Advanced Memory Checking for MPI Parallel Applications Using MemPin

Shiqing Fan, Rainer Keller, and Michael Resch

Abstract In this paper, we describe the implementation of memory checking functionality that is based on instrumentation tools. The combination of instrumentation based checking functions and the MPI-implementation offers superior debugging functionalities, for errors that otherwise are not possible to detect with comparable MPI-debugging tools. Our implementation contains three parts: first, a memory callback extension that is implemented on top of the `Valgrind Memcheck` tool for advanced memory checking in parallel applications; second, a new instrumentation tool was developed based on the Intel Pin framework, which provides similar functionality as `Memcheck`. It can be used in Windows environments that have no access to the Valgrind suite; third, all the checking functionalities are integrated as the so-called `memchecker` framework within Open MPI. This will also allow other memory debuggers that offer a similar API to be integrated. The tight control of the user's memory passed to Open MPI, allows us to detect application errors and to track bugs within Open MPI itself. The extension of the callback mechanism targets communication buffer checks in both pre- and post-communication phases, in order to analyze the usage of the received data, e.g. whether the received data has been overwritten before it is used in an computation or whether the data is never used. We describe our actual checks, how memory buffers are being handled internally, show errors actually found in user's code, and the performance improvement of our instrumentation.

S. Fan (✉) · M. Resch
High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, 70550 Stuttgart, Germany
e-mail: fan@hlrs.de

R. Keller
Hochschule für Technik Stuttgart, Schellingstr. 24, 70174 Stuttgart, Germany

1 Introduction

Parallel programming with the Message Passing Interface MPI [7] as a distributed memory paradigm is an error-prone process. Great effort has been put into parallelizing libraries and applications using MPI. However when it comes to maintaining software, optimizing for new hardware, or even porting codes to other platforms and other MPI implementations, developers will face additional difficulties [2]. They may experience errors due to hard-to-track interleaving dependent bugs, deadlocks due to communication characteristics, and MPI-implementation defined or even hardware dependent behavior. One class of bugs that are hard-to-track are memory errors, specifically in non-blocking and one-sided communication.

In the beginning of previous work, we introduced new implementations of the `memchecker` framework within Open MPI to check for memory problems in parallel applications [3, 11]. It extends and integrates the `Valgrind` [10] `Memcheck` tool in `memchecker` to observe communication buffers during the communication as well as user specified parameters. We also introduced a newly developed memory checking tool, MemPin, which has similar functionalities as `Memcheck` and its extension. This new tool provide more flexibilities to be integrated in the MPI libraries. It supports both Linux and Windows platforms, and more new memory check functionalities may be implemented. On the other hand, the two phase MPI communication checks, i. e. pre- and post-communication checks have been defined in previous work. Performance implications based on these checks were also discussed, which showed that the introduced overhead by the debugging feature is minimum when the user application is not running with the debugger.

`MemPin` has been integrated into Open MPI for pre- and post-communication checks. The communication buffers errors, such as accessing buffers of active non-blocking operations, writing communicated buffer before reading it, or transferring unused data are being checked and reported. This kind of functionalities would otherwise not be detectable within traditional MPI-debuggers based on the PMPI-interface. In this paper, we continue the work on `MemPin`. Details of the `MemPin` implementation will be discussed. The integration of the tool and Open MPI is based on pre- and post-communication checks for non-blocking and collective communications. We will take a closer look into how `MemPin` and Open MPI work together, as well as how new checks may be implemented. Furthermore, several MPI parallel applications will be introduced to run with our memory checking tool. Problems, bugs and critical issues in these applications, which have been found using the debugging tool, will be mainly focused on.

The structure of this paper is as follows: Sect. 2 shows the basic idea and functionalities of Intel PIN; Sect. 3 first introduces the functionalities and architecture of `MemPin`, then it gives a detailed description on the integration the tool in Open MPI libraries; Sect. 4 shows details of the pre- and post-communication checks for parallel applications; then in Sect. 5 we give more performance improvement results based on previous work; finally; in Sect. 6, we introduce a 2D heat conduction application, and discuss issues found by running the application with `MemPin`; in Sect. 7, we make a comparison with other available tools and conclude the paper with an outlook of our future work.

