Estimating Parallel Performance

Oleg Lobachev[a,1,*], Michael Guthe[b], Rita Loogen[b]

[a]*Scuola Superiore Sant'Anna, TeCIP Institute, Real-Time Systems Laboratory*
Via G. Moruzzi 1, 56124 Pisa, Italy
Fon. +39 (0) 50 549 2257, Fax +39 (0) 50 882 003, `o.lobachev@sssup.it`
[b]*Philipps-Universität Marburg, Fachbereich Mathematik und Informatik*
Hans-Meerwein-Straße, D-35032 Marburg, Germany
`{guthe, loogen}@informatik.uni-marburg.de`

Abstract

In this paper we introduce our estimation method for parallel execution times, based on identifying separate "parts" of the work done by parallel programs. Our run time analysis works without any source code inspection. The time of parallel program execution is expressed in terms of the sequential work and the parallel penalty. We measure these values for different problem sizes and numbers of processors and estimate them for unknown values in both dimensions using statistical methods. This allows us to predict parallel execution time for unknown inputs and non-available processor numbers with high precision. Our prediction methods require orders of magnitude less data points than existing approaches. We verified our approach on parallel machines ranging from a multicore computer to a peta-scale supercomputer.

Another useful application of our formalism is a new measure of parallel program quality. We analyse the values for parallel penalty both for growing input size and for increasing numbers of processing elements. From these data, conclusions on parallel performance and scalability are drawn.

Keywords: parallel run time estimation, scalability measure

1. Introduction

Since Amdahl's law [3, 25] the quest for modelling parallel performance is open. Kumar and Gupta presented a nice summary of existing approaches [35, 22]. We suggest a model for a subdivision of parallel execution time into "good" and "bad" parts. Contrary to the popular thought of parallel run time being the sequential one "sped up" to some factor less than the number of processing elements, we envision parallel run time as the sequential "work" distributed over a number of processing elements plus an additional penalty term.

[*]Corresponding author.
[1]Most of the work for this paper has been done at Philipps-Universität Marburg.

The first goal of this paper is an accurate prediction of parallel run times for new input sizes and for non-available numbers of processors. Our approach is to measure additionally the sequential work and to obtain the parallel overhead for a set of sample input sizes or sample numbers of processors. Statistical techniques are then used to extrapolate and to estimate the values for further input sizes or other numbers of processors. As the parallel execution time is straightforwardly expressed in terms of these values, this enables the precise forecast of parallel run time. To our knowledge we are the first to propose such kind of a division of parallel execution time into components. Related approaches either assign cost to basic elements of a parallel program [7, 47, 15, 40] or predict execution time as a single entity [48, 28, 27]. Our method differs from both these research directions. We do not use that many components; they are not directly expressing, e.g., message passing latency or network throughput. But still we are splitting the parallel execution time into sequential work and parallel overhead, instead of predicting the parallel execution time directly. A further benefit: our approach, basing on statistical techniques, requires orders of magnitude less data points than other approaches from the literature. We also call the parallel overhead term a "parallel penalty".

Secondly, we introduce here the parallel overhead term. Used as a measure for the scalability, it provides insight into the performance properties of parallel programs. Is the "bad" part increasing, bottlenecks or similar problems in the code are likely. However, identifying them is beyond the scope of this work. A related approach is the serial fraction [30], however, as discussed in Section 5, it is different from the parallel penalty.

We show the practicality of our approach for selected programs from scientific computing, either implemented in C+MPI [49] and run on a supercomputer (Section 4.2) and on a network of Sun workstations (Section 4.6), or implemented in the parallel functional programming language Eden [41] and run on multicore machines (Sections 4.1, 4.3, 4.4, and 4.5). The latter programs are parallelised using standard algorithmic skeletons [14] from Eden's skeleton library [40, 23]. Although our Eden system is used for the majority of the experiments, our approach is completely language-independent. We chose our applications to be non-trivial, in fact we use real-life codes from parallel computing projects.

Our technique—as this paper demonstrates—is applicable to any parallel system, ranging from a multicore machine to a supercomputer. In a contrast to a typical execution time prediction, we treat workload–hardware combination as a black box. This is novel. We *intentionally* avoid using the knowledge of an algorithm structure—though specialisations of our method are viable [38].

This paper is structured as follows. In Section 2, we present formulae for expressing execution time and work. Section 3 discusses the way of estimating the workload and parallel penalty from a number of execution time measurements. We present six scientific computing examples and predict their execution times in Section 4. Section 5 compares our approach with the serial fraction approach by Karp and Flatt [30]. Further related work is considered in Section 6, Section 7 gives an outlook on future work, Section 8 concludes.

2. Execution Time Estimation

Let n denote the input size[2] and p the number of processing elements (PEs). The work of a program is denoted by $W(n)$, the sequential execution time by $T(n)$. We assume that $T(n) = W(n)$. The common notation for execution time on p PEs is $T(n,p)$. We denote the work done with p PEs by $W(n,p)$ and assume that $W(n,p) = p\,T(n,p)$. In a parallel execution, the sequential work is distributed over p PEs. The distribution causes a total parallel overhead denoted by $A(n,p)$ (see [22]) which is however also distributed over the parallel PEs. We call the parallel overhead per PE, $\bar{A}(n,p)$, i.e., $A(n,p) = p\,\bar{A}(n,p)$. We can now express $T(n,p)$ as

$$T(n,p) = T(n)/p + \bar{A}(n,p). \tag{1}$$

The total amount of work performed on p PEs is

$$W(n,p) = T(n) + p\,\bar{A}(n,p) = T(n) + A(n,p).$$

Our goal is to find good approximations for $T(n)$ and $\bar{A}(n,p)$ to estimate the parallel run time $T(n,p)$ using Equation 1. Moreover, we will use $\bar{A}(n,p)$ as a measure for scalability. As $\bar{A}(n,p)$ depends on two parameters, we will investigate the behaviour of $\bar{A}(n,p)$ depending on one of its parameters while the other one is fixed.

The distinction between the sequential time $T(n)$ and the "parallel" time on a single PE $T(n,1)$ is essential for distinguishing between the *absolute speedup* $T(n)/T(n,p)$ and the *relative speedup* $T(n,1)/T(n,p)$, the latter usually being higher than the former because of the overhead of the parallel system on a single PE. Analogously we distinguish between an *absolute reference point* for our estimations, using sequential time $T(n)$, and a *relative reference point,* when $T(n,1)$ is used.

The parallel overhead may be significantly different on different PEs, which seems to be not reflected in our model. Our goal is not to capture an array of *local* parallel overheads for each of the PEs, but to capture how the total parallel overhead A *changes* with increasing number of processors p. The relation of the total parallel overhead $A(n,p)$ to our measure $\bar{A}(n,p) = A(n,p)/p$ is similar to the relation of the speedup $T(n)/T(n,p)$ to the efficiency $T(n)/(p\,T(n,p))$.

3. Methodology

Estimation procedure. We estimate $T(n)$ and $\bar{A}(n,p)$. Our aim is to find separate approximations for these two terms, as $T(n)$ and $\bar{A}(n,p)$ have a different nature. In order to do so, we determine several values of $T(n)$ and $\bar{A}(n,p)$, then we use statistical techniques on the resulting data sets. The results are combined to form an estimation of $T(n,p)$ using Equation (1).

[2]The value of n is the input size in our programs, however for the statistical methods it is rather an unique designation of the input. For two radically different, e.g., randomly generated, inputs of approximately the same size the values of n should be different.

How do we obtain values for $\bar{A}(n,p)$? By transforming Equation 1 we get

$$\bar{A}(n,p) = T(n,p) - T(n)/p.$$

Thus, we can compute the parallel overhead per PE from the total parallel run time for p PEs minus the sequential run time divided by p, where these values can be measured or estimated. Note that $\bar{A}(n,p)$ depends on p.

The shape of $\bar{A}(n,p)$ is the key to rating the parallel performance *quality*. As this penalty term depends on both the problem size n and the number of PEs p, it is important that it does not increase for growing p. Otherwise, the implementation does not scale well. We detail on this in Section 5.

We perform either an estimation w.r.t. the input size n or w.r.t. the number of PEs p. In the first case we predict $T(n)$ and $\bar{A}(n,p)$ w.r.t. n. In the case of an estimation w.r.t. p we typically already have the corresponding value of $T(n)$ and predict the second component, the parallel penalty $\bar{A}(n,p)$ w.r.t. p.

Statistical methods. We use different methods to predict values of $T(n)$ and $\bar{A}(n,p)$ for non-measured input sizes. Note however, this paper is not about statistical techniques, but about applications thereof. We *could* have used straightforward polynomial interpolation, but for better results we sample more points and use one of the following methods. One approach is cubic spline interpolation [17], another one is local polynomial regression fitting [9, Chapter 8] (*cf.* [11, 12]). We refer to these approaches using the R function names spline and loess [45]. Also we use linear model fitting with orthogonal polynomials constructed from the actual input [9, Chapter 4]. We denote this approach with lm(poly). A simple linear model fitting is just lm. Finally, mean is not a real method, but the mean of the two best methods for a particular approach.

