MPI Application Development Using the Analysis Tool MARMOT

Bettina Krammer, Matthias S. Müller, and Michael M. Resch

High Performance Computing Center Stuttgart
Allmandring 30, D-70550 Stuttgart, Germany
{krammer,mueller,resch}@hlrs.de

Abstract. The Message Passing Interface (MPI) is widely used to write parallel programs using message passing. Due to the complexity of parallel programming there is a need for tools supporting the development process. There are many situations where incorrect usage of MPI by the application programmer can automatically be detected. Examples are the introduction of irreproducibility, deadlocks and incorrect management of resources like communicators, groups, datatypes and operators. We also describe the tool MARMOT that implements some of these tests. Finally we describe our experiences with three applications of the CrossGrid project regarding the usability and performance of this tool.

1 Introduction

The Message Passing Interface (MPI) is a widely used standard [7] to write parallel programs. The main reason for its popularity is probably the availability of an MPI implementation on basically all parallel platforms. Another reason is that the standard contains a large number of calls to solve standard parallel problems in a convenient and efficient manner. The drawback is that the MPI 1.2 standard is with 129 calls large enough to introduce a complexity that also offers the possibility to use the MPI API in an incorrect way. According to our experience there are several reasons for this: first, the MPI standard leaves many decisions to the implementation, e.g. whether or not a standard communication is blocking. Second, parallel applications get more and more complex and especially with the introduction of optimisations like the use of non-blocking communication also more error prone.

2 Related Work

Debugging MPI programs has been addressed in various ways. The different solutions can roughly be grouped in three different approaches: classical debuggers, special MPI libraries and tools.

1. Classical debuggers have been extended to address MPI programs. This is done by attaching the debugger to all processes of the MPI program. There

   are many parallel debuggers, among them the very well-known commercial debugger Totalview [1]. The freely available debugger gdb has currently no support for MPI, however, it may be used as a back-end debugger in conjunction with a front-end that supports MPI, e.g. mpigdb. Another example of such an approach is the commercial debugger DDT by streamline computing, or the non-freely available p2d2 [3,9].
2. The second approach is to provide a debug version of the MPI library (e.g. mpich). This version is not only used to catch internal errors in the MPI library, but it also detects some incorrect usage of MPI by the user, e.g. a type mismatch of sending and receiving messages [2].
3. Another possibility is to develop tools dedicated to look for problems within MPI applications. Currently three different tools are under active development: MPI-CHECK [6], Umpire [12] and MARMOT [4]. MPI-CHECK is currently restricted to Fortran code and performs argument type checking or finds problems like deadlocks [6]. Like MARMOT, Umpire [12] uses the profiling interface. But in contrast to our tool, Umpire is limited to shared memory platforms.

3 Description of MARMOT

Among the design goals of MARMOT are portability, scalability and reproducibility. The portability of an application should be improved by verifying that the program adheres to the MPI standard [7]. The tool issues warnings and error messages if the application relies on non-portable MPI construct. Scalability is addressed with the use of automatic techniques that do not need user intervention. The tool contains a mechanism to detect possible race conditions to improve the reproducibility. It also automatically detects deadlocks and notifies the user where and why these have occurred. MARMOT uses the MPI profiling interface to intercept the MPI calls and analyse them. MARMOT can be used with any MPI implementation that provides this interface. It adds an additional MPI process for all tasks that cannot be handled within the context of a single MPI process, like deadlock detection. Information between the MPI processes and this additional debug process are transferred using MPI. Another possible approach is to use a thread instead of an MPI process and use shared memory communication instead of MPI [12]. The advantage of the approach taken here is that the MPI library does not need to be thread safe. Without the limitation to shared memory systems the tool can also be used on a wider range of platforms. Since the improvement of portability was one of the design goals we did not want to limit the portability of MARMOT. This allows to use the tool on any development platform used by the programmer.

4 Description of the Applications

To measure its overhead when using MARMOT with a real application, we chose different applications from the CrossGrid project.

4.1 Weather Forecast and Air Pollution Modeling

The MPI parallel application [8] of Task 1.4.3 of the CrossGrid project calculates the acid deposition caused by a power plant. The used STEM-II model is one of the most complex air quality models. The transport equations are solved through the Petrov-Crank-Nicolson-Galerkin method (FEM). The chemistry and mass transfer terms are integrated using a semi-implicit Euler and a pseudo-analytic method.

The STEM application is written in FORTRAN and consists of 15500 lines of code. 12 different MPI-calls are used within the application: MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Type_extent, MPI_Type_struct, MPI_Type_commit, MPI_Type_hvector, MPI_Bcast, MPI_Scatterv, MPI_Barrier, MPI_Gatherv, MPI_Finalize.

