Performance Penalty Detection in MPI Applications

Manuk, Akopyan

Institute for system programming
Moscow, Russian Federation
e-mail: manuk@ispras.ru

ABSTRACT

Most of the developed tools for analysis of various libraries (MPI, OpenMP) and languages for parallel programming use low level approaches to analyze the performance of parallel applications (profilers, trace visualizers). In most cases the developer has to manually look for bottlenecks and opportunities for performance improvement in the produced runtime data. The amount of information developer has to handle manually increase dramatically with number of cores, number of processes and the problem size of application. Therefore, new methods of automated performance analysis of output information will be more favorable. In this paper code patterns resulting in performance penalties are discussed. Patterns of parallel MPI applications for parallel computing systems with distributed memory are considered. A method for automatic detection of inefficiency patterns in parallel MPI-1 applications and UPC programs is proposed. It allows to reduce the tuning time of a parallel application and improve the productivity of parallel program development.

Keywords

parallel programming, semantic errors, inefficiency patterns, MPI, UPC

1. INTRODUCTION

Most of the developed performance analysis tools for different libraries and languages of parallel programming use low-level approaches to performance analysis of parallel programs. Mostly, those are profiling utilities or traces visualizers. As a result of the analysis programmer receives tables and graphs with program execution statistics. This information does not give a clear view of possible troubles and bottlenecks of application. Developer looks through graphs manually searching for program slowdown and possible optimization capabilities. Given the fast growth of number of cores in modern high-performance computing systems, the amount of data the programmer has to process becomes unacceptably large and manual analysis methods become inapplicable. Therefore, new methods of performance analysis implementing full or partial automatization of obtained data processing are required for parallel applications in modern environment.

Under this research a method of automated detection of inefficiency patterns in parallel MPI [1] applications and UPC [2] programs has been developed. In this paper terms performance error and inefficiency pattern would be used interchangeably. Patterns of inefficient usage of MPI-1 p2p functions are discussed. As long as MPI is an industrial standard for parallel programs with distributed memory, this article discusses a method for detection of performance errors mostly in MPI programs. The method is based on the analysis of data obtained during the parallel program execution (post-mortem analysis). Description of patterns for programs using MPI is given in the paper.

The paper is organized as follows. Section 2 discusses some related works. Section 3 describes the proposed method of automated error detection in parallel MPI and UPC applications and performance error types for these programming models. Finally, section 4 concludes the paper by summarizing the main points addressed through this paper.

2. RELATED WORKS

One of the most known systems for parallel applications performance improvement is TAU [3]. TAU is a toolkit for parallel programs performance analysis and was developed by researchers from the Oregon University, National laboratory of Los Alamos and Juelich research center (Germany). TAU provides a set of static and dynamic tools, which through interaction with the user perform a complex analysis of parallel applications in Fortran, C, C++, Java and Python. tools for automated instrumentation are also developed within TAU. Hercule [4] tool of the TAU system is a prototype of module which uses knowledge base to detect and find out causes of performance bottlenecks in accordance with programming paradigm (such as master-worker, pipeline, etc.) instead of programming model (MPI, OpenMP). Hercule allows analyzing the applications written in any of programming models. However this tool cannot process the applications developed using combination of different paradigms.

The PPW system [5] was developed in the HCS (High-performance Computing and Simulation) laboratory of the Florida University. The system was created to analyze performance of parallel PGAS program (in particular UPC and SHMEM programs). At first, the program is instrumented and run. As a result of instrumented program run a program profile (statistical data of execution time) and trace (trace is created in its own format) are gathered. The gathered data can be used for parallel program analysis and bottlenecks detection. Also, there are convertors of program trace to popular formats. That allows users to apply well known visualization tools (Vampir, JumpShot, etc.) for manual optimization. PPW is an actively developed package with a graphical user interface and rich functionality. However methods underlying the package are low-level and do not use automated approaches to analysis.