The prediction methods we use in this paper exhibit different behaviour. The spline method interpolates the measured data points exactly, while the other methods utilise regression fitting. The latter means that it is not attempted to fit all the input data points, but rather to capture the "trend". The method lm tries to fit a straight line, hence it is less appropriate for our purposes. Its generalisation lm(poly) uses orthogonal polynomials to weaken this drawback. The loess method is a modern statistical approach to polynomial regression. It is local, so distant data points have little influence on the shape of the fitted curve. loess is similar to spline with respect to this property.

We need to choose the most fitting prediction method every time at least one of the parameters changes—be it hardware, application, range of task sizes, or range of processors. We perform estimations using *all* prediction methods in our repertoire and then choose the best fit. The choice can be mechanised using the procedure below. Once we have decided on the best prediction method for the given parameters, we can use this method to predict the actual values. This choice is done separately for the sequential time and parallel overhead.

Mechanising the choice. The decision on the best estimation method *can* be done automatically. Given an $\varepsilon > 0$ and a set of known values, we predict a known(!) value with other ones using various methods.

- First, discard methods producing nonsense results, e. g., time estimation < 0.
- If some of the remaining methods produce a relative error $< \varepsilon$, we will pick one with the smallest relative error.
- If none of the methods produces a relative error $< \varepsilon$, but the relative error of the mean of the two nearest methods w. r. t. the actual value is $< \varepsilon$, we will pick the mean.
- If none of the methods yields a satisfying result, reconsider ε. Make further measurements. Otherwise fail.

Using this algorithm we can decide on the prediction method to be used to estimate not measured values. If we have to resort to the mean of two methods, two strategies can be used to choose the best two methods. Both require some "training": we need to predict few known values first. Then, for predicting an unknown value, we use the information from the training. Either we pick the mean of the two methods, which produced best results in the training. Or we decide in the training phase on *three* best methods. Then, in the "real life" estimation we would discard the more distant value and compute the mean of the two remaining values.

The black box. Changing the range of input sizes, the range of PEs, the hardware or the program would probably lead to a different statistical method to be the best fit in this situation. But this does not matter for our methodology. We will see below that we can determine the best fit for the *given setting* with a significant accuracy. Our case studies show that even doubling the input size does not harm the successful prediction with the method that we designated as a best fit for this setting—see Section 4.3. It is not very hard to decide on this best fit: it suffices to perform an estimation with all four available methods and to choose the best.

We trade the dependency from the parameters of a particular set of program executions against an extreme "universality" of our method: it is applicable to any parallel program on any parallel platform. All the knowledge of the local particularities of the hardware and of the software is encoded in the set of estimators, but not in the approach itself. We regard this a very beneficial exchange.

4. Experiments

To support our presentation with practical evidence, we present multiple experimental case studies. We chose the example programs for our experiments to cover popular "patterns" of parallel computation. We regard multiple parallel maps with various kinds of load balancing, a divide and conquer method, examples of a parallel iteration. These are supplemented by applications for large-scale scientific computing. These include peta-scaled lattice-Boltzmann method and an

n	10	20	30	40	50	60	70	80	90	$\boxed{\mathit{100}}$	**120**	**150**
$T(n)$	0.02	0.13	0.39	0.82	1.48	2.44	3.76	5.50	8.07	11.03	**19.14**	**38.36**
$T(n, 8)$	0.02	0.06	0.14	0.29	0.51	0.79	1.18	1.75	2.39	$\boxed{\mathit{3.48}}$	**5.74**	**10.55**

Figure 1: Gauß elimination. Measured time. **Bold** items will be forecast w. r. t. n, $\boxed{\textit{boxed}}$ items will be forecast w. r. t. p.

implementation of a linear solver. For more examples and an in-depth treatment of the parallel computing "patterns" for scientific computation we refer to the PhD thesis of the first author [36]. Such patterns can be expressed as algorithmic skeletons [14]. We state the kind of the skeleton if it is relevant for the the case studies presented here. The analysed program is not required to be implemented using skeletons. In fact, we have no requirements at all for an analysed program—another strong point of our approach.

Hardware and measurement methodology. Most our run time experiments have been performed on an eight core 64 bit Intel Xeon machine with 16 GB RAM. We have used the Glasgow Haskell Compiler for the sequential program executions and the parallel Eden extension of this compiler for the parallel executions. We always determined the mean run time of five program runs. We started the programs with default settings for memory allocation and garbage collection.

However, Section 4.2 presents results from the measurements conducted from a physical simulation on a supercomputer. These data originate from the Jülich Blue Gene/P machine. It is built using a system-on-a-chip approach with quad core PowerPC chips with 2 GB of RAM as the base. Each core is a 32 bit processor, running at 850 MHz. So, a single node is a traditional multicore processor. The nodes are assembled into racks with 1024 nodes in each.

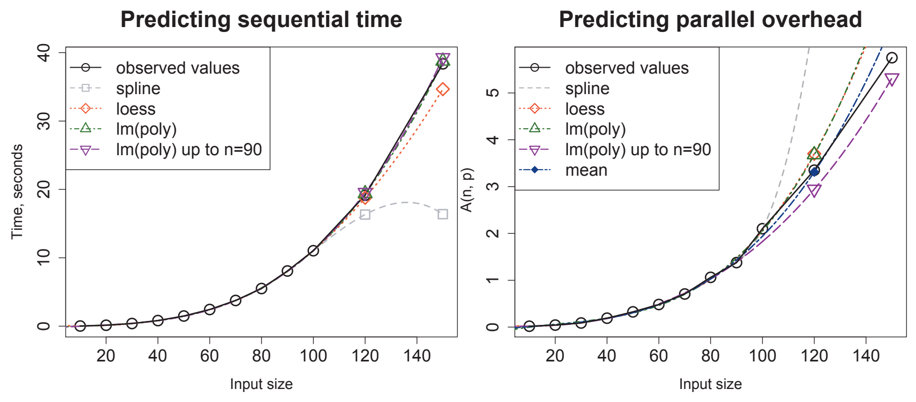
Further, Section 4.6 presents the data, obtained from the tests of a linear solver on a SMP machine with 18 UltraSPARC II processors, working at 400 MHz. The machine had 18 GB of RAM. Please refer to the paper by Gondzio and Sarkissian [20] for details.

Section 4.4 also features some Eden-based experiments on an AMD multicore machine with 48 cores and 64 GB of RAM. All our Eden methodology applies here.

4.1. Gauß Elimination – Parallel Map

We have measured the time needed to compute the LU decomposition of a permuted scaled $n \times n$ Pascal matrix modulo r primes. The program has been parallelised using the simple `farm` skeleton with an input list of size r, as the map is done over the different residue classes [39].

In our setting, r corresponds to the total number of PEs, i.e., $r = 8$ and 8 parallel processes will thus be created. Note that this is not the optimal way



Rel. error for $n = 120$

Method	Rel. err., %
spline	-14.71
loess	-1.90
lm(poly)	1.28
lm(poly) w. $n \leq 90$	2.13

Rel. error for $n = 120$

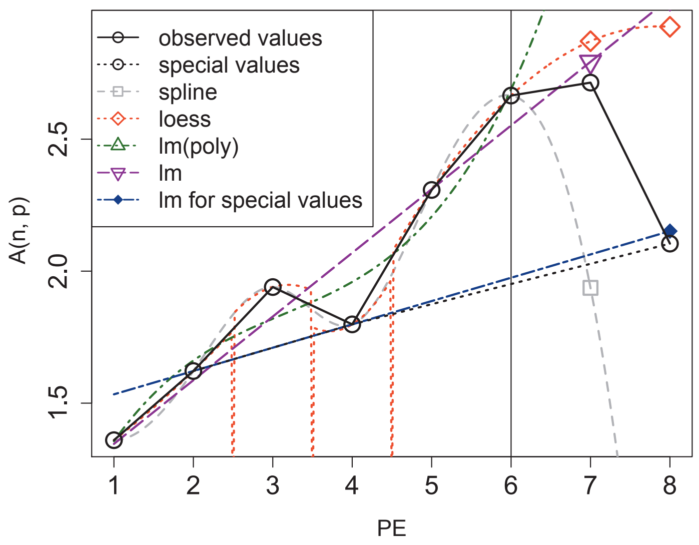
Method	Rel. err., %
loess	10.52
lm(poly)	9.88
lm(poly) w. $n \leq 90$	-12.14
mean	-1.13

Figure 2: Gauß elimination. Predicting sequential run time (top left) and parallel penalty values w.r.t. n (top right) for $n = 120, 150$. The bottom parts show the corresponding relative errors.

to parallelise this program for $p < 8$. When we have more processes than PEs, multiple processes will be executed by the same PE. This causes an imbalance when the processes cannot be evenly distributed to PEs, i.e., when 8 is not a multiple of the number of PEs. Thus, the 2, 4 and 8 PEs configurations perform best. This knowledge could be acquired using source code inspection or process activity profiles, e.g., using Eden TV [6]. In Section 5 we will see, how to obtain the same information with our approach.

