Title

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

Department name

by
Author
B.S

Acknowledgments

Table of Contents

List of Tables

List of Figures

Abstract

Abstract Blaze is a template based high performance C++ math library. Blaze provides four different backends for parallelization, one of which is HPX - a C++ library for concurrency and parallelism. Here we are suggesting to use a set of compile-time and run-time parameters to improve the performance of a Blaze when used with HPX backend.

Chapter 1. Chap1

1.1. Introduction

Linear algebra libraries like ATLAS, SPIRAL,... try to use hardware-specific optimizations to improve their performance. In this work, we are trying to optimize the performance based on the application parameters such as matrix size, operation, and data layout.

Scientific applications tend to contain a lot of task parallelism, performing same set of operations on different chunks of the data. Using a parallel for-loop for this purpose can lead to significant speed-ups.

Defining chunk as the group of iterations that would be assigned to a processor, loop scheduling methods propose different approaches for creation and assignment of these chunks to the processors.

Loop scheduling refers to different ways iterations could be assigned to the processors and the order of their execution. The main reason for performance degradation in loop scheduling is load imbalance, which refers to situations where different amount of work is assigned to different processors[1].

The simplest loop scheduling method is static scheduling, in which, the iterations are divided evenly among all the processors statically, either as a consecutive block -also called cyclic- or in a round-robin manner[2]. Since all the assignments happen at compile time or before execution of the application, this method imposes no runtime scheduling overhead. Several factors including interprocessor communication, cache misses, and page faults can lead to different execution times for different iterations, leading to load imbalance among the processors[3].

In the meanwhile, dynamic scheduling methods postpone the assignment to runtime, which tends to improve load balancing, at the cost of higher scheduling overhead. Some of dynamic scheduling methods include: Pure Self-scheduling, Chunk Self-scheduling, Guided Self-scheduling[4], Factoring[5] and Trapazoid Self-scheduling[6],[2].We briefly go over some of these loop scheduling techniques here.

In Pure Self-scheduling everytime a processors becomes idle, it fetches one loop iteration. This approach, while achieving a high load balance, imposes a considerable amount of scheduling overhead when we are dealing with a fine-grain workload, and a large number of iterations. Also frequent access to shared variables like loop index could lead to memory contention[2].

In order to decrease the high scheduling overhead of Pure Self-scheduling methods, Chunk Self-scheduling method assigns a certain number of iterations(called chunk size) to each idle processor. This method trades lower scheduling overhead with higher load imbalance. Selection of the chunk size plays a very important role in the performance, as so a large chunk size increases the scheduling overhead decreases and causes load imbalance, while a small chunk size increases memory contention and scheduling overhead[2].

As an adaptive loop scheduling technique, Guided Self-scheduling[4] divides the remaining number of iterations at each request evenly among the processors, and assigns it to the processor that made the request, while updating the number of remaining iterations. This causes larger number of iterations to be assigned to the processors at the beginning of the loop execution, which results in lower scheduling overhead. The number of iterations assigned to each processor decreases as it approaches to the end of the execution, generating tasks containing only one or two iterations, causing an increase in the scheduling overhead. In order to tackle this issue, a minimum number of chunks could be set to avoid creation of very small chunks[7].

Very similar to Guided Self-scheduling, Factoring also decreases the chunk size as the loop execution proceeds, with this difference that -dynamic -self-scheduling -factoring

talk about load balancing and work stealing

But each of these methods work well for specific problem. We are looking for a general solution which can automatically decide on the chunk size parameter to achieve the best performance.

1.2. Problem Statement

Importance of compile time configuration on task scheduling

1.2.1. HPX

HPX[8] is a C++ runtime system for parallel and distributed applications based on ParalleX execution model[9]. HPX contains 5 main modules: Performance Monitoring System, Local Control Objects(LCOs), Thread Scheduling System, Parcel Transport Layer, Active Global Address Space (AGAS).