The Scalasca [6] system is a toolkit designed for performance analysis and was developed especially for using on large systems with tens of thousands of cores, but it also has proved its worth for small and medium HPC platforms. Scalasca supports measurement and analysis of MPI and OpenMP syntax constructions as well as hybrid programming constructions widely used in HPC applications written in C, C++, Fortran. The system was developed in the Julich Supercomputing Centre and the German Research School for Simulation Sciences. At first, parallel application is instrumented. On launch each process creates a trace file, containing records for local events of the given process. After the completion of parallel program execution, Scalasca allows to perform a post-mortem analysis of trace events. First, local traces of processes are merged in a single trace.

For clock synchronization of different processes the method described in [7] is used. After merging of local traces in a global one, EXPERT tool [8, 9] can be used for inefficiency patterns detection. The EXPERT sequentially scans events in the global trace and looks for predefined patterns included in system distributive. Only terminal events (SEND, RECV, etc.) can be met in the trace. Each event contains timestamp among other properties. A pattern is a combination of terminal events matching certain predicates. About 30 patterns for MPI, OpenMP and SHMEM programs are defined in the system.

At this moment there are no toolkits supporting the development of parallel applications and automatically detecting performance errors in MPI and UPC programs. Existing systems allow looking for errors manually or do not cover the necessary error types for MPI and UPC applications.

3. AUTOMATED PERFORMANCE ERROR DETECTION

3.1. Method description

In this paper errors of MPI functions usage leading to parallel applications performance loss and inefficiency patterns will be considered as equals. Method of automated detection of inefficiency patterns in parallel programs is based on the analysis of data obtained during the parallel program execution in data gathering mode (post-mortem analysis). To automatically detect patterns first we need to obtain execution time data on critical functions potentially leading to patterns of certain types. After that, analysis of gathered data is performed in order to detect these patterns. The developed approach is based on usage of open source libraries of the Scalasca [6].

Therefore, the method for automated detection of inefficiency patterns in parallel programs consists of the following stages:
Stage 1. Gathering the runtime data of parallel program.
Stage 2. Analysis of the data obtained at the Stage 1 and detection of patterns in parallel program.
Stage 3. Creation of report on detected errors with binding to parallel program source code.

Building the trace of parallel application includes instrumentation stage and execution of instrumented program on target platform. Program instrumentation means adding calls to instrumental library in certain positions of the original program. During the program execution these calls register a certain event and make a record in trace. After that the instrumented program is transferred onto target platform and a parallel program is launched. As a result a trace is created for each process of the program.

At the second stage after the event trace is obtained a post-mortem analysis is performed – trace of parallel program is analyzed in order to detect certain errors. Certain criterion corresponds to each pattern - predicate of event timestamp, timestamp of corresponding paired event, etc. To detect patterns, events from trace are looked over and corresponding pattern is registered on certain predicate execution.

At the third stage the gathered data is sent to report generator, which creates a final report in convenient format. The final report contains a list of error descriptors.

The developed method is applicable to detection of performance errors in both MPI and UPC applications.

3.2. Error types description in MPI-programs

Let's look upon patterns when using point to point communications in parallel MPI-programs. Let F represent the MPI communication function. We'll define the reference time right before the F function as Time_start(F). Time_end(F) is a timestamp for event after the F function. I_T(pid, p_i, c_j) will indicate idleness time of process with pid identifier as a result of communication c_j={sendId,recvId} caused by detection of error p_i. sendId and recvId represent sending and receiving identifiers correspondingly. Let ε be the threshold value.

3.2.1 Blocking point-to-point communication patterns

During message transfer from one process to another, idle state may occur on one or another side. This effect does not affect computation correctness but it will have a negative impact on program execution speed. Removing those idle states (when possible) will lead to performance improvement of parallel program.

