On Predicting the Impact of Resource Redistributions in Streaming Applications

Merijn Verstraaten
Heriot-Watt University
m.verstraaten@hw.ac.uk

Sven-Bodo Scholz
Heriot-Watt University
s.scholz@hw.ac.uk

Abstract

We propose a method for black box performance modelling of executions of data-parallel operations on shared memory multi-core systems. In particular, we predict runtimes of data-parallel operations from two inputs: a given input characteristics such as the size of the input and the numbers of cores that can be exclusively used for the task.

The paper describes the rationale as well as the technical details of the approach. We discuss several design choices of the technique and we experimentally explore their implications. We also discuss an online implementation of the proposed approach and we show that the model can be used very effectively in a streaming context.

1. Introduction

With the omnipresence of multi-core machines the ability to predict the runtime of programs or even parts of programs as a function of the amount of resources provided rapidly gains in importance. The better such predictions are the more effectively resources can be shared between competing parties. Whether we look at several applications that run on a shared infrastructure such as a cloud service or we look at a single application that tries to stream its data through existing compute resources, an effective mapping of resources is not possible without being able to predict the effect of additional resources.

Traditionally, runtime prediction builds on cost models and is tightly coupled to compiler infrastructures. However, this is problematic for several reasons: it requires all parts of an application to go through the same compiler, ruling out the use of library-based legacy code; it requires the entire compiler tool chain to be cost aware, including high-level as well as low level code transformations; it requires a suitable precise cost model of the hardware being used; the list goes on.

In this paper, we try to take a radically different approach. It is based on the idea of using a streaming based approaches to coordinate parallel programs. In such a setting, we can use runtime observations of previous stream-processing operations to predict the behaviour of future invocations of the same operations. This allows for a black-box performance model that does not require intimate interaction with the compiler tool-chain. Instead, a rather minimalistic interaction with the runtime system suffices. In previous work (Sykora and Scholz 2013), we have shown that this can be done effectively whenever the individual operations have input independent execution times. In this paper, we extend our previous model by taking input dependent runtimes into account. The only restriction we still impose is that the sequential runtime can be predicted as a polynomial of some predictor value and that such a predictor value can be computed from the input.

Our paper is structured as follows: Section 2 provides some of the background this work is based on. Section 3 provides our runtime model and some experimental validation for it. Section 4 presents related work before Section 5 concludes.

2. Modelling Data-Parallel Black Box Components

In the introduction we mentioned the desirability of being able to keep using existing legacy code. In our research we have been investigating the possibilities of using a coordination language to construct programs by combining existing components together. Within this coordination framework we then try to apply statistical tools and methods to the problem of optimising concurrent and parallel programs. This with the goal to create and environment where it is possible to reuse legacy code without all the painstaking manual optimisation.

2.1 Coordination

Our research takes place within the context of the declarative coordination language S-NET (Grelck et al. 2008, 2010). S-NET lets programmers specify programs by composing black-box computational components together into a streaming network.

This approach exposes the data dependencies of the computations and provides a framework in which the behaviour of the various computational components can be analysed independently.

There are no technical restrictions on which language is used to implement these components, and, as mentioned in the introduction, the ability to (re)use existing components is one of the driving motivations behind the design of the S-NET language and the work discussed in this paper.

Most of our research, including that discussed in this paper, has been centered around components implemented in (a subset of) ANSI C and SAC (Single Assignment C) (Grelck and bodo Scholz; Grelck et al. 2007), a purely functional array programming language, well-suited to exposing data parallelism in matrix and vector computations. However, we believe that there is no fundamental reason why the results could not be generalised to include other languages and approaches.

2.2 Data-Parallelism

Using a data parallel language like SAC to implement an S-NET component results in the exposure of an additional level of concurrency, the internal data parallelism of that component. This is important, because it means that the benefit of allocating additional computational resources to a component depends on the amount of parallelism that component exposes.

In other words, effectively scheduling an S-NET network that includes data parallel components on parallel hardware requires that the scheduler can predict the benefit of allocating additional resources to a component.

