Statistical Methods for Automatic Performance Bottleneck Detection in MPI Based Programs

Michael Kluge, Andreas Knüpfer, and Wolfgang E. Nagel

Technische Universität Dresden, Dresden, Germany
{kluge, knuepfer, nagel}@zhr.tu-dresden.de

Abstract. With increasing number of processors the visualization of trace data for MPI communication primitives becomes harder due to many limitations. An automatic analysis of the recorded data should be able to identify some clues for the optimization of the source code. The approach presented with this paper is able to divide the communication time for each communication relationship into a necessary part and the overhead. We use statistical methods to define a time for each single communication event that we will accept as the 'usual' time for this event. Additional runtime will be considered as overhead. At the end of this article, the benefits from this method are demonstrated on the Sweep3D benchmark.

1 Introduction

2005, the supercomputer IBM 'Blue Gene' will be installed with a processor count of 131072. The first desktop systems with more than a single processor are already available for everyone. To use such systems efficiently parallel programs have to be developed that utilizes all processors. Writing parallel programs today is commonly done by using one (or both) of the two popular programming paradigms, OpenMP [2] and MPI [4]. This work presents a new approach for automatic bottleneck detection in MPI programs.

The first section gives an overview on actual approaches in MPI program analysis. The next sections are dedicated to the motivation and our statistical approach for finding a maximal communication time. Then we will present some tests for the practical relevance of these maximal communication times. At the end of this paper we will show how a profiler using our approach can be used for automatic performance bottleneck detection in the Sweep3D benchmark.

2 State of the Art in MPI Program Analysis

The question how well a program is scaling with an increasing amount of processors has driven the development of many tools (see [11], [10], [3] etc.). Recording all program activities during runtime and an appropriate post-mortem analysis of the recorded data is still state of the art in parallel program analysis. This is due to two main reasons. First of all, in the beginning of an optimization process one does typically not know where to start. What is needed is a trace of all program activities. The outcome of this is at the same time the second reason for the post-mortem analysis, a huge amount of data. The user himself will not be able to do any analysis during runtime. An automated analysis will most probably interfere with the program execution in a way that no meaningful conclusions can be drawn from these data. Actually there exist several approaches for trace file compression or removing of redundant information during the trace process itself ([7],[1]) and during the post-mortem analysis ([8]).

3 Motivation

The aim of this article is to define a value that reflects the maximum execution time for a MPI function call under ideal conditions. Within our approch we try to determine, how the data within a sample generated with a repeated execution of this function are distributed. If those data are distributed like a theoretical distribution (like a Normal/Gaussian or Exponential distribution) it is possible to define for example the time that is associated with 99% of this distribution as the maximum time. First experiments have shown that the data we are looking at are usually not normally distributed. Instead of this we have seen asymetric distributions. As a result of this, a simple approach like calculating the mean μ and variation σ^2 and afterwards using the time associated with $\mu + 3\sigma$ would end up with a maximum execution time without meaning. Because a calculation that $\mu + 3\sigma$ includes 99.73% of the distribution is only true for the assumption of a Gaussian distribution.

4 The Statistical Approach

To be able to divide the time t needed for each MPI communication event into a necessary and a wasted part we will define a time t_{max} as a maximum time we want to accept for an execution without overhead. A communication event is a call of an MPI function of the MPI 1.1 standard that transmits data or completes one of the non-blocking communication calls [4–p. 41]. The time t is the execution time a function call needs during the program execution. To determine t_{max} we take a sample $S = \{t_1, \ldots, t_N\}$ with the sample size N of execution times for a communication event and define the wasted time t_{oh} as follows:

$$\begin{array}{l} t \leq t_{max} \rightarrow t_{oh} = 0 \\ t > t_{max} \rightarrow t_{oh} = t - t_{max} \end{array}$$