Estimating the execution time w.r.t. n. Figure 1 shows the measured times. The values are stated in seconds, rounded up to two digits after decimal dot. We have measured the execution times on eight PEs and estimate the parallel time also on eight PE, but for a larger input size. The data for $n = 120$ and 150 are not known to the estimation routines. Figure 2, left, shows the estimation of the sequential run time $T(n)$. The same figure, right, shows the estimation of $\bar{A}(n,p)$ w.r.t. n. With mean we denote the mean of lm(poly) and of lm(poly), restricted to $n \leq 90$. The reason for this decision is a small increase of $\bar{A}(n,p)$ at $n = 100$, which misleads multiple methods.

Now, as can be seen in the figures, we have the best method for estimating $\bar{A}$—mean—and the best method for estimating $T(n)$—lm(poly). Note, that we can disregard spline in both cases for its very poor performance. Combined, we



Relative error

Method PEs, p	spline 7	loess 7	lm(poly) 7	lm w. all 7	lm w. special 8
Rel. err., %	−28.64	5.78	30.24	2.90	2.25

Figure 3: Gauß elimination. Estimation of penalty values w.r.t. p. We fix $n = 100$ and predict the values for $p = 7, 8$ using the values for $p \leq 6$.

can apply Equation 1 and obtain the complete time estimation. We obtain an estimate $T(120, 8) = 5.736$ seconds, which corresponds to the appropriate value 5.743 up to the relative error −0.125%.

Estimating the execution time w.r.t. p. In the previous example we have assumed, there was the possibility to measure time on an 8 PE machine, but no one had run the test program for the input length 120 or 150. Now we do the converse: assume we have measurements for task size 100 on smaller PE numbers, but do not have a machine with 7 PEs to measure time there. We choose 7, not 8 PEs because of the task distribution issues in our program. We have 8 tasks, which are distributed evenly to PEs. The implicit assumption is the equal "cost" of single tasks. For $p = 8$ this special case is not connected with 6 and 7 PEs configurations. We *can* use only the special cases, but then we would not have enough data points for most of our methods, as we perform our measurements on an 8 PE machine. See "lm w. special" estimation in Figure 3 for this approach.

All other approaches target $p = 7$, as Figure 3 shows. To enable a better insight, we have separated the available values ($p \leq 6$) from values-to-estimate ($p = 7, 8$) with a straight vertical line. Acceptable results were produced by loess. The best method was lm. We used it with all the values from 1 to 6 as "lm w. all". So we use the data from the estimation of $\bar{A}(n, p)$ w.r.t. p and the already

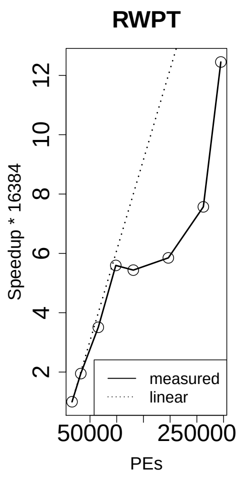
RWPT

PE, p	16384	32768	65536	98304	131072	196608	262144	$\boxed{\mathit{294912}}$
$T(n,p)$	1.93	0.99	0.55	0.345	0.355	0.33	0.255	$\boxed{\mathit{0.155}}$
"Slack off", %	0	0	50	0	75	50	87.5	$\boxed{\mathit{0}}$

LBM

PEs, p	32768	65536	98304	131072	196608	$\boxed{\mathit{262144}}$	294912
$T(n,p)$	16.285	9.99	6.82	6.80	5.284	$\boxed{\mathit{5.273}}$	3.675

Figure 4: RWPT and LBM on a supercomputer, courtesy of Siarhei Khirevich [31]. The task size n is fixed. Top: RWPT execution times. Bottom: LBM execution times. The $\boxed{\mathit{boxed}}$ values will be estimated.



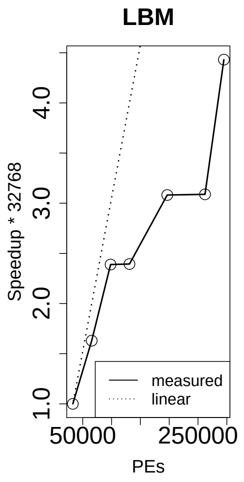


Figure 5: The speedup plots for both RWPT and LBM. Note the bad run time performance of RWPT with non-perfect task distribution.

measured $T(100)$ to predict the parallel execution time. We obtain an estimate $\hat{T}(100,7) = 4.369$ seconds, which is 1.838% accurate. The measured value is 4.290. Now, using lm to estimate $\bar{A}(100,8)$, based only on data for $p = 2,4$ ("lm w. special") we obtain $\hat{T}(100,8) = 3.531$. This result shows 1.361% relative error, compared to the measured value. We see, it is possible to estimate the parallel run time both w.r.t. task size and w.r.t. PE count with significant accuracy.

4.2. Mass Transport in Porous Media

As a further example we consider the lattice-Boltzmann method from fluid flow and mass transport simulation. The data originate from the experiments were performed by Siarhei Khirevich and Anton Daneyko [32, 31] on the Jülich

Blue Gene/P supercomputer. They used all available PE nodes, resulting in an experiment on up to 294912 cores.

Overview. The aim of Khirevich et al. [32, 33, 34] was to simulate the transport processes in porous media, e.g., in the chromatographic separations. Pumped into a long thin pipe, filled with some matter, what paths does an injected solution follow? The matter is modelled with spheres. The pumped solution is simulated in two steps: first fluid flow is simulated, subsequently the actual movement of the matter is studied. The simulation consists of several phases:

1. Random close-sphere packing and its spatial discretisation. We do not consider this phase.
2. Simulation of the fluid flow with lattice-Boltzmann method (LBM). This is the phase we focus on.
3. Simulation of the advective-diffusive mass transport. It is performed with the random-walk particle tracking method (RWPT).

The latter two phases clearly dominate the computational complexity of the method. We chose to focus on the LBM phase for the reasons discussed below. Due to the dimensions of the chosen lattice—$632 \times 632 \times 294912$—a one-dimensional decomposition is possible. Basically, the pipe is cut in length and each "slice" is assigned to a PE. Hence, we have a "`map`-like" parallel implementation. The amount of spheres in each slice varies, the maximal difference is 27%.

The LBM phase generates the data used later in the RWPT phase. Basically, the flow of the fluid around the sphere packing is computed, it is done with sophisticated variants of cellular automata [10]. We show the speedup curves for both LBM and RWPT in Figure 5.

Discussion of RWPT. The RWPT method consumes data, generated in the LBM phase. It *traces* the paths of single molecules through the pipe. The simulation, regarded here, uses 40 million "tracings". Each tracing adds the closest neighbourhood vector of the fluid flow, obtained with LBM, and some random diffusion vector.

The separate slices—tasks!—are distributed to PEs. Thus in their paper [32], Khirevich and Daneyko observe lower speedups in cases when *some* PEs have more tasks than others. As the computation time for a single task is proportional to the amount of spheres, the developers decided against dynamic task balancing. However, bad static task balancing leads to problems with speedups, as we will explain in the following. We show the execution times for thousand iterations of RWPT, starting with 16384 PEs and up to 294912 PEs in the top part of Figure 4. As the program data is too large to fit in memory in the sequential case, we take the time on 16384 PEs to be the "sequential" point of reference for our computations.

As we see in Figure 5, RWPT has major speedup problems in the middle part of the scale, i.e., when using between 100000 and 250000 PEs. The reason is task distribution imbalance, *cf.* the "slack off" values given in Figure 4. Let p be the number of PEs. As in total $n = 294912$ tasks are to be computed, $n \bmod p$

remaining tasks are computed by some PEs, while some other PEs are idling. If many PEs are idle, the imbalance is severe. The “slack off” is the percentage of idle PEs in the final round, when not all PEs can be saturated. We do not consider RWPT further, because we have no suitable prediction approach. LBM has a similar problem, but it is not as severe as in the RWPT case.

The application of the prediction methods presented here to known execution times of RWPT yields too large relative errors. Such behaviour signals the failure of our prediction method, see Section 3. The user would notice the misbehaviour of the prediction approach, she will be able to address this issue by applying further, more advanced statistical prediction methods or by adding further data points.

LBM: Time measurement. The time measurement data for LBM on Jülich Blue Gene/P presented in the bottom part of Figure 4 have been provided by S. Khirevich [31]. The time required for 10 iterations is given. We assume $T(n)$ in our computations to be $T(n, 32786) \cdot 32768$. In other words: a perfect speedup for up to 32768 PEs is assumed. This is rather a convenience convention than an assumption: we could as well “downscale” the PE numbers by dividing them by 32768. It is impossible to obtain the real sequential time, as the data to be processed does not fit into the memory of a single machine.

Estimation. Given the time measurements, we start our prediction round. The task size is fixed, but we can estimate the scalability w.r.t. the PE number p. Note that we have neither the source code nor a binary version of the program we investigate.

