Automated MPI Correctness Checking What if there was a magic option?

Patrick Ohly and Werner Krotz-Vogel

Intel GmbH
Hermülheimer Straße 8a
D–50321 Brühl, Germany

Abstract. Writing correct *and* portable MPI programs is hard. Out of bound parameters, inconsistent use of data types and many other complex violations of using MPI correctly can now automatically be detected at runtime. While interactive debuggers have been extended to handle the concurrent processes of MPI applications, still there are numerous bugs which are hard (because it would require a prohibitive amount of manual work) or even impossible (because it requires more control over the program execution than available via a debugger) to find with just a debugger. This paper presents the new MPI correctness checking feature of Intel® Trace Analyzer and Collector, which solves this problem. We describe its design, implementation and performance. As a use case an analysis of the HPCC benchmark is presented which actually contains some violations of the MPI standard in the 1.0.0 release.
Keywords: MPI, correctness checking, errors, HPCC

1 Introduction

Writing correct *and* portable MPI programs is hard: the sheer number of different MPI calls with their sometimes subtle semantic differences can easily be used inconsistently or incorrectly, even by expert programmers. Furthermore, inconsistent or incorrect use of parameters may only occur at runtime. Thus significant development time for handcraft in debugging, testing and quality assurance is required to assure correctness and quality of an MPI application, and this time spent is just on top of mastering the complexities of parallel programming with MPI, while a programmer, expert, or scientist originally wanted to just focus on the solution of a computational problem. Add to that the problem that MPI implementations are given considerable freedom in how they implement certain features of the standard, then it is clear that developers benefit from an MPI correctness checking tool that helps mastering this complexity.

MPI implementations themselves typically implement only minimal sanity checks. Their focus is on achieving the highest possible performance: expensive checks in terms of runtime overhead are therefore prohibitive; the less expensive checks might not get implemented because they are not considered important enough. Hence a correct program should run perfect, while an incorrect program may or may not fail during its run, or just produce incorrect results.

Using traditional debuggers fails with this class of correctness violations. They have traditionally been used to investigate running programs or core dumps. Even though there are debuggers which explicitly support MPI, they are limited to presenting the current status of an application. By single stepping through the execution the user can try to understand how the application works, but this is a time-consuming task and to fully check an MPI application not only requires monitoring local variables, but also all data sent between processes.

Clearly these are checks that a computer is much more capable of executing automatically, reliably and without a gap. By encoding knowledge of MPI and MPI constructs like data types into the checking code it becomes possible to check MPI specific semantic. This does not replace an interactive debugger, but rather augments it: once a problem is found, the easiest way to investigate it maybe still to stop the application and go back to the source code where the erroneous MPI occurred while the process is still running and all its data is available.

The Intel® Trace Analyzer and Collector (referred to in this paper as TAC) [4] now supports this kind of automated MPI correctness checking. As a starting point for the analysis of a problem a message text is printed to stderr. Because these reports contain all relevant information, they are also sufficient for most post-mortem analysis, or when using a debugger is impossible or not desired, as in automatic regression testing. The MPI correctness checking with TAC also includes full support for all common MPI debuggers as well as post-mortem analysis of an application run.

2 Design

2.1 Goals

Out-of-the-box usability has been the primary goal in the design of the MPI correctness checking feature for TAC, as a tool with a very fast learning curve and immediate results will be most helpful for MPI developers. This implies that a problem must be reported with enough information so that the report is comprehensible on its own, ideally without even having to consult a manual first.

Another usability aspect is that the information given on errors must be sufficient for the user to understand which call and which condition triggered the error report and to verify whether it actually constitutes a problem. If the user has to gather more information himself first from other sources, he might either fail to find it or give up because it is too much work. If the user cannot verify easily the condition that caused such an error report, he might ignore it as a false positive even if it is not.

A further design goal of the MPI correctness checking was to never report false positive messages. False positives severely impact the usefulness of such a tool because they put the effort of distinguishing between false and genuine problem reports on the user: there is a tradeoff between finding every conceivable

problem with the risk of producing lots of warning at one end of the spectrum and detecting some problems reliably without extraneous reports at the other end. If a check cannot be implemented reliably, then it should either not be implemented or at least not be enabled by default. As will be demonstrated below, an ample range of real-world MPI problems can be detected reliably. So an application run without reports does at least guarantee that none of the conditions checked for error or warning occurred, hence it is a very good indication for correct MPI usage, however it does not provide an absolute guarantee for correct execution beyond that. Chances are high that most existing problems are found.

The goal for performance of MPI correctness checking was to keep the impact on execution times of typical applications smaller than a 3-5x slowdown, which allows an application to still complete in an acceptable time frame. The impact should be independent of the number of processes, hence preserve the level of scalability of an MPI application. Some errors might only occur with high number of processes or large problem sizes. Obviously runtime dependent errors will be affected, but they might both occur or be hidden by different runtimes.

Intel® MPI was (and still is) the only MPI implementation targeted, but the possibility to compile for other MPIs was considered important. Therefore the focus on Intel MPI was used to limit implementation work to those tests not already offered by Intel MPI and to simplify testing.

2.2 Overview

Figure 1 shows how the different components fit together: in order to do the checking, all MPI calls are intercepted using the standard MPI profiling interface. The checks are executed inside the MPI wrappers. This gives them full access to all MPI parameters and local state of the processes without requiring intermediate disk storage or transmitting them to another process first.

The wrappers also have the possibility to influence which PMPI functions are invoked to implement the call made by the application. For example, to find problems caused by head-to-head sends a normal MPI_Send() (which may or may not complete, depending on message size and MPI implementation) can be replaced by a synchronous MPI_Ssend() (which will always block and thus reveal the problem).

Another benefit is that once a problem is found inside a wrapper, the user can go up the call stack and directly investigate the parameters of the MPI call and where they come from. Because the application might have been compiled without debug information, the first layer of MPI wrappers provided by TAC contains debug information, so at least at that level a debugger is able to display parameters. This also helps if the parameter is the result of a complex calculation which was passed directly into the MPI call without storing it in a local variable.

MPI errors can be classified along two dimensions. First, by which information is necessary to detect them:

local checks only need information available in the process itself and thus do not require additional communication between processes

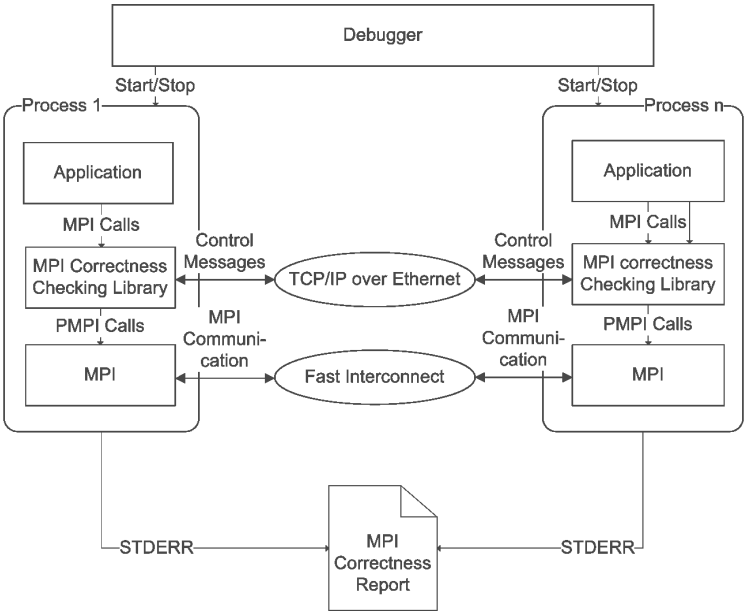

Fig. 1. Overview of TAC MPI Correctness Checking

global information from other processes is required

Second, what the effect on the application is when the problem occurs:

warning suspicious behavior is detected, but it is not clear whether it really constitutes a problem, so the application should be left running: e.g. unusually high numbers of pending requests might be due to missing checks for completion, but until the application completes it is impossible to tell for sure

error something is detected which is definitely not correct, but the application might be able to continue running, albeit with incorrect data or follow-up errors: invalid parameters in an MPI call can be ignored by simply not executing the call

fatal error the state of the application or the MPI implementation has been changed so that continuing is impossible: deadlocks fall into this category because there is no way to resolve them

The tables 4 and 3 in the appendix contain a full list of all local and global errors currently supported and their classification. As reference the table also contains the corresponding error number and name of the older Intel® Message Checker Tech Preview [1].

```
[0] WARNING: LOCAL:MEMORY:OVERLAP: warning
[0] WARNING:    New send buffer overlaps with currently active send buffer at address 0x7fbfffea24.
[0] WARNING:    Control over active buffer was transferred to MPI at:
[0] WARNING:       MPI_Isend(*buf=0x7fbfffea24, count=1, datatype=MPI_CHAR, dest=1, tag=100,\
 comm=MPI_COMM_WORLD, *request=0xaefca0)
[0] WARNING:       main (local/memory/overlap/MPI_Isend.c:62)
[0] WARNING:    Control over new buffer is about to be transferred to MPI at:
[0] WARNING:       MPI_Isend(*buf=0x7fbfffea24, count=1, datatype=MPI_CHAR, dest=2, tag=100,\
 comm=MPI_COMM_WORLD, *request=0xaefca4)
[0] WARNING:       main (local/memory/overlap/MPI_Isend.c:62)
```

Fig. 2. Local memory overlap problem, from the local/memory/overlap/MPI_Isend.c example

3 Implementation

3.1 Infrastructure

Most of the existing infrastructure for wrapping MPI calls and tracking data types, requests and communicators could be reused from the existing trace collector and now both the library for performance data collection and the checking library are compiled from the same source code.

