Accurately Measuring Overhead, Communication Time and Progression of Blocking and Nonblocking Collective Operations at Massive Scale

Torsten Hoefler, Timo Schneider and Andrew Lumsdaine

Open Systems Laboratory, Indiana University
501 N. Morton Street, Bloomington, IN 47404 USA
{htor,timoschn,lums}@cs.indiana.edu

Abstract

Accurate, reproducible and comparable measurement of the overheads, communication times and progression behavior of blocking and nonblocking collective operations is a complicated task. Although Different measurement schemes for blocking collective operations are implemented in well-known benchmarks, many of these schemes introduce different systematic errors in their measurements. We characterize these errors and select a window-based approach as the most accurate method. However, this approach complicates measurements significantly and introduces clock synchronization as a new source of errors. We analyze approaches to avoid or correct those errors and develop a scalable synchronization scheme to conduct benchmarks on massively parallel systems. Our results are compared to the window-based scheme implemented in the SKaMPI benchmarks and show a reduction of the synchronization overhead by a factor of 16 on 128 processes. We also describe two different measurement schemes for the overhead and asynchronous progress of nonblocking collective communications. An implementation and results of both measurement schemes are presented.

Keywords: benchmarking, collective operations, MPI, time synchronization, scalable synchronization

1 Introduction

Collective operations, i.e., operations that are defined on a group of processes rather than a single process, are an important part of parallel scientific computing. Their advantages comprise the separation of communication and computation which enables machine specific communication optimization, performance portability among different parallel machines, better programmability and the reduction of implementation errors. All those benefits have been recognized by the scientific community and collective operations play an important role in many parallel applications (cf. [23]). Thus, the performance, i.e., latency, of the collective operations is crucial for the performance and running time of numerous applications.

A new class of collective operation, nonblocking collective operations [11], begins to gain importance. Those operations change the parameters slightly. While low latency is the most important feature of blocking collective operations, programs using nonblocking collectives efficiently can often deal with higher latencies if there is sufficient computation that can be performed in parallel. To utilize the parallel execution of collective communication and computation, the communication has to proceed independently from the main CPU (asynchronous). This is achieved by decoupling the communication in the network from the computation on the host. New optimization criteria for nonblocking collective operations are low CPU overhead and high asynchronous progress [10]. We also discuss benchmarking schemes to assess overhead and independent progress of nonblocking collective operations.

The complex interaction between communication and computation in real-world applications requires models to understand and optimize parallel codes. Serial computation models, such as Modeling Assertions [2], have to be combined with network models such as the LogP [4]. Unfortunately, today's systems are too complex to be described entirely by an execution or communication model. It is necessary to assess the model parameters for every real-world system, in the serial execution case in [2] as well as in the communication case in [5, 8, 14]. Pjesivac-Grbovic et al. showed in [22] that the latency of collective operations that are implemented on top of point-to-point messages can be modeled with the LogP model family and we showed in an earlier work [7] that it is also possible to predict application performance by modeling the communication and computa-

```
MPI_Gather(...); /* warmup */
MPI_Barrier(...); /* synchronization */
t0 = MPI_Wtime(); /* take time */
for (i=0; i<reps; i++) {
        MPI_Gather(...); /* execute benchmark */
}
t1 = MPI_Wtime(); /* take time */
MPI_Barrier(...);
time = t1−t0;
```

Listing 1. MPPTEST Benchmark Scheme

```
TIMER_START; /* take time */
for (i=0; i<cnt; i++) {
    if (...) flushall(1); /* invalidate cache if desired */
    MPI_Alltoall(...); /* perform benchmark */
}
TIMER_STOP; /* take time */
total = TIMER_ELAPSED;
total −= calibrate_cache_flush(cnt); /* subtract
                                cache clearing time */
```

Listing 2. MPBench Benchmark Scheme

```
for(i=0; i<numbarr; i++) MPI_Barrier(...);
t0 = MPI_Wtime(); /* take time */
for (i=0; i<reps; i++) {
        MPI_Alltoall(...); /* execute benchmark */
}
t1 = MPI_Wtime(); /* take time */
time = (t1−t0)/reps;
```

Listing 3. Intel MPI Benchmark Scheme

tion separately. Thus, it is crucial to the modeling of parallel applications to have accurate models for the latency of collective operations. Several benchmarking studies have been done for different systems [3, 21, 24].

In this paper, we discuss and analyze different established measurement methods for collective operations on parallel computing systems and point out common systematic errors. Our measurement method is derived from the SKaMPI benchmarks [27] and is universal and not limited to point-to-point based methods or specific algorithms. We also extend the benchmarking principles to capture the new parameters, host overhead and asynchronous progression, for nonblocking collective operations. The following section discusses established benchmark methods and their problems.

1.1 Related Work

Different benchmark schemes for blocking collectives have been proposed. Currently known methods can be divided into three groups. The first group synchronizes the processes explicitly with the use of synchronization routines (i.e., MPI_Barrier). The second scheme, presented in [27], establishes the notion of a global time and the processes start the operation synchronously. The third scheme assesses the quality of a collective implementation by comparison to point-to-point operations [25] and is thus limited to algorithms using point-to-point messages. We investigate several publicly available benchmarks in the following and characterize them in the three groups.

MPPTEST implements the discussions on reproducible MPI performance measurements [6]. As described in the article, the operation to measure is executed in a warm-up round before the actual benchmark is run. The processes are synchronized with a single MPI_Barrier operation before the operation is run N times in a loop. A pseudo-code is shown in Listing 1. Only the time measurements at rank 0 are reported to the user.

MPBench was developed by Mucci et al. [18]. MPBench does not synchronize at all before the benchmarks. Rank 0 takes the start time, runs N times the collective operation to benchmark and takes the end time. A pseudo-code is shown in Listing 2. The timer can use the `RDTSC` CPU instruction [15] or gettimeofday(). Time measurement is only performed and printed on rank 0.

