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. 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.

1.1. Problem Statement

Importance of compile time configuration on task scheduling

Chapter 2. Background

2.1. Loop Scheduling

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.

2.2. Asynchronous Many Task Runtime Systems

2.3. 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).

2.4. Blaze

Blaze Math Library[10] is a C++ library for linear algebra. Blaze, based upon Expression Templates(ETs)[11], introduces "smart" expression templates(SETs)[10] to optimize the performance for array-based operations. Expression Templates[11] 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[10]. 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[10]. As a solution, Blaze proposes smart ETs with these three main additions: integration with architecture-specific highly optimized compute kernels, creation of intermediate temporaries when needed, and selecting optimal evaluation method automatically for compound expressions[10].

Some of the ET-based linear algebra libraries are: Blitz++[12], Boost uBLAS[13], MTL[14], and Eigen[15]. Among these libraries, Eigen, MTL, alongside Blaze, impose different conceptual changes to ETs in order to make them suitable for HPC.

2.5. Task Granularity

Defining the grain size as the amount of work assigned to one HPX thread, Grubel[16] studies the effect of grain size on the execution time for a fixed number of cores. The results show that, for small grain sizes the overhead of creating the tasks, and for large grain sizes the starvation, is the dominant factor affecting the execution time[16]. When grain size is small, to perform same amount of work, higher number of tasks is created, and there is an overhead associated with creation of each task. Although this overhead is very small (order of

microseconds), when the amount of work performed by each thread is also small, this overhead becomes significant. A the grain size increases, these overheads are amortized by the time it takes to execute the task.

On the other hand, when grain size is increases, the number of tasks being created decreases, up until a point where the number of tasks being created is smaller than the number of cores. At this point another factor would interfere with the performance, which is referred to as starvation. Starvation happens where a large amount of work is assigned to some of the cores while the other cores are idling. At this point we are not using our resources efficiently.

While overheads of creating tasks degrades the performance for small grain sizes and starvation causes the execution time to increase for large grain sizes, there is a region in between where changing the grain size does not affect the performance.

2.6. Modeling performance

Gunther states the factors involved in creating overheads as exchanging data between memory and processors, waiting for completion of a memory access or an I/O,

-
-

2.6.1. Universal Scalibility Law

Amdahl's law[17], states that the amount of achievable speed up by adding more processors when running a parallel application, is restricted by the amount of code that could actually be parallelized. Equation 2.1, shows the relationship between speedup and number of processors, where σ is the serial fraction of the execution time, based on Amdahl's law[18].

$$S(p) = \frac{p}{1+\sigma(p-1)} \tag{2.1}$$

On the other hand, Gunther[18] extends Amdahl's law by incorporating the effect of three factors, namely concurrency, contention, and coherency, as shown in Equation 2.2.

$$S(p) = \frac{p}{1 + \sigma(p-1) + \kappa p(p-1)} \tag{2.2}$$

Concurrency(p) represents the linear speedup that could have been achieved if no interaction existed among the processors, contention(σ) represents the serialization effect of shared writable data, and finally coherency or data consistency(κ) represents the effort that needs to be made for keeping shared writable data consistent[18].

Figure 2.1 shows an example of the ideal linear speedup we expect to see when increasing the number of the processors, against the actual achievable speedup based on Amdahl's law and USL.

Equation 2.3 generalizes Equation 2.2 to represent the throughput by adding another parameter(γ) to represent the serial throughput.

$$X(p) = \frac{\gamma p}{1 + \sigma(p-1) + \kappa p(p-1)} \tag{2.3}$$

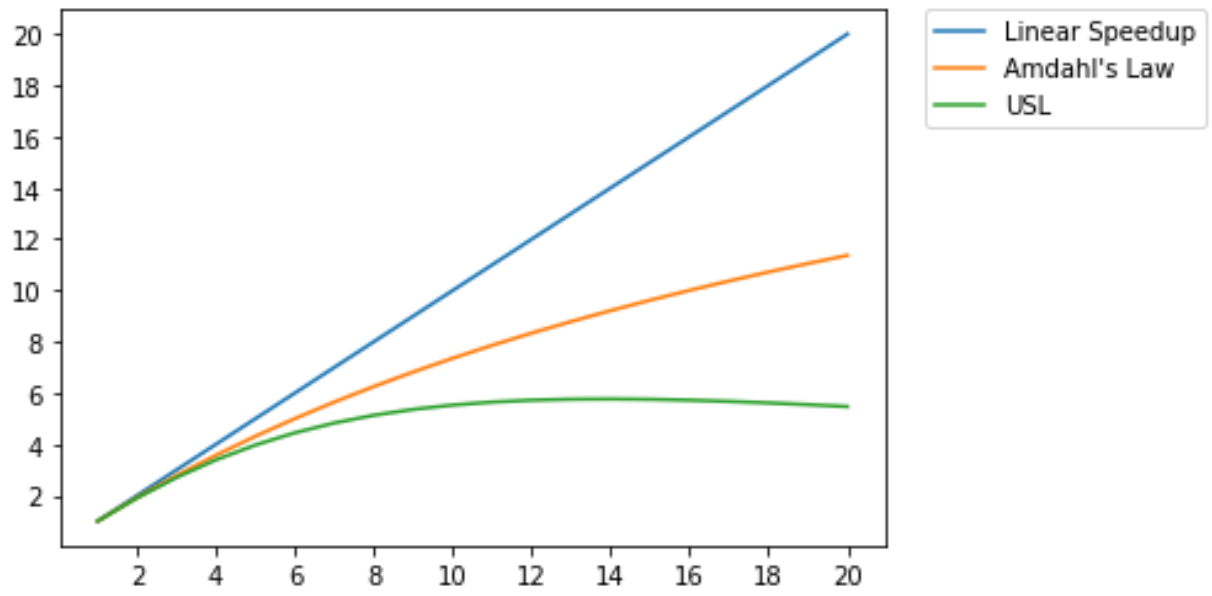


Figure 2.1. An example of the achievable speedup based on Amdahl's law and USL compared to the ideal linear speedup where $\sigma = 0.04$ and $\kappa = 0.005$.

Universal scalibility law also suggests that for some values of σ and κ there could be a certain number of processors that yield to maximum performance[18]. Increasing the number of processor beyond that point would only cause performance degradation.

