Available online at www.sciencedirect.com

ScienceDirect

Parallel Computing 33 (2007) 624–633

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Optimizing a conjugate gradient solver with non-blocking collective operations

Torsten Hoefler [a,b,*], Peter Gottschling [a], Andrew Lumsdaine [a], Wolfgang Rehm [b]

[a] *Indiana University, Open Systems Lab, Bloomington, IN 47404, USA*
[b] *Technical University of Chemnitz, Department of Computer Science, 09107 Chemnitz, Germany*

Received 22 December 2006; received in revised form 7 June 2007; accepted 29 June 2007
Available online 27 July 2007

Abstract

This paper presents a case study that analyzes the suitability and usage of non-blocking collective operations in parallel applications. As with their point-to-point counterparts, non-blocking collective operations provide the ability to overlap communication with computation and to avoid unnecessary synchronization. These operations are provided for MPI programs with LibNBC, a portable low-overhead implementation of non-blocking collective operations built on MPI-1. The straightforward applicability of the LibNBC is demonstrated by incorporating non-blocking collective operations into a parallel conjugate gradient solver. Although only minor changes are required to use them, non-blocking collective operations allow most of the communication costs to be hidden and provide performance improvements of up to 34%. We also show that, because of overlap, there is no significant performance difference between Gigabit Ethernet and InfiniBand^{TM} for special cases of our calculation.

Keywords: Message passing interface (MPI); Communication; Computation overlap; Collective operations; Non-blocking collective operations; Poisson solver

1. Introduction

Historically, overlapping communication and computation is a common approach for scientists to leverage parallelism between processing and communication units [16]. Applications with overlapped computation and communication are less latency sensitive and can, up to a certain extent, still achieve good parallel performance and scalability on high-latency networks. Non-blocking operations (by which overlap is typically effected) can also help to isolate applications from process skew or network jitter—effects which often have a substantial negative impact on the scalability of parallel applications [22]. The ability to ignore process skew and hide message transmission latencies can be especially beneficial on cluster computers (also known as Networks of Workstations, NOW) and on Grid-based systems.

* Corresponding author. Address: Indiana University, Open Systems Lab, Bloomington, IN 47404, USA.
E-mail addresses: htor@cs.indiana.edu (T. Hoefler), pgottsch@cs.indiana.edu (P. Gottschling), lums@cs.indiana.edu (A. Lumsdaine), rehm@cs.tu-chemnitz.de (W. Rehm).

0167-8191/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2007.06.006

The Message Passing Interface (MPI) standard [18,19] is currently the de facto standard for parallel computing and many scientific programs use MPI as their communication layer. MPI-1 is able to overlap communication and computation for point-to-point messages (e.g., with MPI_ISEND, MPI_IRECV). Many applications also benefit from using MPI collective operations, because they are often optimized for the underlying hardware (e.g., [10,17]) and deliver much better performance than equivalent point-to-point communication schemes. Another advantage of collective communication is the programming abstraction it provides and the resulting ease of use for parallel programs. A survey by Gorlatch about reasons to use collective communication recently appeared in [6].

Scientific computing applications are particularly well-suited to benefit from the more abstract expression of parallel communication afforded by collective operations. Moreover, many algorithms in scientific computing, e.g., linear solvers, provide a high potential for overlapping communication and computation. In order to combine the advantages of this overlapping with the advantages of collective communication, we introduce non-blocking collective operations as a natural addition to the MPI-1 standard and demonstrate the performance benefits with a parallel conjugate gradient solver. A theoretical discussion of possible benefits is presented in [12] and shows very high optimization potential.

1.1. Related work

Although, the conventional wisdom has been that non-blocking collective operations could be emulated or implemented by using threads and blocking operations, this approach has not been proven to be useful in practice. Emulating non-blocking operations in this way is contrary to the basic philosophy of MPI because of the programming burden it imposes on the user (the functionality is better provided by a functional interface). Implementing non-blocking collectives in this way (where the threading machinery is hidden from the user by the library) also has disadvantages, which we discuss in Section 2.

The original 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 added to the MPI-2 Journal of Development (JoD [20]). However, these operations are too limited to be useful 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 MPI/RT standard [15] 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 [14]. To the best of the authors' knowledge, there are neither explicit studies on performance gain nor optimized implementations of non-blocking collective operations available.