Locating the place where MPI calls are made and from where that code is called is essential for investigating problems. In contrast to other MPI checking tools, TAC is able to unwind the call stack and uses debug information to map instruction addresses to source code. For stack unwinding both a fast frame pointer based approach is available and a slower, more reliable approach using `libunwind`. `libunwind` also works with optimized code that has no frame pointers. By default `libunwind` is used for correctness checking and for performance analysis the frame pointer approach. Debug information is read via `libdwarf`. Specifically for correctness checking the reading of symbol information via `libelf` and tracking of shared library names was added.

To complement the information about where an MPI call is made, the checking library also generates a string which represents the function name with all its parameters. Together this forms a *call site* in a *problem report*.

A problem report[1] like the one in Fig. 2 always follows the same format:

```
<error type>: <severity level>
  <error description (possibly with line breaks)>
  [<introduction for call site (no line breaks)>:
     <MPI call string>?
     <call stack>?]*
```

[1] For this paper the reports were pruned and long lines were wrapped manually: normally files are listed with an absolute path and there may be call stack entries above the main() function.

Several different places in the checking library detect errors and prepare a semantically rich reports, but they all use this format and then hand an internal representation over to one routine which handles all reports in a consistent way:

- counting reports by type
- suppressing reports if reported too often
- keeping statistics about error reports
- printing of the report to stderr
- alerting a debugger by calling a specific function in which a debugger can set a breakpoint
- aborting if necessary (depends on severity, report counters, configuration)

3.2 Local Checks

Because the checking library is meant to be used together with Intel® MPI and Intel MPI already detects out of bound parameters and invalid handles, these checks do not have to be reimplemented in the checking library. These problems detected inside the MPI implementation are always returned back to the wrappers and then converted into a normal report because the error handling policy is set to `MPI_ERRORS_RETURN`, even if the application tries to install its own handler.

The checking library also installs signal handlers for those signals that would otherwise just abort the application. In contrast to the application's MPI error handler which is always ignored, the application's signal handlers are left in place because premature exits caused by these handlers are detected via an `atexit()` callback. This behavior is configurable.

On top of the Intel MPI parameter checking the checking library itself also tracks active requests, communicators and data types so that it has additional information not available otherwise, for example where these entities were created. This improves the reports that involve such entities and allows printing warnings about excessive resource usage or leaks (Fig. 3). This leak report can also be generated at runtime to handle those cases where a leak or some other problem prevents reaching `MPI_Finalize()`. It lists the most frequent places where entities are leaked and truncates the report after a configurable number of entries to prevent excessively long reports.

The additional information kept by the checking library for data types also includes an internal representation of the full type map. The `MPI_Type_commit()` wrapper extracts this information from MPI by calling `MPI_Type_get_envelope()` and `MPI_Type_get_contents()`. Given that information, the checking library is able to iterate over the content of buffers and uses this to implement several additional checks:

- tracking which memory regions are currently owned by MPI and therefore must not be reused by the application in other MPI calls
- checksumming read-only buffers to detect illegal modifications while the buffer is owned by MPI
- detecting transmission problems by comparing sender and receiver checksums in global checks

```
[0] WARNING: LOCAL:DATATYPE:NOT_FREED: warning
[0] WARNING:    When calling MPI_Finalize() there were unfreed user-defined datatypes:
[0] WARNING:    12 in this process.
[0] WARNING:    This may indicate that resources are leaked at runtime.
[0] WARNING:    To clean up properly MPI_Type_free() should be called for
[0] WARNING:    all user-defined datatypes.
[0] WARNING:    1. 3 times:
[0] WARNING:       MPI_Type_vector(count=1, blocklen=10, stride=10, old_type=MPI_CHAR,\
 *newtype=...)
[0] WARNING:       main (local/datatype/not_freed.c:81)
[0] WARNING:    2. 1 time:
[0] WARNING:       MPI_Type_struct(count=1, *blocklens=0x7fbfffe974, *indices=0x7fbfffe948,\
 *old_types=0x7fbfffe978, *newtype=0x7fbfffe970)
[0] WARNING:       main (local/datatype/not_freed.c:94)
...
[0] WARNING:    5. 1 time:
[0] WARNING:       MPI_Type_hvector(count=1, blocklen=2, stride=2, old_type=MPI_CHAR,\
 *newtype=0x7fbfffe970)
[0] WARNING:       main (local/datatype/not_freed.c:87)
[0] WARNING:    Summary truncated at CHECK-LEAK-REPORT-SIZE 5.
```

Fig. 3. Leaked data types problem, from the local/datatype/not_freed.c example

3.3 Global Checks

Global checks are harder to implement because they require data exchange between processes. This exchange must be transparent to the application. Currently there are three different ways how that information is exchanged.

The first one is an MPI-independent communication layer which connects processes via TCP streams, accessible via an API similar to MPI, but simplified so that it can be implemented without ever buffering data. This communication layer is used by an additional background thread per process to monitor the state of the whole application.

Deadlock detection is one of its purposes: if all processes are stuck inside an MPI call for more than a certain time, then it is assumed that these calls will not complete at all because of a deadlock. Implementing this heuristic is considerably easier than correctly analyzing what each process is waiting for to identify the cycle which constitutes the deadlock. However, just one process which uses busy polling to check for progress defeats this heuristic. Therefore there is also a warning which is triggered when the average blocking time in MPI exceeds another, much larger threshold. The drawback is that this warning can also be triggered by correct programs with a load balance problem.

Some deadlocks will always occur because of a true data dependency. Others, so called *potential deadlocks*, depend on MPI implementation aspects: for example, two processes sending data in both directions by first calling `MPI_Send()` and then `MPI_Recv()` (*head-to-head send*) may or may not block depending on

whether the message is buffered. Usually this will work for small messages and deadlock for larger ones. This problem is detected for all message sizes because the checking library replaces all such potentially blocking sends with a corresponding synchronous send: this send is guaranteed to wait until the recipient enters a matching receive and thus the normal deadlock detection will report the problem.

The second communication method is used for checking collective operations: because collective operations are not allowed to overlap, their communicator can be used inside the collective MPI wrapper call to exchange additional information. Note that only other collective operations are allowed: messages might interfere with application messages or requests currently pending on the communicator. The actual checks are similar to the ones described in [2], but the implementation described in that paper uses messages and therefore might break if the application overlaps collective operations with message transmission. Another improvement compared to that implementation is that one error report per faulty operation is generated which then contains information about the problem detected in each involved process in a condensed format.

Finally, for messages one extra message on a shadow communicator is sent. There are alternative approaches (using derived data types, explicit pack/unpack, modifying the MPI implementation), but they all have certain disadvantages (no support for variable amount of additional data, performance penalty, costly to implement and less flexible). Compared to those disadvantages the drawback of the extra message approach is minor: if the recipient uses non-blocking receives with MPI_ANY_SOURCE then it becomes difficult to reliably associate the application message with the corresponding extra message. There is a certain chance that messages are not matched correctly when multiple processes send at the same time, but this can be detected, so instead of producing a false positive report the checking of the affected application messages is silently skipped.

One of the important global checks, both for collective operations and messages, is the one which compares data type signatures. Mismatches lead to data transmission problems that might go completely unnoticed because MPI implementations typically do not include the sender's data type in messages and therefore do not detect if the incoming data is interpreted incorrectly. A data type signature hash code generated according to [7] is included in the extra data exchanged for messages and collective operations to detect this.