2 Overview of Intel Pin

Intel Pin [6] is a framework for building robust and powerful software instrumentation tools, such as profiling, performance evaluation, and error detection tools. Intel Pin is capable of analyzing and instrumenting application code, and it is easy-to-use, portable, transparent, and efficient for building instrumentation tools (Pintools) written in C/C++. The Pin framework follows the ATOM [13] model that allows the Pintools to analyze the application at the instruction level without detailed knowledge of the underlying instruction set. The framework API is designed to be platform independent in order to make the Pintools compatible across different architectures. It can provide architecture specific details when necessary. The instrumentation process is transparent as both the application and the Pintool observe the original application code. Pin uses techniques like inlining, register re-allocation, and instruction scheduling, in order to run more efficiently. The basic overhead of the Pin framework is approximately 10–20 %, and extra overhead might be caused by the Pintool.

Figure 1 shows a basic runtime architecture of Pin on Windows. It consists of three main processes, the launcher process, the server process and the instrumented process. The launcher process creates the other two processes, injects the Pin modules, and instrumented code into the instrumented process, then it waits for the process to terminate. The server process provides services for managing symbol information, injecting Pin, or communicating with the instrumented process via shared memory. The instrumented process includes the Pin Virtual Machine Monitor (VMM), the user defined Pintool library, and a copy of the application executable and libraries. The VMM is the core engine of the entire instrumented process that includes a system call emulator, event and thread dispatchers, and also a (Just In Time) JIT compiler. After Pin takes over the control of the application, the VMM coordinates its execution. The JIT compiler instruments the code and passes it to the dispatcher, which launches the execution. The compiled code is stored in the code cache. The Pin tool contains the instrumentation and analysis routines. It is a plug-in and linked with the Pin library, which allows it to communicate with the Pin VMM.

The Pin JIT compiler recompiles and instruments small chunks of binary code immediately before executing them. The modified instructions are stored in a software code cache. It allows code regions to be generated once and reused for the remainder of program execution, in order to reduce the costs of recompilation. The overhead introduced by the compilation is highly dependent on the application and workload [6].

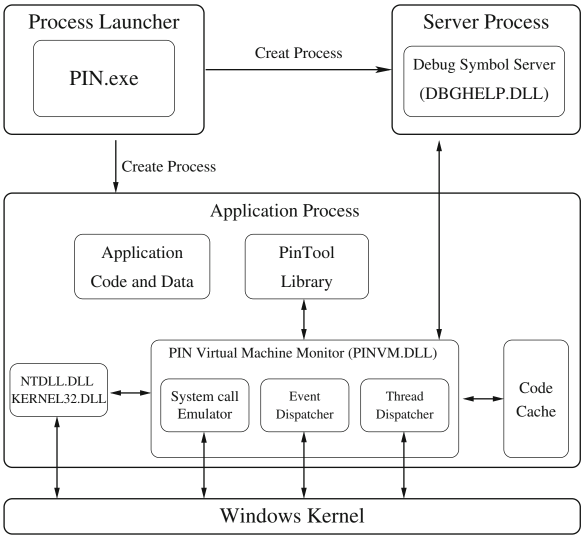


Fig. 1 Overview of program execution with an Intel Pin tool on Windows

In this paper, we will describe the implementation of `Memcheck` extension and the integration of a newly developed Pintool as the second `memchecker` component in Open MPI, which may help MPI application and Open MPI developers to track erroneous use of memory, such as reading or writing buffers of active non-blocking receive operations, writing to buffers of active one-sided get operations as well as erroneous use of communicated data, such as writing the communicated buffer before reading.

3 Design and Implementation

In previous work, we have taken advantage of an instrumentation API offered by `Memcheck` to find MPI-related hard-to-track bugs in applications (and within Open MPI). In order to allow other kinds of memory-debuggers, such as `bcheck` [1] or Totalview's memory debugging features [15], we have implemented the functionality as a module within Open MPI's Modular Component Architecture [17]. The module is therefore called `memchecker` and may be enabled with the configure-option `--enable-memchecker`.

However, that did not meet all the requirement of advanced MPI semantic memory checking, and the current functionalities of `Valgrind Memcheck` do not suffice. For example, the future MPI standard will change behavior compared to the current version of the standard in that it allows read access to send buffers of non-blocking operations. In order to detect send buffer rewrite errors, `Memcheck` has to know which memory region is readable or writable. On the other hand, checking the communicated buffer is also important, for example, writing the received buffer before reading may cause computation errors. And for performance concerns, data transferred but never used (read) might also be necessary to detect. The extensions for `Memcheck` have been implemented for this purpose.