We discuss our high-performance and low-overhead implementation and show its advantages for a parallel three-dimensional conjugate gradient solver. The following section discusses implementation issues of LibNBC and presents necessary details about its working principles. Section 3 describes the general applicability of non-blocking collective communication to applications in scientific computing, presents a specific example code and discusses performance benefits. The work is concluded in Section 4 and directions of future work are discussed.

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 prototype library is portable to all MPI-1 capable parallel computers. Furthermore, because we implemented optimized algorithms for all collective operations, we deliver approximately the same performance as the hardware independent blocking collective operations in MPICH2 1.0.2 [21] and Open MPI 1.0 [5] if we use the non-blocking collectives in a blocking manner.

A second way to implement non-blocking collective operations would be to use blocking collective operations in a separate thread. However, we decided to base our implementation on non-blocking point-to-point messages because using the threaded version would require an MPI-2 compliant library with full MPI_THREAD_MULTIPLE support and thus lower the portability. Furthermore, a separate thread would keep the processing units busy and incur additional overhead. A detailed analysis and comparison of those two techniques is subject of future work and preliminary results show significant advantages in using non-blocking point-to-point messages.

Listing 1. Pseudo-code for a non-blocking reduction

```
 1  MPI_Request req;
 2  int sbufl[SIZE], rbufl[SIZE], buf2[SIZE];
 3
 4  /* compute sbufl */
 5  compute(sbufl, SIZE);
 6  /* start non-blocking allreduce of sbufl */
 7  MPI_Iallreduce(sbufl, rbufl, SIZE, MPI_INT, MPI_SUM,
 8     MPI_COMM_WORLD, &req);
 9  /* compute buf2 (independent of bufl) */
10  compute(buf2, SIZE);
11  MPI_Wait(&req, &stat);
12  /* use data in rbufl */
13  evaluate(rbufl, buf2, SIZE);
```

The interface to the calls is very similar to the blocking MPI collective operations. However, to ensure non-blocking operations, a handler is returned which is comparable to an MPI_REQUEST. A pseudo-code in Listing 1 shows a non-blocking reduction call as an example. The communication of the data in `sbufl` is started in the background (line 7) and independent data is computed concurrently in `buf2` (line 10). The result of the communication can only be used after MPI_WAIT (line 11) returns success. In the ideal case if the computation of `buf2` (line 10) takes at least as long as the communication in the background, most of the communication latency can be ignored. The detailed behavior of all non-blocking collective operations and the application programming interface (API) is defined in [11].

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 between the definition in Listing 1 and our earlier publication [11] 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 in a non-blocking manner. This framework, our scheduling engine, builds and executes a so called "schedule" to perform collective operations. Each collective operation, defined in the MPI standard, can be expressed as a row of sends or receives between ranks of a specific communicator and rank-local operations. These steps 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 are 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 been finished $\forall 1 \leqslant n \leqslant r$.

2.2. Building a schedule

The schedule defines all actions that are necessary 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 (e.g., `send`, `receive`) 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 an MPI_BARRIER implemented with the two-way dissemination principle on a 9-node communicator, is shown in Fig. 1.

send to 1	send to 2	recv from 7	recv from 8	end	send to 3	send to 6	recv from 3	recv from 6

Fig. 1. Memory layout of a schedule at rank 0, implementing a two-way dissemination barrier between nine nodes.

The two-way dissemination barrier (a special case of the n-way dissemination barrier, described in [13]) on P nodes uses $\log_3 P$ rounds to complete. Every node sends and receives two synchronizing messages each round. Rank p's two peers to send to ($speer_i$) and to receive from ($rpeer_i$) $\forall 1 \leqslant i \leqslant 2$ in round r ($0 \leqslant r < \log_3 P$) are determined as follows:

$$speer_i = (p + i \cdot (n+1)^r) \bmod P$$
$$rpeer_i = (p - i \cdot (n+1)^r) \bmod P$$

This schedule has two send operations to ranks 1 and 2 and two receive operations from ranks 7 and 8 in the first round. The round is ended by the `end` flag. The second round issues two sends to ranks 3 and 6 and two receives from ranks 6 and 3. The dissemination barrier is finished after those operations. User-calls to NBC_TEST or NBC_WAIT do now return NBC_OK instead of NBC_CONTINUE.

This design benefits from several network properties. Having four outstanding requests in each round enables efficient overlap of communication with communication (cf. LogGP [1,3]) and maximum asynchronous progress. All state transitions (proceeding from one round to the next) are fully local operations, i.e., there is no additional communication necessary to execute a schedule.