As mentioned in the introduction, we want to approach this in an implementation agnostic way, as this means our approach will be applicable to all components, regardless of implementation. This would allow the use of existing legacy components without losing out on more advanced scheduling functionality.

In Sykora and Scholz (2013) it was shown that a naive model that only considers the data parallelism of components can already result significant latency and throughput improvements. In this paper we will generalise the simple Amdahl model from that paper to one that also accounts for variations in workload sizes, allowing for more accurate predictions in environments where workloads change over time.

2.3 The Simple Amdahl Model

Since the desire is to have an implementation agnostic model, we need to consider which information can be obtained from data parallel components without requiring implementation specific details. Obvious candidates are the observed runtime, the input to the component and the number of processing units it is executing on.

Amdahl's law states that the speedup of an algorithm is dependent on the fraction of the algorithm that can be parallelized. If, for simplicity's sake, we restrict ourself to components whose parallelism is independent of their input, we have the following simple model. Given a sequential runtime T_{seq} and a parallel fraction α, the parallel runtime on p processing units is given by:

$$T_p = \frac{\alpha T_{seq}}{p} + (1 - \alpha) T_{seq}$$

The main idea from Sykora and Scholz (2013) is the following. If we have a network of components with statically known sequential runtimes, we can approximate the parallelism α of each component by varying the number of processing units they are executing on, measuring this parallel runtime and solving the above formula for α and T_{seq}.

We can then use the obtained αs and sequential runtimes to make determine how different resource allocations will affect throughput and latency.

This approach meets our requirement of being implementation agnostic. The only information necessary are the measured runtimes and the number of processing units used, both should be readily available for most environments/languages. Despite it's remarkable simplicity Sykora and Scholz (2013) shows that this model lets us obtain significant improvements in latency and throughput of a network.

The utility of the model is rather limited, as it breaks down in the presence of varying sequential runtimes. Most interesting algorithms and components have sequential runtimes that are dependent on their input, causing the runtime to vary when the inputs vary. To handle these scenarios we need to extend the model to account for the variation in runtime.

3. A New Extended Amdahl Model

The simple Amdahl model discussed in the previous section considers the observed runtimes and the number of processing units used, however, it ignores the input values. If we want to model components whose runtime depends on their input, we cannot ignore this source of information.

In many cases the programmers/user of a component has some cost model describing how the runtime of a component relates to the inputs of that component. For example, the runtime of adding two vectors depends linearly on the length of those vectors.

In all likelihood only some characteristics of the input will have an impact on the runtime of a component, such as the size of a vector or matrix. We introduce the term *predictor value* to refer to that part of the information in an input value that determines the component's runtime.

If the programmer/users of a component were to give us a function that maps input values to a suitable predictor value, we can then use this predictor value in our model to describe input dependent effects on the runtime by modelling the sequential runtime as a function of a predictor value.

3.1 The Model

If we model the sequential runtime as a function $T_{seq}(x)$ of some predictor value x, we can generalise our earlier Amdahl model of the parallel runtime to be a function of the same predictor x:

$$T_p(x) = \frac{\alpha T_{seq}(x)}{p} + (1 - \alpha) T_{seq}(x)$$

This model requires the following implementation agnostic parameters:

1. A function mapping inputs to predictor values.
2. A function modelling the sequential runtime.
3. The parallelisable fraction of the algorithm.

Having programmers explicitly specify a function modelling the sequential runtime T_{seq} or providing the value for the parallel fraction α is undesirable, as obtaining accurate estimates of these from a component is likely to be infeasibly complex. We do assume that providing a function that maps inputs to predictor values is feasible, as programmers and users should, in most cases, be aware which information in the input values impacts the runtime.

To obtain an estimate of T_{seq} and α we restrict ourselves to algorithms whose complexity can be described by a polynomial of known degree k over the provided predictor value. We can then use this information to estimate the values of T_{seq} and α based on the observed behaviour of the component, our knowledge of k and the predictor values. This should keep the approach general enough to work for any component, regardless of its implementation.

