Estimating Parallel Performance,
A Skeleton-Based Approach*

Oleg Lobachev and Rita Loogen

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans-Meerwein-Straße, D-35032 Marburg, Germany
{lobachev,loogen}@informatik.uni-marburg.de

Abstract. In this paper we estimate parallel execution times, based on identifying separate “parts” of the work done by parallel programs which are defined using algorithmic skeletons. Our runtime 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 obtain these values for different problem sizes and numbers of processors and estimate them for unknown values in both dimensions. This allows us to predict parallel execution time for unknown inputs and non-available processor numbers.
Another useful application of our formalism is a measure of parallel program quality. We analyse the values for parallel penalty both for growing input size and for increasing the number of processing elements. From the behaviour of these data, conclusions on parallel performance and scalability are drawn.

Keywords: parallel runtime, algorithmic skeletons, forecasting, polynomial regression, Amdahl’s law, scalability measure, serial fraction

1 Introduction

Since Amdahl’s law [1,9] the quest for modelling parallel performance is open. A nice summary of existing approaches is presented in [7,11]. We suggest a model for a coarse subdivision of parallel runtime into “good” and “bad” parts. Contrary to the popular thought of parallel runtime being the sequential one “sped up” to some factor less than the processing elements (PE) count, we envision parallel runtime as the sequential “work” distributed over a number of PEs plus an additional penalty term.

A simple parallelisation of programs can be achieved using algorithmic skeletons [1]. The latter capture common patterns of parallel computation and can straightforwardly be instantiated for specific problem areas. The knowledge about the parallel program evaluation implemented in the skeletons can be used to derive specialised expressions for the parallel runtime, again in terms of sequential work and parallel overhead.

* Supported by DFG grant LO 630-3/1.
The main goal of this paper is an accurate prediction of parallel runtimes for new input values and for non-available numbers of processors. Our approach is to measure the sequential work and to obtain the parallel overhead for a set of sample inputs or sample numbers of processors. Statistical techniques are then used to extrapolate and to estimate the values for further inputs or other numbers of processors. As the parallel execution time is expressed in terms of these values, this enables the forecast of parallel runtime. Moreover, the estimated parallel overhead is a measure for the scalability of a given parallel program, similar to [10].

We show the practicality of our approach for test programs from symbolic computing, implemented in the parallel functional programming language Eden [14]. The test programs are parallelised using standard skeletons from Eden’s skeleton library [13]. Although our Eden system is used for the experiments, our approach is completely language-independent. The technique is applicable to any parallel system.

In Section 2, we present formulae for expressing execution time and work in general and for three different types of algorithmic skeletons. In particular, we consider parallel maps in Subsection 2.1, a divide-&-conquer skeleton in Subsection 2.2 and an iteration skeleton in Subsection 2.3. Section 3 discusses the way of estimating the workload and parallel penalty from a number of execution time measurements. We present three symbolic computing examples and predict their execution times in Section 4. Section 5 compares the serial fraction approach by Karp and Flatt [10] with our approach. Further related work is considered in Section 6 and Section 7 concludes and gives an outlook on future work.

2 Estimation of Runtime

Let $n$ denote the problem size 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 $W(n, p) = pT(n, p)$. In a parallel execution, the sequential work is distributed over $p$ PEs. The distribution causes a total parallel overhead denoted by $A(n, p)$ (cf. [7]) 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

$$T(n, p) = \frac{T(n)}{p} + \bar{A}(n, p). \quad (*)$$

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 runtime $T(n, p)$ using Equation (*). 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 development 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. In the following, we consider three different algorithmic skeletons which cover a huge amount of typical parallel program structures. We will derive special instances of the above general formulae for $T(n,p)$ and $W(n,p)$ for the different skeletons.