2.3. Schedule execution

The schedule array in Fig. 1 consists of eight operations in two rounds. The schedule represents all necessary operations to perform an MPI_BARRIER on rank 0 out of 9. The non-blocking execution of the schedule begins when 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.

The following section explains the applicability of the new interface to the application class of iterative linear solvers.

3. Optimization of linear solvers

Accelerating parallel applications in scientific computing is a main topic of many research projects. Non-blocking collective communication can be an important contribution to it and we will demonstrate this on a selected case study.

Iterative linear solvers are important components of most applications in SC. 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: three-dimensional Poisson equation

For the sake of simplicity, we use the well-known Poisson equation with Dirichlet boundary conditions, e.g., [8]

$$
\begin{aligned}
-\Delta u &= 0 \quad \text{in } \Omega = (0,1)\times(0,1)\times(0,1), && (1)\\
u &= 1 \quad \text{on } \Gamma. && (2)
\end{aligned}
$$

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

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

Thus, each point in G can be represented by a triple of indices $(i,j,k\cdot)$ and we denote $u(ih,jh,kh)$ as $u_{i,j,k}$. Lexicographical order allows for storing 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++

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

The differential operator $-\Delta$ is discretized for each $x \in G$ with the standard 7-point stencil

$$-\Delta_h u_{i,j,k} = \frac{6u_{i,j,k} - u_{i-1,j,k} - u_{i+1,j,k} - u_{i,j-1,k} - u_{i,j+1,k} - u_{i,j,k-1} - u_{i,j,k+1}}{h^2}.$$

Setting this equation equal to zero for all $x \in G$ provides an approximation of (1) on Ω. Considering that the function u is given on the boundary, the corresponding terms can be transferred to the right hand side, e.g., for $-\Delta_h u_{1,3,1}$ the equation reads

$$\frac{6u_{1,3,1} - u_{2,3,1} - u_{1,2,1} - u_{1,4,1} - u_{1,3,2}}{h^2} = \frac{u_{0,3,1} + u_{1,3,0}}{h^2} = \frac{2}{h^2}.$$

The linear operator $-\Delta_h$ can be represented as a sparse matrix in $\mathbb{R}^{n\times n}$ using the memory layout from (3), confer e.g. [8] for the 2D case.

Listing 2. Pseudo-code for CG method

```
while (sqrt(gamma) > epsilon * error_0) {
  if (iteration > 1)
    q = r + gamma / gamma_old * q;
  v = A * q;
  delta = dot(v, q);
  alpha = delta / gamma;
  x = x + alpha * q;
  r = r - alpha * v;
  gamma_old = gamma;
  gamma = dot(r, r);
  iteration = iteration + 1;
}
```

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.

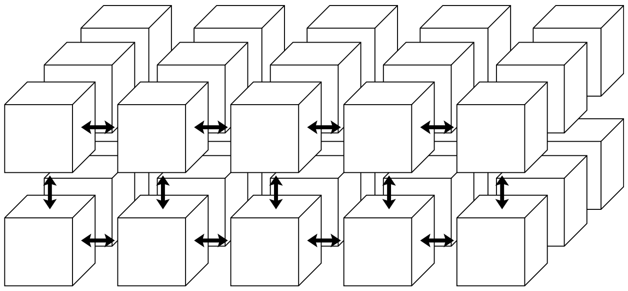


Fig. 2. Processor grid.

Each sub-grid has 3–6 adjoint sub-grids if all $p_i > 1$. Two processors P and P' storing adjoint sub-grids are neighbors, written as the relation $Nb(P, P')$. This neighborhood can be characterized by the processors' Cartesian coordinates $P \equiv (P_1, P_2, P_3)$ and $P' \equiv (P'_1, P'_2, P'_3)$

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

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

3.3. Design and optimization of the CG solver

The conjugate gradient method (CG) by Hestenes and Stiefel [9] 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 refrain from preconditioning [8] and from aggressive performance tuning [7]. However, the local 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 2 is straightforward 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 2 – line 1, 2, 6, 9, and 11 – and part of the vector operations – line 3, 7, and 8 – are completely local. The dot products in lines 5 and 10 require communication in order to combine local results with MPI_ALLREDUCE to the global value. Unfortunately, computational dependencies avoid overlapping these reductions. Therefore, the whole potential to save communication time in a CG method lies in the matrix vector roduct – line 4 of Listing 2.

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.