2.6.2. Other Models

There are a few other models that have also been suggested to simulate the scalibility. Among which, Geometric model

Chapter 3. Literature Review

3.1. Literature Review

Loop scheduling techniques has been extensively studied by different researchers. In [19] 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 [19], 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[19]. They were able to improve data locality and scheduling overhead, while creating a more balanced workload.

[20] [21],[4],[5],[22]

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[23].

The analytical models[24],[25],[26], 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[27],[28],[29]. 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[28]. 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[23]. This type of modeling includes machine learning based approaches.

In [30], the authors use neural networks to predict the performance focusing on SMG2000 application, a parallel multigrid solver for linear systems[31], 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, [30] 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, [28] 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[28].

In [23] 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, [23] suggest utilizing transfer learning through deep neural networks to predict performance on other platforms.

[32] [33]

Although concentrating on GPUs, [34] 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, [34] 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.

[28]

As another field to use machine learning, [35] 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 [36] 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 runtime 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 [36] 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.

[37] profiling information about the application on a given architecture [38],[39],[40]

Machine learning models [41], [42], [35]

[43]

Chapter 4. Method

4.0.1. Blazemark

Blazemark is a benchmark suite provided by Blaze to compare the performance of Blaze with other linear algebra libraries including Blitz++[12], Boost uBLAS[44], GMM++[45], Armadillo[46], MTL4[14], and Eigen3[47], alongside plain BLAS libraries like Atlas[48], Goto[49], and Intel MKL.[50]

```
Dense Vector/Dense Vector Addition:
  C-like implementation [MFlop/s]:
    100         1115.44
    10000000    206.317
  Classic operator overloading [MFlop/s]:
    100         415.703
    10000000    112.557
  Blaze [MFlop/s]:
    100         2602.56
    10000000    292.569
  Boost uBLAS [MFlop/s]:
    100         1056.75
    10000000    208.639
  Blitz++ [MFlop/s]:
    100         1011.1
    10000000    207.855
  GMM++ [MFlop/s]:
    100         1115.42
    10000000    207.699
  Armadillo [MFlop/s]:
    100         1095.86
    10000000    208.658
  MTL [MFlop/s]:
    100         1018.47
    10000000    209.065
  Eigen [MFlop/s]:
    100         2173.48
    10000000    209.899
```

```
N=100, steps=55116257
  C-like      = 2.33322  (4.94123)
  Classic     = 6.26062  (13.2586)
  Blaze       = 1        (2.11777)
  Boost uBLAS = 2.4628   (5.21565)
  Blitz++     = 2.57398  (5.4511)
  GMM++       = 2.33325  (4.94129)
  Armadillo   = 2.3749   (5.0295)
  MTL         = 2.55537  (5.41168)
  Eigen       = 1.19742  (2.53585)
N=10000000, steps=8
  C-like      = 1.41805  (0.387753)
  Classic     = 2.5993   (0.710753)
  Blaze       = 1        (0.27344)
  Boost uBLAS = 1.40227  (0.383437)
  Blitz++     = 1.40756  (0.384884)
  GMM++       = 1.40862  (0.385172)
  Armadillo   = 1.40215  (0.383403)
  MTL         = 1.39941  (0.382656)
  Eigen       = 1.39386  (0.381136)
```

Figure 4.1. An example of the results obtained from running $DVECDVECADD$ benchmark through Blazemark

4.1. Parallelization in Blaze

Depending on the operation and the size of operands, this assignment could be parallelized through four different backends, namely, HPX, OpenMP[51], C++ threads, and Boost[44]. Table 4.1 shows the default value for some of the threshold for parallelization applied to operations performed in Blaze. It should be noted that these thresholds should be tuned based on the parallelization backend and also the system architecture.

Benchmark	Array size
$DVECDVCEADD, DVECDVECMULT$	38000
$DMATDMATADD$	36100 elements equivalent to a 175×175 matrix
$DMATDMATMULT$	3025 elements equivalent to a 55×55 matrix

Table 4.1. List of some of the thresholds applied to the operations performed by Blaze, starting from which the operation is executed in parallel

4.1.1. Implementation of HPX Backend

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.

The four mentioned backends, parallelize this assignment process through a parallel for-loop, in which at each iteration a specific section of each of the vectors or matrices(called a block) is selected and assigned to a core. Each core then performs the operation on the block they have been assigned to.

Each backend uses their own method for parallelizing this for loop. For HPX backend, current implementation uses a HPX *parallel::for_loop* with static chunking policy and chunk size of 1. This way, knowing the number of cores to run the application on, we can divide the original matrix equally among the cores, while the order of assignment of blocks to the cores is known at compile time. Listings4.1 shows the current implementation of the HPX backend in Blaze.

What we suggest here is that, some prior knowledge for example, architecture of the system we are running the application on, the expression that has to be executed, number of cores of the system, size and type of the arrays we are dealing with, and etc. should be able to help us to achieve a higher performance. For this purpose we introduced two parameters block_size and chunk_size.