We assume that a sample taken with a synthetic program represents the typical behavior of the underlying MPI implementation and hardware. The problem that we have to face is that although each time t_i within the sample is taken by executing the same function with the same parameters those times are not identical. They are distributed in a way that differs from machine to machine. Each time t_i can not be smaller than a minimum t_{min} that is determined by the MPI implementation and the hardware. It is most likely that none of the times t_i will match t_{min} exactly. Instead of that each is superimposed with one or more errors. If we assume an exclusive use of the communication network we can consider that those errors result from the overlapping execution of processes, from the operation system or from the resolution of the timer used for the measurement. If we assume further on that those processes are activated periodically and each activation does only influence one measurement they can be represented by a Poisson distribution. A sum of Poisson distributions is again a Poisson distribution. If we presume t as

$$t = t_{min} + t_{err}, \tag{1}$$

we can model the distribution of the error t_{err} as a Poisson distribution. During our experiments we have seen that there are other statistical distributions that sometimes fit to the error within a sample better than a Poisson distribution. The upper bound of the error that can be accepted can be fixed by establishing a quantile q_{max} for the adapted theoretical distribution. With this quantile (reasonable values are $0.8 \leq q_{max} \leq 1$) we are now able to calculate a maximum error time $t_{err,max}$. If we assume that $\min(S)$ is close to t_{min} we can define t_{max} as

$$t_{max} = \min(S) + t_{err,max}. \tag{2}$$

5 Used Models

In this evaluation we have used two theoretical distributions as the base for the models. The first one is the already mentioned Poisson distribution. The probability density function

$$p_k = \frac{\lambda^k}{k!} e^{-\lambda},\ (k = 0, 1, \ldots) \tag{3}$$

with the parameter lambda is fitted to the histogram of the error.

The second used model is the discretization of an originally continuous exponential distribution. We have seen in our experiments on different computer architectures that the type of the distribution differs from architecture to architecture. The observed error is sometimes distributed in a way that is similar to an exponential distribution. For the discretization of an exponential we will part the continuous function

$$f(x) = \begin{cases} 0 & : \quad x < 0 \\ \lambda e^{-\lambda x} & : \quad x \geq 0 \end{cases} \tag{4}$$

into windows of equal size.

By introducing a window size i in a way that

$$p_k = F((k+1) * i) - F(k * i), \tag{5}$$

where k is the class number we get

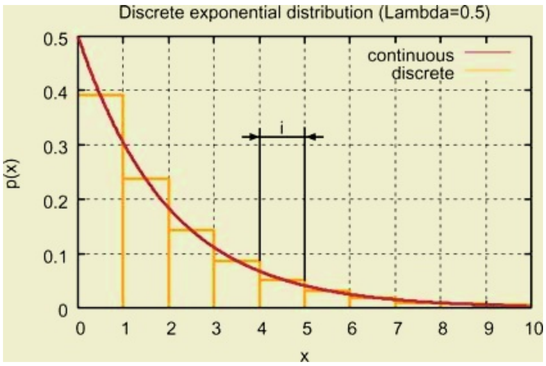

Fig. 1. Relationship between the continuous and discrete exponential distribution

$$p_k = e^{-\lambda k i}(1 - e^{-\lambda i}) \tag{6}$$

as a discrete exponential distribution. Some connections of the original and the discretization can be taken from Figure 1.

6 Fitting the Model to the Data

To actually fit a sample to a model we first create a histogram from the sample. To do this we need to sort the data into windows of a certain size. This window size should not be smaller than the resolution of the timer used to get the sample. Now we are able to use the maximum likelihood method

$$L(x_1, \ldots, x_n; \gamma) = p_1^{f_1}(\gamma) * p_2^{f_2}(\gamma) * \ldots * p_r^{f_r}(\gamma), \tag{7}$$

to fit the model to the data. The result of this process is a set of parameters for a model.

7 Quality of the Model Fitting

To determine how well a model fits to the distribution the χ^2 test is a way to check the quality of the model fitting in the case of discrete distributions. The χ^2 test uses a weighted sum of the square of the difference between the theoretical frequency and the practical frequency for each class.

$$\hat{\chi}^2 = \sum_{i=1}^{k} \frac{(B_i - E_i)^2}{E_i} \tag{8}$$

If the sum $\hat{\chi}^2$ does not exceed an upper limit, we accept the model fitting. This upper limit can be found in tables in adequate books (for example [12]) or can be approximated for those cases that use more than 30 classes. This approximation is defined as

$$\chi^2_\nu \approx \nu \left(1 - \frac{2}{9\nu} + z_\alpha \sqrt{\frac{2}{9\nu}}\right)^3 , \tag{9}$$

where ν are the degrees of freedom and z_α depends on the selected level of significance and has to be taken from an adequate table. We always try to fit each model to the sample and use the model with the lowest $\hat{\chi}^2$. If all models are rejected, there are two choices: generation of a new sample or building a new model.

