Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI

Torsten Hoefler
Open Systems Laboratory
Indiana University
501 N. Morton Street
Bloomington, IN 47404 USA
htor@cs.indiana.edu

Andrew Lumsdaine
Open Systems Laboratory
Indiana University
501 N. Morton Street
Bloomington, IN 47404 USA
lums@cs.indiana.edu

Wolfgang Rehm
Dept. of Computer Science
Chemnitz University of Technology
Strasse der Nationen 62
Chemnitz, 09107 GERMANY
rehm@cs.tu-chemnitz.de

ABSTRACT

Collective operations and non-blocking point-to-point operations have always been part of MPI. Although non-blocking collective operations are an obvious extension to MPI, there have been no comprehensive studies of this functionality. In this paper we present LibNBC, a portable high-performance library for implementing non-blocking collective MPI communication operations. LibNBC provides non-blocking versions of all MPI collective operations, is layered on top of MPI-1, and is portable to nearly all parallel architectures. To measure the performance characteristics of our implementation, we also present a microbenchmark for measuring both latency and overlap of computation and communication. Experimental results demonstrate that the blocking performance of the collective operations in our library is comparable to that of collective operations in other high-performance MPI implementations. Our library introduces a very low overhead between the application and the underlying MPI and thus, in conjunction with the potential to overlap communication with computation, offers the potential for optimizing real-world applications.

Keywords

MPI, non-blocking communication, collective operations, non-blocking collective operations, overlap

1. INTRODUCTION

The Message Passing Interface (MPI) standard is a widely used programming interface for parallel high-performance computing that includes a wide variety of point-to-point and group communication operations. The blocking collective operations currently defined by MPI offer a high-level interface to the user, insulating the user from implementation details and giving MPI implementers the freedom to optimize their implementations for specific architectures. That is, although collective algorithms do not provide unique functionality *per se* (they can be implemented manually with basic point-to-point operations), collective operations provide important advantages in programmability, safety (with regards to programming errors) and performance.

In this respect, collective operations can be compared to BLAS [31] operations. For example a high-level BLAS matrix multiply (e.g., DGEMM) operation could be easily composed of three nested loops[1], but the vendor supplied DGEMM implementation, because of special machine optimized tuning (e.g., cache/register optimization), usually provides much better performance. The same principle is used for collective operations as these operations can be optimized for the communication subsystem of a specific machine. Thus, many research groups have provided machine-optimized implementations and have investigated the optimal and non-trivial implementation of collective algorithms for particular machine architectures (cf. [7, 13, 19, 34, 41, 43, 44]).

The performance portability benefits of collective operations have long been recognized and collective operations play an important role in many applications (cf. [40]). Consider for example a three-dimensional Fast Fourier Transformation implemented for a central-switch-based architecture (e.g., InfiniBand™). If the developer does not use the MPI_Alltoall function, a fully connected send pattern (literally an all-to-all) should deliver the best performance[2]. However, if this implementation were to be ported to torus-based systems (e.g., an IBM BlueGene), the performance of the send-pattern mentioned above would be much worse than a torus-optimized MPI_Alltoall on that machine. However, because the collective operation interface is architecture independent, using it can avoid this performance decrease transparently, i.e., without changes to the user application.

A second MPI feature that plays a significant role in parallel programming is non-blocking point-to-point communication. These operations potentially allow communication and computation to be overlapped and thus to leverage hardware parallelism. The parallelism exists because most high-performance interconnect networks (like InfiniBand™ [42], Quadrics [39], Myrinet [38], Portals, or Ethernet with TOE) have their own communication co-processors that take the

SC07 November 10-16, 2007, Reno, Nevada, USA

[1] or lower level BLAS operations

[2] in fact, most MPI implementations use this communication pattern to implement MPI_Alltoall on central-switch-based architectures

burden of message processing of the main CPU. However, this parallelism does not decrease the latency significantly, and it does not show its full potential if the programmer uses blocking send/receive. Non-blocking send/receive techniques allow the programmer to leverage the CPU during the asynchronous (and network-offloaded) message transmission. Several studies showed that the performance of parallel applications can be significantly enhanced with overlapping techniques (cf. [1, 3, 6, 9, 33]).

Our work investigates the possibility of combining the advantages of collective operations with overlapped communication and computation in modern communication architectures. We propose a low-overhead and portable implementation of non-blocking collective operations that hides all the complexity of the internal implementation from the user. A new benchmark that measures the possible overlap of collective operations and overlap-optimized application kernels demonstrate the potential of our implementation. The following section will introduce the idea of non-blocking collective operations. Section 3 describes our portable implementation, followed by a microbenchmark that assess the possible overlap and evaluates our implementation and the underlying layers in Section 4. Application kernels are discussed in Section 5. Final conclusions and an outlook to future work in this field are presented in the last section.

2. NON-BLOCKING COLLECTIVE OPERATIONS

Non-blocking collective operations to overlap communication and computation are not directly supported by the MPI standard. The approach (long accepted by the conventional wisdom) to emulating this functionality is to perform the blocking collective operation in a separate thread on the a duplicated communicator. Unfortunately, this approach has not been proven to be usable in practice. First, this approach would require an MPI-2 implementation that offers full MPI_THREAD_MULTIPLE (cp. [15]) support without a "big lock" (the use of which would limit performance significantly). There have been few, if any, such implementations widely available. Second, the administration of threads is a complex task for programmers (particularly with languages like Fortran that were not intended for systems programming) and would consist of significant amounts of "boilerplate" code. This approach is contrary to the philosophy of MPI which is to isolate the user from these kinds of complexities. Third, the needed communicator duplication, being a collective operation in itself, can also be very expensive. Finally, we remark that the conventional wisdom seems to be unevenly applied. That is, absent the above objections (e.g., even if the administration of threads could be simplified), one could make the argument that non-blocking point-to-point operations could also be provided using blocking operations plus threads. Yet, MPI provides a direct interface to non-blocking point-to-point operations.