Listing 4.1: 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 4.2: 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 );
}
} );
}
```

4.1.2. 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.

4.2. 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 of these 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 of size 190×190) 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 1587×1587. Table 4.2 show the matrix sizes and the number of cores chosen for our experiments with $DMATDMATADD$ benchmark.

Matrix sizes	200, 230, 264, 300, 396, 455, 523, 600, 690, 793, 912, 1048, 1200, 1380, 1587
Number of cores	1, 2, 3, 4, 5, 6, 7, 8
Number of rows in the block	4, 8, 12, 16, 20, 32
Number of columns in the block	64, 128, 256, 512, 1024
Chunk size	Between 1 and total number of blocks (logarithmic increase)

Table 4.2. List of different values used for each variable for running the $DMATDMATADD$ benchmark

Figure 4.2 shows the results of running $DMatDMatADD$ benchmark for matrix sizes and number of cores listed in Tablename based on grain size.

On the other hand, Figure 4.3 integrates the results obtained from running the same benchmark with different matrix sizes. Each color in this graph represents a specific matrix size.

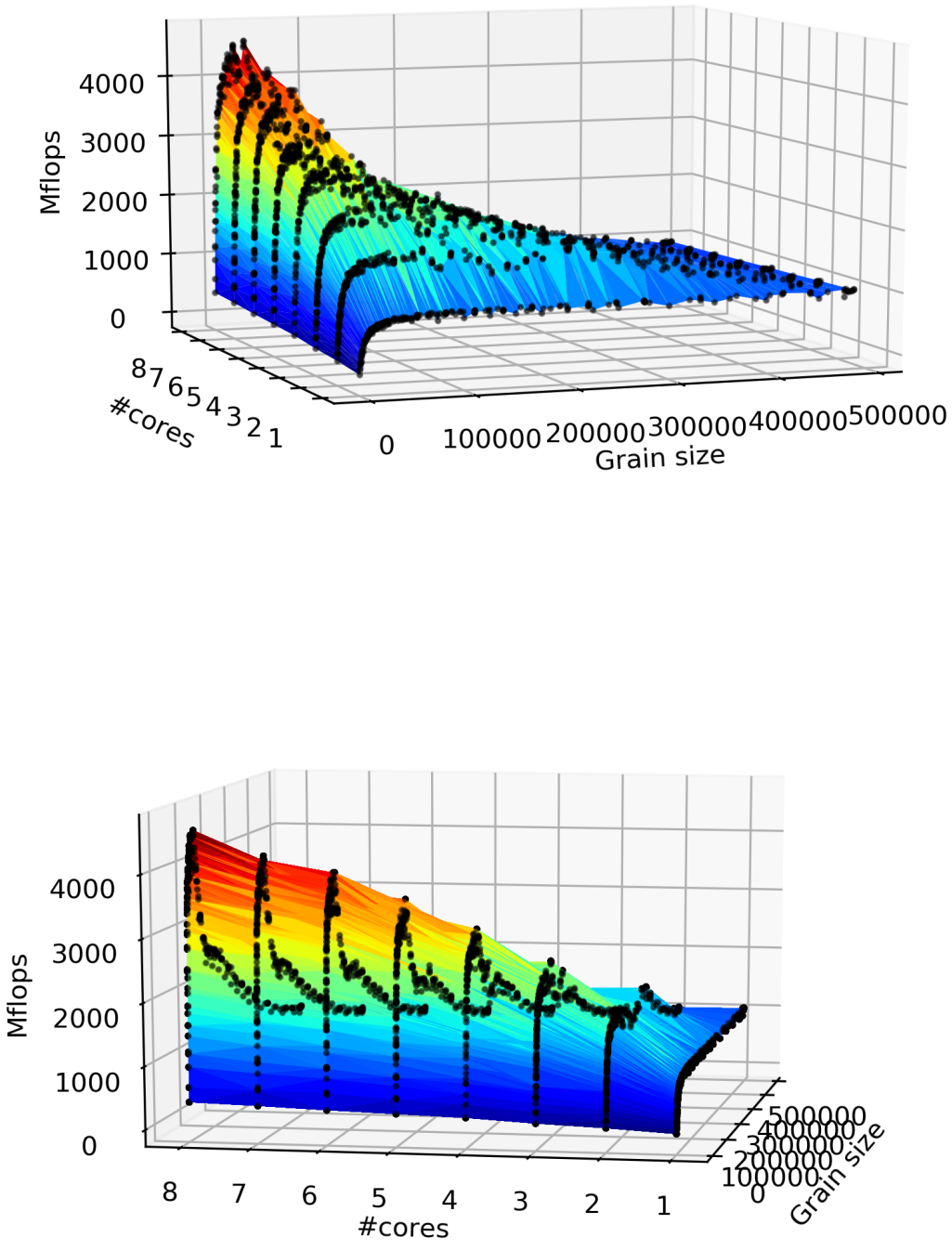


Figure 4.2. The results obtained from running $DMATDMATADD$ benchmark through Blazemark for matrix of size 690×690 from two different angles

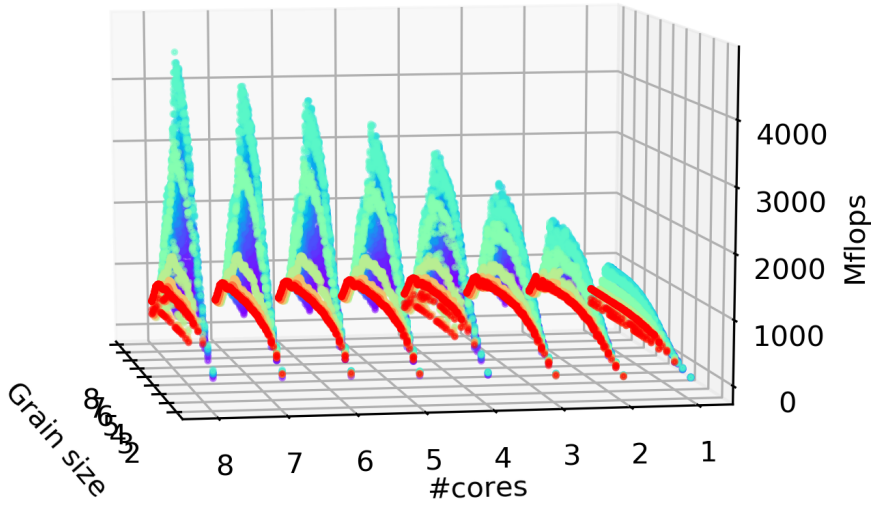


Figure 4.3. The results obtained from running $DMATDMATADD$ benchmark through Blazemark for matrix sizes from 200×200 to 1587×1587

4.2.1. Observation

The final purpose of our experiments is to find a chunk size that gives us the best performance for a given matrix size on a given machine. This chunk size should also be tailored to the expression being executed, and this all is based on assuming that we have already fixed the block size. So the first step appeared to be selecting the block size. For this purpose, we ran the experiments with a selection of block sizes as shown in Table 4.2.

