Modeling the Communication Behavior on Distributed Memory Machines by Genetic Programming*

L. Heinrich-Litan, U. Fissgus, St. Sutter, P. Molitor, and Th. Rauber

Computer Science Institute, Department for Mathematics and Computer Science,
Martin-Luther-Universität Halle-Wittenberg, D-06099 Halle (Saale), Germany
email: <*name*>@informatik.uni-halle.de

Abstract. Distributed memory machines are still not broadly accepted although they provide large computing power. One of the reason is the costly development process for a specific parallel algorithm on a specific distributed memory machine. Due to the complicated runtime behavior caused by communication overhead and load imbalance finding an efficient parallel version of an algorithm often takes a lot of time. There is request for efficient and accurate runtime functions which easily predict the performance of the algorithms. The contribution of this paper is to show that runtime functions predicting the execution time of the communication operations necessary for any performance prediction mechanism can be generated by means of the genetic programming paradigm. The runtime functions generated dominate the runtime functions presented in literature, till today.

1 Introduction

Distributed memory machines (DMMs) provide large computing power which can be exploited for solving large problems or computing solutions with high accuracy. For the communication between the processors of DMMs, the message passing paradigm is usually used. Portable message passing standards as PVM [BDG+91] or MPI [MPI94] are available for a large variety of parallel machines und are widely accepted by application programmers. Nevertheless, DMMs are still not broadly accepted. One of the main reasons is the costly development process for a specific parallel algorithm on a specific DMM. Finding an efficient parallel version of an algorithm often takes a lot of time for programming and measuring different parallel versions of an algorithm. This is due to the fact that parallel algorithms on DMMs may show a complicated runtime behavior caused by communication overhead and load balance.

Thus, there is considerable research effort to model the performance of DMMs. This includes modeling the runtimes of communication operations with parametrized formulas [Jo91,HXA96,XH96,HEB96,RR97,FRR97]. In these papers,

* This work has been supported in part by DFG grant Ra 524/5 and Mo 645/5.

communication operators from different communication libraries have been investigated and compared for specific DMMs including the Connection Machine CM-2 [Jo91], the Cray T3D [XH96,HEB96], the IBM SP2 [XH96,RR97], and the Intel Paragon [XH96,FRR97]. The modeling of the execution time of the communication operations can be used in compiler tools, in parallelizing compilers or simply as concise information of the performance behavior for the application programmer.

The contribution of this paper is to show that runtime functions of high quality, which model the execution time of communication operations, can be modeled by the genetic programming approach [Ko92]. To illustrate the effectiveness of the approach we have chosen the paper of Rauber and Rünger [RR97] for comparison. In [RR97], collective communication operations from the communication libraries PVM and MPI are investigated and compared on the IBM SP2. The communication operations investigated include single-to-single and collective communication operations like single-broadcast, multi-broadcast, scatter, gather and reduction operations. For each operation, [RR97] considers several realizations within PVM and MPI exploiting mechanisms that are available to the application programmer, like packaging or buffering options. They model the execution time of the communication operations by runtime functions in closed form which depend on the message size and on some architectural parameters. The modeling itself has been done by curve fitting.

Since the 1960s, many optimization problems, which are very complex in nature and quite hard to solve by conventional optimization techniques have been attacked by stochastic optimization techniques which simulate the natural evolutionary process of human beings. These evolutionary algorithms (EAs) often outperform conventional optimization methods when applied to difficult real-world problems. We refer to [GC97] for an overview on complex problems in the area of industrial engineering, where EAs have been successfully applied. There are currently three main avenues of this research: genetic algorithms (GAs), genetic programming (GP), and evolution strategies (ES).

GP has been introduced by Koza [Ko92]. This approach attacks the problem of how computers can learn to solve problems without being explicitly programmed, i.e., how computers can be made to do what is needed to be done, without being told exactly how to do it. As any EA, GPs simulate the evolutionary process which occurs when the following four conditions are satisfied: (1) an entity has the ability to reproduce itself and to create offsprings either by crossover with another entity or by mutation; (2) there is a population of such entities; (3) there is some variety among the entities, and (4) some difference in ability to survive and to generate offsprings in the environment is associated with the variety. In the application of modeling the execution time of communication operations on DMMs, the entities are runtime functions and their ability to survive or to generate offsprings is some fitness function quantifying their deviation from the measured runtimes. The crossover operator is some merging of the chromosomes of two entities which represent the varieties of the entities. The mutation operator modifies a chromosome of an entity.

