MaSiF: Machine learning guided auto-tuning of parallel skeletons

MaSiF: Machine Learning Guided Auto-tuning of Parallel Skeletons

Alexander Collins, Christian Fensch, Hugh Leather, and Murray Cole
School of Informatics, University of Edinburgh, Edinburgh EH8 9AB, UK
a.collins@ed.ac.uk, c.fensch@ed.ac.uk, hleather@inf.ed.ac.uk, mic@inf.ed.ac.uk

***Abstract*—Parallel skeletons provide a predefined set of parallel templates that can be combined, nested and parameterized with sequential code to produce complex parallel programs. The implementation of each skeleton includes parameters that have a significant effect on performance; so carefully tuning them is vital. The optimization space formed by these parameters is complex, non-linear, exhibits multiple local optima and is program dependent. This makes manual tuning impractical. Effective automatic tuning is therefore essential for the performance of parallel skeleton programs.**

In this paper we present MaSiF, a novel tool to auto-tune the parallelization parameters of skeleton parallel programs. It reduces the size of the parameter space using a combination of machine learning, via nearest neighbor classification, and linear dimensionality reduction using Principal Components Analysis. To auto-tune a new program, a set of program features is determined statically and used to compute k nearest neighbors from a set of training programs. Previously collected performance data for the nearest neighbors is used to reduce the size of the search space using Principal Components Analysis. Good parallelization parameters are found quickly by searching this smaller search space. We evaluate MaSiF for two existing parallel frameworks: Threading Building Blocks and FastFlow. MaSiF achieves 89% of the performance of the oracle on average. This exploration requires just 45 parameters values on average, which is ~0.05% of the optimization space. In contrast, a state-of-the-art machine learning approach achieves 51%. MaSiF achieves an average speedup of 1.32× over parallelization parameters chosen by human experts.

I. Introduction

The use of multiple simpler processing elements, or cores, is now a focus for industry and research as it is a promising avenue to continue improving processor performance. By utilizing 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 [1], [2]. They separate algorithm description from implementation, whilst preserving performance. This enables platform independence and removes low-level implementation concerns from the application programmer. The idea is to provide a collection of templates, or skeletons, each of which implements a standard algorithmic technique. A skeleton library provides implementations for the skeletons, which are used to compile the program into a form that can be executed. Common skeletons include task farms, pipelines, divide and conquer, and data-parallel map and reduce [1], [3].

The performance of parallel skeleton programs relies on an appropriate choice of skeletons, efficient implementation of the sequential code fragments, and efficient library implementations of of each skeleton [4], [5]. The latter involves parallelization parameters. Finding the optimal settings for these parameters is complex, since they depend on both target machine and application. By tuning these parameters, we improve the performance of two skeleton parallel frameworks: Intel's Threading Building Blocks (TBB) [6] by 1.47× and FastFlow [7], [8] by 1.17×, on average compared to settings chosen by expert developers (the authors of FastFlow, TBB and the PARSEC [9] TBB benchmarks).

As the search space is too large for exhaustive search, previous work on auto-tuning parameters either predicts the best settings directly or uses iterative compilation (which may be guided). We show that a state of the art predictive approach fails to achieve better performance than the human expert, whereas our approach improves performance.

We present MaSiF, a domain-specific tool that auto-tunes parallel skeleton programs. In an offline phase, we identify a set of parallelization parameters for MaSiF to tune and collect performance data for a set of training programs. We identify 4 such parameters in TBB and 5 in FastFlow. Given a new program, static program features are extracted and used to identify similar programs from the training set. The performance data for these training programs is used to search for optimal parameter values for the new program. Domain specific search space reduction techniques are used to reduce the number of parameters investigated.

It is not feasible to perform an exhaustive search of the space due to its size (10^4 for TBB and 10^5 for FastFlow) therefore our oracle examines a random 10% subset of the space. Our results show that MaSiF achieves 91% of the oracle performance for TBB and 86% for FastFlow. This demonstrates that our technique is effective at automatically optimizing parallel skeleton programs without the need for human expertise. In contrast, the parameters predicted by a state of the art machine-learning approach developed by Wang and O'Boyle [10] achieve 55% of the oracle for TBB and 47% for FastFlow. Compared to parallelization parameters chosen by a human expert, MaSiF achieves an average speedup of 1.47× for TBB and 1.17× for FastFlow. MaSiF achieves this performance by searching 0.05% of the parameter space.

A. Contributions

We make the following contributions:

- We present MaSiF, a tool for machine learning guided auto-tuning of parallelization parameters in parallel skeleton frameworks. It uses a novel parameter space reduction and search technique based on Principal Components Analysis, reducing the size of the space to 0.05%. Unlike previous approaches, it uses skeleton-based program features that are directly available to the application developer.
- We demonstrate that MaSiF works for both the Threading Building Blocks [6] and FastFlow [7], [8] parallel skeleton frameworks.
- We show that our technique achieves 91% of the performance of the oracle for TBB and 86% for FastFlow. This equates to a speedup of 1.47× over human experts for TBB and 1.17× for FastFlow.

The remainder of this paper is structured as follows. Section II provides a brief introduction to the MaSiF search strategy. Section III describes the tuning parameters we identified in Intel's TBB [6] and the FastFlow skeleton framework developed by Aldinucci et al. [8], [7]. Section IV describes MaSiF; our novel, machine learning guided auto-tuning tool for parallel skeleton programs. Section V explains our experimental methodology to evaluate MaSiF, the results of which are presented in Section VI. Section VII follows with related work, and Section VIII summarizes our conclusions and future work.

II. The MaSiF Search Strategy

The left plot in Fig. 1 shows the optimization space of the FastFlow *mandelbrot* program with Principal Components Analysis (PCA) applied to reduce the dimensionality to 2. Grey shaded areas mark parallelization parameters that obtain 95% or more of the best performance. The plot also shows the mean and the two most significant eigenvectors.

The right plot in Fig. 1 shows the optimization space of a different program: *bzip2*, under the same transformation that was applied to the *mandelbrot* program. Again, the grey shaded area indicates 95% or more of the best available performance. Even though the optimization space looks completely different, we can apply a search technique that uses the mean and eigenvectors from the *mandelbrot* optimization space. Starting at that mean, we exhaustively search along the first eigenvector in both directions to find the best point. From there, we now search along the direction of the second eigenvector. The search finds a point that obtains 97% of the best available performance.