2.1 Parallel Map Skeletons

The `parMap` skeleton captures a simple form of data parallelism. Each element of an input list of type `[a]` is in parallel transformed via a parameter function of type `(a -> b)`, in total yielding a list of type `[b]`. Correspondingly, in Haskell and in Eden the `parMap` skeleton has the type `(a -> b) -> [a] -> [b]`. The skeleton instantiates a new worker process for each element of the input list. We assume that the work (and time) needed to process different elements of the list is the same. Otherwise, one should use a dynamically load-balanced `workpool` instead of `parMap`. Our assumption means that $T(n) = nT(1)$. Thus, the amount of work for `parMap` is $W_{parMap}(n,p) = nT(1) + p\bar{A}(n,p)$, while the time is

$$T_{parMap}(n,p) = \frac{n}{p}T(1) + \bar{A}(n,p).$$

If $n \gg p$, process creation overhead can substantially be reduced by creating only as many worker processes as processing elements are available. The input list is statically divided into almost equally sized blocks, one per worker process which process their blocks sequentially. This *statically load-balanced* version of a parallel map is called `farm`. The time needed for a `farm` can be described by

$$T_{farm}(n,p) = T(n/p) + \bar{A}(n,p),$$

with the work being $W_{farm}(n,p) = pT(n/p) + p\bar{A}(n,p)$.

In Section 4.1, we consider a parallelisation of an example program (Gauß elimination) with the `parMap` skeleton.

2.2 Divide-\&-Conquer

![Fig. 1. A binary tree of depth 3 for flat divide-\&-conquer expansion scheme.](image)

Divide-\&-conquer is a typical example for task parallelism. We consider a regular divide-\&-conquer scheme with a fixed branching degree $r$. There are different possibilities to parallelise such a scheme. In the following, we use a *flat expansion scheme*, i.e. the input is split sequentially up to a given depth. Independent worker processes are then created to evaluate the sub-trees on this tree level. Figure 1 shows a binary divide-\&-conquer tree unfolded up to depth 3. Processes are indicated by squares.
Again, we assume a regular distribution of work complexity among the tasks of the same tree level. When descending to the $d$-th level of the divide-\&-conquer tree, $r^d$ processes will be created. Starting with input size $n$, these subtasks will process problems of size $n/r^d$. At a certain level of parallel descent with input size $k$, we have the penalty of $\tilde{A}(k, p)$ for spawning parallel tasks and for communication. It might be hard to distinguish between communication overhead and parallel overhead, but we can estimate the sum of them. However, $T(n)$ might not be linear, i.e. $T(n) \neq lT(n/l)$. Therefore, we add another term $O(n, k, p)$ for the work needed for dividing and merging the tasks from length $n$ to length $k$.

So $r$-ary divide-\&-conquer of depth $d$ for input length $n$ is

$$T(n) = r^d T(n/r^d) + O(n, n/r^d, 1).$$

The work of the flat expansion skeleton amounts to

$$W_{\text{flat DC}}(n, p) = p \sum_{i=0}^{d-1} r^i \tilde{A}(n/r^i, p) + r^d T(n/r^d) + pO(n, n/r^d, p),$$

and time is

$$T_{\text{flat DC}}(n, p) = \sum_{i=0}^{d-1} r^i \tilde{A}(n/r^i, p) + \frac{r^d}{p} T(n/r^d) + O(n, n/r^d, p).$$

Section 4.2 shows an example (Karatsuba multiplication) using this divide-\&-conquer skeleton.

### 2.3 Iteration Skeleton

The iteration skeleton represents a parallel do-while loop, where $p$ iterations are done in parallel and only then the predicate is evaluated. Based on its value the program either processes further $p$ iterations or it terminates. It is natural, that the runtime of this skeleton directly depends on the worker function for a single iteration, execution time of which we denote with $s(n)$. We assume that the loop comprises exactly $k$ iterations and that the work $s(n)$ is the same for each iteration. Hence: $T(n) = ks(n)$ and $W_{\text{iter}}(n, p, k) = ks(n) + p\tilde{A}(n, p)$. It follows

$$T_{\text{iter}}(n, p, k) = \frac{k}{p} s(n) + \tilde{A}(n, p).$$

In Section 4.3 we discuss parallelisation of an example program (Rabin-Miller test) with this skeleton.

### 3 Analysing the Penalty Term

The shape of $\tilde{A}(n, p)$ is the key to rating the parallel performance quality. As this penalty term depends on both the problem size $n$ and number of PEs $p$, it is important that it does not increase for growing $p$. Otherwise, the implementation does not scale well.
We estimate $T(n)$ and $A(n, p)$. Our aim is to find separate approximations for these two terms, as $T(n)$ and $A(n, p)$ have different nature. In order to do so, we determine several values of $T(n)$ and $A(n, p)$, then we use statistical techniques on the resulting data sets.