1.3. Literature Review

Loop scheduling techniques has been extensively studied by different researchers. In [10] the authors propose a hybrid static/dynamic method for loop scheduling that improves the performance of dense matrix factorization, compared to both fully static and fully dynamic scheduling. The authors of [10], divide the dependency graph into two subgraphs, one of which is scheduled dynamically and the other one is scheduled statically. The tasks on the critical path are scheduled statically and each thread is forced to prioritize the static tasks[10]. They were able to improve data locality and scheduling overhead, while creating a more balanced workload.

[11] [12],[4],[5],[13]

The previous work on predicting the performance of a parallel application mainly focuses on three major types of models: analytical, trace-based, and empirical models[14].

The analytical models[15],[16],[17], while providing an arithmetic formula to represent the execution time of an application, require a deep understanding of the application, to apply platform-specific optimizations, and can not be generalized to different domains and architectures[18],[19],[20]. Traced-based models, on the other hand, use the traces collected through instrumentation, to predict the performance. These models, opposed to analytical models, do not rely on an expert's knowledge of the application, but while adding some overhead to the runtime, these models require a large storage space to save the traces, and are hard to interpret[19]. In empirical modeling, the results obtained from running an application with a set of parameters on a specific set of machines to build a model for unknown set of application and system parameters[14]. This type of modeling includes machine learning based approaches.

In [21], the authors use neural networks to predict the performance focusing on SMG2000 application, a parallel multigrid solver for linear systems[22], on two different platforms. Defining application parameters N_x, N_y, N_z, representing the working set size per processor, and P_x, P_y, P_z, describing the three-dimension processor topology, as the features, [21] uses a fully connected neural network to learn the model. Since they use absolute mean square error as the loss function, they use stratification to replicate samples with lower values by a factor which is proportional to their target value. They also apply bagging technique to decrease the variance in the model. As they increase the size of the training set to 5K points, they reach an error rate of 4.9%.

As a trace-based model, [19] analyzes the abstract syntax tree of the code and collects data through inserting special code for instrumentation when encounters 4 different situations, namely, assignments, branches, loops, and MPI communications. The authors then use 5 different machine learning methods including random forests, support vector machine, and ridge regression to build a prediction model from the collected data. Through applying two filtration processes, they were able to decrease the amount of overhead introduced along with the storage space requirement. Their results were inclined towards random forest, mainly because of the lower impact of categorical features on it, which is helpful in general cases where we do not have any knowledge about the type of features[19].

In [14] the authors investigate a set of machine learning techniques, including deep neural networks, support vector machine, decision tree, random forest, and k-nearest neighbor to predict the execution time of 4 different applications. Each of these applications require a certain set of features as input, for example, for the miniMD application in molecular dynamics, the number of processes and the number of atoms were considered as the input features, while for miniAMR, an application for studying adaptive mesh refinement, number of processes and also block sizes in x, y, and z direction, where used as the input features. While achieving promising results especially for deep neural networks, bagging, and boosting methods, [14] suggest utilizing transfer learning through deep neural networks to predict performance on other platforms.

[23] [24]

Although concentrating on GPUs, [25] proposes a lightweight machine learning based performance model to choose the number of threads to use for parallelization for a specific data size and operation. With the final goal of improving the training time in a neural network, [25] selects 4 performance features collected by hardware counters namely, number of CPU cycles, number of cache misses, cache accesses for the last cache level, and number of level 1 cache hits. Then they take two different approaches to build their model. In the first on they try 10 different regression models including random forest, and in the second one they use hill climbing algorithm to choose the number of threads. In addition to hardware independent, and not requiring the training process, hill climbing algorithm achieves a much higher accuracy compared to the best performing regression model.

In this paper, we suggest using machine learning to directly predict the optimal chunk-size to achieve the best performance instead of predicting the execution time or the optimal number of cores to run the application on. For this purpose, we have offered a set of general features that are not specific to an application and could easily be extracted at compile time or at run time. Once the data has been collected and our model has been created, the prediction results could be easily applied to a new application with a negligible overhead.