This motivating example shows that the eigenvectors provided by the PCA for one program can be used to guide the search in the optimization space of a different program, if the programs are a good match. In Section IV-C, we show how matching programs can be found.

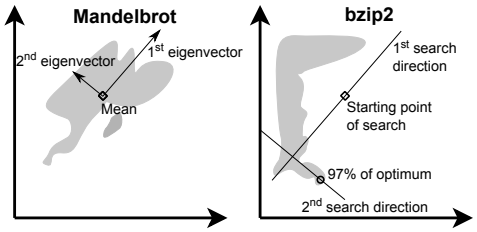


Fig. 1: The left plot shows the optimization space of the *mandelbrot* program after PCA has been applied to reduce the dimensionality to 2. The shaded areas show points in the space that are within 95% or closer of the optimal performance available. The plot of the *mandelbrot* program shows in addition the mean and the two most important eigenvectors. In the right plot, we show the optimization space of the *bzip2* program after the same transformation produced by PCA on the *mandelbrot* has been applied. Starting at the mean of the *mandelbrot* program in the *bzip2* space and searching in the direction of the first and then the second eigenvector, we are able to find a point that provides 97% of the best performance.

Parameter	Description	Values
threads	Number of threads	1, ..., # cores × 1.5
grainsize	Grain size for subdividing inputs	1, 2, 4, 8, ..., 2^{16}
partitioner	Input partitioning strategy	`auto_partitioner`, `affinity_partitioner`, `simple_partitioner`
allocator	Memory allocator	`std::allocator`, `tbb_allocator`, `zero_allocator`, `cache_aligned_allocator`, `scalable_allocator`

TABLE I: Optimization parameters in Threading Building Blocks. The value column specifies the points considered in the search space.

III. Parallel Skeleton Frameworks

In this section we discuss the two parallel skeleton frameworks that MaSiF integrates with and describe parameters that MaSiF uses to tune performance, and we evaluate the success of this tuning in Sections V and VI.

A. Threading Building Blocks

Intel Threading Building Blocks (TBB) [6] is a parallel skeleton library for shared-memory multi-core systems. It provides a set of parallel patterns, including `parallel_for` and `parallel_reduce`, built on top of a work-stealing task scheduler, or *thread pool*. Its programming interface is similar to the algorithms and containers provided by the C++ Standard Template Library.

Table I summarizes the four tunable parameters in TBB. In more detail, these parameters are:

- *Number of threads.* This controls the number of threads in TBB's thread pool.
- *Grain size.* This configures how TBB divides the input to `parallel_for` and `parallel_reduce` skeletons into tasks. It controls the granularity of tasks executed by the thread pool.
- *Partitioner.* This provides further control of how inputs are subdivided, and affects the distribution of tasks by the scheduler.

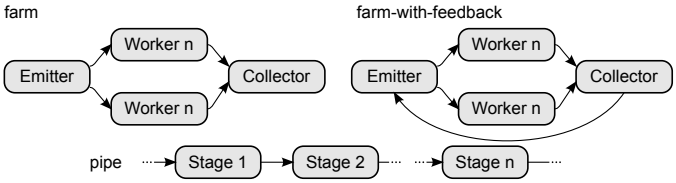


Fig. 2: FastFlow's skeletons. Arrows represent single-producer single-consumer queues and boxes represent threads.

Parameter	Description	Values
numworkers	Number of threads	1, . . . , # cores × 1.5
buffertype	Type of queue	Bounded or unbounded
buffersize	Buffer size of queues	1, 2, 4, 8, . . . , 2^{20}
batchsize	Number of items in a batch	1, 2, 4, 8, . . . , 2^{20}
cachealign	Memory allocation alignment of items	64, 128 or 256 bytes

TABLE II: Optimization parameters in FastFlow. The values column specifies the points considered in the search space.

- *Memory allocator.* TBB provides several memory allocators that can be used with its collection classes (including `tbb::concurrent_vector`). The best choice of allocator is application dependent. For example, making thread-local memory allocations improves performance for some applications but harms it for others.

B. FastFlow

FastFlow [11], [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. [12], [8], [13], [11] 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 for parallel skeleton libraries. It is implemented as a library in C++.

FastFlow provides three parallel skeletons: `farm`, `farm-with-feedback` and `pipe`. These are implemented using threads and single-producer single-consumer queues as shown in Fig. 2. They are arbitrarily nestable and are parameterized with sequential C++ code.

Table II summarizes the five tunable parameters in FastFlow. In more detail, these parameters are:

- *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. The developer can specify 'unsafe' regions of the parameter space so that MaSiF can avoid deadlock.
- *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. For unbounded queues, this controls the size of each chunk of memory allocated to buffer items.
- *Task granularity.* Tasks sent between threads can be grouped into batches of a chosen size. This is likely to reduce communication/synchronization overhead at the cost of reducing the available task-parallelism and adversely effecting load balancing.
- *Cache alignment.* FastFlow's memory allocator adds padding between tasks to better exploit the cache, and the amount of padding is controlled by this parameter.

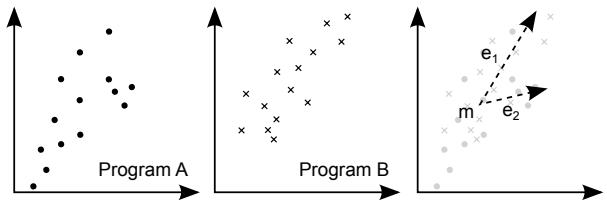


Fig. 3: Extracting the search vectors from the training data. The left and middle plot show a set of 'near optimal' parameter values for two nearest neighbor programs from the training set. The right plot shows the mean m and eigenvectors e_1 and e_2 (as dotted arrows) resulting from the PCA.

