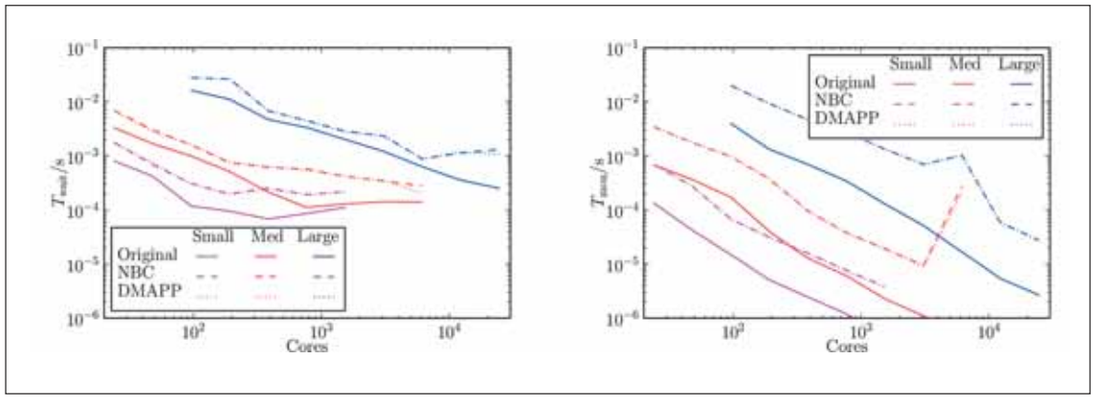

Figure 13: Left: time spent in MPI Waitcalls. Right: time spent in monitoring calculations. The line colour indicates which size of problem was used (small: magenta; medium: red; large: blue) and the style of line indicates which type of collectives were used (default: solid; NBC: dashed; NBC with DMAPP: dotted).

To summarise, we implemented MPI 3.0 non-blocking collectives within a complex, high-performance application with relative ease. The implementation significantly reduced the code complexity of the affected components and does not significantly change the performance, despite allowing significantly more frequent monitoring of global quantities. This also concerns the results obtained with both above considered benchmarks.

7 CONCLUSION AND OUTLOOK

In the first part of this paper, we have evaluated the impact of late arrivals on the collective operations: `MPI_Barrier`; `MPI_Allreduce`; `MPI_Alltoall`; `MPI_Ibarrier`; `MPI_Iallreduce` and `MPI_Ialltoall` on performance. The results show that both blocking and non-blocking collective barriers can tolerate small delays, i.e. hide a part of the load imbalance. The collectives `MPI_Allreduce` and `MPI_Iallreduce` tolerate small delays for up to 1,024 processes but are badly affected for larger numbers of processes. The `MPI_Alltoall` and `MPI_Ialltoall` operations tolerate small delays well for up to 1,024 processes and see no negative effects for larger numbers of processes. The non-blocking `MPI_Ialltoall` version is slightly more performant than its blocking counterpart.

The comparison of the two different implementations of the non-blocking collectives, with DMAPP and without DMAPP, shows that the benefit from the DMAPP implementation is in many cases non-negligible but still the programmer should bear in mind that cases exist where the performance of the DMAPP implementation is worse.

In section 5 we define the overlap availability for non-blocking collectives and show that the benefit of the overlapping depends on the type of the collective operations, size of the communicator and the amount of data to be communicated. The programmer should also verify that the overlap is not causing performance degradation.

The non-blocking collectives were considered not only using synthetic benchmarks but also in an already optimized production application, HemeLB. The detailed description of necessary changes to benefit from the non-blocking collectives were presented. Although our performance measurement has shown that the integration of the non-blocking collectives does not significantly change the performance, the usage of the non-blocking collectives has significantly simplified the monitoring part of HemeLB.

This work shows that the state of the art implementation of the MPI 3.0 non blocking collectives in Cray MPI is as good or better than their blocking counterparts - in benchmarks and real world applications. As the specification of this MPI 3.0 interface is relatively new, we expect new algorithms with better overlapping capabilities and hardware with even better support for offloading communication for the future. The techniques for overlapping communication may also improve collective operations in the case of late arrivals. Our preliminary work in this area shows already some potential to hide small delays of single processes for the barrier, all-reduce and all-to-all operations.

8 ACKNOWLEDGEMENT

This work was supported by the CRESTA project that has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703. This work made use of computational resources provided by EPCC at The University of Edinburgh, United Kingdom (ARCHER) and by the High Performance Computing Center Stuttgart, Germany (Hermit).

9 REFERENCES

[1] Hywel B Carver, Derek Groen, James Hetherington, Rupert W Nash, Miguel O Bernabeu, and Peter V Coveney. Coalesced communication: a design pattern for complex parallel scienti c software. Submitted to Advances in Engineering Software, 2014.

[2] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, Sep. 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[3] Derek Groen, James Hetherington, Hywel B Carver, Rupert W Nash, Miguel O Bernabeu, and Pe-ter V Coveney. Analysing and modelling the performance of the HemeLB lattice-Boltzmann simulation environment. J. Comput. Sci., 4(5):412{422, September 2012.

[4] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21(7):558{565, July 1978.

[5] Pekka Manninen. D4.5.3 - non-blocking collectives runtime library. Technical report, CRESTA FP7-287703, 2013.

[6] Marco D Mazzeo, Steven Manos, and Peter V Coveney. In situ ray tracing and computational steering for interactive blood flow simulation. Comput. Phys. Commun., 181:355{370, 2010.

[7] Steven J. Murdoch. Hot or not: Revealing hidden services by their clock skew. In the 13th ACM Conference on Computer and Communications Security (CCS 2006), pages 27{36. ACM Press, 2006.

[8] Rupert W Nash, Hywel B Carver, Miguel O Bernabeu, James Hetherington, Derek Groen, Timm Kruger, and Peter V Coveney. Choice of boundary condition for lattice-Boltzmann simulation of moderate-Reynolds-number flow in complex domains. Phys. Rev. E, 89:023303, 2014.