Optimizing the Performance of Multi-threaded Linear Algebra Libraries, a Task Granularity based Approach

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 of Electrical Engineering and Computer Science

by
Shahrzad Shirzad
B.Sc., Sharif University of Technology, 2006
M.Sc., K. N. Toosi University of Technology, 2009
December 2019

Acknowledgments

Table of Contents

List of Tables

List of Figures

Abstract

Linear algebra libraries play a very important role in HPC applications. In this work we propose a method to tune the performance of a linear algebra library based on a set of compile-time and runtime characteristics including the machine architecture, the expression being evaluated, the number of cores to run the application on, the type of the operation, and also the size of the matrices, to be able to get as close as possible to the highest performance.

There has been an extensive amount of work and study done on how to optimize the compute kernels for matrix-matrix multiplication. In this thesis, we are interested in all types of operations, and our focus is on machine learning applications, where we are potentially dealing with very large matrices and creating temporaries could be very expensive.

For this purpose we decided to use Blaze C++ library, a high performance template-based math library that gives us this option to access the expression tree at compile time, along with HPX, a C++ standard library for concurrency and parallelism, as our runtime system. We propose here that instead of dividing the work equally among the cores and assigning one chunk to each core, we should be able to achieve a higher performance by selecting the right amount of work to be assigned to one core, based on a set of runtime and compile-time parameters.

Studying the significant amount of data we collected with different configurations, we concluded that, grain size, which is the amount of work assigned to each task, is the key factor for the performance when number of cores is fixed. With this assumption, we tried two different approaches to model the relationship between performance and grain size, in order to find a range of grain size that could lead us to the maximum performance.

First model uses a polynomial function to fit the data both for throughput vs. grain size, and throughput vs. number of cores. On the other hand, the second model was developed by studying the behavior of throughput vs. grain size , and integrating it with the an extension of the Universal Scalibility Law(USL). This developed model fitted our data reasonably and could be very important in further understanding of the parallelism.

The primary results suggest that, using the data collected for matrix-matrix addition, we are able to improve the performance for matrix-matrix addition, by finding the right range of grain size.

Having the mentioned models, we changed the current implementation of the HPX backend for Blaze by adding two parameters to represent the unit of work and the number of these units included in each task, for fine grained control of the parallelism, which is possible through HPX runtime system. Also, a complexity estimation function has been added to Blaze as an estimate of the number of floating point operations occurring in each unit of work.

Our data was only limited to one specific operation, matrix-matrix addition for two raw-major matrices, one matrix size at a time. In the next step, this problem should be generalized to different matrix sizes, architectures, and arbitrary complex expressions.

Chapter 1. Introduction

The path toward exascale computing, would include more complex machine architectures, deeper memory hierarchies, heterogeneous nodes, complicated networks[2]. Current programming models would not be sufficient. Advanced runtime systems with support for new programming languages and models are needed to manage the huge amount of parallelism that would be available[3]. Asynchronous many-task(AMT) models and their corresponding run-times are the solution to keep application developers safe from the upcoming architectures, by mitigating exascale difficulties to run-time level[4].

On the other hand, the core element of many high performance computing applications is the linear algebra library. The performance of these application heavily relies on the performance of their linear algebra library. BLAS(Basic Linear Algebra Subprograms) are the fundamental routines for basic vector and matrix operations. But in order to tune BLAS for a specific architecture, a lot of effort needs to be made, and requires a deep understanding of memory hierarchy and registers from the programmer[5]. Linear algebra libraries like ATLAS[5], SPIRAL[6] try to use hardware-specific optimizations to improve their performance.

In this work, we are trying to optimize the performance of a linear algebra library based on the application parameters such as matrix size, operation, the expression, data layout, and also the machine architecture. Some of these parameters could be extracted at compile time, while extraction of the others should be postponed to runtime.

1.1. Thesis Statement

The main objective of this thesis is to propose a hybrid runtime and compile-time solution for a linear algebra library to fully take advantage of the available parallelism and resources.

We chose Blaze math library since it is a nice high performance template-based C++ library that allows you to access the expression tree for each assignment at compile time, and we chose HPX as an asynchronous may-task runtime system to manage the parallelism.

HPX makes it possible to create thousands to millions of lightweight user threads, to avoid

expensive context switching. On the other hand, although the overhead of creating one task is negligible, creating millions of tasks when the execution time of the program itself is small, could become significant and cause performance degradation. On the other hand if we create too few tasks it would be very likely for us to not use our resources properly. So in very application in many task systems, it's very important to chose the amount of work assigned to each task, called grain size, properly.

Through analyzing and modeling the relationship between throughput and grain size, we would be able to identify a range of grain size that leads us to maximum performance. Once decided how big one unit of work should be, based on the identified range we would be able to decide on how many units of work should be packed into one task.

On the other hand, there are different models to express the relationship between the throughput and the number of cores. Here, we are interested in developing a model to be as realistic as possible to imitate the behavior of the throughput against both grain size and number of cores. This would help us find how to manage the parallelism in our system to achieve the highest performance possible.

1.2. Contributions

There has been a wide study, mostly by Gunther[7, 8, 9, 10], on different models to represent the relationship between the throughput and the number of cores, for a fixed size problem. Grubel et.al[1] has studied the effect of task granularity on the performance with a fixed number of cores. Our contributions could be summarized into:

- We propose a novel physical model to represent how the execution time is expected to change based on grain size.
- To our knowledge, there has not been a work to create a 3D model of the throughput, grain size, and number of cores.

- We are proposing a method to apply the developed model to a linear algebra library, in a way specific to our application, and the machine architecture.

1.3. Document Organization

In Chapter 2, we will explain briefly the background needed for this thesis, including Blaze and HPX library, the effect of task granularity, and the Universal Scaling Law(USL) method for modeling the throughput based on number of cores. Chapter 3 refers to other works that have been done in our area of our focus. We explain our proposed method to optimize the performance, along with the models we used in Chapter 4. Our work heavily relies on the collected data, the environment we collected the data from and also the library versions are mentioned in Chapter 7. Finally we discuss our concerns and the further steps that needs to be taken in Chapter **??**.

Chapter 2. Background

2.1. Asynchronous Many-task Runtime Systems

A parallel programming model includes a programming model, which refers to the mechanism for a program to express the concurrency, and an execution model, indicating how the program creates and controls concurrency[11, 4]. Some of the common existing execution models include fork-join, Communicating Sequential Processes(CSP), event-based models and actor model[4]. Some of the current parallel programming frameworks include, accelerator-based programming models like OpenACC[12], OpenCL[13], and CUDA[14], shared-memory programming models like Intel TBB[15], Cilk[16], OpenMP[17], and distributed-memory programming models like MPI[18], Charm++[19], Parallex[20], UPC[11].

A runtime system is in charge of creating and managing the concurrency, through implementing parts of an execution model[4]. An asynchronous many-task(AMT) model on the other hand, is a category of programming model and execution model. An AMT programming model breaks the work into small and transferable tasks along with their associated input. In an AMT execution model, tasks are being executed when their inputs are available rather than in a well defined order[4].

Some of well known AMT runtimes include: HPX[3], Charm++[19], Uintah[21], Legion[22].

2.1.1. HPX

HPX[3] is a C++ runtime system for parallel and distributed applications based on ParalleX execution model[20]. HPX provides you with lightweight user-level threads with fast context switching[11]. Whenever one thread is blocked, the scheduler picks a ready thread based on a scheduling policy. This allows you to hide the latency, and avoid starvation while keeping a high utilization of the resources[11].

2.1.1.1. Execution Model

The "SLOW" model identifies four potential sources for performance degradation as: Starvation, Latency, Overheads, and Waiting or contention[20].

Starvation refers to a situation where there is insufficient amount work for the computing resource. this could be due to insufficient total amount of work available, or unbalanced distribution of work among resources[11].

Latency is the time distance, usually measured in processor clock, of accessing remote data or services[23].

Overhead refers to the effort that needs to be taken to manage parallel resources and actions on the critical path[11].

Finally, waiting is the contention of the shared physical and logical resources causing one request to be blocked by another access of the same resource[23]. This could happen due to limited network bandwidth, shared communication channels, memory bank conflicts[11],[23].

2.2. Blaze

Blaze Math Library[24] is a C++ library for linear algebra. Blaze, based upon Expression Templates(ETs)[25], introduces "smart" expression templates(SETs)[24] to optimize the performance for array-based operations. Expression Templates[25] 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[24]. 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[24]. As a solu-

tion, 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[24].

Some of the ET-based linear algebra libraries are: Blitz++[26], Boost uBLAS[27], MTL[28], and Eigen[29]. Among these libraries, Eigen, MTL, alongside Blaze, impose different conceptual changes to ETs in order to make them suitable for HPC. Blaze also makes it possible for programmers to utilize SIMD(Single Instruction Multiple Data) vectorization simply by adding a compile time flag.

2.3. Task Granularity

Defining the grain size as the amount of work assigned to one HPX thread, Grubel[1] 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[1]. 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.

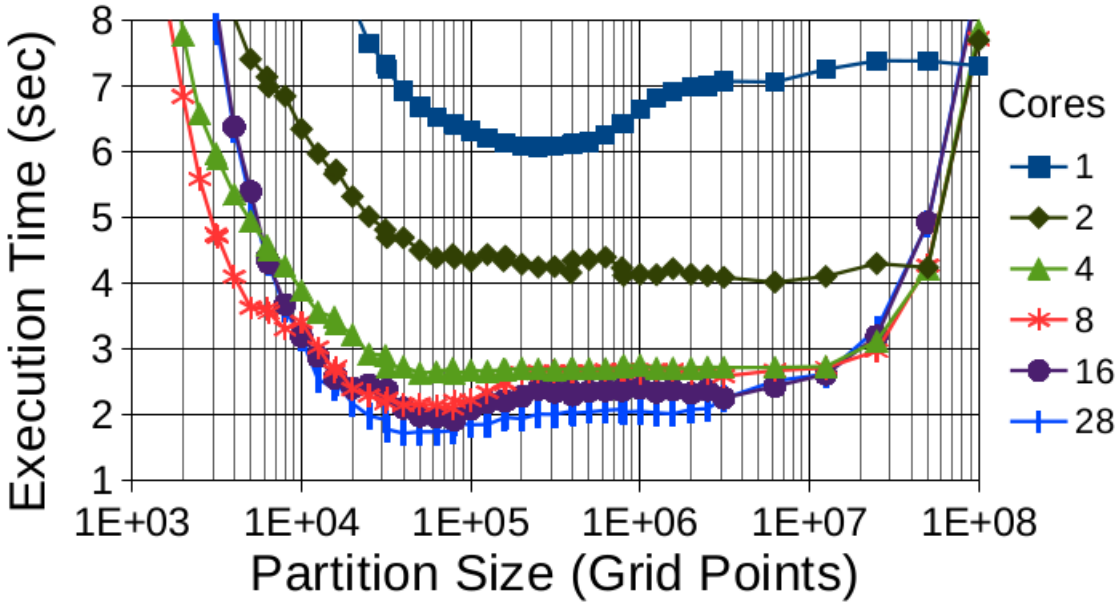


Figure 2.1. The effect of task size on execution time for Stencil application [1]

2.4. Analytical modeling of parallel programs

The execution time of a parallel program is highly dependent of the parallel algorithm to be evaluated, alongside the architecture it is implemented on[30].

2.4.1. Universal Scalibility Law

Amdahl's law[31], 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[9].

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

On the other hand, Gunther[9] 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[9].

Figure 2.2 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.

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

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[9]. Increasing the number of processor beyond that point would only cause performance degradation.

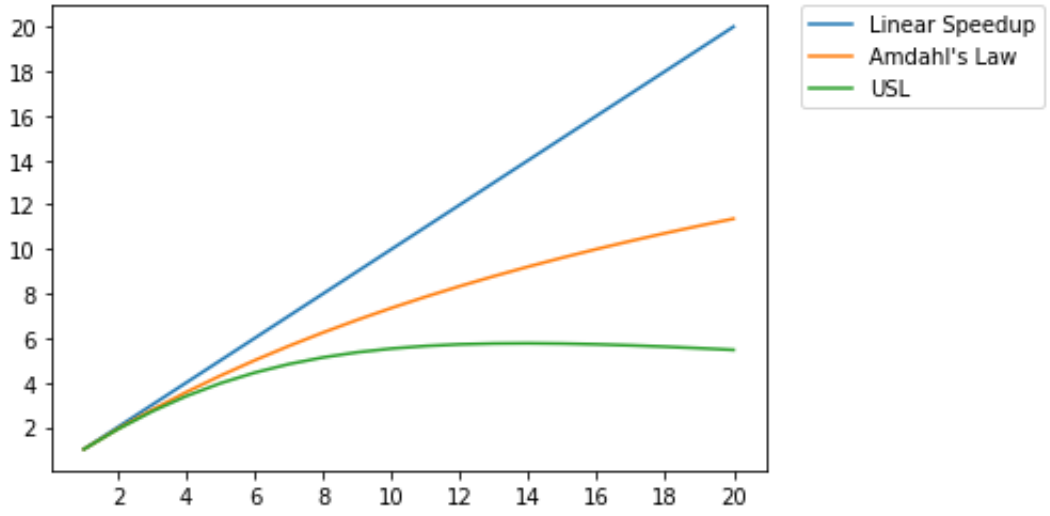


Figure 2.2. An example of the achievable speedup based on Amdahl's law and USL compared to the ideal linear speedup where $\sigma = 0.04$ and $\kappa = 0.005$.

2.4.2. Other Models

There are a few other models that have also been suggested to simulate the scalibility. Geometric model is a one-parameter model, in which speedup has the following relationship with the number of processors:

$$S(p) = \frac{1-\phi^p}{1-\phi} \tag{2.4}$$

The parameter ϕ, where $0 < \phi \leqslant 1$, is called the MP factor, and it represents the remaining section of the processor capacity after deducting overheads. The geometric model is non-physical for large number of processors, due to inconsistency with Coxian queuing model[8].

Quadratic model[7], with overhead parameter γ, where $0 \leqslant \gamma < 1$, is represented in Equationquad.

$$S(p) = p - \gamma p(p-1) \tag{2.5}$$

The quadratic model has a critical point at:

$$p^* = \lfloor \frac{1+\gamma}{2\gamma} \rfloor \tag{2.6}$$

The problem with this model is that its not physical. This model represents an inverted parabola that will intersect the x axis at two points, implying that there with be a certain number of processors starting from which the speedup would be negative.