Intel MPI Benchmarks (formerly Pallas MPI benchmarks [20]), measure a wide variety of MPI calls including many collective functions. The code issues a definable number of MPI_Barrier operations before every benchmark and measures the collective operation in a loop afterwards. The time needed to execute the loop is taken as a measurement point. The scheme is shown in Listing 3. The benchmark prints minimum, maximum and average time over all processes.

SKaMPI The SKaMPI benchmark uses a time-window based approach, described in [27], that ensures that all processes start the operation at the same time. No explicit synchronization is used and the times are either reported per process or cumulative.

1.2 Systematic Errors in Common Measurement Methods

Benchmarking collective operations is a controversial field. It is impossible to find a single correct scheme to mea-

sure collective operations because the variety of real-world applications is tremendously high. Thus, every possible benchmark scheme may have its justification. However, microbenchmarks are often used to compare implementations and to model the influence of the communication to several different applications. This is why a benchmark should represents the *average* or at least the majority of applications. Our model application for this work is a well balanced application that issues at least two different collective operations in a computational loop (cf. [23]). This model application would benefit from well balanced collective operations that do not introduce process skew. The following paragraphs describe common systematic errors done in the measurement of collective operations. This section is concluded with the selection of a benchmark method. All discussions apply in the same way to nonblocking collective operations because the problem of distributed time measurement and synchronization is identical in the blocking and nonblocking case.

Implementation Assumptions The implementation of collective communication operations can usually not be prescribed to provide as much optimization space as possible. Thus, any point-to-point algorithm or hardware supported operation that offers the functionality defined in the interface is a valid implementation. Some research groups used elaborate techniques (e.g., hardware optimization and/or specialized algorithms) to optimize collective communication on different systems (cf. [13, 26, 28]). A portable benchmark to measure the collective interface can not make any assumptions about the internal implementation like [25].

Results on multiple processes A second problem is that benchmarks are usually providing a single number to the user, while all processes benchmark their own execution time. Some benchmarks just return the time measured at a single process (e.g., the rank 0), some use the average of all times and some the maximum time of all processes. The decision which time to use for the evaluation is not trivial. Sometimes, it is even desirable to include the times of all ranks in the evaluation of the implementation. Worsch et al. define three schemes to reduce the times to a single number, (1) the time needed at a designated process, (2) maximum time on all processes and (3) the time between the start of the first process and the finish of the last. This list can be extended further, e.g., (4) the average time of all processes or (5) the minimum time might play a role and is returned by certain benchmarks.

A simplified LogP model derived from [4] is used to model the network transmissions and effects in collective communication. The LogP model uses four parameters to describe a parallel system. The parameter **L** models the network latency and **g** is the time that has to be waited between two packets. The CPU overhead **o** does not influence the network transmission and is thus omitted in our simplified model. The number of participating processes **P** is constantly four in our examples.

Pipelined Measurements Another source for systematic errors are pipelining effects that occur when many operations are executed in a row. A common scheme is to execute N operations in a loop, measure the time and divide this time by N. This scheme was introduced to avoid the relative high inaccuracy of timers when short intervals are measured. We show in Section 2 that this is not necessary for high precision timers. An example LogP modeling for MPI_Bcast with root 0, implemented with a linear scheme, is shown in Figure 1. A single execution is much more likely to model the behavior of real optimized applications. Both schemes result in different execution times, e.g., the worst-case (maximum among all processes) returned latency for a single execution is $L + 2g$ for a single operation and $(8g + L)/3$ for three successive operations. The pipelined measurement tends to underestimate the latency in this example.

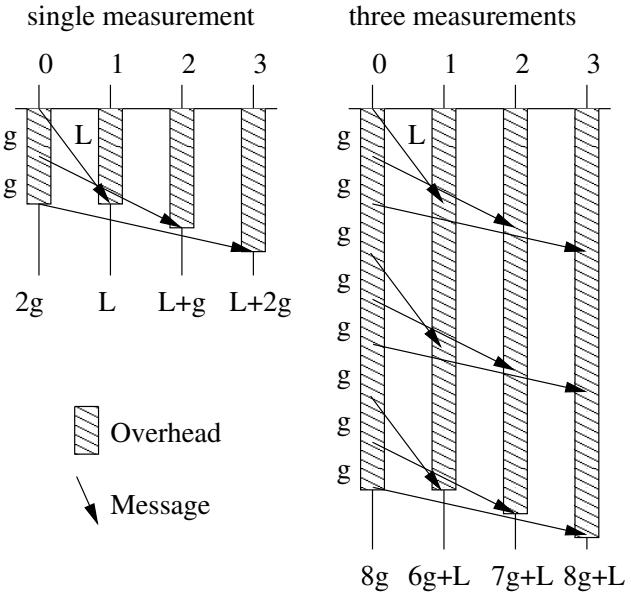

Figure 1. Pipeline Effects for a linear Broadcast Implementation

Process Skew The LogP models in the previous paragraph assumed that the first operation started at exactly the same (global) time. This is hardly possible in real parallel

systems. The processes arrive at the benchmark in random order and at undefined times. Process skew is influenced by operating system noise [1, 16] or other collective operations (cf. Figure 1 where rank 0 leaves the operation after $2g$ and rank 3 finishes only after $L + 2g$). A LogP example for the influence of process skew is shown in Figure 2. The left