We present the values for $\bar{A}(n, p)$ w.r.t. p in the top part of Figure 6. The lm(poly) method with polynomials of degree 3 results in the estimation 3.16 seconds for $\bar{A}(n, 262144)$. Using it and approximating $T(n)$ with $16.285 \cdot 32768$, we obtain the estimation for the execution time $\hat{T}(n, 262144) = 5.196$ seconds. This estimation is exact up to the relative error -1.47%. We present it graphically in the bottom part of Figure 6. Notably, a direct run time estimation—an attempt to predict the parallel time directly from the number of PEs, using the same data, but without using Equation 1—fails. The all-best relative error for direct estimation is -31%.

This shows that our approach is applicable to large-scale production applications. An accurate forecast has been made with a non-application centred approach: we do not require any knowledge of the application, aside from a set of execution time measurements.

Generalisation. We have seen a good performance of lm(poly) for estimating $\bar{A}(n, p)$ w.r.t. p for LBM and an acceptable performance of two lm(poly) methods for Gauß elimination. Hence, we can conjecture that for parallel `map`-based programs lm(poly) is the most suitable method of the statistical techniques regarded here.

PEs, p	32768	65536	98304	131072	196608	$\boxed{\mathit{262144}}$
$\bar{A}(n,p)$	0	1.85	1.39	2.73	2.57	$\boxed{\mathit{3.24}}$

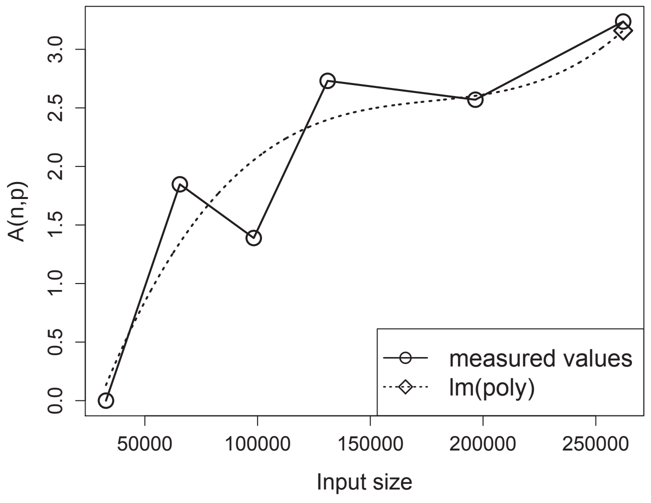


Figure 6: Computing $\bar{A}(n,p)$ for LBM. Top: computed values for $\bar{A}$. The $\boxed{\mathit{boxed}}$ value will be estimated on the bottom.

4.3. Karatsuba Multiplication – Divide and Conquer

The Karatsuba multiplication is a ternary divide and conquer algorithm for multiplying large integers [29]. The following results have been obtained with Eden on an Intel multicore machine. We perform the computation for integers of equal size; the size of an integer is its number of digits. The integer size for the input has been uniformly distributed between 16000 and 64000 digits. We predict the values for 60000 and 64000. See the top of Figure 7 for a snapshot of the data. We use the relative reference point. The usage of relative reference point here and in some further examples aims to show the low dependency of our method from the number of data points. In fact, with the relative reference point, $\bar{A}(n,1)$ is always zero. So, in a sense we have "one point less". Still, our prediction methods work as expected. This underlines their strength and robustness.

We have estimated $T(60000)$ for the Karatsuba multiplication with the spline method producing a relative error of -0.014%. The next best estimation has been achieved with lm(poly) of the third degree with relative error 1.30%. The latter method for $T(64000)$ has the best relative error of 1.90%, whereas the spline method produces a relative error of 2.76%. The loess method is not significantly worse. See Figure 8 for more details. As for $\bar{A}(n,p)$ w.r.t. n, we obtain a relative error of 2.3% for $\bar{A}(60000,8)$ with lm(poly) of degree 3. The most reliable estimation is produced by loess with -5.443% and 2.078% relative errors for the estimations of input lengths 60000 and 64000 respectively. This

Uniform

$n \cdot 1000$	16	20	24	28	32	36	40
$T(n)$	9.88	13.86	20.27	24.78	29.58	34.83	41.42
$T(n, 8)$	1.29	1.78	2.61	3.19	3.74	4.47	5.37

$n \cdot 1000$	44	48	52	56	**60**	**64**
$T(n)$	53.58	60.94	67.55	74.39	**82.02**	**88.94**
$T(n, 8)$	7.14	8.05	8.98	9.95	**11.0**	**11.86**

Non-uniform

$n \cdot 1000$	0.5	1	2	4	8	16	32	64	**128**
$T(n)$	0.11	0.19	0.41	1.10	3.30	9.87	29.54	89.22	**267.25**
$T(n, 8)$	0.0654	0.0818	0.129	0.222	0.47	1.28	3.74	11.86	**36.66**

Figure 7: Karatsuba multiplication on 8 PEs. The **bold** values will be estimated.

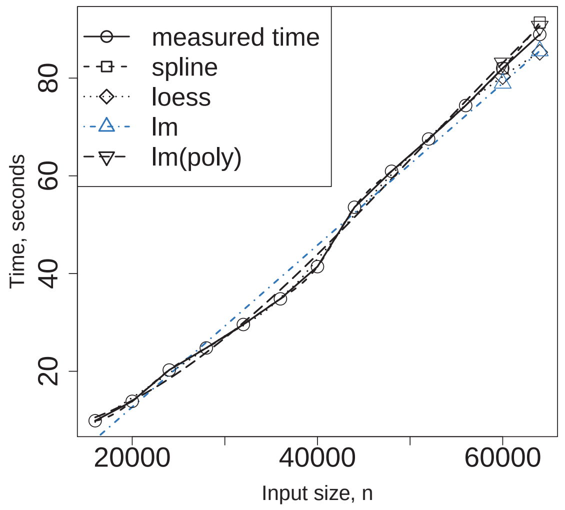
results in parallel run time estimation values 11.01 seconds and 12.09 seconds for the same inputs, corresponding to relative errors 0.14% and 1.78%. Thus an appropriate estimation of run time does also work for a divide and conquer-based computation.

We also experimented with a non-uniform input data distribution, given at the bottom of Figure 7. The input size varied between 500 and 128000 digits. We estimate $T(n)$ and $\bar{A}(n, p)$ w.r.t. n. Under the assumption that the first 8 data points are available, we have predicted the ninth point with $n = 128000$. We predict the value of 36.66 seconds with an acceptable quality using the spline method: 8.984% relative error. But we are exceptionally successful with the lm(poly) approach of degree 3 for both components. We obtain the final estimation result of 36.67 seconds (rounded up to second digit) with prior knowledge of the run times only up to $n \leq 64000$. Hence our formalism is also applicable to such "long-distance" run time estimations.

4.4. Rabin-Miller Primality Test – Iteration

The Rabin-Miller primality test is an iterative application, which performs k iterations to check whether its input value is prime or not [42, 46]. The positive result of the test does not ensure that the input is prime, but yields with a certain probability of this fact. We perform the test on Mersenne primes, the parameter n means that the number $2^n - 1$ is tested for primality. We have chosen n in such a way that $2^n - 1$ is a prime number, in order to guarantee that all k iterations are performed. This leads to a non-uniform distribution of the input values.

We show the time measurement data for an Eden-based implementation on the eight-core Intel machine in the top part of Figure 9. For estimating $\bar{A}(11213, 8)$ we used spline, loess, lm(poly) of degree 3 and mean of latter two. We



Relative error for $T(60000)$

Method	spline	loess	lm	lm(poly)
Rel. error, %	−0.01424	−2.081	−3.7609	1.3032

Figure 8: Karatsuba multiplication. Estimating the sequential run time $T(n)$ for $n = 60000$ and 64000 on uniform data.

use $T(n, 1)$ as an estimate for $T(n)$: in other words, we calculate with relative and not with absolute reference point. The results are presented in the bottom part of Figure 9. An overview for single components is available in Figure 10. Our best method is using lm(poly) and mean for estimating $T(n)$ and $\bar{A}(n, p)$ respectively, resulting in 0.01% relative error for the final result. Thus the prediction of run time of parallel Rabin-Miller test has been very accurate.

Scaling up. To show again that our approach scales well, we have performed estimations of the same application on a larger multicore machine, available at the RETIS laboratory of Scuola Superiore Sant'Anna. It is a quad-CPU AMD Opteron 6168 design with 12 cores each, i.e., 48 CPU cores in total. The CPUs are clocked at 1.9 GHz, we have 64 GB of shared memory available. We were using 64 bit Debian Linux OS with Linux kernel 3.4.