In this paper we demonstrate the effectiveness of GPs to model the execution time of communication operations on DMMs, by example. We investigate the IBM SP2 and some communication operations from the MPI library.

The paper is structured as follows. First, we give a brief overview on the investigations made by [RR97], where runtime functions have been provided for communication operations on the IBM SP2 (Section 2). Section 3 presents how performance modeling can be attacked by GPs. Experimental results are shown and discussed in Section 4. Especially we demonstrate that the GP approach generates runtime functions of higher quality than those published in literature till now.

2 The IBM SP2

In this section we summarize the results presented in [RR97]. First, we give an overview on the IBM SP2 architecture (Section 2.1) and on the communication operations from the MPI library on the IBM SP2 which we use in our experiments (Section 2.2). In Section 2.3, the model of the communication behavior on the IBM SP2 obtained by curve fitting in [RR97] is presented.

2.1 The architecture

The IBM SP2 is a DMM with up to 512 nodes each of which is a 66MHz POWER2 superscalar processor with a peak performance of 266 MFlops/s. The processors are connected by an SP2 High-Performance switch which is a bidirectional multistage interconnection network composed of switching elements [SSA+95]. Each basic switching element contains 8 receive modules, 8 send modules, an 8×8 crossbar, and a central queue for storing messages that cannot proceed because of contention (see Figure 1). The 8 input and output ports are paired together resulting in 8 bidirectional ports to connect to SP2 links.

Up to 8 processors can be attached to a network consisting of exactly one switching element. Systems with more than 8 processors use SP2 switch boards. A switch board contains 8 switching elements organized in 2 stages of 4 switching elements. Up to 16 processors are connected to one side of a switch board. For systems up to 80 nodes, the 16 unused links (which all are located on one side) of a switch board are connected directly to the unused links of other switch boards (up to 5 switch boards). For networks containing more than 80 processors, the unused links connect to intermediate stages of switch boards by a Banyan network. For illustration, please see Figure 1.

Routing is realized by buffered cut-through wormhole routing, i.e., messages are divided into packets which are in turn divided into 8-bit flow-control digits, the flits.

The specific execution platform that [RR97] used for their experiments is an IBM SP2 with 37 nodes and 128 MByte main memory per processor from GMD St.Augustin, Germany.

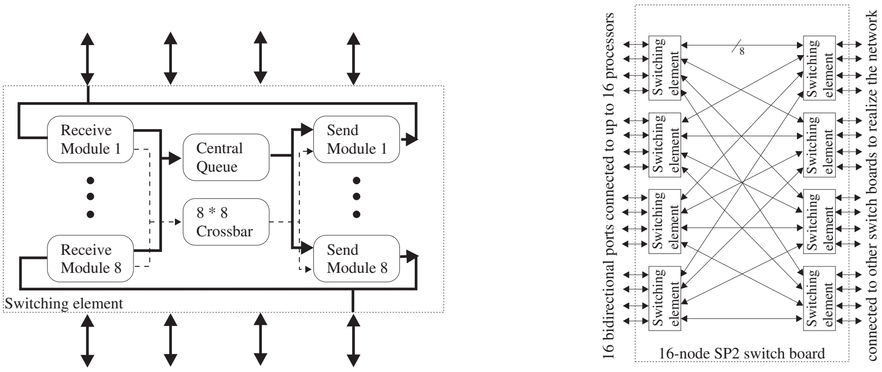

Fig. 1. Switching element and switch board of the IBM SP2

2.2 Communication Operations

The paper [RR97] uses IBM MPI-F, Version 1.41 [FWR+95], and IBM PVMe for AIX, Version 2 Release 1 [IBM95], in their experiments. They investigate and compare different communication operations from these libraries. Due to the limited space, we consider only IBM MPI-F and only report on two communication operations for which we present the runtime functions generated by our GP algorithm to demonstrate the effectiveness of our approach.