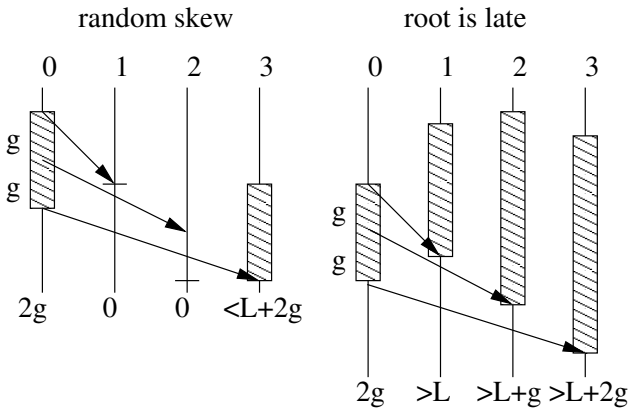

Figure 2. Influence of Process Skew on a Broadcast Benchmark

side shows a random skew pattern where rank 1 and 2 arrive relatively late and do not have to wait for their message while the right side shows a situation where the root (rank 0 in this example) arrives late and all ranks have to wait much longer than usual. Process skew can not be avoided and is usually introduced during the runtime of the parallel program. This effect is well known and several benchmarks use an MPI_Barrier before the measurement to correct skew.

Synchronization Perturbation and Congestion The MPI_Barrier operation has two problems, the first one is that this operation may be implemented with any algorithm because it only guarantees that all processes arrived before the first returns from the call. There is no guarantee that the processes leave at the same time (i.e., the barrier operation may introduce process skew). The second one is that it may use the same network as other collective operations which may influence the messages of the investigated collective operation. An example with a linear barrier implementation is shown in Figure 3.

Network Congestion Network congestion can occur if multiple operations are started successively or synchronization messages interfere with the measurement. This influences the measured latencies.

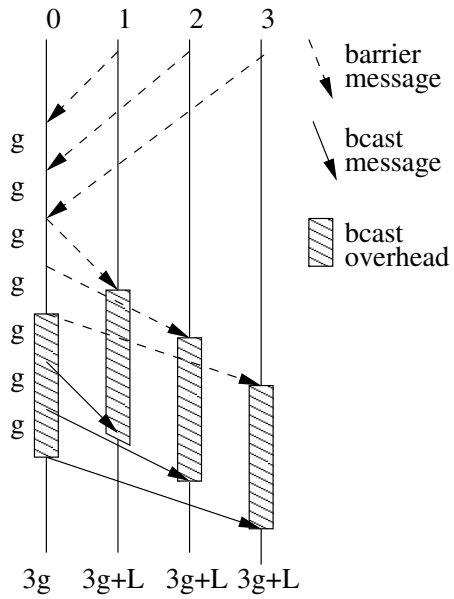

Figure 3. Possible Effect of Barrier Synchronization on Measurements

1.3 Selecting a Benchmark Scheme

The SKaMPI benchmarks avoid most of the systematic errors with the window-based mechanism [27]. This mechanism relies heavily on the assumption, that the difference between the local clocks does not change (drift) or changes in a predictable way (can be corrected). We analyze the clock drift in the following sections. Another problem might be the variation in the latency of point-to-point messages. This variation is also analyzed and a new accurate point-to-point synchronization scheme is presented. Furthermore, we propose and implement a new scalable group synchronization algorithm which scales logarithmically instead of the linear SKaMPI approach.

2 Measurements in Parallel Systems

Several restrictions apply to measurements in parallel systems. One of the biggest problems is the missing time synchronization. Especially in cluster systems, where every node is a complete and independent system with its own local time source, one has to assume that potentially all processes of a parallel job run at slightly different clock speeds. However, to perform the necessary measurements, we need to synchronize all clocks or have at least the time offsets of all processes to a global time. It can also not be assumed that the processes start in any synchronous state. To ensure portability, MPI mechanisms have to be used to

synchronize. However, collective operations semantics do not guarantee any timing, thus, we need to synchronize the processes with point-to-point operations. Those operations do not guarantee timing either but are less complex than collective operations (no communication patterns). We analyze local time sources and their accuracy in the following. This is followed by an analysis of the clock skew in parallel systems and the distribution of latencies. This analysis is used to derive a novel and precise synchronization scheme in the next section.

2.1 Local Time Measurement

All time sources in computing systems work in a similar way: They use a crystal that oscillates with a fixed frequency and a register that counts the number of oscillations since a certain point in time (for example the system startup). However, the way to access this information can be different. Some timing devices can be configured to issue an interrupt when the register reaches a certain value or overflows and others just enable the programmer to read the register.

A very important timer in a modern PC is the Real Time Clock (RTC) which is powered by a battery or capacitor so that it continues to tick even when the PC is turned off. It is used to get the initial time of the day at system startup, but since it is often very inaccurate (it is optimized for low power consumption, not for accuracy), it should not be used to measure short time differences.

Another time-source is the Programmable Interval Timer (PIT). It can be configured to issue interrupts at a certain frequency. These interrupts are used to update the system time and perform several operating system functions. When using the system time (for example via `gettimeofday()`) one must be aware that the returned value might be influenced by `ntp` or other time-synchronization mechanisms and that there is a whole software stack behind this simple system call which might add additional perturbation (i.e., interrupts, scheduling, etc.).

The resolution of the discussed time sources is not very high and not accurate enough for the benchmarking of fast events like single message transmissions. Thus, many instruction set architectures (ISA) offer calls to read the CPU's clock register which is usually incremented at every tick. For example the x86 and x86-64 ISAs offer the atomic instruction `RDTSC` call [15] to read a 64 bit CPU tick register. In fact most modern ISAs support similar features. The resolution of those timers is usually very high (e.g., $0.5ns$ on a 2GHz system). It has to be noted that this mechanism introduces several problems on modern CPUs. The first issue is caused by techniques that dynamically change the CPU frequency (e.g., to save energy) such as "Intel-SpeedStep" or "AMD-PowerNow". Changing the CPU clock results in invalid time measurements. Thus, we recommend to disable those mechanisms in cluster systems. A second problem, called "process hopping", may occur on multi-processor systems where the process is re-scheduled between multiple CPUs. The counters on the CPUs are not necessarily identical. This might also influence the measurement. This problem is also minor because most modern operating systems (e.g., Linux 2.6) offer interfaces to bind a process to a specific CPU (e.g., CPU affinity), and in fact, most operating systems avoid "process hopping" by default.