These complications have led different groups to the conclusion that a non-blocking interface for collective operations would be useful. The MPI Journal of Development [36] defines so called "split collectives" that enable overlapping in a very limited way. Another interface was proposed by IBM for their parallel environment [24]. Some research groups experimented with non-blocking collective operations based on non-blocking point-to-point messages (e.g., [12, 28]). MPI/RT [29] also offers a non-blocking interface to collective operations. But due to the channel semantics, this interface is different from the MPI interface and can not be easily adopted to existing programs. Additionally, to the best knowledge of the authors, neither performance analyses nor implementation details about overlappable collective operations in MPI/RT have been published.

```
1  MPI_Request req;
   int arr[100];

   MPI_Ibcast(arr, 100, MPI_INT, 0,
5              MPI_COMM_WORLD, &req);
   /* do independent computation */
   MPI_Wait(&req);
```

Listing 1: Non-blocking Interface to Collective Routines

Using key ideas from these approaches we defined a standard proposal for non-blocking collective operations in [20]. For example, an MPI_Ibcast is nearly identical to its blocking variant MPI_Bcast. Listing 1 shows an example code for a non-blocking broadcast.

Non-blocking collectives offer the possibility of overlapping communication and computation for collective operations while additionally mitigating the effects of pseudo-synchronization. Pseudo-synchronization occurs for many collective algorithms due to data-dependencies in the communication pattern (receivers have to wait for senders). This turns out to be a problem on large-scale machines when the process skew between processing units becomes significant [4, 27]. Non-blocking collective operations can move the pseudo-synchronization to the background and allow the user application to tolerate process skew to a certain extent. A detailed discussion of pseudo-synchronization and its effect on parallel program runs is given in [20, 21].

All those effects can benefit real-world applications. Preliminary results show that non-blocking collectives are also able to decrease the latency and bandwidth sensitivity of some applications. A study with a 3d-Poisson solver [17] shows that Gigabit Ethernet is able to deliver nearly the same performance as InfiniBand™ because the algorithm supports full overlap.

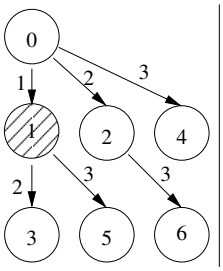
3. IMPLEMENTATION IN LIBNBC

LibNBC is written in ANSI C and is a portable implementation of our non-blocking collective proposal on top of MPI-1 [18]. All currently implemented algorithms are optimized for central-switch based networks and extensively tested with InfiniBand™ connected systems. The full implementation is open source and available for public download on the LibNBC website [32]. The central part of LibNBC is the collective schedule. A schedule consists of the operations that have to be performed to accomplish the collective task (e.g., an MPI_Isend, MPI_Irecv). The collective algorithm is implemented as in the blocking case, based on point-to-point messages. However, instead of performing all operations immediately, they are saved in the collective schedule together with their dependencies. New algorithms can readily be incorporated into LibNBC as the internal API to do so is nearly identical to MPI. A detailed manual about the

addition of new collective algorithms and the internal and external programming interfaces of LibNBC is available in [18].

Data-dependencies in many collective algorithms (e.g., an intermediate rank in a broadcast tree can not pass the data on until it receives it) make it necessary to provide some synchronization mechanism. Our collective schedule introduces a round-based scheme, where operations of round n can only be started if **all** operations of round $n-1$ have been finished locally. The round based scheme is applicable to all collective algorithms because the simplest schedule, where all rounds consist of a single operation, exists for all algorithms (equals to a blocking execution). However, the ability to issue more than one operation per round provides an additional level of parallelism and benefits the non-blocking execution (i.e., it adds the ability to overlap communication with communication as described by Bell et al. in [5]).

Each schedule is specific to each rank of a communicator and depends on the MPI-argument set. A particular example for the creation of a binomial tree-based MPI_Bcast schedule is shown in Figure 1. The left side if Figure 1 shows the broadcast tree with 7 processes where the vertices in the graphs denote the ranks in the communicator; the edges stand for the send operations. The edge-numbers indicate the virtual communication round in which the send occurs. The pseudocode to create the schedule is shown in the right side of Figure 1. The schedule for rank 1 consists of three rounds. The first round contains only a single receive (NBC_Sched_recv which represents an MPI_Irecv) and is terminated with a call to NBC_Sched_barr() (which represents an MPI_Waitall on all open requests). This means that the second round is not started before the message is received (necessary for correctness of the broadcast). The following two rounds could be merged into a single round with two sends (NBC_Sched_send which represents an MPI_Isend), but we begin the second send after waiting for the first one to finish to avoid network congestion. Note that a schedule could be reused if the same collective operation with the same arguments is issued by the user again.




```
Pseudocode for schedule at rank 1:
NBC_Sched_recv(buf, count, dt, 0, schedule);
NBC_Sched_barr(schedule);
NBC_Sched_send(buf, count, dt, 3, schedule);
NBC_Sched_barr(schedule);
NBC_Sched_send(buf, count, dt, 5, schedule);
```

Figure 1: Example of a schedule creation of rank 1 in a communicator with 7 processes.

To ensure cache-efficiency, a schedule is stored as a linear array in memory. Its building blocks are C structs that represent the arguments of an operation. A send operation structure, for example, accommodates buffer pointer (void*), count (int), datatype (MPI_Datatype) and destination rank (int), which equals a total of 32 bytes on a typical 64 bit machine. The biggest struct, representing the execution of an MPI_Op, is 48 bytes. This design makes the schedule-execution extremely cache-friendly and keeps the memory overhead low. However, the schedule creation can be expensive because of the need to use dynamically sized contiguous arrays.

recv from 0	end	send to 3	end	send to 5

Figure 2: Schedule layout in memory.