4 Case Study: HPCC

4.1 Error Analysis

Running the complete HPCC [9] benchmark, version 1.0.0, under control of the checking library revealed some problems. This section describes the steps taken to investigate and fix these problems. After setting up the benchmark normally for Intel® MPI, it was run with:

```
mpiexec -genv LD_PRELOAD libVTmc.so -n <process count> ./hpcc
```

No recompilation is necessary to switch between runs with and without checking. Having debug information in the executable helps to locate the code which calls MPI, but is not absolutely required. Without it the checking library would still display function names (if available) and the name of the executable or shared library containing it (always available).

The first run with a larger problem size produced output like the one in Fig. 4 for each process. Note that only the first ten instances of each error type were reported, because there is a configurable limit to avoid excessive output for problems that occur repeatedly. Currently this does not take the call site into account, so a problem of one kind at one place can hide the same problem at another. The problem was harmless in this case, so the benchmark continued to run. At the end another problem was found related to unfreed requests.

It is faster and easier to work with a smaller problem size during debugging. The HPCC startup code in `src/io.c` was patched so that only that benchmark was executed. Running with just 4 processes and a smaller problem size still triggered the same warnings about `RandomAccess`.

It is good practice to fix the errors at the beginning first because they might hide and/or cause ensuing problems. Currently it is only possible to turn off checking by type and not possible to selectively suppress problems by their location in the source code.

In this case fixing the reuse of the send buffer at `time_bound.c:233` was trivial: the actual content of the message was irrelevant, sending an empty message is allowed by MPI and has the same effect of telling the recipient to stop processing (patch 16). Rerunning with that fix led to reports about to the same problem in `MPIRandomAccess.c` (which seems to be a direct copy with some modifications), which was fixed the same way.

With these fixes the remaining buffer overlaps are directly related to the leaked request (Fig. 5): the buffer with address `0x747540` of an active request created at `MPI_Isend()` in `time_bound.c:222` was reused in `MPIRandomAccess.c:549` while the previous send request was still active. The source shows that both calls use the same global `LocalSendBuffer`. The request leak confirmed that the application neither deleted it, nor did it wait or check for completion. This is a potentially dangerous problem: if the application overwrites the send buffer before MPI is able to transmit it, then the recipient of the first message will not receive the data it was supposed to receive. This is not just a theoretical problem, during at least one run the checking library also found a corrupted transmission where the send buffer was the same `LocalSendBuffer` (Fig. 6).

Because the leaked request handles are stored in local variables, it seemed appropriate to add `MPI_Wait()` calls at the end of the `HPCC_Power2NodesTime()` and `Power2NodesMPIRandomAccessUpdate()` functions to ensure that the requests are completed before leaving these functions. A code review found two more similar code patterns that apparently were not active during the runs. The final patch is in Fig. 17.

```
[0] WARNING: LOCAL:MEMORY:OVERLAP: warning
[0] WARNING:    New send buffer overlaps with currently active send buffer at address 0x7fbfffdb80.
[0] WARNING:    Control over active buffer was transferred to MPI at:
[0] WARNING:       MPI_Isend(*buf=0x7fbfffdb80, count=1, datatype=MPI_LONG_LONG, dest=1, tag=1,\
 comm=MPI_COMM_WORLD, *request=0x7fbfffd924)
[0] WARNING:       HPCC_Power2NodesTime (RandomAccess/time_bound.c:233)
[0] WARNING:       HPCC_MPIRandomAccess (RandomAccess/MPIRandomAccess.c:786)
[0] WARNING:       main (src/hpcc.c:143)
[0] WARNING:    Control over new buffer is about to be transferred to MPI at:
[0] WARNING:       MPI_Isend(*buf=0x7fbfffdb80, count=1, datatype=MPI_LONG_LONG, dest=2, tag=1,\
 comm=MPI_COMM_WORLD, *request=0x7fbfffd928)
[0] WARNING:       HPCC_Power2NodesTime (RandomAccess/time_bound.c:233)
[0] WARNING:       HPCC_MPIRandomAccess (RandomAccess/MPIRandomAccess.c:786)
[0] WARNING:       main (src/hpcc.c:143)
...
[0] INFO: LOCAL:MEMORY:OVERLAP: reported 10 times, limit CHECK-SUPPRESSION-LIMIT reached =>\
 not reporting further occurrence
...
[0] WARNING: LOCAL:REQUEST:NOT_FREED: warning
[0] WARNING:    When calling MPI_Finalize() there were unfreed requests:
[0] WARNING:    2 in this process.
[0] WARNING:    This may indicate that resources are leaked at runtime.
[0] WARNING:    To clean up properly MPI_Request_free() should be called
[0] WARNING:    for each persistent request and MPI_Wait() for normal
[0] WARNING:    requests.
[0] WARNING:    1. 1 time:
[0] WARNING:       MPI_Isend(*buf=0x7475c0, count=1, datatype=MPI_LONG_LONG, dest=2, tag=2,\
 comm=MPI_COMM_WORLD, *request=0x7fbfffdd7c)
[0] WARNING:       Power2NodesMPIRandomAccessUpdate (RandomAccess/MPIRandomAccess.c:602)
[0] WARNING:       HPCC_MPIRandomAccess (RandomAccess/MPIRandomAccess.c:837)
[0] WARNING:       main (src/hpcc.c:143)
[0] WARNING:    2. 1 time:
[0] WARNING:       MPI_Isend(*buf=0x7475c0, count=2, datatype=MPI_LONG_LONG, dest=2, tag=2,\
 comm=MPI_COMM_WORLD, *request=0x7fbfffdd7c)
[0] WARNING:       HPCC_Power2NodesTime (RandomAccess/time_bound.c:222)
[0] WARNING:       HPCC_MPIRandomAccess (RandomAccess/MPIRandomAccess.c:786)
[0] WARNING:       main (src/hpcc.c:143)
```

Fig. 4. HPCC: initial output for process #0

```
[0] WARNING: LOCAL:MEMORY:OVERLAP: warning
[0] WARNING:    New send buffer overlaps with currently active send buffer at address 0x747540.
[0] WARNING:    Control over active buffer was transferred to MPI at:
[0] WARNING:       MPI_Isend(*buf=0x747540, count=2, datatype=MPI_LONG_LONG, dest=2, tag=2,\
 comm=MPI_COMM_WORLD, *request=0x7fbfffdc6c)
[0] WARNING:       HPCC_Power2NodesTime (RandomAccess/time_bound.c:222)
[0] WARNING:       HPCC_MPIRandomAccess (RandomAccess/MPIRandomAccess.c:786)
[0] WARNING:       main (src/hpcc.c:143)
[0] WARNING:    Control over new buffer is about to be transferred to MPI at:
[0] WARNING:       MPI_Isend(*buf=0x747540, count=382, datatype=MPI_LONG_LONG, dest=1, tag=2,\
 comm=MPI_COMM_WORLD, *request=0x7fbfffdc6c)
[0] WARNING:       Power2NodesMPIRandomAccessUpdate (RandomAccess/MPIRandomAccess.c:549)
[0] WARNING:       HPCC_MPIRandomAccess (RandomAccess/MPIRandomAccess.c:837)
[0] WARNING:       main (src/hpcc.c:143)
...
[0] WARNING: LOCAL:REQUEST:NOT_FREED: warning
[0] WARNING:    When calling MPI_Finalize() there were unfreed requests:
[0] WARNING:    2 in this process.
[0] WARNING:    This may indicate that resources are leaked at runtime.
[0] WARNING:    To clean up properly MPI_Request_free() should be called
[0] WARNING:    for each persistent request and MPI_Wait() for normal
[0] WARNING:    requests.
[0] WARNING:    1. 1 time:
[0] WARNING:       MPI_Isend(*buf=0x747540, count=17, datatype=MPI_LONG_LONG, dest=1, tag=2,\
 comm=MPI_COMM_WORLD, *request=0x7fbfffdc6c)
[0] WARNING:       Power2NodesMPIRandomAccessUpdate (RandomAccess/MPIRandomAccess.c:602)
[0] WARNING:       HPCC_MPIRandomAccess (RandomAccess/MPIRandomAccess.c:837)
[0] WARNING:       main (src/hpcc.c:143)
[0] WARNING:    2. 1 time:
[0] WARNING:       MPI_Isend(*buf=0x747540, count=2, datatype=MPI_LONG_LONG, dest=2, tag=2,\
 comm=MPI_COMM_WORLD, *request=0x7fbfffdc6c)
[0] WARNING:       HPCC_Power2NodesTime (RandomAccess/time_bound.c:222)
[0] WARNING:       HPCC_MPIRandomAccess (RandomAccess/MPIRandomAccess.c:786)
[0] WARNING:       main (src/hpcc.c:143)
```