3.1 MemPin

As Open MPI is supported on Windows platforms, a debugging tool for checking memory errors is also necessary. A new tool named `MemPin` has been designed on top of Intel Pin framework to meet this need.[1]

The `MemPin` tool uses Intel Pin's instrumentation API to provide the same callback functionalities as the `Memcheck` extension for the user application. Furthermore it may be used to perform the basic functionalities of `Memcheck`, such as make memory readable or inaccessible, but through a different approach (see Sect. 4.2). The available interfaces and descriptions are:

- `MEMPIN_RUNNING_WITH_PIN`

  Checks whether the user application is running under Pin and Pintool.
- `MEMPIN_REG_MEM_WATCH`

  Registers the memory entry for specific memory operation.
- `MEMPIN_UPDATE_MEM_WATCH`

  Updates the memory entry parameters for the specific memory operation.
- `MEMPIN_UNREG_MEM_WATCH`

  Deregisters one memory entry.
- `MEMPIN_SEARCH_MEM_INDEX`

  Returns the memory entry index from the memory address storage.
- `MEMPIN_PRINT_CALLSTACK`

  Prints the current callstack to standard output or a file.

The user may use the `MemPin` API to register memory regions with specific callback function and parameter pointers. When the user application is not running with the Pintool, all `MemPin` calls will be taken as empty macros, and add no overhead. But if running with `MemPin` tool, Pin first reads the entire executable, and all the `MemPin` calls will be replaced with the corresponding function calls that are defined inside `MemPin`. The generated instrumented codes will be then executed and `MemPin` will observe and respond to the behavior of the user application.

[1]The MemPin tool in this work is developed only targeting at Windows platforms, although it may be used under Linux too.

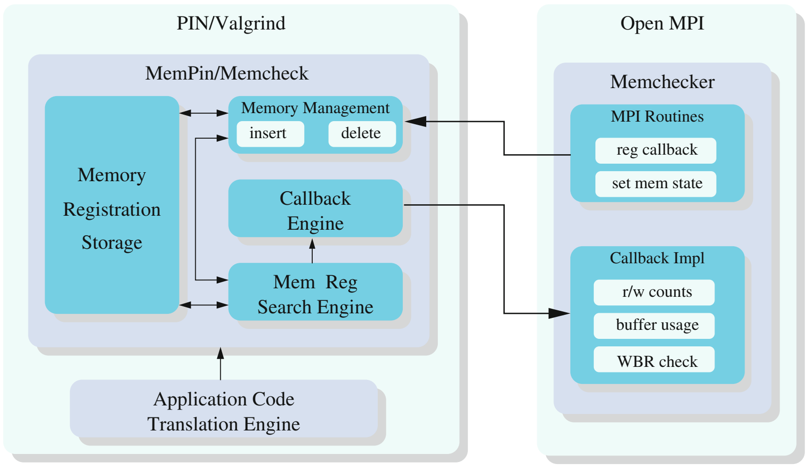


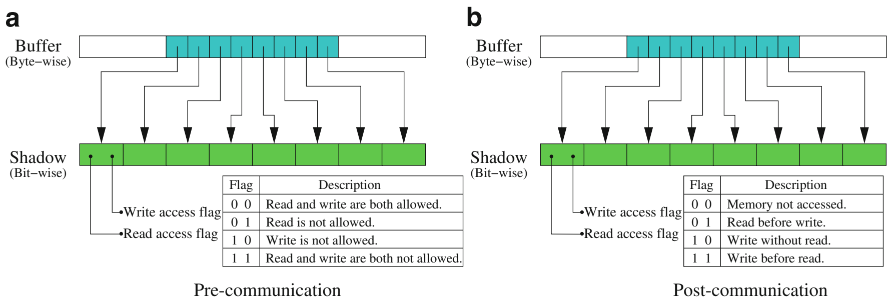
Fig. 2 Run-time structure of `MemPin`

`MemPin` uses image and trace instrumentations in user applications. The image instrumentation is done when the image is loaded. In this stage, all the `MemPin` calls used in the user application will be replaced, and the main entry function, like `main` will be instrumented for starting the trace engine and the callstack log of `MemPin`.