2.2 Clock Skew and Network Latencies

The crystals used for hardware timers are not ideal with respect to their frequency. They may be a little bit slower or faster than their nominal rate. This drift is also temperature dependent and has been analyzed in [17] and [19]. It was shown that this effect is significant enough to distinguish / identify single computers and sometimes even the timezone which they are located in. Due to other effects, such as the NTP daemon that synchronizes every 11 minutes (local time!) by default, the clock difference between two nodes may behave very unpredictable. Therefore the usage of a software independent clock like the TSC is generally a viable alternative. Of course those effects could have a negative influence on collective benchmarks which rely on time synchronization, especially if the synchronization is done only once before a long series of benchmarks. In our experiments it turns out that two clocks (even on identical hardware) always run at slightly different speeds. Therefore we analyzed the clock skew between various nodes in a cluster system over a longer period of time.

Similar to the clocks, that do not run totally synchronously, we do also expect a variance in the network transmission parameters for different messages. The most important parameter for benchmarks and synchronization is the network latency (or round-trip-time (RTT) in ping-pong benchmarks). Thus, we have to analyze the variance of RTT for different networks.

We used a simple ping pong scheme to determine the RTT: Rank 1 sends its local time t_1 as an eight byte message with a blocking send to rank 2. As soon as rank 2 has completed the corresponding recv, it sends its local time t_2 back to rank 1. After rank 1 is finished receiving that timestamp it checks his local time t_3. To use a portable high-precision timing interface and to support many network interconnects, the benchmark scheme was implemented in the Netgauge performance measurement framework [12]. A pseudo-code is shown in Listing 4. The difference between the first and the second timestamp obtained by node 1 is the *roundtrip time* ($t_{rtt} \leftarrow t_3 - t_1$). On our x86 systems, we used `RDTSC` in the `take_time()` macro because this gives us a very high resolution and accuracy. However, we

```
if (rank == 0) {
  t1 = take_time();
  module→ send(1, &t1, 8);
  module→ recv(1, &t2, 8);
  t3 = take_time();
} else {
  module→ recv(0, &t1, 8);
  t2 = take_time();
  module→ send(0, &t2, 8);
}
```

Listing 4. The RTT Benchmark

double-checked our findings with MPI_Wtime (Which uses `gettimeofday()` or similar functions) to avoid common pitfalls described in Section 2.1.

This measurement was repeated 50,000 times, once every second over a period of 14 hours. We gather data to get information about the RTT distribution and also the clock skew between two nodes. Thus, we define the difference between t_1 and t_2 as *clock-difference* ($t_{diff} \leftarrow |t_1 - t_2|$) and collect statistical data for the clock differences too.

The benchmark results, a histogram of 50,000 RTT measurements in 200 uniform bins, for the latency (RTT/2) distributions of InfiniBand, Myrinet and Gigabit Ethernet are shown in Figure 4.

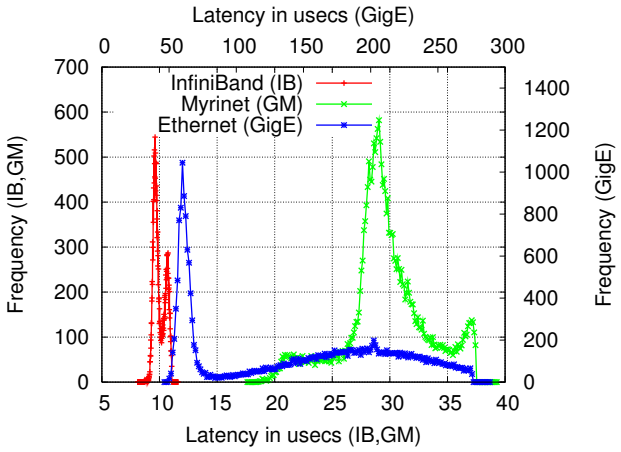

Figure 4. Distribution of Roundtrip Times

The clock skew results of 6 different node pairs on our test systems are shown in Figure 5. We see that the clock difference behaves relatively linear at a coarse scale.

3 Time Synchronization

We describe a new time synchronization scheme that bases on our analysis of clock skew and latency variation.

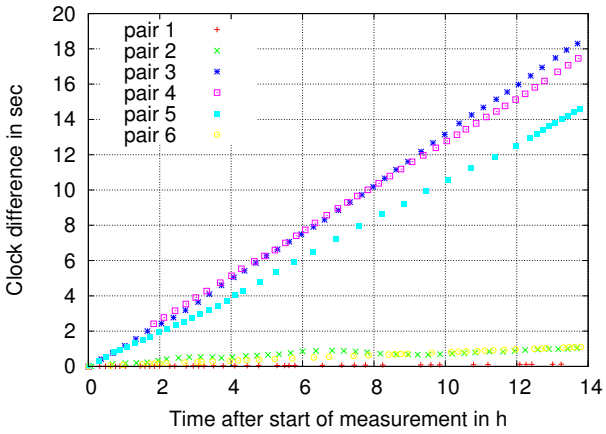

Figure 5. Clock Drift for Different Pairs

We begin by defining a scheme to synchronize two processes and derive a scalable scheme to synchronize large process groups. We used this schemes to synchronize the processes in our collective benchmark NBCBench [11] that uses a window-based benchmark scheme.

3.1 Synchronizing two Processes

A clock synchronization between two peers is often accomplished with a ping pong scheme similar to the one described above: Two processes calculate their clock difference so that the client node knows his clock offset relative to the server node. This offset can be subtracted from the client's local time when clock synchronization is required. However this procedure has certain pitfalls one has to be aware of.