Fig. 5. HPCC: further output for process #0 after fixing the tag=1 (= FINISHED_TAG) sends

```
[7] ERROR: GLOBAL:MSG:DATA_TRANSMISSION_CORRUPTED: error
[7] ERROR:    Data was corrupted during transmission.
[7] ERROR:    Data was sent by process [4] at:
[7] ERROR:       MPI_Isend(*buf=0x7475c0, count=283, datatype=MPI_LONG_LONG, dest=7, tag=2,\
 comm=MPI_COMM_WORLD, *request=0x7fbfffdd6c)
[7] ERROR:       HPCC_Power2NodesTime (RandomAccess/time_bound.c:170)
[7] ERROR:       HPCC_MPIRandomAccess (RandomAccess/MPIRandomAccess.c:786)
[7] ERROR:       main (src/hpcc.c:143)
[7] ERROR:    Receive request activated at:
[7] ERROR:       MPI_Irecv(*buf=0x72d5c0, count=1024, datatype=MPI_LONG_LONG,\
 source=MPI_ANY_SOURCE, tag=MPI_ANY_TAG, comm=MPI_COMM_WORLD, *request=0x7fbfffdbac)
[7] ERROR:       HPCC_Power2NodesTime (RandomAccess/time_bound.c:140)
[7] ERROR:       HPCC_MPIRandomAccess (RandomAccess/MPIRandomAccess.c:786)
[7] ERROR:       main (src/hpcc.c:143)
[7] ERROR:    Data was received by process [7] at:
[7] ERROR:       MPI_Waitany(count=4, *array_of_requests=0x7fbfffdba0, *index=0x7fbfffdd88,\
 *status=0x7fbfffdc20)
[7] ERROR:       HPCC_Power2NodesTime (RandomAccess/time_bound.c:241)
[7] ERROR:       HPCC_MPIRandomAccess (RandomAccess/MPIRandomAccess.c:786)
[7] ERROR:       main (src/hpcc.c:143)
```

Fig. 6. HPCC: corrupted message originating from the reused LocalSendBuffer

After fixing `RandomAccess`, all benchmarks were enabled again. Then another problem was found in the communication bandwidth and latency benchmark: one process executes an `MPI_Send()` while every other process is waiting for it to to join an `MPI_Bcast()` (Fig. 7). That the `MPI_Bcast()` blocks the recipient of the message depends on the implementation of the operation; with other MPI implementations or different process layouts it might not block. Because the MPI standard does not guarantee that `MPI_Send()` always proceeds even for zero-sized messages and `MPI_Bcast()` also may block as it does here, the benchmark needs to be fixed. This can be done by replacing the `MPI_Send()` with an `MPI_Isend()`, as in patch 15.

5 Performance

Several different factors are expected to affect MPI performance with checking enabled and therefore this section investigates performance along several different dimensions.

- additional memory accesses to calculate checksums of communication buffers
- sending one additional, small message for each application message
- executing all message sends in rendezvous-mode without buffering small messages
- executing additional collective operations for each application operation, again with small amounts of data

```
[0] ERROR: no progress observed in any process for over 1:00 minutes, aborting application
[0] WARNING: starting premature shutdown

[0] ERROR: GLOBAL:DEADLOCK:HARD: fatal error
[0] ERROR:    Application aborted because no progress was observed for over 1:00 minutes,
[0] ERROR:    check for real deadlock (cycle of processes waiting for data) or
[0] ERROR:    potential deadlock (processes sending data to each other and getting blocked
[0] ERROR:    because the MPI might wait for the corresponding receive).
[0] ERROR:    [0] no progress observed for over 1:00 minutes, process is currently in MPI call:
[0] ERROR:       MPI_Send(*buf=0x75f04c0, count=0, datatype=MPI_BYTE, dest=1, tag=102,\
 comm=MPI_COMM_WORLD)
[0] ERROR:       cross_ping_pong_set (src/bench_lat_bw_1.5.2.c:583)
[0] ERROR:       cross_ping_pong_controlled (src/bench_lat_bw_1.5.2.c:766)
[0] ERROR:       bench_lat_bw (src/bench_lat_bw_1.5.2.c:1188)
[0] ERROR:       bench_lat_bw_print (src/bench_lat_bw_1.5.2.c:1309)
[0] ERROR:       main_bench_lat_bw (src/bench_lat_bw_1.5.2.c:1387)
[0] ERROR:       main (src/hpcc.c:251)
[0] ERROR:    [1] no progress observed for over 1:01 minutes, process is currently in MPI call:
[0] ERROR:       MPI_Bcast(*buffer=0x912cf50, count=0, datatype=MPI_BYTE, root=3, comm=MPI_COMM_WORLD)
[0] ERROR:       cross_ping_pong_set (src/bench_lat_bw_1.5.2.c:586)
[0] ERROR:       cross_ping_pong_controlled (src/bench_lat_bw_1.5.2.c:766)
[0] ERROR:       bench_lat_bw (src/bench_lat_bw_1.5.2.c:1188)
[0] ERROR:       bench_lat_bw_print (src/bench_lat_bw_1.5.2.c:1309)
[0] ERROR:       main_bench_lat_bw (src/bench_lat_bw_1.5.2.c:1387)
[0] ERROR:       main (src/hpcc.c:251)
[same MPI_Bcast() callstack for processes #2 and #3]
```

Fig. 7. HPCC: potential deadlock triggered by blocking in MPI_Send()

kind of communication involved different point-to-point calls, different collective operations

configuration of checking checks can be enabled and disabled selectively and there are different ways to generate stack back traces; enabling more tests will incur a higher overhead

number of processes to test scalability

process placement affects whether communication is between nodes or inside nodes

The results are given as slowdown, i.e. the time required for a run with checking enabled in a certain configuration divided by the time required for the same problem without checking. The set of nodes assigned by the job scheduler had an effect because inter-node communication latencies were different, therefore care was taken to only compare results generated on the same set of nodes in the same job.

The influence of the different factors listed above was investigated by running the Intel® MPI Benchmarks [3] `PingPong` (blocking send/receive), `PingPing` (non-blocking sends), `Sendrecv` (data exchange in a one-way ring implemented with `MPI_Sendrecv()`), `Bcast` and `Allgatherv`. Checking collective operations requires one additional `MPI_Allreduce()` to ensure that the operation and some common parameters match, usually one gather or scatter operation to exchange data type and checksum information, one `MPI_Allreduce()` to check whether any process found a problem before executing the actual operation and–if transmission error checking is enabled–two more `MPI_Allreduce()\verb` calls to check for correction execution afterwards (these two could be rolled into one call).

The choice of MPI benchmarks covers the effect on different point-to-point and collective calls. Using different configurations covers the effect of the different checks and how stack back traces are generated. To limit the number of combinations which had to be run, each configuration simplifies the work that the checking library has to do:

default: all checks enabled; reliable, but slow stack unwinding

fast stack unwinding: all checks enabled; faster stack unwinding which relies on frame pointers

```
PCTRACE-FAST ON
```

no stack unwinding: all checks enabled; stack unwinding disabled

```
PCTRACE OFF
```

no buffer content checks: all checks which require reading buffer content before and after communication disabled; stack unwinding disabled

```
CHECK LOCAL:MEMORY:ILLEGAL_MODIFICATION OFF
CHECK GLOBAL:MSG:DATA_TRANSMISSION_CORRUPTED OFF
CHECK GLOBAL:COLLECTIVE:DATA_TRANSMISSION_CORRUPTED OFF
```

```
# Note that wild cards can be used to match checks,
# this has the same effect as the two previous entries:
CHECK GLOBAL:*:DATA_TRANSMISSION_CORRUPTED OFF

PCTRACE OFF
```

no potential deadlock detection: same as before, but also no check for potential deadlocks

```
CHECK LOCAL:MEMORY:ILLEGAL_MODIFICATION OFF
CHECK GLOBAL:*:DATA_TRANSMISSION_CORRUPTED OFF
CHECK GLOBAL:DEADLOCK:POTENTIAL OFF

PCTRACE OFF
```

The version of TAC used in all benchmarks was the official 7.0.1 release and the version of Intel MPI 3.0 build 021. The cluster had 256 nodes with two Intel® Xeon® 5100 processors per node and two cores per processor. Infiniband was used as high-speed interconnect. For benchmarks involving two processes, these two processes were started on different nodes. For all other benchmarks the number of nodes was varied from 1 to 128 with four processes per node, usually placed so that processes with ranks #0–#3 ran on the first node, #4–7 on the second, etc. With this configuration adjacent processes in the `Sendrecv` benchmark can directly communicate through shared memory. To simulate the case where all processes are forced to communicate via the network, the slowdown was also investigated with processes placed in a round-robin fashion.