Exponential model, is also a single parameter model α where $0 < \alpha \leqslant 1$. This parameter is a combination of coherency and contention.

$$S(p) = p(1-\alpha)^{(p-1)} \tag{2.7}$$

This model also has a critical point, but this point is very sensitive to α. Although this model works very well for small number of processors, it imposes a severe capacity degradation for large number of processors.

2.5. 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[32].

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[33]. 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[34].

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[35], Factoring[36] and Trapazoid Self-scheduling[37],[33].We briefly go over some of these loop scheduling techniques here.

In Pure Self-scheduling every time 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[33].

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[33].

As an adaptive loop scheduling technique, Guided Self-scheduling[35] 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[38].

Very similar to Guided Self-scheduling, Factoring[36] also decreases the chunk size as the loop execution proceeds, with this difference that it does it in batches of equal sized chunks. If the first iterations of the loop are more time consuming then the rest of the iterations, Factoring performs better than Guided scheduling[39].

Along with the mentioned loop scheduling approaches, two other scheduling techniques can be utilized for load balancing. Work stealing[40] lets the processors to steal work from other processors queue, resulting in a more balanced load distribution. In work sharing on the other hand, each time a processor creates new threads, the scheduler would try to migrate some of them to other processors for a more balanced load distribution[40].

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.

Chapter 3. Literature Review

3.1. Literature Review

Loop scheduling techniques has been extensively studied by different researchers. In [41] 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 [41], 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[41]. They were able to improve data locality and scheduling overhead, while creating a more balanced workload.

needs to be changed: ////////////////////////// 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[42].

The analytical models[43],[44],[45], 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[46],[47],[48]. 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[47]. 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[42]. This type of modeling includes machine learning based approaches.

In [49], the authors use neural networks to predict the performance focusing on SMG2000 application, a parallel multigrid solver for linear systems[50], 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, [49] 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, [47] 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[47].

In [42] 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, [42] suggest utilizing transfer learning through deep neural networks to predict performance on other platforms.

Although concentrating on GPUs, [51] 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, [51] 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.

//////////////////////////

As another field to use machine learning, [52] 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.

The authors of [53] 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 [53] 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.

adaptive runtime solutions: In [54], the authors offer a combination of compile-time and run-time solution for adaptive control of task granularity. They create multiple transformed versions of the code with different levels of task unrolling at compile time and then use a heuristic based on task demand (the number of unsuccessful steal attempts by other workers) and each worker's queue length, to select one of the versions each time a new task is spawned[54]. Their solution relies completely on the compiler and the run-time, and eliminates the need for manual support.

Tzannes et al. proposed a scheduling technique that adjusts the available parallelism based on the inferred load at run-time, to avoid the unnecessary parallelism overhead[55].

[56] [57] [58]

Chapter 4. Method

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[17], C++ threads, and Boost[59]. 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.

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. For simplification we are only studying raw-major matrices at this point. Our final goal is to extend the work to cover arbitrary data layouts for arrays.

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.