Single-transfer operation: In MPI, the standard point-to-point communication operations are the blocking `MPI_Send()` and `MPI_Recv()`. For the blocking send, the control does not return to the executing process before the send buffer can be reused safely. MPI leaves it to the implementation whether the standard send uses an intermediate system buffer. A buffered send is obtained by the operation `MPI_Bsend()`. This has the advantage that the control can be returned to the calling process after the copy operation to a system buffer has been performed, whether or not a matching receive operation has been executed by the receiver. A synchronous send is performed by `MPI_Ssend()`. It will complete successfully only if a matching receive has been posted, and the receive operation has started to receive the message.

Here, we will only consider the standard single-transfer operation from the MPI library to illustrate the effectiveness of genetic programming applied to runtime prediction. The results for the other transfer operations are of same quality.

Scatter operation: A single-scatter operation is executed by the global operation `MPI_Scatter()` which must be called by all participating processes. The effect is that the specified root process sends a part of its own local data to each other process.

2.3 Runtime functions generated by curve functions

In [RR97], the runtimes of the communication operations from MPI on the IBM SP2 are modeled by runtime formulas found by curve fitting that depend on various machine parameters including the number of processors, the bandwidth of the interconnecting networks, and the startup times of the corresponding operations. The investigations made in [RR97] resulted in the parameterized runtime formula

$$f^{cf}_{scatter}(p, b) \;=\; 10.28 \cdot 10^{-6} + 91.59 \cdot 10^{-6} \cdot p + 0.030 \cdot 10^{-6} \cdot p \cdot b \;\; [\mu\text{sec}]$$

for the scatter operation, and

$$f^{cf}_{s-transfer}(b) \;=\; 211.8 \cdot 10^{-6} + 0.030 \cdot 10^{-6} \cdot b \;\; [\mu\text{sec}]$$

for the single-transfer operation. The values b and p are the message size in bytes and the number of processors, respectively.

In [RR97], the authors state that the runtime formulas give a quite accurate description of the execution time of the communication operations. Detailed information on this with respect to the scatter operation and the single-tranfer operation can be found in Section 4.1.

3 The Genetic Programming Approach

The usual form of genetic algorithm was described by Goldberg [Go89]. Genetic algorithms are stochastic search techniques based on the mechanism of natural selection and natural genetics. They start with an initial population, i.e., an initial set of random solutions which are represented by chromosomes. A chromosome is usually a binary string of symbols. The chromosomes evolve through successive iterations, called generations. During each generation, the solutions which are represented by the chromosomes of the population are evaluated, using some measures of fitness. To create the next generation, new chromosomes, called offsprings, are formed by either merging two chromosomes from the current generation using a crossover operator, modifying a chromosome using a mutation operator, or copying a chromosome of the current generation. The chromosomes involved in the generation of the new population are selected according to their fitness values. Fitter chromosomes, i.e., chromosomes whose phenotypes have high fitness, have higher probabilities of being selected. After several generations, the algorithms converge to the best chromosome which hopefully represents the optimum or suboptimal solution to the problem. The general structure of GAs is described in Figure 2. $P(t)$ is the population in current generation t.

GAs whose chromosomes are syntax trees of formulas, i.e., computer programs, are GPs. They have been introduced by Koza [Ko92] to attack one of the central questions in computer science, namely: How can computers learn to solve problems without being explicitly programmed?

In the next sections we present the GP we have applied for the problem of finding runtime functions of high quality modeling the execution time of the

```
genetic_algorithm()
{ t=0;
  initialize(P(t));
  evaluate(P(t));
  while (not termination condition) do
  { P(t+1) = generate(P(t));
    evaluate(P(t+1));
    t = t+1;
  }
}
```

Fig. 2. General structure of GAs

MPI communication operations on the IBM SP2. We use terminology given in [Ko92].

3.1 The chromosomes

In genetic programming, a chromosome is an element of the set of all possible combinations of functions that can be composed recursively from a set F of basic functions and a set T of terminals. Each particular function $f \in F$ takes a specified number $z(f)$ of arguments, specifying the arity of f. The basic functions we allowed in our application are addition ($+$) and multiplication ($\cdot$) both of arity 2, and the operations sqr, $sqrt$, ln, and exp, all of arity 1. The set T is composed of one or two variables p and b depending on whether the measured data are taken over only the parameter p which specifies the number of processors or over the parameter p and the parameter b which specifies the message size, and the constants 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0.1, and 10^{-6}. The reason we put constant 10^{-6} in set T is due to the observation that the values of the measured data were about factor 10^{-6} smaller than the values of the parameters. As a chromosome is a well defined expression, we represent each chromosome by the corresponding syntax tree. Figure 3 shows the chromosome $(x + (2 + (2 \cdot 0.1))) \cdot \sqrt{4}$ which has the function $f : \mathbf{R} \rightarrow \mathbf{R}$ defined by $f(x) = 2 \cdot x + 4.4$ as phenotype.