We introduced a low overhead implementation technique for non-blocking collective operations. The next section presents data from a benchmark we designed to document the benefits inherent in overlapping communication and computation.

4. MEASURING THE POSSIBLE OVERLAP

We designed a microbenchmark, named NBCBench, to measure the performance of all blocking and non-blocking collective operations. The benchmark source code is available at the LibNBC website [32]. One important difference of our design, compared to established benchmarks such as the Intel (formerly Pallas) Microbenchmarks (IMB), is that we benchmark the exact time for a single collective operation on each node. We do not measure pipelining effects as most benchmarks do by issuing N collective operations in a row and dividing the measured time by N. Measuring single operations can often deliver interesting results that differ significantly from pipelined transfers (cf. [22]). Single message measurements require a high precision timer that is typically offered as a platform dependent CPU instruction and a low process skew. Our implementation uses the X86 RDTSC [26] instruction to measure times with the accuracy of CPU-ticks. However, the timing routine can be easily exchanged with any other high-precision timer (e.g., MPI_Wtime). A low inter-process skew is guaranteed by a call to MPI_Barrier or an internal Dissemination Barrier (cf. [16]) implementation, which limits the inter-process skew to a single latency, before every measurement.

4.1 Benchmarking MPI collective operations

The use of high-precision timers enables us to accurately measure the node-local time of every operation. As a result, every node measures the execution time of its part of the collective operation. The local time to perform a collective operation can be split up into two parts. The first part is the synchronization time, i.e., the time a node waits until it receives its first data element. This time depends on the algorithm[3] and on the parallel skew among the processes (e.g., if one process is delayed). The second part is the time during which the nodes perform the actual algorithm. A benchmark is usually only able to measure the sum of these parts. However, our benchmark minimizes the process skew by synchronizing all nodes with a barrier call before every measurement. The user is free to select between MPI_Barrier or an internal dissemination barrier implementation based on MPI_Send/MPI_Recv that guarantees the skew will be no more than a single point-to-point latency (cf. dissemination barrier described in [16]) .

The benchmark performs a user-defined number of operations (default: 50) for every communicator size and message

[3]Nota bene: the time is zero for some operations or nodes, e.g., all nodes in MPI_Alltoall, or the root node in MPI_Bcast.

```
MPI_Barrier(comm) or Internal_Barrier(comm);
start_time = take_time();
MPI_Bcast(buf, count, type, root, comm);
total_time = take_time() - start_time;
```

Listing 2: Pseudocode for NBCBench to measure (example MPI_Bcast)

size (also user-definable). Listing 2 shows pseudo-code for the part of the inner loop that measures the MPI performance of the blocking collective operations.

4.2 Benchmarking non-blocking collective operations

The goal of the microbenchmark introduced in this article is to study the overlap potential and the remaining non-overlappable overhead of non-blocking collective operations. In order to study and discuss these issues we first make the distinction between network *l*atency and CPU *o*verhead (cf. LogP model in [11]). Latency is the time to perform the collective communication operation and overhead is the CPU time that is needed in this process. In the case of blocking operations, the overhead is larger than the latency because the function may only return after the communication has finished. Non-blocking operations allow the user to overlap parts of the latency with independent computation (cf. Section 1) while the overhead still accounts towards the parallel running time. As long as independent computation can proceed, the latency is not significant for the running time of the parallel computation because it is overlapped. However, the overhead, because it blocks the CPU, still increases the parallel execution time. In practice, the overhead denotes the time that is spent in communication functions and in the network stack of the operating system. We assume that the full communication latency can potentially be overlapped by the user application[4].

The inner benchmark loop is similar to the loop explained in the previous section. However, the time to issue a non-blocking operation in a blocking manner without any overlap (calling the initiation function and the finish function without any delay between them) is benchmarked before the actual run. The measurement, presented as pseudo-code in Listing 3, begins, like in the blocking case, with a synchronization to avoid process skews larger than a single point-to-point latency. The next operation issues a non-blocking communication operation and measure the time of this function call that is added to the overhead later. The second operation simulates the independent computation that runs at least as long as the blocking latency measured before the actual benchmark. This computation issues several NBC_Test calls to progress LibNBC internally[5] (progression strategies are discussed later). The time to perform the test calls is also added to the overhead. The last operation, NBC_Wait, finishes the communication and is ideally a no-op (i.e., the

```
MPI_Barrier(comm) or Internal_Barrier(comm);
start_time = take_time();
NBC_Ibcast(buf, count, type, root, comm,
             handle);
init_time = take_time() - start_time;
test_time = do_computation_test(duration);
before_wait = take_time();
NBC_Wait(handle)
wait_time = take_time() - before_wait;
total_time = take_time() - start_time;
```

Listing 3: Pseudocode for NBCBench (example MPI_Bcast)

communication has already been finished). The time to perform this call is also measured seperately and added to the overhead.

The results gathered with this benchmark can be considered a bound on the minimum possible overhead and the maximum possible overlap.

The measurement results in a distributed two-dimensional matrix of values. Every node has its own times for the multiple measurements. The reduction of those node-specific results can be chosen by a command-line parameter (minimum, maximum, average, median); the median will be used for all results of this paper. The final output can either be the reduced times of all nodes, which gets messy if the node-number is reasonably high but it is useful to analyze process skew introduced by the collective algorithm itself, or the maximum time of all nodes. We chose the maximum, because this will usually determine the parallel running time of a load-balanced parallel application.

The possible overlap of LibNBC is highly MPI implementation and network dependent because non-blocking MPI send/receive calls are used to perform the operations. Several studies and benchmarks [23, 25, 30] assess the possible overlap of non-blocking point-to-point operations for different MPI implementations and show different results. Our benchmark is able to measure the overlap potential and overhead of blocking and non-blocking point-to-point as well as blocking and non-blocking collective operations.