Many implementations of the scheme described above, for example the one found in the SKaMPI code, use MPI_Wtime to acquire timestamps. This is of course the most portable solution and works for homogeneous nodes as well as heterogeneous ones. But you can not be sure which timing source is used by MPI_Wtime, for example the usage of `gettimeofday()` is, due to its portability, quite likely. But the precision of this clock can vary (cf. Section 2.1).

We showed in Section 2 that measured network latencies are varying with an unpredictable distribution. The effect of this pseudo-random variation of the latency to the clock synchronization has to be minimized. Many codes use the average or median of the clock differences and roundtrip times. This is not a viable option because their distribution is asymmetric and their variance is high. A better approach is the measurement of roundtrip time and clock difference at the same time and only use the clock differences obtained in measurements with round trip times below a certain thresh-

old.

The distribution of roundtrip times can not be known in advance. That implies that this threshold can not be selected easily. We chose a different approach to ensure accurate measurements. For the measurements in Section 2, we used only the 25% of the results that had the smallest roundtrip time. However, this requires a fixed number of measurements to be conducted every time and makes this scheme unusable for online measurements. For this purpose, we developed another approach. We conduct as many measurements as we needed so that the minimal observed roundtrip time does not become smaller for N consecutive measurements.

While this scheme is guaranteed to converge, it is not possible to predict how many measurements have to be conducted for a certain N. It is also not possible to select N such that it ensures a certain quality of the observed *minimal* roundtrip time. Thus, we performed a simulation to find suitable values for N for different interconnection networks.

The simulation takes a random roundtrip time from our list[1] and checks if it was bigger than the smallest one observed in this run. If this condition is met N consecutive times the run is completed and we compute the difference (in percent) between the smallest RTT in our whole dataset and the one observed in the current run. The average difference is shown in the "quality" graphs in Figure 6. In the same figure we graphed how many iterations of this simulation (which is equal to the number of measurements needed in the online scenario) were needed until the selected minimum was the smallest for N iterations. This is graphed as "cost". Since this graph looked very similar for all tested networks we only plotted it once.

It shows that the quality of the measurement and the measurement costs (i.e., the number of measurements) increase relatively independent of the network with N. However, the quality of the results improves fast for small N and seems to saturate around 5%. To support every network, we chose $N = 100$, which has an error of less than 10% for our tested networks and costs approximately 180 measurements.

3.2 Scalable Group Time Synchronization

Collective time synchronization means that all processes know the time difference to a single process so that every process can compute a global time locally. Rank 0 is conveniently chosen as global time source, i.e., after the time synchronization, every rank knows the difference between its own clock and rank 0's clock. This enables globally synchronous events initiated by rank 0. For example, rank 0 can broadcast a message that some function is to be executed at

[1] we used the 50,000 RTTs gathered as described in Section 2

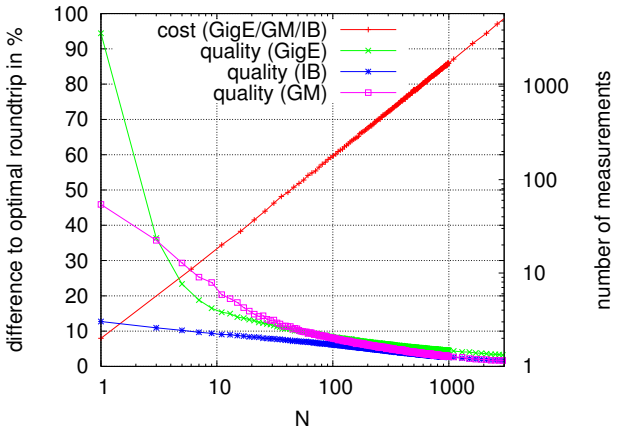

Figure 6. Influence of N on Quality

rank 0's local time x. Every rank can now calculate its local time when this operation has to be executed.

A common scheme to synchronize all ranks is start the point-to-point synchronization procedure between rank 0 and every other rank. The disadvantage of this scheme is that it takes $P - 1$ synchronization steps to synchronize P processes (each process sends and receives multiple messages in each step). We propose a new and scalable time synchronization algorithm for our scalable benchmark. Our algorithm uses $\lceil log_2 P \rceil$ communication steps to synchronize P processes.

The algorithm divides all ranks into two groups. The first group consists of the maximum power-of-two ranks, $t = 2^k; max(k \in \mathbb{N}), t \leq P$ beginning from rank 0 to rank $t - 1$. The second group includes the remaining ranks t to $P - 1$.

The algorithm works in two steps, the first group synchronizes with a tree-based scheme in $log_2 t$ synchronization rounds. The point-to-point scheme, described in Section 3.1 is used to synchronize two processes. Every rank r in round r acts as a client if $r \ mod \ 2^r = 0$ and as a server if $r \ mod \ 2^r = 2^{(r-1)}$. All clients r use rank $r + 2^{(r-1)}$ as server and all servers rank $r - 2^{(r-1)}$ as client. All client-server groups do the point-to-point synchronization scheme in parallel. Every server gathers some time difference data in every round. This gathered data has to be communicated at the end of every round. To be precise, a server communicates $2^{(r-1)}$ time differences to its client at the end of every round. The clients receive the data and update their local differences to pass them on in the next step. After $log_2 t$ rounds, rank 0 knows the time differences to all processes in group 1.

In the second step, all processes in group 2 choose peer $r - t$ in group 1 to synchronize with. All processes in group 2 synchronize in a single step with their peers and send the

result to rank 0 which in turn calculates all the time offsets for all processes and scatters them accordingly. After this step, all processes are time synchronized, i.e., know their time difference to the global clock of rank 0. The whole algorithm for an example with $P = 7$ is shown in Figure 7.

