Optimization Space Exploration of the FastFlow Parallel Skeleton Framework

Alexander Collins Christian Fensch Hugh Leather
University of Edinburgh
a.collins@ed.ac.uk c.fensch@ed.ac.uk hleather@inf.ed.ac.uk

Abstract

Parallel skeletons are a structured parallel programming abstraction that provide programmers with a predefined set of algorithmic templates that can be combined, nested and parametrized with sequential code to produce complex programs. The implementation of these skeletons is currently a manual process, requiring human expertise to choose suitable implementation parameters that provide good performance. This paper presents an empirical exploration of the optimization space of the FastFlow parallel skeleton framework. We performed this using a Monte Carlo search of a random subset of the space, for a representative set of platforms and programs. The results show that the space is program and platform dependent, non-linear, and that automatic search achieves a significant average speedup in program execution time of $1.6\times$ over a human expert. An exploratory data analysis of the results shows a linear dependence between two of the parameters, and that another two parameters have little effect on performance. These properties are then used to reduce the size of the space by a factor of 6, reducing the cost of the search. This provides a starting point for automatically optimizing parallel skeleton programs without the need for human expertise, and with a large improvement in execution time compared to that achievable using human expert tuning.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Experimentation, Performance

Keywords Algorithmic skeletons, Multicore, Performance measurement, Optimization space exploration

1. Introduction

Over the past few decades, relentless improvements in processor performance have come from increasing processor frequency and complexity. This progress has slowed as processor designs come up against physical limits, such as our limited capacity to remove heat [8]. The use of multiple, simpler processing elements, or cores, has now become a focus for industry and research, as it is a promising avenue to continue improving processor performance. By providing multiple cores, separate parts of a program can be executed in parallel. However, performance relies heavily on the ability of the program to fully utilize these cores. This requirement adds additional complexity at the software level, and has led to the design of parallel abstractions that aim to hide this from the programmer without introducing significant or unpredictable overheads.

Parallel skeletons, also known as algorithmic skeletons, are one such abstraction [13, 14]. They are a structured approach to parallel programming designed to preserve performance, be platform independent and provide an abstraction over low-level issues. The idea is to provide a collection of templates, or skeletons, each of which implements a standard algorithmic technique. They allow the programmer to produce complex parallel programs by composing and nesting these skeletons together, and parametrizing them with sequential, application specific code fragments. A skeleton library provides an implementation for each skeleton, which are used to compile the program into a form that can be executed. Commonly provided skeletons include task farms, pipelines, divide and conquer, and data-parallel map and reduce [13, 18].

The performance of a parallel skeleton program relies on three things: an appropriate choice of skeletons, implementing the sequential code fragments efficiently, and that the library provides an efficient implementation for each skeleton [2, 11]. This paper explores the performance increase available by optimizing the latter of these factors.

To do this, we empirically explore the optimization space of the FastFlow parallel skeleton framework [3, 5]. We identified six implementation parameters and a finite range of values for each. Program execution time for ten programs on five platforms is then measured using a Monte Carlo search of a random subset of the optimization space. The best program execution time found is then compared to that achieved using human expert chosen parameters. Our results show an average speedup of $1.6\times$ over a human expert, that the space is non-linear, and platform and program dependent. This demonstrates why it is difficult for a human expert to manually tune these programs. We also perform an exploratory data analysis to find properties of the optimization space that can be exploited by automatic optimization techniques. These results provide a starting point for automatically optimizing parallel skeleton programs without the need for human expertise, and future work will develop static heuristics to optimize programs with minimal additional compile time cost.

The rest of the paper is structured as follows. Section 2 provides motivation for this research. Section 3 describes our approach to explore the optimization space of parallel skeleton programs. Section 4 describes the FastFlow skeleton framework library developed by Aldinucci et al. [3, 5]. Section 5 explains the methodology behind the Monte Carlo search of the optimization space, the results of which are analyzed in Section 6. Section 7 follows with a discussion of related work, and Section 8 summarizes the conclusions and discusses future work.

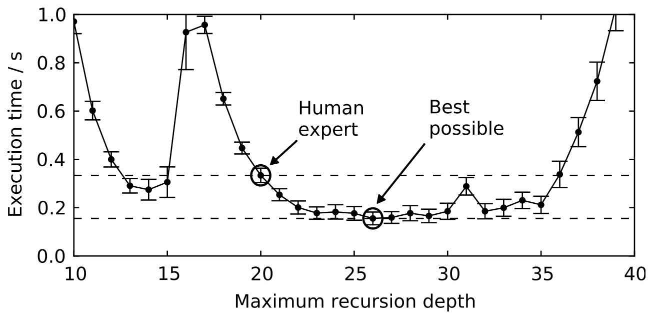

Figure 1. Comparison of the maximum recursion depth chosen by a human expert and the best possible choice for a program computing the 40th Fibonacci number. This shows that a significant speedup of $2.2\times$ over a human expert is available. The error bars show 99% confidence intervals for the mean.