Listing 3. Implementation of parallel matrix vector product

```
void matrix_vector_mult(struct array_3d *v_in,
                        struct array_3d *v_out,
            struct comm_data_t *comm_data)
{
    fill_buffers(v_in, &comm_data->send_buffers);
    start_send_boundaries(comm_data);
    volume_mult(v_in, v_out, comm_data);

    finish_send_boundaries(comm_data);
    mult_boundaries(v_out, &comm_data->recv_buffers);
}
```

However, most operations can be already executed with locally available data during communication as shown in Listing 3. 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 a blocking operation using MPI_ALLTOALLV or a non-blocking operation using NBC_IALLTOALLV, Listing 4. The last function has the same arguments as the first one with an additional NBC_HANDLE that is used to identify the operation later. The command `volume_mult` computes the local part of the matrix–vector product and in case of non-blocking communication, NBC_TEST is called periodically with the handle returned by NBC_IALLTOALLV in order to progress the 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` with an NBC_WAIT on the NBC_HANDLE, Listing 5.

Listing 4. Code for starting communication

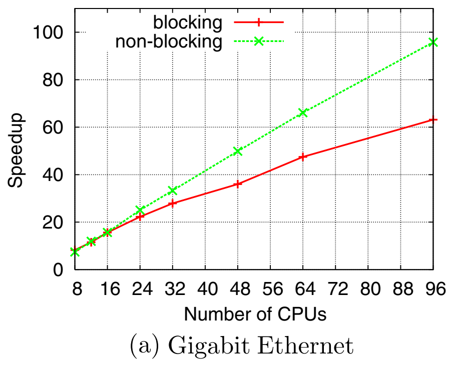
```
void start_send_boundaries(struct comm_data_t *comm_data)
{
  /* Compute displacements */
  if (comm_data->non_blocking)
    NBC_Ialltoallv(sbuf.start, scounts, sdispls, MPI_DOUBLE,
            rbuf.start, rcounts, rdispls, MPI_DOUBLE,
            processor_grid, comm_data->handle);
  else{
    MPI_Alltoallv(sbuf.start, scounts, sdispls, MPI_DOUBLE,
          rbuf.start, rcounts, rdispls, MPI_DOUBLE, processor_grid);
}
```

Listing 5. Code for finishing communication

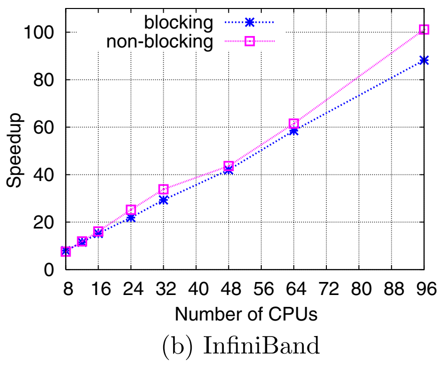
```
void finish_send_boundaries(struct comm_data_t *comm_data)
{
  if (comm_data->non_blocking)
    NBC_Wait(comm_data->handle);
  gt2 = MPI_Wtime();
}
```

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 for practical reasons in order to allow more tests on the cluster. We verified on selected tests with much stronger termination criteria 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^{TM} and Gigabit Ethernet networks. Fig. 4 shows the benchmark results using Gigabit Ethernet and InfiniBand^{TM} up to 96 nodes. The presented speedups are relative single-processor run without any communication. We see that the usage of our NBC library resulted in a reasonable performance gain for nearly all node counts. The explicit performance advantage is shown in Fig. 3. The performance loss at eight 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.



(a) Gigabit Ethernet



(b) InfiniBand

Fig. 3. Parallel speedup for different network interconnects.

The overall results show that for both networks, InfiniBand^{TM} and Gigabit Ethernet, nearly all communication can be overlapped and the parallel execution times are similar. The factor of 10 in bandwidth and the big difference in the latency of both interconnects does not impact the run-time significantly, even if the application has high communication needs. The partially superlinear speedup is due to cache effects in the inner part of the matrix–vector product.

3.6. Comparison to non-blocking point-to-point messaging

Due to the our design, non-blocking point-to-point communication would perform almost equally while requiring the user to program the management for multiple communication handlers including the progress enforcement. The implementation of NBC_ALLTOALLV in LibNBC is not optimized yet and uses a linear set of non-blocking point-to-point operations (like the "manual" implementation would do). However, using collective communication simplifies programming, increases maintainability and enables portable performance tuning. Possible optimizations for ALLTOALLV calls include the exploitation of network concurrency (cf. [25]), optimization for special parallel systems like BlueGene/L (cf. [2]) or automatically tuning the communication plans (cf. [4]).