To increase the quality of the fitting we can use function (9) as the objective function for an optimization

$$\min_{\lambda \in \mathbb{R}^+} \sum_i \|f_i - p_i(\lambda)\|. \tag{10}$$

In this case we can use the parameters we got from the maximum likelihood method as the start point for the optimization. The optimization itself will result in slight changes of this parameters and a better fitting of the model in the sense of the χ^2 test.

8 Creating a Complete Profile

One problem that arises while analyzing the MPI related behavior of different applications at different numbers of processors is that there are a lot of varying message sizes. It is not necessary to pay attention to each possible single message size at each possible processor count for each MPI function if the assumptions is made, that the application will always be analyzed with the same environment (hardware, MPI library, environment variables) the model has been built in. If this environment does not change during the model fitting (and the generation of the program traces later) we are able to define a discrete function

$$T_f : p, b \rightarrow t_{max} \tag{11}$$

for each MPI function that uses the number of used processors (p) and the number of bytes (b) to define the time t_{max} for this pair (p, b). In order to be able to evaluate T_f for each possible pair (p, b) we will define T_f for some base points and interpolate all other points from these. To keep the error introduced by this technique as low as possible the initial pairs (p, b) have to be chosen carefully. Within the BenchIt project [5] we have seen that for those communication relations the following heuristic will work well for a start:

– Numbers of processors: each power of two and a number in between. (2^n and $1.5 * 2^n$ with $n = \{1, 2, \ldots\}$)
– Message size: 1 Byte, 10, 20, ..., 100, 200, ..., 1000, and so on up to 4 MB

By using a bilinear interpolation between this base points we are now able to evaluate T_f for each reasonable pair (p, b).

9 Profile for a SGI Origin 3800

As a first example, we present the measurements for a subset of what we have proposed within the last section. We will use just a single processor count and a single message size to create a profile for a MPI_Allreduce() with one integer and eight processors on a SGI Origin 3800. For a first try, we have taken a sample

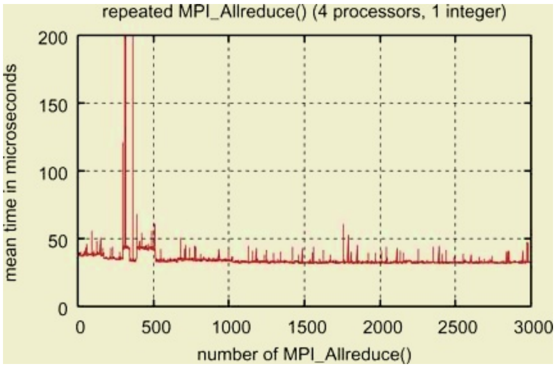

Fig. 2. Runtimes for repeated execution of a MPI_Allreduce() with one integer and eight processors on a SGI Origin 3800

of size 3000. This sample is shown in Figure 2. To actually see how large the sample really needs to be, we have fitted the models to subsets of this sample. The first subset size we used is 50. This size is increased by 10 (up to 3000). Each try produces a t_{max} which is plotted in Figure 3 against the subset size.

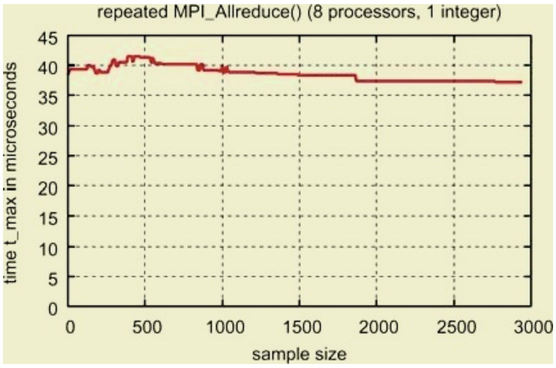

Fig. 3. Result of the model fitting for the model with the Poisson distribution plotted against the sample size

One conclusions from this example and our experiments at all [6] is that sample sizes around 200 are already enough to fit a model to the data. What we have seen during the experiments is that the error $\hat{\chi}^2$ is sensitive to the sample size and has always increased with it even when the result t_{max} is stable.