2. Motivation

Figure 1 shows the one-dimensional optimization space for a program that calculates the 40th number in the Fibonacci series, using a naïve recursive algorithm without memoization. The optimization space consists of a single parameter: the maximum recursion depth of the divide and conquer skeleton used. The experiment was performed on a 32-core machine with $4\times$ Intel Xeon L7555 processors, called `xxxii`, using the methodology described in Section 5.4.

This example demonstrates that the optimization space of a skeleton parallel program can be complex and non-linear, exhibiting multiple optimal parameter values, even for an extremely simple optimization space consisting of a single parameter. A human expert choose a maximum recursion depth of 20, whereas a value of 26 achieves the best execution time. This is a significant speedup of $2.2\times$ over the human expert chosen parameter. This shows that there is large scope for improving program execution time over a human expert, and suggests that the optimization space for multiple parameters will be extremely complex, making it very difficult for a human expert to tune programs.

This motivates the use of automatic optimization techniques to optimize parallel skeleton programs. As a first step along the road to developing these, we present the results of an empirical exploration of the optimization space of the FastFlow parallel skeleton framework followed by an exploratory analysis to look for interesting characteristics of the space. In future work, these characteristics will be combined with the static information about communication and computation provided by the choice of skeleton to either reduce the cost of automatic search or design static heuristics with minimal compile time cost.

3. Optimization Space Exploration

To explore the optimization space of FastFlow we identified a set of implementation parameters that may have an effect on program execution time. For each parameter we chose a finite range of values. These parameters form the optimization space explored in the experiment.

We then collect a suite of representative programs and platforms, and perform a Monte Carlo search of a subset of the optimization space for each program-platform combination. On each step of the search, a uniformly random point in the space that has not been seen before is chosen, and the program compiled with the chosen parameters. Program execution time is then measured by executing the program. This results in a mean program execution time measured for a representative subset of the space, for each program-platform combination.

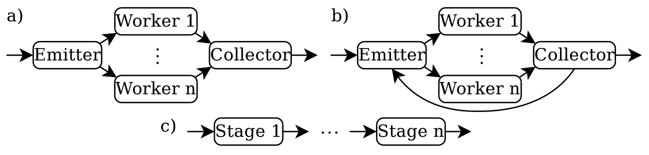

Figure 2. The construction of FastFlow's skeletons; showing a) `farm`, b) `farm-with-feedback` and c) `pipe`. Arrows represent SPSC queues and boxes represent threads.

The program execution time using human expert chosen parameters is also measured for every program-platform combination. This allows us to compare the execution time of expert tuned programs against that found by an automatic search of the space.

Measuring the performance of a representative subset of the optimization space for each program also allows us to perform statistical analysis of the space, to find characteristics of the space that can be used to reduce the cost of the automated search. We apply several statistical techniques to evaluate the importance of each parameter on the optimisation space.

4. FastFlow

FastFlow [3–7] is a parallel programming framework for shared memory multi-core systems. FastFlow was specifically designed to introduce minimal overhead and be scalable. Aldinucci et al. [4–7] demonstrate this by comparing the performance of a variety of programs implemented using FastFlow to carefully hand optimized versions written using lower-level parallel abstractions.

FastFlow therefore represents the state-of-the-art in terms of performance, and so is a good starting point when exploring the scope for performance improvement. It is implemented as a library in C++ and consists of six layers, each of which provides an incremental abstraction over the previous layer. These are as follows:

1. *Hardware*. FastFlow is designed for multi-core shared memory architectures.
2. *Run-time support* provides a general purpose threading model built on top of POSIX threads, a custom memory allocator and two lock-free single-producer single-consumer (SPSC) queue implementations [19] which threads use to communicate. The two types are:
   - *Bounded queues*, which impose a fixed upper limit on the number of items they can buffer. If a producer attempts to push an item onto a full queue they will block until a consumer consumes an item.
   - *Unbounded queues*, which do not impose a limit on the number of items. They can grow to accommodate as many items as the memory system allows.

