MPI collectives at scale

Christoph Niethammer*, Pekka Manninen†, Rupert W. Nash‡, Dmitry Khabi*, Jose Gracia*

* High Performance Computing Center Stuttgart (HLRS)
University of Stuttgart, Nobelstr. 19, D-70569 Stuttgart
Email: {niethammer, khabi, gracia}@hlrs.de
† Cray Finland
Email:manninen@cray.com
‡ EPCC, School of Physics and Astronomy, Edinburgh University, Edinburgh, EJ9 3JZ, UK
Email:rupert.nash@ed.ac.uk

Abstract—Collective operations improve the performance and reduce code complexity of many applications parallelized with the message-passing interface (MPI) paradigm. In this article, we will investigate the impact of load imbalance on the performance of collective operations and possibility for hiding parallel overhead caused by a collective communication pattern, by overlapping the communication with computation. Finally, we will present the use case of non-blocking collectives in a real-world application, namely established lattice-Boltzmann fluids solver, HemeLB.

I. INTRODUCTION

The MPI 3.0 [1] standard introduced non-blocking collective operations which give the new opportunity for the applications by allowing overlap of communication with computation, and reducing the synchronisation costs of delayed processes. In our research we take into account not only the benchmark results for blocking and non-blocking collective operations but also their performance impact on real world applications. To explore the performance consequences of using non-blocking collectives in a production application, we have reimplemented the global monitoring aspects of HemeLB using these operations. HemeLB is a computational fluid dynamics application written in C++ with good scalability up to 32k cores. The implementation significantly reduced the code complexity of the affected components and does not significantly change the performance, despite allowing a higher monitoring frequency.

II. IMPACT OF THE ENTERING TIME ON COLLECTIVE PERFORMANCE

One critical issue for the performance of collective operations is load imbalance, which causes processes to enter collective operations at different times. The influence of such delayed processes is not well understood at the moment. A micro benchmark suite was designed to study the effect of late arrivals on collective operations. This study requires the use of a global clock. For this purpose the micro benchmark suite determines the clock skews [2] between all processes by a modified ping pong experiment. Based on the global time the benchmark performs the following tasks:

1) Measures start and end times of all involved MPI processes.
2) Determines earliest start and latest end time over all involved MPI processes.

Benchmarks are performed for different number of processes and different data sizes. Each benchmark is run initially for skip count times to warm up buffers and network and then the timings for loop count benchmark runs are recorded. A synchronization of all processes is done at the begin of each benchmark run using a barrier. This is not perfect but so far the best way. The time difference at the exit of the barrier is in the order of 4 µs for 32 processes on a Cray XE6.

To study the influence of delayed processes on the overall collective time, the benchmark suite allows to delay a single processes by a given time, see Figure 1.

![Fig. 1. Processes are synchronized at time $t_a$ and enter the collective except one. The delayed process enters the collective at time $t_b = t_a + \delta$.](image)

A. Results

The influence of different delay times and number of processes were studied for blocking collectives and their non-blocking counterparts. Beside the actual collective times the benefit $b$ of internal overlap of the delay with communication within the collectives itself will be examined:

$$b = \frac{t_0 + \delta - t_\delta}{t_\delta},$$

with $t_0$ being the collective time for no delay and $t_\delta$ the collective time for delay $\delta$.

Results for different collective operations on the Hermit system at HLRS are reported. Hermit is a Cray XE6 system with 3552 dual-socket G34 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.

1) Allreduce and Alltoall:

An important collective used to aggregate data of multiple processes into a single value is the allreduce operation. 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. Another important collective pattern is alltoall which is used in many parallel codes to distribute data in an application. For these operations the influence of delaying a single process for different number of processes were studied.

Results for the overlap capability of blocking and non blocking allreduce and alltoall with a 50 µs delay of rank 0 are presented in figure 2. A slight overlap benefit can be seen for smaller number of processes. However for more than 1024 processors the delay has a negative effect onto the overall performance. The peak for 4096 processes is caused by a too high value for the collective time $t_0$.

![allreduce delay overlap](image)

**Fig. 2.** Delay benefit of the blocking and non blocking allreduce and alltoall global times for different a delay time of 50 µs (message size 8 B).

### III. On the Overlap Availability of Non-blocking Collectives

One key argument for non-blocking collective communication is the theoretical possibility for hiding parallel overhead caused by a collective communication pattern, by overlapping the communication with computation or other work. The usage pattern would then be: initialize the collective operation; do work (not accessing 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 takes truly place simultaneously with the overlapped work, or it occurs only at the waiting phase.