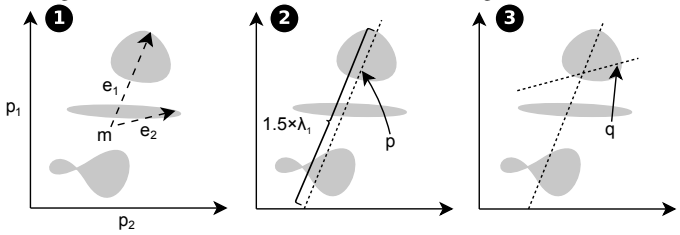


Fig. 4: Visualization of the eigenvector search on a two-dimensional parameter space, formed by parameters p_1 and p_2. Plot ❶ shows the parameter space. Parameter values that provide near optimal performance are shaded. Applying PCA to these near optimal parameter values gives mean m, eigenvectors e_1 and e_2, and eigenvalues λ_1 and λ_2. First, we search exhaustively along a line centered on m with direction e_1 and length $1.5 \times \lambda_1$ as shown in ❷. The best parameter values found are at p. We then search exhaustively along a line centered on p with direction e_2 and length $1.5 \times \lambda_2$ as shown in ❸. The best parameter values found are q. This is repeated for a chosen number of eigenvectors, and the best parameters found are the result of the search.

IV. Auto-tuning using MaSiF

This section describes the operation of MaSiF, our machine learning guided auto-tuning tool. Sections IV-A and IV-B describe how MaSiF optimizes a new program given knowledge of where regions of good parameter values lie. Section IV-C details how MaSiF estimates where these good regions lie, using training data collected a priori. Section IV-D describes how a developer uses the tool.

A. Reducing the Size of the Search Space

The parameter space that needs to be searched contains about 10^4 points for TBB, and 10^5 points for FastFlow. MaSiF uses Principal Components Analysis [14] to reduce this significantly, without excluding all of the parameter values that provide good performance. PCA is used in two ways: to reduce the size of the space and to provide directions for the search, explained in Section IV-B.

Feature	Description
ske	The skeleton used; 0 = parallel_for, 1 = parallel_reduce
rs	Data structure read from; 0 = blocked_range, 1 = shared array, 2 = concurrent_vector
wr	Data structure written to; 0 = atomic variable, 1 = shared array, 2 = concurrent_vector, 3 = shared variable
cost	Computational time complexity per task; 0 = $\mathcal{O}(1)$, 1 = $\mathcal{O}(n)$ (potentially statically unknowable)

(a) TBB

Feature	Description
ske	The skeleton used; 0 = farm, 1 = farm-with-feedback
col	Whether the program uses a collector thread; Yes/No
bf	The number of tasks created by a task (potentially statically unknowable)
ts	The size of the task passed to workers (potentially statically unknowable)

(b) FastFlow

TABLE III: Program features for TBB and FastFlow. Some of these features might be statically unknowable.

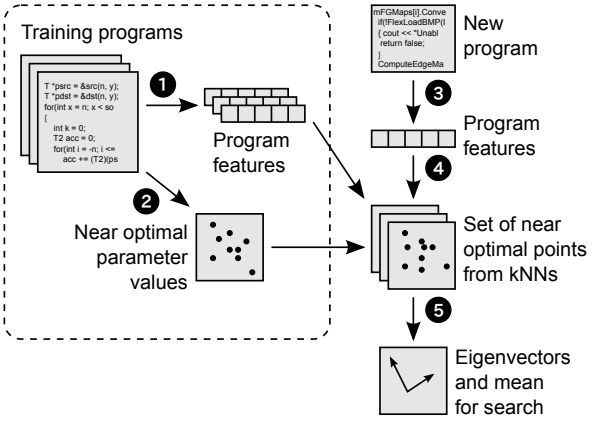


Fig. 5: Schematic of how machine learning guides the search. The dotted box indicates the offline training phase. ❶ Extract program features from training programs. ❷ Find a set of near optimal parameter values for each training program, by measuring a random subset of the space. ❸ Extract program features from the new program. ❹ Compute the k-nearest neighbors, using the program features, to create a set of near optimal parameter values for all nearest neighbors. ❺ Apply PCA to the set of near optimal parameter values. The resulting mean and eigenvectors are used to perform the search.

To reduce the parameter space, all 'near optimal' parameter values are collected. We define 'near optimal' as parameter values with performance within 5% of the maximum performance. Maximum performance refers to the best found performance by either conducting an exhaustive search or sampling of a subspace. Applying PCA to this set of 'near optimal' parameter values provides a new set of orthogonal basis vectors (see Fig. 3). This basis spans the original space, so we can describe any parameter value in this new basis. An example of this is shown in the left hand plot of Fig. 4.

PCA also returns a measure of the variance of the data in the direction of each eigenvector, called the eigenvalues. We can use these to scale the eigenvectors so that they only cover regions of the space where near optimal parameter values are present. These scaled eigenvectors are shown as dotted arrows in the left hand plot of Fig. 4. In our search, we start at the mean and search along $1.5\times$ in the direction of the eigenvectors, which corresponds to covering 3σ (99.7%) of the variance. The eigenvectors returned by PCA are also ordered by their associated eigenvalue. This allows further space size reduction: we can remove the dimensions (eigenvectors) which only capture a small variance in the data.

$$D(\mathbf{x},\mathbf{y}) = \sqrt{\sum d(x_i,y_i)^2}$$
$$d(x,y) = \begin{cases} 0 & \text{if } x = \diamond \text{ or } y = \diamond \\ y - x & \text{otherwise} \end{cases}$$

Fig. 6: The modified Euclidean distance metric used to compute the k-nearest neighbors. $D(\mathbf{x},\mathbf{y})$ computes the distance between feature vectors $\mathbf{x}$ and $\mathbf{y}$. $d(x,y)$ takes account of statically unknowable values ($\diamond$).

B. Searching the Reduced Space

On top of the technique used to reduce the size of the search space, MaSiF searches the space in a way that further reduces the number of parameter values that need to be measured.

The previous step that uses PCA to reduce the search space results in a new set of new basis vectors for the space, called eigenvectors. These form a set of orthogonal bases ordered by the amount of variation in the data that they each capture. This means that most of the variation in the near optimal parameters is in the direction of the first eigenvector, the next most is in the direction of the second eigenvector, and so on.