The next stage will mainly take care of the memory access, callback functions and the user application callstack. The trace instrumentation is analyzed according to each Basic Block (BBL), and every memory operation in the BBL is checked. When the memory is read or written, the single instruction of the memory operation is instrumented with an analysis function with memory information as operands. For generating useful information of where exactly the memory operation has happened, a callstack log engine is instrumented also in this stage. The callstack engine is implemented using a simple C++ stack structure, which stores only the necessary historical instruction addresses of the application and translates the addresses into source information when required. The new function entry address from the caller will be pushed onto the stack, and it is popped off at the end of the callee. To achieve this goal, the tail instruction of each BBL has to be analyzed. More precisely, every "call" and "return" instruction are instrumented for pushing and popping the instruction address stack.

3.2 *Integration of MemPin with Open MPI*

`MemPin` has been successfully integrated into Open MPI as an MCA component, in order to achieve the parallel memory checking discussed in previous sections. Figure 2 shows the basic architecture of `MemPin` and how it can work with



Flag	Description
0 0	Read and write are both allowed.
0 1	Read is not allowed.
1 0	Write is not allowed.
1 1	Read and write are both not allowed.

Flag	Description
0 0	Memory not accessed.
0 1	Read before write.
1 0	Write without read.
1 1	Write before read.

Fig. 3 Shadow memory for pre- and post-communication using `MemPin`. (**a**) Pre-communication. (**b**) Post-communication

Open MPI. `memchecker` component can directly communicate with `MemPin` at runtime. When a communication is initialized, `memchecker` send a request to `MemPin` to record or update the memory information. The Memory Management component is in charge of inserting, deleting or updating memory entries. An memory entry contains information of a communication buffer, including starting address, size, callback function pointer and memory operation flag. All memory entries are stored in the Memory Registration Storage, which is a multiple map data structure. The Search Engine intercepts the translated application code from Intel Pin, and find the match access according to the storage. If a match is found, it will send corresponding memory information to the Callback Engine, which then will directly call the callback function registered with the memory entry.

`MemPin` has no information about whether the memory is readable or writable. But similar functionalities for making memory readable and writable have been implemented in Open MPIusing the callback scenario of `MemPin`. In the callback function defined in Open MPI, for both pre- and post-communication checks, the same memory states Bits are used. For pre-communication checks, the two Bits of memory state are used for marking the memory readable or writable, as shown in Fig. 3a. If the first Bit is set to "1", the memory is marked as not readable. The second Bit is the writable Bit, which means a "1" is for not writable. When both Bits are set to "1", then the memory is marked as not accessible at all.

For post-communication checks, the same Bit table will be used in order to save the storage. But the two bits have different meanings, see Fig. 3b. The first Bit indicates whether the byte of memory has been read or not. The second bit is for whether the byte of memory has been written or not. The callback function will check for each registered receive buffer, whether they are read before written, otherwise reporting a Write Before Read error. Furthermore, in the `MPI_Finalize` call, all memory state Bits are checked for buffer that are not used after communication.

These two phases of memory checks in MPI communication may change automatically to the other phase. If a parallel application has several communications, whenever the communication is started, for example calling the `MPI_Isend`, the pre-communication check will be enabled. When the communication is finished, for example `MPI_Wait` is called, the post-communication check will be executed. The shadow memory for both checks does not need to be reallocated, as they have the same format but different meaning of the Bits. All the registered memory checks will be cleared in the `MPI_Finalize`, which is the end of the parallel computation.

4 Memory Checks in Parallel Application

4.1 Pre-communication Checks

In Open MPI objects such as communicators, types and requests are declared as pointers to structures. These objects when passed to MPI-calls are being immediately checked for definedness and together with `MPI_Status` are checked upon exit.[2] Memory being passed to send operations is being checked for accessibility and definedness, while pointers in receive operations are checked for accessibility, only.

Reading or writing to buffers of active, non-blocking receive operations and writing to buffers of active, non-blocking Send-operations are obvious bugs. Buffers being passed to non-blocking operations (after the above checking) are being set to undefined within the MPI-layer of Open MPI until the corresponding completion operation is issued. This setting of the visibility is being set independent of non-blocking `MPI_Isend` or `MPI_Irecv` function. When the application touches the corresponding part in memory before the completion with `MPI_Wait`, `MPI_Test` or multiple completion calls, an error message will be issued. In order to allow the lower-level MPI-functionality to send the user-buffer as fragment, the lower-layer BTLs (Byte Transfer Layers) are adapted to set the fragment in question to accessible and defined, as may be seen in Fig. 4. Care has been taken to handle derived datatypes and its implications. Complex datatypes are checked according to their definitions, which means gaps will be ignored to avoid false positive messages.