[19]

As another field to use machine learning, [26] collects seven runtime events and uses machine learning not to predict the performance, but to schedule the tasks. These events include, task creation, suspension, execution, completion, implicit/explicit barrier, parallel region, and finally loop/master/single region runtime events, collected through the OMPT using

ORA API. Experimenting with four different machine learning techniques, including support vector machine, random forest, neural networks, and naive bayes, they would select one specific task pool configuration out of the three pre-defined options as the final classification result. Testing this framework on a real life molecular dynamics application, they observed an up to 31% improvement in performance.

Compiler-based methods:

The authors of [27] propose using machine learning to predict the optimal number of threads, and also the optimal scheduling policy for running an OpenMP application. Through that, they were able to develop an automatic compiler-based method to map a parallel application to a multicore processor. They collect three type of features namely, code, data, and run-time features. Code features are extracted from the code directly, and they include cycles per instruction, number of branches, load and store instructions, and computations per instruction. While the code features could be collected statically at compile time, the data and run-time features are collected through low-cost profiling runs. This group of features include loop iteration count, branch miss rate, and $L1$ data cache miss rate. The authors of [27] then use an artificial neural network to predict the speedup achieved for a program with certain number of threads, and at the same time they use a support vector machine model to predict the best scheduling policy, out of block, cyclic, dynamic, and guided scheduling policies, for an unseen program.

[28] profiling information about the application on a given architecture [29],[30],[31]

Machine learning models [32], [33], [26]

[34]

1.4. Blaze

Blaze Math Library[35] is a C++ library for linear algebra. Blaze, based upon Expression Templates(ETs)[36], introduces "smart" expression templates(SETs)[35] to optimize the performance for array-based operations. Expression Templates[36] is an abstraction technique that uses overloaded operators in C++ to prevent creation of unnecessary temporaries, while evaluating arithmetic expressions, in order to improve the performance[35]. The ET-based approaches create a parse tree of the expression at compile time and postpone the actual evaluation to when the expression is assigned to a target.

Although being able to achieve promising performances for element-wise operations, these methods are not suitable for high performance computing for the following reasons. Due to their abstraction from both the data type and also the operation itself, they do not allow optimizations specific to the type of the arrays, alongside the operation[35]. As a solution, Blaze proposes smart ETs with these two main additions: integration with highly optimized compute kernels, and creation of intermediate temporaries when needed[35]. Some of the ET-based linear algebra libraries are: Blitz++[37], Boost uBLAS[38], MTL[39], and Eigen[40]. Among these libraries, Eigen, MTL, alongside Blaze, impose different conceptual changes to ETs in order to make them suitable for HPC.

As stated earlier, as an ET-based library, blaze performs the calculations when an expression is assigned to a target, which is implemented through the *blaze::Assign* function. Depending on the operation and the size of operands, this assignment could be parallelized through four different backends, namely, HPX, OpenMP[41], C++ threads, and Boost[42].

1.4.0.1. Blazemark

Blazemark is a benchmark suite provided by Blaze to compare the performance of Blaze with other linear algebra libraries.

In this experiment we were trying to reach the performance achieved by OpenMP backend. In order to do that we used the benchmarks offered by Blazemark. Starting from level 1 blas functions, we chose daxpy and dvecdvecadd benchmarks to start with. Implementation-wise we introduced two parameters namely block_size and chunk_size. chunk_size was used to specify the number of iterations of the for loop being executed by each thread, and vector_block_size denotes the number of consecutive elements from the vector on which we perform the operation at each iteration, and matrix _block_size represents a block of matrix selected at each iteration. We ran the benchmarks with different chunk_sizes and block_sizes.

The results suggested a range of values for chunk_size and block_size with which the results are improved rather than one optimum value.

1.4.0.2. Implementation of HPX Backend

Different backends are implemented in Blaze through a for-loop in which at each iteration a section of the vector or matrix is selected and the result of the operation is assigned to the corresponding section of the result. Each backend uses their own method for parallelizing this for loop. For HPX backend the HPX *parallel::for_loop* is used for this purpose. Listing **??** shows the modified implementation of HPX backend in Blaze.

1.4.0.2.1. HPX *for_loop* HPX *for_loop* takes an execution policy as first argument, which is set to *dynamic_chunk_size* execution policy in case of HPX backend for Blaze.