Our restriction to algorithms with a polynomial complexity of known degree keeps the model simple enough to remain tractable, as polynomial regression is a well-known and understood problem. At the same time, the restriction is sufficiently general that many interesting algorithms can still be modelled.

3.2 Sanity-checking the Model

Let us start by reviewing the assumptions we have made so far:

1. Parallelism is independent of input.
2. Sequential runtime is dependent on predictor value of input.
3. Runtime is described by a polynomial of known degree of a predictor.

Our model should describe any component meeting these criteria. As an initial validation of the model we see how well we

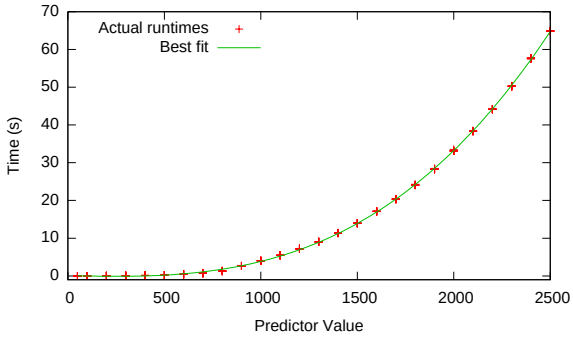

Figure 1. Approximation of sequential runtime.

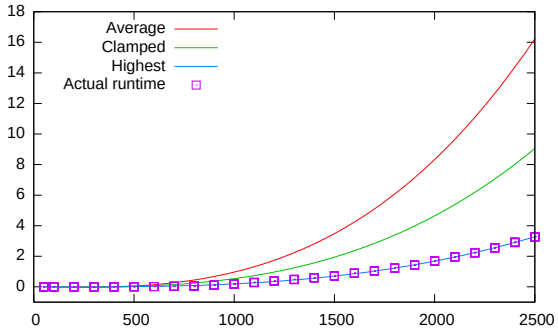

Figure 4. Approximation of parallel runtime on 47 cores.

can model a naive implementation of matrix multiplication. We use a simple S-NET network consisting of two components. The first components allocates an $N \times N$ matrix of random elements and sends this matrix to the component we are modelling. This second component is a simple SAC implementation of matrix multiplication that multiplies the input matrix with itself.

There are several reasons for choosing matrix multiplication. Firstly, the algorithm is simple to implement and its behaviour is well understood. Secondly, its complexity is $O(n^3)$ where n relates to the number of rows/columns. That way, we can use the array size as predictor value, which gives us a runtime described by a polynomial of degree 3. Lastly, due to the memory intensiveness of the algorithm, it is very sensitive to the memory hierarchy it operates on, making it a worst case scenario from a modelling perspective.

All experiments were performed on a 48-core system of four 12-core AMD Opteron(tm) 6172 2.1 GHz processors and a total of 128 GB of memory. We tried a large range of different predictor values, each running 10 times for each allocation of cores to the SAC component. The core allocations measured are 1, 2, 4, 8, 16, 32, 40 and 47[1] cores.

The first step in validating our model is to obtain an estimate of T_{seq} and to see how well it matches the actual sequential runtime of our algorithm. As mentioned above, the complexity of matrix multiplication tells us that we are looking for a polynomial of degree three. We also know that our predictor value is the length of one side of each matrix.

Armed with this information we can use least squares polynomial regression to estimate T_{seq} from a set of predictor values and observed runtimes. Our observation dataset covers a range of predictor values from 50 to 2500. Figure 1 shows a plot of both our estimate for T_{seq}, the green line, and the actual observed runtimes, the red crosses. As shown by the graph, our estimated T_{seq} matches the observed runtimes almost perfectly.

An accurate approximation of T_{seq} is sufficient to help us approximate the value of α. Since we know the number of processing units used and the observed runtime, finding α for a specific datapoint is a matter of solving:

$$T_p(x) = \frac{\alpha T_{seq}(x)}{p} + (1-\alpha)T_{seq}(x)$$

Here p is the number of cores, x the predictor value and $T_p(x)$ the observed runtime, resulting in:

$$\alpha = \frac{p}{1-p} * \frac{T_p(x)}{T_{seq}(x)}$$

[1] The use of 47 instead of 48 cores is to avoid any unpredictable interaction between the S-NET runtime system and our data parallel components.