We are aware that the (slow but steady) linear growing of the displacement and count arrays with the communicator size can introduce scalability problems on very large machines (e.g. BlueGene/L). However, the better programmability, high optimization potential, and clearer code-structure (cf. [6]) outweigh those concerns.

Although the arguments in the previous paragraphs favor ALLTOALLV over point-to-point communication, we find that neither of them is suitable at a large scale for the application at hand. MPI provides a creation of Cartesian grid communicators with MPI_CART_CREATE and the topology of these communicators matches perfectly with the regular n-dimensional domain decomposition of cuboidal spaces like the one considered in our application. We believe that an operation like MPI_CART_NEIGHBOR_ALLTOALL could facil-

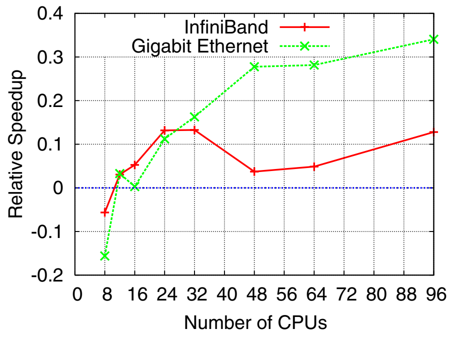


Fig. 4. Speedup of the non-blocking compared to the blocking implementation.

itate the programming of nearest-neighbor communication considerably over both ALLTOALLV and point-to-point implementations.

Furthermore, the information of the Cartesian grid's periodicity can be used to block out communications at the grid's outer boundaries. For instance, in our 3D-grid, sub-domains in the center have six nearest neighbors and sub-domains at the corners only three. Thus, it can be determined from the communicator, with how many neighbors each processor communicates. In addition, the consideration of the grid topology (opposed to ALLTOALL) enables fixed-size messages in many cases, for instance exchanging equally sized squared surfaces of cubic sub-domains. Last but not least, a collective communication based on a regular grid instead of an ALLTOALL or a point-to-point exchange provides the potential of optimizing the internal schedule toward the underlying network.

3.7. Optimization impact on other linear solvers

The approach described above can be used to optimize other linear solvers in addition to CG. We discuss some ideas for the application of non-blocking collective operations to different algorithms.

As with CG, other Krylov sub-space methods, such as GMRES [23], CG Squared [24], or BI-CGStab [27], have dependencies that similarly limit the potential of overlapping communication and computation for their reduction operations. On the other hand, the preconditioners that are typically used in conjunction with Krylov sub-space iterations often consist of operations that are similar to matrix–vector product, e.g., incomplete LU or Cholesky factorization, and thus have the potential of overlapping.

Classical iterative solvers — Richardson iteration, Jacobi, and Gauß–Seidel relaxation — consist only of operations similar to matrix–vector product. Such iterations therefore offer the potential for significant overlap in contrast to CG and related Krylov sub-space methods that require reduction operations. Unfortunately, due to the slow convergence of these methods, their importance as iterative solvers is limited.

The classical methods are important, however, as "smoothers" in multigrid methods (MG) [26]. In addition to the smoothing process within each level of the multigrid, corresponding sub-grids of adjoint levels are related to each other by interpolation operators. These operators involve communication to interpolate values close to sub-grid boundaries. The amount of communication increases for higher orders of interpolation. Grid values inside the sub-grids can be interpolated with local data—assuming the grids are similarly decomposed—so that the interpolation operators allow for communication overlapping. Particularly on smaller grids, communication becomes a severe bottleneck and non-blocking communication provides the potential for significant improvements. As multigrid methods are solvers with minimal numerical complexity, they are extremely important in scientific computing 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 application principle of non-blocking collectives to 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. The possibility of asynchronous progress, which removes the need for testing, with a separate thread will also be investigated. However, this may have other implications because the user cannot control when the library gets called and possibly wipes out the CPU cache.