1.4.0.2.2. Intuition It's hard to write a code that performs very well for all the applications, here we are interested to make the whole process as automatic as possible(without interference of a human expert), so that scientists could run their applications which highly depend on linear algebra libraries.

1.5. Setup

Marvin: cache level 1 coherency line size: 64 number of sets: 512 ways of associativity: 8 type: Instruction size: 32K

cache level 2 coherency line size: 64 number of sets: 512 ways of associativity: 8 type: Unified size: 256K

cache level 3 coherency line size: 64 number of sets: 512 ways of associativity: 20 type: Unified size: 20480K

Trillian: cache level 1 coherency line size: 64 number of sets: 64 ways of associativity: 4 type: Data size: 16K

cache level 1 coherency line size: 64 number of sets: 512 ways of associativity: 2 type: Instruction size: 64K

cache level 2 coherency line size: 64 number of sets: 2048 ways of associativity: 16 type: Unified size: 2048K

Listing 1.1: Previous implementation of Assign function for HPX backend in Blaze.

```
template< typename MT1   // Type of the left-hand side dense matrix
, bool SO1       // Storage order of the left-hand side dense matrix
, typename MT2   // Type of the right-hand side dense matrix
, bool SO2       // Storage order of the right-hand side dense matrix
, typename OP >  // Type of the assignment operation
void hpxAssign( DenseMatrix<MT1,SO1>& lhs, const DenseMatrix<MT2,SO2>& rhs, OP op )
{
using hpx::parallel::for_loop;
using hpx::parallel::execution::par;

BLAZE_FUNCTION_TRACE;

using ET1 = ElementType_t<MT1>;
using ET2 = ElementType_t<MT2>;

constexpr bool simdEnabled( MT1::simdEnabled && MT2::simdEnabled && IsSIMDCombinable_v<ET1,ET2> );
constexpr size_t SIMDSIZE( SIMDTrait< ElementType_t<MT1> >::size );

const bool lhsAligned( (~lhs).isAligned() );
const bool rhsAligned( (~rhs).isAligned() );

const size_t threads     ( getNumThreads() );
const ThreadMapping threadmap( createThreadMapping( threads, ~rhs ) );

const size_t addon1      ( ( ( (~rhs).rows() % threadmap.first ) != 0UL )? 1UL : 0UL );
const size_t equalShare1( (~rhs).rows() / threadmap.first + addon1 );
const size_t rest1       ( equalShare1 & ( SIMDSIZE - 1UL ) );
const size_t rowsPerThread( ( simdEnabled && rest1 )?( equalShare1 - rest1 + SIMDSIZE ):( equalShare1 ) );

const size_t addon2      ( ( ( (~rhs).columns() % threadmap.second ) != 0UL )? 1UL : 0UL );
const size_t equalShare2( (~rhs).columns() / threadmap.second + addon2 );
const size_t rest2       ( equalShare2 & ( SIMDSIZE - 1UL ) );
const size_t colsPerThread( ( simdEnabled && rest2 )?( equalShare2 - rest2 + SIMDSIZE ):( equalShare2 ) );

for_loop( par, size_t(0), threads, [&](int i)
{
const size_t row    ( ( i / threadmap.second ) * rowsPerThread );
const size_t column( ( i % threadmap.second ) * colsPerThread );

if( row >= (~rhs).rows() || column >= (~rhs).columns() )
return;

const size_t m( min( rowsPerThread, (~rhs).rows()    - row    ) );
const size_t n( min( colsPerThread, (~rhs).columns() - column ) );

if( simdEnabled && lhsAligned && rhsAligned ) {
auto       target( submatrix<aligned>( ~lhs, row, column, m, n ) );
const auto source( submatrix<aligned>( ~rhs, row, column, m, n ) );
op( target, source );
}
else if( simdEnabled && lhsAligned ) {
auto       target( submatrix<aligned>( ~lhs, row, column, m, n ) );
const auto source( submatrix<unaligned>( ~rhs, row, column, m, n ) );
op( target, source );
}
else if( simdEnabled && rhsAligned ) {
auto       target( submatrix<unaligned>( ~lhs, row, column, m, n ) );
const auto source( submatrix<aligned>( ~rhs, row, column, m, n ) );
op( target, source );
}
else {
auto       target( submatrix<unaligned>( ~lhs, row, column, m, n ) );
const auto source( submatrix<unaligned>( ~rhs, row, column, m, n ) );
op( target, source );
}
} );
}
```