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 );
}
} );
}
```

Category	Configuration
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 Table 4.2 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.

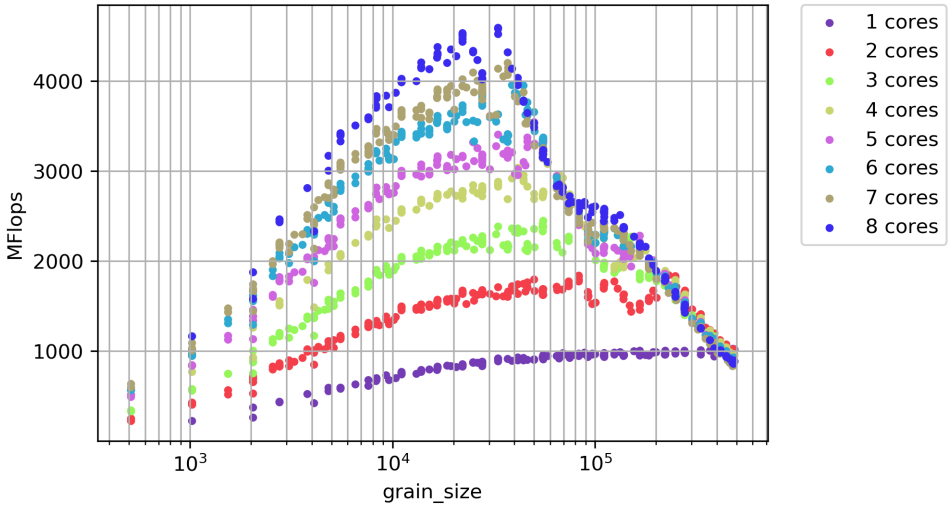


Figure 4.1. The results obtained from running $DMATDMATADD$ benchmark through Blazemark for matrix size 690×690 on different number of cores.

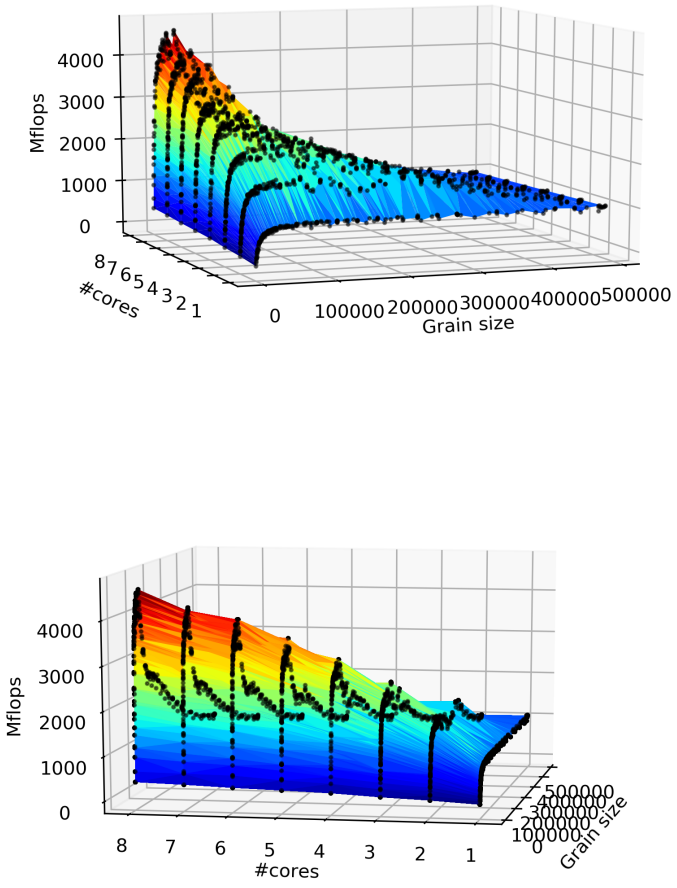


Figure 4.2. The results obtained from running $DMATDMATADD$ benchmark through Blazemark for matrix of size 690×690 from two different angles

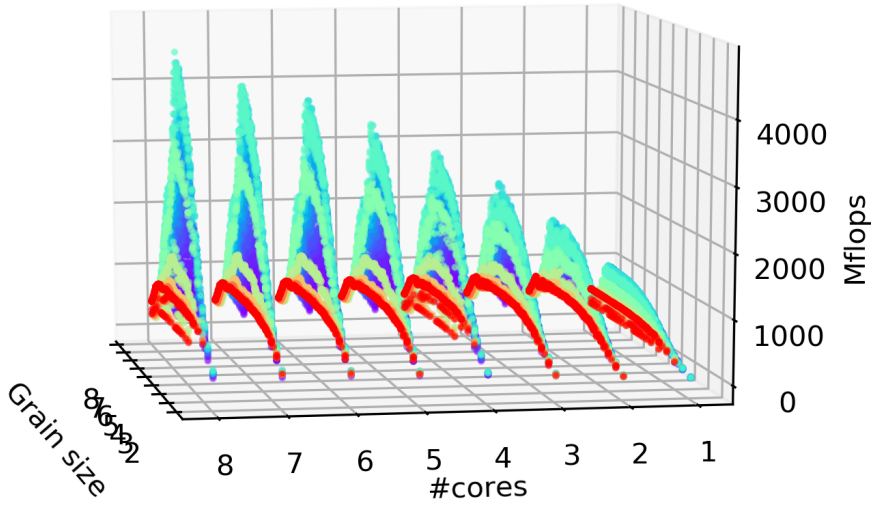


Figure 4.3. The results obtained from running $DMATDMATADD$ benchmark through Blazemark for matrix sizes from 200×200 to 1587×1587

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.

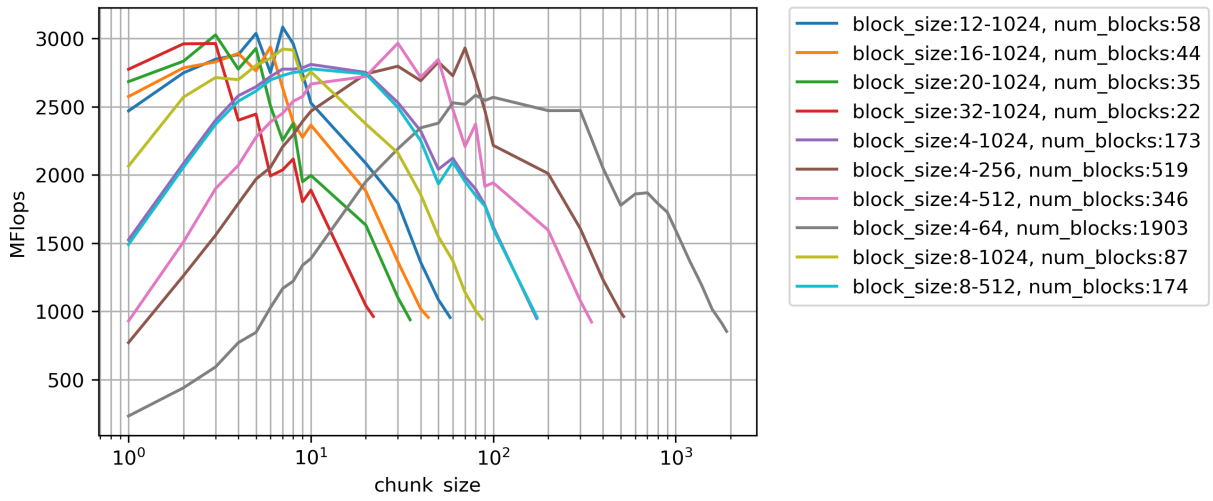


Figure 4.4. The results obtained from running $DMATDMATADD$ benchmark through Blazemark for matrix size 690×690 with different combinations of block size and chunk size on 4 cores

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 4.1 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.

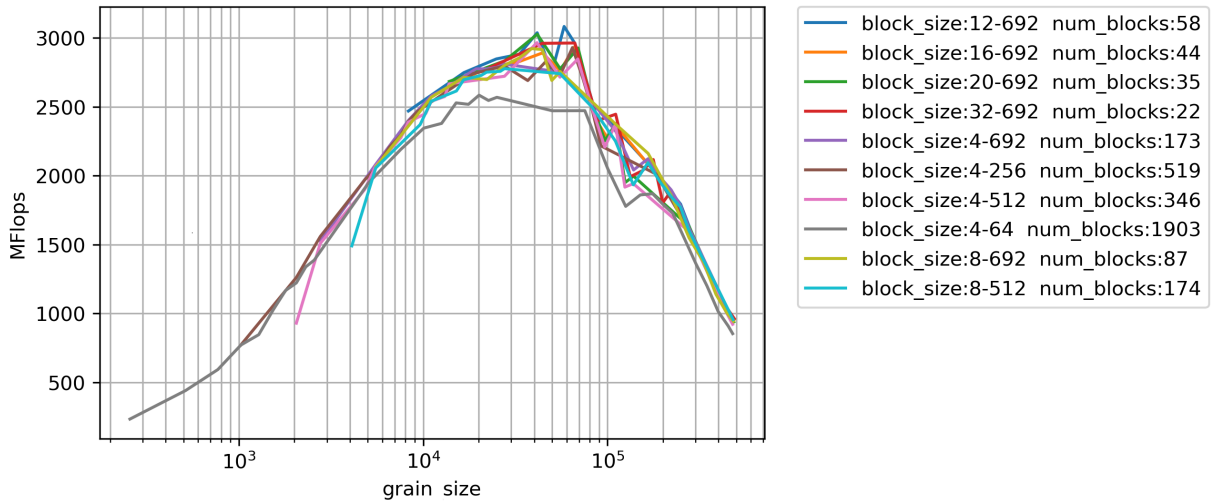


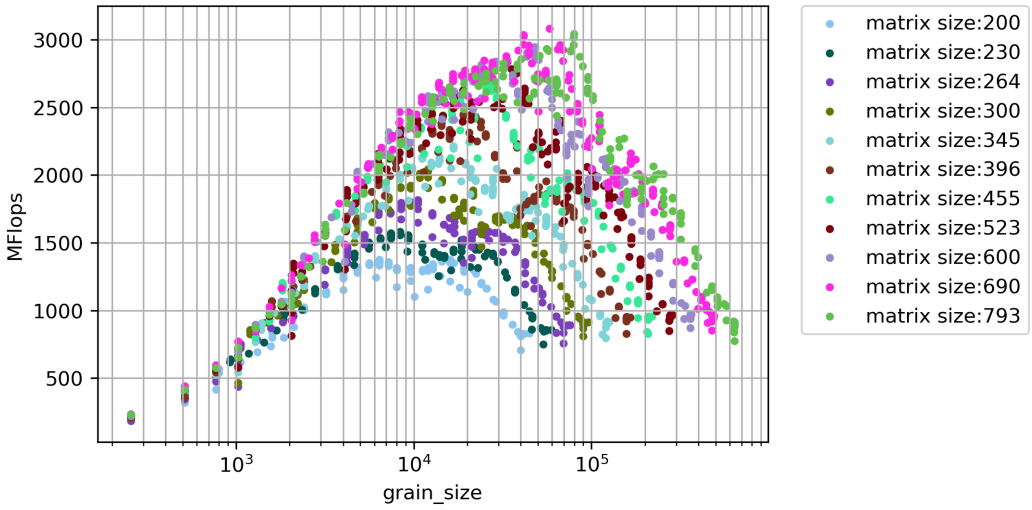
Figure 4.5. The results obtained from running $DMATDMATADD$ benchmark through Blazemark for matrix size 690×690 on 4 cores.

4.3. Method

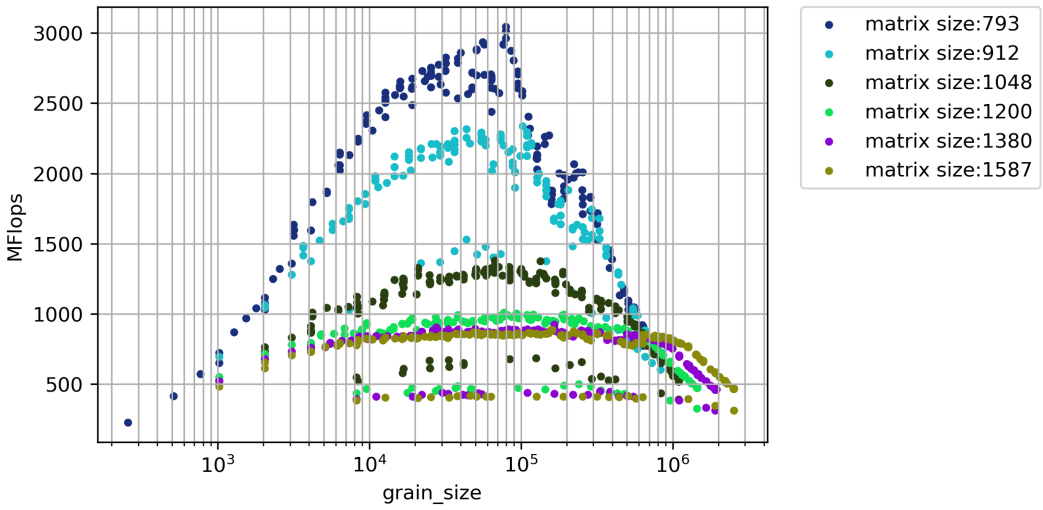
Looking at the throughput vs. grain size graphs and the consistent pattern observable motivated us to try to model the relationship between throughput and grain size. In order to simplify the process and eliminate the effect of different possible factors, we started with limiting the problem to a fixed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

4.3.1. Polynomial 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 4.6 to a second degree polynomial.



(a)



(b)

Figure 4.6. 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.

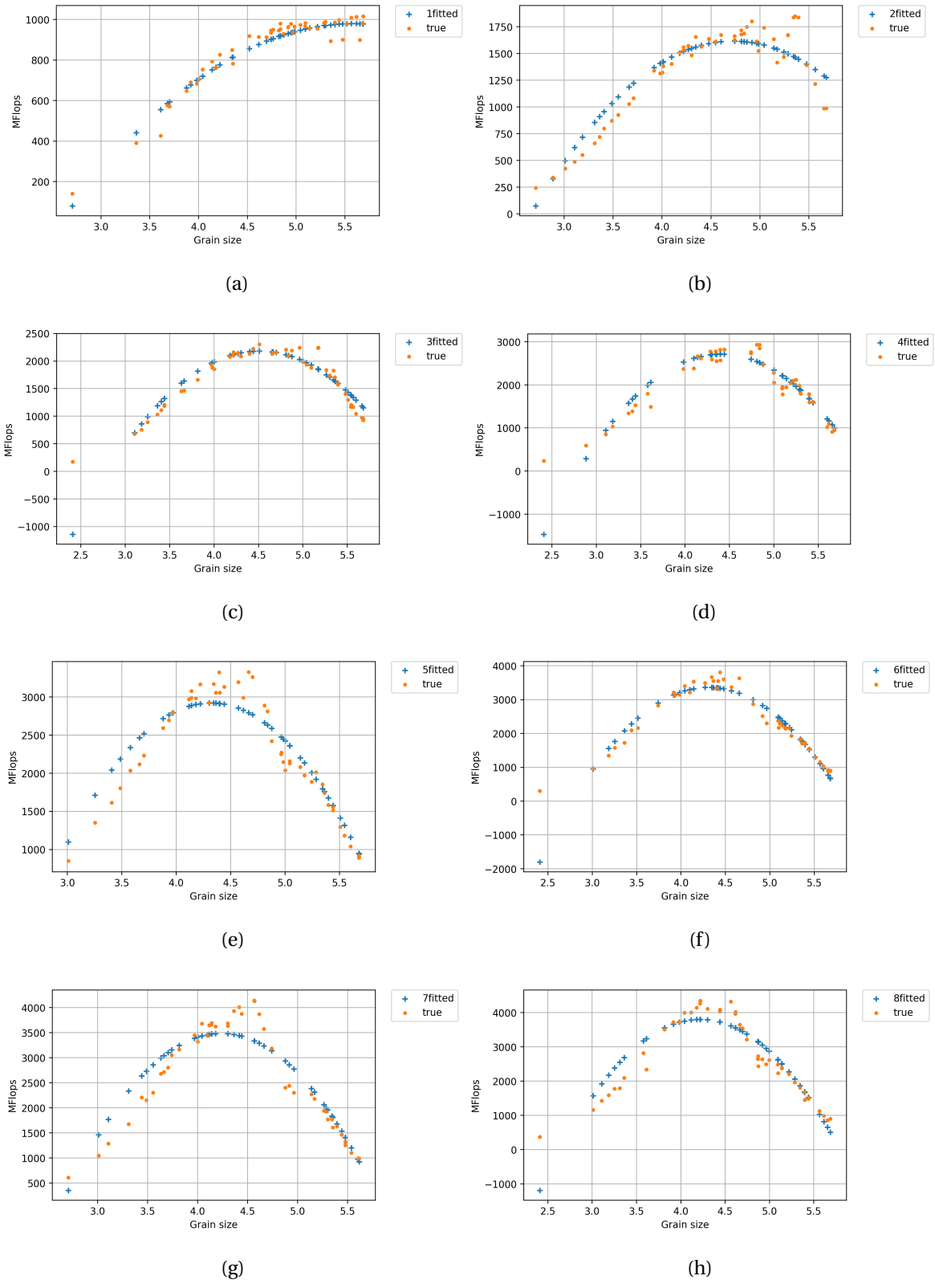


Figure 4.7. The results of fitting the throughput vs grain size data into a 2d polynomial for $DMATDMATADD$ benchmark for matrix size 690×690 with different number of cores on the test data set (a) 1 core, (b) 2 cores, (c) 3 cores, (d) 4 cores, (e) 5 cores, (f) 6 cores, (g) 7 cores, (h) 8 cores.

Figure 4.7 shows the results of using a quadratic function to fit the data for one matrix size with different number of threads. We used th *polyfit* package from *numpy* library in *python* for this purpose, which tries to minimize the least-square error over all the samples.

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 4.8 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.

$$Relative\_error = \frac{1}{n}\sum_{i=1}^{n}|1-\frac{p_i}{t_i}| \tag{4.1}$$

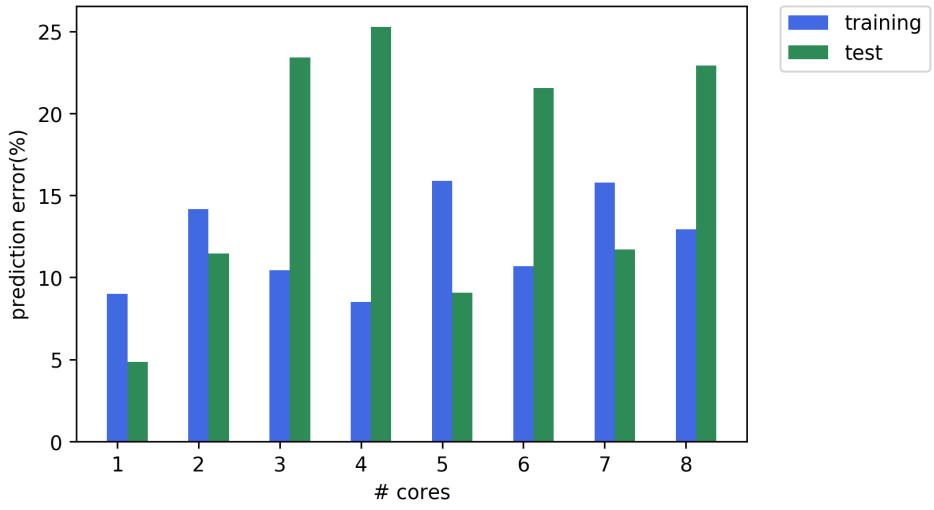
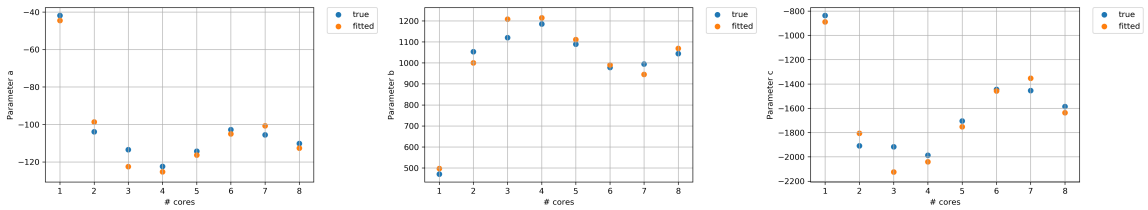


Figure 4.8. The training and test error for fitting data obtained from the $DMATDMATADD$ benchmark for matrix size 690 × 690 against different number of cores cores.

4.3.1.1. 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 4.9. 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.

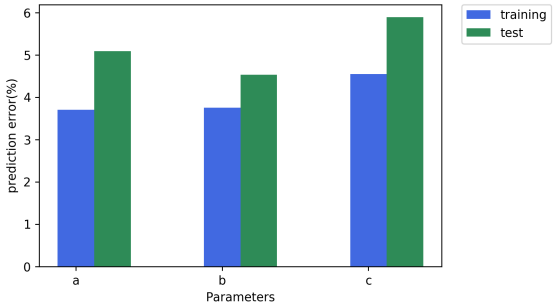


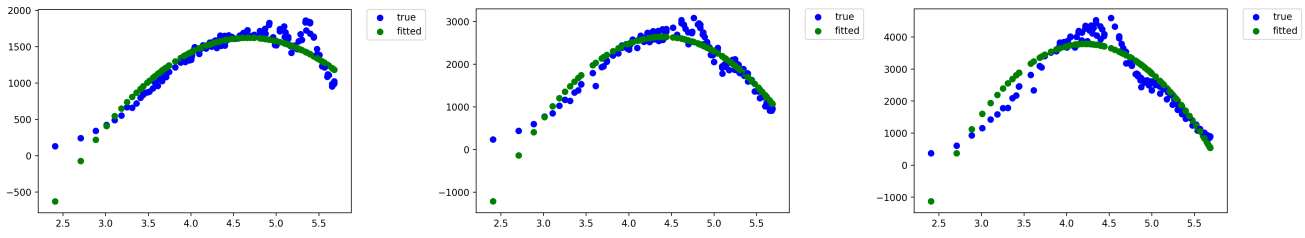
Figure 4.10. The error in fitting the parameters a, b, and c for matrix size 690 × 690.

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

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

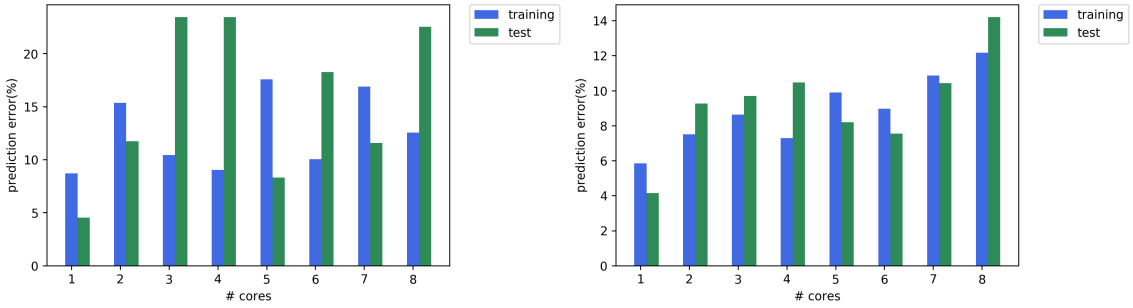
where P is the throughput, g is the grain size, and N is the number of cores and coefficients $a_{11},...,a_0$ are the real values.

Knowing that a polynomial of degree 2 in terms of 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). The results of the original data obtained from running $DMATDMATADD$ benchmark, the fitted polynomial based on Equation 4.2, is represented in Figure 4.11, for 2,4, and 8 cores for a matrix of size 690×690.



(a) (b) (c)

Figure 4.11. Results of fitting the data from $DMATDMATADD$ benchmark with a polynomial of degree 2 in terms of grain size and of degree 3 in terms of number of cores for matrix size 690×690 for (a) 2 core, (b) 4 cores, (c) 8 cores.



(a) (b)

Figure 4.12. The training and test error obtained fitting the data to a polynomial of degree 2 in terms of grain size and of degree 3 in terms of number of cores for matrix size 690×690, for each number of cores. (a) All the data points are include in calculation of error, (b) the leftmost sample was removed from error calculation.

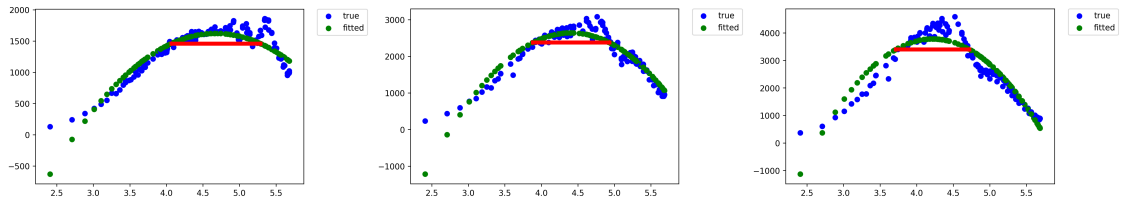
Figure 4.12a shows the obtained relative error on both training and test sets. The graph

suggests a higher test error compared to the training error, mostly caused by the left hand side of the graph. The effect of removing the leftmost sample from error calculations is depicted in Figure 4.12b.

Although we are interested in finding a model that results in a low training and test error, our purpose is mainly finding the region that generates the highest performance. So, even though our model might not match the original data in all data points, due to having a different nature than a quadratic function, our focus would be on how this fit can help us to find which range of grain sizes, or how big the task sizes should be, to achieve the highest performance.

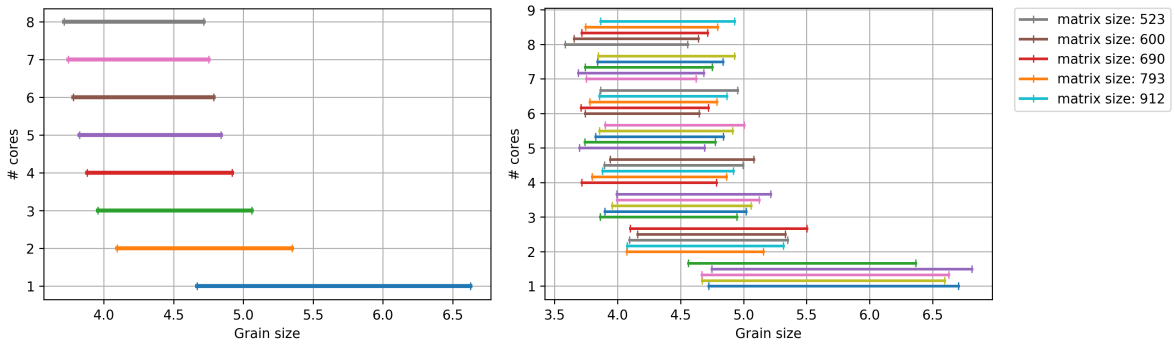
4.3.1.2. Finding the range of grain size to achieve the highest performance

The major advantage of using a quadratic function to fit the data in terms of grain size, when number of cores is fixed, 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 $p^* = \frac{-b}{2a}$, and a, b, c are 3rd degree polynomials of number of cores.



(a) (b) (c)

Figure 4.13. The range of grain size (shown as the red line) that leads to a performance within 10% of the maximum performance for (a) 2 cores, (b) 4 cores and (b) 8 cores.



(a) (b)

Figure 4.14. The range of grain size within 10% of the maximum performance of the fitted polynomial function for $DMATDMATADD$ benchmark for different number of cores for (a) matrix size 690×690 (b)matrix size 523×523 to 912×912.

Figure 4.14a shows the calculated range for matrix size 690×690 for each specific number of threads, while Figure 4.14b compares the range for different matrix sizes.

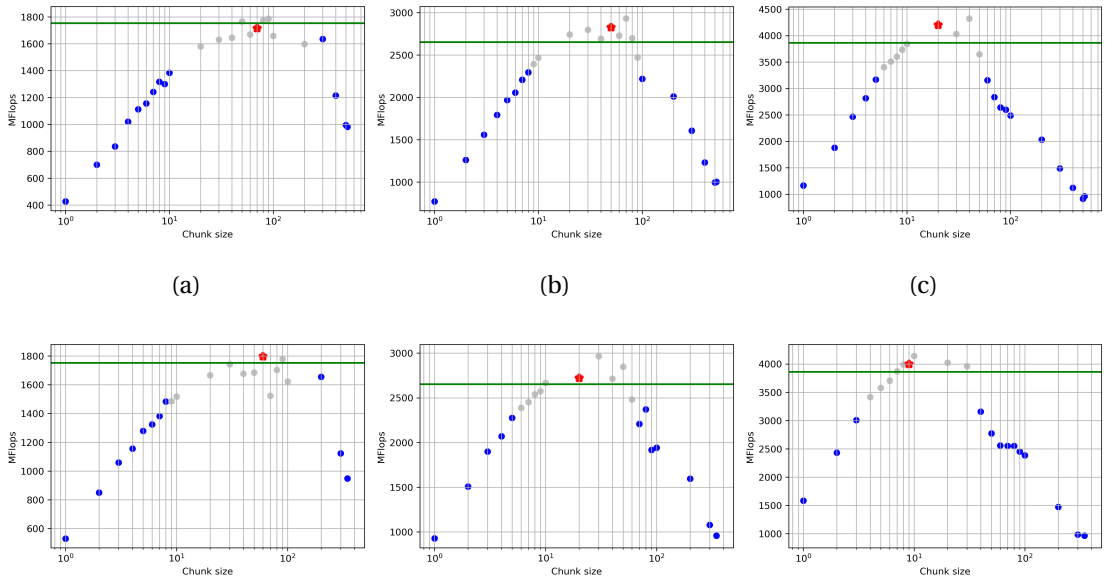
4.3.1.3. Estimating the chunk size

Once we identified a range of grain sizes that is expected to leads us to highest achievable performances for a specific matrix size and a specific number of cores, 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 4 cores to be $[3.88, 4.92]$ in logarithmic scale which is equivalent to $[7586, 83176]$. Setting the block size to 4×256, this range forces the chunk size to be within the range $[9, 90]$. 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. The green line is the throughput achieved by the current implementation of HPX backend. Since the graph from the original data is skewed to right, we selected the point after the median of all the chunk sizes in the range, as our candidate chunk size for this specific configuration.



(a) (b) (c)

(d) (e) (f)

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, and the red star shows the median point.

4.4. Conclusion

In this chapter we offered a simple method to estimate the range of grain size, and consequently the range of chunk size to lie in the region of the throughput against grain size graph, with the highest throughput.

This quadratic model is very simple and finding the range of maximum performance is very easy due to the nature of the function. But this model has some limitations. First of all, it is not physical. The parameters of the model does not have a physical implication. The

other issue is that in this method one function was fitted for each matrix size individually for simplification. Even though the characteristics of the function is the same for all matrix sizes, the fitted parameters vary from one matrix size to another.

The matrix size should somehow be integrated into the model itself. For this purpose, first we need to understand how changing the matrix size affects the relationship between grain size and performance. One immediate effect of increasing the matrix size is an increase in maximum possible grain size (right hand side of Figure 4.16) , while the minimum possible grain size is the same.

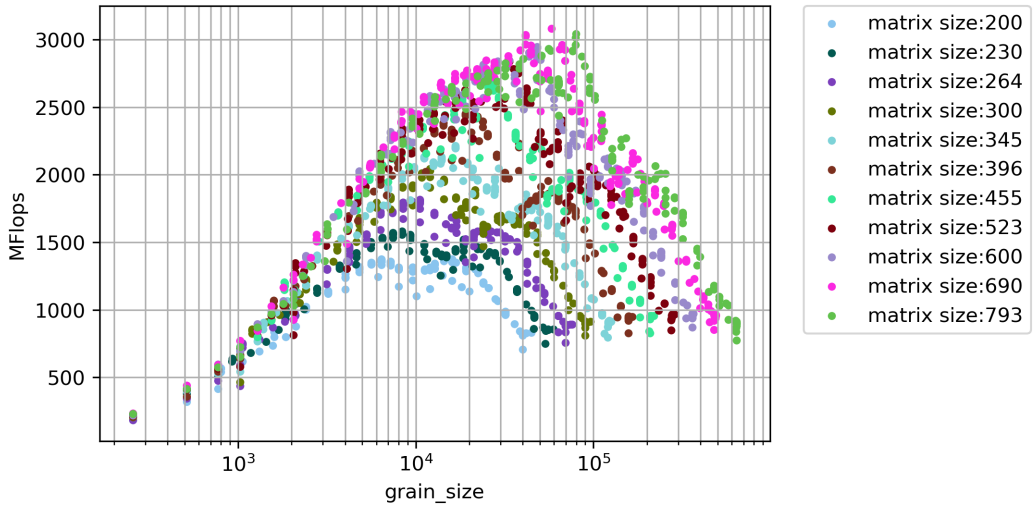


Figure 4.16. Throughput vs. grain size graph obtained from running *DMATDMATADD* benchmark on 4 cores.

Moreover, Figure 4.17 shows how the predicted grain size range changes for different matrix sizes for each number of cores.

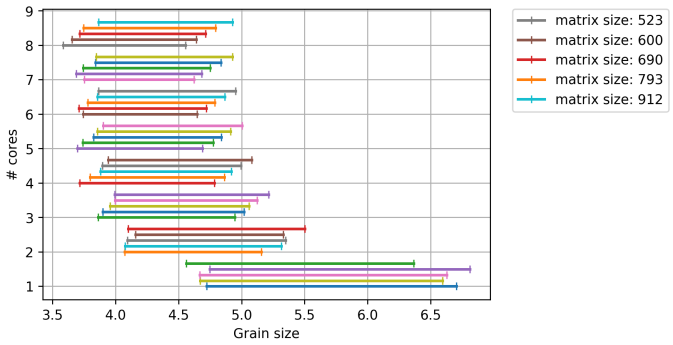


