Estimating Parallel Performance, A Skeleton-Based Approach *

Oleg Lobachev 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. We assume that programs are described using algorithmic skeletons. Therefore 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 measure 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 numbers of processing elements. From these data, conclusions on parallel performance and scalability are drawn.

Categories and Subject Descriptors C.4 [*PERFORMANCE OF SYSTEMS*]: Modeling techniques, Performance attributes; D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming—Parallel programming

General Terms Measurement, Performance

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

1. Introduction

Since Amdahl's law [1, 19] the quest for modelling parallel performance is open. A nice summary of existing approaches is presented in [17, 25]. 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 number of processing elements, we envision parallel runtime as the sequential "work" distributed over a number of processing elements plus an additional penalty term.

* Supported by DFG grant LO 630-3/1.

HLPP'10, September 25, 2010, Baltimore, Maryland, USA.

A simple parallelisation of programs can be achieved using algorithmic skeletons [10]. The latter capture common patterns of parallel computation and can straightforwardly be instantiated for specific problem areas. The skeleton abstraction of a parallel program can be used to derive specialised expressions for the parallel runtime, again in terms of sequential work and parallel overhead.

The first goal of this paper is an accurate prediction of parallel runtimes for new input sizes 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 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 forecast of parallel runtime. Secondly, the estimated parallel overhead is a measure for the scalability of a given parallel program, similar to [20]. This provides insight into the performance properties of the parallel program. Is the "bad" part increasing, bottlenecks or similar problems in the code are likely. However, identifying them is beyond the scope of this work.

We show the practicality of our approach for selected programs from scientific computing, implemented both on a large-scale C+MPI [29] system and in the parallel functional programming language Eden [28]. The latter test programs are parallelised using standard skeletons from Eden's skeleton library [27]. Although our Eden system is used for most of the experiments, our approach is completely language-independent. The technique is applicable to any parallel system, ranging from a multicore machine to a supercomputer. The skeletons are also not a must for program *implementation*, they merely describe a particular pattern of parallel computation in the analysed program.

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 and 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 four 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 [20]. Further related work is considered in Section 6 and Section 7 concludes and gives an outlook on future work.

2. Runtime Estimation

Let n denote the input 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.* [17]) 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

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

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 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. 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. We use skeletons as an abstract parallelisation paradigm description. The program to be analysed is not required to be implemented using skeletons, although our Eden examples in Section 4 are.

2.1 Parallel Map

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 input list is the same. Otherwise, one should use a dynamically load-balancing `workpool` instead of `parMap`. Our assumption means that $T(n) = nT(1)$. Thus, the amount of work for `parMap` is $W_{\texttt{parMap}}(n,p) = nT(1) + p\bar{A}(n,p)$, while the time is

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

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 then statically divided into almost equally sized blocks, one per worker process. Each worker process works on its block sequentially. The implicit assumption is still the equal "cost" of single tasks. The tasks are mapped *statically* onto the processes, thus implementing a static load-balancing. Following [30], we call this skeleton `farm`. The time needed for such a `farm` can be described by

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

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

In Section 4.1, we consider a parallelisation of an example program (Gauß elimination) with the `farm` skeleton. In Section 4.2 we consider a large-scale data parallel program.

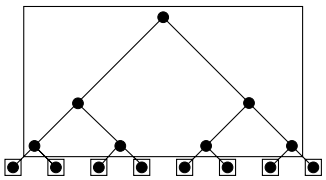


Figure 1. A binary tree of depth 3 for flat divide and conquer expansion skeleton.

2.2 Divide and Conquer

Divide and conquer is a typical example for task parallelism. We consider a regular divide and 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* skeleton, 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 and 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 and conquer tree, r^d processes will be created. Starting with input size n, these processes will process subtasks of size n/r^d. At a certain level of parallel descent with input size k, we have the penalty of $\bar{A}(k,p)$ 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 size n to size k. So the sequential time of r-ary divide and conquer of depth d for input size n can be expressed as

$$T(n) = r^d T\Big(\frac{n}{r^d}\Big) + O\Big(n, \frac{n}{r^d}, 1\Big).$$

The work of the flat expansion skeleton amounts to

$$W_{\text{flat DC}}(n,p) = \underbrace{p\sum_{i=0}^{d-1} r^i \bar{A}\Big(\frac{n}{r^i}, p\Big)}_{=:\bar{A}(n,p)} + r^d T\Big(\frac{n}{r^d}\Big) + pO\Big(n, \frac{n}{r^d}, p\Big),$$

and time is

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

Section 4.3 shows an example (Karatsuba multiplication) using this divide and conquer skeleton.