Here we study 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 the global collective computation (MPI_allreduce). For many applications it would be beneficial to be able to perform overlapping work during these expensive operations.

The benchmark is performed as follows:

1) Measure the average time over all MPI ranks needed for performing the non-blocking collective operation ($T_{coll}$)

2) Measure the average time over all MPI ranks needed for performing a matrix-vector multiplication of size equaling 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 maxiter iterations, where maxiter is 100 for messages under 8 kB and 10 for larger messages. Then, the benefit from the overlap is

$$T_B = (T_{coll} + T_{comp}) - T_{overlap}$$

and we report the proportional benefit percentage, that is

$$b = 100 \times \frac{T_B}{T_{overlap}}.$$  

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 II). The benchmark can be found from the CRESTA Collective Communication Library [3].

In the top part of Figure 3 the proportional benefit of using overlap in the MPI_alltoall operation is measured as described above for 64, 128, 1024 and 4096 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 benefits reduces to close to zero for messages over 512 bytes. Benefit of 5…15% can be seen for smaller messages. The largest benefit is observed with the largest communicator studied.

In the bottom in Figure 3, the overlap in MPI_allreduce is measured. Here the message size equals to the size of the reduction (i.e. the sendbuffer) divided by the number of tasks. It appears that this operation in practice does not allow for overlap with larger (1024 and 4096 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 within smaller communications, a very modest benefit of up to 4% can be seen with all sizes of reductions.

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.

### IV. Non-blocking Collectives in a Production Application

HemeLB [4] is a lattice-Boltzmann (LB) based fluids solver, optimised for simulation of blood flow in domains derived from 3D angiography data. It is a distributed memory application, parallelized with MPI and written in C++ using an object-oriented design. The software is available online under the open-source GNU Lesser General Public License (LGPL).

The core LB 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\footnote{http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview} previous work has shown that its computational performance scales linearly up to at least 32,768 cores\footnote{http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview} on HECToR, a Cray XE6 supercomputer.

However, a production application also requires monitoring for, e.g., convergence or 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 \texttt{MPI\textunderscore Allreduce}) are split over multiple timesteps, performing the reduction using a tree communication pattern\footnote{http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview}. This keeps the performance impact of the global monitoring 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.

An individual time step is divided into distinct stages: before communication, begin non-blocking communication, overlappable computation, waiting for communication to finish, and after communication. Each aspect of the simulation (e.g. core LB, convergence monitoring) must implement methods that correspond to these stages. For example, the incompressibility monitor performs the following tasks during a simulations. Constructor: create a user-defined \texttt{MPI\textunderscore Op} and a private MPI communicator. Begin step: calculate the maximum and minimum density of all sites updated by the current rank. Send: Begin an \texttt{MPI\textunderscore Iallreduce} of the data. Overlap: no operation. Wait: wait for the \texttt{MPI\textunderscore Iallreduce} to complete. EndStep: log status. Destructor: free the \texttt{MPI\textunderscore Op}. The other monitors, for stability, convergence, and the entropy monitoring (to ensure that Boltzmann’s H-theorem is obeyed), are implemented similarly.

This implementation process was relatively straightforward, taking approximately one week, and 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 harderto-quantify simplifications of code design.

We used the UK’s national supercomputer ARCHER, a Cray XC30 with 3008 nodes, linked with an Aries interconnect. Each node contains two 2.7 GHz, 12-core Intel Ivy Bridge processors with 64 GB of RAM. We selected three problems based on real cerebral artery networks with 650,492, 3,164,555, and 73,039,365 fluid sites and sparsity fractions of 10%, 1%, and 2%. 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.

For LB codes, the most widely used measure of application performance is SUPS: the number of site-updates per second. We observed good strong scaling when core counts increase by a factor of up to 100, before performance saturates and begins to decrease at the largest scales. This decrease in performance occurs once the average per-core problem size decreases to approximately 2000 sites. The three versions 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. We also examine the average time per time step spent waiting for MPI operations to complete, finding that the waiting times do appear to be generally lower with non-blocking collectives.

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 confirms the results obtained with both above considered benchmarks.

**ACKNOWLEDGEMENT**

This work has been 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 the EPCC in Edinburgh, United Kingdom (ARCHER) and by the High Performance Computing Center Stuttgart (Hermit).
REFERENCES