Figure 4.17. The range of grain size within 10% of the maximum performance of the fitted polynomial function for $DMATDMATADD$ benchmark for different number of cores for matrix size 523×523 to 912×912.

Also, larger matrix sizes should be added to the experiments to validate the current results.

In the next section we look into a more robust and physical representation of how performance changes with grain size, and consequently predicting the range of grain size and chunk size to achieve the maximum performance.

Chapter 5. Understanding the effect of grain size on concurrency in an asynchronous many-task runtime system

5.1. Analytical Modeling

In the previous section we studied the possibility of using a polynomial regression model to capture the relationship between grain size, number of cores, and throughput for a fixed matrix size, with the purpose of finding a range of grain size that leads us to maximum performance. Although the polynomial function was helpful in directing us toward our objective of finding the region of grain size with maximum performance, it lacked a physical implication, and was not quite able to capture the overall behavior of the system.

This motivated us to change our view, and instead of looking just at the data and trying to find a function to fit the data, study the behavior of the data, and then find a function that would be likely to fit and explain the data. That function would be a good fit mostly because that's how we expect the throughput to change with grain size, and not solely how the data behaves.

In this chapter we attempt to understand the effect of grain size on the achievable speedup in an asynchronous many-task runtime system, in order to develop an analytical model for predicting the execution time in an asynchronous many task runtime system.

We have to note here that even if we were able to identify all the factors affecting the execution time, it is still very hard to find an analytical model describing the relationship between these factors and the execution time. In this chapter we explain our effort in this direction by starting from a simple benchmark. We suggest a formula based on our knowledge of the behavior of the system, and the collected data. Afterwards, we try to generalize the proposed formula to arbitrary parallel for-loops with balanced work-load for each iteration.

Throughout this chapter we will be using the following terms and definitions to simplify our work. We represent the number of cores available with N, the number of tasks created as num_tasks, the maximum amount of work assigned to one core as w_c, the number of cores that are actually doing the work(when there is not enough work for all the cores) as M

($M \leq N$), the total amount of work available as $problem\_size$, and the sequential execution time as t_seq.

In an attempt to find this analytical model for execution time of a balanced parallel for-loop, we started with looking into two factors that we believe play the most important role in determining the execution time: the overhead of creating tasks, and the maximum amount of work assigned to one core.

In order to understand how these factors contribute to the execution time in a balanced parallel for-loop, we define *iter_length* as the amount of time to execute one iteration, *num_iterations* as total number of iterations, $chunk\_size$ as number of iterations to be executed by one thread.

The total execution time could be assumed to be roughly the amount of time it takes for the core with the maximum amount of work to finish it's job. Here we call the core with maximum expected amount of work as $core_0$, and the associated amount of work as w_c. With this assumption, the main factors contributing to the execution time are the overhead of creating and managing tasks on $core_0$, the time it takes to run w_c amount of work on $core_0$, denoted with t_c, and the number of cores that will be executing the work(M). Depending on the amount of work available, either all N cores or less than N cores will be performing the work.

Formula 5.1 shows the expected formula in it's simplest form.

$$execution\_time = t\_overhead + t\_c \tag{5.1}$$

In Formula5.1, $t\_overhead$ represents the penalty that we have to pay for running the program in parallel. We hold two major factors accountable for this overhead.

The first factor is the overhead of creating and managing the tasks. Although this overhead is negligible for small number of tasks, it becomes significant as the number of created tasks becomes larger, playing an important role in the overall execution time.

When num_tasks tasks are created, $\left\lceil \frac{num\_tasks}{N} \right\rceil$ tasks would be created on $core\_0$. If we

represent the overhead of creating and managing a task on one core with α, the portion of $t\_overhead$ resulted from num_tasks tasks could be stated as $\alpha \times \left\lceil \frac{num\_tasks}{N} \right\rceil$.

The second factor is the overhead caused by work stealing. Although work stealing is a very helpful method for load balancing, it induces an overhead due to constant efforts of idle cores to steal work from the other cores. Each of these efforts would result in trashing the local cache of the busy cores. This effect helps for load balancing when the number of tasks created is greater than or equal to the number of cores, but as the number of tasks get smaller than the number of cores, the effect of the unsuccessful steal efforts become more noticeable.

HPX by default uses the priority local scheduling policy which creates one queue for each core. When there is no more work left in one core's queue, it will attempt to steal work from other cores starting from it's neighbors. If the attempt is not successful, it will move on to the next level neighbors. This continues until the core that initiates the steals(called the thief) is able to find a core with tasks in their queue which can be stolen(called the victim).

HPX provides an optional command line parameter $--hpx:numa-sensitive$ to make sure that the thief would try the queues of the cores in the same NUMA domain first. This option is provided based on the fact that it is much faster to access the local memory of a processor than the local memory of another processor.

Expressing the total overhead as the summation of the overhead of creating and managing the tasks, and the overhead resulted from work stealing, as shown in Formula 5.2,

$$t_{overhead} = \alpha \times \left\lceil \frac{num\_tasks}{N} \right\rceil + t\_overhead\_ws \tag{5.2}$$

Formula 5.1 then becomes:

$$execution\_time = \alpha \times \left\lceil \frac{num\_tasks}{N} \right\rceil + t\_overhead\_ws + t\_c \tag{5.3}$$

We had previously defined t_c as the time it takes to perform w_c amount of work, where w_c is the maximum amount of work assigned to the cores.

For a balanced parallel for-loop, w_c can be calculated as:

$$w\_c = \begin{cases} problem\_size & \text{if } N = 1 \\ problem\_size - g \times (N-1) \times (\left\lceil \frac{num\_tasks}{N} \right\rceil - 1) & \\ \qquad \text{if } num\_tasks\%N = 1 \ \ \& \ \ num\_iterations\%chunk\_size \neq 0 & \\ g \times \left\lceil \frac{num\_tasks}{N} \right\rceil & \text{otherwise} \end{cases} \tag{5.4}$$

When $N = 1$, all the work will be assigned to the only available core, causing w_c to be equal to all the available work, $problem\_size$.

When $N > 1$, in the general case at most $\left\lceil \frac{num\_tasks}{N} \right\rceil$ tasks would be assigned to a core. Therefore, with a grain size of g each task would contain g amount of work, resulting in a maximum amount of work of $g \times \left\lceil \frac{num\_tasks}{N} \right\rceil$ assigned to a core.

It should be noted that the last chunk of work created could contain less amount of work than the $chunk_size$, and this is the case when $num\_iterations\%chunk\_size \neq 0$. This although causing less amount of work to be assigned to one of the cores, would not affect the maximum amount of work assigned to the cores, unless when $num\_tasks\%N = 1$. For these special cases where $num\_tasks\%N = 1$ and $num\_iterations\%chunk\_size \neq 0$ the maximum amount of work assigned to the cores would be $problem\_size - g \times (N-1) \times (\left\lceil \frac{num\_tasks}{N} \right\rceil - 1)$.

Figure 5.1 illustrates how w_c is calculated for this case. Each square represents a chunk or one task, containing g amount of work generating num_tasks chunks in total, and each column represents one core. The last chunk contains $g' = num\_iterations\%chunk\_size$ amount of work, where $g' < g$ and $g \times (N) \times (\left\lceil \frac{num\_tasks}{N} \right\rceil - 1) + g' = problem\_size$.