We use different methods to predict values of $T(n)$ and $A(n, p)$ at non-measured input points. We could use straightforward polynomial interpolation, but for better results we sample more points and use one of the following methods. One approach is cubic spline interpolation [5], another one is local polynomial regression fitting [3, Chapter 8], as implemented in R [15] functions `spline` and `loess`. This will be how we refer to these approaches. Also we use linear model fitting with orthogonal polynomials constructed from actual input [3, 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 a mean of two best methods for a particular approach.

The `spline` method interpolates the measured data points exactly. Contrary to that, all other methods here utilise regression fitting. 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 almost always it is not applicable for us. Its generalisation `lm(poly)` uses orthogonal polynomials to relax 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. In this property `loess` is similar to `spline`.

By transforming Equation (*) we get

$$A(n, p) = T(n, p) - T(n)/p$$

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

4 Examples

Our runtime experiments have been performed on an 8 core 64 bit Intel machine with 16 GB RAM. We use the Glasgow Haskell Compiler 6.8.3 for the sequential program executions and parallel Eden extension of this compiler for the parallel executions. We always determined the mean runtime of five program runs without using any options for memory allocation or garbage collector tuning.

4.1 Gauß Elimination – `parMap`

We measure the time needed to compute the $LU$ decomposition of a permuted scaled $n \times n$ Pascal matrix modulo $r$ primes. The program is parallelised using the simple `parMap` skeleton with an input list of size $r$, because the map is done over the different residue classes [12].

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 to parallelise this program for $p < 8$. When using less than the maximum number...
of PEs, multiple processes will be executed by the same PE. This causes an imbalance when processes cannot be evenly distributed to PEs, i.e. when 8 is not a multiple of the number of PEs. Thus, 2, 4 and 8 PE configurations perform best.

In spite of this, we refrain from estimating the execution time at 8 PEs, as this special case is not connected with 6 and 7 PE configurations. We could use only the special cases, but this way we would have not enough data points, as we perform our measurements on an 8 PE machine. We stress again that we use a non-optimal program to demonstrate the strength of our approach.

Table 1 shows the measured times. Figure 2 shows the estimation of sequential runtime $T(n)$ for $n = 120, 150$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
<th>120</th>
<th>150</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n, 1)$</td>
<td>0.7368</td>
<td>1.3365</td>
<td>2.3677</td>
<td>3.6826</td>
<td>5.1556</td>
<td>7.4163</td>
<td>10.059</td>
<td>17.352</td>
<td>34.50</td>
</tr>
<tr>
<td>$T(n, 7)$</td>
<td>0.2772</td>
<td>0.4833</td>
<td>0.8069</td>
<td>1.2549</td>
<td>1.7971</td>
<td>2.6108</td>
<td>3.6038</td>
<td>6.2055</td>
<td>11.535</td>
</tr>
<tr>
<td>$T(n, 8)$</td>
<td>0.2535</td>
<td>0.4447</td>
<td>0.6985</td>
<td>1.0636</td>
<td>1.3628</td>
<td>2.0783</td>
<td>2.6459</td>
<td>4.6065</td>
<td>10.036</td>
</tr>
</tbody>
</table>

Table 1. Gauß elimination. Measured time. **Bold** items will be forecast w. r. t. $n$, **boxed** items will be forecast w. r. t. $p$.

Now, as 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 did not consider lm(poly) for estimating $\bar{A}$ for its poor performance there. Combined, we can apply Equation (*) and obtain the complete time estimation. We obtain an estimate $T(120, 7) = 6.10$ seconds, which corresponds to the appropriate value from Table 1 up to a relative error $-1.69\%$. In this example we have assumed, there was the possibility to measure time on a 7 PE machine, but no one had run the test program for the input length 120 or 150. Now we do the converse: we have measurements for task size 100 on smaller PE numbers, but do not have a machine with 7 PEs to measure time there. So we use the data for the estimation of $\bar{A}(n, p)$ w. r. t. $n$ and $p$. With mean we denote the mean of spline and loess methods.

Fig. 2. Gauß elimination. Predicting $T(n)$ for $n = 120, 150$. 

