Optimizing a Conjugate Gradient Solver with Non Blocking Collective Operations

Torsten Hoefler[1,2], Peter Gottschling[1], Wolfgang Rehm[2], and Andrew Lumsdaine[1]

[1] Indiana University, Open Systems Lab, Bloomington, IN 47404 USA
{htor,pgottsch,lums}@cs.indiana.edu
[2] Technical University of Chemnitz, Department of Computer Science, 09107 Chemnitz, Germany
{htor,rehm}@cs.tu-chemnitz.de

Abstract. This paper presents a case study about the applicability and usage of non blocking collective operations. These operations provide the ability to overlap communication with computation and to avoid unnecessary synchronization. We introduce our NBC library, a portable low-overhead implementation of non blocking collectives on top of MPI-1. We demonstrate the easy usage of the NBC library with the optimization of a conjugate gradient solver with only minor changes to the traditional parallel implementation of the program. The optimized solver runs up to 34% faster and is able to overlap most of the communication. We show that there is, due to the overlap, no performance difference between Gigabit Ethernet and InfiniBand™ for our calculation.

1 Introduction

Historically, overlapping communication and computation is the most common approach for scientists to leverage parallelism between processing and communication units [1]. The resulting application is less latency sensitive, and can even, up to a certain extent, run on high latency networks without any change in the parallel speedup. The non blocking operations allow the applications to ignore process skew or network jitter, which often has negative effects on the running time [2]. Both can be very beneficial on Cluster-Computers (also known as Networks of Workstations, NOW) and on Grid-based systems.

The Message Passing Interface (MPI) standard is currently the de-facto standard for parallel computing and many scientific programs exist which use MPI as their communication layer. MPI-1 offers the possibility to overlap communication and computation and to avoid unnecessary synchronization for point-to-point messages (MPI_ISEND, MPI_IRECV). However, many applications can benefit from using MPI collective communication, which is often optimized for the underlying hardware (e.g., [3, 4]) and delivers much better performance than comparable point-to-point communication schemes. Another advantage of collective communication is their abstraction of communication and the resulting ease of use for parallel programs. Gorlatch recently published a good survey about possible reasons to use collective communication [5].

We want to combine both advantages and offer non blocking collective operations, which enable efficient overlapping of communication and computation, for the MPI-1 standard. An assessment of possible benefits has been presented in [6].

1.1 Related Work

The idea to provide non blocking collective operations grew out of discussions for the MPI-2 standard. The MPI Forum defined split collectives which were not standardized in MPI-2, but were written down in the MPI-2 Journal of Development (JoD [7]). However, these operations are too limited to be easily usable for scientists. IBM extended the interface and implemented non blocking collectives as part of their Parallel Environment, but they dropped the support for them in the latest version because they were not part of the MPI standard and were only rarely used by scientists who preferred portability. The upcoming MPI/RT standard [8] defines all operations, including collective operations, in a non blocking manner. Kale et. al. implemented a non blocking all-to-all communication as part of the CHARM++ framework [9]. To the best of the authors' knowledge, there are neither explicit studies on performance gain and nor optimized implementations of non blocking collective operations available.

2 Implementing Non Blocking Collective Operations

Our implementation aims mainly at portability, low overhead, and ease of use. We built the first prototype library on top of non-blocking point-to-point operations defined in the MPI-1 standard. Therefore, although we cannot leverage special hardware features, the protoype library is portable to all MPI-1 capable parallel computers. Further because we implemented optimized algorithms for all collective operations, we deliver the same performance as the hardware independent blocking collective operations in MPICH2 [10] and Open MPI 1.0 [11].

The interface to the calls is very similar to the blocking MPI collective operations. HOwever, to ensure non blocking operation, a handler is returned which is comparable to a MPI_REQUEST. The behavior and the application programming interface (API) of those non blocking collective calls are defined in [12].

The following subsections provide an overview of the implementation of our non blocking collectives (NBC) library, which offers asynchronous collective support on top of MPI-1. The only difference to the definition in [12] is that all calls and constants are prefixed with NBC_ instead of MPI_ to avoid confusion with MPI standardized operations.