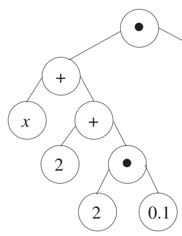

Fig. 3. A chromosome

In the following, we identify a chromosome with the runtime function which is represented by that chromosome in order to make the diction easier.

3.2 Fitness function

Fitness is the driving force of Darwinian natural selection and, likewise, of genetic programming. It may be measured in many different ways. The accuracy of the fitness function with respect to the application is crucial for the quality of the results produced by GP. In the application handled here experiments have shown that the average percental error of a chromosome as fitness results in best solutions. Of course, as the aim is to minimize the average percental error, a chromosome is said to be fit if its average percental error is small. Let Pos be the set of measuring points and $measured(\alpha)$ be the measured data at measuring point α, then the percental error $error_f(\alpha)$ of a runtime function f at point α and the average percental error $error_f$ of f are given by

$$error_f(\alpha) = \frac{abs(measured(\alpha) - f(\alpha))}{measured(\alpha)} \cdot 100$$

and

$$error_f = \frac{\sum_{\alpha \in Pos} error_f(\alpha)}{\mid Pos \mid},$$

respectively.

There are two problems we have to care about when applying this fitness function: The populations can contain chromosomes which are very bad, e.g., whose average percental error is greater than the maximal decimal value `DBL_MAX` (which is about $1.79 \cdot 10^{308}$ in GNU g++). In this case, we have to redefine the error of these chromosomes to be some large value, e.g., `DBL_MAX/` $\mid P \mid$ where $\mid P \mid$ denotes the size of the populations which is constant during GP evolution. The other problem is that some operations of the syntax trees are not defined, e.g., $\ln a$ and $\sqrt{b}$ with $a \leq 0$ and $b < 0$. The fitness function of our GP replaces these faulty operations by some constants and adds a penalty to the error. Chromosomes which contain faulty operations are not taken as final result of the GP.

3.3 Crossover and mutation operators

The crossover (sexual recombination) operation for GP creates variation in the population by producing new offsprings that consist of parts taken from each parent. It starts with two parental syntax trees, which are selected with error-inverse-proportionate selection, and produces two offspring syntax trees by independently selecting, using a uniform probability distribution, one random point in each parent to be the crossover point for that parent. The offsprings are produced by exchanging the crossover fragments. The crossover operation is illustrated in Figure 4.

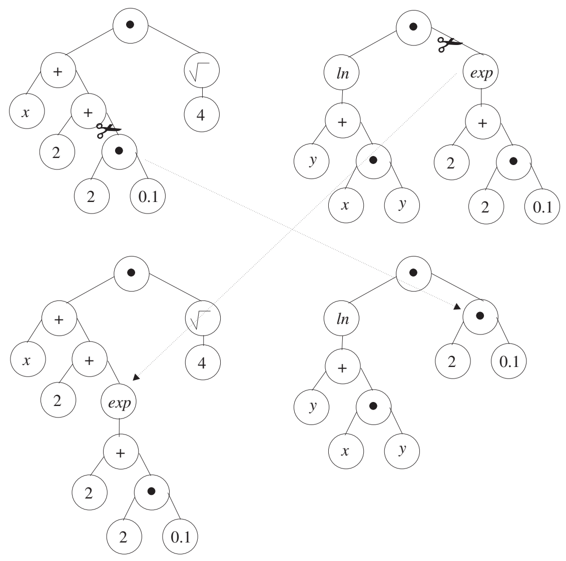

Fig. 4. Crossover operator

To direct the algorithm to generate smooth runtime functions we restricted the solution region to syntax trees of depth up to a constant c. As one of the two offsprings can have depth greater than c even if both parents have depth less than or equal c, the crossover operation we used only generates the offspring which has allowed depth. If both offsprings have depth less than or equal c, then the new crossover operation chooses one at random.