MaSiF searches along each of these eigenvectors in turn, as shown in Fig. 4. Compared to an exhaustive search of the reduced space, this reduces the number of parameters that need to be searched significantly. Given m eigenvectors with n parameter values along each, we only measure $\mathcal{O}(nm)$ parameter values in a space whose size is $\mathcal{O}(n^m)$.

C. Guiding Search using Machine Learning

Our eigenvector search technique assumes that we know where the near optimal parameters are. However, we do not know this for a new program before executing it. MaSiF uses machine learning to estimate where these regions lie. This is shown in Fig. 5.

Given a set of existing training programs, we statically extract a set of program features from each. The program features for TBB and FastFlow are summarized in Table III. In an offline training phase, we search a random subset of their parameter space to find a set of near optimal parameters for each training program. Given a new program, we perform the same feature extraction, and use the feature vector to select a set of similar programs from the training set. The search space reduction and search techniques, described previously in Section IV-A and Section IV-B, are then applied to the set

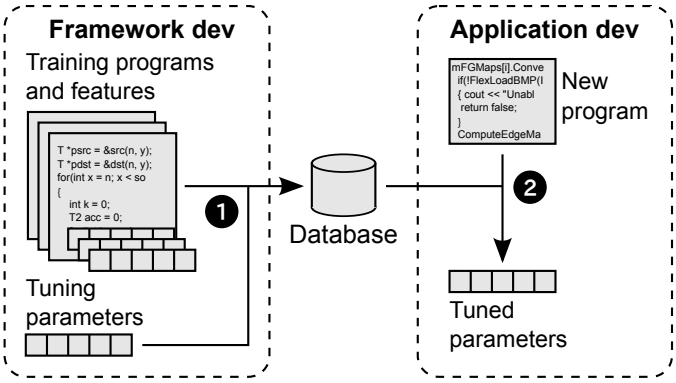


Fig. 7: Schematic of how MaSiF is used. ❶ In an offline training phase, the framework developer specifies tunable parallelization parameters, high-level static program features and collects performance data for a set of training programs. ❷ An application developer uses the MaSiF optimization tool to find good parallelization parameters for a new program.

of near optimal parameter values from the similar programs. The set of similar programs is determined using the k-nearest neighbors algorithm [14]. This selects the k programs from the training set with the smallest distance between their feature vector and the new programs feature vector. The distance between feature vectors is determined by a distance metric. We use a modified Euclidean distance metric defined in Fig. 6.

Some of the features we have chosen may not be determinable statically, and can therefore have an 'unknown' ($\diamond$) value. The modification to the Euclidean distance takes this into account, by assigning a distance of 0 to between any value and an 'unknown' ($\diamond$). Note that this design choice artifically pushes programs with 'unknown' feature values closer together. However, a different value could push programs further apart. By definition, there is no good choice for the value that is used as substitution.

D. Using MaSiF

MaSiF is a program optimization tool for parallel skeleton frameworks. A schematic summarizing its use is shown in Fig. 7.

Application developers use MaSiF to automatically tune a new program. After implementing their application using the parallel framework, they run MaSiF's optimization script on the target architecture. The script executes their program to search for the best parameters, using the database to drastically reduce the number of parameters that are evaluated.

E. Extending MaSiF

To extended MaSiF for use on further skeleton frameworks, it is only necessary to specify the following:

- The set of tunable parallelization parameters present in the framework, and a range of valid values for each.
- Provide a compatible interface for MaSiF to set the parameters.
- The set of static program features, and the range of valid values for each.
- A set of training programs (implemented using the parallel framework) with the values of their static program features.

MaSiF then needs to be trained by running its training script, which collects performance data and stores it in a database. This offline process needs to be performed once for every target architecture. It is important to note that these changes do not need to be made by an application developer, only by the system developer.

V. Experimental Setup

This section describes our experiments to evaluate MaSiF. The experimental hardware, parallel frameworks and benchmark programs are described in Section V-A. Section V-B explains our 'oracle' approach, which is used to provide an estimate of the maximum performance of each program. Section V-C describes alternative approaches against which we compare MaSiF. Section V-D describes the evaluation methodology. Section V-E describes the collection of training data for MaSiF.

A. Experimental Platform and Benchmarks

We use a 32-core shared memory machine to run the experiments. It has 4× Intel Xeon L7555 8-core processors and 64GB of main memory, running Linux 2.6.37.6 (64-bit). This processor includes several features to adjust CPU parameters according to the workload of the system, including frequency scaling. These introduce significant noise into performance measurements; therefore we disable them for our experiments.

To show that MaSiF works across a range of parallel frameworks, we evaluate it for both Threading Building Blocks and FastFlow. These frameworks have different static program features, different tunable parallelization parameters and different sets of benchmark programs.

We use 10 existing TBB programs and 10 existing FastFlow programs for the experiments. They are listed in Table IV. Of the TBB programs, 4 are from the PARSEC 3.0 benchmark suite [9] and 6 are from the TBB 4.1 source distribution. The FastFlow programs are from the FastFlow source distribution. The benchmarks are implemented in C++ and compiled using GCC 4.5.1. The TBB programs use a combination of TBB's `parallel_for` and `parallel_reduce` skeletons. The FastFlow programs use two of the skeletons provided by FastFlow: `farm` and `farm-with-feedback`.

B. The Oracle Approach

We compare MaSiF against an oracle, to determine whether MaSiF achieves maximum program performance. Ideally the oracle would measure program performance for every point in the optimization space. However, the parameter space is large; with approximately 10^4 parameter choices for TBB and 10^5 for FastFlow. Finding the best program performance by exhaustively measuring the performance of each program, for every parameter choice, with repeats to quantify measurement noise would take several years.