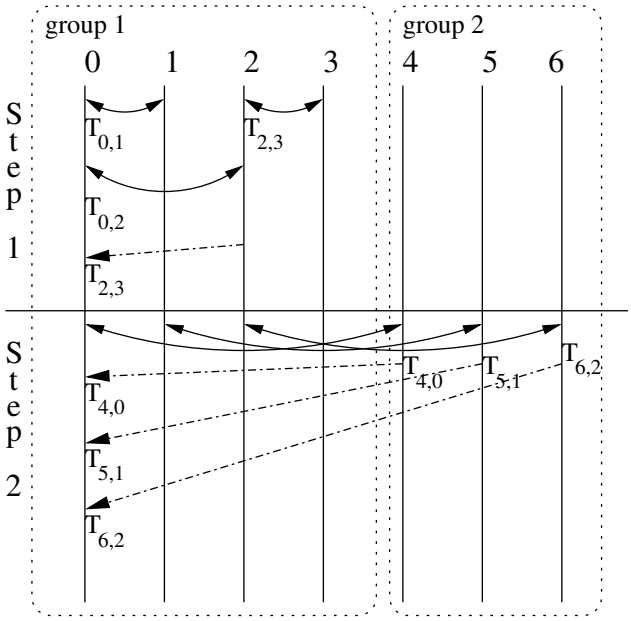

Figure 7. Synchronization Method

Figure 8 shows the difference in synchronization time between a linear scheme and the proposed logarithmic algorithm. The benchmark, which measures the synchronization time at rank 0, was run on the *Odin* cluster at Indiana University, the cluster consists of 128 dual Opteron dual-core nodes. To simulate a real application, we used all four cores of the machine. The synchronization time is greatly reduced (up to a factor of more than 16 for 128 processes) with the new scheme.

4 Designing the Microbenchmark

We used our new findings to implement a new benchmark scheme in the benchmark suite Netgauge. The implementation bases on NBCBench which was first introduced to measure the performance and overlap characteristics of nonblocking collective operations in [11]. However, the first part of this paper focuses on measurement methods for blocking collective operations which is also supported by Netgauge. This section gives a rough overview about the necessary parts of Netgauge to understand how the proposed measurement method is implemented. In our method, we use the hints of [6] and omit the implementation details

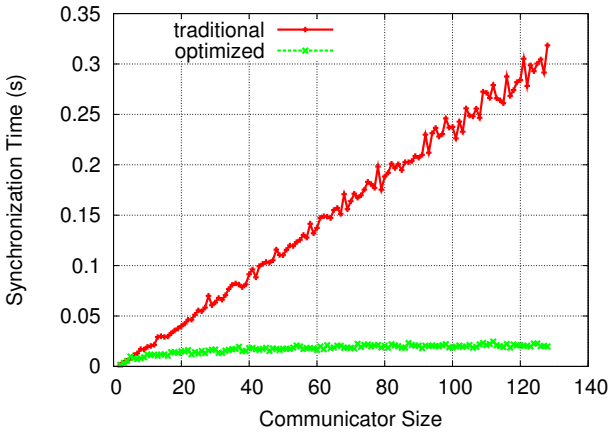

Figure 8. Synchronization Time over IB

```
for(p=2;p<numprocs;p=p*2) {
  MPI_Comm_create(...); /* create sub-communicator */
  sync_init_stage1(...);
  for(s=1;s<size,s=s*2) {
    for(i=1;i<warm;i++) MPI_Allreduce(...); /* warmup */
    sync_init_stage2(...);
    for(b=0;b<numbench;b++) {
      sync(...);
      t = -take_time();
      MPI_Allreduce(...);
      t += take_time();
} } }
```

Listing 5. Netgauge Collective Benchmark Scheme for Allreduce

due to space restrictions. We implement a SKaMPI-like window-based mechanism with the synchronization techniques described above. We introduced an abstract synchronization interface that calls three different functions. Netgauge mainly consists of three nested loops, the outer loop runs over different process numbers, the second loop runs over different data sizes and the inner loop performs multiple time measurements. The first call to a synchronization function (`sync_init_stage1()`) is done in the outer loop and determines the time offsets to rank 0. The second call (`sync_init_stage2()`) is inside the loop over all data sizes, and checks and changes the window size if necessary (cf. [27]). The inner loop calls `sync()` before every benchmark which waits until the next window starts. The scheme is shown in Figure 9 and a pseudo-code is presented in Listing 5. The output can be customized by the user. Netgauge is able to print the times for all processes or the median, average, minimum or maximum time of all pro-

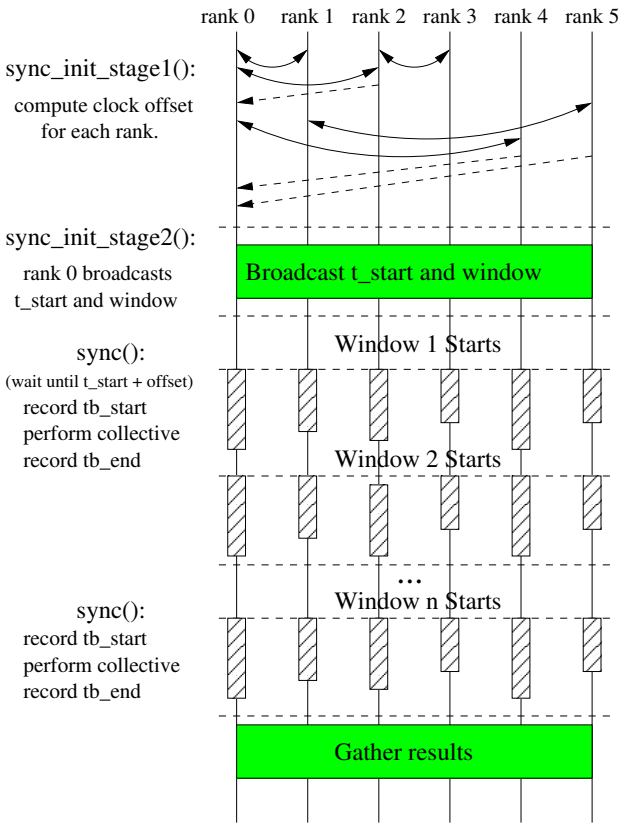

Figure 9. Benchmarking blocking collectives

cesses. It is also possible to select between the minimum, maximum, average or median time for all measurements.