For send operations, the MPI-1 standard also defines, that the application may not access the send-buffer at all (see [7], p. 30). Many applications do not obey this strict policy, domain-decomposition based applications that communicate ghost-cells, still read from the send-buffer. To the authors' knowledge, no existing implementation requires this policy, therefore the setting to undefined on the Send side is only done when strict-checking is enabled.

For one-sided communications, MPI-2 standard defines that, any conflicting accesses to the same memory location in a window are erroneous (see [8], p. 112). If a location is updated by a put or an accumulate operation, then this location cannot be accessed by a load or another RMA operation until the updating operation is

[2]For example this showed up uninitialized data in derived objects, e.g. communicators created using `MPI_Comm_dup`.

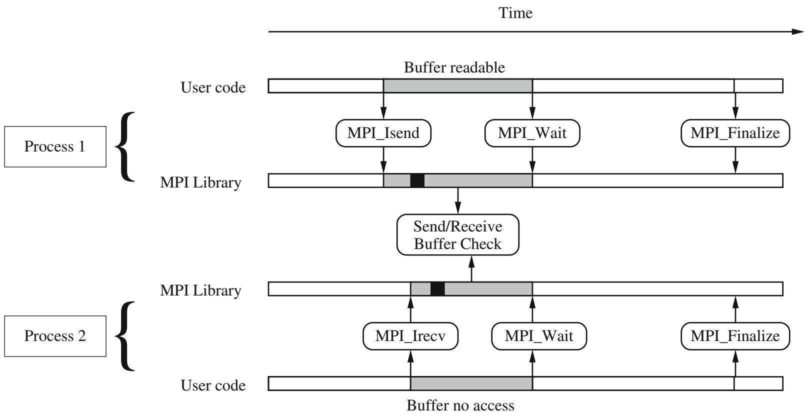


Fig. 4 An example of non-blocking communication using pre-communication checks with fragment handling to set accessibility and definedness

completed on the target. If a location is fetched by a get operation, this location cannot be accessed by other operations as well. When a synchronization call starts, the local communication buffer of an RMA call and a get call should not be updated until it is finished. User buffer of `MPI_Put` or `MPI_Accumulate`, for instance, are set not accessible when these operations are initiated, until the completion operation finished. `Valgrind` will produce an error message, if there is any read or write to the memory area of the user buffer before corresponding completion operation terminates.

In Open MPI, there are two One-sided communication modules, point-to-point and RDMA. Similar checks have been implemented for `MPI_Get`, `MPI_Put`, `MPI_Fence` and `MPI_Accumulate` in point-to-point module.

For the above communication buffer checking, the original features of `Valgrind` are used and integrated in Open MPI on Linux. On the other hand, in order to implement the same checks on Windows, the callback scenario of `MemPin` is used. When the communication starts, for example, the `MPI_Isend` and `MPI_Irecv` are called, all the communication buffers are registered, and all read and write access on the buffers will be checked whether they are legal according to the standards. If an illegal access is found, a warning message with sufficient callstack information will be generated for the user. However, in order to make the `MemPin` efficient and lightweight, memory checks for accessibility and definedness cannot be easily implemented due to complex and large shadow memory consumption.

4.2 Post-communication Checking

For more detailed memory checking, one may further implement an interface to encode read and write accesses of previously communicated data. This new

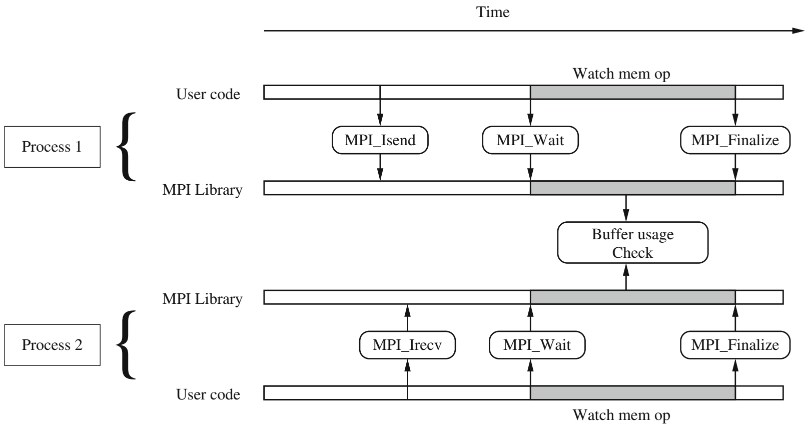


Fig. 5 An example of non-blocking communication using post-communication checks

analysis mechanism may help user applications to detect whether data has been sent needlessly, i. e. data that has been sent but might be overwritten before reading or may be never accessed in the receiving process.