4.3 Experimental Setup

We conducted all benchmarks on the odin cluster system at Indiana University. This system consists of 128 dual Opteron 270 dual core nodes with 4 GB RAM. The nodes are interconnected with 10 GBit/s InfiniBand™ and Gigabit Ethernet.

We used MVAPICH 0.9.4 [35] for the communication over InfiniBand™ and Open MPI 1.1 [14] for the communication over InfiniBand™ and Gigabit Ethernet (TCP). We used LibNBC version 0.9 available at [32].

4.4 Benchmark Results

We investigated two MPI implementations, MVAPICH and Open MPI, that are able to use InfiniBand™ to communicate. The left graph in Figure 3 shows the relation between communicated datasize and the overhead for a non-blocking all-gather on 64 nodes for LibNBC (`NBC`) and the native MPI implementation (`MPI`) running with Open MPI and MVAPICH on InfiniBand™. The share of the over-

[4]If the time of the independent communication and the overhead is known, it is trivial to calculate the potential overlap for real applications, even if the full-overlap criteria is not met.

[5]This is not necessary to ensure the non-blocking semantics but improves the overlap, similar to calling MPI_Test on outstanding MPI_Requests.

lappable latency (compared to a blocking execution) using LibNBC with the both MPI implementations is presented in the right graph in Figure 3. Both graphs show that the performance as well as the overlap depends heavily on the used MPI library. A blocking execution of LibNBC, i.e., NBC_Iallgather immediately followed by NBC_Wait, delivers similar performance to the implementation of MPI_Allgather in MVAPICH because it is based on a similar point-to-point communication pattern.

Figure 4 shows comparative results between fully overlapped operations with LibNBC and blocking operations with MVAPICH, which offers currently the fastest implementation of collective algorithms on InfiniBand™. Benchmark results for all collective operations of Open MPI with TCP and InfiniBand™, MVAPICH, and MPICH2 can be found on [32].

These results show that the advantage of non-blocking collective operations can be quite substantial on InfiniBand™ cluster systems. This assumption can theoretically be extended to all offloading-based interconnects. However, the overlap potential for small messages is naturally much lower than for large messages. This is mostly due to non-overlappable overhead, which we discuss in the next section. It can also be seen that some algorithms, as for example MPI_Bcast, require further tuning to support better overlap. This is subject of future research and can be done with the collective schedule framework of LibNBC.

4.5 Sources of Overhead

To understand the results gathered by the microbenchmark, we discuss the sources of overhead and the overhead scaling with the communicator size. This overhead is the part of the latency of a non-blocking collective operation that is executed by the main CPU and can thus not be overlapped. This makes the overhead-reduction one of the most important goals in implementing a library that offers non-blocking operations. The library can of course only use the CPU when it gets called and overhead is spent in the initialization (e.g., NBC_Ibcast), test and wait functions. Furthermore, the sum of time spent in those functions is the communication overhead.

The different parts of the overhead, named in the previous sections, depend on many factors. All those partial costs scale differently with communicator size and message size and add up to the final overhead of LibNBC. The different sources of non-overlappable overhead are:

- Schedule creation (costs to create the node-local schedule). The schedule creation cost does not depend on the message size but grows with the communicator size.

- Copy overhead (costs for the required local copy operation from sendbuffer to receivebuffer, e.g., MPI_Alltoall). The local copy overhead depends only on the message size because the problem is massively parallel.

- MPI_Isend/MPI_Irecv overhead (overhead of the MPI_Isend/MPI_Irecv in the MPI library). The cost of issuing MPI_Isends and MPI_Irecvs is determined by both factors. The number of issued operations can depend on the communicator size and data size (if segmenting and pipelining is used).

- MPI_Test/MPI_Wait overhead (costs to progress/finish these operations in the MPI library). The time to issue such an operation could depend on the datasize. This is especially important for InfiniBand™ where the registration of send and receive memory can be expensive [37].

4.6 Different Progress Types

The current implementation of LibNBC does not offer true asynchronous progress. The addition of a progress-thread would be simple but would require the underlying MPI implementation to be thread-safe (support of MPI_THREAD_MULTIPLE). Most current MPI implementations do not offer this feature in a well performing way. Thus, we did not use threads to keep LibNBC portable to many parallel architectures and MPI libraries.

Because round-transitions in collective schedules can only be done by LibNBC itself, it should be called periodically to ensure the best overlap potential. NBC_Test tries to progress all outstanding operations and starts a new round if the currently active one is finished. Besides LibNBC, the MPI library also needs to progress outstanding message transfers. Without a separate progress thread, there is only one way to progress an outstanding MPI operation. This is done by giving the control to the MPI library by calling some MPI function (e.g., MPI_Test). Most MPI implementations (e.g. Open MPI, MVAPICH and MPICH) try to finish active requests internally. NBC_Test calls MPI_Testall on all MPI requests, so that NBC and MPI progress can be guaranteed. NBC_Wait calls NBC_Test until the collective operation has finished.

However, calling test functions introduces a constant overhead (at least the function call and the status-check of all requests). On one hand, progressing outstanding operations with tests may accelerate the operation in the background, but the constant overhead may decrease the overall performance in case of many tests/requests.

The number of necessary tests depends mainly on the length of outstanding requests because each NBC_Test tests all outstanding operations to all hosts. The majority of the overhead in NBC_Test is caused by the internal call to MPI_Testall that is in itself highly implementation dependent. It can be assumed that the cost for NBC_Test grows linearly with the number of outstanding requests.

The overhead for a single NBC_Test for increasing communicator sizes on top of Open MPI and MVAPICH is shown in Figure 5. The test-frequency was chosen proportional to the data-size in this example; a NBC_Test is issued for each 2048 bytes sent/received.