4.1 Nonblocking Collective Operations

The previous sections discussed distributed synchronization and measurement principles for collective operations. In the case of blocking collective operations, the results can be directly reported to the user or used in communication models. However, measuring nonblocking collective communication is more complicated. A nonblocking communication consists of two phases, the initialization and the wait phase. Initialization is usually the time when the operation is posted, for example a call to NBC_Ibcast to start a nonblocking broadcast. The corresponding NBC_Wait waits for the completion of the operation. Ideally, there is enough computation between the initialization and wait and the communication progresses independently. This effectively turns NBC_Wait into a no-op (the communication is already finished before wait is called). Thus, the only overheads that the user-code faces is the time needed by NBC_Ibcast and the time spent in NBC_Wait. Thus, the communication-latency can be ignored as long as there is enough computation to overlap and the communication progresses independently.

Progression can either be independent from the user (e.g., in the threaded case [9]) or has to be performed manually (e.g., by calling NBC_Test frequently [10]). Thus, we conclude that the two important parameters are CPU overhead and asynchronous progression.

In order to assess all overheads, we propose a two-stage scheme to benchmark nonblocking collective operations. The first stage benchmarks the time T_b for the blocking execution of non-blocking collective operations (initialization followed by a wait). The second stage initializes the nonblocking communication simultaneously on every rank, computes for time t_b and then waits for completion. Optionally, progression calls can be done during the computation phase. The benchmark schema is shown in Figure 10 and Listing 6.

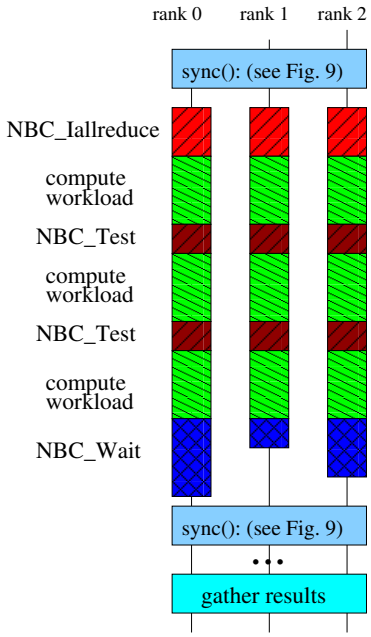

Figure 10. Black-box test for manual progression strategy

The computation loop might contain calls to NBC_Test in order to progress the outstanding communication requests. This time-based scheme is able to assess all overheads related to calls on the CPU. However, it fails to capture other overheads for example interrupt times or scheduling overheads because it's time-based. In order to assess all additional overheads one can employ a work-based scheme, such as:

1. benchmark the time t_b for a blocking communication with scheme described before

```
for(p=2;p<numprocs;p=p*2) {
  MPI_Comm_create(...); /* create sub-communicator */
  sync_init_stage1(...);
  for(s=1;s<size,s=s*2) {
    for(i=1;i<warm;i++) /* warmup */
      NBC_Iallreduce(..., &req); NBC_Wait(req);
    t_b -= take_time()
    NBC_Iallreduce(..., &req); NBC_Wait(req);
    t_b += take_time()
    t_b = max t_b of all processes
    sync_init_stage2(...);
    for(b=0;b<numbench;b++) {
      sync(...);
      t_init -= take_time();
      NBC_Iallreduce(..., &req);
      t_init += take_time();
      do_compute(t_b);
      t_wait -= take_time();
      NBC_Wait(req);
      t_wait += take_time();
} } }
```

Listing 6. Netgauge Benchmark Scheme for Iallreduce

2. find workload λ that needs t_b to be computed
   (a) $\lambda = 0$
   (b) while($t_\lambda < t_b$) { increase workload λ by δ;
   t_λ = time to compute workload λ }
3. collect maximum time workload λ of all processes
4. start timers t_{ov} simultaneously on all processes
5. start communication (also simultaneously, e.g., NBC_Ibcast)
6. compute fixed workload λ
7. wait for communication (NBC_Wait)
8. stop timer t_{ov}

Both, the time-based and the workload-based methods, are useful to determine and compare different sources of overhead. Thus, we implemented both methods in Netgauge. Figure 11 shows the benchmark result for the blocking and nonblocking collective NBC_Iallreduce operations on 32 InfiniBand nodes. The nonblocking operations are either implemented with user-progression ("nb/time/test" with tests for every 1024 bytes, which was the optimal result for the test-interval determined before) or with an asynchronous progress thread running on a separate core. Both, the time- and the work-based benchmarking scheme delivered similar results for the test-based progression. However, there were clear differences in the threaded progression case because the time-based scheme hides overheads such as context switching. We see that threaded progression seems only useful when more then $512\,kiB$ are reduced. We also see the eager/rendezvous switching point at 128 bytes and the collective algorithm switching point at $512\,kiB$. More measurements can easily be collected with LibNBC and Netgauge.