We issued 96 tasks and performed estimations w.r.t. p. We perform two kinds of an estimation. Firstly, we aim to obtain $\hat{T}(11213, 48)$ from $T(11213)$ and an estimated $\bar{A}(11213, 48)$. This estimation used *special* values of p: sufficiently large numbers of processors for which the tasks can be distributed evenly. This is justified as for $p = 48$ tasks are also distributed evenly, in contrast to $32 < p < 48$. The special values of p are distributed in a non-uniform manner. The data is available in Figure 11(a). The both methods lm and lm(poly) of degree 2 alternated the sign of the relative error, we introduced their mean. This was the best method, *Time*

n	2203	2281	3217	4253	4423	9689	**11213**
$T(n,1)$	1.882	2.094	5.284	10.77	12.16	96.95	**144.82**
$T(n,7)$	0.304	0.332	0.814	1.639	1.849	14.63	21.80
$T(n,8)$	0.304	0.334	0.812	1.635	1.843	14.66	**21.78**

Estimation

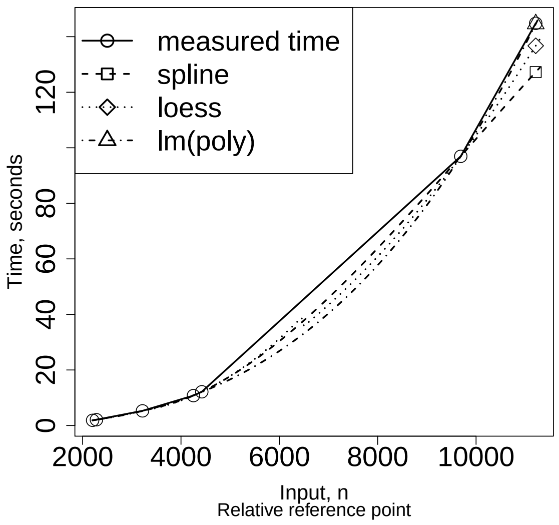
Methods	Estimate for $T(n)$	$\bar{A}(n,8)$	$T(n,8)$	Relative error, %
spline + spline	127.25	3.32	19.23	−11.70
spline + loess	127.25	3.59	19.50	−10.56
spline + lm(poly)	127.25	3.82	19.73	−9.42
spline + lm(poly)	127.25	3.71	19.61	−9.94
loess + spline	136.73	3.32	20.41	−6.26
loess + loess	136.73	3.59	20.69	−5.02
loess + lm(poly)	136.73	3.82	20.91	−3.98
loess + mean	136.73	3.71	20.80	−4.50
lm(poly) + spline	144.59	3.32	21.40	−1.75
lm(poly) + loess	144.59	3.59	21.67	−0.50
lm(poly) + lm(poly)	144.59	3.82	21.89	0.53
lm(poly) + mean	144.59	3.71	21.78	0.01

Figure 9: Rabin-Miller test. On the top: measured times. The **bold** value is estimated at the bottom with all available methods.

yielding less than 1% relative error. We present the complete estimation for $T(n,p)$ in Figure 12, top. The bottom part of the same figure shows the relative errors for the $\bar{A}(n,p)$ estimation. The best relative error for the estimation of $T(n,p)$ was merely 0.1745%, we rate this as a very good result.

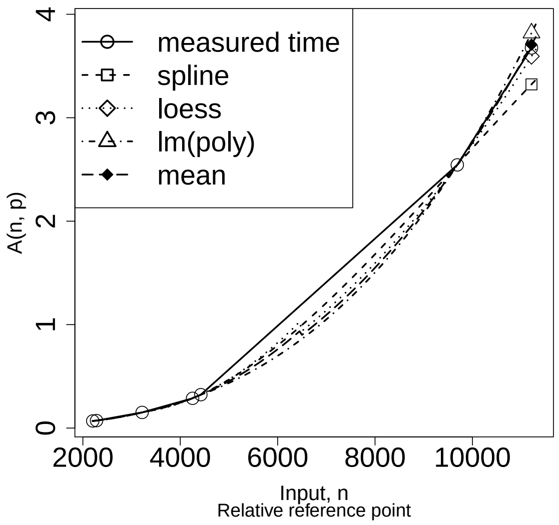
The second estimation experiment was to estimate the parallel penalty, and from it: the parallel execution time for $p = 47$, basing on all the data for $p < 47$. Here, values of p are uniformly distributed. We did not choose $p = 48$ for this estimation because of its special configuration; with 96 tasks there is no task imbalance for $p = 48$, but there is one for $p = 47$ and most other configurations. We used $n = 19937$, i.e., the input Mersenne prime was $2^{19937} - 1$. This is a prime number, the application performs all 96 iterations. The measurement data is in Figure 11(b). We managed to obtain an estimation for $\bar{A}(19937, 47)$ that was exact up to a relative error of −0.91% with loess. This results in the estimation of the parallel execution time that varies only in −0.315% from the measured value. The details and relative errors are reported in Figure 13.

In these two examples we have seen that our method can scale to larger number of PEs on multicore hardware. See also Section 4.2 for an estimation of a program, run on a peta-scale supercomputer with hundreds thousands of cores.



Relative error for $T(n)$

Method	spline	loess	lm(poly)
Rel. error, %	-12.13	-5.59	-0.1566



Relative error for $\bar{A}(n,p)$

Method	spline	loess	lm(poly)	mean
Rel. error, %	-9.58	-2.21	3.91	0.851

Figure 10: Predicting values for both components for Rabin-Miller test. Top $T(n)$, bottom $\bar{A}(n,p)$.

Rabin-Miller test, non-uniform p

p	1	12	16	24	32	$\boxed{\mathit{48}}$
$T(n,p)$	128.90	11.30	8.83	5.87	4.66	$\boxed{\mathit{3.29}}$

(a) First example: non-uniform distribution of p, $n = 11213$

Rabin-Miller test, uniform p

p	1	2	3	4	5	6	7	8	9	10	11	12
$T(n,p)$	560.74	282.29	189.37	142.97	120.06	98.14	87.07	77.28	74.67	60.98	54.99	50.90
p	13	14	15	16	17	18	19	20	21	22	23	24
$T(n,p)$	49.15	43.26	44.93	39.80	38.00	37.04	37.00	31.56	31.18	31.25	31.24	25.71
p	25	26	27	28	29	30	31	32	33	34	35	36
$T(n,p)$	25.19	25.21	25.85	25.19	25.51	25.13	25.08	22.28	19.24	19.25	19.41	19.26
p	37	38	39	40	41	42	43	44	45	46	$\boxed{\mathit{47}}$	48
$T(n,p)$	19.98	19.26	19.38	19.18	19.16	19.25	19.15	19.16	19.18	19.23	$\boxed{\mathit{19.22}}$	13.53

(b) Second example: uniform distribution of p, $n = 19937$

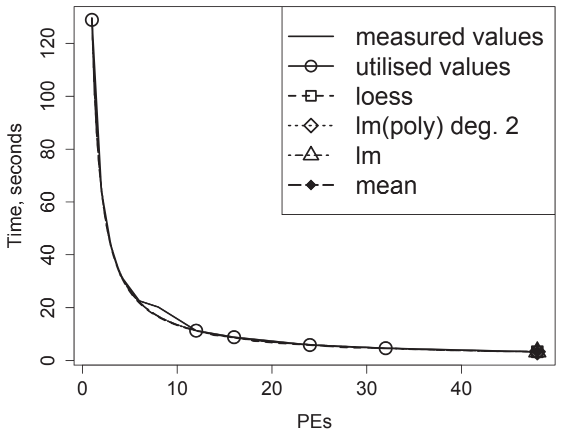
Figure 11: The measured execution time for a larger-scale Rabin-Miller test. The $\boxed{\textit{boxed}}$ values will be estimated.

4.5. APRCL Primality Test – Iteration

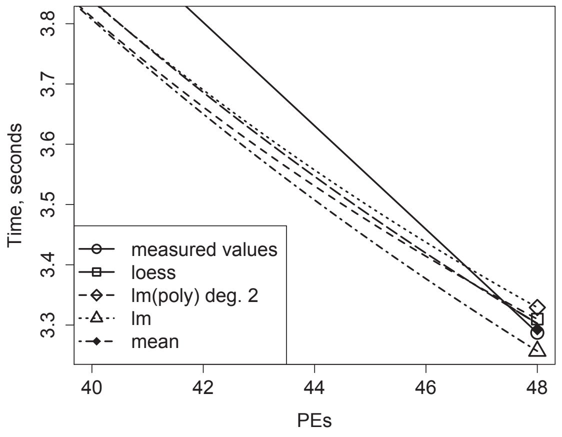
In this section we discuss the estimation of execution time of the APRCL primality test [1, 13]. It is a sophisticated primality test, contrary to the Rabin-Miller test, APRCL *proves* primality. Here, the work needed to process a single task varies significantly. The test is implemented in Eden, we used a dynamically load-balanced workpool skeleton with some advanced tuning of task order to implement this primality test [36, 37]. The major problem was in scattered[3] run times of single tasks in the workpool. The run time experiments were performed on an Intel multicore machine. We used primes of size $\approx 2^{600}$–2^{619} as the input. We designate the exponents as the input sizes, i.e., they range from 600 to 619. The aim is to estimate the execution time for the input size 619 from smaller inputs. We use relative speedup and, correspondingly, relative reference point.