It should be mentioned that there were three constraints on selecting the block sizes. First, Blaze forces the number of columns in a raw-major matrix to be divisible to SIMD register size in order to be able to take advantage of vectorization. Second, we have selected the number of columns in our blocks to be either divisible by cache line or to contain all the columns of the matrix.

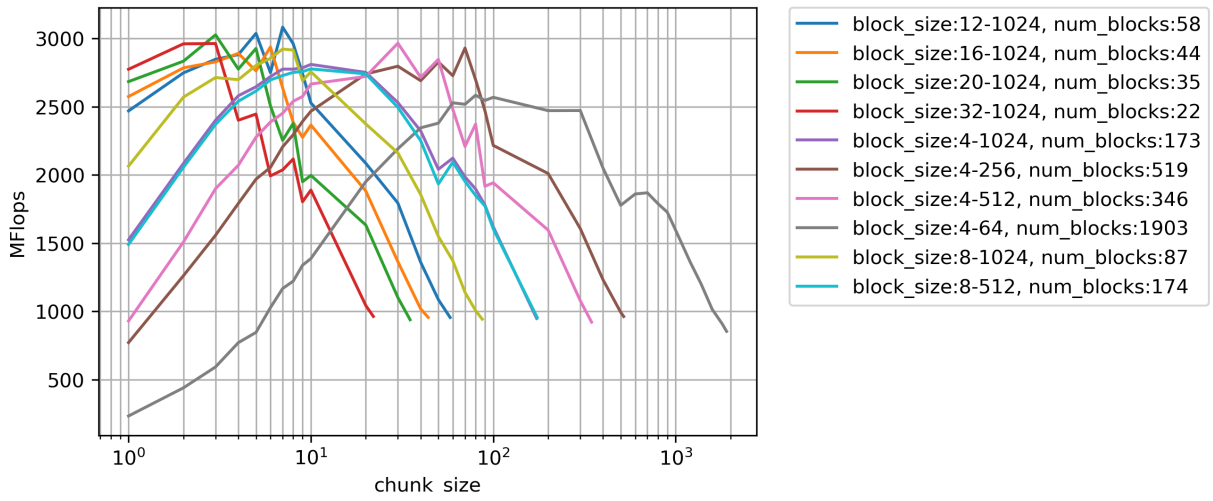


Figure 4.4. The results obtained from running $DMATDMATADD$ benchmark through Blazemark for matrix sizes from 690×690 with different combinations of block size and chunk size on 4 cores

The collected data, as seen in Figure 4.4, suggests two main points:

- For each selected block size, there is a range of chunk sizes that gives us the best performance.
- Except for some uncommon cases, no matter which block size we choose, we are able to achieve the maximum performance if we select the right chunk size.

This motivated us to move our search parameter from chunk size to grain size. As stated earlier, grain size is the amount of work assigned to one HPX thread. Here we represent grain size by number of floating point operations performed by a HPX thread. For example, performing addition among two matrices, if we choose the block size as 4×64 and chunk size as 3, the grain size would be $3 \times 4 \times 64 = 768$. Note that in our experiments whenever the number of columns of the original matrix is not divisible to the selected number of columns for block size, there would be a set of blocks with less number of elements than the selected block size, this has been considered when calculating the grain size.

By changing our focus to the grain size instead of the block size and the chunk size, Figure 4.5 shows how the throughput changes with regards to the grain size for the $DMATDMATADD$

benchmark, for each specific block size. Each combination of block size and chunk size generates a point in the graph. On the other hand, Figure 4.8 looks at these graphs from another aspect, keeping the problem size constant but changing the number of the cores to run the benchmark on, instead.

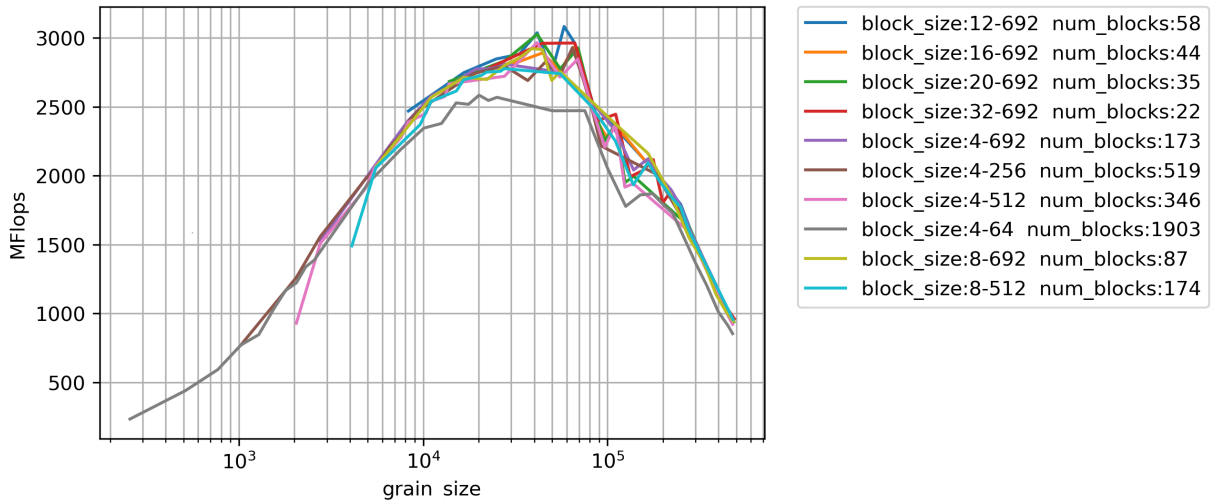


Figure 4.5. The results obtained from running $DMATDMATADD$ benchmark through Blazemark for matrix size 690×690 on 4 cores

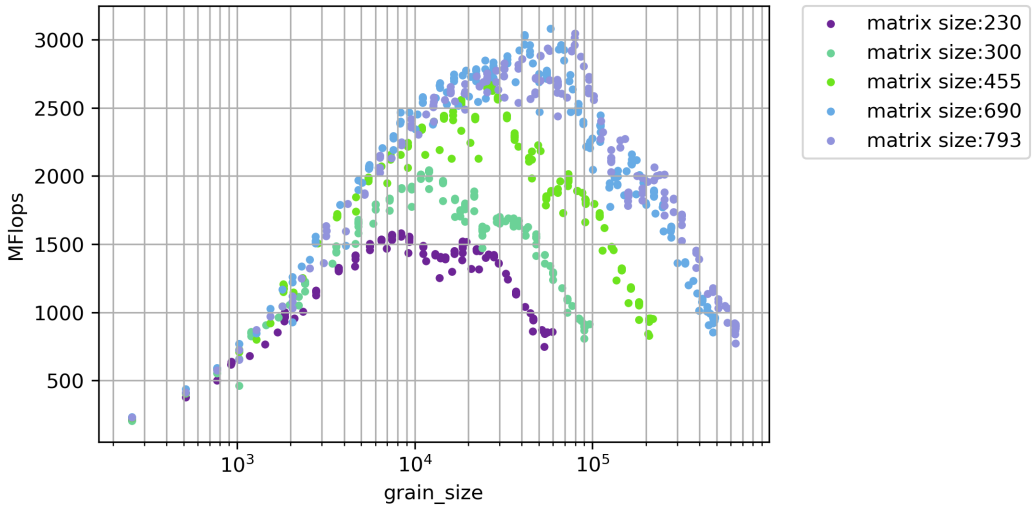
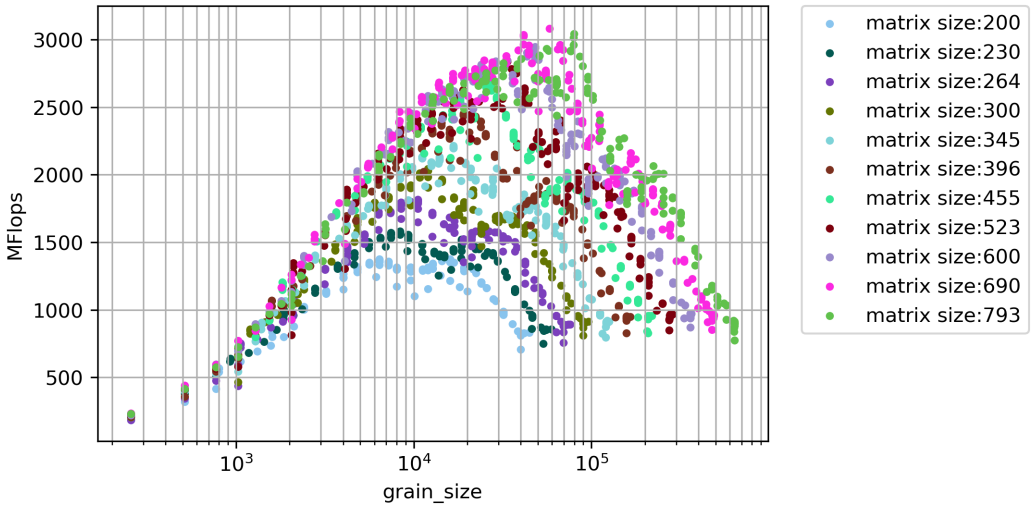