The mutation operation introduces random changes in structures randomly selected from the population. It begins by selecting a point within the syntax tree by random. Then, it removes the fragment which is below this mutation point and inserts a randomly generated subtree at that point. The mutation operation also takes attention that the depth of the new syntax tree is not greater than the maximal tree depth allowed.

3.4 Initialization and Termination

There are several methods to initialize the GP, i.e., to generate a first population, whose size is determined by the user. In our experiments we generate each chromosome of the first population by random. We only pay attention that the depth of each syntax tree is less than or equal constant c. The chromosomes are generated top down. The semantics of a node v of depth less than c is taken from $F \cup T$ by random. If a node has depth c, the semantics is randomly taken from set T.

In our experiments, the GP stops after having considered a predefined number of generations.

4 Experimental Results

First, we begin by quantifying the quality of the runtime functions given in [RR97] using the fitness function given in Section 3.2. Then, we compare these runtime functions, which have been found by curve fitting, to those generated by genetic programming and discuss the results. We close the section by presenting some runtime formula generated by GP.

4.1 Deviation of the runtime functions from literature

We evaluated the runtime functions (see Section 2.3) given in [RR97] by using the fitness function specified in Section 3.2. The test set of measured data contained communication times for $p = 4, 8, 16, 32$ processors and message sizes between 4 and 800 bytes with stepsize 20, between 800 and 4.000 with stepsize 400, and between 4.000 and 240.000 with stepsize 4.000. Table 1 presents the results. The first column specifies the communication operation, the second column shows the average percental error of the corresponding runtime function from [RR97]. Columns 3 to 9 give the percentage of measuring points $\alpha \in Pos$ for which the percental error $error_f(\alpha)$ is less than x for $x = 0.01$, 0.1, 1, 10, 25 50 and 100, respectively.

Table 1. Deviations of the runtime functions found by curve fitting

Operation	$error_f$	percentage of those $\alpha \in Pos$ with $error_f(\alpha) < x$						
		< 0.01	< 0.1	< 1	< 10	< 25	< 50	< 100
scatter	57.36	0.0	0.6	2.6	23.8	49.0	72.5	83.4
single-transfer	71.90	0.0	1.2	20.9	47.4	59.4	67.4	73.4

Taking a closer look at the runtime functions modeled by [RR97], we observe that the runtime functions predict the execution time of the operations very good if the value of parameter b, which specifies the message size, is large. However, the average percental error is relative large for small message sizes. Since the number of small messages in the test set is large, the average percental error is large. This observation will be confirmed by the comparison between these runtime functions and those generated by GP (see Section 4.2).

4.2 Deviations of the runtime functions generated by GP

There are some parameters of our GP which have to be set, namely the size of each population (`size`), the number of generations which have to be realized (`generations`), the maximum depth of syntax trees (`max_depth`), and the probabilities for applying crossover ($p_{crossover}$), mutation ($p_{mutation}$), and the copy operation (p_{copy}). By setting `size`$= 20$, `generations`$= 12000$, `max_depth`$= 15$,

$p_{crossover} = 80\%$, $p_{mutation} = 10\%$, and $p_{copy} = 10\%$, we obtained runtime functions by GP which dominate by far the runtime functions from [RR97]. Table 2 shows the deviations of the best runtime function generated after having run the GP algorithm 3 times, which takes about 6 hours on a SUN SPARCstation Ultra 1, 128 MByte RAM.

Table 2. Deviations of the runtime functions generated by GP

Operation	$error_f$	percentage of those $\alpha \in Pos$ with $error_f(\alpha) < x$						
		< 0.01	< 0.1	< 1	< 10	< 25	< 50	< 100
scatter	13.72	0.7	2.7	14.6	64.3	84.8	92.7	100.0
single-transfer	15.65	0.3	1.7	20.6	55.7	74.8	92.5	100.0

Figure 5 graphically compares the deviation of the runtime function from [RR97] found by curve fitting and the deviation of the runtime function generated by GP for the scatter operation with respect to some set of measured data. The curves differ very much for small values of parameter b whereby the curve generated by GP dominates by far the curve found by curve fitting. For large parameter values, both runtime functions predict the execution time of the communication operation roughly alike. Same observations can be made when comparing the deviations of the runtime functions for other communication operations. Figure 6 shows the comparison of the deviations of the runtime functions found by curve fitting and GP, respectively, for the single-transfer operation with respect to several sets of measured data. Note that the runtime functions for the single-transfer operation only depend on message size b. In order to better illustrate the observation that the curve generated by GP dominates the curve found by curve fitting for small parameter values, Figure 6 only shows the cut-out determined by the small message sizes (up to 500 byte).