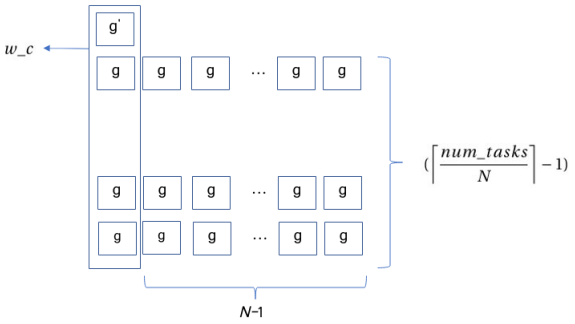


Figure 5.1. The way w_c is calculated illustrated for cases where $num\_tasks\%N = 1$ and $num\_iterations\%chunk\_size \neq 0$.

As discussed in chapter 2, Amdahl's law and Universal Scalibility Law suggest that for a fixed problem size, as we increase the number of cores in a multi-core system, the observed speedup is less than the linear speedup. which is mainly resulted from latency and coherency. Universal Scalibilty Law[9] suggests that an overhead associated with the number of cores should be added to the expected execution time. Formula 5.5 shows the effect of contention(σ) and coherency(κ) in the overall execution time, with sequential execution time of t_s, on N cores.

$$\begin{aligned} speedup &= \frac{t_s}{t} = \frac{N}{1+\sigma\times(N-1)+\kappa\times N(N-1)} \Rightarrow \\ t &= \frac{t_s}{N} + \sigma\times(N-1)\frac{t_s}{N} + \kappa\times N\times(N-1)\frac{t_s}{N} \end{aligned} \tag{5.5}$$

Formula 5.5 shows how USL models the overheads due to contention and coherency in execution time, when $\frac{t_s}{N}$ is the expected execution time on N cores in ideal case when the mentioned overheads did not exist.

Based on Formula 5.5, we added the term $\sigma\times(N-1)\times t_c$ to Formula 5.3 to represent the overhead observed due to contention effect, assuming t_c is the ideal execution time in this problem. We ignored the overhead due to coherency since we do not expect to experience any inter-process communication overheads in this problem.

$$execution\_time = \alpha \times \left\lceil \frac{num\_tasks}{N} \right\rceil + t\_overhead\_ws + t\_c + t\_c \times \sigma \times (N-1) \tag{5.6}$$

Moreover, we adjusted Formula 5.6 by changing the total number of cores (N) to the number of cores that are actually executing the work (M), since there are cases where there is not enough work for all the cores to execute, and we expect the overheads due to contention to be a factor of the cores that are actually performing the work and not just all the cores.

Assuming that we are running our application on N cores, with a grain size equal to g, num_tasks tasks are being created, and M cores are actually doing the work. If $num\_tasks < N$, M would be equal to num_tasks, otherwise $M = N$.

$$M = \begin{cases} num\_tasks & \text{if } num\_tasks < N \\ N & \text{otherwise} \end{cases} \tag{5.7}$$

Formula 5.6 is then converted into:

$$\begin{aligned} execution\_time = \alpha \times \left\lceil \frac{num\_tasks}{N} \right\rceil + t\_c + \sigma \times t\_c \times (M-1) \\ + t\_overhead\_ws \end{aligned} \tag{5.8}$$

As for t_c, the time to execute w_c amount of work, where total amount of work available is $problem\_size$, since we are looking into balanced parallel for-loops with almost same amount of work at each iteration, we can estimate t_c as:

$$t\_c = t\_seq \times \frac{w\_c}{problem\_size} \tag{5.9}$$

Where t_seq is the to run the whole amount of work, $problem\_size$ sequentially.

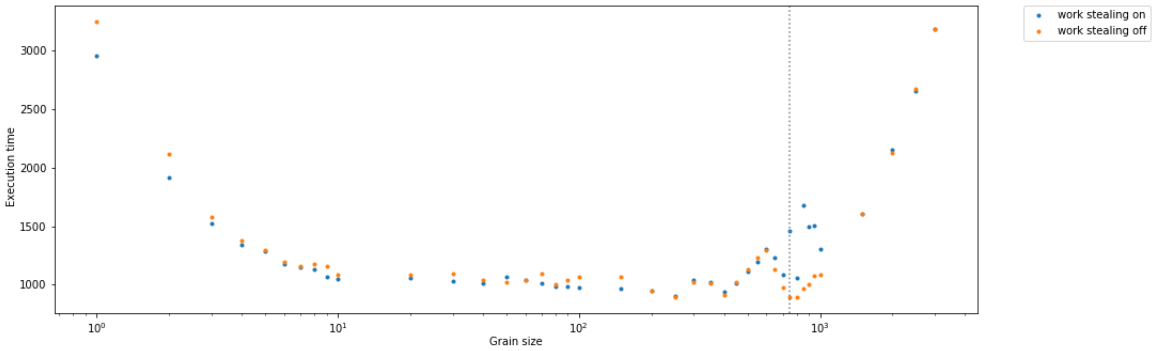
$$execution\_time = \alpha \times \left\lceil \frac{num\_tasks}{N} \right\rceil + t\_seq \times \frac{w\_c}{problem\_size} \times (1 + \sigma \times (M-1)) + t\_overhead\_ws \tag{5.10}$$

Here $t\_overhead\_ws$ represents all the overhead created due to work stealing. We did some investigations on where this term is originated from for a more precise modeling, but we were not successful.

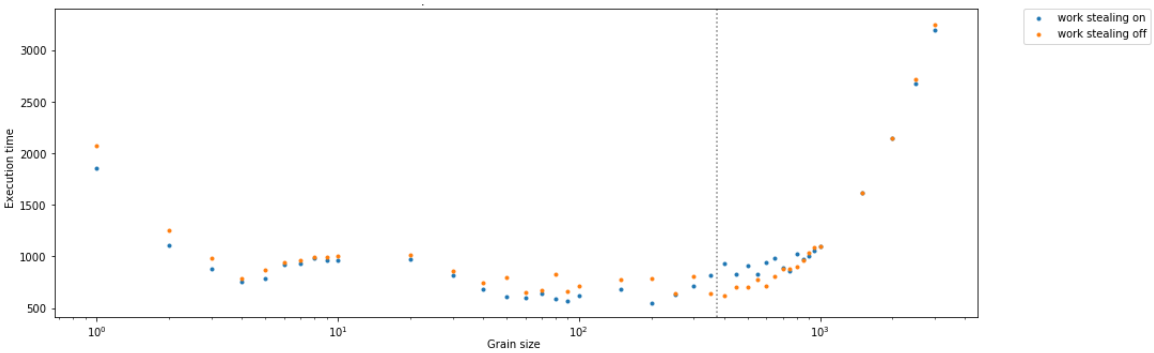
In order to study this effect, we ran a set of experiments while work stealing was turned of. This is possible through adding the command line option *–hpx:queuing = static-priority*.

Figure 5.2 shows the results of running a benchmark with and without work stealing. The benchmark consists of a HPX parallel for-loop with 3000 iterations and is executed with different number of chunk sizes from 1 to 3000. Each iteration will keep the core busy for $1\mu sec$. This benchmark is explained more in the next chapter.

As it can be observed, in the region where the number of tasks created is same as the number of cores, this effect is more significant.



(a)



(b)

Figure 5.2. The results of running the parallel for-loop benchmark with $problem\_size = 3000$, on different (a)4 cores, and (b) 8 cores. The vertical dotted line shows the grain size that would generate same number of tasks as the number of cores, with the same amount of work for all the cores.

Although we are not sure how work stealing is causing this effect, the effect is most noticeable at certain points. Since it only adds to the execution time at those points, and our interest is finding the flat region of the graph with the minimum execution time, at this point and for this purpose we decided to ignore this term in our upcoming calculations.

By ignoring the overhead caused by work stealing at this point, Formula 5.8 is simplified into Formula 5.11, which will be referred to as our proposed analytical model in the next sections.

$$
\begin{aligned}
execution\_time &= \alpha \times \left\lceil \frac{num\_tasks}{N} \right\rceil \; + \; t\_c \; + \; \sigma \times t\_c \times (M-1) \\
&= \alpha \times \left\lceil \frac{num\_tasks}{N} \right\rceil \; + \; t\_seq \times \frac{w\_c}{problem\_size} \times (1 \; + \; \sigma \times (M-1))
\end{aligned}
\tag{5.11}
$$

To summarize, we believe the important factors in determining the execution time in a balanced parallel for-loop are: number of tasks being created, number of cores the program is ran on, and the maximum amount of work one core has to perform.

The maximum number of tasks assigned to one core, and the number of cores that are actually performing the work, are two other important factors that can be deducted from the aforementioned factors.

5.1.1. Validating the Proposed Model

In order to validate the proposed model, we created a benchmark based on a simple *for_loop* with different number of iterations, and chunk sizes, as shown in Listing 5.1. We refer to this benchmark as the parallel for-loop benchmark from now on.

Each iteration consists of a while loop that makes sure the iteration lasts a certain amount of time ($iter\_length$). By setting $iter\_length = 1\mu sec$, and changing $num\_iterations$, and *chunk_size*, we can see how the execution time changes when the problem is executed on different number of cores. Having defined $problem\_size$ at the total amount of work that has to be performed, for this benchmark *problem_size* would be the time it takes to execute all the iterations, which is:

$$
problem\_size = iter\_length \times num\_iterations \tag{5.12}
$$

Since we have set $iter\_length = 1\mu sec$ in this benchmark, $problem\_size = num\_iterations$.

On the other hand, for this specific problem $t\_c = w\_c$ since there is no actual work happening $t\_seq = problem\_size$ and $t\_c = w\_c$.

In order to test how well the suggested model in Formula 5.11 could fit the data for this problem, we collected a considerable amount of data (around 50,000 data points on three different architectures) with different configurations of number of cores(N), number of iterations($num\_iterations$), and chunk size ($chunk\_size$).

For each $num\_iterations$, we change the $chunk\_size$ from 1 to $num\_iterations$ in logarithmic scale. Each of these runs was executed for $N = 1, 2, 3, ..., 8$.

Figure 5.3 shows an example of the results obtained from running the parallel for-loop benchmark in Listing 5.1, for $problem\_size = 100,000$, on 8 cores.

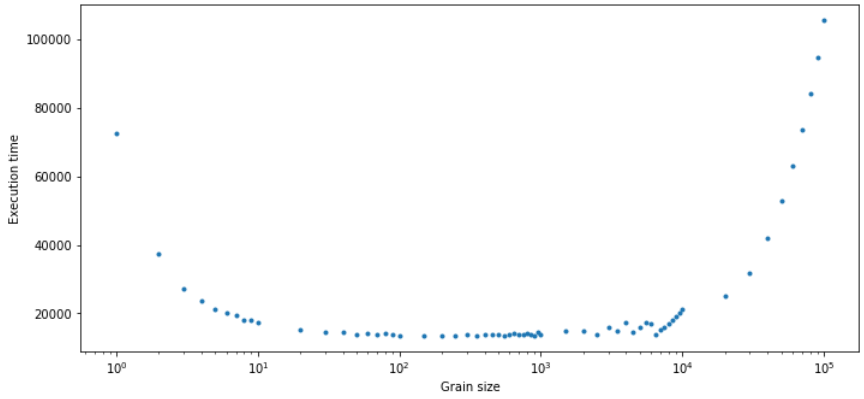


Figure 5.3. The results of running the parallel for-loop benchmark with $problem\_size = 100,000$, on 8 cores. The unit for execution time is microseconds.

As stated in Chapter 2, the graph of execution time vs grain size looks like a bathtub curve, with three major regions we refer to as left hand side, right hand side, and the flat region of the graph.

At the right hand side of the graph in with large grain sizes, the number of tasks created is smaller than the number of cores which results in making at least one of the cores idle, while the other cores are assigned a rather big chunk of work. The performance degradation we observe at those points is associated with starvation. Starvation refers to situations where we are not utilizing our computation resources to the full extent. At these points, the number of cores actually performing the work is equal to the number of tasks, since each core gets to execute at most one task.

At this region of the graph, the time to execute the maximum assigned work to a core (t_c)

Listing 5.1: For-loop benchmark, a simple hpx for_loop used to study the effect of grain size on the achieved parallelism.