We were able to predict the execution time w.r.t. input size quite accurately. We show the initial data in Figure 14. Note, that the increasing input size does not always result in increased run time. The plots of our prediction approaches are in Figure 15. We see that we were able to predict the sequential time very well. Indeed, the relative error for the lm(poly) method with orthogonal polynomials of degree 4 was merely $4 \cdot 10^{-5}\%$. The spline and loess methods were also quite good, with 0.40% and −0.61% relative error appropriately. On the other hand, the

[3]To quantify: for the data set of run time measurements of single tasks for the APRCL, being standardised to the variance of 1 and mean 0, the smallest value was −1.27, the largest peaked at 4.31.



(a) Execution time estimation: full picture

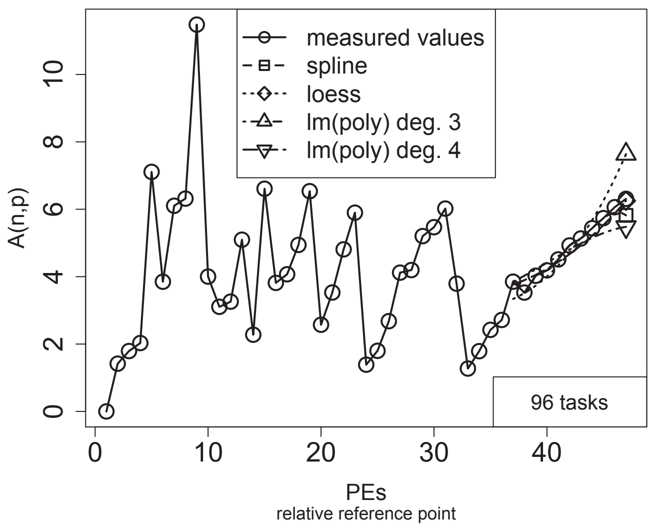


(b) Execution time estimation: zoomed

Method	loess	lm(poly) deg. 2	lm	mean
Rel. err. %	3.77	6.96	−5.06	0.953

(c) Relative errors for $\bar{A}(n, p)$

Figure 12: Rabin-Miller test: large-scale estimation for special values. Top: combined execution time estimations, bottom: relative errors for the estimations of the relative penalty.



(a) Estimation of $\bar{A}(n,p)$ w.r.t. p

Method	spline	loess	lm(poly) deg. 3	lm(poly) deg. 4
Rel. err. %	-7.77	-0.9096	21.04	-13.17

(b) Relative errors for the parallel overhead w.r.t. p

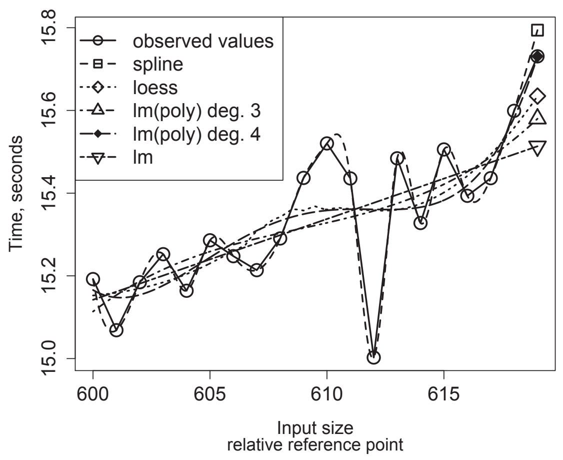
Figure 13: Rabin-Miller test: large-scale estimation for all values. We estimate $\bar{A}(n,p)$ w.r.t. p for $n = 19937$ and $p = 47$. Top: estimation process, bottom: relative errors for it.

n	600	601	602	603	604	605	606	607	608	609
$T(n,1)$	15.19	15.07	15.18	15.25	15.16	15.27	15.25	15.21	15.29	15.44
$T(n,8)$	2.59	2.58	2.66	2.62	2.68	2.60	2.55	2.59	2.58	2.62
n	610	611	612	613	614	615	616	617	618	**619**
$T(n,1)$	15.52	15.44	15.00	15.48	15.33	15.51	15.39	15.44	15.60	**15.73**
$T(n,8)$	2.76	2.66	2.53	2.71	2.59	2.65	2.59	2.69	2.64	**2.78**

Figure 14: Timings for APRCL test. **Bold** value will be estimated.

estimation of the parallel overhead w.r.t. n was disappointing, as we obtained a relative error of -9.08% with lm(poly) of degree 3. To do so, we disregarded the value at $n = 618$. Still, with both predicted values for $T(n,1)$ and $\bar{A}(n,p)$ combined, we obtain the time for $T(619,8)$ up to -2.66% relative error.

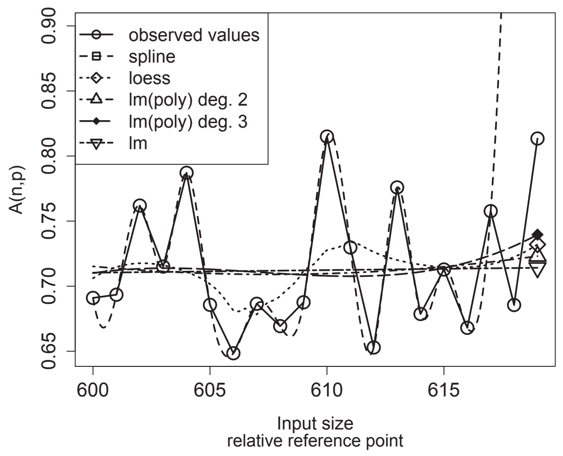
Even more interesting is the prediction of $\bar{A}(n,p)$ w.r.t. p. We assume the values on $p \leq 7$ as given. The times for $n = 619$ are used. The estimation is not easy, but still we obtain the value 0.79 with lm(poly) of degree 2. This corresponds with the measured value for $\bar{A}(619,8)$ up to -2.51% relative error. Using it for the estimation of $T(619,8)$ in a combination with a given $T(619,1)$, we obtain



(a) Sequential time

Method	spline	loess	lm(poly) of degree 2	3	4	lm
Rel. err., %	0.40	-0.61	-1.35	-0.959	$4.3298 \cdot 10^{-5}$	-1.383

(b) Relative error for seq. time

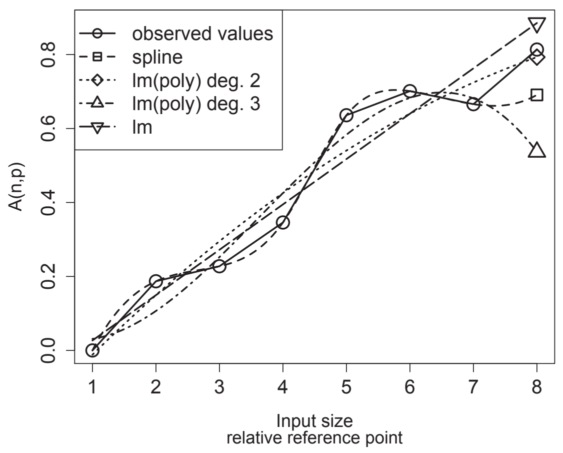


(c) Parallel overhead

Method	spline	loess	lm(poly) of degree 2	3	lm
Rel. err., %	183	-10.01	-11.11	-9.081	-12.21

(d) Relative error for parallel overhead w.r.t. p

Figure 15: Predicting the both components for APRCL. Top: sequential time, bottom: parallel overhead w.r.t. n.



(a) Estimation

Method	spline	lm(poly) deg. 2	lm(poly) deg. 3	lm
Rel. err., %	−15.07	−2.508	−34.04	8.82

(b) Relative error

Figure 16: APRCL test. Estimating $\bar{A}(n, p)$ w.r.t. p.

PEs, p	1	2	4	8	16
$T(n, p)$	3899	1947	1003	538	$\boxed{\mathit{333}}$

Figure 17: The scaling data for the input "random 20" from Gondzio and Sarkissian [20]. Used with a kind permission. The $\boxed{\textit{boxed}}$ value will be estimated.

the execution time up to −0.73% exact.

We have seen that our method can also be applied to a sophisticated computation featuring dynamic load-balancing and strongly varying execution times for single tasks.

4.6. Linear Solver

In this section we regard the data from the paper by Jacek Gondzio and Robert Sarkissian [20]. It presents a parallel solver for the problems in linear programming. We stress that the only data we use are the measurements from the paper, we restate the values in Figure 17. The data was obtained on a network of 18 Sun machines. We cannot state a particular pattern in the program.

We perform an estimation w.r.t. p. As n is fix, we do not need to estimate sequential time; we use $T(n, 1)$ as $T(n)$. We need merely to estimate the value for $\bar{A}(n, 16)$. The measured value is 89.3. With lm(poly) of degree 2 we obtain the value 66.36 and lm results in 115.64. However, the mean of these two values is 91.00, which is exact up to 1.89% relative error. With this value we estimate $T(n, 16)$ as 334.69, which is 0.507% exact.

We have shown again, we can use our method, basing only on execution time measurements of a third party's program.