2.3 Iteration

The iteration skeleton represents a parallel `do-while` loop, where p iterations are done in parallel before the predicate is evaluated. Depending 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, the 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 $W_{\text{iter}}(n,p,k) = ks(n) + p\bar{A}(n,p)$. It follows

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

In Section 4.4 we discuss parallelisation of an example program (Rabin-Miller test) with this skeleton.

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

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

3. Analysing the Penalty Term

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 p of PEs, it is important that it does not increase for growing p. Otherwise, the implementation does not scale well.

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.

We use different methods to predict values of $T(n)$ and $\bar{A}(n,p)$ for non-measured input sizes. 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 [13], another one is local polynomial regression fitting [6, Chapter 8] (*cf.* [8, 9]). We refer to these approaches using the `R` function names `spline` and `loess` [31]. Also we use linear model fitting with orthogonal polynomials constructed from the actual input [6, 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 decision on the best method *can* be done automatically. Given an $\varepsilon > 0$ and some existing runtime measurements, 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 produces 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 the real value is unknown ("real life" estimation) and we have to resort to the mean of two methods, two strategies exist on choosing 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. In the "real life" estimation we discard the more distant value and compute the mean of the two remaining values.
- If none of the methods yields a satisfying result, reconsider ε. Make further measurements. Otherwise fail.

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.

By transforming Equation (*) 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 runtime for p PEs minus the sequential runtime divided by p, where these values can be measured or estimated. Note however that $\bar{A}(n,p)$ depends on p.

4. Experiments

Most our runtime experiments have been performed on an eight core 64 bit Intel machine with 16 GB RAM. We have used the Glasgow Haskell Compiler 6.8.3 for the sequential program executions and the parallel Eden extension of this compiler for the parallel executions. We always determined the mean runtime 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 quadcore 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.

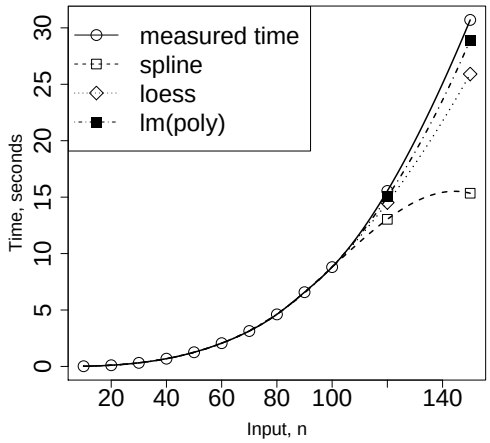
4.1 Gauß Elimination – Parallel Map

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 [26].

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 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 PE configurations perform best. This knowledge could be acquired using source code inspection or process activity profiles, e.g. using Eden TV [4]. In Section 5 we will see, how to obtain the same information with our approach.

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 then we would not have enough data points, as we perform our measurements on an 8 PE machine. We stress again that we use a non-optimal program with known weaknesses to demonstrate the strength of our approach.

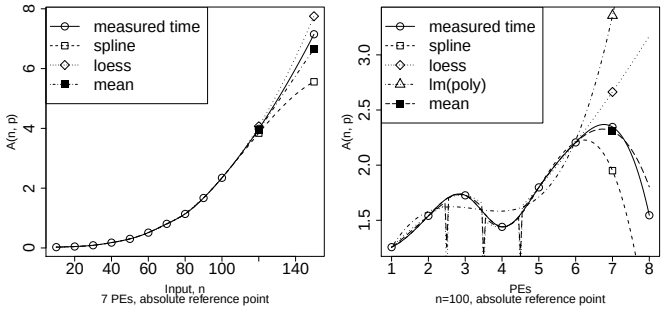
Table 1 shows the measured times. Figure 2 shows the estimation of the sequential runtime $T(n)$. Figure 3 shows the estimation of $\bar{A}(n,p)$ w.r.t. n and p. With `mean` we denote the mean of `spline` and `loess` methods.