   The choice to use bounded or unbounded queues depends on the required semantics. A bounded queue with a buffer size of one can be used to implement a synchronization primitive between two threads, and in queuing networks that contain a cycle unbounded queues prevent the possibility of deadlock. The absence of locks in these queues means that the synchronization overhead is low. This is one of the main factors that contributes to FastFlow's performance and scalability. [4, 6]
3. *Low-level programming* provides non-blocking, lock-free synchronization mechanisms built using the SPSC queues provided by the previous layer including multiple-producer multiple-consumer queues.
4. *High-level programming* provides three parallel skeletons: `farm`, `farm-with-feedback` and `pipe` implemented using

Platform	Processor	# Cores	Frequency	Main Memory	L3 Cache	L2 Cache
xxxii	4× Intel Xeon L7555	32	1.87GHz	64GB	4× 24MB	32× 256KB
xxxii16	2× Intel Xeon L7555	16	1.87GHz	64GB	2× 24MB	16× 256KB
scuttle	AMD Phenom II X6 1055T (95W)	6	3.3GHz	8GB	1× 6MB	1× 512KB
phantom	Intel Xeon E5430	4	2.67GHz	8GB	None	2× 6MB
desktop	Intel Core 2 Duo E6400	2	2.13GHz	3GB	None	1× 2MB

Table 1. Platforms used in the experiments

threads and SPSC queues as shown in Figure 2. They are arbitrarily nestable and are parametrized with sequential C++ code. The `farm` skeleton implements a bag of tasks pattern, which processes a set of independent tasks using multiple worker threads. The `farm-with-feedback` skeleton extends the `farm` skeleton by adding a feedback path. This is equivalent to a divide and conquer skeleton, as the emitter can schedule separate divide and conquer tasks to the workers. The `pipe` skeleton implements a pipeline pattern with a fixed number of stages. This is constructed from a fixed number of threads, one for each stage, connected by SPSC queues.

5. *Problem solving environments* (PSEs) are components that can be used to exploit parallelism and are built using skeletons from the previous layer.
6. *Applications* are implemented at the highest layer, and can make use of any of the tools provide by the *low-level programming*, *high-level programming* and *PSE* layers.

5. Experimental Setup

This section describes the experiment performed to explore FastFlow's optimization space. First, we identified a set of six implementation parameters affecting the performance of FastFlow programs. A Monte Carlo search then measures program execution time for a subset of the optimization space formed by these parameters.

Section 5.1 details the platforms and programs used and Section 5.2 explains the parameters that make up the optimization space. Section 5.3 then explains details of the Monte Carlo search. Section 5.4 describes the methodology used to measure program execution time.

5.1 Platforms and Programs

The experiment was carried out on five platforms, each with differing numbers of cores, internal architectures and memory hierarchies, so that the effect of the hardware on the optimization space could be investigated. Table 1 shows the details of these platforms.

Many of these platforms utilize frequency scaling, where the frequency of the processor is scaled according to the workload of the system. The aim of this is to reduce power consumption, however it can introduce significant noise into performance measurements. To avoid this problem, frequency scaling was disabled and the platforms were configured to use their maximum frequency.

We used a suite of ten existing FastFlow programs in the experiment. Some of the programs are real world applications that have been modified to use FastFlow. This is the case for the `pbzip2` program, which uses a FastFlow `farm` to parallelize the compression of individual blocks of the input file using the bzip2 compression algorithm. Other programs are implementations of standard algorithms such as Quicksort (`quicksort`) and Adaptive Quadrature (`aquad`).

The programs use two of the skeletons provided by FastFlow: `farm` and `farm-with-feedback`. Unfortunately, programs using the `pipe` skeleton were not available and could not be implemented due to time constraints. Each of the programs was modified to accept run-time and compile-time arguments in a common format, to simplify running the experiment. This has negligible effect on performance. Sample input data was provided with the source code for each program. To avoid changing the behavior of each program on each platform the same input data was used throughout.

Parameter	Description
numworkers	Number of threads
buffertype	Type of queue: *bounded* or *unbounded*
buffersize	Buffer size of queues
batchsize	Number of items in a batch
cachealign	Memory allocation alignment of items
seqthresh	Maximum recursion depth for `farm-with-feedback`

Table 2. Parameters explored in the experiment

5.2 Optimization Space

The following describes the optimization parameters that were chosen and why some were not considered. The final set of six parameters explored in the experiment are summarized in Table 2.

5.2.1 Existing FastFlow Parameters

FastFlow's low-level mechanisms and skeleton implementations include several parameters that have been tuned by a human expert. These are either fixed in its implementation, or are chosen on a per-program basis. They include:

- *Number of workers.* This controls the number of threads used by the `farm` and `farm-with-feedback` skeletons.
- *Bounded/unbounded queues.* Bounded queues may be more efficient than unbounded queues, as they do not need to resize their buffer, however they may cause deadlock in programs with cyclic communication patterns.
- *Queue length.* In the case of bounded queues, this is the maximum number of items that can be buffered by the queue. If this limit is reached, producers for the queue will block when attempting to push items into it. In the case of unbounded queues, this controls the size of each chunk of memory allocated to buffer items.
- *Task scheduling.* Custom scheduling policies can be defined for both the `farm` and `farm-with-feedback` skeletons, which determine how tasks are distributed to worker threads.