To avoid this feasibility problem, our oracle measures the performance of a random 10% subset of the optimization space. Program execution time for this subspace was measured for each program, taking approximately 3 months. The best

Program	Description
bs	blackscholes (from PARSEC)
bt	bodytrack (from PARSEC)
ch	Convex hull of a set of points
gol	Game of life simulation
po	Polygon overlay
pr	Prime number sieve
sc	streamcluster (from PARSEC)
sm	Seismic wave simulation
swa	swaptions (from PARSEC)
tc	Tachyon ray-tracer

(a) TBB

Program	Description
aq	Adaptive Quadrature algorithm
bz2	Parallel bzip2 compression
cwc	Simulation of CWC calculus for biological systems
dt	Implementation of the C4.5 decision tree algorithm
fib	Naïve recursive algorithm compute Fibonacci numbers
mb	Mandelbrot fractal generator
mm	$O(n^3)$ nested-loops matrix multiplication
nq	n-queens problem solver
qs	Parallel quicksort
sw	Smith-Waterman algorithm for gene sequence alignment

(b) FastFlow

TABLE IV: Programs used in the experiments.

	bs	bt	ch	gol	po	pr	sc	sm	swa	tc
ske	1	1,1,1,1	1,0,0	1	1	0	1,0,0	1,1	1	1
rs	1	1,1,1,1	0,2,2	1	2	0	1,1,1	1,1	1	1
ws	1	1,1,1,1	2,3,3	1	3	3	1,0,0	1,1	1	1
cost	0	0,1,1,1	0,0,0	0	◇	◇	0,0,0	1,1	◇	◇

(a) TBB

	aq	bz2	cwc	dt	fib	mb	mm	nq	qs	sw
ske	0	0	1	1	1	0	0	1	1	0
col	0	1	1	0	0	1	0	0	0	0
bf	2	0	0	◇	2	0	0	0	2	0
ts	8	◇	◇	◇	8	◇	◇	16	12	◇

(b) FastFlow

TABLE V: Statically determined feature vectors for each TBB and FastFlow program. ◇ indicates an unknown value, if the feature cannot be determined statically. Some of the TBB benchmarks have multiple feature vectors, shown in the table as lists of values. These are averaged before applying the k-nearest neighbor algorithm.

execution time found in this 10% subset provides the oracle's estimate of best performance.

C. Comparison Against Other Approaches

We compare MaSiF against a state of the art machine-learning approach developed by Wang and O'Boyle [10]. This approach uses a combination of Support Vector Machines (SVM) and Artificial Neural Networks (ANN) to predict the optimal number of threads and scheduling strategy for OpenMP applications. Lacking a name and having a somewhat cumbersome abbreviation (SVM+ANN), we refer to this approach by the easier abbreviation *EZ*. We adapt EZ to tune the same parallelization parameters as MaSiF, for both TBB and FastFlow. This allows us to directly compare the program performance found by both MaSiF and EZ.

Both MaSiF and EZ use program features to classify programs for training. The choice of these features affects the results of this classification. MaSiF uses skeleton-based features extracted from the source code. These are naturally apparent from the skeleton programming model, and include features such as the choice of parallel skeleton or the size of each task. The intuition is that these features capture high-level structure of the algorithm, and divide the programs into algorithmic classes. In contrast EZ uses features extracted from profiling runs of the compiled binary. These include the number of branch instructions and load/store instructions.

We compare MaSiF against two variants of EZ. The first uses the same binary features devised by Wang and O'Boyle [10] (called *EZ-BF*). The second uses the same skeleton-based features as MaSiF (called *EZ-SF*).

We also compare MaSiF against human expert chosen parameters. The TBB and FastFlow benchmark programs contain manually tuned parameters, which have been set by each application developer. These were tuned by the TBB developers, FastFlow developers and PARSEC [9] TBB benchmark developers. We compare against the performance achieved by these parameter choices to demonstrate that MaSiF outperforms human experts.

D. Evaluation Methodology

We use a leave-one-out cross-validation (a standard technique) to evaluate MaSiF and EZ. For our set of 10 benchmark programs, we use 9 of them for the training. The remaining benchmark is then optimized by the auto-tuner. We repeat this for each of the programs. This means that we always test the auto-tuner using an *unseen* program.

The following methodology is used to measure program execution time:

- Repeated measurements are made so that error in the sample mean can be quantified using confidence intervals. Where appropriate, 99% confidence intervals for the mean are reported alongside the results.
- Outlier removal is performed using interquartile range removal [15].

E. Training Data Collection

The performance measurements used to provide the oracle are also used as training data. For each program, all parameter values that are within 5% of the oracle performance are collected together. These constitute the 'best points' on which MaSiF performs its training.

VI. Results

This section presents our evaluation of MaSiF. Section VI-A shows the results of MaSiF's classification of the training pro-

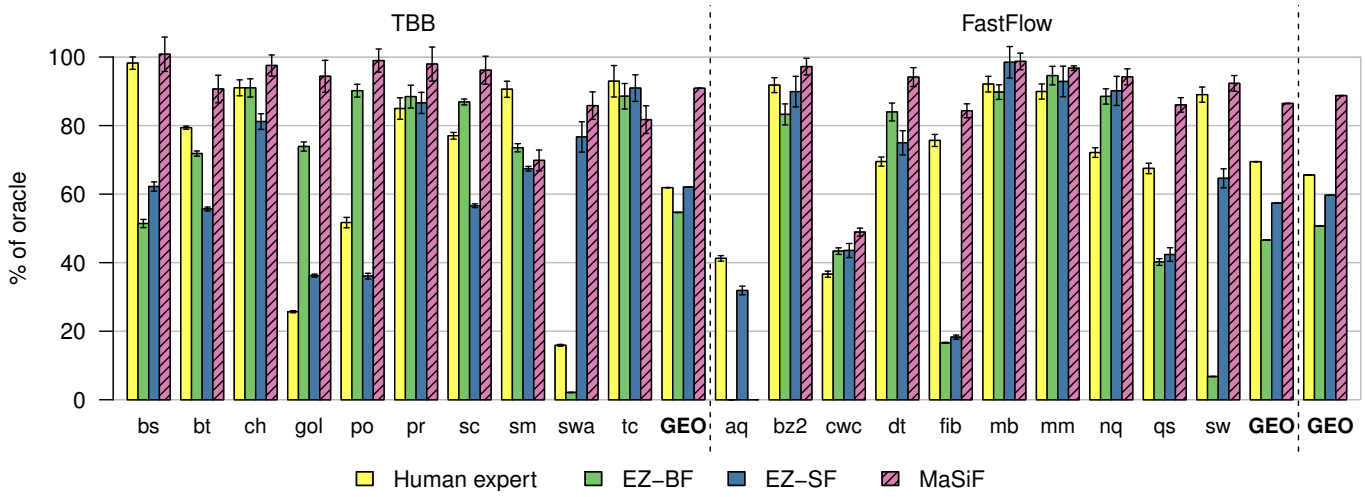


Fig. 8: The percentage of the oracle performance achieved by a human expert, the best competing approach and MaSiF. GEO is the geometric mean.