Acknowledgements

Pro Siobhan. The authors want to thank Jeff Squyres, George Bosilca 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] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, Chris Scheiman, LogGP: incorporating long messages into the LogP model, Journal of Parallel and Distributed Computing 44 (1) (1995) 71–79.
[2] George Almasi, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, Jose E. Moreira, B. Steinmacher-Burow, Yili Zheng, Optimization of MPI collective communication on BlueGene/L systems, in: ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing, ACM Press, New York, NY, USA, 2005, pp. 253–262.
[3] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael Welcome, Katherine Yelick, An evaluation of current high-performance networks, in: IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, Washington, DC, USA, IEEE Computer Society, 2003, p. 28.1.
[4] Ahmad Faraj, Xin Yuan, Automatic generation and tuning of MPI collective communication routines, in: ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing, ACM Press, New York, NY, USA, 2005, pp. 393–402.
[5] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, Timothy S. Woodall, Open MPI: goals, concept, and design of a next generation MPI implementation, in: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004.
[6] Sergei Gorlatch, Send-receive considered harmful: myths and realities of message passing, ACM Trans. Program. Lang. Syst. 26 (1) (2004) 47–56.
[7] Peter Gottschling, Wolfgang E. Nagel, An efficient parallel linear solver with a cascadic conjugate gradient method, in: EuroPar 2000, LNCS, 1900, 2000.
[8] Wolfgang Hackbusch, Iterative Solution of Large Sparse Systems of Equations, Springer, 1994.
[9] M.R. Hestenes, E. Stiefel, Methods of conjugate gradients for solving linear systems, J. Res. Natl. Bur. Stand. 49 (1952) 409–436.
[10] T.Hoefler, T. Mehlan, F. Mietke, W, Rehm. Adding low-cost hardware barrier support to small commodity clusters, in: Proceedings of 19th International Conference on Architecture and Computing Systems – ARCS'06, vol. 3, 2006, pp. 343–250.
[11] T. Hoefler, J. Squyres, G. Bosilca, G. Fagg, A. Lumsdaine, W. Rehm. Non-Blocking Collective Operations for MPI-2. Technical report, Open Systems Lab, Indiana University, 08 2006.
[12] T. Hoefler, J. Squyres, W. Rehm, A. Lumsdaine, A case for non-blocking collective operations, in: Frontiers of High Performance Computing and Networking – ISPA 2006 Workshops, vol. 4331/2006, Springer, Berlin, Heidelberg, 2006, 12, pp. 155–164.
[13] Torsten Hoefler, Torsten Mehlan, Frank Mietke, Wolfgang Rehm, Fast barrier synchronization for InfiniBand, in: Proceedings, 20th International Parallel and Distributed Processing Symposium IPDPS 2006 (CAC 06), April 2006.
[14] L.V. Kale, Sameer Kumar, Krishnan Vardarajan, A framework for collective personalized communication, in: Proceedings of IPDPS'03, Nice, France, April 2003.
[15] Arkady Kanevsky, Anthony Skjellum, Anna Rounbehler, MPI/RT – an emerging standard for high-performance real-time systems, in: HICSS, vol. 3, 1998, pp. 157–166.
[16] G. Liu, T.S. Abdelrahman, Computation-communication overlap on network-of-workstation multiprocessors, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, July 1998, pp. 1635–1642.
[17] J. Liu, A. Mamidala, D. Panda, Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support, Technical report, OSU-CISRC-10/03-TR57, 2003.
[18] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. 1995.
[19] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1997.
[20] Message Passing Interface Forum. MPI-2 Journal of Development, July (1997).
[21] MPICH2 Developers. <http://www-unix.mcs.anl.gov/mpi/mpich2/>, 2006.
[22] Fabrizio Petrini, Darren J. Kerbyson, Scott Pakin, 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, ACM, Phoenix, AZ, USA, CD-Rom, 2003, pp. 55.
[23] Y. Saad, M.H. Schultz, GMRES: a generalized minimum residual algorithm for solving nonsymmetric linear systems, SIAM J. Sci. Statist. Comput. 7 (3) (1986) 856–869.
[24] Peter Sonnefeld, CGS, a fast Lanczos-type solver for nonsymmetric linear systems, SIAM J. Sci. Statist. Comput. 10 (1989) 36–52.
[25] Vinod Tipparaju, Jarek Nieplocha. Optimizing all-to-all collective communication by exploiting concurrency in modern networks, in: SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, Washington, DC, USA, 2005, pp. 46.
[26] Ulrich Trottenberg, Cornelis Oosterlee, Anton Schüller, Multigrid, Academic Press, 2000.
[27] Henk van der Vorst, Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM J. Sci. Statist. Comput. 13 (1992) 631–644.