5.1 Overhead depending on Configuration

Figure 8 shows the slowdown measured for the `PingPong` benchmark when using the different configurations described above. Note that it was necessary to use a logarithmic y-axis because the high overhead for slow stack unwinding caused a slowdown of 100 times for small messages, which is 10 times slower than using the fast stack unwinding.

For an MPI operation where the MPI call and the correctness checking itself are already costly the contribution of the stack unwinding is less pronounced but still considerable, like for `Allreduce` in Fig. 9. In that comparison the less costly configuration without stack unwinding has a higher overhead for small message sizes than with fast unwinding; this anomaly indicates that MPI performance varied between different invocations. Only one run was executed during benchmarking. Running multiple times and averaging might have produced better results, but as the results were mostly stable already with one run that was not done.

There are ideas how the performance penalty for reliable stack unwinding can be avoided; until then it is recommended to compile with frame pointers enabled and configure the checking library to use the faster method.

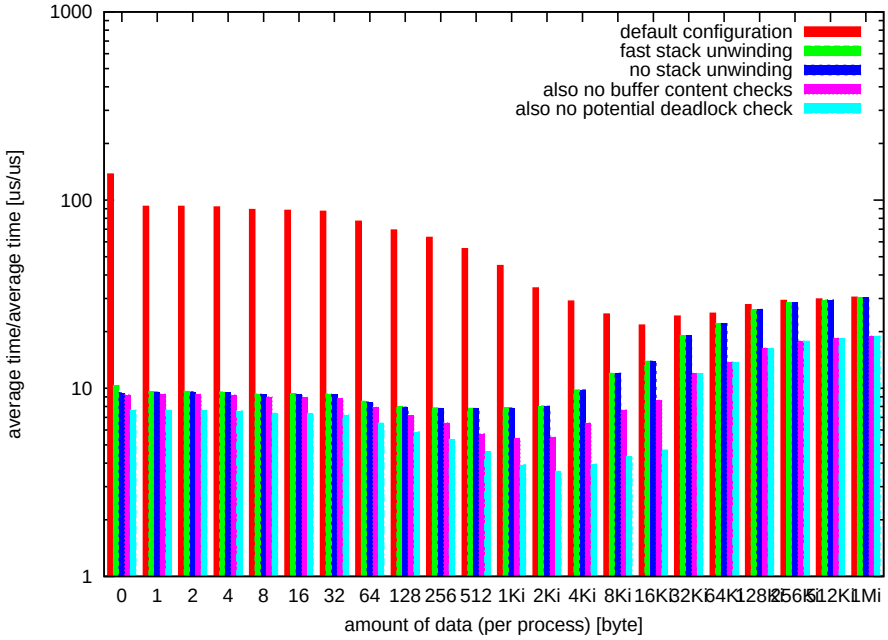

Fig. 8. Slowdown for different configurations (PingPong between 2 nodes)

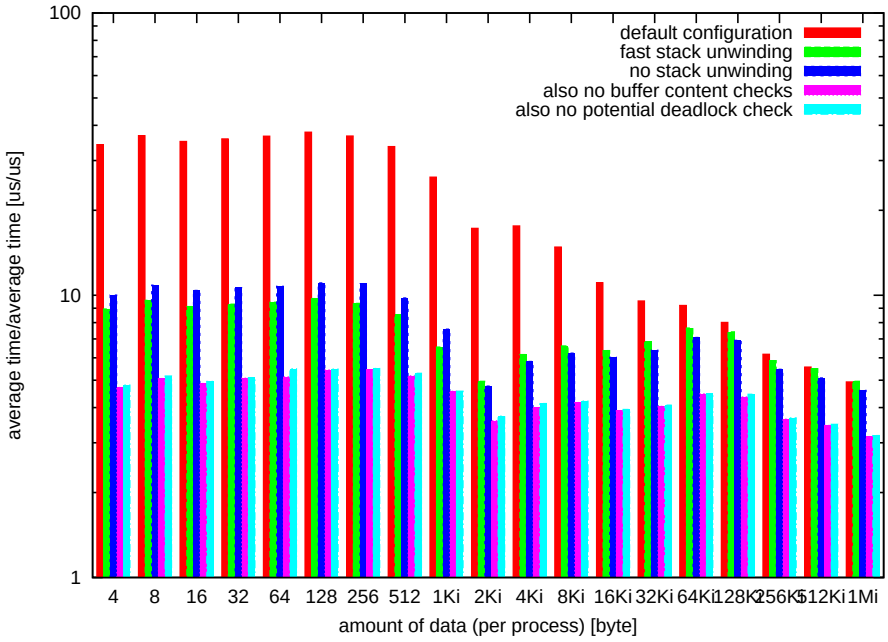

Fig. 9. Slowdown for different configurations (Allreduce, 8 processes, 2 nodes)

The other configuration options do not have the same dramatic effect. For very large message sizes disabling the buffer checking should have a more noticeable effect, but this is only visible for medium-sized messages. Also the overhead should go down as the message size increases, but instead it increases again at the end. Profiling the checking library showed that this was due to overlap checking inside the receive buffer. The initial, straight-forward implementation of that feature has an unnecessarily high runtime complexity so that it completely dominates the overhead for large messages; rewriting it should improve performance considerably. This check could not be turned off either and thus prevented running with another less costly configuration. This will also be changed in the next version.

Turning of potential deadlock detection only has an effect for those message sizes where the MPI implementation itself does not yet use synchronous sends. For messages larger than 262144 bytes Intel® MPI itself switches to synchronous sends so that from then one there is no difference between the last two configurations. Potential deadlock detection does not affect performance of collective calls at all.

5.2 Effect on Different MPI Calls and Scalability

As seen in the previous section, the most interesting configuration is the "fast stack unwinding" one because it leaves all checks enabled and can be used in practice without unfavorable effects as long as the application was compiled with frame pointers; otherwise checking still works, but might report incomplete callstacks.

This section investigates the effect of this configuration across different combinations of chunk size, number of processes and calls. The `PingPing` test only involves two processes, so Fig. 10 shows the slowdown for `Sendrecv` instead. Note that the y-axis with the process count is reversed because low process counts typically incur the higher overhead and thus would obscure the graph if they were plotted at the front.

The slowdown for `Sendrecv` is overall higher than for `PingPong` because the wrapper for `MPI_Sendrecv()` must replace one optimized call with several different calls to do the checking. The slowdown is almost constant for all combinations of message size and process count, with some variations for 8 processes; perhaps that run was disturbed more than the others. For the round-robin process layout the baseline run of `Sendrecv` is much slower as soon as more than one node is involved. The checking overhead is not affected to the same extend, therefore the slowdown in Fig. 11 is actually lower compared to the consecutive layout in 10.

`Bcast` (Fig. 12) again shows an anomaly for 8 processes; this needs further investigations. The high slowdown of `Bcast` with small message sizes and high number of processes is caused by the additional `MPI_Allreduce()`: without those the original `MPI_Bcast()` is implemented using non-blocking sends and multiple different broadcast operations overlap in time, leading to a very efficient execution of multiple broadcasts in the same direction. With checking each broadcast

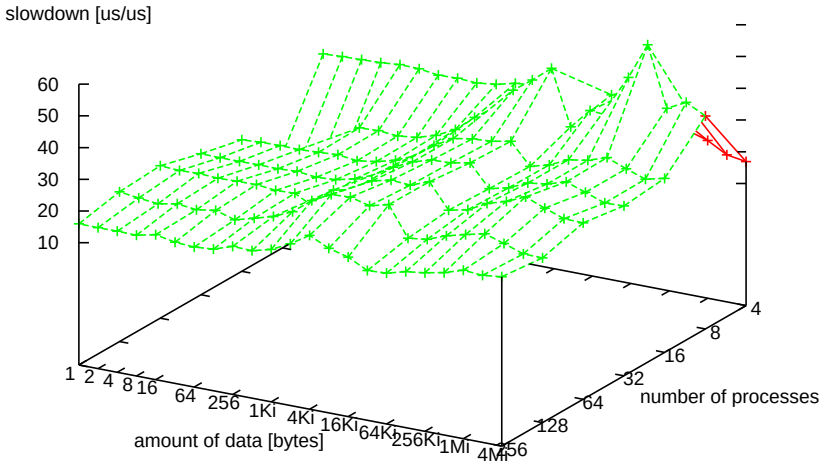

Fig. 10. Slowdown for Sendrecv (consecutive process layout, fast stack unwinding)