Solving α for all predictor values and all numbers of cores produces the graph shown in Figure 2. The graph shows that there is a significant amount of noise in the values found for predictors below 500, with a particularly excessive peak at 400. Aside from this, the values of α appear to quickly converge on a single value.

The scale of Figure 2 makes it rather hard to see whether the αs do indeed converge to a small range of values. Figure 3 contains a magnification of the plot, where we see that the αs quickly converge on a value close to 1.

One explanation for the inaccuracy and noise for the sub 500 measurements is that the sequential runtime at 400 is $\tilde{6}0$ ms, while our measurement error is approximately 5 ms. This noise represents a significant fraction of the sequential runtime and an even bigger fraction of parallel runtimes. This results in approximations of α that can be off by significant amounts at the low end of the spectrum.

While the above shows that α quickly converges, there are two questions still unanswered. One, how should we aggregate the various differing estimates of α into a single value for use in our model? Two, how accurate are the predictions that we make based on our aggregated α value?

It may seem a natural choice to simply average all α values. Unfortunately, the significant errors in the α estimates for small runtimes result in very big errors once we try to predict runtimes for larger numbers of cores.

Since it doesn't make sense to have parallelizable fraction bigger than 100% or smaller than 0% we could clamp our αs between 0 and 1 and then take the average. The graph in Figure 4 plots the observed runtimes on 47 cores, together with our runtime predictions. As the red and green line show, both the overall average and the average of clamped αs produce results that are off by a considerable margin.

We can conclude that averaging αs is not a good enough aggregation method for estimating α. If we look back at Figure 2 and Figure 3 we see that most of the noise is happening for small predictor values. This makes sense when we look at our extended Amdahl model. Since α is multiplied with T_{seq} any error in α will scale linearly with the sequential runtime; in this example that means it scales cubically with our predictor value. At the same time, the effect of wrong estimates for the sequential fraction of our algorithm becomes more pronounced as the number of processing units increases.

Based on these observations, we decided to only consider α values from datapoints at the highest number of cores and with the highest predictor value and average these. The blue line in Figure 4 shows the results of using the α obtained this way. As the graph shows, this method of estimating α is significantly more accurate than the other approaches.

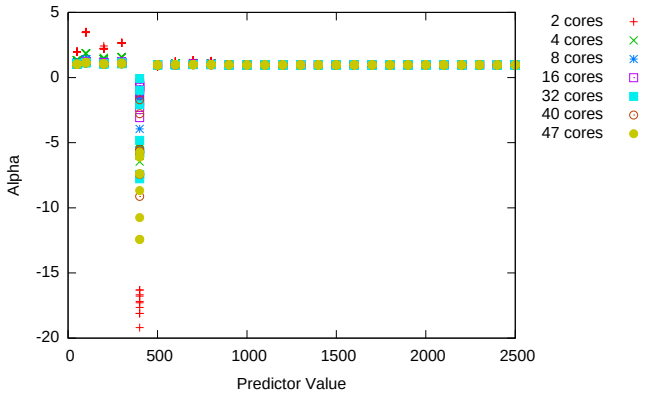

Figure 2. Values of α.

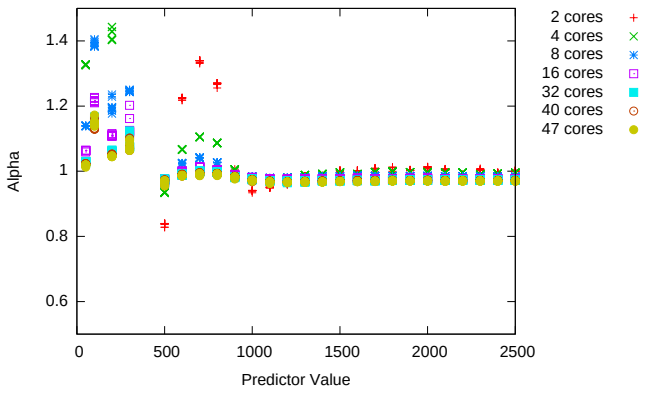

Figure 3. Values of α closeup.