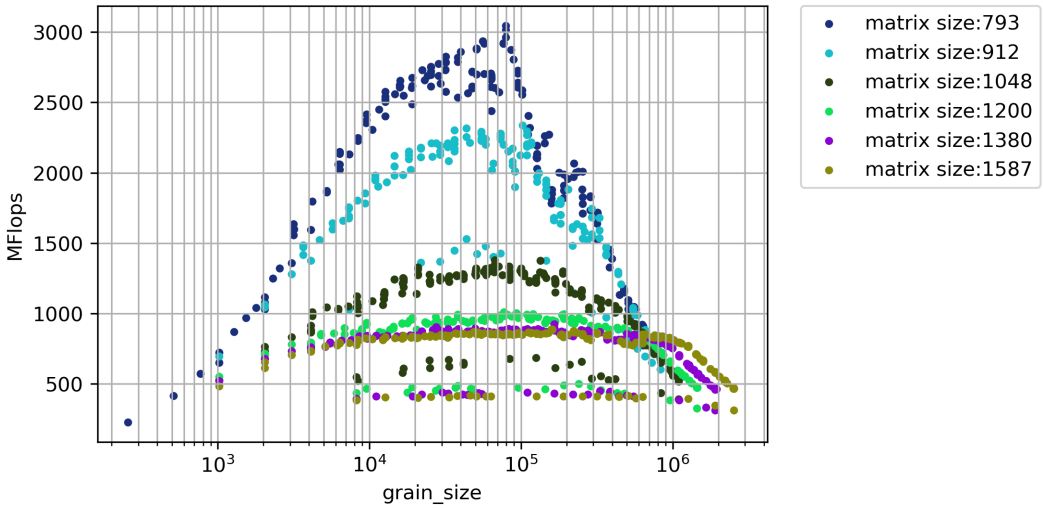


Figure 4.6. The results obtained from running $DMATDMATADD$ benchmark through Blazemark for 5 different matrix sizes on 4 cores



(a)



(b)

Figure 4.7. Throughput vs. grain size graph obtained from running *DMATDMATADD* benchmark on 4 cores for matrix sizes (a) smaller than 793×793 and (b) larger than 793×793

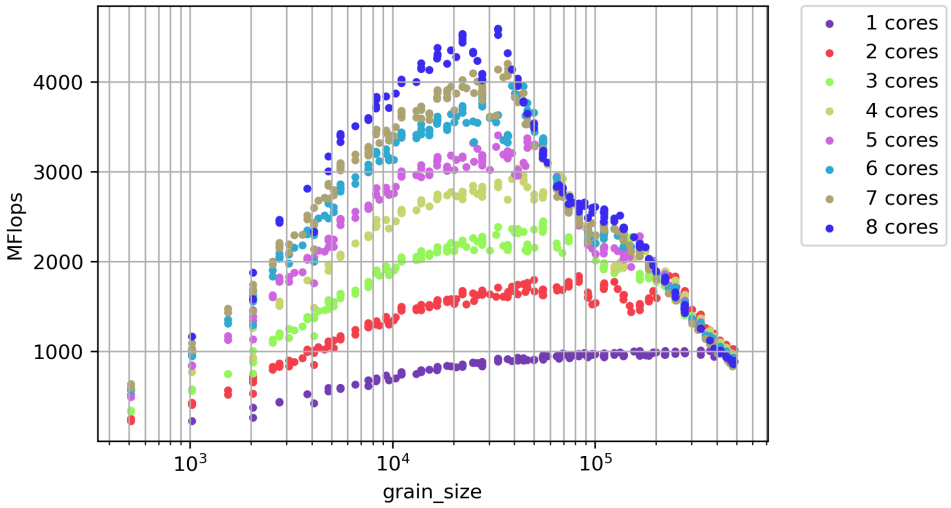


Figure 4.8. The results obtained from running $DMATDMATADD$ benchmark through Blazemark for matrix size 690×690 on different number of cores

4.3. Method

Looking at the throughput vs. grain size graphs and the consistent pattern observable motivated us to try to model the relationship between throughput and grain size. In order to simplify the process and eliminate the effect of different possible factors, we started with limiting the problem to a fixed matrix size.

4.3.1. Quadratic Fit

In our first attempt we used a 2nd degree polynomial to model throughput against grain size. For each matrix size, we fitted the corresponding graphs shown in Figure 4.7 to a second degree polynomial.

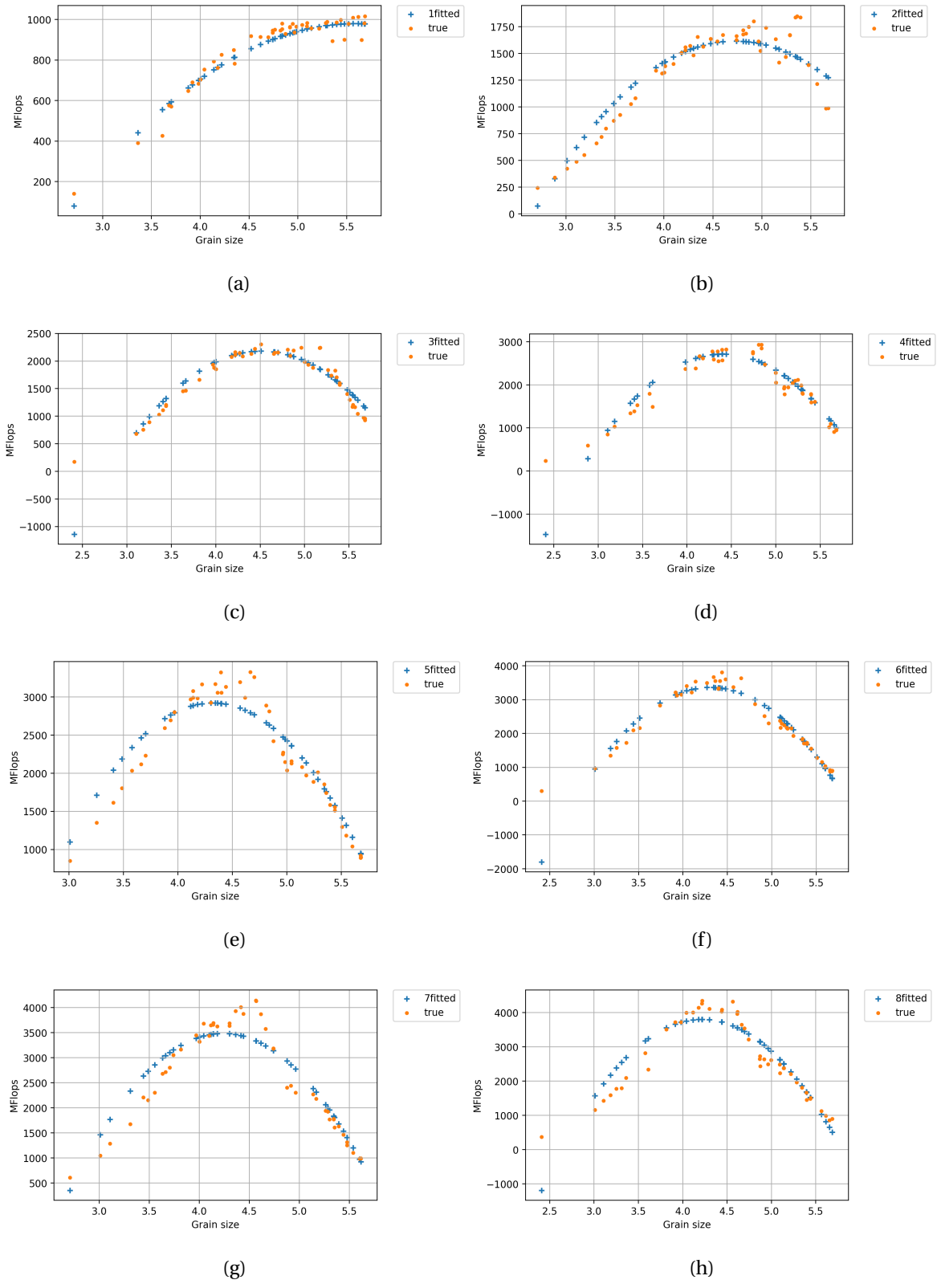


Figure 4.9. The results of fitting the throughput vs grain size data into a 2d polynomial for $DMATDMATADD$ benchmark for matrix size 690×690 with different number of cores on the test data set (a) 1 core, (b) 2 cores, (c) 3 cores, (d) 4 cores, (e) 5 cores, (f) 6 cores, (g) 7 cores, (h) 8 cores

Figure 4.9 shows the results of using a quadratic function to fit the data for one matrix size with different number of threads.

For our experiment, we divided the data into two sections, training and test. 60% of the data was randomly chosen for the training part and the rest was considered as the test set. The training set was used to find the best 2nd degree polynomial for the data, and once the parameters were identified, the generated 2nd degree polynomial was applied to the test set to measure how good our fit was performing.

For the matrix size 690 × 690 our dataset contained 117 data points, 72 of which was randomly selected to build the model. The mean relative error for each number of cores, calculated using Equation 4.1, is represented in Figure 4.10 for training and test set. In this equation, t_i and p_i denote the true value and the predicted value of the ith sample respectively, where n is the number of samples with the particular number of cores.