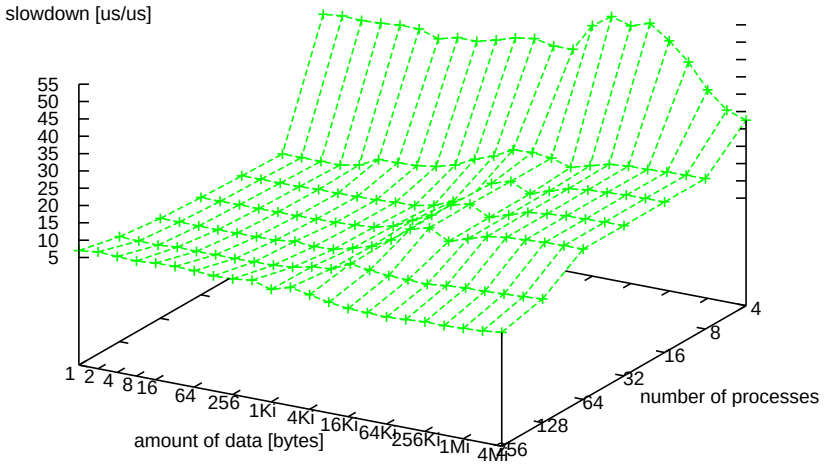

Fig. 11. Slowdown for Sendrecv (round-robin process layout, fast stack unwinding)

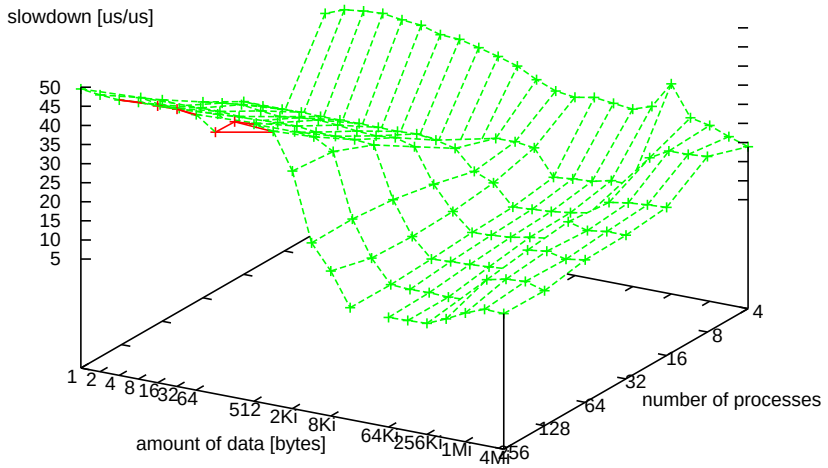

Fig. 12. Slowdown for Bcast (consecutive process layout, fast stack unwinding)

operations blocks because the checking library has to move data to and from all processes.

In the figure for `Allreduce` (13) the slowdown gets close to 5 times starting with process counters ≥ 16 and message sizes $\geq$ 1Ki. Both operations show the same behavior regardless of the process layout, which is not surprising because the MPI automatically implements the involved collective calls so that they work equally well in both cases.

It is worth pointing out that in almost all cases the correctness checking overhead scales better with chunk size and process count than the implementation of the underlying MPI call and that the microbenchmarks used in this section are not representative of normal MPI applications because they continuously call MPI and/or simulate somewhat unrealistic communication patterns. In real applications the slowdown will be proportional to the number of MPI calls; application code is hardly affected apart from some minor background load for deadlock monitoring. Therefore the actual slowdown for specific applications depends largely on how often they call MPI. The next section looks at the slowdown of HPCC to give a feeling how more complex MPI code is affected.

5.3 HPCC Slowdown

HPCC was run with different process counts with and without checking, using the fast stack unwinding configuration and both process layouts. HPCC was com-

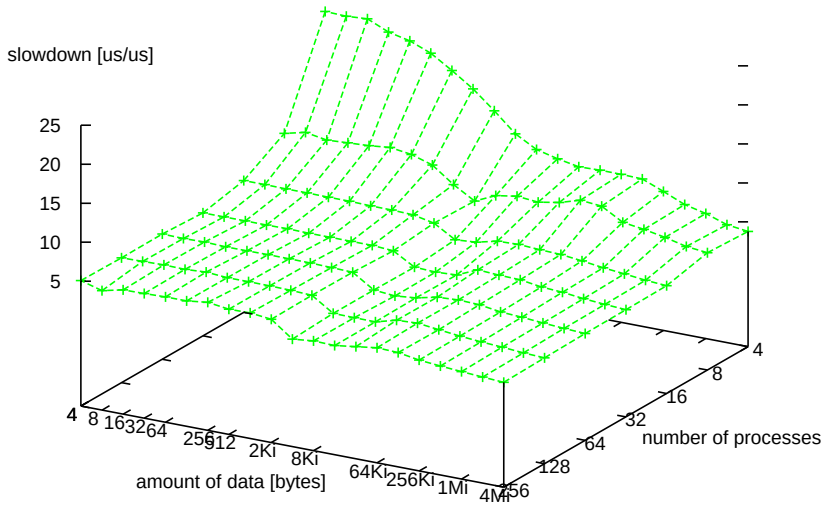

Fig. 13. Slowdown for Allreduce (consecutive process layout, fast stack unwinding)

Table 1. HPCC configuration

Processes (4 per node)	Ns	Ps	Qs
8	40000	2	4
16	60000	4	4
32	80000	4	8
64	100000	8	8
128	120000	8	16

piled with frame pointers and full optimization. Because error reporting takes unpredictable amounts of time (part of it is an intentional random delay to flush the output), the problems mentioned above were fixed before compilation by applying the patches in the appendix. The problem sizes and the process grid were adapted to match the number of processes (Tab. 1). With these configurations the baseline run took between half an hour and an hour for all benchmarks.

This section investigates the overall slowdown for complete benchmarks, i.e. including their setup phase. For that the different performance numbers generated by each benchmark were ignored and only the wallclock times between start and end of each benchmark were used. Figure 14 shows the slowdown for the consecutive process layout. The job with 64 processes did not run in time to be included in this paper.

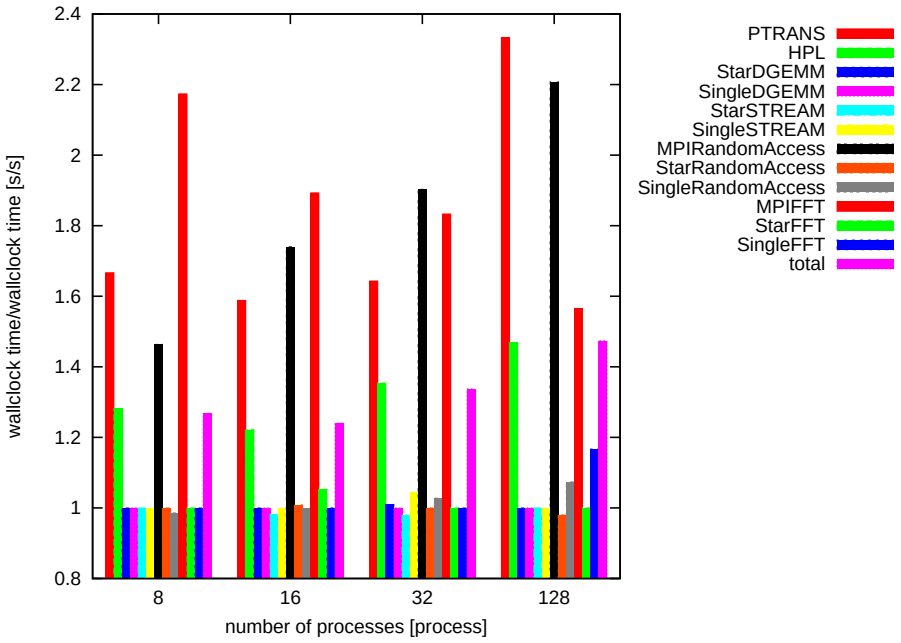

Fig. 14. Slowdown for HPCC (consecutive process layout, fast stack unwinding)

The `LatencyBandwidth` was excluded from the plot because its slowdown is considerably higher than for the other benchmarks: 20.2x slower for 8 processes, 11.9x for 16, 4.9x for 32 and 2.2x for 128. Thus for `LatencyBandwidth` and also the `MPIFFT` benchmark the slowdown decreases for higher process counts, whereas it increases for `PTRANS`, `HPL` and `MPIRandomAccess`. Overall the slowdown increases moderately from 1.3x to slightly less than 1.5x.