2.1 The Scheduling Engine

To ease implementation, we propose a general framework to support all operations. This framework, our scheduling engine, builds and executes a schedule to perform collective operations. Each collective operation, defined in the MPI standard, can be expressed as a row of sends, receives and operations between ranks

of a specific communicator. These functions can be arranged into r communication rounds to build a communicator-specific schedule for each rank. Each round may consist of one or more operations which have to be independent and will be executed simultaneously. Operations in different rounds depend on each other, in a way that operations on round n can only be started after **all** operations in round $n-1$ have finished $\forall\, 0 \leq n \leq r$.

2.2 Building a Schedule

The schedule defines all required actions to perform the collective operation for a specific rank and a specific communicator. A rank's schedule is specific to each communicator and MPI argument set. It is designed to be reusable if it is saved in association to the communicator and the arguments.

A schedule consists of actions (send, receive, operation) and rounds. It is laid out as a contiguous array in memory to be cache friendly. The memory layout of the simplified example schedule for rank 0, for a MPI_BARRIER implemented with the dissemination principle on a four-node communicator is shown in Fig. 1. This schedule has a send operation to rank 1 and a receive operation from rank 3 in the first round. The round is ended by the `end` flag. The second round issues a send to rank 2 and a receive from rank 2. The dissemination barrier is finished after those operations and NBC_TEST or NBC_WAIT calls return NBC_OK.

send to 1	recv from 3	end	send to 2	recv from 2	end

Fig. 1. Memory Layout of a schedule at rank 0, implementing a Dissemination Barrier between 4 nodes

2.3 Schedule Execution

The schedule array in Fig. 1 consists of four operations in two rounds. The schedule represents the necessary operations to perform a MPI_BARRIER on rank 0 of 4. The non-blocking execution of the schedule begins if the user calls NBC_IBARRIER(comm, handle). The first call to NBC_IBARRIER builds the schedule (if not already done), starts all operations of the first round in a non blocking manner, initializes the handle, and returns immediately to the user. The user can perform any computation while the operations are processed in the background. The amount of progress made in the background depends on the actual MPI implementation. The current implementation of the NBC library is runnable in environments which offer no thread support. This means that the user should progress the operation manually by calling NBC_TEST(handle). NBC_TEST checks all pending operations for completion and proceeds to the next round if the current round is completed. It returns NBC_OK if the operation (all rounds) is finished, otherwise NBC_CONTINUE to indicate that the operation is still running.

3 Optimization of Linear Solvers

Iterative linear solvers are important components of most applications in scientific computing. They consume, with very few exceptions, a significant part of the overall run-time of typical applications. In many cases, they even dominate the overall execution time of parallel code. Reducing the computational needs of linear solvers will thus be a huge benefit for the whole scientific community.

Despite the very different algorithms and varying implementations of many of them, one common operation is the multiplication of very large and sparse matrices with vectors. Assuming an appropriate distribution of the matrix, large parts of the computation can be realized on local data and the communication of required remote data — also referred to as inner boundaries or halo — can be overlapped with the local part of the matrix vector product.

3.1 Case Study: 3-Dimensional Poisson Equation

For the sake of simplicity, we use the well-known Poisson equation with Dirichlet boundary conditions, e.g., [13]

$$-\Delta u = 0 \quad \text{in } \Omega = (0,1) \times (0,1) \times (0,1), \tag{1}$$

$$u = 1 \quad \text{on } \Gamma. \tag{2}$$

The domain Ω is equidistantly discretized. Each dimension is split into $N+1$ intervals of size $h = 1/(N+1)$. Within Ω one defines $n = N^3$ grid points

$$G = \{(x_1, x_2, x_3) | \forall i, j, k \in \mathbb{N}, 0 < i, j, k \leq N : x_1 = ih, x_2 = jh, x_3 = kh\}.$$