The experiment considers the first three of these parameters. Task scheduling is not considered as FastFlow allows arbitrary scheduling methods to be defined and a representative set of schedulers could not be developed due to time constraints.

5.2.2 Additional FastFlow Parameters

FastFlow is still being actively developed, and some of its interfaces have not been fully implemented. For example it has no control

Parameter	Values
numworkers	1, . . . , # cores × 1.5
buffertype	Bounded or unbounded
buffersize	1, 2, 4, 8, . . . , 2^{20}
batchsize	1, 2, 4, 8, . . . , 2^{20}
cachealign	64, 128 or 256 bytes
seqthresh with aquad	0.02, 0.04, 0.06, . . . , 1
seqthresh with fibonacci	10, 11, 12, . . . , 44
seqthresh with nqueens	3, 4, 5, . . . , 15
seqthresh with quicksort	1, 2, 4, 8, . . . , 2^{21}

Table 3. Parameter values explored by the experiment

over task granularity or the recursion depth of divide and conquer algorithms implemented using the `farm-with-feedback` skeleton. Both of these concerns are left to the application programmer. Beyond the parameters already identified in FastFlow's implementation, there is scope for further optimization:

- *Task granularity.* Tasks sent between threads can be grouped into batches of a chosen size. This is likely to reduce communication/synchronisation overhead at the cost of reducing the available task-parallelism and adversely effecting load balancing.
- *Maximum recursion depth.* In the `farm-with-feedback` skeleton, the cut-off between performing parallel and sequential computation could be controlled.

In order to control task granularity without needing to modify the benchmark programs, an additional type of queue was implemented: the *batched queue*. These are built on top of either a bounded or unbounded queue. The producer thread accumulates items locally into a fixed sized *batch*. When the batch is full, they are pushed onto the underlying queue as a single item. The consumer thread pops this batch off of the queue and consumes its contents locally.

Maximum recursion depth applies to divide and conquer algorithms implemented using the `farm-with-feedback` skeleton. The emitter thread controls the point at which the sub-tasks are solved sequentially instead of being further sub-divided into smaller problems to be solved in parallel. The current implementation of this skeleton is quite low-level compared to divide and conquer skeletons provided by other libraries [18] as it does not allow the skeleton to be parametrized with code fragments specifically for the divide and conquer stages of the computation. It also provides no mechanism for specifying a maximum recursion depth. It is likely that a divide and conquer skeleton providing control over recursion depth will be included in future versions of FastFlow, therefore it is considered in this experiment.

5.3 Monte Carlo Search

A Monte Carlo search was performed across the entire optimization space to measure the execution time of each program on each platform for a random subset of the optimization space. As the optimization space is enormous, exhaustively measuring program execution time performance at every point in the space would take a prohibitively long time. It is only practical to measure program execution time for a small subset of the entire space.

Table 3 details the parameter values that constitute the optimization space. Program execution time was measured for a random subset of all permutations of these values. It is assumed that this subset is representative of the entire space, and this assumption is justified in Section 6.4.

5.4 Measuring Program Execution Time

Measurement of program execution time of a program on a given platform needs to be performed in a way such that the significance of any hypotheses can be evaluated. Georges et al. [17] present a statistically rigorous method for benchmarking Java programs, and many of their statistical and methodological techniques are applicable here. The following methodology was used to measure the execution time of each program on each platform for a given set of values for the optimization parameters:

- *Compilation.* Each of the benchmark programs are implemented in C++, and were compiled to native binaries on each platform using GCC version 4.5 with `-O3` optimization flags enabled.
- *Timing method.* Program execution times were measured to millisecond accuracy.
- *Maximum execution time.* A trial of the experiment revealed that for many parameter values, program execution time is prohibitively long. As the experiment is looking for optimal regions of the space, such points are terminated after reaching a fixed threshold and given an infinite execution time.
- *Sufficient repeats.* By performing repeated measurements the random error in the sample mean can be quantified using confidence intervals, which provide a range of values that include the true mean with a given probability. They can also be used as a stopping condition for the number of repeats: when the range of the confidence interval relative to the mean (the *coefficient of variation*) falls below a chosen threshold, stop the measurements. The stopping criteria used are:
  - Perform between 10 and 100 repeats.
  - Stop if the coefficient of variation drops below 1%, for a 99% confidence interval.