```
1
2 ////////////////////////////////////////////////////////////////////////////
3 void measure_function_futures_for_loop(std::uint64_t count, bool csv, std::uint64_t chunk_size, std::uint64_t
      iter_length)
4 {
5   // start the clock
6   high_resolution_timer walltime;
7   hpx::parallel::for_loop(hpx::parallel::execution::par.with(
8   hpx::parallel::execution::dynamic_chunk_size( chunk_size )),
9     0, count, [&](std::uint64_t) { worker_timed(iter_length*1000); });
10  // stop the clock
11  const double duration = walltime.elapsed();
12  print_stats("for_loop", "par", "parallel_executor", count, duration, csv);
13 }
14
15 ////////////////////////////////////////////////////////////////////////////
16 int hpx_main(variables_map& vm)
17 {
18   const int repetitions = vm["repetitions"].as<int>();
19   num_threads = hpx::get_num_worker_threads();
20   const std::uint64_t chunk_size = vm["chunk_size"].as<std::uint64_t>();
21   const std::uint64_t iter_length = vm["iter_length"].as<std::uint64_t>();
22   const std::uint64_t count = vm["num_iterations"].as<std::uint64_t>();
23   bool csv = vm.count("csv") != 0;
24   if (HPX_UNLIKELY(0 == count))
25     throw std::logic_error("error: count of 0 futures specified\n");
26   for (int i = 0; i < repetitions; i++)
27   {
28     measure_function_futures_for_loop(count, csv, chunk_size, iter_length);
29   }
30   return hpx::finalize();
31 }
32 ////////////////////////////////////////////////////////////////////////////
33 inline void worker_timed(std::uint64_t delay_ns)
34 {
35   if (delay_ns == 0)
36     return;
37   std::uint64_t start = hpx::util::high_resolution_clock::now();
38   while (true)
39   {
40     // Check if we've reached the specified delay.
41     if ((hpx::util::high_resolution_clock::now() - start) >= delay_ns)
42       break;
43   }
44 }
45 ////////////////////////////////////////////////////////////////////////////
46 int main(int argc, char* argv[])
47 {
48 // Configure application-specific options.
49   options_description cmdline("usage: " HPX_APPLICATION_STRING " [options]");
50   cmdline.add_options()("num_iterations",
51     value<std::uint64_t>()->default_value(500000), "number of iterations to invoke")
52     ("repetitions", value<int>()->default_value(1),
53     "number of repetitions of the full benchmark")
54     ("iter_length",value<std::uint64_t>()->default_value(1), "length of each iteration")
55     ("chunk_size",value<std::uint64_t>()->default_value(1), "chunk size");
56   // Initialize and run HPX.
57   return init(cmdline, argc, argv);
58 }
```

is the dominant factor in determining the execution time.

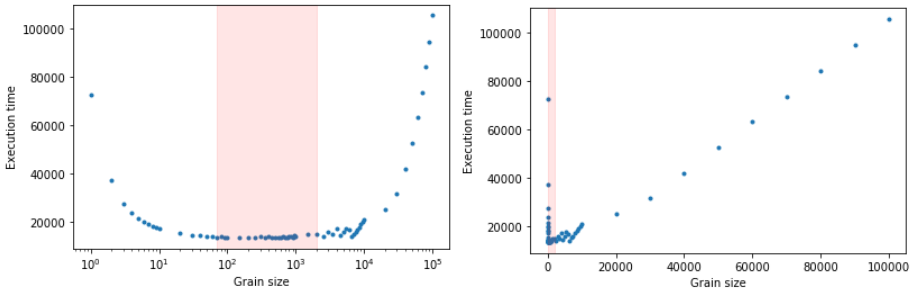
On the other hand, on the left hand side of the graph with small grain sizes, we end up with creating a large number of tasks. Since there is an overhead associated with each created task, we observe a performance degradation in that region. As the grain size increases, the number of created tasks, and the overhead associated to that decreases consequently. At this region of the graph, the overhead of creating and managing the tasks is the dominant factor in determining the execution time.

The third region of the graph, which we referred to as the flat region of the graph, is the area between the two mentioned areas where we observe very little change in the execution time. At this region, the overhead associated with creating and managing the tasks is amortized by the execution time and since all the cores are utilized we do not observe he effect a performance degradation due to starvation.

Our intention in this thesis is to find this flat region of the bathtub curve, as shown in Figure 5.4a, in order to ensure that the effect of the two major sources of performance degradation (overheads of creating and managing tasks, and starvation) in execution time is minimized.

In Figure 5.4 grain size is shown in logarithmic scale, but looking at Figure 5.4, with normal scale for grain size, it can be realized that the range we are interested in is a small range of values of grain size.

In this thesis we propose a new methodology to study the performance based on task granularity for a balanced parallel for loop, and we suggest an approach to locate this small region of grain size in order to achieve the minimum execution time.



(a) (b)

Figure 5.4. The results of running the parallel for-loop benchmark with $problem\_size$ = 100000, on 8 cores. (a) represents the graph in logarithmic scale, while (b) shows the same graph in linear scale. The unit for execution time is microseconds.

Since our proposed model does not depend on the $problem\_size$ but relies on the number of tasks created, number of cores, and the time to execute the maximum amount of work assigned to a core, we are suggesting here that it might be sufficient to use the data collected from one $problem\_size$ to find the parameters α and σ.

In order to test the viability of our proposition, we selected 5 different $problem_size$s (10000, 100000, 1000000, 10000000, 100000000) to use as our base $problem\_size$ for finding the parameters of our model(α and σ). Using all the collected data points for each *problem_size*, we utilized the *optimize.curve_fit* package from $scipy$ library in Python to fit our model to the collected data.

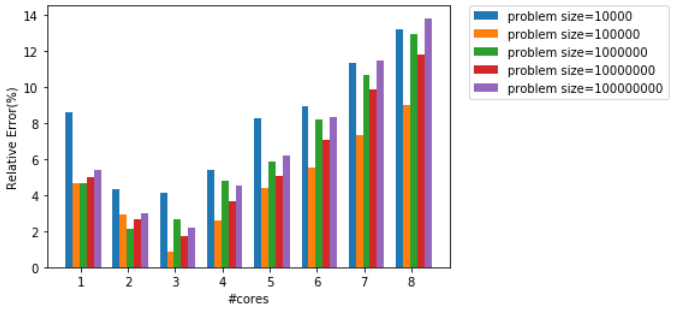
For any of the base *problem_sizes*, we measured the relative error and R^2 score of prediction for the base $problem\_size$ itself as shown in Formula 5.13 and Formula 5.14. p_i is the predicted value from curve fitting for sample i, t_i is true value of that sample, and n is the total number of samples. The relative error and R^2 score are calculated for each specific number of cores individually.

$$Relative\_error = \frac{1}{n}\sum_{i=1}^{n}|1-\frac{p_i}{t_i}| \tag{5.13}$$

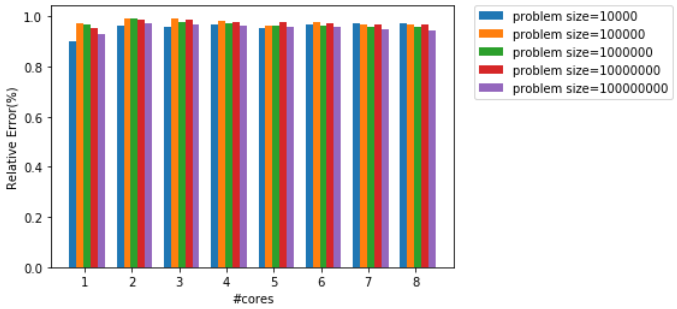
$$R^2 = 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}(t_i-p_i)^2}{Var(t)} \tag{5.14}$$

Figure 5.5 illustrates the calculated relative error and R^2 score for *problem_size=10000, 100000, 1000000, 1000000, 100000000*. For each of the $problem_size$s, 440, 512, 440, 512, and 728 data points were collected respectively.

As the $problem\_size$ increases, we need to collect more data points to cover different regions of the grain size, which results in longer total execution time of the benchmark. In order to rule out the uncertainties in our experiment as much as possible, we run each benchmark 6 times and take the average of the measured execution times. This means even larger overall execution time for data collection for larger $problem\_sizes$.



(a)

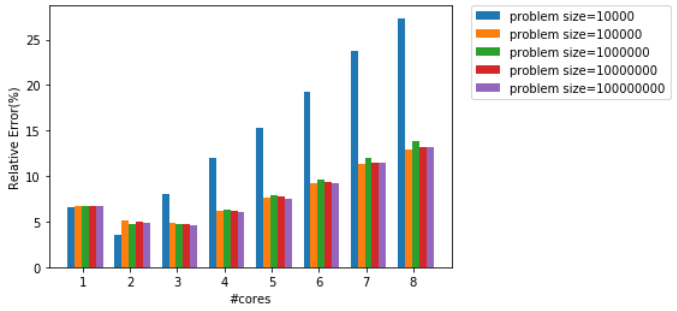


(b)

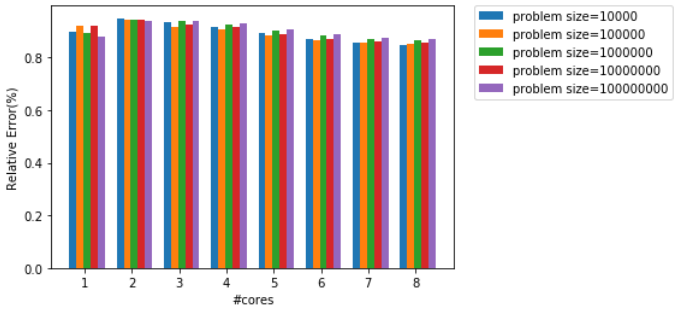
Figure 5.5. The (a) relative error, and (b) R^2 score of fitting the collected data to the proposed analytical model for different $problem\_sizes$, calculated for each number of cores.

Figure 5.5 shows that for each number of cores, the calculated errors for each $problem\_size$ are very close to each other, and the the least amount of relative error was obtained when $problem\_size$ = 100000, which is most likely due to higher amount of data collected for this specific $problem\_size$.

In the next step we measured the relative error and R^2 score of using the model parameters found from each base $problem\_size$ to predict the execution time for all other $problem\_sizes$. The result is illustrated in Figure 5.6.



(a)



(b)

Figure 5.6. The (a)relative error and (b)R^2 score of using the model parameters found from each base $problem\_size$ fitting the proposed analytical model to the collected data for different $problem\_sizes$, calculated over each number of cores.

Based on Figure 5.4 and Figure5.6, we selected $problem\_size = 100000$ as the base*problem_size*, since it shows similar prediction error and R^2 score as larger $problem\_sizes$, while taking less time to complete.

Having selected $problem\_size = 100000$ as a reasonable(not too small nor too large) base for our parameter estimation, Figure 5.7 shows the measured execution time in terms of grain size for $problem\_size = 100000$, with $N = 1, 2, ..., 8$.