$$error = \frac{1}{n}\sum_{i=1}^{n} 1 - p_i/t_i \tag{4.1}$$

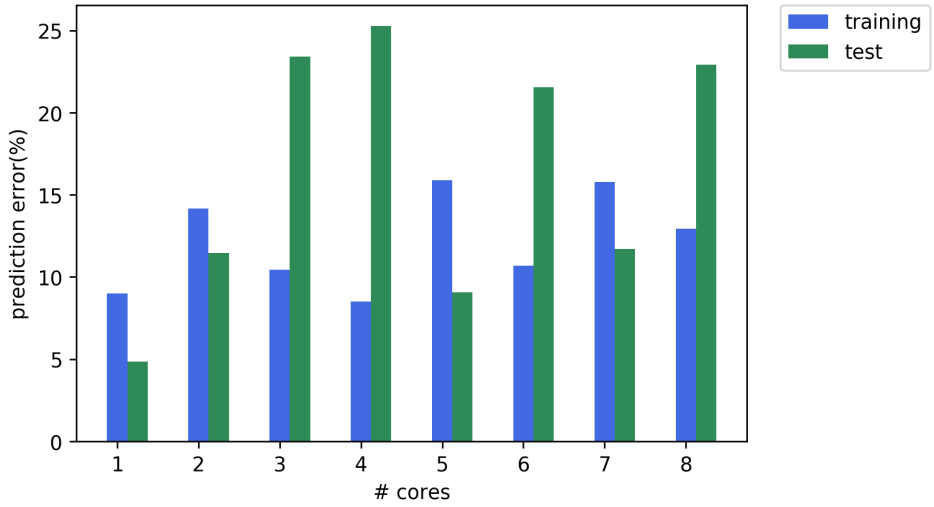
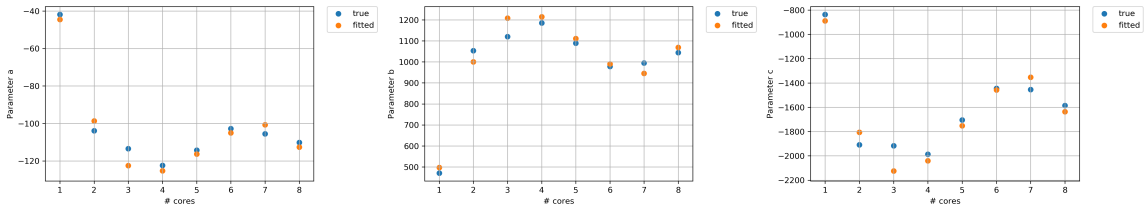


Figure 4.10. The training and test error for fitting data obtained from the $DMATDMATADD$ benchmark for matrix size 690 × 690 against different number of cores cores

4.3.2. Generalizing the fitted function to include number of cores

In this step, we try to generalize the fitted 2nd degree polynomial obtained from the previous step, represented by $P = ag^2 + bg + c$, where P is the throughput and g is the grain size,by looking at how the three parameters a, b, and c change when number of cores changes. A 3rd degree polynomial seems to a reasonable fit for each of these parameters, in regards to number of cores. In order to avoid overfitting, we excluded two of the data points(2 and 5) from the data points used for fitting the polynomial and tested the fitted function on those two points to see how well the function is working on unseen data points.



(a) (b) (c)

Figure 4.11. Fitting the parameters of the quadratic function with a 3rd degree polynomial from the $DMATDMATADD$ benchmark for matrix size 690×690 against different number of cores.

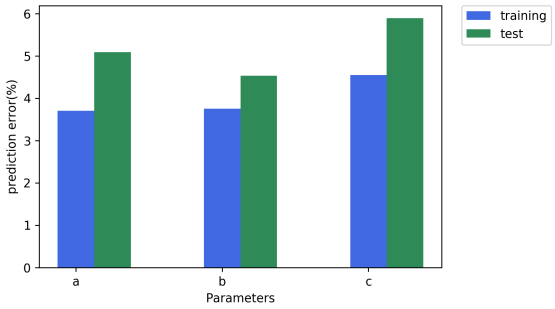


Figure 4.12. The error in fitting the parameters a, b, and c for matrix size 690×690.

Using this 3rd degree polynomial to fit the parameters, we can generalize the relationship between throughput and grain size in the following equation:

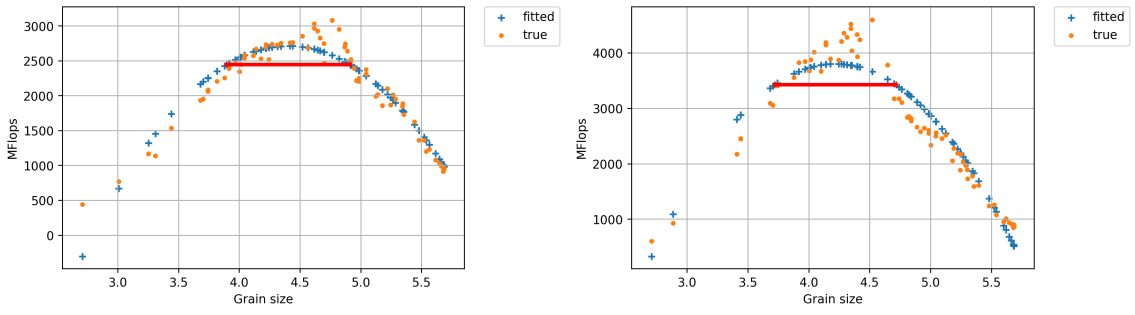
$$P = a_{11}g^2N^3 + a_{10}g^2N^2 + ... + a_1N + a_0 \tag{4.2}$$

where P is the throughput, g is the grain size, and N is the number of cores and coefficients $a_{11}, ..., a_0$ are the real values.

Knowing that a polynomial of degree 2 based on grain size and of degree 3 in terms of number of cores, we can try to fit our original data directly to the above mentioned formula (Equation 4.2)).

4.3.3. Finding the range of grain size to achieve high performance