Rel. error for $n = 120$

Method	Rel. err., %
`spline`	-16.17
`loess`	-6.605
`lm(poly)`	-3.41

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



(a) Penalty values w.r.t. n (b) Penalty values w.r.t. p

Method	`spline`	`loess`	`mean`
Rel. error, %	-3.601	2.143	-0.7294

(c) Relative error for $n = 120$

Method	`spline`	`loess`	`lm(poly)`	`mean`
Rel. error, %	-16.9	13.53	43.064	-1.682

(d) Relative error for $p = 7$

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

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 did not consider `lm(poly)` for estimating $\bar{A}$ due to the 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 the 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. p and the already measured $T(100)$. We obtain an estimate $T(100, 7) = 3.564$ seconds, which is -1.1% accurate. So, it is possible to estimate the parallel runtime both w.r.t. task size and w.r.t. PE count with significant accuracy.

4.2 Mass Transport in Porous Media – Parallel Map

As a further example we consider the lattice-Boltzmann method from fluid flow and mass transport simulation. The data originate from [21, 22], the experiments were performed by Siarhei Khirevich and Anton Daneyko 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 [22–24] 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).

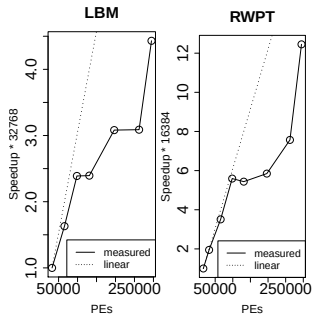
The latter two phases clearly dominate the computational complexity of the method. We chose to focus on the LBM phase. Due to the dimensions of the chosen lattice—$632 \times 632 \times 294912$—a one-dimensional decomposition is possible. Basically, the pipe is cut in length and each "slice" is assigned to a PE. Hence, we have a data parallel implementation. The amount of spheres in each slice also 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, *cf.* [7]. We show the speedup curves for both LBM and RWPT in the left part of Table 2.

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 in a round-robin manner. Thus in [22], 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 right part of Table 2. 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 left part of Table 2, RWPT has major speedup problems in the middle part of the scale, i. e. when using between 100000 and 250000 PEs. The reason is a task distribution imbal-



RWPT

PE, p	16384	32768	65536	98304	131072	196608	262144	*294912*
$T(n,p)$	1.93	0.99	0.55	0.345	0.355	0.33	0.255	*0.155*
Imbalance, %	0	0	50	0	25	50	12.5	*0*

LBM

PEs, p	32768	65536	98304	131072	196608	*262144*	294912
$T(n,p)$	16.285	9.99	6.82	6.80	5.284	*5.273*	3.675

Table 2. RWPT and LBM on a supercomputer, courtesy of [21]. The task size n is fixed. Left: the speedups for both computations. Right, top: RWPT execution times. Right, bottom: LBM execution times. The *boxed* values will be estimated.

ance, *cf.* the imbalance values given in the top right part of Table 2. 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. This is bad, as the factor $\lfloor n/p \rfloor$ is small, so the imbalance is severe. Our `farm` scheme is not suitable for this, as it assumes $n \gg p$. 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.

LBM: Time measurement The time measurement data for LBM on Jülich Blue Gene/P presented in the right bottom part of Table 2 have been taken from [21]. 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 and the knowledge, that the program in question performs computations in an essentially data-parallel manner (i. e. using parallel map), 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. All we know is the scheme used for parallelisation.

We present the values for $\bar{A}(n,p)$ w. r. t. p in the top part of Figure 4. 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 4. Notably, a direct runtime estimation—an attempt to predict the parallel time directly from the number of PEs, using the same data, but without using Equation (*)—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.

Generalisation We have seen a good performance of `lm(poly)` for estimating $\bar{A}$ w. r. t. p for LBM. However, the same method seems to fail for Gauß elimination as reported in Section 4.1. We assume that this is due to the "regular" cases of perfectly distributed tasks. We have tried to estimate $\bar{A}$ w. r. t. p for Gauß elimination without the regular cases, that is for $p \neq 2, 4, 8$. We were estimating the parallel overhead for $p = 7$ using all other

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

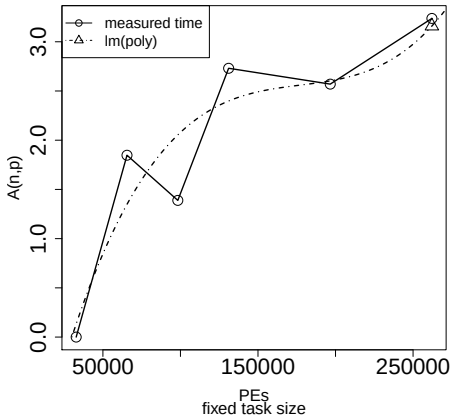


Figure 4. Computing $\bar{A}(n,p)$ for LBM. Top: computed values for $\bar{A}$. The *boxed* value will be estimated on the bottom.

available values. We obtained quite bad results because too few measured data points remained. However, judging by relative error, `spline` and `lm(poly)` of degree 3 were equally good. Hence, we can conjecture that for `parMap`-related skeletons `lm(poly)` is the most suitable method of the statistical techniques regarded here.