Listing 1.2: New implementation of Assign function for HPX backend in Blaze.

```
template< typename MT1   // Type of the left-hand side dense matrix
, bool SO1        // Storage order of the left-hand side dense matrix
, typename MT2    // Type of the right-hand side dense matrix
, bool SO2        // Storage order of the right-hand side dense matrix
, typename OP >   // Type of the assignment operation
void hpxAssign( DenseMatrix<MT1,SO1>& lhs, const DenseMatrix<MT2,SO2>& rhs, OP op )
{
using hpx::parallel::for_loop;
using hpx::parallel::execution::par;

BLAZE_FUNCTION_TRACE;

using ET1 = ElementType_t<MT1>;
using ET2 = ElementType_t<MT2>;

constexpr bool simdEnabled( MT1::simdEnabled && MT2::simdEnabled && IsSIMDCombinable_v<ET1,ET2> );
constexpr size_t SIMDSIZE( SIMDTrait< ElementType_t<MT1> >::size );

const bool lhsAligned( (~lhs).isAligned() );
const bool rhsAligned( (~rhs).isAligned() );

const size_t threads    ( getNumThreads() );
const size_t numRows ( min( static_cast<std::size_t>( BLAZE_HPX_MATRIX_BLOCK_SIZE_ROW ), (~rhs).rows() ) );
const size_t numCols ( min( static_cast<std::size_t>( BLAZE_HPX_MATRIX_BLOCK_SIZE_COLUMN ), (~rhs).columns() ) );

const size_t rest1       ( numRows & ( SIMDSIZE - 1UL ) );
const size_t rowsPerIter( ( simdEnabled && rest1 )?( numRows - rest1 + SIMDSIZE ):( numRows ) );
const size_t addon1      ( ( ( (~rhs).rows() % rowsPerIter ) != 0UL )? 1UL : 0UL );
const size_t equalShare1( (~rhs).rows() / rowsPerIter + addon1 );

const size_t rest2       ( numCols & ( SIMDSIZE - 1UL ) );
const size_t colsPerIter( ( simdEnabled && rest2 )?( numCols - rest2 + SIMDSIZE ):( numCols ) );
const size_t addon2      ( ( ( (~rhs).columns() % colsPerIter ) != 0UL )? 1UL : 0UL );
const size_t equalShare2( (~rhs).columns() / colsPerIter + addon2 );

hpx::parallel::execution::dynamic_chunk_size chunkSize ( BLAZE_HPX_MATRIX_CHUNK_SIZE );

for_loop( par.with( chunkSize ), size_t(0), equalShare1 * equalShare2, [&](int i)
{
const size_t row    ( ( i / equalShare2 ) * rowsPerIter );
const size_t column( ( i % equalShare2 ) * colsPerIter );

if( row >= (~rhs).rows() || column >= (~rhs).columns() )
return;

const size_t m( min( rowsPerIter, (~rhs).rows()    - row     ) );
const size_t n( min( colsPerIter, (~rhs).columns() - column ) );

if( simdEnabled && lhsAligned && rhsAligned ) {
auto         target( submatrix<aligned>( ~lhs, row, column, m, n ) );
const auto source( submatrix<aligned>( ~rhs, row, column, m, n ) );
op( target, source );
}
else if( simdEnabled && lhsAligned ) {
auto         target( submatrix<aligned>( ~lhs, row, column, m, n ) );
const auto source( submatrix<unaligned>( ~rhs, row, column, m, n ) );
op( target, source );
}
else if( simdEnabled && rhsAligned ) {
auto         target( submatrix<unaligned>( ~lhs, row, column, m, n ) );
const auto source( submatrix<aligned>( ~rhs, row, column, m, n ) );
op( target, source );
}
else {
auto         target( submatrix<unaligned>( ~lhs, row, column, m, n ) );
const auto source( submatrix<unaligned>( ~rhs, row, column, m, n ) );
op( target, source );
}
} );
}
```

cache level 3 coherency line size: 64 number of sets: 2048 ways of associativity: 48 type: Unified size: 6144K