More precisely, read accesses on the received data is counted as meaningful, while the first write access is not, for the communicated buffer is overwritten before any actual use. To achieve this goal, we make use of the tools introduced in Sect. 3.1, in order to notify Open MPI of the corresponding tasks based on the type of operations, i.e. read or write to the received buffer.

All the communication buffers are registered when the communication finishes. For example, in Fig. 5, when the `MPI_Wait` is called for non-blocking communication, every read and write access on the received buffer will be processed by the callback implementation in Open MPI, and a write before read is marked as illegal. In the finalization phase of the parallel application, all registered memory will be checked for whether there are communicated data but never used, i. e. whether there are buffers without a read access.

5 Performance Comparison

In previous work [12], we showed the performance when the benchmarks were not run with debuggers. The results proved that the overhead introduced by the debugging frameworks are neglectable. However, when the benchmarks are run with debuggers, there will be approximately 30–50 % slowdown.

Figure 6 shows a comparison between running with `Valgrind` and `MemPin` using `NetPIPE`. NetPIPE was run under supervision of `Valgrind` and `MemPin` respectively using two compute nodes interconnected with InfinBand on Nehalem cluster at HLRS. The latency (in Fig. 6a) of using `MemPin` is nearly 50 % less than

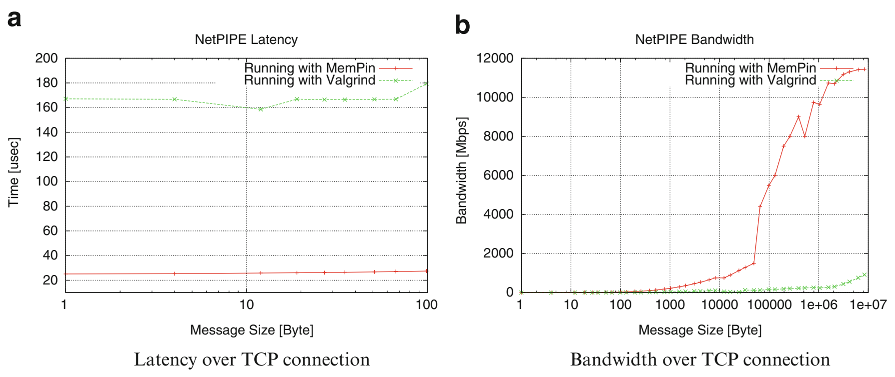


Fig. 6 NetPIPE latency (*left*) and bandwidth (*right*) comparison of Open MPI run with the memchecker framework over InfiniBand. (**a**) Latency over TCP connection. (**b**) Bandwidth over TCP connection

the latency using `Valgrind`. The difference of bandwidth between the two tests increases largely when the message size increases, as shown in Fig. 6b. Using the memchecker framework based on `MemPin` has a better performance than using framework based on `Valgrind`.

6 2D Heat Conduction Program with MemPin

During the course of development, several software packages have been tested with the `memchecker` functionality. Among them problems showed up in Open MPI itself (failed in initialization of fields of the status copied to user-space), an MPI testsuite [4], where tests for the `MPI_ERROR` triggered an error. In order to reduce the number of false positives in Infiniband networks, the `ibverbs` library of the OFED stack [14] was extended with instrumentation for buffer passed back from kernel-space.

A 2d heat conduction algorithm has been used for running with the new implemented memory checking framework on both Linux and Windows. The algorithm is based on Parallel CFD Test Case [9]. It solves the partial differential equation for unsteady heat conduction over a square domain. It was firstly run with two processes under control of `MemPin`, where warnings about 112 bytes of communicated but not used data were reported. A small amount of data on two processes may be not critical to the communication time. But when running with large number of processes with the application, it may differ.