«Early standard send». Let's look at the situation when sending occurs earlier than receiving (fig. 1). In this case sender process loses time. Pattern criteria:

$$\mathrm{I\_T}(\mathrm{pid}, p_i, c_j) = \begin{cases} 0, \text{ if } (\mathrm{Time\_start}(\mathrm{MPI\_Recv}) - \mathrm{Time\_start}(\mathrm{MPI\_Send})) < \varepsilon \\ \mathrm{Time\_start}(\mathrm{MPI\_Recv}) - \mathrm{Time\_start}(\mathrm{MPI\_Send}), \text{ otherwise} \end{cases} > 0$$

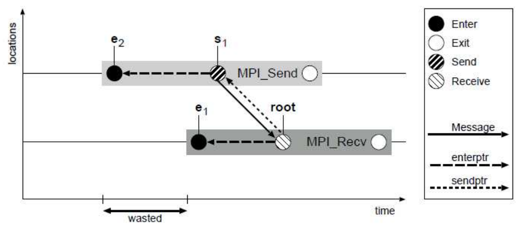

Fig. 1. Early send with MPI_Send.

We must also take into account MPI implementation features (also mentioned in MPI standard). If forward sending protocol has been used in implementation of MPI library (e.g. MPICH [10], MPICH2, MVAPICH [11], MVAPICH2,…), then call to function MPI_Send could be local (non-blocking) if the size of sending message is less then the predefined constant. Therefore sender will not become idle and we must exclude this case during the pattern search process.

«Early buffered send». Message sending begins earlier than corresponding receiving. However in this case the sender process is not idle because the MPI_Bsend function is local – the function copies the message into buffer and returns control to the program and the MPI runtime system sends the message from buffer.

«Early synchronous send». This pattern is similar to the «**Early standard send**», but MPI_Ssend is used instead of MPI_Send. Pattern criteria:

$$\mathrm{I\_T}(\mathrm{pid}, p_i, c_j) = \begin{cases} 0, \text{ if } (\mathrm{Time\_start}(\mathrm{MPI\_Recv}) - \mathrm{Time\_start}(\mathrm{MPI\_Ssend})) < \varepsilon \\ \mathrm{Time\_start}(\mathrm{MPI\_Recv}) - \mathrm{Time\_start}(\mathrm{MPI\_Ssend}), \text{ else} \end{cases} > 0$$

«Early ready send». This pattern is similar to «**Early standard send**», but MPI_Rsend is used instead of MPI_Send. Pattern criteria:

$$\text{I\_T}(\text{pid}, p_i, c_j) = \begin{cases} 0, \text{ if } (\text{Time\_start(MPI\_Recv) - Time\_start(MPI\_Rsend)}) < \varepsilon \\ \text{Time\_start(MPI\_Recv) - Time\_start(MPI\_Rsend)}, \text{ else} \end{cases} > 0$$

«Late standard send». Let's look at the situation when receiving occurs earlier than sending (fig. 2). In this case receiver process loses time. Pattern criteria:

$$\text{I\_T}(\text{pid}, p_i, c_j) = \begin{cases} 0, \text{ if } (\text{Time\_start(MPI\_Send) - Time\_start(MPI\_Recv)}) < \varepsilon \\ \text{Time\_start(MPI\_Send) - Time\_start(MPI\_Recv)}, \text{ otherwise} \end{cases} > 0$$

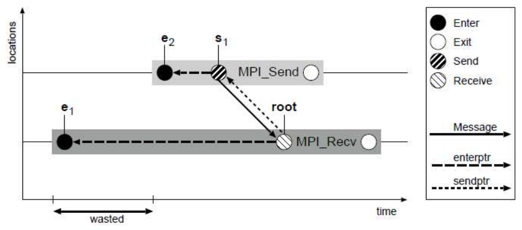

Fig. 2. Early receive with MPI_Send.