10 Application to Sweep3D

As a more practical example for the usefulness we have implemented a profiler that is able to trace each call of a MPI function back to a source code location. This source code location is either a source file and the associated line number or the name of the function that called this MPI function. By using the arguments applied to the MPI function (mainly the message size s and the execution time t) and the number of processors participating in a collective MPI function call it is possible to find the (p,b) pair for each call. Now we can evaluate T_f to figure out how long this call should have taken at maximum. By adding up t_{oh} for each source code location it is possible to automatically find the source code locations, where the application is wasting time during communication. These position will be a good entry point for optimizing the communication behavior.

Table 1. Total communication time in seconds for Sweep3D for different numbers of processors

numbers of processors	4	8	12	14	21	24	28
communication time	26.79	55.03	73.78	109.42	129.35	124.40	166.10

As an practical example we have chosen the Sweep3D application [9] from the ASCI benchmark collection. The communication time is mainly claimed by point to point communication. In Table 2 we see the overhead for those two lines in the source code in the source file `msg_stuff.cpp` that need most of the communication time.

Table 2. Overhead in seconds caused by the two source code lines performing the point to point communication within the file msg_stuff.cpp at different number of processors

		Numbers of processors						
Line	Function	4	8	12	14	21	24	28
311	MPI_Send()	5.9508	11.7368	9.3950	12.5326	13.9212	10.0274	12.2072
364	MPI_Recv()	18.0278	36.9847	54.6950	85.4923	98.2872	95.7525	129.9080

With this information and the knowledge about the total time spent during communication (Table 1) we can conclude that there is a late sender/late receiver problem dominating the communication time with the bigger fraction for the late sender. By replacing the blocking MPI function calls with non-blocking calls and one MPI_Waitall() we were able to attenuate this problem.

11 Conclusion and Future Work

We have shown that it is possible to use statistical distributions to model the overhead that occur during a repeated execution of a MPI function. By using a fixed quantile of the distribution we are able to define the time each MPI function call should not exceed. With this information we are able to find the overhead within communication relations. Mapping this overhead back to the source code location gives the user clues about good entry points for program optimization.

Future work will be the an automatic measurement for a complete MPI profile without the heuristics.

References

1. Dong H. Ahn and Jeffrey S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of Supercomputing 2002*, pages 1–16, 2002.
2. OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 2.5. `http://www.openmp.org/drupal/mp-documents/draft_spec25.pdf`, November 2004.
3. Luiz DeRose and Felix Wolf. CATCH - A Call-Graph Based Automatic Tool for Capture of Hardware Perfromance Metrics for MPI and OpenMP Applications. In *Euro-Par 2002*, 2002.
4. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, 1995.
5. Guido Juckeland, Stefan Börner, Michael Kluge, Sebastian Kölling, Wolfgang E. Nagel, Stefan Pflüger, Heike Rödling, Stephan Seidl, Thomas William, and Robert Wloch. BenchIt-Performance Measurement and Comparison for Scientific Applications. In *G. R. Joubert et. al., Eds., Advances in Parallel Computing, Vol. 13*, pages 501–508. Elsevier, 2004.
6. Michael Kluge. Statistische Analyse von Programmspuren für MPI-Programme. Diploma thesis, November 2004.
7. Andreas Knüpfer. A New Data Compression Technique for Event Based Program Traces. In *International Conference on Computational Science 2003*, pages 956–965, 2003.
8. Andreas Knüpfer and Wolfgang E. Nagel. Compressible Memory Data Structures for Event Based Trace Analysis. *Future Generation Computer Systems by Elsevier*, 2004.
9. Lawrence Livermore National Laboratory. The ASCI Sweep3D Benchmark Code. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html`, 1995.
10. Tom LeBlanc and Wagner Meira. Measurement and Prediction of Parallel Program Performance, http://www.cs.rochester.edu/u/leblanc/prediction.html. `http://www.cs.rochester.edu/u/leblanc/prediction.html`, 1997.
11. Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. In *Supercomputer 63, Volume XII, Number 1*, pages 69–80, 1996.
12. Lothar Sachs. *Applied Statistics: A Handbook of Techniques.* Springer, 11 edition, 2002.