1.6. methods

To start, after collecting the data I tried to build a simple regression model to predict the performance based on the input features, only for the 'dmatdmatadd' benchmark.

features:

option1: Random Forest

option2: neural networks
option3: XGBoost

Chapter 2. Chap2

2.0.1. Experiments

In order to capture the relationship between number of cores, chunk_size, block_size, and the performance, we ran a series of experiments with different values for first three parameters and measured the number of floating point operations per second performed.

For these experiments ,at the first step we selected the *DMatDMatADD* benchmark which was implemented in Blazemark. *DMatDMatADD* benchmark is a level 3 BLAS function to perform matrix-matrix addition in the form of $A = B + C$, where A, B, C are square matrices of the same size.

To avoid adding the scheduling overhead for small matrix sizes, Blaze uses a threshold to start parallelization, which is specific to the type of operation. For matrix-matrix addition, if the number of elements in the matrix is greater than 36100 elements(which is equivalent to a square matrix size of size 190 by 190) Blaze uses the configured backend to parallelize the assignment operation. For this reason, we start our experiments with matrix size of 200x200 and gradually increase the size to 1024x1024.

2.0.2. L2 cache miss analysis

In this set of experiments we used the performance counters integrated into HPX to measure the cache misse rate for different grain_sizes w
-grain_size

Chapter 3. Chap3

Chapter 4. Chap4

Chapter 5. Chap5

Chapter 6. Chap6

References

[1] Florina M Ciorba, Christian Iwainsky, and Patrick Buder. Openmp loop scheduling revisited: making a case for more schedules. In *International Workshop on OpenMP*, pages 21–36. Springer, 2018.

[2] Jie Liu, Vikram A Saletore, and Ted G Lewis. Safe self-scheduling: a parallel loop scheduling scheme for shared-memory multiprocessors. *International Journal of Parallel Programming*, 22(6):589–616, 1994.

[3] Teebu Philip. *Increasing chunk size loop scheduling algorithms for data independent loops*. PhD thesis, Citeseer, 1995.

[4] Constantine D Polychronopoulos and David J Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *Ieee transactions on computers*, 100(12):1425–1439, 1987.

[5] Susan Flynn Hummel, Edith Schonberg, and Lawrence E Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–102, 1992.

[6] Ten H Tzen and Lionel M Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on parallel and distributed systems*, 4(1):87–98, 1993.

[7] David J Lilja. Exploiting the parallelism available in loops. *Computer*, 27(2):13–26, 1994.

[8] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 6. ACM, 2014.

[9] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *2009 International Conference on Parallel Processing Workshops*, pages 394–401. IEEE, 2009.

[10] Simplice Donfack, Laura Grigori, William D Gropp, and Vivek Kale. Hybrid static/dynamic scheduling for already optimized dense matrix factorization. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 496–507. IEEE, 2012.

[11] Liping Xue, M Kandemir, Guilin Chen, Feihui Li, Ozcan Ozturk, Rajaraman Ramanarayanan, and Balaji Vaidyanathan. Locality-aware distributed loop scheduling for chip multiprocessors. In *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, pages 251–258. IEEE, 2007.

[12] Peiyi Tang and Pen-Chung Yew. Processor self-scheduling for multiple-nested parallel loops. In *ICPP*, volume 86, pages 528–535, 1986.

[13] Clyde P. Kruskal and Alan Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software engineering*, (10):1001–1016, 1985.

[14] Preeti Malakar, Prasanna Balaprakash, Venkatram Vishwanath, Vitali Morozov, and Kalyan Kumaran. Benchmarking machine learning methods for performance modeling of scientific applications. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 33–44. IEEE, 2018.