4.2 High Energy Physics Application

The HEP application [10] from CrossGrid Task 1.3 performs an analysis of the physical data produced by the Large Hadron Collider (LHC) at CERN. All collisions will be recorded by detectors, the corresponding information being stored in distributed databases with a volume of millions of gigabytes. On-line filtering techniques as well as mathematical algorithms, such as neural networks, will be used to select those events and analyse them by physicists working in research centers across the world.

The MPI parallel application ANN (Artificial Neural Network) is a neural network application that is part of the data analysis described above. It currently consists of 11500 lines of C code and uses 11 different MPI calls: MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Get_processor_name, MPI_Barrier, MPI_Gather, MPI_Recv, MPI_Send, MPI_Bcast, MPI_Reduce, MPI_Finalize.

4.3 Medical Application

The application [11] from Task 1.1 is a system used for pre-treatment planning in vascular interventional and surgical procedures through real-time interactive simulation of vascular structure and flow. A 3D model of the arteries serves as input to a real-time simulation environment for blood flow calculations. The user will be allowed to change the structure of the arteries, thus mimicking a surgical procedure. The effects of any modification is analysed in real time while the results are presented to the user in a virtual environment. A stripped down version of the MPI parallel application calculating the blood flow consists of 7500 lines of C code. The code makes use of the following MPI calls: MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Pack, MPI_Bcast, MPI_Unpack, MPI_Cart_create, MPI_Cart_shift, MPI_Send, MPI_Recv, MPI_Barrier, MPI_Reduce, MPI_Sendrecv, MPI_Finalize.

5 Possible Checks for the Used MPI-Calls

This section is to present in more detail what kind of checks MARMOT performs on the calls made by an MPI application. For example, the ANN-application and the STEM-application described above employ the following calls.

– Environmental calls (MPI_Init, MPI_Get_processor_name, MPI_Finalize): A possible test is to check for illegal MPI-calls before MPI_Init and after MPI_Finalize. It is also illegal to call MPI_Init more than once in an application. Currently MARMOT does not perform any checks for MPI_Get_processor_name, because it is a local call without much room for errors. Concerning MPI_Finalize, MARMOT checks if active requests and pending messages are left over in any communicator.
– MPI_Comm_size, MPI_Comm_rank, MPI_Barrier: MARMOT checks if the communicator of the call is valid, i.e. if it is MPI_COMM_NULL or if it is a user-defined communicator, which has been created and committed properly or which might have been freed again before using it.
– Construction of datatypes (MPI_Type_extent, MPI_Type_struct, MPI_Type_hvector, MPI_Type_commit): MARMOT inspects the validity of the datatype argument, and for MPI_Type_struct and MPI_Type_hvector it also inspects if the count and the block length are greater than zero. The tool also verifies if MPI-Type_commit is used to commit a type again that has already been committed.
– Point-to-point communication (MPI_Send, MPI_Recv,MPI_Sendrecv): MARMOT inspects the correctness of the communicator, count, datatype, rank and tag arguments. Similar to the way communicators are checked, it is verified if the datatype is MPI_DATATYPE_NULL or if it has been created and registered correctly by the user. If the count is zero or negative, a warning will be issued, also if ranks and tags are beyond valid ranges. The MPI standard also requires that the program does not rely on any buffering made by the standard send and receive operations. Since the amount and type of buffering is different between the various MPI implementations this is one of the problems that would limit the portability of the MPI program.
– Other collective operations (MPI_Bcast, MPI_Reduce, MPI_Gather, MPI_Gatherv, MPI_Scatterv): The tool checks if the communicator, count, datatype and the rank of the root are valid. Additionally for MPI_Reduce, it is also checked if the operator is valid, e.g. if it has been created properly. For MPI_Gatherv and MPI_Scatterv, also the displacements are examined.

Besides these calls, MARMOT supports the complete MPI-1.2 standard, although not all possible tests have been implemented so far. It also issues warnings when a deadlock occurs and allows the user to trace back the last few calls on each node. Currently the deadlock detection is based on a timeout mechanism. MARMOT's debug server surveys the time each process is waiting in an MPI call. If this time exceeds a certain user-defined limit on all processes at the same time, the debug process issues a deadlock warning.

Possible race conditions can be identified by locating the calls that may be sources of race conditions. One example is the use of a receive call with `MPI_ANY_SOURCE` as source argument. MARMOT does not use methods like record and replay to identify and track down bugs in parallel programs [5].