For the round-robin process placement the results are very similar and thus not presented in another diagram. For 128 processes the `MPIRandomAccess` triggered a warning about using more than 100 concurrently active requests, most of them created in `RandomAccess/time_bound.c:233`. For reasons that were not investigated further this did not occur when laying out processes consecutively. The threshold for printing warnings at 100, 200, etc. requests is configurable; perhaps this default value (and some others) should be proportional to the number of processes.

6 Related Work

The taxonomy of errors and the importance of detecting them is based on the experience with the Intel® Message Checker prototype, in particular the user survey described in [1]. The post-mortem trace analysis approach of the Intel

Message Checker was replaced with an online analysis because that is more scalable and better supports error analysis with an interactive debugger.

NEC MPI/SX ([10]) and MPICH2 ([2]) collectives checking, MARMOT ([5], [6]), Umpire ([11]) and MPI-CHECK ([8]) use the same online approach. MPI-CHECK also includes a compiler component to check for some errors at the source level. Apart from MARMOT and Umpire all of them use distributed checking, just like TAC correctness checking.

Table 2 is an updated version of the same table in [10], taking into account more recent publications and adding information about TAC correctness checking. What sets TAC correctness checking apart from all other tools is that it covers all the important MPI-1 checks and pays special attention to usability: it is the only tool which provides special support for investigating problems interactively with a debugger and that uses debug information, function names and stack back traces to provide problem reports that directly map to the source code.

7 Conclusion

TAC correctness checking, available from http://www3.intel.com/cd/software/products/asmona/eng/306321.htm addresses all of the major concerns for MPI developers in an easy-to-use tool. It has found problems in HPCC that did not show up before. The initial release focused on correct error detection, not performance. This paper identified some obvious ways to improve performance which will be addressed in future releases. For non-MPI intensive applications the slowdown is already below a factor of 2 times. Scalability for high process counts and large data volumes is very good.

Another aspect that could be improved is more fine-grained suppression of specific problems. The tool should already never report false positives, but some of its warnings may refer to problems that cannot be fixed or are of low priority. Currently it is only possible to disable checks completely.

The potential deadlock in HPCC also demonstrates another area for improvement: some users will want to know about potential deadlocks to ensure maximum portability, others might only be interested in problems which really occur with specific MPI implementations. “Checking profiles” could be added to enable just the desired checks.

8 Acknowledgements

We thank Bob Kuhn for starting the MPI Correctness Checking work at Intel and pursuing it together with Jay DeSouza, Victor Samofalov, Sergey Zheltov, Stanislav Bratanov and several other colleagues. The experience gathered while collaborating with them on the Intel® Message Checker has been instrumental for kick-starting the development of TAC Correctness Checking.

For TAC Correctness Checking we thank those who believed in it in 2006 and gave permission to go ahead with its implementation, in particular Bill Magro

Table 2. Key characteristics of different MPI verification approaches

	architecture	collective
NEC MPI/SX	distributed	intra & inter
MPICH2	distributed	intra
MARMOT	centralized (distributed memory)	intra
Umpire	centralized (shared memory)	intra
MPI-CHECK	distributed, instrumentation	intra
MC	tracefile (offline)	intra
TAC	distributed	intra

	point-to-point checking	PMPI
NEC MPI/SX	deadlock	available
MPICH2	data type	used
MARMOT	deadlock	used
Umpire	deadlock, buffer	used
MPI-CHECK	deadlock	used
MC	deadlock, buffer, data type	used
TAC	deadlock (hard + potential), buffer, data type	used

	portable	opaque objects
NEC MPI/SX	no (NEC MPI)	yes
MPICH2	yes	no
MARMOT	yes	yes
Umpire	limited (SMP only)	no
MPI-CHECK	limited (Fortran only)	no
MC	yes (preview for Intel MPI & MPICH)	some
TAC	yes (available for Intel® MPI)	some

	data type checking	MPI-2
NEC MPI/SX	setup of file view	full
MPICH2	communication (hash)	partially
MARMOT	construction	partially
Umpire	no	no
MPI-CHECK	no	no
MC	communication (partly)	no
TAC	communication (full)	no

and Karl Solchenbach. The discussions with colleagues like Alexander Supalov, Georg Bisseling, Werner Krotz-Vogel, Klaus Dieter-Oertel, Hans Plum, Heinrich Bockhorst and Jim Cownie about usability and MPI semantic have been very helpful.

Victor Shumilin joined the TAC Correctness Checking project as a co-developer and has been an invaluable help for getting it released in time as a stable and reliable product, together with all the application engineers who tried out internal versions with different ISV codes. Besides some colleagues already mentioned above, Tim Prince, Scott McMillan, Henry Gabb and Jonathan Anspach must be mentioned here.

Julien Langou and George Bosilca were so kind to publish their data type hashing code [7] and allow reuse in closed-source applications. The open source `libunwind` is another reused component with various contributors outside and inside Intel (Andrey Veskov).

References

[1] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with Intel®Message Checker. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 78–82, New York, NY, USA, 2005. ACM Press.

[2] Chris Falzone, Anthony Chan, Ewing Lusk, and William Gropp. Collective Error Detection for MPI Collective Operations. In *Proc. of the 12th European PVM/MPI Users' Group meeting, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science 3666*, pages 138–147. Springer, 2005. http://www.mcs.anl.gov/ gropp/bib/papers/2005/collective-checking.pdf.

[3] Intel® MPI Benchmarks 3.0. http://www.intel.com/cd/software/products/asmona/eng/307696.htm#mpibenchmarks.

[4] Intel® Trace Analyzer and Collector. http://www.intel.com/cd/software/products/asmona/eng/306321.htm.

[5] Bettina Krammer, Matthias S. Müller, and Michael M. Resch. Runtime Checking of MPI Applications with MARMOT. In *ParCo 2005, Malaga, September 2005*. Elsevier, 2005.

[6] Bettina Krammer and Michael M. Resch. Correctness Checking of MPI One-sided Communication Using MARMOT. In *Proceedings of EuroPVM/MPI 2006, Lecture Notes in Computer Science Vol. 4192*, pages 105–114. Springer, 2006.

[7] Julien Langou, George Bosilca, Graham E. Fagg, and Jack Dongarra. Hash functions for datatype signatures in MPI. In *Proc. of the 12th European PVM/MPI Users' Group meeting, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science 3666*, pages 76–83. Springer, 2005. http://www.cs.utk.edu/ library/TechReports/2005/ut-cs-05-552.pdf.

[8] Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. MPI-CHECK:a tool for checking Fortran 90 MPI programs. In *Concurrency and Computation: Practice and Experience 15(2)*, pages 93–100, 2003.

[9] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. http://icl.cs.utk.edu/hpcc/, 2005.

[10] Jesper-Larsson Träff and Joachim Worringen. The MPI/SX Collectives Verification Library. In *ParCo 2005, Malaga, September 2005*. Elsevier, 2005.

[11] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *In Supercomputing (SC), 2000*, 2000. http://www.sc2000.org/proceedings/techpapr/index.htm#01.

A Patches for HPCC V1.0.0

```
--- src/bench_lat_bw_1.5.2.c.orig       2007-02-21 17:09:03.000000000 +0100
+++ src/bench_lat_bw_1.5.2.c    2007-02-21 17:22:08.000000000 +0100
@@ -442,6 +442,9 @@
   /* do the measurements */
   for (i_meas=0; i_meas < number_of_measurements; i_meas++)
   {
+    MPI_Request tokensendreq = MPI_REQUEST_NULL;
+    MPI_Status ignoredstatus;
+
     result_index = 0;
     for (client_rank=client_rank_low; client_rank <= client_rank_high;
          client_rank += client_rank_stride)
@@ -580,11 +583,12 @@
        * messages to server processes
       */
      if ((myrank == client_rank) && (client_rank < client_rank_high))
-         MPI_Send (sndbuf, 0, MPI_BYTE, client_rank + client_rank_stride,
-                   NEXT_CLIENT, MPI_COMM_WORLD);
+         MPI_Isend (sndbuf, 0, MPI_BYTE, client_rank + client_rank_stride,
+                    NEXT_CLIENT, MPI_COMM_WORLD, &tokensendreq);

      MPI_Bcast (sndbuf, 0, MPI_BYTE, client_rank_high, MPI_COMM_WORLD);
     }
+    MPI_Wait (&tokensendreq, &ignoredstatus);
     number_of_results = result_index;
   }
```