5. APPLICATION RESULTS

We tested the effectiveness of overlapping communication and computation in three situations. In the compression application, we analyze data-gather in a pipelined fashion. We applied similar pipelining techniques to a three-dimensional Fourier Transformation and we optimized a conjugate gradient solver with non-blocking collective operations.

5.1 Parallel Compression

Compression is widely used in the gathering and analysis of scientific data. The tremendous amount of data and computationally demanding compression algorithms (e.g., bzip2 [8]) force scientists to parallelize the computation among

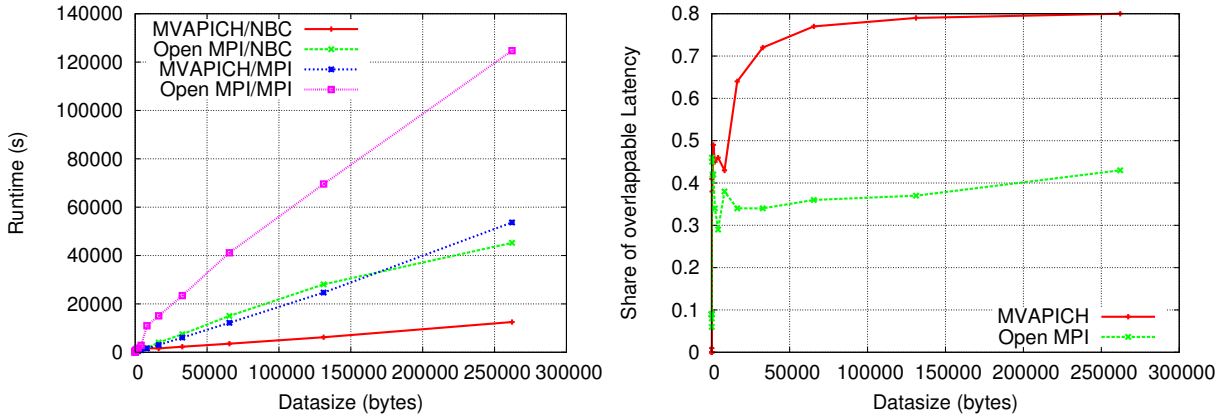


Figure 3: Left: NBC_Iallgather and MPI_Allgather overhead on Open MPI 1.1 and MVAPICH 0.9.4 on 64 InfiniBand™-connected nodes, Right: share of the overlappable latency

multiple processing units. This parallelization requires two communication operations, the distribution of the input and a gathering of the compressed data. Both operations are ideally implemented as collective communications. We analyze only the data gathering and apply a simple pipeline scheme to overlap communication and computation for this problem. We gather the data with MPI_Gather for the original blocking base-case. The non-blocking variant communicates the first block of compressed data as soon as it is produced with NBC_Igather in a non-blocking manner and pipelines the communication of the remaining blocks..

Figure 6 shows the parallel speedup for a fixed problem size of 15.25 MiB random data (the compression is very compute-intensive, i.e., a single processor needs 2:40 min to compress the data). All measurements have been conducted on the odin cluster system with MVAPICH using InfiniBand™and Open MPI using TCP over Gigabit Ethernet. Different parameters as block size and the number of outstanding communications have been adjusted to achieve the best possible result (this tuning process is outside the scope of this article). The use of non-blocking collective operations increases the parallel speedup substantially for both interconnects (the performance gain on InfiniBand™ is naturally smaller because the communication latency is much lower than for Gigabit Ethernet).

5.2 Parallel Fourier Transformation

The same pipelining technique can be applied to a parallel three-dimensional Fourier transform. We chose a well known scheme [2, 10] as the base case to apply our transformations. This scheme transforms the data in the x and y directions, performs a parallel transpose (MPI_Alltoall) and finishes the transformation in the z direction. We applied non-blocking collective operations in a pipelined manner to the transposition. The communication of a z-plane can be started as soon as this plane is transformed in the x and y directions (and runs concurrently to the transformation of the remaining planes). However, we also applied techniques that coalesces multiple planes to reach a reasonable communication data size. Figure 7 shows the results for InfiniBand™. The results show a substantial performance improvement for the parallel Fourier transform.

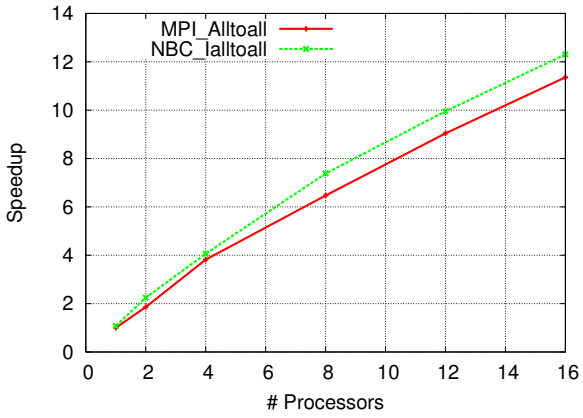


Figure 7: Speedup of a parallel 3d-FFT using InfiniBand™

5.3 Parallel Conjugate Gradient Solver

We showed in a former study [17] that non-blocking collective operations can easily be applied to problems that exhibit some kind of data parallelism (i.e., independent computation is available). Our example application uses a conjugate gradient method to solve a three-dimensional poisson equation. The problem is distributed on a 3d-domain where the communication of the halo zones is naturally overlapped with the computation of the inner matrix elements. The changes to the original algorithm are trivial and the performance benefits are substantial. The communication uses MPI_Alltoallv (in lack of a collective communication operation for nearest-neighbor communication). The performance results ars shown in Figure 8.