  These criteria are fairly arbitrary, however they are only required to provide a stopping condition that achieves sufficiently tight confidence intervals.
- *Outlier removal.* The arithmetic mean is used to provide a single point estimate of program execution time, however it is not a robust statistic. This means that outliers will have a large effect on its value and cause the range of any confidence intervals to increase dramatically. To shrink the confidence interval to an acceptable size many more repeats would be necessary, but performing these additional measurements is not practical. This motivates the removal of outliers. This is done using *interquartile range removal* [9], which calculates the 25% quartile Q_1 and 75% quartile Q_3, and discards points lying outside of the following range:

  $$\left[Q_1 - k(Q_3 - Q_1), Q_3 + k(Q_3 - Q_1)\right]$$

  A value of $k = 3$ is used in this experiment.

6. Results and Analysis

This section analyses the results of the Monte Carlo search and shows that the results support the hypothesis that there is scope for optimisation beyond human expert chosen parameters. An exploratory data analysis is also performed to look for properties of the optimisation space that can be used to reduce the size of the space. This analysis provides a starting point for developing automatic optimisation techniques and static heuristics to improve program performance over that achievable by a human expert with minimal compile time cost.

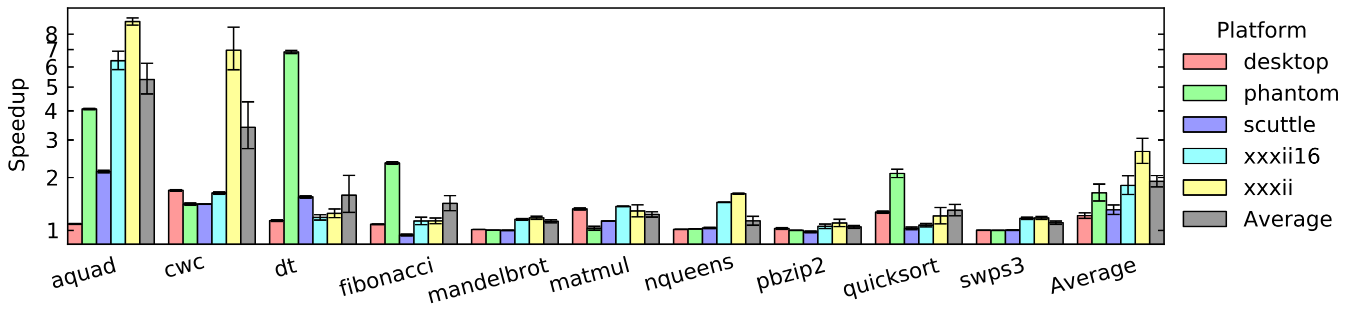

Figure 3. Speedup over human expert chosen parameters found by the Monte Carlo search

6.1 Speedup over human expert chosen parameters

Figure 3 shows the speedup of the best parameters found by the Monte Carlo search over the baseline execution time for human expert chosen parameters. This shows large performance improvement over a human expert is achievable, with an average speedup of $1.6\times$. Each bar shows the mean speedup of each program on each platform. The average speedup across all programs on each platform, across all platforms for each program, and across all platforms and programs are also shown. The error bars show confidence intervals with a significance level of $\frac{0.01}{5\times10}$ set by Bonferroni correction [1]. This allows multi-hypothesis testing with a significance level of 0.01 by direct comparison the confidence intervals.

Most of the programs exhibit a statistically significant speedup, and for some the speedup is large. The difference in average speedup between the Monte Carlo search and human expert chosen parameters is significant with $p < 1\%$ and a critical t value of 11.5. `aquad` achieves a speedup of over $2\times$ on all but `desktop`, with a $9\times$ speedup on `xxxii`. `fibonacci` and `pbzip2` perform worse than human expert chosen parameters on `scuttle`. This shows that the Monte Carlo search did not find the human chosen parameters in these instances. The average speedup for each platform is in the range 1.1–$2.8\times$. Interestingly, the speedup varies across both platforms and programs and the parameter values that provide the best performance differ across platforms and programs. For example, the speedup for `aquad` is $9\times$ on `xxxii` and $1.1\times$ on `desktop`. This suggests that the best choice of parameters is both platform and program dependent.

This shows that there is large scope for improving program execution time over a human expert, which is a surprising result given that human expertise usually performs better than compilers. Reasons for this may include the non-linearity, and program and platform dependent nature of the optimization space. This makes it very difficult for an expert to manually tune programs and achieve performance equivalent to that possible using automatic search.

6.2 Exploratory Analysis of the Optimization Space

This section analyses the space to look for properties that will be used to reduce the cost of automated search by reducing the size search space and, in future work, design static heuristics.

6.2.1 The effect of each parameter on execution time

Figure 4 shows that some parameters have a larger effect on program execution time than others, by plotting the average performance loss across all platforms and programs when each parameter is *not* carefully tuned. The chosen parameter was fixed and the others varied to find the best program execution time. The execution time for all fixed values of the parameter were then averaged, giving a measure of the average performance when the parameter is not tuned.