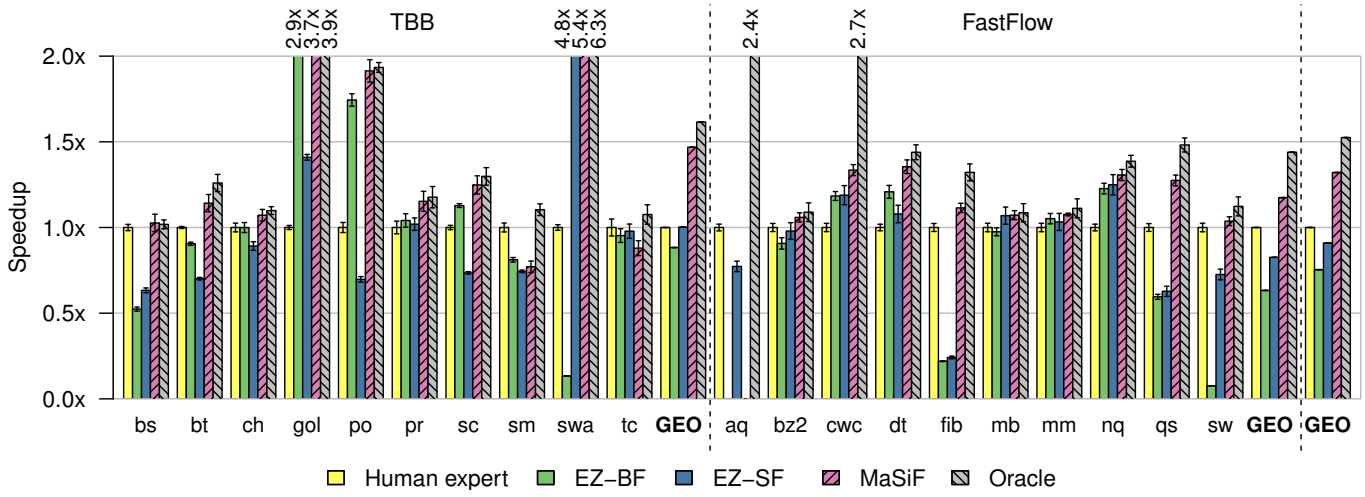


Fig. 9: Speedup over a human expert achieved by the best competing approach, MaSiF and the oracle. GEO is the geometric mean.

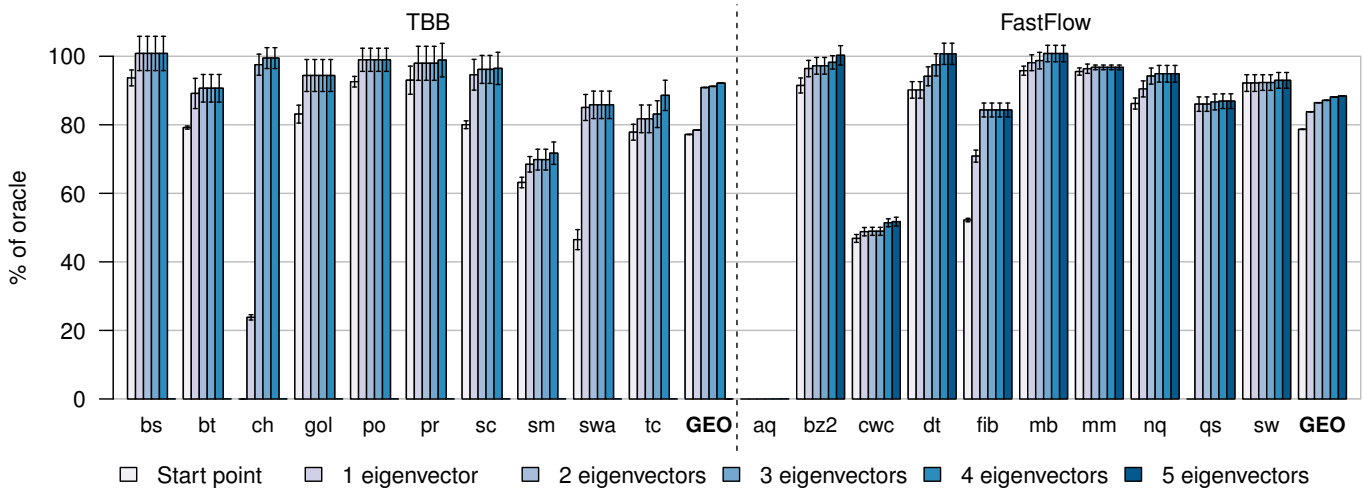


Fig. 10: The effect on MaSiF of varying the number of eigenvectors used for the search, as the percentage of the oracle performance achieved. GEO is the geometric mean.

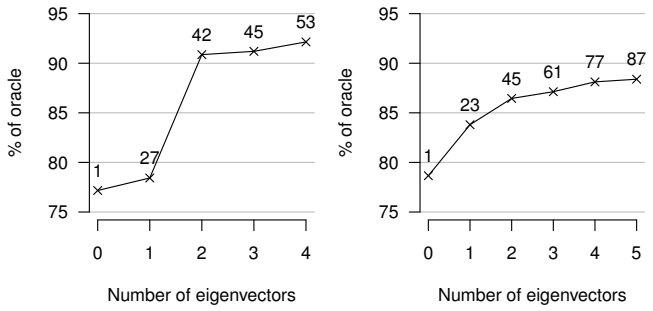
Prog.	Nearest neighbors		
bs	swa	bt	tc
bt	swa	bs	tc
ch	po	swa	bt
gol	swa	bt	tc
po	swa	bt	tc
pr	ch	po	swa
sc	swa	bt	tc
sm	swa	bt	tc
swa	bt	tc	gol
tc	swa	bt	gol

(a) TBB

Prog.	Nearest neighbors		
aq	sw	mm	fib
bz2	mb	sw	mm
cwc	nq	mb	bz2
dt	nq	qs	fib
fib	dt	qs	aq
mb	bz2	sw	mm
mm	sw	nq	mb
nq	dt	sw	mm
qs	dt	fib	nq
sw	mm	nq	mb

(b) FastFlow

TABLE VI: The 3 nearest neighbors for each TBB and FastFlow program ordered by increasing distance from left to right, using the modified Euclidean distance metric.