[15] Filip Blagojevic, Xizhou Feng, Kirk W Cameron, and Dimitrios S Nikolopoulos. Modeling multigrain parallelism on heterogeneous multi-core processors: a case study of the cell be. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 38–52. Springer, 2008.

[16] Darren J Kerbyson, Henry J Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J Wasserman, and Mike Gittings. Predictive performance and scalability modeling of a large-scale application. In *SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pages 39–39. IEEE, 2001.

[17] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[18] Benjamin C Lee, David M Brooks, Bronis R de Supinski, Martin Schulz, Karan Singh, and Sally A McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 249–258. ACM, 2007.

[19] Jingwei Sun, Shiyan Zhan, Guangzhong Sun, and Yong Chen. Automated performance modeling based on runtime feature detection and machine learning. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pages 744–751. IEEE, 2017.

[20] Sabri Pllana, Ivona Brandic, and Siegfried Benkner. Performance modeling and prediction of parallel and distributed computing systems: A survey of the state of the art. In *First International Conference on Complex, Intelligent and Software Intensive Systems (CISIS'07)*, pages 279–284. IEEE, 2007.

[21] Engin Ipek, Bronis R De Supinski, Martin Schulz, and Sally A McKee. An approach to performance prediction for parallel applications. In *European Conference on Parallel Processing*, pages 196–205. Springer, 2005.

[22] Robert D Falgout and Ulrike Meier Yang. hypre: A library of high performance preconditioners. In *International Conference on Computational Science*, pages 632–641. Springer, 2002.

[23] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 116–125. IEEE, 2011.

[24] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 2–13. ACM, 2004.

[25] Jiawen Liu, Dong Li, Gokcen Kestor, and Jeffrey Vetter. Runtime concurrency control and operation scheduling for high performance neural network training. *arXiv preprint arXiv:1810.08955*, 2018.

[26] Ahmad Qawasmeh, Abid M Malik, and Barbara M Chapman. Adaptive openmp task scheduling using runtime apis and machine learning. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 889–895. IEEE, 2015.

[27] Zheng Wang and Michael FP O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *ACM Sigplan notices*, volume 44, pages 75–84. ACM, 2009.

[28] Jan Treibig, Georg Hager, and Gerhard Wellein. Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering. In *European Conference on Parallel Processing*, pages 451–460. Springer, 2012.

[29] Rosario Cammarota, Alexandru Nicolau, and Alexander V Veidenbaum. Just in time load balancing. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 1–16. Springer, 2012.

[30] Yun Zhang, Michael Voss, and ES Rogers. Runtime empirical selection of loop schedulers on hyperthreaded smps. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10–pp. IEEE, 2005.

[31] Peter Thoman, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer. Automatic openmp loop scheduling: a combined compiler and runtime approach. In *International Workshop on OpenMP*, pages 88–101. Springer, 2012.

[32] Karan Singh, Major Bhadauria, and Sally A McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

[33] Albert Y. Zomaya and Yee-Hwei Teh. Observations on using genetic algorithms for dynamic load-balancing. *IEEE transactions on parallel and distributed systems*, 12(9):899–911, 2001.

[34] Jiangtian Li, Xiaosong Ma, Karan Singh, Martin Schulz, Bronis R de Supinski, and Sally A McKee. Machine learning based online performance prediction for runtime parallelization and task scheduling. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–100. IEEE, 2009.

[35] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. Expression templates revisited: a performance analysis of current methodologies. *SIAM Journal on Scientific Computing*, 34(2):C42–C69, 2012.

[36] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[37] Blitz++ Library. `http://www.oonumerics.org/blitz/`.

[38] Boost uBLAS Library. http://www.boost.org/doc/libs/1_45_0/libs/numeric/ublas/doc/index.htm.

[39] MTL4 Library. http://www.simunova.com/de/mtl4.

[40] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 2010.

[41] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*, (1):46–55, 1998.

[42] Boost C++ Framework. https://www.boost.org.

Appendix A. Copyright Information

Vita

VITA