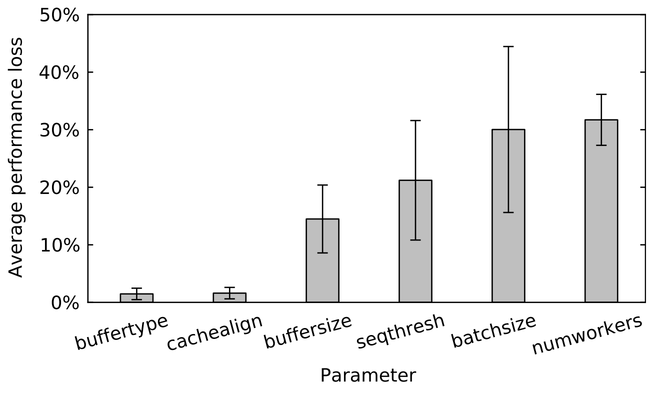

Figure 4. Average performance loss when each parameter is *not* carefully tuned. This shows it is important to tune the four parameters on the right of the graph, and not necessary to tune `buffertype` and `cachealign`. Error bars show 99% confidence intervals.

The results show that `numworkers`, `seqthresh`, `batchsize` and `buffersize` have a large impact on program execution time. This suggests that to minimize program execution time, it is important to tune these parameters. The error bars show that `numworkers` has a larger impact on program execution time than `batchsize`, suggesting that it is more important to carefully tune `numworkers`. The error bars for `batchsize` and `seqthresh` are large, and overlap with the error bars for the other parameters that have an impact on execution time. This suggests that the performance impact of `batchsize` and `seqthresh` varies across platforms and/or programs, therefore the importance of tuning these parameters is platform and/or program dependent. `buffertype` and `cachealign` have little impact on execution time, meaning that the cost of tuning these parameters may not be justified, given the small performance increase achievable. This information can be used to reduce the size of the search space explored by automatic search, by setting these parameters to fixed human expert chosen values and not tuning them.

6.2.2 Linear dimensionality reduction

The previous section identifies that `buffertype` and `cachealign` have little effect on execution time, allowing the optimization space to be reduced by ignoring them. Another technique for reducing the size of the optimization space is to exploit linear dependencies between parameters, using a linear dimensionality reduction technique called Principal Components Analysis (PCA) [10].

PCA finds orthogonal directions in an N-dimensional space that explain the majority of the variability in the data. Applying this to positions that provide near optimal performance (at $> 95\%$

$$N = 6 \qquad P = 5624$$

$$\boldsymbol{p} = \begin{pmatrix} \texttt{batchsize}, \texttt{buffersize}, \texttt{buffertype}, \\ \texttt{cachealign}, \texttt{numworkers}, \texttt{seqthresh} \end{pmatrix}$$

$$\boldsymbol{\lambda} = (0.443, 0.419, 0.204, 0.138, 0.007, 0.003)$$

$$\boldsymbol{e} = \begin{pmatrix} 0.023 & 0.814 & -0.000 & -0.003 & -0.580 & 0.013 \\ -0.015 & 0.581 & 0.002 & 0.002 & 0.814 & -0.014 \\ -0.115 & -0.001 & 0.005 & -0.000 & 0.015 & 0.993 \\ 0.993 & -0.010 & -0.000 & -0.001 & 0.028 & 0.115 \\ -0.001 & -0.001 & 0.003 & -1.000 & 0.003 & -0.001 \\ -0.001 & 0.001 & -1.000 & -0.003 & 0.002 & 0.005 \end{pmatrix}$$

$$\boldsymbol{\nu} = (36\%, 70\%, 87\%, 99\%, 99\%, 100\%)$$

Figure 5. Results of Principal Components Analysis for near optimal parameters (at $> 95\%$ performance) with number of positions P, parameters $\boldsymbol{p}$, eigenvalues $\boldsymbol{\lambda}$, corresponding eigenvectors in the rows of $\boldsymbol{e}$ and cumulative percentage of variability $\boldsymbol{\nu}$.

performance) we find that fewer dimensions are required to encode the majority of the variability in the near optimal positions.

For N-dimensional input data, PCA results in a set of N *eigenvectors* $\boldsymbol{e}_i$ (the rows of $\boldsymbol{e}$ in Figure 5) and corresponding *eigenvalues* λ_i. Each eigenvector represents a direction in the N-dimensional space and its corresponding eigenvalue is proportional to the variation of the data in this direction. The eigenvectors returned are sorted in order of decreasing eigenvalue. The percentage of the variation in the data ν_m explained by the first m eigenvectors can be calculated from their corresponding eigenvalues as follows:

$$\nu_m = \sum_{j=1}^{m} \lambda_j \Big/ \sum_{k=1}^{N} \lambda_k$$

Creating a new space using the first m eigenvectors returned by PCA allows ν_m percent of the variation in the optimal parameter choices to be expressed. If m is less than N then we have reduced this new space will be smaller.