(a) Predicting the sequential runtime

<table>
<thead>
<tr>
<th>Method</th>
<th>spline</th>
<th>loess</th>
<th>lm(poly)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rel. error, %</td>
<td>$-16.17%$</td>
<td>$-6.605%$</td>
<td>$-3.411%$</td>
</tr>
</tbody>
</table>

(b) Relative error for $n = 120$
(a) Predicting the penalty values w. r. t. $n$ (b) Predicting the penalty values w. r. t. $p$

<table>
<thead>
<tr>
<th>Method</th>
<th>spline</th>
<th>loess</th>
<th>mean</th>
<th>Rel. error, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c) Relative error for $n = 120$</td>
<td>-3.601</td>
<td>2.143</td>
<td>-0.7294</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>spline</th>
<th>loess</th>
<th>lm(poly)</th>
<th>mean</th>
<th>Rel. error, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d) Relative error for $p = 7$</td>
<td>-16.9</td>
<td>13.53</td>
<td>43.06</td>
<td>-1.682</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3. Gauß elimination. Left: predicting $A(n, p)$ w. r. t. $n$. We fix $p = 7$ and predict values for $n = 120, 150$ basing on $n \leq 100$. Right: predicting $A(n, p)$ w. r. t. $p$. We fix $n = 100$ and predict the value for $p = 7$ basing on $p \leq 6$.

4.2 Karatsuba Multiplication – Divide-&-Conquer

Karatsuba multiplication is a ternary divide-&-conquer algorithm for multiplying large integers. We perform the computation for factors of equal size, uniformly distributed between 16000 and 64000 digits, predicting the values for 60000 and 64000. Relative speedup is used, we are not able to separate $O(n, k, p)$ from $T(n)$. Still our statistic approach does a good job in predicting the execution time. The possible reason: negligible $O(n, k, p)$ in a chosen setting, as communication cost on a SMP machine is very small. We omit the plots here and state only the essential information. We estimate $T(60000)$ with spline with a relative error of $-0.014\%$. The next best estimation is lm(poly) of third degree with $1.3\%$. The latter method for $T(64000)$ results in the best relative error of $1.9\%$, whereas spline method produces a relative error of $2.76\%$. The loess method is not significantly worse. As for $A(n, p)$ w. r. t. $n$, we obtain relative error of $2.3\%$ for $A(60000, 8)$ with lm(poly) of degree 3. The most reliable estimation is produced by loess with $-5.4\%$ and $2.08\%$ relative error for estimations of input length of 60000 and 64000 correspondingly. This results in a relative error for parallel runtime estimation values of 11.01 seconds and 12.09 seconds for the same inputs. This corresponds to $0.14\%$ and $1.91\%$ relative error. We see, we can successfully estimate the runtime also for divide-&-conquer-based computation.

We also experiment with non-uniform input data distribution. We estimate $T(128000)$ with lm(poly) with $0.021\%$ relative error with prior knowledge of the runtimes only up to 64000. Hence our formalism is applicable to such “long-distance” runtime estimations.
Table 2. The runtimes for Rabin-Miller test. **Bold** value will predicted.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Estimate for</th>
<th>$T(n)$</th>
<th>$\bar{A}(n, 7)$</th>
<th>$T(n, 7)$</th>
<th>Rel. error, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>spline + spline</td>
<td>154.43</td>
<td>8.686</td>
<td>27.99</td>
<td>-3.88</td>
<td></td>
</tr>
<tr>
<td>spline + loess</td>
<td>154.43</td>
<td>8.696</td>
<td>28.00</td>
<td>-3.85</td>
<td></td>
</tr>
<tr>
<td>spline + lm(poly)</td>
<td>154.43</td>
<td>8.890</td>
<td>28.19</td>
<td>-3.18</td>
<td></td>
</tr>
<tr>
<td>loess + spline</td>
<td>153.53</td>
<td>8.686</td>
<td>27.88</td>
<td>-4.27</td>
<td></td>
</tr>
<tr>
<td>loess + loess</td>
<td>153.53</td>
<td>8.696</td>
<td>27.89</td>
<td>-4.23</td>
<td></td>
</tr>
<tr>
<td>loess + lm(poly)</td>
<td>153.53</td>
<td>8.990</td>
<td>28.08</td>
<td>-3.57</td>
<td></td>
</tr>
<tr>
<td>lm(poly) + spline</td>
<td>156.06</td>
<td>8.866</td>
<td>28.19</td>
<td>-3.18</td>
<td></td>
</tr>
<tr>
<td>lm(poly) + loess</td>
<td>156.06</td>
<td>8.696</td>
<td>28.20</td>
<td>-3.15</td>
<td></td>
</tr>
<tr>
<td>lm(poly) + lm(poly)</td>
<td>156.06</td>
<td>8.890</td>
<td>28.40</td>
<td>-2.48</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Results of runtime estimation for Rabin-Miller primality test at $n = 11213$ for 7 PEs. The measured value was 29.12 seconds.