(a) TBB (b) FastFlow

Fig. 11: The effect of varying the number of eigenvectors used by the search on the percentage of the oracle performance achieved, averaged across programs for TBB and FastFlow. With zero eigenvectors, the search only considers the mean parameter values return by the PCA. Each line is annotated with the average number of parameter values that are searched along each eigenvector.

grams. Section VI-B compares MaSiF to the oracle approach and manual tuning performed by human experts. Section VI-C compares against an alternative state of the art machine learning approach. Section VI-D explores the effect that the number of eigenvectors has on MaSiF and Section VI-E examines its convergence time. For our analysis, we use the geometric mean to compute average speedup.

A. Training Program Classification

Table V shows the values of the feature vectors for each of the TBB and FastFlow programs. In some cases, the value of a feature cannot be determined statically. These are marked as unknown, denoted '$\diamond$'. For example, for some of the FastFlow programs the branching factor is dependent on input data. Some of the TBB benchmarks have multiple feature vectors, shown in the table as lists of values. These are averaged before applying the k-nearest neighbor algorithm.

Table VI shows the results of MaSiF's classification of training programs. For each program, a set of three nearest neighbors are identified. The distance metric used is defined in Section IV-C. When auto-tuning one of the programs, these sets of nearest neighbor programs are used to provide training data for the search.

B. Comparison Against the Oracle and Human Experts

Fig. 8 shows the percentage of the oracle performance achieved by several approaches, including MaSiF and parameters chosen by human experts. For 13 of the 20 programs, MaSiF achieves over 90% of the oracle performance. On average, MaSiF achieves 91% of the oracle performance for TBB and 86% for FastFlow. In contrast, the parameters chosen by human experts only achieve 62% of the oracle performance for TBB and 69% for FastFlow. Human expert chosen parameters outperform MaSiF on just 3 of the programs (*sm* and *tc* for TBB, and *aq* for FastFlow).

Fig. 9 shows the speedup over human expert parameters achieved by the different approaches, including MaSiF. On average, MaSiF achieves a speedup of 1.47× over human expert chosen parameters for TBB and 1.17× for FastFlow.

It is important to note that MaSiF only measures the performance of 45 parameter values, on average, in order to obtain this performance. This is far fewer than the size of the space, which consists of approximately 10^4 parameter values for TBB and 10^5 for FastFlow.

MaSiF fails to optimize the Adaptive Quadrature (*aq*) FastFlow program. We found that none of the nearest neighbors identified by the machine learning behaves similarly enough that their optimization space is useful when training the search. This could be fixed by increasing the number and variety of training programs.

The results of our experiments demonstrate that MaSiF is portable within the domain of parallel skeleton frameworks. On average, MaSiF beats both human experts and a state of the art predictive approach for both TBB and FastFlow.

C. Comparison Against Other Approaches

Fig. 8 also shows the percentage of the oracle performance achieved by a state of the art machine learning approach developed by Wang and O'Boyle [10]. EZ-BF uses features extracted from the program binaries using profiling, and EZ-SF uses the same skeleton-based features as MaSiF.

On average, EZ-SF performs better than EZ-BF. EZ-SF achieves 62% and 57% of the oracle for TBB and FastFlow respectively, and EZ-BF achieves 55% and 47%. This suggests that using skeleton features works better.

For TBB, EZ provides a speedup over the human experts of between 0.88× and 1.00×, compared to MaSiF at 1.47×. For FastFlow, EZ achieves a slowdown compared to the human experts of between 0.63× and 0.83×, compared to the speedup achieved by MaSiF at 1.17×.

Both EZ-BF and MaSiF failed to optimize the FastFlow Adaptive Quadrature (*aq*) benchmark, compared to the oracle and human expert chosen parameters.

In summary, MaSiF achieves better performance than both variants of EZ, for both TBB and FastFlow.

D. Varying the Number of Eigenvectors

MaSiF includes a parameter whose value needs to be manually chosen: the number of eigenvectors to search. Given an n-dimensional parameter space, MaSiF can search along

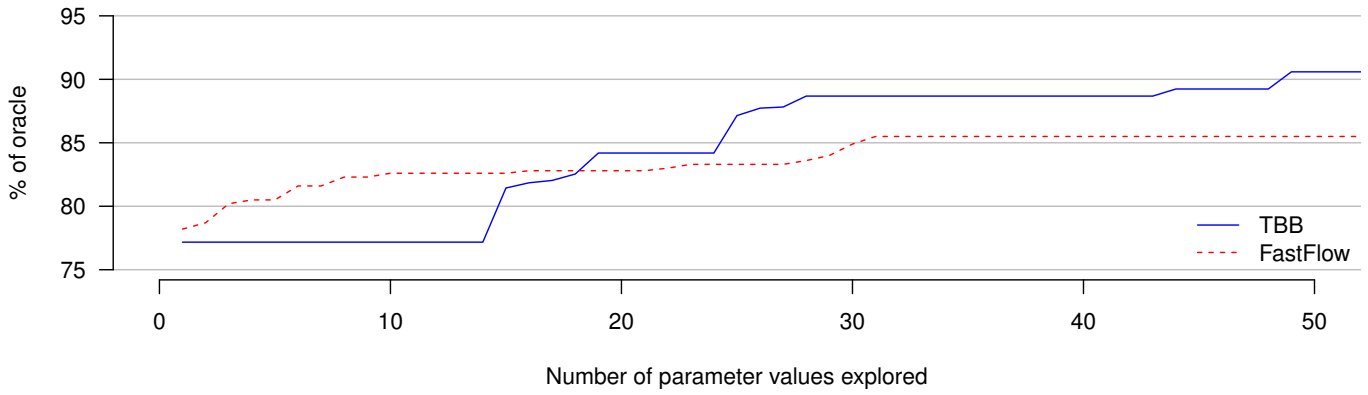


Fig. 12: The evolution of the best execution time found against number of parameter values explored, for TBB and FastFlow, averaged over all programs.

anywhere from 1 to n eigenvectors. The effect of varying this parameter on the program performance achieved is shown in Fig. 10.