4.3 Karatsuba Multiplication – Divide and Conquer

The Karatsuba multiplication is a ternary divide and conquer algorithm for multiplying large integers. The following results have been obtained with Eden on a multicore machine. We perform the computation for integers of equal size, where the size of a large integer is its number of digits. The integer size has been uniformly distributed between 16000 and 64000 digits. We predict the values for 60000 and 64000. See the top of Table 3 for a snapshot of the data. Since the relative reference point is used, we are not able to separate $O(n,k,p)$ from $T(n)$. We discard $O(n,k,p)$ in our setting, as the communication cost on the multicore machine is very small. We have estimated $T(60000)$ 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 1.3%. The latter method for $T(64000)$ has the best relative error of 1.9%, whereas the `spline` method produces a relative error

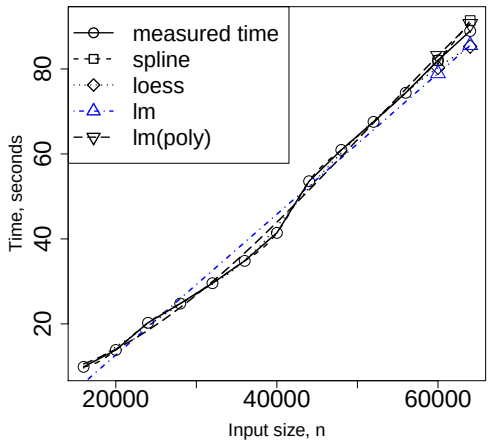
Uniform

$n \cdot 1000$	16	20	24	28	32	36	40	44	48	52	56	**60**	**64**
$T(n,8)$	1.29	1.78	2.61	3.19	3.74	4.47	5.37	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**
$T(n,8)$	0.0654	0.0818	0.129	0.222	0.47	1.28	3.74	11.86	**36.66**

Table 3. Karatsuba multiplication on 8 PEs. The **bold** values will be estimated.



Relative error for $T(60000)$

Method	`spline`	`loess`	`lm`	`lm(poly)`
Rel. error, %	-0.014	-2.08	-3.76	1.30

Figure 5. Karatsuba multiplication. Estimating the sequential runtime $T(n)$ for $n = 60000$ and 64000 on uniform data.

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**

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

Table 4. Rabin-Miller test. On the top: measured times. The **bold** value is estimated at the bottom with all available methods.

of 2.76%. The `loess` method is not significantly worse. See Figure 5 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 -5.4% and 2.08% relative errors for the estimations of input lengths 60000 and 64000 respectively. This results in parallel runtime estimation values 11.01 seconds and 12.09 seconds for the same inputs. This corresponds to relative errors 0.14% and 1.78%. Thus an appropriate estimation of runtime does also work for a divide and conquer-based computation.

We have also experimented with a non-uniform input data distribution, given at the bottom of Table 3. The input size varied between 500 and 128000 digits. 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. We obtain the estimation value of 36.67 seconds (rounded up to second digit) with a remarkable relative error of 0.021% with prior knowledge of the runtimes only up to 64000. Hence our formalism is also applicable to such "long-distance" runtime 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. 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, 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 input values. We show the time measurement data in the top part of Table 4.

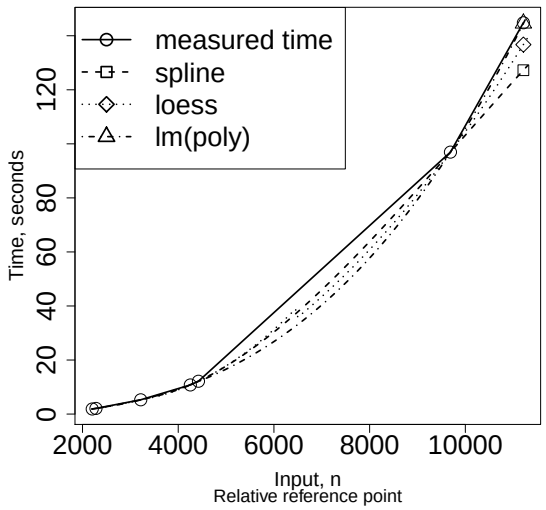
For estimating $\bar{A}(11213,8)$ we used `spline`, `loess`, `lm(poly)` of degree 3 and `mean` of latter two. We use $T(n,1)$ as an estimate for $T(n)$: in other words, we calculate with relative and not with absolute reference point. The results are presented in the bottom part of Table 4. An overview for single components is available in Figure 6. 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. Thus the prediction of runtime of parallel Rabin-Miller test has been quite accurate.