6. CONCLUSIONS AND FUTURE WORK

Traditionally, scientists have not been able to fully leverage the performance benefit of overlapping communication and computation because MPI only supports blocking collective operations. In this paper, we introduce LibNBC, a portable high-performance library that provides non-blocking

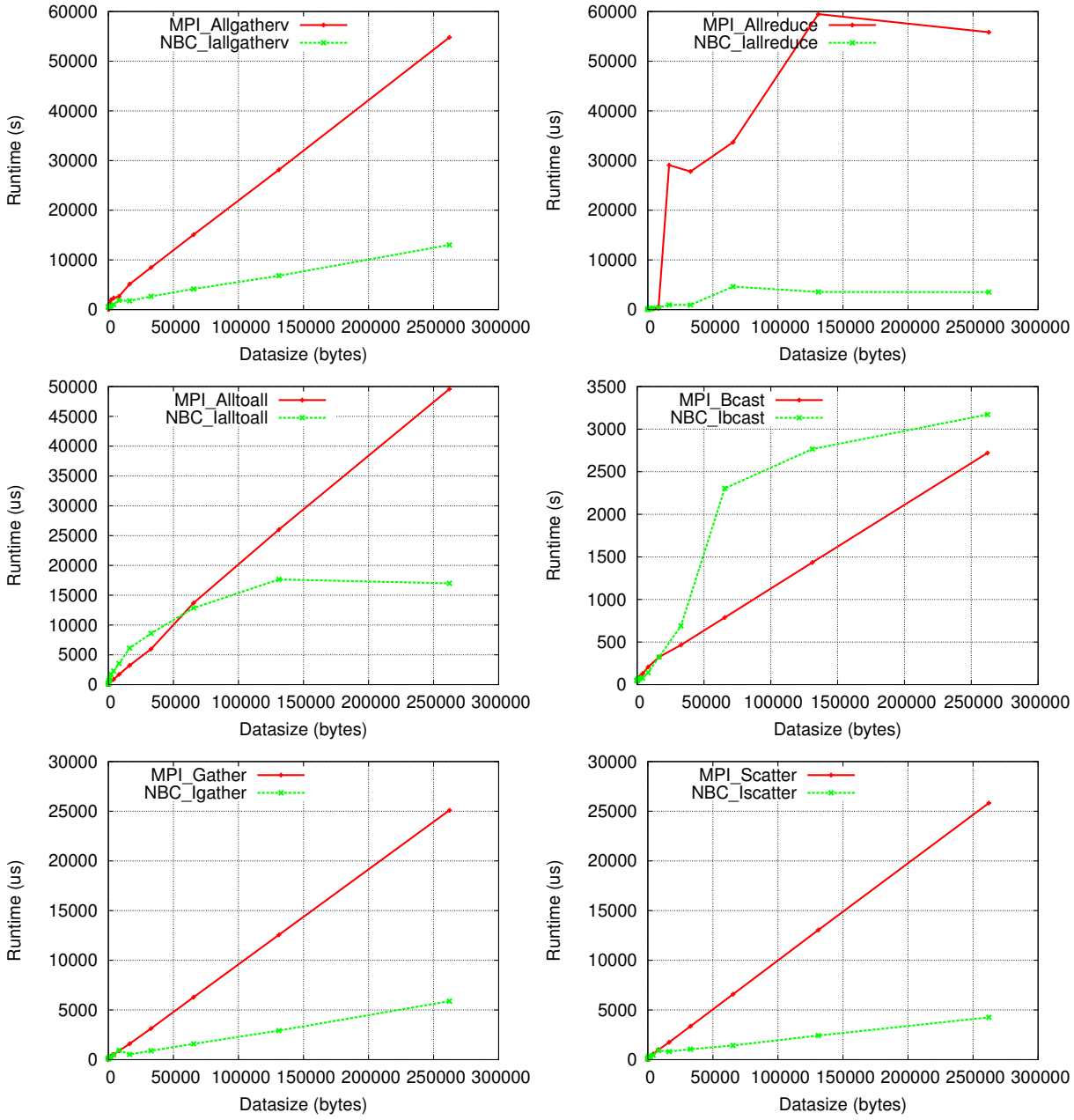


Figure 4: Overhead comparison between blocking (e.g., MPI_Allgatherv) and non-blocking (e.g., NBC_Iallgatherv) implementations with overlap for different collective operations on top of MVAPICH

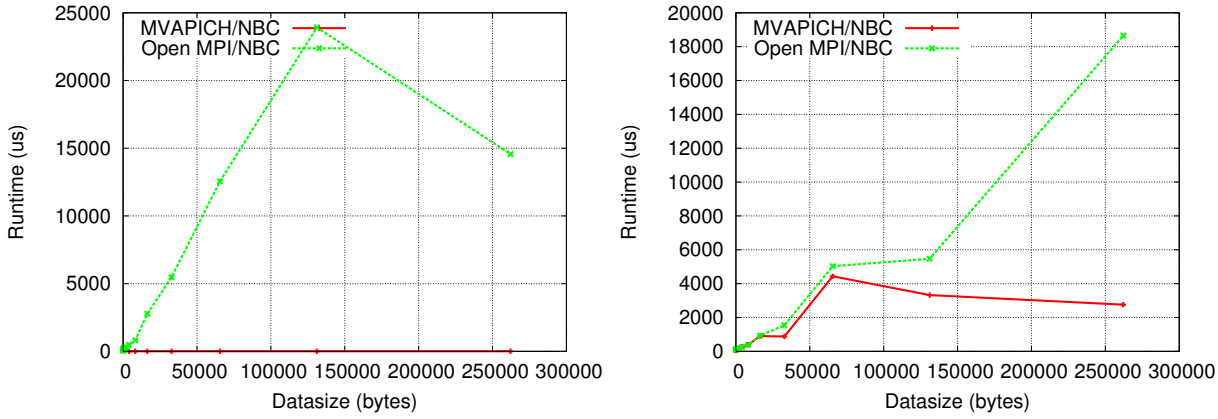


Figure 5: NBC_Test **overhead for** NBC_Iallgather **(left) and** NBC_Iallreduce **(right) for Open MPI and MVAPICH on 64 InfiniBand™ nodes.**