6 Performance with the Applications

Our main interest is whether the performance penalty induced by the usage of MARMOT is small enough to use MARMOT in the daily development work of the application. A typical development run is between 10 minutes and one hour. In our experience a run like this should not exceed a time limit of two hours. In order to be useful MARMOT's overhead should therefore be less than 100%. In rare cases where a bug is only present on high processor count an overnight run (12 hours) should be feasible allowing an overhead of up to 1000%. The applications were run with and without MARMOT, on an IA32-cluster using mpich and Myrinet interconnect. For these runs, the execution times were kept in the range of several minutes by reducing the number of iterations. This does not limit the validity of the analysis because evey single iteration shows the same communication behaviour. For example, the STEM application executes 300 iterations to simulate 5 hours of real time, each iteration corresponding to a minute of real time.

6.1 Weather Forecast and Air Pollution Modelling

The scalability of the weather forecast application is limited. It has a parallel efficiency on 16 processes of only 14% (19s of 26s execution time is spent in MPI calls). The scalability with and without MARMOT is shown in Fig. 1. With MARMOT the total time increases from 69.4 s to 70.6 s on 1 process, and from 29 s to 50 s on 16 processes. MARMOT's overhead is approximately between 2 and 72% on up to 16 processes.

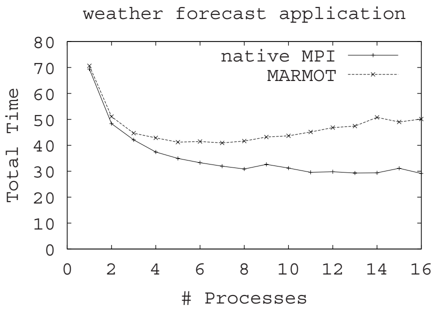

Fig. 1. Comparison of total execution times for the STEM-application between the native MPI approach and the approach with MARMOT.

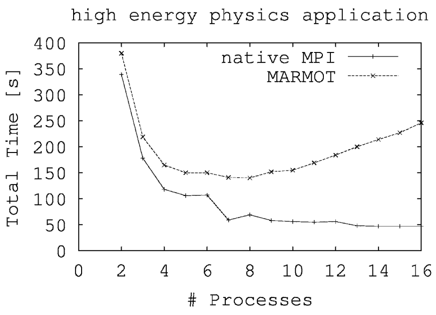

Fig. 2. Comparison of total execution times for the ANN-application between the native MPI approach and the approach with MARMOT.

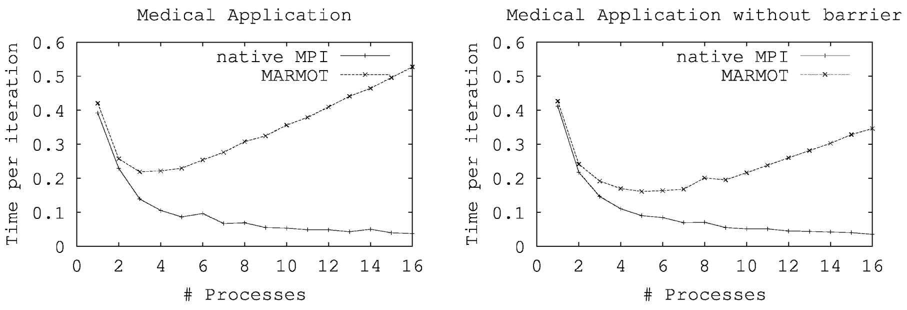

Fig. 3. Comparison of total execution times for the medical application between the native MPI approach and the approach with MARMOT. The left side shows the original result, on the right side one barrier in the application was removed.

6.2 High Energy Physics Application

The results of the HEP application can be seen in Fig. 2. The total time increases from 339 s to 380 s on 2 processes when MARMOT is linked to the application, and from 47 s to 246 s on 16 processes.

MARMOT's overhead is approximately between 20 and 420% on up to 16 processes. On 16 processes the applications runs with 50% parallel efficiency, 26s of 48s execution time are spent in MPI calls. Most of the communication time is spent for broadcast and reduce operations.

6.3 Medical Application

The results of the medical application can be seen in Fig. 3. The time per iteration increases from 0.39s to 0.42 s on 1 process when MARMOT is linked to the application, and from 0.037 s to 0.53 s on 16 processes.