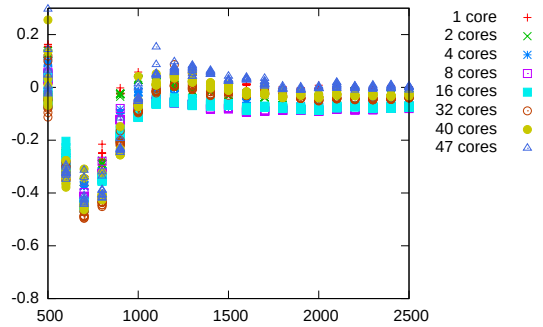

Figure 5. Mispredictions in percent of predicted time.

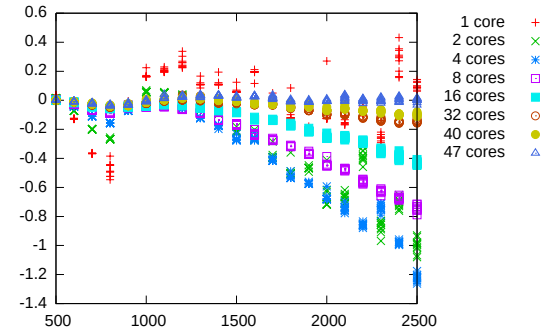

Figure 6. Misprediction in absolute time.

Figure 5 and Figure 6 illustrate the predictions errors for different predictor value and core combinations in percent of predicted time and absolute time, respectively. For predictor values below 500 we see that there is simply too much noise in the results to obtain very useful predictions. However, as we move to higher predictions we see that our mispredictions quickly drop from about 50% to within 10% of the observed runtime.

A 50% misprediction sounds absurdly large, but given the small overall runtimes at the low end of the graph it ends up at a misprediction of less than 200 milliseconds, as shown in Figure 6. For the worst prediction, 4 cores with a predictor value of 2500, the error amounts to only slightly more than 1 second on wallclock runtime of well over 15 seconds.

We believe that a prediction accuracy to within 10% will prove to sufficiently accurate to be useful, especially since the generality of the model lets us use it, regardless of the implementation details of a component.

3.3 Pushing the Limits

The naive version of matrix multiply that we used so far already puts a lot of pressure on the memory subsystem. To further increase potential effects of the cache hierarchy when looking at different problem sizes, we also investigate the effect of a more sophisticated implementation of matrix multiply which enforces blocking. As blocking vastly improves the temporal locality, we expect to see much more pronounced cache effects. We look at three different variations, using block sizes of 16×16, 32×32 and 64×64 elements.

These blocked implementations are much more prone to various cache effects, introducing an additional source of noise and variation in the computation. We are interested in seeing how badly these effects end up influencing our prediction accuracy.

Because the optimisation decreases the runtime by a significant amount we expect more noise for smaller predictor values and increased the maximum predictor value to produce a non-trivial amount of work for the 47 cores to do.

We can use the same evaluation as in subsection 3.2 to validate our model's accuracy. In Figure 7 and Figure 8 we see the measured and predicted runtimes on 1 and 47 cores, respectively. We use the same aggregation method for α as explained in the last section.

We observe a number of jumps in the actual runtimes, we have managed to trace some of these back to cache thresholds being exceeded. Other jumps we have not yet managed to explain adequately, but seem to stem from more complicated cache interactions.

The more interesting aspect to observe is that, even in the presence of complicated cache effects, our predicted runtimes manage to approximate the observed runtimes with surprising accuracy.

In Figure 9 and Figure 10 we revisit the misprediction graphs, the former showing mispredictions in percent of the predicted time, the latter showing mispredictions in seconds.