«Late buffered send». This pattern is similar to the «**Late standard send**», but MPI_Bsend is used instead of MPI_Send. Pattern criteria:

$$\text{I\_T}(\text{pid}, p_i, c_j) = \begin{cases} 0, \text{ if } (\text{Time\_start(MPI\_Bsend) - Time\_start(MPI\_Recv)}) < \varepsilon \\ \text{Time\_start(MPI\_Bsend) - Time\_start(MPI\_Recv)}, \text{ else} \end{cases} > 0$$

«Late synchronous send». This pattern is similar to the «**Late standard send**», but MPI_Ssend is used instead of MPI_Send. Pattern criteria:

$$\text{I\_T}(\text{pid}, p_i, c_j) = \begin{cases} 0, \text{ if } (\text{Time\_start(MPI\_Ssend) - Time\_start(MPI\_Recv)}) < \varepsilon \\ \text{Time\_start(MPI\_Ssend) - Time\_start(MPI\_Recv)}, \text{ else} \end{cases} > 0$$

«Late ready send». This pattern is similar to the «**Late standard send**», but MPI_Rsend is used instead of MPI_Send. Pattern criteria:

$$\text{I\_T}(\text{pid}, p_i, c_j) = \begin{cases} 0, \text{ if } (\text{Time\_start(MPI\_Rsend) - Time\_start(MPI\_Recv)}) < \varepsilon \\ \text{Time\_start(MPI\_Rsend) - Time\_start(MPI\_Recv)}, \text{ else} \end{cases} > 0$$

3.2.2 «Message misarrangement» patterns

Effect of «message misarrangement» may occur when receiver process awaits messages in one sequence and sender process sends messages in another order. By rearranging messages we will not only speed up the program but will also need less buffer size for unprocessed messages storing. If during send-receive the rendezvous protocol is used, the program will obviously go to deadlock. But if send is local (buffered send or standard send with short message size and MPI implementation send message through internal buffer) there will be no blocking, but an ineffective communications arrangement instead.

«Misarrangement with the use of MPI_Send». Fig. 3. represents a graphical view of inefficiency pattern during messages sending in wrong order with the use of MPI_Send function.

Pattern criteria is given below:
Time_end(MPI_Send$_0$) < Time_end(MPI_Send$_1$) < Time_end(MPI_Send$_2$) and Time_end(MPI_Recv$_2$) < Time_end(MPI_Recv$_1$) < Time_end(MPI_Recv$_0$) and Time_start(MPI_Recv$_2$) < Time_start(MPI_Send$_2$)

«Misarrangement with the use of MPI_Bsend». This pattern is similar to «**Misarrangement with the use of MPI_Send**», but MPI_Bsend is used instead of MPI_Send.

«Misarrangement with the use of MPI_Ssend». Message misarrangement pattern is impossible when using {MPI_Ssend, MPI_Recv} function couple, because in this case subsequent MPI_Ssend inquiries will be blocked without meeting corresponding MPI_Recv-s.

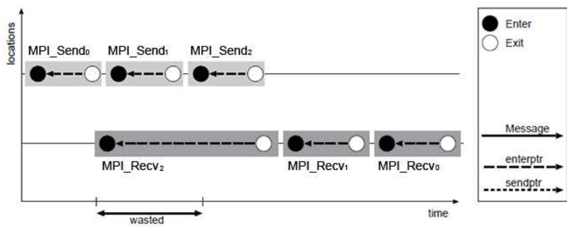

Fig. 3- Pattern «misarrangement» with the use of {MPI_Send, MPI_Recv} functions couple.

«Misarrangement with the use of MPI_Rsend». Pattern «misarrangement» is not applicable to situation when a sending sequence via MPI_Rsend-s and backward receiving sequence by MPI_Recv-s is used.

3.2.3 Non-blocking point-to-point communication patterns

Consider we have a couple of calls {MPI_Isend,MPI_Wait} in process p_0 and {MPI_Irecv, MPI_Wait} in process p_1. Let's consider errors which occur in this case in processes pid$_0$ and pid$_1$.