Thus, each point in G can be represented by a triple of indices (i, j, k) and we denote $u(ih, jh, kh)$ as $u_{i,j,k}$. Lexicographical order allows to store the values of the three-dimensional domain into a one-dimensional array. For distinction we use a typewriter font for the memory representation and start indexing from zero as in C/C++

$$u_{i,j,k} \equiv \mathtt{u}[(\mathtt{i}-1) + (\mathtt{j}-1) * \mathtt{N} + (\mathtt{k}-1) * \mathtt{N}^2] \quad \forall 0 < i, j, k \leq N. \tag{3}$$

The differential operator $-\Delta$ is discretized for each $x \in G$ with the standard 7 point stencil represented as a sparse matrix in $\mathbb{R}^{n \times n}$ using the memory layout from (3), confer e.g. [13] for the 2D case.

3.2 Domain Decomposition

The grid G is partitioned into p sub-grids $G_1, \ldots, G_p$ where p is the number of processors. The processors are arranged in a non-periodic Cartesian grid $p_1 \times p_2 \times p_3$ with $p = p_1 \cdot p_2 \cdot p_3$, provided by MPI_DIMS_CREATE. In case that N is divisible by $p_i \forall i$ the local grids on each processor have size $N/p_1 \times N/p_2 \times N/p_3$, otherwise the local grids are such that the whole grid is partitioned and the sizes along each dimension vary at most by one.

Each sub-grid has 3 to 6 adjoint sub-grids if all $p_i > 1$. Two processors P and P' storing adjoint sub-grids are neighbors, written as the relation $Nb(P, P')$.

```
1  while (sqrt(gamma) > epsilon * error_0) {
2    if (iteration > 1)
3       q = r + gamma / gamma_old * q;
4    v = A * q;
5    delta = dot(v, q);
6    alpha = delta / gamma;
7    x = x + alpha * q;
8    r = r - alpha * v;
9    gamma_old = gamma;
10   gamma = dot(r, r);
11   iteration = iteration + 1;
12 }
```

Listing 1.1. Pseudo-code for CG method

This neighborhood can be characterized by the processors' Cartesian coordinates $P \equiv (P_1, P_2, P_3)$ and $P' \equiv (P'_1, P'_2, P'_3)$

$$Nb(P, P') \quad \text{iff} \quad |P_1 - P'_1| + |P_2 - P'_2| + |P_3 - P'_3| = 1. \tag{4}$$

Fig. 2 shows the partition of G into sub-grids and necessary communication.

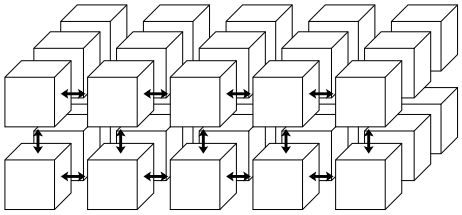

Fig. 2. Processor Grid

3.3 Design and Optimization of the CG Solver

The conjugate gradient method (CG) by Hestenes and Stiefel [14] is a widely used iterative solver for systems of linear equations when the matrix is symmetric and positive definite. To provide a simple base of comparison we restrain from preconditioning [13] and from aggressive performance tuning [15]. However, the local part part of the dot product is unrolled using multiple temporaries, the two vector updates are fused in one loop, and the number of branches is minimized in order to provide a high-performance base case. The parallelization of CG in the form of Listing 1.1 is straight-forward by distributing the matrix and vectors and computing the vector operations and the contained matrix vector product in parallel.

Neglecting the operations outside the iteration, the scalar operations in Listing 1.1 — line 1, 5, 6, 9, and 11 — and part of the vector operations — line 3, 7, and 8 — are completely local. The dot products in line 5 and 10 require communication in order to combine local results with MPI_ALLREDUCE to the global value. Unfortunately, computational dependencies avoid overlapping this

```
1    fill_buffers(v_in, send_buffers);
2    start_send_boundaries(comm_data);
3    volume_mult(v_in, v_out, comm_data);
4    finish_send_boundaries(comm_data);
5    mult_boundaries(v_out, recv_buffers);
```

Listing 1.2. Pseudo-code for parallel matrix vector product

reductions. Therefore, the whole potential to save communication time in a CG method lies in the matrix vector product — line 4 of Listing 1.1.