4.3 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. The positive result of the test does not ensure that the input is prime, but does so with a certain probability. We perform the test on Mersenne primes, parameter $n$ means that the number $2^n - 1$ is tested for primality. We choose $n$ 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 non-uniform distribution of input values.

For estimating $\bar{A}(n, 7)$ we used *spline*, *loess* and *lm(poly)* of degree 3 for $n = 11213$ on 7 PEs. We use $T(n, 1)$ as an estimate for $T(n)$: in other words, we calculate with relative and not with absolute speedups. The results are presented in Table 3. An overview for single components is available in Figures 4(a)–4(d).

Our best method is using *lm(poly)* for both components, resulting in a relative error $< 2.5\%$, whereas the worst method was only $-4.27\%$ inexact. We can predict the runtime of parallel Rabin-Miller test quite accurate.

5 Serial Fraction

In their work “Measuring Parallel Processor Performance” [10] A. H. Karp and H. P. Flatt introduced the notion of **serial fraction**. We state it here with absolute speedups. Given parallel time $T(n, p)$ on $p$ PEs and 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. On the other hand, if the serial fraction decreases, this shows problems with the sequential implementation. We obtain the serial fraction...
(a) Predicting $T(n)$

(b) Predicting $\bar{A}(n, p)$

(c) Relative error for $T(n)$

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

Fig. 4. Predicting values for both components for Rabin-Miller test.

Table 4. Serial fraction and parallel penalty values for Rabin-Miller test. The boxed value will be estimated w. r. t. $p$.

<table>
<thead>
<tr>
<th>$p$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bar{A}(9689, p)$</td>
<td>0</td>
<td>4.89</td>
<td>3.39</td>
<td>7.41</td>
<td>7.91</td>
<td>6.61</td>
<td>4.23</td>
<td>6.19</td>
</tr>
<tr>
<td>$f(9689, p)$</td>
<td>0.0915</td>
<td>0.0476</td>
<td>0.0926</td>
<td>0.0926</td>
<td>0.0743</td>
<td>0.0463</td>
<td>0.0663</td>
<td></td>
</tr>
</tbody>
</table>

for the test cases presented above. For Gauß elimination, we use a $100 \times 100$ input matrix, both measures—our parallel penalty $\bar{A}$ and the serial fraction—are presented in Figure 5(a). The shapes of the curves correspond to the load distribution in the computation, 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 PE. The value of the parallel penalty $\bar{A}(n, p)$ is almost constant for 2, 4 and 8 PE versions. But we know that exactly these versions are optimal in the load distribution. The serial fraction shows the same for 4 and 8 PE versions, but states a larger serial component for 2 PE.

The serial fraction for Rabin-Miller primality test with the input 9689 is shown in Figure 5(b). Here we can clearly see the same problematic for PE 3 and 7, as we need to perform exactly 20 iterations. Here, again, this issue can be observed in both approaches. Serial fraction might be a bit more clear in issuing one with this message, but our approach has other benefits we discuss below.

We predict the values of the serial fraction $f(n, p)$ and of $\bar{A}(n, p)$, w. r. t. the PE count $p$. For estimations of the serial fraction for Rabin-Miller test, we use all
possible methods, but yield no sustainable result. The best result is produced by \texttt{loess} with a bit less than 20\% relative error, all other approaches are worse by far, cf. Figure 6, bottom right. Admittedly, the estimation of our parallel penalty w. r. t. \( p \) was also not the best one, but nevertheless we obtained an estimate with less than 2\% relative error, as shown on the same figure, left and top right.

We have tried to predict the penalty values \( \tilde{A}(n, p) \) for different \( p \) values in the Rabin-Miller test, using input length 9689 and relative speedups. With data from 1 PE to 6 PEs we estimated the overhead for 7 PEs.\footnote{Due to irregularities} Relative speedup is used.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Parallel penalty} & & & \\
\hline
Method & spline & loess & \\
\hline
Rel. error, \% & 46.15 & -16.76 & \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Serial fraction} & & \\
\hline
Method & spline & loess & lm & \\
\hline
Rel. error, \% & 58.65 & 19.28 & 79.23 & \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Serial fraction} & & & & \\
\hline
Method & lm & mean & & \\
\hline
Rel. error, \% & 86.00 & -238\% & 386\% & \\
\hline
\end{tabular}
\end{table}