Let us discuss the observation just made. The typical way to derive runtime functions with the curve fitting approach is to inspect the measured runtimes for a specific communication operation, to select a parametric function with a similar shape (or to use a function derived by a theoretical analysis of the underlying network), and to determine the coefficients of the function such that the measured runtimes fit best to the selected function. The disadvantage of this approach is that the result of the curve fitting strongly depends on the selected base function and that usually only one (simple) function is selected to model a communication operation for a large range of message sizes. This may lead to poor results in some regions since often different methods are used to collect the results of the communication operations for different message sizes. The GP approach on the other hand allows for generating complex functions that describe the usually complex shape of the measured data quite accurately.

We close the section by presenting the runtime formula generated for the scatter operation. The runtime formula $f^{GP}_{scatter}$ predicting the execution time of

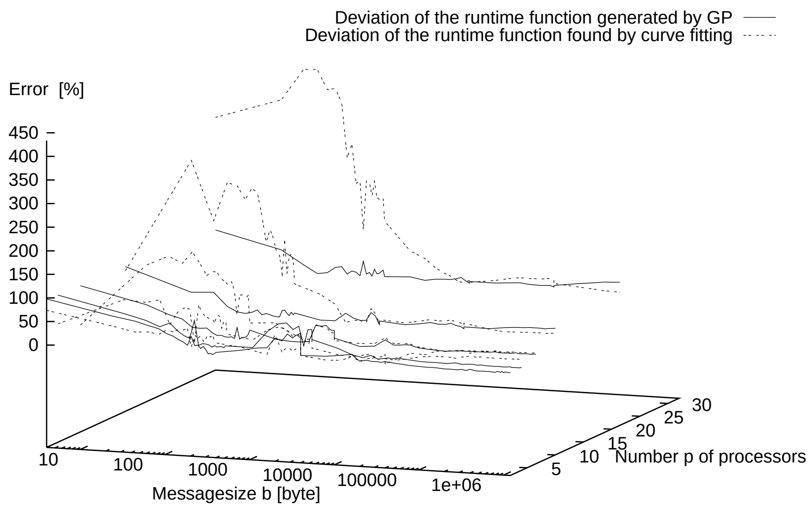

Fig. 5. Comparison between the deviation of the runtime function $f^{cf}_{scatter}$ found by curve fitting and the deviation of the runtime function $f^{GP}_{scatter}$ generated by GP for the scatter operation.

the scatter operation which has been generated by GP is given by

$$
\begin{aligned}
f^{GP}_{scatter}(p,b) \;&= \\
&= \;10^{-6}\cdot\sqrt{b\cdot p\cdot\sqrt{\sqrt{p}-1}}\;\cdot \\
&\quad \sqrt{\left(b\cdot p\cdot 0.000625+0.00005\cdot b+0.00375\cdot p^2+0.003\cdot p+\ln\left(\sqrt{e^p}\right)+49\right)}.
\end{aligned}
$$

It shows that the runtime formulas generated by GPs are rather complex and difficult to discuss. (The runtime formula $f^{GP}_{s-transfer}$ generated for the single-transfer operation is even more complex.) To our opinion, this is the only disadvantage of the approach we have presented.

5 Conclusions

We showed by example that genetic programming provides runtime functions predicting the execution time of communication operations of DMMs which dominate the runtime functions found by curve fitting.