An issue with PCA is that it is sensitive to scaling: if each dimension has different units of measurement, they need to be scaled. *Autoscaling* is one method of doing this. It divides the values along each dimension by the variance in that dimension, resulting in unit variance along each dimension. The units of measurement for `numworkers` also vary across platforms, taking values in the range $[1, 1.5 \times \text{number of cores}]$. This parameter is scaled to the range $[0, 1.5]$, representing the percentage of the cores used.

Figure 5 shows the results of the PCA. The directions of greatest variability are given by the eigenvectors, which are the rows $\boldsymbol{e}_i$ in matrix $\boldsymbol{e}$. Each column of this matrix corresponds to a parameter in p. Each eigenvector $\boldsymbol{e}_i$ has a corresponding eigenvalue λ_i which is proportional to the variance of the data in the direction of $\boldsymbol{e}_i$.

In order to explain 99% of the variability in the data, only the first four eigenvectors are required, as shown by the cumulative percentage of variability ν_i. This agrees with the analysis in Section 6.2.1, which also shows that `buffertype` and `cachealign` have little effect on execution time.

Eigenvectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ show a linear dependence between `numworkers` and `buffersize`. $\boldsymbol{e}_3$ and $\boldsymbol{e}_4$ are almost identical to the directions of `seqthresh` and `batchsize`, showing that these parameters have no linear dependence on any other parameters.

The implication of this is that the search space can be reduced from six dimensions to four (by removing `buffertype` and `cachealign`) whilst still representing 99% of the parameters that provide near optimal performance. The linear dependence between `numworkers` and `buffersize` can be used to reduce the size of the optimization space further, by combining them into a single parameter. Furthermore, the first two eigenvectors explain 70% of the variation in the near optimal parameters. Automatic search could be applied using only these two parameters (`buffersize` and `numworkers`) and achieve reasonable program performance.

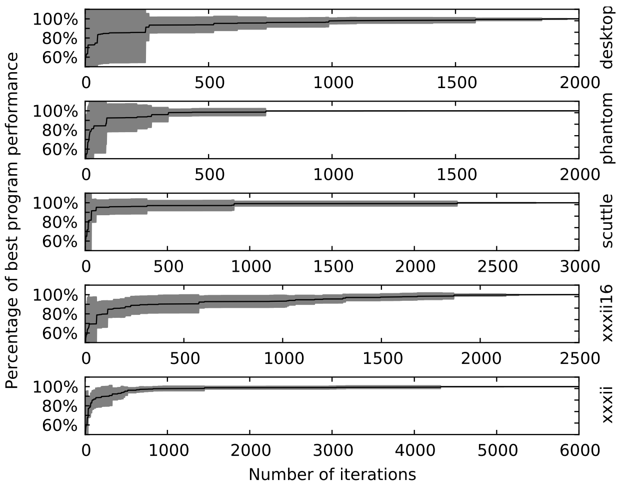

Figure 6. Convergence of the Monte Carlo search. The shaded areas show 99% confidence intervals for the mean.

These results reveal properties of the space that provide a starting point for reducing the cost of automatic search and could be used, in future work, to develop static heuristics that can optimize programs with minimal compile time cost.

6.3 Search Space Reduction

Removing the pair of parameters that have been identified as providing minimal effect on program execution time reduces the size of the search space by a factor of 6. The results of the PCA also show a linear dependency between another two parameters. This could be used to combine the parameters into a single parameter, reducing the size of the space further.

6.4 Is the subset explored representative?

The reliability of the evidence derived from the results of the Monte Carlo search depends on whether the subset of the space explored is representative of the entire space. Unfortunately this cannot be determined accurately without measuring the performance of the entire space, which is not feasible. However, the convergence of the best found execution time provides some evidence that the subset is representative.

Figure 6 shows the best execution time found against the number of iterations of the search. As a different total number of iterations were performed on each platform, a separate graph is shown for each. The line on each graph shows execution time averaged across all programs and the shaded region shows the variance.

These curves show, for every platform, that the best found execution time converges before the maximum number of iterations is reached, and at this point of convergence the variance is small. This means that the search is likely to have found a best execution time close to the true value across the entire space. Unfortunately, this does not show that the search has identified all of the optimal regions of the space.

A further issue with only measuring a subset of the optimization space is that some of the subsequent analysis requires the performance at all points in the space to be known. Nearest neighbor interpolation is used to overcome this by filling in possible values for the missing points. However, it should be noted that this could produce an estimate that does not accurately approximate the space as sharp peaks could be smoothed out.