In the 2D domain decomposition algorithm, it requires calculating elements from their horizontal and vertical neighbor elements, but the whole border element arrays are updated from neighbor sub-domain. This results that, for the border update in every sub-domain, there will be four corner elements that will never be used for

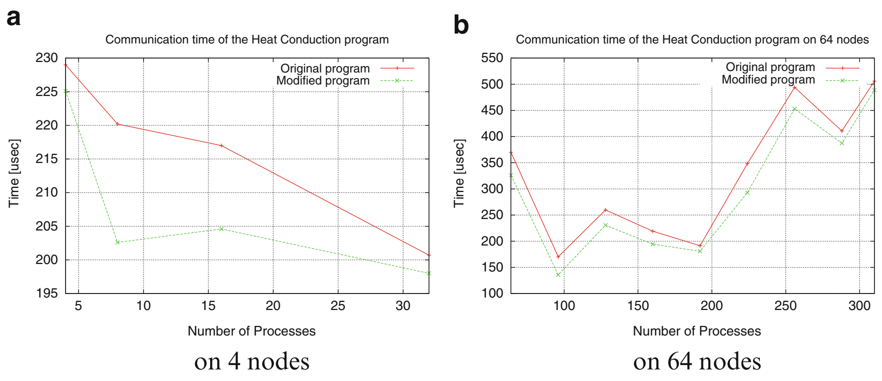


Fig. 7 Comparison of the communication time between the original and modified heat conduction program on Nehalem cluster. (**a**) On 4 nodes. (**b**) On 64 nodes

calculation (without periodic boundary condition, the virtual border elements are not taken into account in the example). This might be no harm for the calculation result of the algorithm. But when decomposing the entire problem into a large number of sub-domains, the total amount of transferred but unused data may be high, and as consequence communication might require more time.

Take a 4×4 domain decomposition as an example, where every element calculation requires horizontal and vertical neighbor elements. In this specific condition, there will be 72 elements transferred but not used (36 corner elements transferred two times). When scaling this code by doubling the number of processors used to compute this domain, the number of elements communicated but not used increases dramatically. In the case of a 8×8 domain decomposition, that has 392 elements (196 corner elements transferred two times) might not be communicated. Assuming we have a $M \times N$ domain decomposition, the total amount of such elements are described by:

$$(M-1) \times (N-1) \times 4 \times 2 \tag{1}$$

It is obvious that, the number of unnecessary communicated data grows superlinearly with the domain decomposition.

The heat program has further been tested with more processes on different number of nodes on BWGrid and Nehalem Cluster at HLRS, in order to discover the relationship between the communicated but unused corner elements and the communication performance. For the first test, the heat program was set to a $1{,}500 \times 1{,}500$ domain, and parallelized with different number of processes over four compute node (eight cores on each node) on Nehalem cluster. A modified version of the heat program was also used for the test, which does not send any unused corner elements. The average communication time and overall run time are measured based on five executions on different number of processes, as shown in Fig. 7a. When not oversubscribing the nodes (each core has no more than one process), the modified version is generally better than the original version ranging from 3 to 7 %. As we can see, communicating the corner elements of the sub-domains will indeed affect the communication time of the program.

Another test on 64 nodes was made on Nehalem cluster to start large number of processes without oversubscribing the compute node cores. The processes are assigned using round-robin algorithm among the nodes, in order to achieve a better load balancing for the simulation. Figure 7b shows the communication time of running the same simulation with different number of processes on the cluster. It presents the communication time for different number of processes (64–310). The modified version has a shorter communication time on average, which is 10 % better. The best case is even 20 % better than the original version.

The communication time does not increase of decrease linearly with the number of processes, because the domain decomposition will influence the communication efficiency. Assuming we have eight bytes data in each corner element, for a 96 processes run (12×8 decomposition), the number of border exchange is $(12 - 1) \times (8 - 1) \times 4 \times 2$, which is 616 times. This results to 4,928 bytes of communicated but never used data. For the same configuration, if running with 128 processes (16×8 decomposition), the size of each border element is halved, i. e. four bytes. But the number of border exchange is now $(16 - 1) \times (8 - 1) \times 4 \times 2$, which is 840 with 3,360 bytes in total. One may argue that the total size of transferred data is smaller with high resolution of domain decomposition, the communication speed should increase. However, this is not true. The overall communication speed is highly determined by the number of communication but not the data size that is transferred. In Open MPI, for blocking and non-blocking communication, there are two transmission protocols, i.e. Eager and Rendezvous. When the data size is small than 12 kB, the data will be sent in one package (Eager protocol). But when the data size is larger than 12 kB, the data will be divided into smaller packages (Rendezvous protocol), so there is not only one send and receive operation on this data. When the data size does not exceed the limit, the number of the communication will determine the overall communication speed. This also explains why the communication time is larger when running with 64 processes. In this case, the corner data is much larger than 12 kB, so the number of communication is doubled or even tripled.