While the spread of mispredictions in Figure 9 is bigger than for our naive version, the vast majority of errors is still within 20% or 10%. And due to the shorter runtimes in the optimised version this again translates sub-second errors, as illustrated by Figure 10.

There are some large peaks Figure 10, but upon closer inspection these are not as dramatic as they seem at first. With a single core sequential runtime of 450 seconds, a misprediction of 25 seconds boils down to a prediction error of $\tilde{5}\%$. We consider this accurate enough to be useful for our scheduling purposes.

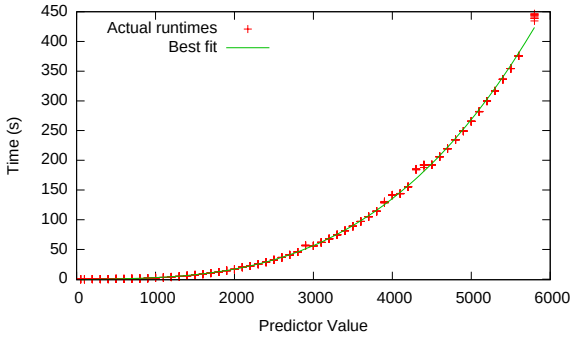

Figure 7. Approximation of sequential runtime.

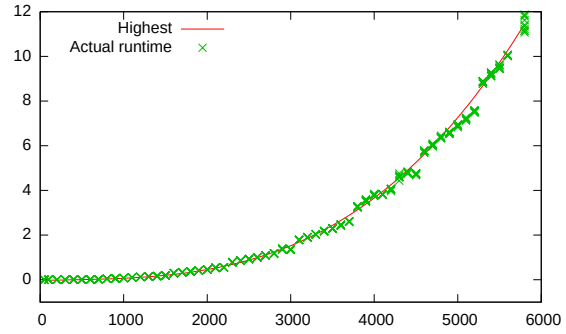

Figure 8. Approximation of parallel runtime.

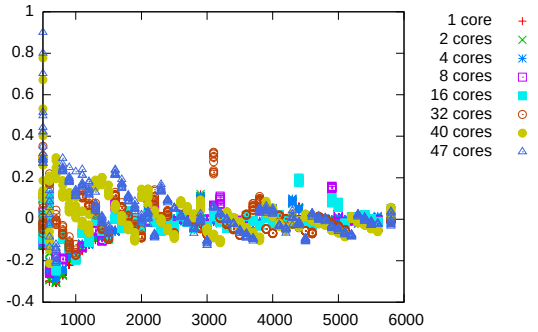

Figure 9. Misprediction in percent of predicted time.

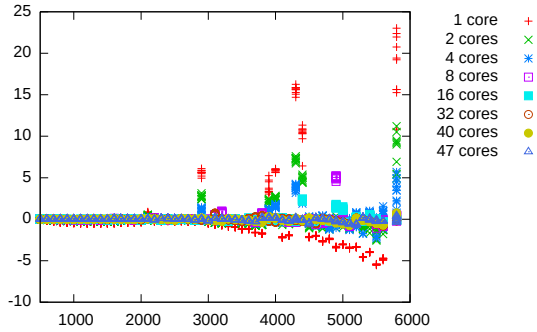

Figure 10. Misprediction in absolute time.

3.4 Online Model Approximation

The main purpose of this work is to use it as a basis for resource scheduling in a streaming context. In that context, any attempt to try all possible combinations of predictor values and numbers of processing units for each component quickly becomes prohibitively expensive. Instead, we look for an online technique to continuously improve our predictor as computation results become available.

The main challenge here lies in the fact that we need to approximate the polynom for the sequential runtime and the α value from the same measurements and that we do not want to apply our least squares fitting for incresing amounts of data points.

Assuming that we have a reasonably good approximation for the polynom, approximating α incrementally can be achieved through a running average. The only thing to consider is that our running average should only consider datapoints that match the current "highest" datapoint when it comes to predictor value and number of cores. Additionally, when we encounter a datapoint that is higher than the current maximum, this running average needs to be reset to this. Currently we prefer higher core numbers over higher workloads, but whether this approach is something we still have to investigate.