5. Serial Fraction

In their work "Measuring Parallel Processor Performance" [20] 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

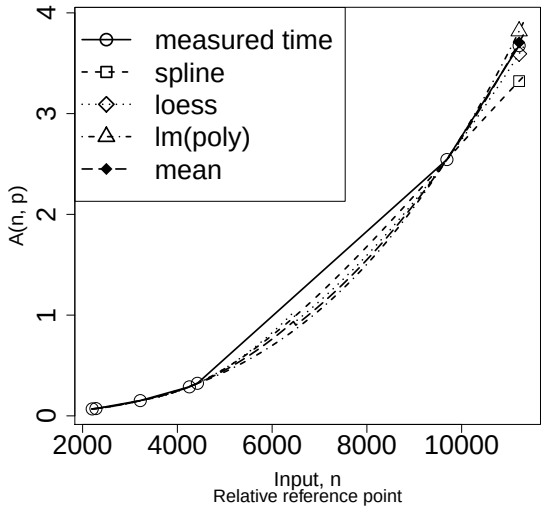
$$f(n,p) = \frac{T(n,p)/T(n) - 1/p}{1 - 1/p}.$$

The serial fraction should be constant: if it increases, we have a parallelisation resulting in poor speedups. If the serial fraction



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 6. Predicting values for both components for Rabin-Miller test. Left $T(n)$, right $\bar{A}(n,p)$.

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—are shown in Figure 7(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 PE. The value of the parallel penalty $\bar{A}(n,p)$ is almost constant for 2, 4 and 8 PE versions. Exactly these versions are optimal in the load distribution. The serial fraction shows the same for 4 and 8 PE versions, but indicates a larger serial component for 2 PEs.

The serial fraction for Rabin-Miller primality test with the input 9689 is shown in Figure 7(b), the numerical values are in Figure 7(c). We can clearly see problems with load balancing for 3 and 6 PEs, as we need to perform exactly 20 tasks. However, 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. We deduce, 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. Here, again, all the discussed effects can be observed in both approaches.

6. Related Work

Our approach bears—as everything on this topic—a certain grade of similarity to Amdahl's law [1]. 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 [27] and the formal cost model of NESL [5]. 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 in [27] is based on latency and message-passing costs. The NESL complexity model [5] 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.

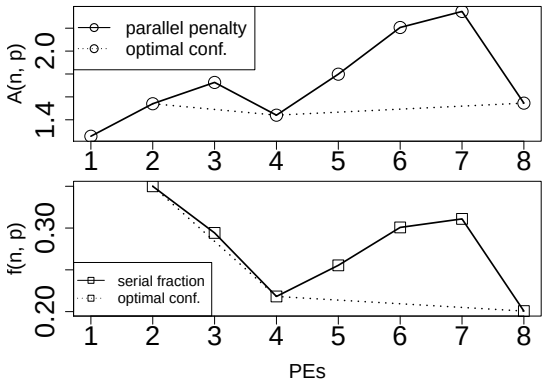
Further approaches on skeleton-based performance evaluation include [3, 11]. In [11] Cole and Hayashi regard a BSP-like cost model, assigning costs to basic elements of a parallel program. A similar approach to ours, but more fine-grain and still focusing on BSP is [35]. Beside BSP [33], models like LogP [12] and message passing models exist. See [32] for a runtime prediction approach for the latter. A rather theoretical well-investigated performance model is the PRAM model [14] and its variants. An alternative program classification approach are Berkley dwarfs [2], however in this case several particular types of tasks are identified, in contrast to abstracting from the task type with skeletons [10].

Regarding approaches for parallel quality measure, the notion of serial fraction as defined in [25] is similar to ours. Isoefficiency [16], scaled speedup [18] and other approaches are less similar to penalty values $\bar{A}(n,p)$, as we do not aim for a larger input on a larger PE count, which keeps the efficiency the same.