MARMOT's overhead is approximately between 8 and 1312% on up to 16 processes. A detailed analysis with Vampir showed several reasons for the large performance penalty caused by MARMOT. First, the communication pattern

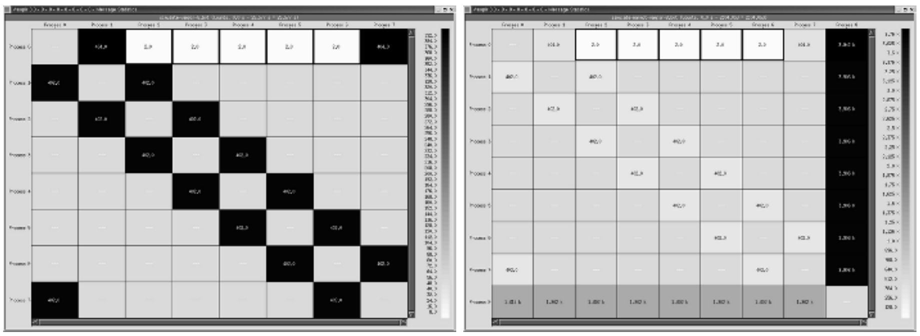

Fig. 4. Message statistics for the medical application without (left) and with MARMOT (right).

of the application is modified, because all application processes have to notify MARMOT's debug server about every MPI call. Figure 4 shows a typical communication pattern that is dominated by message exchanges with two neigbours. As soon as MARMOT is used the communication is dominated by the messages between all application processes and the debug server. Second, the application performs a large number of MPI_Barriers. This is a fast operation for most MPI implementations. However, inside MARMOT each client will register with the debug server to notify it about its participation in this operation. This results in a linear scaling with the number of processes. If the barrier that is called after each iteration is removed the execution time with MARMOT is reduced from 0.53s to 0.35s (see Fig. 3).

7 Conclusions and Future Work

In this paper we have presented the tool MARMOT. It analyses the behaviour of an MPI application and checks for errors frequently made in the use of the MPI API. We demonstrated the functionality of the tool with three real world applications from the CrossGrid IST project. The applications cover the C and Fortran binding of the MPI standard. The inevitable performance penalty induced by the performed analysis and checks depends strongly on the application. For the applications of the CrossGrid project used in our tests, the runtimes with MARMOT are in the range of several minutes, which allows a regular usage during the development and verification process of an application. However, especially for the communication intensive applications with a high number of collective communications the overhead caused by MARMOT was above 1000%. Future work will include improvements for this type of applications.

Acknowledgements. The development of MARMOT is supported by the European Union through the IST-2001-32243 project "CrossGrid".

References

1. WWW. `http://www.etnus.com/Products/TotalView`.
2. William D. Gropp. Runtime checking of datatype signatures in MPI. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1908 of *Lecture Notes In Computer Science*, pages 160–167. Springer, Balatonfüred, Lake Balaton, Hungary, Sept. 2000. 7th European PVM/MPI Users' Group Meeting.
3. Robert Hood. Debugging computational grid programs with the portable parallel/distributed debugger (p2d2). In *The NASA HPCC Annual Report for 1999.* NASA, 1999. `http://hpcc.arc.nasa.gov:80/reports/report99/99index.htm`.
4. Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: An MPI analysis and checking tool. In *Proceedings of PARCO 2003*, Dresden, Germany, September 2003.
5. Dieter Kranzlmüller. *Event Graph Analysis For Debugging Massively Parallel Programs.* PhD thesis, Joh. Kepler University Linz, Austria, 2000.
6. Glenn Luecke, Yan Zou, James Coyle, Jim Hoekstra, and Marina Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14:911–932, 2002.
7. Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. `http://www.mpi-forum.org`.
8. J.C. Mourino, M.J. Martin, R. Doallo, D.E. Singh, F.F. Rivera, and J.D. Bruguera. The stem-ii air quality model on a distributed memory system, 2004.
9. Sue Reynolds. System software makes it easy. *Insights Magazine*, 2000. NASA, `http://hpcc.arc.nasa.gov:80/insights/vol12`.
10. D. Rodriguez, J. Gomes, J. Marco, R. Marco, and C. Martinez-Rivero. MPICH-G2 implementation of an interactive artificial neural network training. In *2nd European Across Grids Conference, Nicosia, Cyprus*, January 28-30 2004.
11. A. Tirado-Ramos, H. Ragas, D. Shamonin, H. Rosmanith, and D. Kranzlmueller. Integration of blood flow visualization on the grid: the flowfish/gvk approach. In *2nd European Across Grids Conference, Nicosia, Cyprus*, January 28-30 2004.
12. J.S. Vetter and B.R. de Supinski. Dynamic software testing of mpi applications with umpire. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference (SC 2000)*, Dallas, Texas, 2000. ACM/IEEE. CD-ROM.