It is possible to solve the least squares polynomial regression for T_{seq} in constant space too. We note that for an arbitrary polynomial of degree k:

$$y = a_k x^k + \cdots + a_2 x^2 + a_1 x + a_0$$

we can compute the least squares polynomial regression using:

$$\vec{a} = (X^T X)^{-1} X^T \vec{y}$$

where, assuming n data points with $n \geq k+1$:

$$\vec{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} \quad \vec{y} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}$$

$$X = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^k \\ 1 & x_2 & x_2^2 & \cdots & x_2^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_2^n & \cdots & x_n^k \end{pmatrix}$$

To see how we can solve $\vec{a}$ in constant space, we observe that:

$$X^T X = \begin{pmatrix} \sum 1 & \sum x & \cdots & \sum x^k \\ \sum x & \sum x^2 & \cdots & \sum x^{k+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum x^k & \sum x^{k+1} & \cdots & \sum x^{2k} \end{pmatrix}$$

$$X^T \vec{y} = \begin{pmatrix} \sum y \\ \sum xy \\ \vdots \\ \sum x^k y \end{pmatrix}$$

Both $X^T X$ and $X^T \vec{y}$ only contain sums over all x and y values, meaning it's possible to incrementally update these matrices as data points come in. As we saw in the earlier formula $\vec{a}$ can be solved by inverting $X^T X$ and then multiplying with $X^T \vec{y}$. This approach results in constant space usage to approximate T_{seq}, since both matrices have a constant size determined by the order k of the polynomial being approximated. This means we only need to store $k(k+1)$ values to solve T_{seq}.

3.5 Evaluation of Online Model Approximation

To test the methods described in the previous section we take the datapoints from experiments in the earlier sections, shuffle them into a random order and treat them as incoming datapoints. For every incoming point we compare the our predicted time with the actual observed time and then update the model.

Our expectation was a graph that starts out with a big range of mispredictions and then narrows as our approximation becomes better. However, what we find is that the model converges so quickly that we barely see any narrowing of the misprediction range, it stays almost constant from the beginning.

To verify that this streaming approach is actually an improvement over simply taking the first possible estimate of T_{seq} we rerun our experiment, this time without updating our approximation. This shows that the prediction errors of the incremental approach are clustered significantly tighter.

In Figure 11 and Figure 12 we show the difference in mispredictions as percent of the predicted time. The former shows the results of stopping at the first approximation of T_{seq}, the latter shows the results of incrementally adjusting our model. We see that the incremental approach has a smaller average error and reduces the number and the range of outliers.

Figure 13 and Figure 14 show the same comparison, but this time with the misprediction plotted as an absolute time. The incremental approach reduces outliers from close to 40 seconds to within 3 seconds. Where the 3 seconds is just a single datapoint, all others falling within 1 second.

The marked improvement in prediction accuracy while using an incremental approach leads us to believe that this is the right approach, especially considering the very small number resource consumption of the approach.

4. Conclusion

In this paper we present the extended Amdahl model, an implementation agnostic model for predicting the effect of resource allocations for data parallel algorithms and components. Our work is motivated by the need for making informed resource allocations in the presence of multiple data parallel components.

The extended model is a generalisation of previous work in Sykora and Scholz (2013). The simple Amdahl model discussed in that paper shows that significant throughput and latency improvements can be achieved by simply observing the runtimes and parallel behaviour of components with constant sequential runtimes.

The main contribution of this work is the generalisation from constant sequential runtimes to a model that can handle sequential runtimes that are dependent on the inputs of a component.

We demonstrate that, in the presence of a well-chosen predictor value and polynomial of known order, using least squares polynomial regression is sufficiently accurate to predict the sequential runtime of a component. In addition, we show that using this approximation of the sequential runtime we can find an approximation of the algorithm's parallel fraction enabling us to predict the parallel runtimes for inputs with an accuracy of within 10% of the actual runtime.

Another contribution of this paper is the insight that it is both, worthwhile and feasible, to construct and update the extended Amdahl model on the fly. This approach eliminates the need to benchmark or otherwise set up the model in advance, lending itself to an auto-tuning like implementation.