The major advantage of using a quadratic function to fit the data, is the simplicity of the formula, which makes it possible for us to find the peak of the graph very easily. n order to add some uncertainty to our prediction, instead of finding the maximum of the quadratic function, we identified the range of grain size that results in a performance within 10% of the maximum performance. For a second degree polynomial in terms of g, $P = ag^2 + bg + c$, the minimum or maximum of the polynomial is located at $p^* = \frac{-b}{2a}$.



(a) (b)

Figure 4.13. The range of grain size that leads to a performance within 10% of the maximum performance for (a) 4 cores and (b) 8 cores.

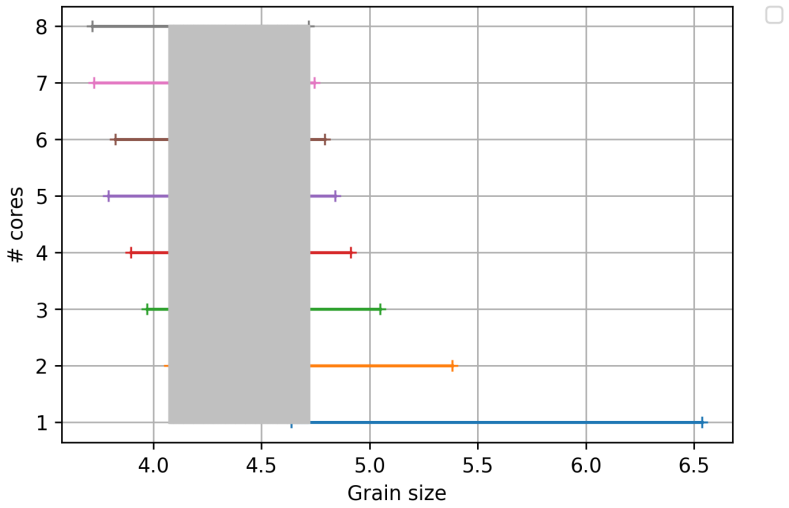


Figure 4.14. The range of grain size within 10% of the maximum performance of the fitted quadratic function for $DMATDMATADD$ benchmark and matrix size 690×690 against different number of cores cores.

Figure 4.14 shows the calculated range for matrix size 690×690 with different number of threads. In order to generalize the solution, we selected the intersection of all the ranges to find a range of grain size that we expect to work well for all the different number of cores in our experiment. This silver region in Figure 4.14 corresponds to this range. The green line is the throughput achieved by the current implementation of HPX backend.

4.3.4. Estimating the chunk size

Once we identified a range of grain sizes that is expected to leads us to highest achievable performances for a specific matrix size, the next step is finding the possible combinations of block size and chunk size to achieve that range of grain sizes. As stated earlier in this chapter, results obtained from Figure 4.5 suggests that with a fixed grain size, our choice of block size does not affect the performance directly, as long as there exist a chunk size that when combined by the block size could result in the specified grain size.

In our experiment, we selected our block size to be 4×256. With this assumption, in order for the grain size to be within the specified range for each matrix size, chunk size has to be within a specific range size too.