«Waiting on sender-side when using {MPI_Isend, MPI_Irecv}». Let's consider the process pid$_0$. In this process after the call of MPI_Isend, control returns to the process pid$_0$ and computing instructions are executed, then the MPI_Wait function is called. If the MPI_Wait call was made too early, the process is blocked and stands idle. Event trace contains timestamps for each event, therefore, the difference between timestamps of events after and before the MPI_Wait call allows to calculate the time of process idleness. Pattern criteria:

$$\text{I\_T}(\text{pid}_0, p_i, c_j) = \begin{cases} 0, \text{ if } (\text{Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait)}) < \varepsilon \\ \text{Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait)}, \text{ otherwise} \end{cases} > 0$$

Apart from a pattern detection, we can throw a diagnostic message with estimation of optimal distance (O_D(pid,p_i,c_j)) for MPI_Wait call.

$$\text{O\_D}(\text{pid}_0, p_i, c_j) = \begin{cases} \Delta Ti + T_{send}, \text{ if Time\_start(MPI\_Isend)} > \text{Time\_start(MPI\_Irecv)} \\ (\text{Time\_start(MPI\_Irecv) - Time\_start(MPI\_Isend)}) + \Delta Ti + T_{send}, \text{ else} \end{cases}$$

where ΔT_i – is the MPI_Isend function execution time, T_{send} estimation of time for real sending through communication network.

Patterns of the following types are defined in similar way: **«Waiting on receiver-side when using {MPI_Isend, MPI_Irecv}»**, «**Waiting on receiver-side when using {MPI_Ibsend, MPI_Irecv}»**, **«Waiting on sender-side when using {MPI_Issend, MPI_Irecv}»**, «**Waiting on receiver-side when using {MPI_Issend, MPI_Irecv}»**, **«Waiting on receiver-side when using {MPI_Irsend, MPI_Irecv}», «Waiting on sender-side when using {MPI_Irsend, MPI_Irecv}».**

«Waiting on sender-side when using {MPI_Ibsend, MPI_Irecv}». In this case there will be no error because the MPI_Ibsend nonblocking function is local – the function copies a message into buffer and returns control to the program then MPI runtime system sends a message from buffer.

3.2.4 Close send-receive pattern

Let's consider a program, where message send and receive with process pid_j is used in process pid_i and calls to these functions are close to each other in source code (fig. 4).

```
if( rank == pidi )
{
  int *send_buf = (int
*)malloc(sizeof(int) * 12001);
  int *recv_buf = (int
*)malloc(sizeof(int) * 12001);
  MPI_Status stat;
  MPI_Send(send_buf,12001,MPI_INT,
pidj,0,MPI_COMM_WORLD);
…
  MPI_Recv(recv_buf,12001,MPI_INT,
pidj,0,MPI_COMM_WORLD,stat);
}
else if( rank == pidj )
{
  int * send_buf = (int
*)malloc(sizeof(int) * 12001);
  int * recv_buf = (int
*)malloc(sizeof(int) * 12001);
    MPI_Status stat2;
  MPI_Recv(recv_buf,12001,MPI_INT,
pidi,0,MPI_COMM_WORLD, stat2);
…
  MPI_Send(send_buf,12001,MPI_INT,
pidi,0,MPI_COMM_WORLD);
}
```

Fig. 4- Example of using close send-receive.