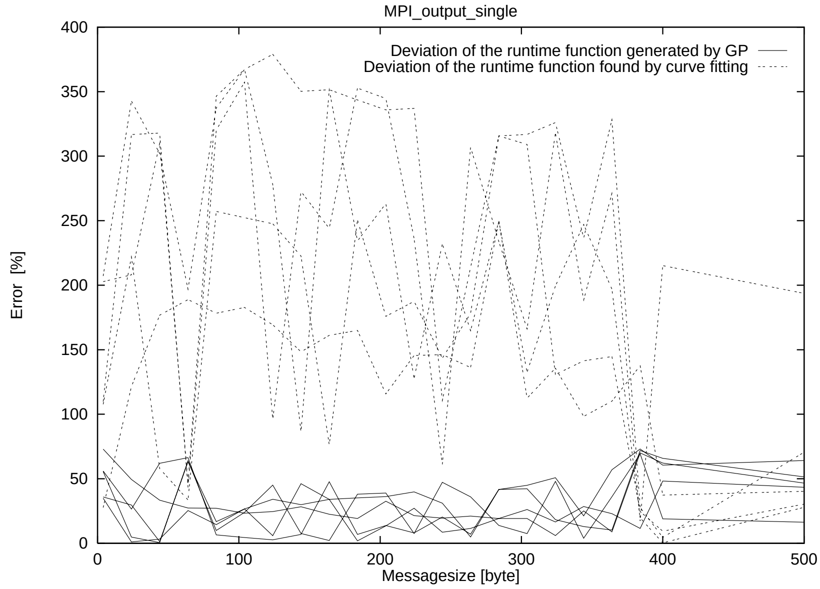

Fig. 6. Comparison between the deviation of the runtime function $f^{cf}_{s-transfer}$ obtained by curve fitting and the deviation of the runtime function $f^{GP}_{s-transfer}$ generated by GP for the single-transfer operation.

The next step of our investigations we will make is to automatically generate different runtime functions for different intervals of the parameters. The intervals themselves have to be determined by genetic programming, too. Using such non uniform runtime functions seems to be necessary to predict the execution functions more accurately, as DMMs use various communication protocols depending on the message size.

References

[BDG+91] Beguelin, A., Dongarra, J., Geist, G., Manchek, R., and Sunderam, V.: A User's Guide to PVM: Parallel Virtual Machine. Technical Report ORNL-TM-11826, Oak Ridge National Laboratory, 1991.

[FRR97] Foschia, R., Rauber, Th., and Rünger, G.: Prediction of the Communication Behavior of the Intel Paragon. In *Proceedings of the 1997 IEEE MASCOTS Conference*, pp.117-124, 1997.

[FWR+95] Franke, H., Wu, C., Riviere, M., Pattnaik, P., and Snir, M.: MPI Programming Environment for IBM SP1/Sp2. In *Proceedings of ICDCS 95*, Vancouver, 1995.

[GC97] Gen, M., and Cheng, R.: *Genetic Algorithms* & Engineering Design. John Wiley & Sons, Inc., New York, 1997.

[Go89] Goldberg, D.: *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley, Reading, MA, 1989.

[HEB96] Hu, Y., Emerson, D., and Blake, R.: The communication performance of the Cray T3D and its effect on iterative solvers. *Parallel Computing*, 22:829-844, 1996.

[HXA96] Hwang, K., Xu, Z., and Arakawa, M.: Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):522-536, 1996.

[IBM95] IBM PVMe for AIX. User's Guide and Subroutine Reference, Version 2, Release 1. Technical Report GC23-3884-00, IBM Corporation, October 1995.

[Jo91] Johnson, L.: Performance Modeling of Distributed Memory Architecture. *Journal of Parallel and Distributed Computing*, 12:300-312, 1991.

[Ko92] Koza, J.: *Genetic Programming*. The MIT Press, 1992.

[RR97] Rauber, Th., and Rünger, G.: PVM and MPI Communication Operations on the IBM SP2: Modeling and Comparison. In Proceedings of HPCS 97, 1997.

[SSA+95] Stunkel, C., Shea, D., Abadi, B., Bender, M., Grice, D., Hochschild, P., Joseph, D., Nathanson, B., Swetz, R., Stucke., R, Tsao, M., and Varker, P.: The SP2 High-Performance Switch. *IBM System Journal* 34(2):185-204, 1995.

[MPI94] The MPI Forum. MPI: A Message Passing Interface Standard. Technical report, University Tennessee, April 1994.

[XH96] Xu, Z., and Hwang, K.: Early Prediction of MPP Performance: SP2, T3D and Paragon Experiences. *Parallel Computing*, 22:917-942, 1996.