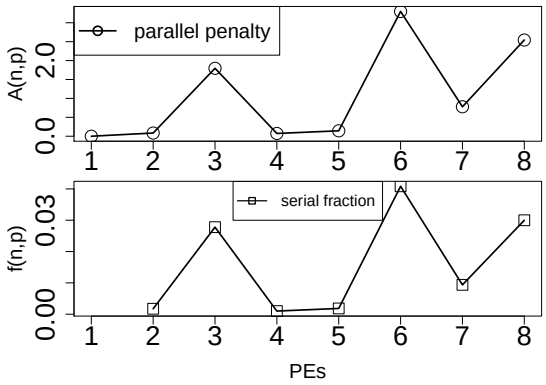
A further research direction, orthogonal to our approach is hotspot and bottleneck analysis. We refer to modern visualisation tools, like [15]. In parallel functional programming, Eden TV [4] and ThreadScope [34] are used. We were able to confirm our assumptions on bad process placement of programs from Sections 4.1 and 4.4 with Eden TV.

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 special semantic rules. Instead, our method is empowered by computational statistics and the skeleton abstraction. We separate estimations for parallel computation and parallel overhead, hence differ-



(a) Gauß elimination



(b) Rabin-Miller test

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
$f(9689, p)$	–	0.001741	0.027790	0.001005	0.001847	0.04084	0.009379	0.029992

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

Figure 7. Comparing the serial fraction with our approach.

ent forecasting models can be used for each. This makes sense, given the different nature of these processes, and enables better estimations. We have used our technique to predict execution times for four scientific computing methods, abstracted in three algorithmic skeletons.

In this paper, we have tested our technique primarily with Eden and multicore SMP hardware, but also on a peta-scale supercomputer, using C+MPI. We look forward to more experiments with large-scale systems, further languages and middleware. We also would like to perform detailed analyses of further, more complex skeletons. We need to experiment with the estimation of *combined* skeletons, the two prominent cases are skeleton composition and nested skeletons. We also need to extend our formalism for imbalanced parallel map: i. e. `farm` where $p < n < kp$ for a *small* k. 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 are necessary. A proper direction would be, for instance, the separation of special performance cases (like 4 and 8 PE versions of Gauß elimination in Section 4.1 or the aforementioned task imbalance). Statistically sound statements on the *distance* of robust extrapolation using the structure of the presented model is also left for future work. An approach for absolute reference point divide and conquer estimation would be interesting (relative reference point in Section 4.3). Another interesting topic is predicting how many measured values do we need for a good prediction of runtime. A machine learning approach for estimating of separate sequences and an adaptive runtime environment for parallel programs present rather long-term ideas for future work.

Acknowledgments

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 the anonymous reviewers for their helpful comments.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the spring joint computer conf.* ACM, 1967.

[2] K. Asanovic, R. 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.

[3] A. Benoit, M. I. Cole, S. Gilmore, and J. Hillston. Evaluating the performance of skeleton-based high level parallel programs. In *The International Conference on Computational Science (ICCS 2004), Part III*, LNCS 3038.

[4] J. Berthold and R. 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.

[5] G. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

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

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

[8] W. S. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368): 829–836, 1979. ISSN 01621459.

[9] W. S. Cleveland and S. 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.

[10] M. I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

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

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

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

[14] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] A. H. Karp and H. P. Flatt. Measuring parallel processor performance. *Comm. ACM*, 33(5):539–543, 1990.

[21] S. Khirevich. Private communication, May 2010.

[22] S. Khirevich and A. Daneyko. Simulation of fluid flow and mass transport at extreme scale. In B. Mohr and W. Frings, editors, *Jülich Blue Gene/P Extreme Scaling Workshop 2010*. Jülich Supercomputing Centre, March 2010.

[23] S. Khirevich, A. Höltzel, S. Ehlert, A. Seidel-Morgenstern, and U. Tallarek. Large-scale simulation of flow and transport in reconstructed hplc-microchip packings. *Analytical Chemistry*, 81(12):4937–4945, 2009.

[24] S. Khirevich, A. Höltzel, A. Seidel-Morgenstern, and U. Tallarek. Time and length scales of eddy dispersion in chromatographic beds. *Analytical Chemistry*, 81(16):7057–7066, 2009.

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

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

[27] 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, 2002.

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

[29] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, July 1997.

[30] R. Peña and F. Rubio. Parallel functional programming at two levels of abstraction. In *PPDP '01: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 187–198. ACM, 2001.

[31] 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`.

[32] 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.

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

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

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