7. Related Work

There are several works that investigate the autotuning of skeletons [12, 16, 21]. Unlike our work, all these works only investigate a single pattern and none investigate how the size of the search space can be reduced. Christen et al. designed an autotuned framework for stencil computation [12]. The framework uses Powell and Nelder-Mead search strategies. There is no information about the number of searches performed nor how close the strategy came to an optimal configuration. Wang and O'Boyle [21] use machine learning to autotune pipeline computation. Their method requires 3,000 iterations and achieves about 60% of the best obtainable performance. Dastgeer et al. [16] use machine learning to autotune a skeleton for simple data parallel operations. However, this work only evaluates autotuning for one parameter and with one application.

In addition, several groups have looked at autotuning of generic parallel programming frameworks. Contreras and Martonosi investigate how the scheduler in Intel's Threading Building Blocks library can be optimized [15]; while Wang and O'Boyle optimize thread number and scheduling strategy for OpenMP programs [20].

8. Conclusions and Future Work

The experiments carried out have demonstrated that there is significant scope for improvement in performance over a human expert, with an average speedup across all the programs and platforms of $1.6\times$. This is an impressive result, as human expertise is usually capable of doing as well as, if not better than, a compiler. They also demonstrate that the space is non-linear, and program and platform dependent, which makes it very difficult for a human expert to manually tune programs.

Our exploratory analysis of the optimization space performs an initial step in developing automatic optimization techniques for parallel skeleton programs that can achieve significantly better program performance than that achievable by a human expert. Two of the parameters were identified as having a minimal effect on program performance and the optimal regions of the optimization space are platform and program dependent. Principal Components Analysis also identified a linear dependency between the number of workers and buffer size. This pair of parameters account for 70% of the variability in the optimal regions of the optimization space, therefore performing automatic search in this reduced two dimensional space may be able to provide near optimal performance with only a couple of iterations.

In future work, we will develop static heuristics from these results that improve program performance without requiring a costly search of the optimization space at compile time.

Acknowledgments

We would like to thank Marco Aldinucci for sharing his expertise on the implementation of FastFlow and Christopher Thompson for the use of his 6-core machine.

References

[1] H. Abdi. *The Bonferroni and Šidák corrections for multiple comparisons*. Sage, 2007.

[2] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimization. In *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 955–962, 1999.

[3] M. Aldinucci and M. Torquati. FastFlow website, 2011. URL `http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about`.

[4] M. Aldinucci, M. Danelutto, M. Meneghin, M. Torquati, and P. Kilpatrick. Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed. In *Proceedings of the International Conference on Parallel Computing*, volume 11 of *Advances in Parallel Computing*, pages 273–280, 2009.

[5] M. Aldinucci, M. Torquati, and M. Meneghin. FastFlow: Efficient parallel streaming applications on multi-core. Technical Report TR-09-12, Università di Pisa, Dipartimento di Informatica, Italy, 2009.

[6] M. Aldinucci, M. Meneghin, and M. Torquati. Efficient smith-waterman on multi-core with FastFlow. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 195–199, 2010.

[7] M. Aldinucci, S. Ruggieri, and M. Torquati. Porting decision tree algorithms to multicore using FastFlow. In *Proceedings of the 2010 European conference on Machine Learning and Knowledge Discovery in Databases: Part I*, ECML PKDD'10, pages 7–23, 2010.

[8] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52:56–67, 2009.

[9] V. Barnett and T. Lewis. *Outliers in Statistical Data*. Wiley Series in Probability & Statistics. Wiley-Blackwell, 1994.

[10] C. M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006.

[11] D. Caromel and M. Leyton. Fine tuning algorithmic skeletons. In *13th International Euro-Par Conference: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 2007.

[12] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 25th International Parallel Distributed Processing Symposium*, pages 676–687, 2011.

[13] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[14] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30:389–406, 2004.

[15] G. Contreras and M. Martonosi. Characterising and improving the performance of intel threading building blocks. In *IEEE International Symposium on Workload Characterization*, pages 57–66, 2008.

[16] U. Dastgeer, J. Enmyren, and C. W. Kessler. Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-gpu systems. In *Proceeding of the 4th International workshop on Multicore software engineering*, IWMSE '11, pages 25–32, 2011.

[17] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, 2007.

[18] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software: Practice and Experience*, 40:1135–1160, 2010.

[19] M. Torquati. Single-producer/single-consumer queue on shared cache multi-core systems. Technical Report TR-10-20, Università di Pisa, Dipartimento di Informatica, Italy, 2010.

[20] Z. Wang and M. F. P. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 75–84, 2009.

[21] Z. Wang and M. F. P. O'Boyle. Partitioning streaming parallelism for multi-cores: A machine learning based approach. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 307–318, 2010.