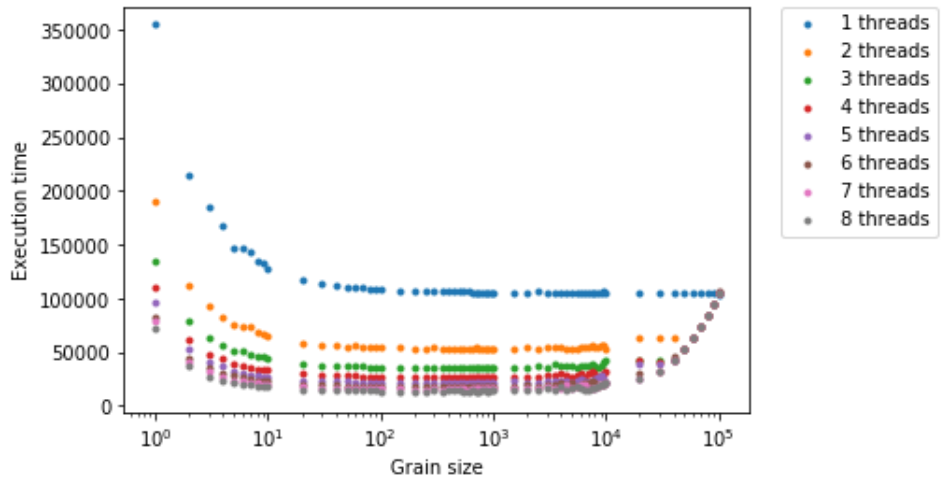


Figure 5.7. The results of running the parallel for-loop benchmark with $problem\_size$ = 100000, on different number of cores. The unit for execution time is microseconds.

Fitting our model to all the 512 data points collected resulted in model parameters $alpha$ = 2.674 and σ = 0.0268, where α represents the overhead of creating and managing the tasks in $\mu secs$, while σ represents the contention based on Universal Scalibility Law.

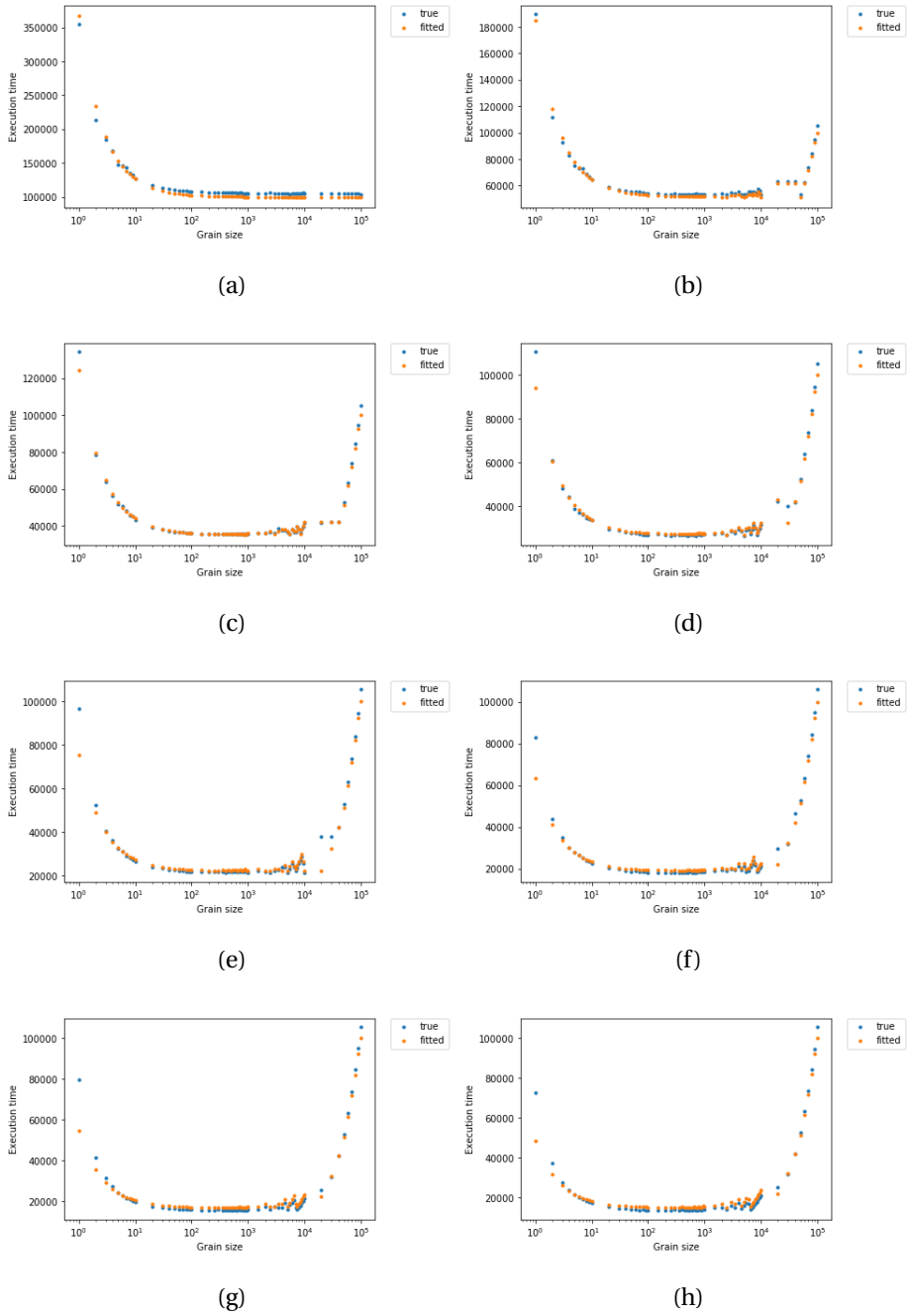


Figure 5.8. The results of predicted values of execution time through curve fitting vs the real data for $problem\_size = 100000$, for (a) 1 core, (b) 2 cores, (c) 3 cores, (d) 4 cores, (e) 5 cores, (f) 6 cores, (g) 7 cores, (h) 8 cores. The unit for execution time is microseconds.

Figure 5.8 shows the fitted curves along with the original data for different number of cores.

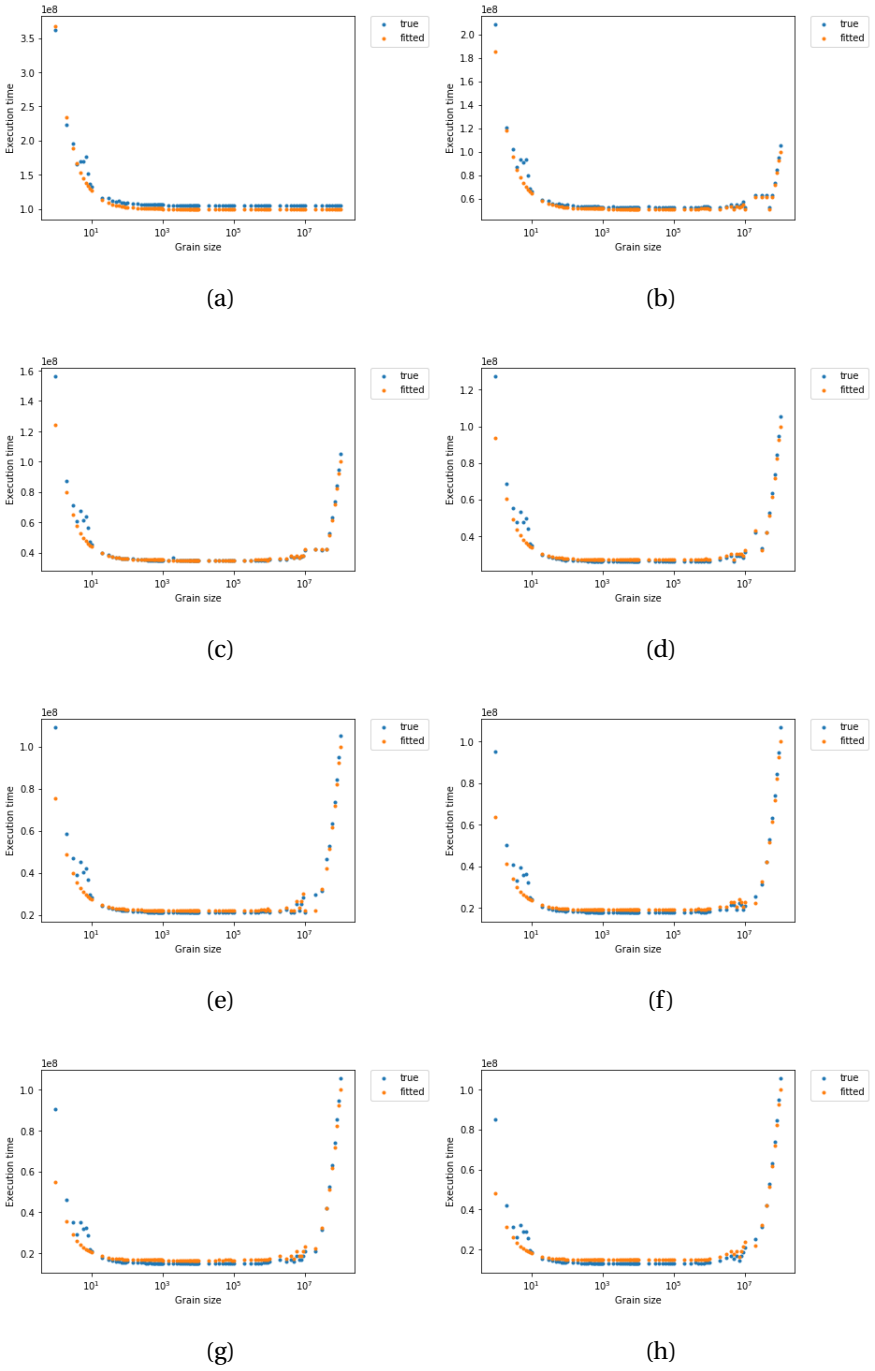


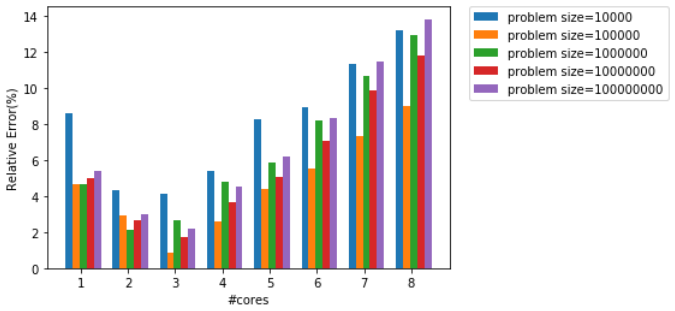
Figure 5.9. The results of predicted values of execution time through curve fitting vs the real data for $problem\_size$ = 100000000, for (a) 1 core, (b) 2 cores, (c) 3 cores, (d) 4 cores, (e) 5 cores, (f) 6 cores, (g) 7 cores, (h) 8 cores. The unit for execution time is microseconds.

5.1.2. Testing the Proposed Model on Other Architectures

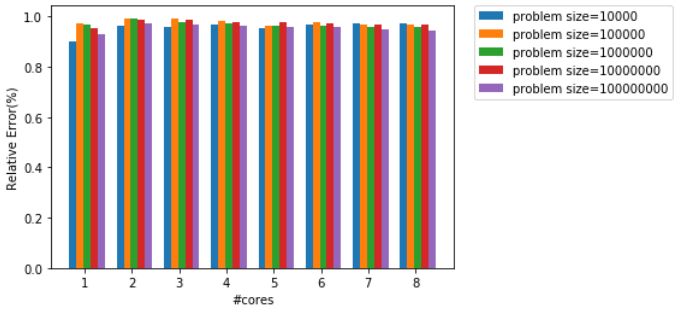
At the next step we repeated the same process for the data collected from two different architectures, *Medusa* node with a Skylake CPU, and a Raspberry Pi 4.

5.1.3. Medusa

We used $problem\_size = 100000$ as the base *problem_size* on both architectures. On Medusa, with $problem\_size = 100000$, the model parameters were found to be $\alpha = 1.832$ and $\sigma = 0.037$. The calculated relative error and R^2 score for each individual number of cores for $problem\_size = 100000$ is demonstrated in Figure 5.10, and Figure 5.11, while the relative error and R^2 score calculated for all other *problem_sizes* is shown in Figure 5.10, and Figure 5.11.

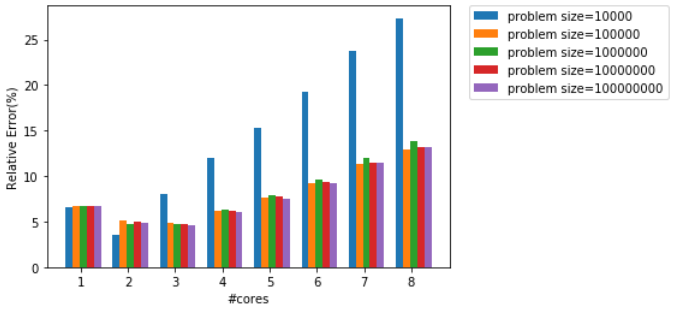


(a)

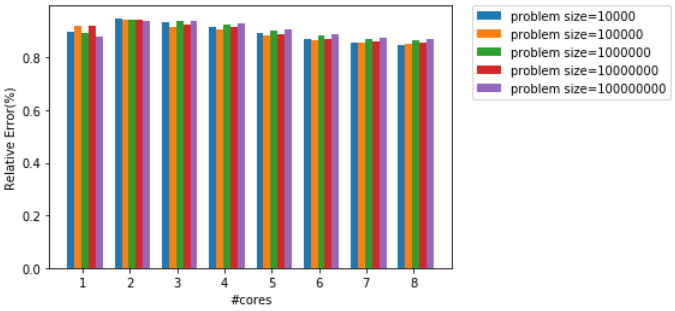


(b)

Figure 5.10. The (a) relative error, and (b) R^2 score of fitting the collected data to the proposed analytical model for different $problem\_sizes$, calculated for each number of cores.



(a)



(b)

Figure 5.11. The (a)relative error and (b)R^2 score of using the model parameters found from each base $problem\_size$ fitting the proposed analytical model to the collected data for different $problem\_sizes$, calculated over each number of cores.

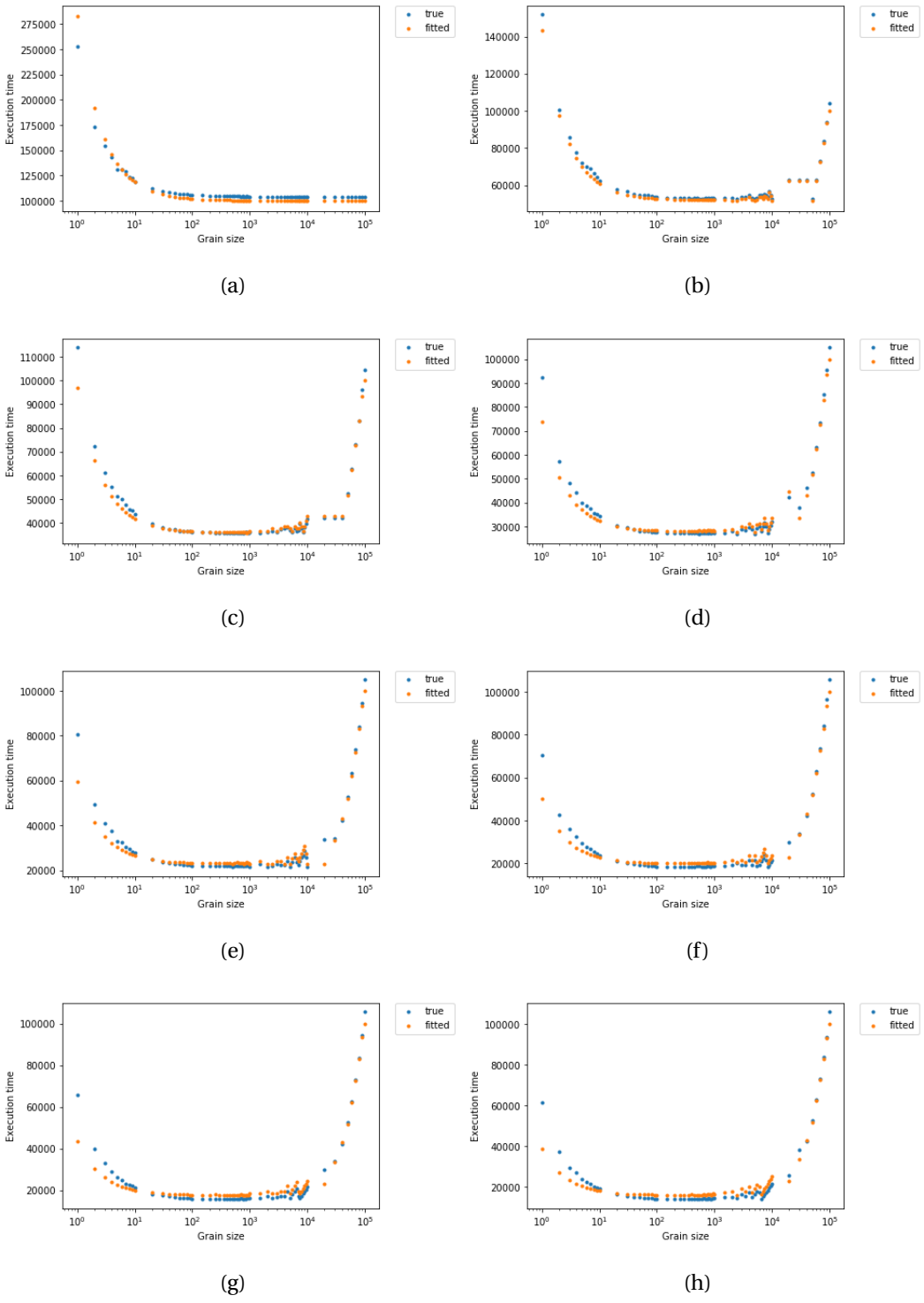


Figure 5.12. The results of predicted values of execution time through curve fitting vs the real data for $problem\_size = 100000$, for (a) 1 core, (b) 2 cores, (c) 3 cores, (d) 4 cores, (e) 5 cores, (f) 6 cores, (g) 7 cores, (h) 8 cores. The unit for execution time is microseconds.

5.1.4. Raspberry Pi 4

5.1.5. Analyzing the Data

Assuming the proposed formula is a good fit for this problem, we can estimate the range of grain sizes for which we achieve the lowest execution time, for a specific problem size, ran on specific number of cores.

5.1.5.1. Left side of the Graph

As stated earlier, in Formula 5.8, for small grain sizes the first term is the dominant factor while the second term roughly stays constant. Same way, for large grain sizes the second factor is the dominant factor.

In order to find the lower-bound of the range for which the execution time stays constant, we assume the second factor is constant in that region. Also we can change N to M, knowing that our concern is on the left hand side of the graph, where num_tasks is definitely greater than the number of cores. Taking the derivative of the function based on the grain size then leads to:

$$\begin{aligned} \frac{\partial execution\_time}{\partial g} &= \frac{\alpha}{N} \times \frac{\partial num\_tasks}{\partial g} \\ &= \frac{\alpha}{N} \times \frac{\partial(\frac{problem\_size}{g})}{\partial g} \\ &= \frac{\alpha}{N} \times problem\_size \times \frac{-1}{g^2} \end{aligned} \tag{5.15}$$

From Formula 5.15, it can be observed that for the left hand side of the graph the rate of changes is negative and decreases as the grain size increases. Here we are looking for the value of the grain size for which the rate of change becomes very small (we introduce a threshold λ_b, where $\lambda_b \ll 1$, for this purpose).

$$\begin{aligned} \frac{\alpha}{N} \times problem\_size \times \frac{1}{g^2} &\leq \lambda_b \\ g^2 &\geq \frac{\frac{\alpha}{N} \times problem\_size}{\lambda_b} \\ g &\geq \sqrt{\frac{\frac{\alpha}{N} \times problem\_size}{\lambda_b}} \end{aligned} \tag{5.16}$$

Formula 5.16 can also be represented as shown in Formula 5.17. This representation shows that when the ratio of the time it takes to execute one task to the total overhead of creating and managing num_tasks tasks on N core, is greater than a threshold, we will end up in the flat region of the graph, close to the left hand side.

$$\begin{aligned} \frac{\alpha}{N} \times \frac{problem\_size}{g} \times \frac{1}{g} &\leq \lambda_b \\ \frac{\alpha}{N} \times num\_tasks \times \frac{1}{g} &\leq \lambda_b \\ \frac{\alpha}{N} \times num\_tasks &\leq g \times \lambda_b \\ \frac{g}{\frac{\alpha}{N} \times num\_tasks} &\geq \frac{1}{\lambda_b} \end{aligned} \tag{5.17}$$

5.1.5.2. Right side of the graph

Now looking at the right hand side of the graph, the overhead of creating the tasks becomes negligible on that side, since only few tasks are being created and the overhead of creating these many tasks is not significant compared to the execution time. On this side, the maximum amount of work executed by one core (w_c) and consequently the time to perform this amount of work (t_c) is the dominant factor.

Formula 5.4 shows how w_c is calculated for different cases, but in general we can estimate w_c with $g \times \left\lceil \frac{num\_tasks}{N} \right\rceil$.

$$
\begin{aligned}
w\_c &\approx g \times \left\lceil \frac{num\_tasks}{N} \right\rceil \\
&\approx g \times \left\lceil \frac{\frac{problem\_size}{g}}{N} \right\rceil \\
&\approx \frac{problem\_size}{N} \quad \text{if } num\_tasks \geq N
\end{aligned}
\tag{5.18}
$$

What happens here is that as the grain size changes, there are points for which $\left\lceil \frac{num\_tasks}{N} \right\rceil$ is the same but since the grain size is different, a different w_c would be resulted. For all the values of g that create the same $\left\lceil \frac{num\_tasks}{N} \right\rceil$, as g increases the difference between w_c and $\frac{problem\_size}{N}$ increases.

For example, considering a case where $problem\_size = 100,000$, and $N = 8$, for the grain sizes in range of [4167, 6249] would result in creating between $\left\lceil \frac{100,000}{6,249} \right\rceil = 17$ and $\left\lceil \frac{100,000}{4,167} \right\rceil = 24$ tasks. This amount of tasks created itself would result in $\lceil \frac{17}{8} \rceil = 3$ and $\lceil \frac{24}{8} \rceil = 3$ tasks. On the other hand, $w\_c = g \times \left\lceil \frac{num\_tasks}{N} \right\rceil = 3 \times g$, would have a value in range of $[3 \times 4167, 3 \times 6249] = [12501, 18747]$, where the average amount of work per core is $\frac{problem\_size}{N} = 12500$. This means that for grain sizes closer to the end of the range, we are observing that a much bigger amount of work is assigned to the core with maximum amount of work, which would result in a higher execution time.

In the general case, if we denote $\left\lceil \frac{num\_tasks}{N} \right\rceil$ as k, then:

$$
\begin{aligned}
k - 1 &< \frac{num\_tasks}{N} \leq k \\
(k-1) \times N &< num\_tasks \leq k \times N \\
(k-1) \times N &< \left\lceil \frac{problem\_size}{g} \right\rceil \leq k \times N \\
(k-1) \times N &< \frac{problem\_size}{g} \leq k \times N
\end{aligned}
\tag{5.19}
$$

If $k = 1$, then,

$$
\begin{aligned}
0 < \frac{problem\_size}{g} &\leq N \\
\frac{problem\_size}{N} &\leq g \leq problem\_size.
\end{aligned}
\tag{5.20}
$$

Otherwise, when $k > 1$,

$$
\frac{problem\_size}{k \times N} \leq g < \frac{problem\_size}{(k-1) \times N}. \tag{5.21}