5. Serial Fraction

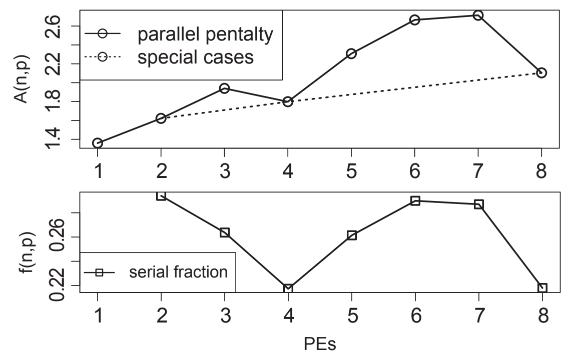
In their work "Measuring Parallel Processor Performance" [30] A. H. Karp and H. P. Flatt introduced the notion of *serial fraction.* Given the parallel time $T(n, p)$ on p PEs and the sequential time $T(n)$, the absolute speedup is $T(n)/T(n, p)$. The serial fraction is

$$f(n,p) = \frac{T(n,p)/T(n) - 1/p}{1 - 1/p}.$$

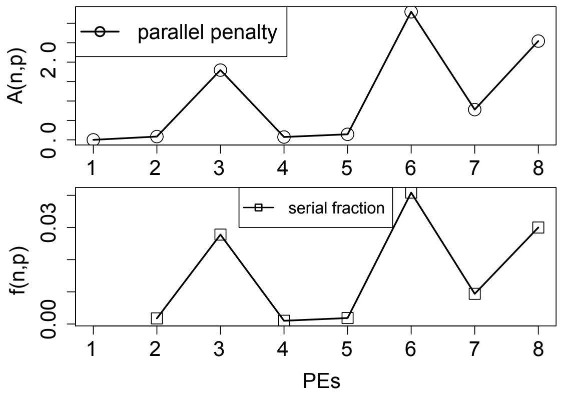
The serial fraction should be constant: if it increases, we have a parallelisation resulting in poor speedups. If the serial fraction decreases, this shows problems with the sequential implementation. We have computed the serial fraction for the test cases presented above. For Gauß elimination, we have used $n = 100$. Both measures—our parallel penalty $\bar{A}$ and the serial fraction f—are shown in Figure 18(a). The shapes of the curves correspond to the load distribution in the computation, as we might deduce from the knowledge of the source code. Noteworthy, both our notion of parallel penalty and the serial fraction have minima at 4 and 8 PEs. The value of the parallel penalty $\bar{A}(n, p)$ is almost constant for 2, 4 and 8 PEs versions. Exactly these versions are optimal in the load distribution. The serial fraction shows the same for 4 and 8 PEs versions, but indicates a larger serial component for 2 PEs.

We want to stress that the curves for both quality measures differ (*viz.* the behaviour at 1–4 and 6–8 PEs in Figure 18(a)). Though bearing a similar *meaning*, parallel penalty and serial fraction differ significantly both in the *shapes* of the plots and in the range of *values*. Thus we cannot think of the parallel penalty being a reformulation of the serial fraction or a product of it with some constant factor.

The parallel penalty and serial fraction for Rabin-Miller primality test with the input 9689 as executed on an Intel multicore machine are shown in Figure 18(b), the numerical values are in Figure 18(c). We can clearly see problems with load balancing for 3 and 6 PEs, as we needed to compute exactly 20 tasks in this experiment. This amount of tasks cannot be fairly distributed to 3 or 6 PEs, some PEs are idling, thus increasing the parallel overhead. Consider 6 PEs. There will be 6—6—6—2 tasks during the four parallel iterations, thus 4 PEs will be idle in the last iteration. Only one PE in the last iteration is idle for a 7 PEs configuration, however the overhead increases again for 8 PEs, resulting in a 8—8—4 scheme. This corresponds with the conclusions, drawn from the shape of the parallel penalty. We deduce also from Figure 18(b), that for executing exactly 20 tasks in parallel, 4 or 5 PEs in a system are ideal. Simple thoughts on task distribution result in the same observation.



(a) Gauß elimination, parallel penalty and serial fraction



(b) Rabin-Miller test, parallel penalty and serial fraction

Rabin-Miller test

p	1	2	3	4	5	6	7	8
$\bar{A}(9689, p)$	0	0.0844	1.796	0.0731	0.1433	3.30	0.7793	2.5442
$f(9689, p) \times 100$	–	0.1741	2.779	0.1005	0.1847	4.084	0.9379	2.9992

(c) Comparing the numerical values for Rabin-Miller test

Figure 18: Comparing the serial fraction with our approach.

6. Related Work

Our approach bears—as everything on this topic—a certain grade of similarity to Amdahl's law [3]. Exactly as Amdahl did, we assume a perfect parallelisation of the computation, but consider also the unavoidable overhead. However, we divide the parallel computation not into Amdahl's perfectly parallel and strictly sequential fractions, but into fractions of effective computation and of parallel overhead.

Related publications on performance forecasting include the book chapter on skeletons in Eden [40] and the formal cost model of NESL [8]. However, our approach is different. We derive the time and work from time *measurements* for the runs on different numbers of PEs, while the skeleton analysis by Loogen et al. [40] is based on latency and message-passing costs. The NESL complexity model [8] takes a “bottom-up” approach, trying to assign cost to single semantic operations. We look in a “top-down” manner on the total run time and divide it into very coarse blocks.

Further approaches on skeleton-based performance evaluation include [15, 5]. Cole and Hayashi [15] regard a BSP-like cost model, assigning costs to basic elements of a parallel program. A fine-grain cost model for some BSP-based skeletons is in the paper by Zavanella [52]. Besides BSP [50], models like LogP [16] and message passing models exist. Roda et al. [47] presented a run time prediction approach for the latter. A rather theoretical well-investigated performance model is the PRAM model [18] and its variants. An alternative to skeletons [14] program classification approach are Berkeley dwarfs [4], however in this case several particular types of tasks are identified, in contrast to abstracting from the task type with skeletons.

An example of the sequential estimation of run time is the paper by Saavedra and Smith [48]. Ipek et al. [27] used neuronal networks to (directly) predict execution times of a multigrid solver. The paper by Akioka and Muraoka [2] focuses on the network load and uses Markov model-based meta-predictor. In a contrast, we used our decomposition of the parallel execution time in Equation (1) and statistical methods. Kapadia et al. [28] found locally weighted polynomial regression definitely superior than other instance-based learning methods (e.g., nearest neighbour) for the estimation of parallel execution time of real programs. However, Kapadia et al. [28] did not use any decomposition of the execution time. Further, the number executions for the machine learning approaches is quite high: 8100 in [28], 10000 in [27]. Our approach provided good results on orders of magnitude less data points: all Eden case studies (up to the scaling study of our method in Section 4.4) feature 10–20 data points, each of them constituted of an average of 5 program runs. The C+MPI case studies have used even less data points and still provided successful estimations. Thus, our method can more easily be used in an adaptive runtime environment. However, an increased number of data points does not harm our method, as the second half of Section 4.4 shows.

Regarding approaches for a parallel quality measure, the notion of serial fraction as defined by Karp and Flatt [30] is related to ours. We discussed it in Section 5. Isoefficiency [21], scaled speedup [24] and other approaches are less similar to parallel penalty $\bar{A}(n,p)$, as we do not aim for a larger input on a larger PE count, which keeps the efficiency the same. Kumar and Gupta [35] present an overview. Speedup bounds have been discussed by Polychronopoulos and Banerjee [44].

A further research direction, orthogonal to our approach is hotspot and bottleneck analysis. We refer to modern visualisation tools, like Scalasca [19]. In parallel functional programming, Eden TV [6] and ThreadScope [51] are used. We were able to confirm our assumptions on bad process placement of programs from Sections 4.1 and 4.4 with Eden TV in the first author’s PhD thesis [36].

7. Future Work

We would like to perform detailed analyses of further programs. We need a way to handle severe imbalance in a parallel map. This has hindered a detailed analysis of RWPT method in Section 4.2. The same problem existed also in other case studies we have considered, however not to such an extent. Some kind of preconditioning the data to the regular case could help. Moreover, we plan to seek for more advanced prediction methods for sequences. More sophisticated statistical methods can be utilised. A proper direction would be, for instance, the further separation of special performance cases. We addressed this in Sections 4.1 and 4.4: we treated there the "balanced" cases separately from "unbalanced" ones—with a success. Statistically sound statements on the *distance* of robust extrapolation using the structure of the presented model are also left for future work. We look forward to more experiments with large-scale systems, further languages and middleware. We would like to apply our methods to parallel GPU computing, specially CUDA [43] in future. A high-level approach for GPU programming is facilitated by the Thrust library [26]. Another interesting topic is an estimation of the number of measured values, which we need for a good estimation of run time. In other words: the estimation of the "cost" of the prediction. A machine learning approach for predicting separate sequences and an adaptive runtime environment for parallel programs present rather long-term ideas for future work.