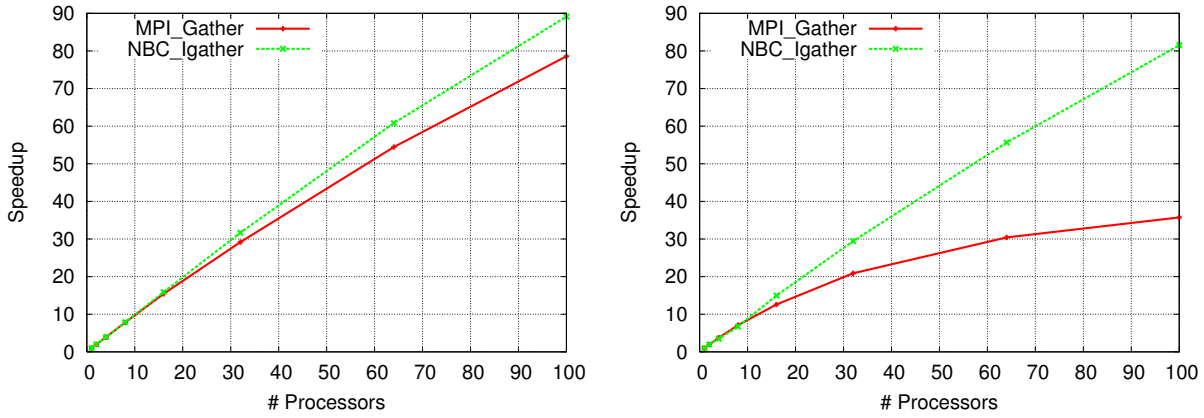


Figure 6: Speedup of a parallel compression using InfiniBand™(left) and Gigabit Ethernet (right).

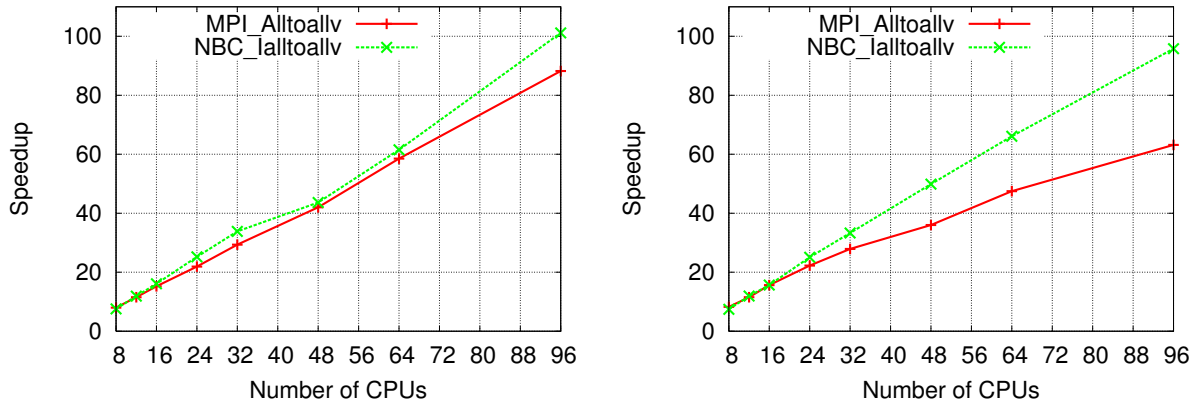


Figure 8: Speedup of a parallel conjugate gradient using InfiniBand™(left) and Gigabit Ethernet (right).

versions of all collective operations. Thus, for the first time, fully portable and MPI compliant non-blocking collective operations are available for overlapping communication and computation.

We further designed a microbenchmark to measure the performance of all blocking and non-blocking operations. Overlapping communication and computation using non-blocking collective operations has been proven to be useful. We showed that the communication of large messages on large communicators can be efficiently overlapped on InfiniBand™ systems. Smaller messages realize a lesser gain due to different overheads. Application results also demonstrate the benefits of our implementation.

One promising direction for future research is the offloading of the progress engine to another processing unit, such as an intelligent network interface, network co-processor or even a dedicated core on a multi-core CPU. Furthermore, we will develop a parallel model to assess running time and overlap potential of our implementation and non-blocking collectives in general. The optimization of the implemented algorithms in LibNBC, to ensure highest parallelism and overlap, will also be a direction of future research.

The NBC-Library is available at:
`http://www.unixer.de/NBC`

Acknowledgements

Pro Siobhan. The authors want to thank Laura Hopkins from Indiana University for editorial comments and helpful discussions. This work was supported by a grant from the Lilly Endowment, National Science Foundation grant EIA-0202048 and a gift the Silicon Valley Community Foundation on behalf of the Cisco Collaborative Research Initiative and a grant by the Saxon Ministry of Science and the Fine Arts.

7. REFERENCES

[1] T. S. Abdelrahman and G. Liu. Overlap of computation and communication on shared-memory networks-of-workstations. *Cluster computing*, pages 35–45, 2001.

[2] A. Adelmann, W. P. P. A. Bonelli and, and C. W. Ueberhuber. Communication efficiency of parallel 3d ffts. In *High Performance Computing for Computational Science - VECPAR 2004, 6th International Conference, Valencia, Spain, June 28-30, 2004, Revised Selected and Invited Papers*, volume 3402 of *Lecture Notes in Computer Science*, pages 901–907. Springer, 2004.

[3] F. Baude, D. Caromel, N. Furmento, and D. Sagnol. Optimizing metacomputing with communication-computation overlap. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, pages 190–204, London, UK, 2001. Springer-Verlag.

[4] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. Operating system issues for petascale systems. *SIGOPS Operating System Review*, 40(2):29–33, 2006.