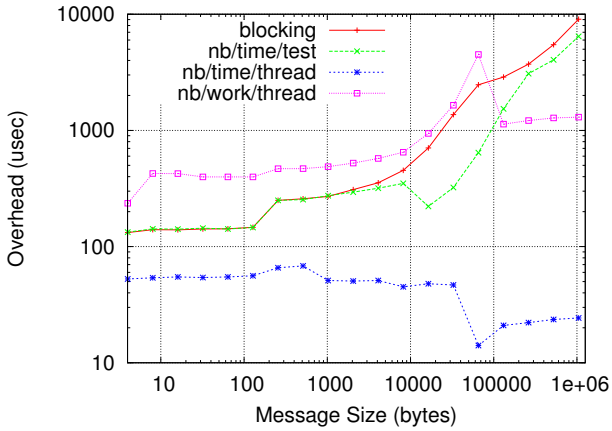

Figure 11. Blocking and nonblocking overhead for NBC_Iallreduce on 32 InfiniBand nodes

4.2 Asynchronous Progress

We presented techniques to benchmark the overhead of nonblocking collective operations. However, assessment of asynchronous progression is still undefined. In this section, we propose an indirect method to measure asynchronous progress by comparing different test strategies. The number of necessary progression calls can also be assessed with Netgauge. We proposed a black box testing scheme in [10] that performs N tests during a message transmission. N is defined as a function of the message size in order to reflect the needs of the pipelined protocol:

$$N = \left\lfloor \frac{size}{interval} \right\rfloor + 1$$

For example, if the datasize is 4096 bytes and the interval is 2048 bytes, the benchmark issues one test at the beginning, one after 50% of the computation and one at the end. The test-interval for Netgauge can be chosen by the user. The scheme is shown in Figure 10. Note that every process which takes part in the benchmark has to record the overhead caused by progression calls. Of course it has to be assured that the workload is long enough to hide the entire communication, as described earlier.

5 Conclusions

In this work, we analyzed different schemes to benchmark the latency, overhead and independent progress of blocking and nonblocking collective operations. We defined certain systematic errors in common methods and proposed a new and scalable scheme which avoids those errors. Our method bases on a window-based measuring method. Therefore, we propose and analyze a new scalable group synchronization method for collective benchmarks. Our method is more than 16 times faster than other established schemes on 128 processes and promises to be more accurate. The described scheme to benchmark overhead, communication time and progression of blocking and nonblocking collective operations has been implemented in the open source tool NBCBench which is available at the author's webpage.

Acknowledgments

This research was funded by a gift from the Silicon Valley Community Foundation, on behalf of the Cisco Collaborative Research Initiative of Cisco Systems.

References

[1] S. Agarwal, R. Garg, and N. Vishnoi. The impact of noise on the scaling of collectives: A theoretical approach. In *12th Annual IEEE International Conference on High Performance Computing*, Goa, India, Dec. 2005.

[2] S. R. Alam, N. Bhatia, and J. S. Vetter. An exploration of performance attributes for symbolic modeling of emerging processing devices. In R. H. Perrott, B. M. Chapman, J. Subhlok, R. F. de Mello, and L. T. Yang, editors, *HPCC*, volume 4782 of *Lecture Notes in Computer Science*, pages 683–694. Springer, 2007.

[3] T. Bönisch, M. M. Resch, and H. Berger. Benchmark evaluation of the message-passing overhead on modern parallel architectures. In *Parallel Computing: Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo97*, pages 411–418, 1997.

[4] 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.

[5] D. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, 16:35–43, February 1996.

[6] W. Gropp and E. L. Lusk. Reproducible measurements of mpi performance characteristics. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 11–18, London, UK, 1999. Springer-Verlag.

[7] T. Hoefler, R. Janisch, and W. Rehm. Parallel scaling of Teter's minimization for Ab Initio calculations. 11 2006. Presented at the workshop HPC Nano in conjunction with SC'06.

[8] T. Hoefler, A. Lichei, and W. Rehm. Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks. 03 2007.

[9] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.

[10] T. Hoefler and A. Lumsdaine. Optimizing non-blocking Collective Operations for InfiniBand. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 04 2008.

[11] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for mpi. In *Accepted for publication in the proceesings of Supercomputing'07*, 2007.

[12] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm. Netgauge: A Network Performance Measurement Framework. In *High Performance Computing and Communications, Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007, Proceedings*, volume 4782, pages 659–671. Springer, 9 2007.

[13] 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.

[14] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. LogfP - A Model for small Messages in InfiniBand. In *Proceedings, 20th International Parallel and Distributed Processing Symposium IPDPS 2006 (PMEO-PDS 06)*, April 2006.

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

[16] 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.

[17] T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, 2005.

[18] P. J. Mucci, K. London, and J. Thurman. The MPIBench Report. Technical report, CEWES/ERDC MSRC/PET, 1998.

[19] S. Murdoch. Hot or not: revealing hidden services by their clock skew. *Proceedings of the 13th ACM conference on Computer and communications security*, pages 27–36, 2006.

[20] Pallas GmbH. Pallas MPI Benchmarks - PMB, Part MPI-1. Technical report, 2000.

[21] J. Pjesivac-Grbovic. Open MPI Collective Operation Performance on Thunderbird. Technical report, The University of Tennessee, Computer Science Department, Knoxville, Technical Report, UT-CS-07-594, 2007.

[22] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance Analysis of MPI Collective Operations. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, Denver, CO, April 2005.

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

[24] S. Saini, R. Ciotti, B. T. N. Gunney, T. E. Spelce, A. E. Koniges, D. Dossa, P. A. Adamidis, R. Rabenseifner, S. R. Tiyyagura, M. Mller, and R. Fatoohi. Performance evaluation of supercomputers using hpcc and imb benchmarks. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006.

[25] M. Shro and R. Geijn. CollMark MPI Collective Communication Benchmark. Technical report, University of Texas at Austin, December 1999.

[26] 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.

[27] T. Worsch, R. Reussner, and W. Augustin. On benchmarking collective mpi operations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 271–279, London, UK, 2002. Springer-Verlag.

[28] 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.