Fig. 6. Rabin-Miller test. Left: predicting the penalty values \( \tilde{A}(n, p) \) w. r. t. \( p \). Right, top: relative error for the latter. Right, bottom: relative error for serial fraction prediction. In both cases the input \( n \) is 9689. We predict values for \( p = 7 \). Relative speedup is used.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.png}
\caption{Comparing the serial fraction with our approach.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6.png}
\caption{Rabin-Miller test. Left: predicting the penalty values \( \tilde{A}(n, p) \) w. r. t. \( p \). Right, top: relative error for the latter. Right, bottom: relative error for serial fraction prediction. In both cases the input \( n \) is 9689. We predict values for \( p = 7 \). Relative speedup is used.}
\end{figure}
in A w. r. t. p, it was a quite hard task. The best estimations were \text{lm(poly)} of degree 3 and \text{loess}, they provided relative error of 14.38% and $-16.76\%$ respectively. However, a \text{mean} of these two estimates has an acceptable accuracy of $-1.19\%$ relative error. These results are depicted in Figure 6, top right.

The parallel penalty values and serial fraction represent similar concepts. However the penalty values are better forecast with the statistical methods used in this paper. One possible reason might be an additional data point at PE 1 for the case with absolute speedups, unavailable for serial fraction. However, as the case of Rabin-Miller test shows, also with relative speedups we provide an estimation, which is better by an order of magnitude.

6 Related Work

Our approach bears—as everything on this topic—a certain grade of similarity to Amdahl’s law \cite{1}. Exactly as Amdahl did, we assume a perfect parallelisation of the computation, but consider also the unavoidable overhead. I.e. 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 \cite{13} and the formal cost model behind NESL \cite{2}. However, our approach is different. First, we derive the time and work from time \textit{measurements} for the runs on different numbers of PEs. Opposed to that, the skeleton analysis in \cite{13} is based on latency and message-passing costs. The NESL complexity model \cite{2} takes a “bottom-up” approach, trying to assign cost to single semantic operations. We look in a “top-down” manner on the total runtime and divide it into very coarse blocks.

Regarding approaches for parallel quality measure: The notion of a serial fraction as defined in \cite{11} is similar to ours. It has been discussed in Section 5 with the result that our approach allows better predictions. Isoefficiency \cite{6}, scaled speedup \cite{5} and other approaches are less similar to penalty values $A(n, p)$, as we do not strive a larger input on a larger PE count, which keeps the efficiency the same.

7 Conclusions and Future Work

We predict the execution times of parallel programs in an elegant manner. Our method is different from previously known approaches. It does not depend on source code analysis or on a special semantic rules. Instead, our method is empowered by computational statistics. We separate estimations for parallel computation and parallel overhead, hence the different forecasting models can be used for each of them. This makes sense, given different nature of these processes, and enables better estimations. We have used our technique to predict execution times for three scientific computing methods encapsulated in different algorithmic skeletons.

In this paper, we have tested our technique with Eden and multicore SMP hardware. It will be necessary to experiment with larger-scale parallel systems, other languages or middleware. Moreover, we plan to seek for more advanced
prediction methods for the sequences. More sophisticated statistical methods 
are due. A proper direction would be, for instance, separation of special per-
formance cases (like 4 and 8 PE versions of Gauß elimination in Section 4.1), 
an approach for absolute speedup divide-&-conquer estimation (relative speedup 
in Section 4.2). A machine learning approach to the estimation of separate se-
quences and an adaptive runtime environment for parallel programs present 
rather long-term ideas for future work.

Acknowledgements. We would like to thank Thomas Horstmeyer for a fruitful 
discussion on divide-&-conquer.

References

1. G. M. Amdahl. Validity of the single processor approach to achieving large scale 
2. G. E. Blelloch and J. Greiner. A Parallel Complexity Model for Functional Lan-
6. A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of 
7. A. Y. Grama, V. Kumar, A. Gupta, and G. Karypis. Introduction to parallel 
11. V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and archi-
12. Oleg Lobachev and Rita Loogen. Implementing data parallel rational multiple-
13. R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Ab-
straction in Eden. In F. A. Rabhi and S. Gorlatch, editors, Patterns and Skeletons 
15. R Development Core Team. R: A Language and Environment for Statistical Com-