[5] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *Proceedings, 20th International Parallel and Distributed Processing Symposium IPDPS 2006 (CAC 06)*, April 2006.

[6] R. Brightwell, R. Riesen, and K. D. Underwood. Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. *Int. J. High Perform. Comput. Appl.*, 19(2):103–117, 2005.

[7] E. D. Brooks. The Butterfly Barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[8] BZIP2. http://www.bzip.org, 2006.

[9] P.-Y. Calland, J. Dongarra, and Y. Robert. Tiling on systems with communication/computation overlap. *Concurrency - Practice and Experience*, 11(3):139–153, 1999.

[10] C. E. Cramer and J. A. Board. The development and integration of a distributed 3d fft for a cluster of workstations. In *Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta*, volume 4. USENIX Association, 2000.

[11] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[12] A. Dubey and D. Tessera. Redistribution strategies for portable parallel FFT: a case study. *Concurrency and Computation: Practice and Experience*, 13(3):209–220, 2001.

[13] L. A. Estefanel and G. Mounie. Fast Tuning of Intra-Cluster Collective Communications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users Group Meeting Budapest, Hungary, September 19 - 22, 2004. Proceedings*, 2004.

[14] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. 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.

[15] W. D. Gropp and R. Thakur. Issues in developing a thread-safe mpi implementation. In B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Proceedings*, volume 4192 of *Lecture Notes in Computer Science*, pages 12–21. Springer, 2006.

[16] T. Hoefler, L. Cerquetti, T. Mehlan, F. Mietke, and W. Rehm. A practical Approach to the Rating of Barrier Algorithms using the LogP Model and Open MPI. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPP'05)*, pages 562–569, June 2005.

[17] T. Hoefler, P. Gottschling, W. Rehm, and A. Lumsdaine. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. In *Recent Advantages in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI User's Group Meeting, Proceedings, LNCS 4192*, pages 374–382. Springer, 9 2006.

[18] T. Hoefler and A. Lumsdaine. Design, Implementation,

and Usage of LibNBC. Technical report, Open Systems Lab, Indiana University, 08 2006.

[19] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. Fast Barrier Synchronization for InfiniBand. In *Proceedings, 20th International Parallel and Distributed Processing Symposium IPDPS 2006 (CAC 06)*, April 2006.

[20] T. Hoefler, J. Squyres, G. Bosilca, G. Fagg, A. Lumsdaine, and W. Rehm. Non-Blocking Collective Operations for MPI-2. Technical report, Open Systems Lab, Indiana University, 08 2006.

[21] T. Hoefler, J. Squyres, W. Rehm, and A. Lumsdaine. A Case for Non-Blocking Collective Operations. In *Frontiers of High Performance Computing and Networking - ISPA 2006 Workshops*, volume 4331/2006, pages 155–164. Springer Berlin / Heidelberg, 12 2006.

[22] T. Hoefler, C. Viertel, T. Mehlan, F. Mietke, and W. Rehm. Assessing Single-Message and Multi-Node Communication Performance of InfiniBand. In *Proceedings of IEEE Inernational Conference on Parallel Computing in Electrical Engineering, PARELEC 2006*, pages 227–232. IEEE Computer Society, 9 2006.

[23] C. Iancu, P. Husbands, and P. Hargrove. Hunting the overlap. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 279–290, Washington, DC, USA, 2005. IEEE Computer Society.

[24] IBM. *IBM Parallel Environment for AIX, MPI Subroutine Reference*, 1993. http://publibfp.boulder.ibm.com/epubs/pdf/a2274230.pdf.

[25] J. W. III and S. Bova. Where's the Overlap? - An Analysis of Popular MPI Implementations, 1999.

[26] Intel Corporation. Intel Application Notes - Using the RDTSC Instruction for Performance Monitoring. Technical report, Intel, 1997.

[27] K. Iskra, P. Beckman, K. Yoshii, and S. Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. In *Proceedings of Cluster Computing, 2006 IEEE International Conference*, 2006.

[28] L. V. Kale, S. Kumar, and K. Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

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

[30] W. Lawry, C. Wilson, A. B. Maccabe, and R. Brightwell. Comb: A portable benchmark suite for assessing mpi overlap. In *2002 IEEE International Conference on Cluster Computing (CLUSTER 2002), 23-26 September 2002, Chicago, IL, USA*, pages 472–475. IEEE Computer Society, 2002.

[31] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. In *In ACM Trans. Math. Soft., 5 (1979), pp. 308-323*, 1979.

[32] LibNBC. http://www.unixer.de/NBC, 2006.

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

[34] J. Liu, A. Mamidala, and D. Panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support. Technical report, OSU-CISRC-10/03-TR57, 2003.

[35] J. Liu, J. Wu, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *Int'l Journal of Parallel Programming, 2004*, 2004.

[36] Message Passing Interface Forum. MPI-2 Journal of Development, July 1997.

[37] F. Mietke, R. Baumgartl, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. 8 2006. Accepted for publication at Euro-Par 2006 Conference.

[38] Myrinet. http://www.myrinet.com, 2006.

[39] Quadrics. http://www.quadrics.com, 2006.

[40] R. Rabenseifner. Automatic MPI Counter Profiling. In *42nd CUG Conference*, 2000.

[41] M. L. Scott and J. M. Mellor-Crummey. Fast, contention-free combining tree barriers for shared-memory multiprocessors. *Int. J. Parallel Program.*, 22(4):449–481, 1994.

[42] M. Technologies. Infiniband - industry standard data center fabric is ready for prime time. *Mellanox White Papers*, December 2005.

[43] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[44] W. Yu, D. Buntinas, R. L. Graham, and D. K. Panda. Efficient and scalable barrier over quadrics and myrinet with a new nic-based collective message passing protocol. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*, 2004.