8. Conclusions

We introduced a method to predict the execution times of parallel programs in a novel and elegant manner. Our method is different from previously known approaches. It does not depend on source code analysis or on special semantic rules, instead, our method is empowered by computational statistics and abstraction. In this paper we treated the hardware and the parallel program as a black box, focusing on the estimations instead. Our methodology is intentionally oblivious of the hardware and software; instead, statistical prediction methods implicitly express the traits of hardware and application, particularities of a given range of input sizes and range of processing elements. We introduced a new measure for the quality of parallel programs—the parallel overhead. This measure denotes the amount of "slack time" pro processor, i.e., the time spent not contributing to the actual work. We were the first to perform separate estimations for the (sequential) execution time and parallel overhead, hence different forecasting models were used for each of them. This makes sense, given the different nature of these processes, and enables more accurate estimations. We have used our technique to predict parallel execution times for six non-trivial scientific computing methods. We obtained very low relative errors with few data points. We tested our technique not only with Eden and various multicore SMP hardware, but also on a peta-scale supercomputer and networked Sun workstations, using C+MPI.

The contributions of this paper include

- Novel division of the parallel execution time into components of a different nature that are predicted separately.
- Highly precise estimations of parallel execution time for the real-world programs: the best relative error was 0.01%, a single component yielded $4 \cdot 10^{-5}\%$.
- Our approach requires orders of magnitude less data points than other known methods.
- This paper introduced the parallel overhead term—a quality measure for parallel programs.

Acknowledgements. We would like to thank Thomas Horstmeyer for fruitful discussions on divide and conquer evaluations. Big thanks go to Siarhei Khirevich for sharing his Blue Gene/P time measurements with us. We thank anonymous reviewers for helping us to improve our presentation.

The first author was supported by Deutsche Forschungsgemeinschaft (DFG) grant LO 630-3/1, Hessian Ministry for Science and the Arts' (HMWK) LOEWE KMU programme, and the S(o)OS project, sponsored by the European Commission under FP7-ICT-2009.8.1, Grant Agreement No. 248465. An earlier version of this paper has been presented at ACM HLPP'10 [38].

Literature

[1] L. M. Adleman, C. Pomerance, and R. S. Rumely. On distinguishing prime numbers from composite numbers. *Annals Of Mathematics*, 117(1):173–206, 1983.

[2] S. Akioka and Y. Muraoka. Extended forecast of CPU and network load on computational grid. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 765–772. IEEE Computer Society, 2004. ISBN 0-7803-8430-X. DOI http://doi.ieeecomputersociety.org/10.1109/CCGrid.2004.1336711.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS 1967 Spring Joint Computer Conference*, pages 483–485. ACM Press, 1967.

[4] Krste Asanovic, Ras Bodik, et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[5] Anne Benoit, Murray I. Cole, Stephen Gilmore, and Jane Hillston. Evaluating the performance of skeleton-based high level parallel programs. In *The International Conference on Computational Science (ICCS 2004), Part III*, LNCS 3038, pages 299–306. Springer, 2004.

[6] Jost Berthold and Rita Loogen. Visualizing Parallel Functional Program Executions: Case Studies with the Eden Trace Viewer. In *Proceedings of the Intl. Conf. ParCo 2007 – Parallel Computing: Architectures, Algorithms and Applications*. IOS Press, 2007.

[7] G. E. Blelloch and J. Greiner. A Parallel Complexity Model for Functional Languages. Carnegie Mellon University Pittsburgh, PA, USA, 1994.

[8] G.E. Blelloch. Programming Parallel Algorithms. *Comm. ACM*, 39(3): 85–97, 1996.

[9] J. M. Chambers and T. J. Hastie. *Statistical models in S*. CRC Press, 1991.

[10] S. Chen and G. D. Doolen. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(1):329–364, 1998.

[11] William S. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368):829–836, 1979. ISSN 01621459. URL http://www.jstor.org/stable/2286407.

[12] William S. Cleveland and Susan J. Devlin. Locally weighted regression: An approach to regression analysis by local fitting. *Journal of the American Statistical Association*, 83(403):596–610, 1988. ISSN 01621459. URL http://www.jstor.org/stable/2289282.

[13] H. Cohen and H. W. Lenstra, Jr. Primality testing and Jacobi sums. *Mathematics Of Computation*, 42(165):297–330, 1984.

[14] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[15] M. I. Cole and Y. Hayashi. Static performance prediction of skeletal programs. *Parallel Algorithms and Applications*, 17(1):59–84, 2002.

[16] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. LogP: Towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7):12, 1993.

[17] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer methods for mathematical computations*. Prentice Hall Professional Technical Reference, 1977.

[18] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th annual ACM Symposium on Theory of Computing*, pages 114–118. ACM, 1978.

[19] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6), 2010.

[20] Jacek Gondzio and Robert Sarkissian. Parallel interior-point solver for structured linear programs. *Mathematical Programming*, 96:561–584, 2003. DOI 10.1007/s10107-003-0379-5.

[21] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Conc.*, 1(3):12–21, 1993. ISSN 1063-6552.

[22] A. Y. Grama, V. Kumar, A. Gupta, and G. Karypis. *Introduction to parallel computing*. Addison Wesley, 2003.

[23] Eden Group. Eden skeleton library. Hackage, 2012. http://hackage.haskell.org/package/edenskel. Retrieved 27.6.2012.

[24] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Stat. Comput.*, 9(4): 609–638, 1988.

[25] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 2008.

[26] J. Hoberock and N. Bell. Thrust: C++ template library for CUDA, 2009.

[27] Engin Ipek, Bronis de Supinski, Martin Schulz, and Sally McKee. An approach to performance prediction for parallel applications. In José Cunha and Pedro Medeiros, editors, *Euro-Par 2005 Parallel Processing*, LNCS 3648, pages 196–205. Springer, 2005. DOI 10.1007/11549468_24.

[28] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. In *Proceedings of the 8th International Symposium on High Performance Distributed Computing*, pages 47–54. IEEE, 1999.

[29] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962. Translation in Physics–Doklady 7, 595–596, 1963.

[30] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Comm. ACM*, 33(5):539–543, 1990. ISSN 0001-0782. DOI 10.1145/78607.78614.

[31] Siarhei Khirevich. Private communication, May 2010.

[32] Siarhei Khirevich and Anton Daneyko. Simulation of fluid flow and mass transport at extreme scale. In Bernd Mohr and Wolfgang Frings, editors, *Jülich Blue Gene/P Extreme Scaling Workshop 2010*. Jülich Supercomputing Centre, March 2010.

[33] Siarhei Khirevich, Alexandra Höltzel, Steffen Ehlert, Andreas Seidel-Morgenstern, and Ulrich Tallarek. Large-scale simulation of flow and transport in reconstructed hplc-microchip packings. *Analytical Chemistry*, 81 (12):4937–4945, 2009.

[34] Siarhei Khirevich, Alexandra Höltzel, Andreas Seidel-Morgenstern, and Ulrich Tallarek. Time and length scales of eddy dispersion in chromatographic beds. *Analytical Chemistry*, 81(16):7057–7066, 2009.

[35] V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *J. Parallel and Distributed Computing*, 22(3):379–391, 1994.

[36] Oleg Lobachev. *Implementation and Evaluation of Algorithmic Skeletons: Parallelisation of Computer Algebra Algorithms.* PhD thesis, Philipps-Universität Marburg, 2011.

[37] Oleg Lobachev. Parallel computation skeletons with premature termination property. In Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming*, LNCS 7294, pages 197–212. Springer-Verlag, 2012. ISBN 978-3-642-29821-9. DOI 10.1007/978-3-642-29822-6_17.

[38] Oleg Lobachev and Rita Loogen. Estimating parallel performance, a skeleton-based approach. In *Proceedings of 4^{th} International Workshop on High-level Parallel Programming and Applications*, pages 25–34. ACM Press, 2010.

[39] Oleg Lobachev and Rita Loogen. Implementing data parallel rational multiple-residue arithmetic in Eden. In *Computer Algebra in Scientific Computing*, LNCS 6244, pages 178–193. Springer, 2010.

[40] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing.* Springer, 2003.

[41] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15 (3):431–475, 2005.

[42] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal Of Computer And System Sciences*, 13(3):300–317, 1976.

[43] NVIDIA. Compute unified device architecture programming guide, 2007.

[44] C. D. Polychronopoulos and U. Banerjee. Speedup bounds and processor allocation for parallel programs on multiprocessors. In *Proc. of Int. Conf. on Parallel Processing*, pages 961–968, 1986.

[45] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2009. URL http://www.R-project.org.

[46] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal Of Number Theory*, 12(1):128–138, 1980.

[47] J. L. Roda, C. Rodríguez, D. G. Morales, and F. Almeida. Predicting the execution time of message passing models. *Concurrency: Practice and Experience*, 11(9):461–477, 1999.

[48] Rafael H. Saavedra and Alan J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14:344–384, 1996. ISSN 0734-2071. DOI http://doi.acm.org/10.1145/235543.235545.

[49] The MPI Forum. *MPI: A Message-Passing Interface Standard — Version 2.2*. High Performance Computing Center Stuttgart, 2009.

[50] L. G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33 (8):111, 1990.

[51] K. B. Wheeler and D. Thain. Visualizing massively multithreaded applications with ThreadScope. *Concurrency and Computation: Practice and Experience*, 22(1):45–67, 2009.

[52] A. Zavanella. Skeletons, BSP and performance portability. *Parallel Processing Letters*, 11(4):393–407, 2001.