7 Conclusion

We have presented an implementation of memory debugging features into Open MPI, using the instrumentation of the `Valgrind` and a newly developed Intel Pintool, and the performance implication of using the instrumentation with several benchmarks. This allows detection of hard-to-find bugs in MPI parallel applications, libraries and Open MPI itself [2]. Up to now, no other debugger is able to find these kinds of errors.

With regard to related work, debuggers such as Umpire [16], Marmot [5] or the Intel Trace Analyzer and Collector [2], actually any other debugger based on the Profiling Interface of MPI, may detect bugs regarding non-standard access to buffers used in active, non-blocking communication without hiding false positives of the MPI-library itself.

Acknowledgements This work was funded by project Int. EU. Grid (Interactive EUropean Grid) with EU-Contract Number 031857, by project DORII (Deployment of Remote Instrumentation Infrastructure) with EU-Contract Number 213110 and by Microsoft Technical Computing Initiative (TCI) since March 2007.

We would like to thank Julian Seward and the open source community for `valgrind`, which has proven invaluable in many software projects.

References

1. bcheck Man Page from SUN developers Website. Internet (2011). http://developers.sun.com/sunstudio/documentation/ss11/mr/man1/bcheck.1.html
2. DeSouza, J., Kuhn, B., de Supinski, B.R.: Automated, scalable debugging of MPI programs with Intel message checker. In: Proceedings of the 2nd International Workshop on Software engineering for high performance computing system applications, vol. 4, pp. 78–82. ACM Press, NY, USA (2005)
3. Keller, R., Fan, S., Resch, M.: Memory debugging of MPI-parallel Applications in Open MPI. In: G. Joubert, C. Bischof, F. Peters, T. Lippert, M. Bucker, P. Gibbon, B. Mohr (eds.) Proceedings of ParCo'07. Julich, Germany (2007)
4. Keller, R., Resch, M.: Testing the Correctness of MPI implementations. In: Proceedings of the 5th Int. Symp. on Parallel and Distributed Computing conference, pp. 291–295. Timisoara, Romania (2006)
5. Krammer, B., Mller, M.S., Resch, M.M.: Runtime checking of MPI applications with Marmot. Malaga, Spain (2005)
6. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 190–200. ACM
7. Message Passing Interface Forum: MPI: A Message Passing Interface Standard (1995). http://www.mpi-forum.org
8. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface (1997). http://www.mpi-forum.org
9. Resch, M., Sander, B., Loebich, I.: A comparison of OpenMP and MPI for the parallel CFD test case. In: Proc. of the First European Workshop on OpenMP, pp. 71–75 (1999)
10. Seward, J., Nethercote, N.: Using Valgrind to detect undefined value errors with bit-precision. In: Proceedings of the USENIX'05 Annual Technical Conference. Anaheim, CA, USA (2005)
11. Shiqing Fan, R.K., Resch, M.: Enhanced memory debugging of mpi-parallel applications in open mpi. In: 4th Parallel Tools Workshop (2010)
12. Shiqing Fan, R.K., Resch, M.: Advanced memory checking frameworks for mpi parallel applications in open mpi. In: 5th Parallel Tools Workshop (2011)
13. Srivastava, A., Eustace, A.: Atom: A system for building customized program analysis tools. pp. 196–205. ACM (1994)
14. The Open Fabrics project webpage. WWW (2007). https://www.openfabrics.org

15. Totalview Memory Debugging capabilities. WWW. http://www.etnus.com/TotalView/Memory.html
16. Vetter, J.S., de Supinski, B.R.: Dynamic Software Testing of MPI Applications with Umpire. In: Proceedings of Supercomputing (SC) (2000). http://www.sc2000.org/proceedings/techpapr/index.htm
17. Woodall, T., Graham, R., Castain, R., Daniel, D., Sukalski, M., Fagg, G., Gabriel, E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A.: Open MPI's TEG Point-to-Point Communications Methodology: Comparison to Existing Implementations. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, vol. 3241, pp. 105–111. Springer, Budapest, Hungary (2004)