Fig. 15. The code should handle the case that MPI_Send() and MPI_Bcast() block

```
--- ./RandomAccess/time_bound.c.orig    2006-12-05 11:06:38.000000000 +0100
+++ ./RandomAccess/time_bound.c 2007-02-20 16:00:38.000000000 +0100
@@ -230,7 +230,7 @@
    for (proc_count = 0 ; proc_count < NumProcs ; ++proc_count) {
      if (proc_count == MyProc) { finish_req[MyProc] = MPI_REQUEST_NULL; continue; }
      /* send garbage - who cares, no one will look at it */
-     MPI_Isend(&Ran, 1, INT64_DT, proc_count, FINISHED_TAG,
+     MPI_Isend(&Ran, 0, INT64_DT, proc_count, FINISHED_TAG,
                MPI_COMM_WORLD, finish_req + proc_count);
    }

@@ -513,7 +514,7 @@
   for (proc_count = 0 ; proc_count < NumProcs ; ++proc_count) {
     if (proc_count == MyProc) { finish_req[MyProc] = MPI_REQUEST_NULL; continue; }
     /* send garbage - who cares, no one will look at it */
-    MPI_Isend(&Ran, 1, INT64_DT, proc_count, FINISHED_TAG,
+    MPI_Isend(&Ran, 0, INT64_DT, proc_count, FINISHED_TAG,
               MPI_COMM_WORLD, finish_req + proc_count);
   }

--- RandomAccess/MPIRandomAccess.c.orig 2007-02-21 14:47:42.000000000 +0100
+++ RandomAccess/MPIRandomAccess.c      2007-02-21 14:48:33.000000000 +0100
@@ -346,7 +346,7 @@
   for (proc_count = 0 ; proc_count < NumProcs ; ++proc_count) {
     if (proc_count == MyProc) { finish_req[MyProc] = MPI_REQUEST_NULL; continue; }
     /* send garbage - who cares, no one will look at it */
-    MPI_Isend(&Ran, 1, INT64_DT, proc_count, FINISHED_TAG,
+    MPI_Isend(&Ran, 0, INT64_DT, proc_count, FINISHED_TAG,
               MPI_COMM_WORLD, finish_req + proc_count);
   }

@@ -610,7 +610,7 @@
   for (proc_count = 0 ; proc_count < NumProcs ; ++proc_count) {
     if (proc_count == MyProc) { finish_req[MyProc] = MPI_REQUEST_NULL; continue; }
     /* send garbage - who cares, no one will look at it */
-    MPI_Isend(&Ran, 1, INT64_DT, proc_count, FINISHED_TAG,
+    MPI_Isend(&Ran, 0, INT64_DT, proc_count, FINISHED_TAG,
               MPI_COMM_WORLD, finish_req + proc_count);
   }
```

Fig. 16. Avoid reuse of Ran in multiple simultaneous messages

```
--- RandomAccess/time_bound.c.2 2007-02-21 15:41:05.000000000 +0100
+++ RandomAccess/time_bound.c   2007-02-21 15:43:03.000000000 +0100
@@ -293,6 +293,7 @@
   MPI_Cancel(&inreq);
   MPI_Wait(&inreq, &ignoredStatus);
 #endif
+  MPI_Wait(&outreq, &ignoredStatus);

 /* end multiprocessor code */
 }
@@ -575,6 +576,7 @@
   MPI_Cancel(&inreq);
   MPI_Wait(&inreq, &ignoredStatus);
 #endif
+  MPI_Wait(&outreq, &ignoredStatus);

   /* end multiprocessor code */
 }
--- RandomAccess/MPIRandomAccess.c.2    2007-02-21 15:40:29.000000000 +0100
+++ RandomAccess/MPIRandomAccess.c      2007-02-21 15:46:13.000000000 +0100
@@ -399,6 +399,7 @@
   MPI_Cancel(&inreq);
   MPI_Wait(&inreq, &ignoredStatus);
 #endif
+  MPI_Wait(&outreq, &ignoredStatus);

   /* end multiprocessor code */
 }
@@ -662,6 +663,7 @@
   MPI_Cancel(&inreq);
   MPI_Wait(&inreq, &ignoredStatus);
 #endif
+  MPI_Wait(&outreq, &ignoredStatus);

   /* end multiprocessor code */
 }
```

Fig. 17. Check for completion of LocalSendBuffer send before leaving functions

B Supported Error Checks

Table 3. Supported Global Errors

Error Name	Type	Description
Error Name in 1.0 Tech Preview		
GLOBAL:MSG/COLLECTIVE:DATATYPE:MISMATCH	error	the type signature does not match
8. Not matched data types in operation 9. Wrong send-recv type signatures 37. Wrong data type in collective operation		
GLOBAL:MSG/COLLECTIVE:DATA_TRANSMISSION_CORRUPTED	error	data modified during transmission
28. Send-receive checksum mismatch 36. Checksum mismatch for collective operation		
GLOBAL:MSG:PENDING	warning	program terminates with unreceived messages
22. Non-paired send		
GLOBAL:DEADLOCK:HARD	fatal	a cycle of processes waiting for each other
23. Non-paired receive 31. Incomplete collective operation 40. Deadlock		
GLOBAL:DEADLOCK:POTENTIAL	fatal[a]	a cycle of processes, one or more in blocking send
41. Potential deadlock		
GLOBAL:DEADLOCK:NO_PROGRESS	warning	warning when application might be stuck
not detected		
GLOBAL:COLLECTIVE:OPERATION_MISMATCH	error	processes enter different collective operations
not detected		
GLOBAL:COLLECTIVE:SIZE_MISMATCH	error	more or less data than expected
35. Send-receive size mismatch for collective operation		
GLOBAL:COLLECTIVE:REDUCTION_OPERATION_MISMATCH	error	reduction operation inconsistent
38. Different reduction operations		
GLOBAL:COLLECTIVE:ROOT_MISMATCH	error	root parameter inconsistent
39. Wrong root process		
GLOBAL:COLLECTIVE:INVALID_PARAMETER	error	invalid parameter for collective operation
reported individually for each process		
GLOBAL:COLLECTIVE:COMM_FREE_MISMATCH	warning	MPI_Comm_free() must be called collectively
not detected		

[a] if check is enabled, otherwise it depends on the MPI implementation

Table 4. Supported Local Errors

Error Name	Type	Description
Error Name in 1.0 Tech Preview		
LOCAL:EXIT:SIGNAL	fatal	process terminated by fatal signal
1. Process abort - forced program termination 4. Incomplete function call - a call to an MPI function without return 32. Unfinished collective operation 42. Process hang-up		
LOCAL:EXIT:BEFORE_MPI_FINALIZE	fatal	process exits without calling MPI_Finalize()
2. Abnormal process end (abend) 4. Incomplete function call - a call to an MPI function without return 42. Process hang-up		
LOCAL:MPI:CALL_FAILED	depends on MPI and error	MPI itself or wrapper detects an error
3. MPI uninitialized: call to MPI functions before MPI_Init() 5. Wrong function parameters 6. Error return code for function 7. Unknown data type 14. Repeated buffer attach 15. Repeated buffer detach 16. No attached buffer for point 17. Detach of unattached buffer 25. Wrong point operation 26. Wrong peer process 30. Wrong size in send-receive operation		
LOCAL:MEMORY:OVERLAP	warning	multiple MPI operations are started using the same memory
10. Overlapped elements in derived data type for receive 24. Overlapping buffers error in Sendrecv 27. Overlapping buffers error with previously started point operation 33. Buffers overlapping error with previous started point-to-point operation 34. Send and receive buffers overlapping in collective operation		
LOCAL:MEMORY:ILLEGAL_MODIFICATION	error	data modified while owned by MPI
21. Wrong checksum at start and end of non-blocking point		
LOCAL:MEMORY:INACCESSIBLE	error	buffer given to MPI cannot be read or written
segfault and incomplete call were reported		
LOCAL:REQUEST:ILLEGAL_CALL	error	invalid sequence of calls
not detected		
LOCAL:REQUEST:NOT_FREED	warning	program creates suspiciously high number of requests or exits with pending requests
11. Non-freed request - non-freed passive request 13. Non-waited request - no MPI_Test() or MPI_Wait() 23. Non-paired receive		
LOCAL:REQUEST:PREMATURE_FREE	warning	freeing an active receive request is discouraged
12. Wrong request free 19. Unfinished send 20. Unfinished receive		
LOCAL:DATATYPE:NOT_FREED	warning	program creates high number of data types
not detected		