3.4 Parallel Matrix Vector Product

Due to the regular shape of the matrix, it is not necessary to store the matrix explicitly. Instead the projection $u \mapsto -\Delta u$ is computed. In the distributed case $p > 1$, values on remote grid points need to be communicated in order to complete the multiplication. In our case study, the data exchange is limited to values on outside planes of the sub-grids in Fig. 2 unless the plane is adjoint to the boundary Γ. Therefore, processors must send and receive up to six messages to their neighbors according to (4) where the size of the message is given by the elements in the corresponding outer plane.

However, most operations can be already executed with locally available data during communication as shown in Listing 1.2. The first command copies the values of `v_in` needed by other processors into the send buffers. Then an all-to-all communication is launched, which can be blocking using MPI_ALLTOALLV or non-blocking using NBC_IALLTOALLV The command `volume_mult` computes the local part of the MVP and in case of non-blocking communication, NBC_TEST is called periodically in order to launch new rounds of non-blocking operations, cf. Section 2.1. Before using remote data in `mult_boundaries`, the completion of NBC_IALLTOALLV is checked in `finish_send_boundaries`.

3.5 Benchmark Results

We performed a CG calculation on a grid of $800 \times 800 \times 800$ points until the residual was reduced by a factor of 100, which took 218 iterations for each run. This weak termination criterion was chosen to allow more tests on the cluster. We verified on selected tests with much stronger termination criteria, resulting in over 2000 iterations, that longer executions have the same relative behavior. The studies were conducted on the odin cluster available at the Indiana University which consists of 128 dual 2 GHz Opteron 246 nodes connected with flat InfiniBand™ and Gigabit Ethernet networks. Fig. 3 shows the benchmark results up to 96 nodes. We see that the usage of our NBC library resulted in a reasonable performance gain for nearly all node counts. The performance loss at 8 processors is caused by relatively high effort to test the progress of communication. Finding simple rules to adapt the testing overhead to communication needs is subject to ongoing research. The overall results show that for both networks, InfiniBand™ and Gigabit Ethernet, nearly all communication can be overlapped and the parallel execution times are similar. The factor of 10 in bandwidth and

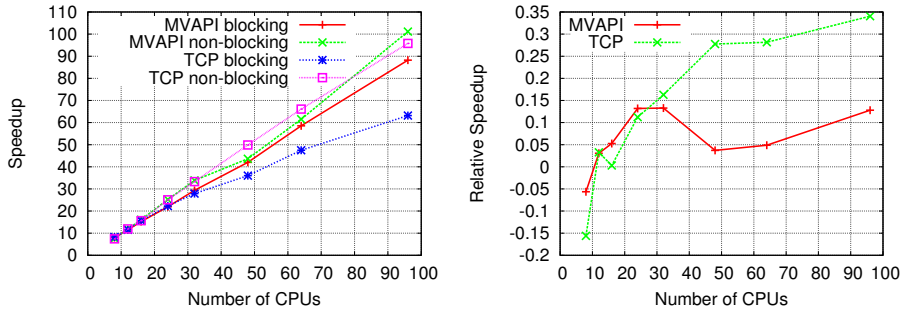

Fig. 3. Parallel Speedup (left) and Relative Performance Gain (right)

the big difference in the latency of both interconnects does not influence the running time, even if the application has high communication needs.

3.6 Optimization Impact on Other Linear Solvers

Other Krylov sub-space methods have comparable dependencies on reduction operations which similarly limit the potential of communication overlapping to parts of the execution. Preconditioners of Krylov sub-space methods are often operations similar to MVP, e.g., incomplete LU or Cholesky factorization, and have to potential of overlapping.