For example, for a 690×690 matrix we calculated the range of maximum performance for all number of cores to be $[4.075, 4.717]$ in logarithmic scale which is equivalent to $[11883, 52122]$. Setting the block size to 4×256, this range forces the chunk size to be within the range $[13, 56]$. The range of chunk sizes to match the range of grain sizes identified, and their corresponding throughput is shown in Figure 4.15, for matrix size 690×690 and block size 4×256.

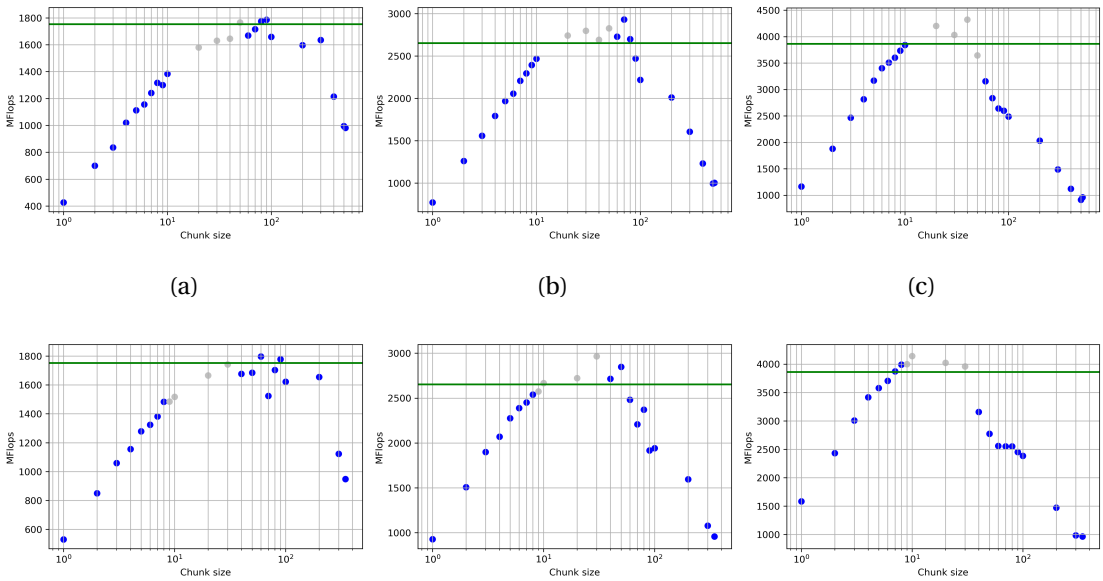


Figure 4.15. The range of chunk sizes to produce a grain size within 10% of the maximum performance of the fitted quadratic function for $DMATDMATADD$ benchmark for matrix size 690×690 with block size of 4×256 on (a) 2 cores, (b) 4 cores, and (c) 8 cores, and block size of 4×512 on (d) 2 cores, (e) 4 cores, and (f) 8 cores. Silver points denotes the detected range of chunk size.

Chapter 5. Results

5.1. 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

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

Chapter 6. Future Plans

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] 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.

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

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

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

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

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

[16] Patricia Grubel, Hartmut Kaiser, Jeanine Cook, and Adrian Serio. The performance implication of task size for applications on the hpx runtime system. In *2015 IEEE International Conference on Cluster Computing*, pages 682–689. IEEE, 2015.

[17] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[18] Neil J Gunther. *What is guerrilla capacity planning?* Springer, 2007.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] Gmm++ Library. `http://getfem.org/gmm/`.

[46] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, 1(2):26, 2016.

[47] Eigen Library. `http://eigen.tuxfamily.org/`.

[48] Automatically Tuned Linear Algebra Software. `http://math-atlas.sourceforge.net/`.

[49] GotoBLAS2. `https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2`.

[50] Intel Math Kernel Library. `https://software.intel.com/en-us/mkl`.

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