We believe that this model is sufficiently accurate for making informed resource allocation decisions. The introduction of predictor values and the abstract notions of complexity on these opens up the applicability of this approach to a wide range of algorithms, far beyond what could be achieved with our previous Amsahl-based model. At the same time the model is sufficiently general that it should work for almost all kinds of data-parallel components regardless of the concrete implementation language or runtime system used, far beyond our particular streaming setting in the context of S-Net and SaC.

Acknowledgments

This work has been supported from EU-funded FP-7 project "Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)" under contract no. IST-248828.

References

C. Grelck and S. bodo Scholz. Sac: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, page 2006.

C. Grelck, S.-B. Scholz, and A. Shafarenko. Coordinating Data Parallel SAC Programs with S-Net. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, USA*. IEEE Computer Society Press, 2007.

C. Grelck, S.-B. Scholz, and A. Shafarenko. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters*, 18(2):221–237, 2008.

C. Grelck, A. S. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, S. Scholz, and A. Shafarenko. S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2010. URL `http://www.snet-home.org/?page_id=7`.

J. Sykora and S.-B. Scholz. Towards self-adaptive concurrent software guided by on-line performance modelling. In *Proc. 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures*, 2013.

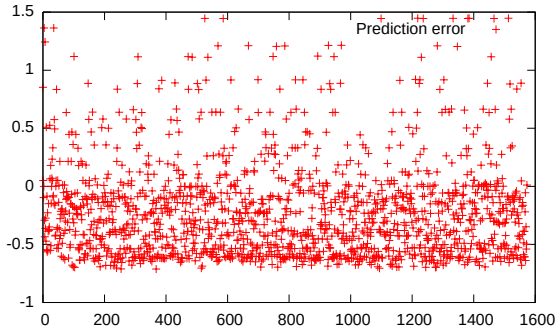

Figure 11. Static model: Mispredictions over time, in percent of predicted time.

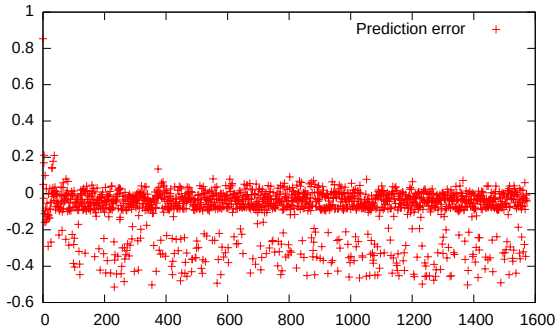

Figure 12. Incremental model: Mispredictions over time, in percent of predicted time.

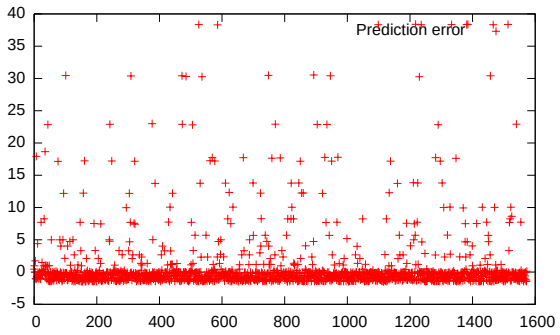

Figure 13. Static model: Mispredictions over time, in absolute time.

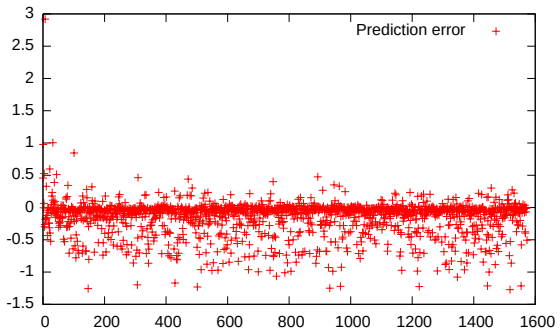

Figure 14. Incremental model: Mispredictions over time, in absolute time.