If such a pattern has been found and program logic allows (user has to make sure that buffer in following MPI_Send/MPI_Recv is not used), user can unite MPI_Send and MPI_Recv functions in MPI_Sendrecv function, which will grant a serious improvement in execution time. It occurs because the MPI_Sendrecv function is implemented effectively by MPI vendors. Also, modern high-performance communication networks (Infiniband [12]) support full-duplex interconnect on low level, which allows to send and receive message for single HCA simultaneously. In this case send and receive operations take place at the same time, instead of pid_i waiting for MPI_Send operation ending and only after that waiting for MPI_Recv ending. Let ΔSR be a certain predefined threshold. Pattern criteria:
$|Time\_start(MPI\_Recv) - Time\_end(MPI\_Send)| < \Delta SR$

3.3. Pattern types in UPC-programs

20 inefficiency patterns for UPC-programs have been developed [13]. The first group contains eight patterns for detecting delays in collective operation of data transfer (also known as relocalization operations in UPC [2]). The second group consists of seven patterns related to explicit and implicit synchronization existing in virtually all parallel programming languages. The third group contains three patterns related to data transfer and they allow to detect hot points and bottlenecks of program (in terms of data amount transferring between threads). The fourth group (contains two patterns) related to the master-slave model of parallel programming, where master thread creates a bunch of slave threads and distributes all tasks between them.

4. RESULTS

Discussed method has been applied to parallel program of calculation of viscous flow around a blunt body [14]. Application is written in C++, volume of source code 4500 lines. Implemented tool detected 4 patterns – two «Waiting on sender-side when using {MPI_Isend, MPI_Irecv}» and two «Waiting on receiver-side when using {MPI_Isend, MPI_Irecv}», correction of which improved performance 3-5% on different number of processes.

5. CONCLUSION

In this paper the method for automated detection of errors in parallel MPI and UPC-programs developed. The method is based on analysis of runtime data (post-mortem analysis). The developed method allows to detect 17 types of performance errors in MPI-programs and 20 types of errors in UPC-programs.

REFERENCES

[1] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. MPI – The complete Reference, Volume 1, The MPI Core, Second edition. / The MIT Press. 1998.

[2] W. Chen, C. Iancu, K. Yelick. Communication Optimizations for Fine-grained UPC Applications. //14th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2005.

[3] Sameer S. Shende Allen D. Malony. “The Tau Parallel Performance System”, International Journal of High Performance Computing Applications, Volume 20, Issue 2, Pages: 287 – 311, May 2006.

[4] L. Li and A.D. Malony, “Model-Based Performance Diagnosis of Master-Worker Parallel Computations,” Lecture Notes in Computer Science, Number 4128, Pages 35-46, 2006.

[5] H. Su, M. Billingsley III, and A. George. "Parallel Performance Wizard: A Performance System for the Analysis of Partitioned Global Address Space Applications," International Journal of High-Performance Computing Applications, Vol. 24, No. 4, Nov. 2010, pp. 485-510.

[6] Ilya Zhukov, Brian J. N. Wylie: Assessing Measurement and Analysis Performance and Scalability of Scalasca 2.0. In Proc. of the Euro-Par 2013: Parallel Processing Workshops, volume 8374 of LNCS, pages 627-636, Springer, January 2014.

[7] Felix Wolf. Automatic Performance Analysis on Parallel Computers with SMP Nodes. PhD thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003, ISBN 3-00-010003-2.

[8] Wolf, F., Mohr, B. Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture 49(10-11) (2003) 421–439.

[9] Wolf, F., Mohr, B., Dongarra, J., Moore, S. Efficient Pattern Search in Large Traces through Successive Refinement. In: Proc. European Conf. on Parallel Computing (Euro-Par, Pisa, Italy), Springer (2004).

[10] MPICH. http://www.mpich.org

[11] MVAPICH. http://mvapich.cse.ohio-state.edu.

[12] Infiniband. http://www.infinibandta.org.

[13] M.S. Akopyan, N.E. Andreev. Research and development of inefficiency patterns in MPI, UPC applications. Trudy ISP RAN [Proceedings of ISP RAS], vol. 24, pp. 49-70, 2013.

[14] F.A. Maksimov, D.A. Churakov, Yu. D. Shevelev. Development of mathematical models and numerical methods for aerodynamic design on multiprocessor computers. Computational Mathematics and Mathematical Physics, Volume 51, Issue , pp 284-307