Classical iterative solvers — Richardson iteration, Jacobi, and Gauß-Seidel relaxation — only consist of operations which have the same communication needs as a matrix vector product. Therefore, the whole computation is subject to overlapping. Due to very slow convergence, their importance as iterative solvers is limited. However, these methods are very important to 'smooth' high-frequency error components in multigrid methods [16]. Besides the smoothing process within each level of the multigrid, there are interpolation operators between corresponding sub-grids of adjoint levels, which can depending on the order of interpolation and the relative position of corresponding sub-grids in adjoint levels require communication. This communication is very likely possible to overlap with local computations in interpolation operators. Especially in smaller grids, the communication becomes a severe bottleneck, and non-blocking communication could provide a significant improvement. As multigrid methods are iterative solvers with minimal complexity, they are extremely important in scientific applications and we will investigate them in detail in future work.

4 Conclusions and Future Work

We demonstrated the easy use of the NBC library and the principle of non blocking collectives for a class of application kernels. We were able to improve the parallel application running time by up to 34% with minor changes to the application. The CG solver source code and the NBC library are available at: `http://www.unixer.de/NBC/`.

Future work includes an optimized MPI-2 implementation of the NBC library, hardware optimized non blocking collective operations, and the analysis of more applications.

4.1 Acknowledgments

The authors want to thank Jeff Squyres, George Bosilca, Graham Fagg and Edgar Gabriel for helpful discussions. This work was supported by a grant from the Lilly Endowment and National Science Foundation grant EIA-0202048.

References

[1] Liu, G., Abdelrahman, T.: Computation-communication overlap on network-of-workstation multiprocessors. In: Proc. of the Int'l Conference on Parallel and Distributed Processing Techniques and Applications. (1998) 1635–1642

[2] Petrini, F., Kerbyson, D.J., Pakin, S.: The case of the missing supercomputer performance: Achieving optimal performance on the 8, 192 processors of asci q. In: Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing, 15-21 November 2003, Phoenix, AZ, USA, CD-Rom, ACM (2003) 55

[3] Hoefler, T., Mehlan, T., Mietke, F., Rehm, W.: Adding Low-Cost Hardware Barrier Support to Small Commodity Clusters. In: 19th International Conference on Architecture and Computing Systems - ARCS'06. (2006) 343–350

[4] Liu, J., Mamidala, A., Panda, D.: Fast and scalable mpi-level broadcast using infiniband's hardware multicast support (2003)

[5] Gorlatch, S.: Send-receive considered harmful: Myths and realities of message passing. ACM Trans. Program. Lang. Syst. **26**(1) (2004) 47–56

[6] Hoefler, T., Squyres, J., Rehm, W., Lumsdaine, A.: A Case for non Blocking Collective Operations (2006) submitted to ISPA - preprint available at: http://www.unixer.de/sec/nbcoll.pdf.

[7] Message Passing Interface Forum: MPI-2 Journal of Development (1997)

[8] Kanevsky, A., Skjellum, A., Rounbehler, A.: MPI/RT - an emerging standard for high-performance real-time systems. In: HICSS (3). (1998) 157–166

[9] Kale, L.V., Kumar, S., Vardarajan, K.: A Framework for Collective Personalized Communication. In: Proceedings of IPDPS'03, Nice, France (2003)

[10] MPICH2 Developers: http://www-unix.mcs.anl.gov/mpi/mpich2/ (2006)

[11] Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (2004)

[12] Hoefler, T., Squyres, J.M., Bosilca, G., Fagg, G.: Non Blocking Collective Operations for MPI-2 (2006) preprint available at: http://www.unixer.de/sec/standard_nbcoll.pdf.

[13] Hackbusch, W.: Iterative solultion of large sparse systems of equations. Springer (1994)

[14] Hestenes, M., Stiefel, E.: Methods of conjugate gradients for solving linear systems. J. Res. Natl. Bur. Stand. **49** (1952) 409–436

[15] Gottschling, P., Nagel, W.E.: An efficient parallel linear solver with a cascadic conjugate gradient method. In: EuroPar 2000. Number 1900 in LNCS (2000)

[16] Trottenberg, U., Oosterlee, C., Schüller, A.: Multigrid. Academic Press (2000)