This shows that for many of the programs, the mean value computed by PCA produces good results, with 91% of the oracle performance for TBB and 86% for FastFlow. However this is not the case for some programs, such as TBB's *ch* benchmark and FastFlow's *fib* benchmark. At the start point these only achieve 1% and 50% of the oracle performance respectively. After searching along two eigenvectors, these improve to 97% and 84%. It is therefore essential to search at least one or two eigenvectors to obtain good performance across all programs.

Fig. 11 shows how varying the number of eigenvectors affects the number of parameter values that need to be measured by the search. This shows a diminishing return when increasing the number of eigenvectors searched above two: the number of parameter values evaluated increases linearly whereas the performance achieved increases logarithmically.

E. Convergence Time

Fig. 12 shows the evolution of the average performance found by MaSiF as the number of parameters is increased. This plot highlights how few parameters values MaSiF requires to reach 88% of the oracle's performance. Our approach converges on parameter values that provide very good performance. The large step change is due to the search switching eigenvectors. When investigating a new eigenvector, the initial performance improvement is large.

VII. Related Work

Related work roughly falls into two distinct categories: auto-tuning skeleton programs and machine learning based search space reduction for iterative compilation.

There are several works that investigate the auto-tuning of skeletons [16], [17], [18]. Unlike our work, they only investigate a single pattern and none investigate how the size of the search space can be reduced. Christen et al. designed an auto-tuned framework for stencil computation [17]. 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 [16] use machine learning to predict the best transformations to auto-tune pipeline computation. Their method requires 3,000 iterations and achieves about 60% of the best obtainable performance. Dastgeer et al. [18] use machine learning to auto-tune a skeleton for simple data parallel operations. However, their work only evaluates auto-tuning for one parameter and with one application. Petabricks [19] implements different algorithms for a given problem. It then automatically selects the most appropriate algorithm for a given problem size. This includes the selection of different algorithms once the problem has been broken down into smaller chunks. In contrast, our work focuses on choosing the optimal tuning parameters for one particular algorithm.

There is a substantial amount of existing work that uses machine learning to predict the effectiveness of compiler optimizations for a given program [20], [21], [22]. The work of Agakov et al. [23] is close to our own. The authors use machine learning to model the shape of the search space of DSP kernels. They then focus the search towards areas of the space that are most promising. Unlike our work, their optimization space consists of compiler flags and they use Markov Chains and IID as their predictive model. Jantz and Kulkarni reduce the search space of optimization phase ordering in compilers [24]. They reduce the search space by eliminating false register dependencies, rather than using a more focused search strategy.

VIII. Conclusions and Future Work

We have demonstrated that MaSiF achieves 91% and 86% of the oracle performance for TBB and FastFlow respectively; by searching a small set of 45 parameters (on average). This is a speedup of 1.47× and 1.17× over human expert chosen parameters, for TBB and FastFlow respectively. MaSiF also outperforms a state of the art machine learning approach which achieves 55% and 47% of the oracle for TBB and FastFlow respectively.

These results show that MaSiF does not significantly impact the maximum possible performance achievable—in fact it

does better than a human expert and predictive machine learning approach—whilst markedly reducing the number of parameters that need to be searched. The PCA space reduction reduces the size of the search space by 275× on average, and searching along just two eigenvectors reduces the space by 1,925×.

Our technique is fully automatic and not constrained to a single parallel skeleton framework. We have demonstrated its efficacy for both Threading Building Blocks and FastFlow. These are two very different parallel skeleton frameworks – FastFlow is a streaming parallel framework whereas TBB is a sophisticated task-stealing scheduler.

In the future, we will investigate searching the reduced parameter space at runtime. By splitting program execution into fixed time epochs, we can perform the machine learning guided search at runtime. We will also investigate more sophisticated techniques to improve over the exhaustive search along each eigenvector. This will reduce the number of parameters that need to be searched, speeding up the optimization phase, without impacting performance.

REFERENCES

[1] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1989.

[2] ——, "Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, pp. 389–406, 2004.

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

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

[5] D. Caromel and M. Leyton, "Fine tuning algorithmic skeletons," in *13th International Euro-Par Conference: Parallel Processing*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2007, vol. 4641, pp. 72–81.

[6] J. Reinders, *Intel Threading Building Blocks*. O'Reilly, 2007.

[7] M. Aldinucci and M. Torquati, "FastFlow website," 2011, http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about. [Online]. Available: http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about

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

[9] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[10] Z. Wang and M. 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*, ser. PPoPP '09, 2009, pp. 75–84.

[11] 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*, ser. ECML PKDD'10, 2010, pp. 7–23.

[12] 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*, ser. Advances in Parallel Computing, vol. 11, 2009, pp. 273–280.

[13] 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*, ser. PDP '10, 2010, pp. 195–199.

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

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

[16] Z. Wang and M. O'Boyle, "Partitioning streaming parallelism for multi-cores: A machine learning based approach," in *Proceedings of Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 307–318.

[17] 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*, 2011, pp. 676–687.

[18] 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)*, 2011, pp. 25–32.

[19] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A Language and Compiler for Algorithmic Choice," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 2009.

[20] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A static performance estimator to guide data partitioning decisions," in *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, 1991, pp. 213–223.

[21] E. Moss, P. Utgoff, J. Cavazos, D. Precup, D. Stefanovic, C. Brodley, and D. Scheeff, "Learning to schedule straight-line code," in *In Proceedings of Neural Information Processing Symposium*. MIT Press, 1997, pp. 929–935.

[22] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *Proceedings of Computing Frontiers (CF)*, 2007, pp. 131–142.

[23] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using machine learning to focus iterative optimization," in *In Proceedings of Code Generation and Optimization (CGO)*, 2006, pp. 295–305.

[24] M. R. Jantz and P. A. Kulkarni, "Eliminating false phase interactions to reduce optimization phase order search space," in *Proceedings of Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2010, pp. 187–196.