$$

Since $\left\lceil \frac{num\_tasks}{N} \right\rceil = k$, and $w\_c = g \times \left\lceil \frac{num\_tasks}{N} \right\rceil = k \times g$ if $num\_tasks\%N \neq 1$, we can conclude for $k > 1$:

$$
\begin{aligned}
k \times \frac{problem\_size}{k \times N} \leq w\_c < k \times \frac{problem\_size}{(k-1) \times N} \\
\frac{problem\_size}{N} \leq w\_c < \frac{k}{k-1} \times \frac{problem\_size}{N} \\
0 \leq w\_c - \frac{problem\_size}{N} < \frac{1}{k-1} \times \frac{problem\_size}{N}
\end{aligned}
\tag{5.22}
$$

For the cases where $k > 1$ and $num\_tasks\%N = 1$, there could be a change in w_c if $problem\_size\%g \neq 0$. For these cases:

$$
\begin{aligned}
\left\lceil \frac{num\_tasks}{N} \right\rceil = k \ \ \& \ \ num\_tasks\%N = 1 \Rightarrow \\
num\_tasks = (k-1) \times N + 1 \Rightarrow \\
(k-1) \times N < \frac{problem\_size}{g} \leq (k-1) \times N + 1 \Rightarrow \\
\frac{problem\_size}{(k-1) \times N + 1} \leq g < \frac{problem\_size}{(k-1) \times N}
\end{aligned}
\tag{5.23}
$$

From Formula 5.4 we know,

$$
w\_c = problem\_size - (k-1) \times (N-1) \times g. \tag{5.24}
$$

Therefore,

$$
\begin{aligned}
(k-1)(N-1)\frac{problem\_size}{(k-1)N+1} \leq (k-1)(N-1)g < (k-1)(N-1)\frac{problem\_size}{(k-1)N} \Rightarrow \\
\frac{problem\_size}{N} < problem\_size - (k-1)(N-1)g \leq k\frac{problem\_size}{(k-1)N+1} \\
\frac{problem\_size}{N} < w\_c \leq k \times \frac{problem\_size}{(k-1)N+1}
\end{aligned}
\tag{5.25}
$$

And for $k = 1$, where $num\_tasks \leq N$,

$$
\begin{aligned}
w\_c = g \\
\frac{problem\_size}{N} \leq g \leq problem\_size \Rightarrow
\end{aligned}
$$

$$
0 \leq w\_c - \frac{problem\_size}{N} = g - \frac{problem\_size}{N} \leq (N-1) \times \frac{problem\_size}{N} \tag{5.26}
$$

Defining $imbalance\_ratio = \frac{w\_c - \frac{problem\_size}{N}}{\frac{problem\_size}{N}}$, then,

$$
\begin{aligned}
&0 \leq imbalance\_ratio \leq N-1 && \text{for } k = 1 \\
&0 \leq imbalance\_ratio < \frac{1}{k-1} && \text{for } k > 1 \text{ and } num\_tasks\%N \neq 1 \\
&0 \leq imbalance\_ratio < \frac{N-1}{N(k-1)+1} = \frac{1}{k-1+\frac{k}{N-1}} && \text{otherwise}
\end{aligned}
\tag{5.27}
$$

Formula 5.27 shows that as number of created tasks increases, as long as number of tasks per core is the same, the imbalance factor decreases.

Figure 5.13 shows the imbalance ratio calculated for different grain sizes for $problem\_size = 10000$, on 8 cores. Each of the regions between two dashed green lines correspond to a specific value for $k = \left\lceil \frac{num\_tasks}{N} \right\rceil$. At each of the regions with $k > 1$, $\left\lceil \frac{num\_tasks}{N} \right\rceil = k$, $imbalance\_ratio$

starts from 0 and approaches $\frac{1}{k-1}$ ($\frac{1}{k-1+\frac{k}{N-1}}$ for regions where $num\_tasks\%N \neq 1$) at the end of the region. When $k = 1$, $imbalance\_ratio$ increases linearly starting from 0 and reaching the maximum of $N-1$ when $g = pronblem\_size$. As we move to larger grain sizes, $\left\lceil \frac{num\_tasks}{N} \right\rceil$ decreases, therefore the upper-bound for $imbalance\_ratio$ increases.

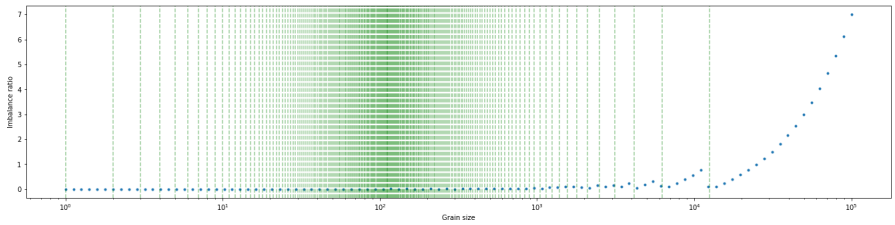


Figure 5.13. The imbalance ratio calculated for different grain sizes for $problem\_size$ = 10000, on 8 cores. At the area between each two green lines $k = \left\lceil \frac{num\_tasks}{N} \right\rceil$ is constant.

Figure 5.14 represents the imbalance ratio, along with the ratio of the sequential execution time over execution time(speed-up) against grain size for $problem\_size$ = 10000, ran on 8 cores. As it can be observed, as the $imbalance\_ratio$ increases, the speed-up decreases.

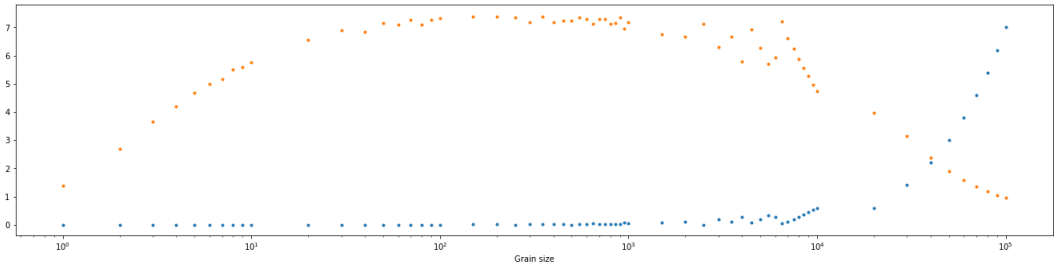


Figure 5.14. The imbalance ratio alongside speedup for different grain sizes for $problem\_size$ = 10000, on 8 cores, where $k = \left\lceil \frac{num\_tasks}{N} \right\rceil$.

To summarize, as the grain size increases the maximum imbalance in the loads assigned to the cores also increases, and some point on, this imbalance has a significant affect in the execution time. We define a threshold, λ_s ($0 < \lambda_s < 1$), where for $imbalance_ratio$s smaller than this threshold the imbalance effect is not significant. As we get close to this threshold, we are likely to reach the right hand side of the flat region of the bathtub curve of the execution time against grain size.

We are interested in finding the maximum grain size that would generate a reasonable imbalance ($imbalance\_ratio \leq \lambda_s$), to make sure we should stay in the flat region of the bathtub curve of execution time against grain size, from load imbalance point of view.

Formula 5.26 states that for grain sizes greater than $problem\_size$, $imblance\_ratio$ increases linearly with grain size from 0 to $N-1$. While for grain sizes smaller than $problem\_size$, the maximum $imbalance\_ratio$ depends on $k = \left\lceil \frac{num\_tasks}{N} \right\rceil$. So, in order to ensure *imbalance_ratio* is smaller than or equal to a threshold (λ_s), first we search the grain sizes smaller than $\frac{problem\_size}{N}$. Since $0 < \lambda_s < 1$, and $k \geq 2$ in this region, there exists a k such that $\frac{1}{k-1} \leq \lambda_s$. If there exists a k_{min} (creating an imbalance ratio between 0 and $\frac{1}{k_{min}-1}$), where $\frac{1}{k_{min}-1} \leq \lambda_s$, $\forall k < k_{min}$ maximum value of $imbalance\_ratio$ would be greater than λ_s. So in order to find the grain size that would create maximum $imbalance\_ratio$ of λ_s:

$$
\begin{aligned}
& imbalance\_ratio \leq \lambda_s \Rightarrow \\
& \frac{1}{k-1} \leq \lambda_s \\
& k \geq 1 + \frac{1}{\lambda_s} \\
& k_{min} = \left\lceil 1 + \frac{1}{\lambda_s} \right\rceil + 1 \\
& g < \frac{problem\_size}{(k_{min}-1) \times N} \\
& g_{max} = \frac{problem\_size}{(k_{min}-1) \times N} - 1 = \frac{problem\_size}{(1 + \left\lceil \frac{1}{\lambda_s} \right\rceil) \times N}
\end{aligned}
\tag{5.28}
$$

If $g < g\_max$, we can ensure that $imbalance\_ratio$ never exceeds λ_s. Since we already found a match at grain sizes smaller than $\frac{problem\_size}{N}$, checking the rest of grain sizes would not be necessary.

5.1.6. Identifying the Range of Grain Size for Minimum Execution Time

In the previous section, we proposed a method to identify the lower-bound and the upper-bound of the grain sizes for which we expect to observe the minimum execution time. Inte-

grating Formula 5.16 and Formula 5.28 suggests the following range for minimum execution time:

$$\sqrt{\frac{(\frac{\alpha}{N}) \times problem\_size}{\lambda_b}} \leq g \leq \frac{problem\_size}{(1 + \left\lceil \frac{1}{\lambda_s} \right\rceil) \times N} \tag{5.29}$$

Where $0 \leq \lambda_s \leq 1$, and $\lambda_b, \lambda_s \ll 1$.

Note that based on Formula 5.29 we can state,

$$\sqrt{\frac{(\frac{\alpha}{N}) \times problem\_size}{\lambda_b}} \leq g \leq \frac{problem\_size}{(1 + \left\lceil \frac{1}{\lambda_s} \right\rceil) \times N} < \frac{problem\_size}{N}. \tag{5.30}$$

Which shows that the identified range of minimum execution time is located at the left side of the grain size that equally divides the work between the cores, $\frac{problem\_size}{N}$.

5.1.7. Locating the Flat Region of the Execution time vs Grain Size Graph for the Benchmark

In this section we used Formula 5.29 to identify the flat region of the execution time vs grain size graph for the parallel for-loop benchmark in Listing 5.1. For this purpose, we set both λ_b and λ_s to 0.1. The minimum and maximum acceptable value for grain size to lay in the flat region of the graph are then calculated based on Formula 5.30. λ_b indicates the slope of the graph at the left hand side of the graph where overhead of tasks is the dominant factor. $\lambda_b = 0.1$ shows an angle of around 5ř, which seems to be a reasonable threshold. Grain sizes smaller than $\sqrt{\frac{(\frac{\alpha}{N}) \times problem\_size}{\lambda_b}}$ would create a slope of more than λ_b. As for λ_s, a grain size greater than $\frac{problem\_size}{(1+\left\lceil \frac{1}{\lambda_s} \right\rceil) \times N}$ could generate an $imbalance\_ratio$ of greater than λ_s. Therefore, for grain sizes within the range in Formula 5.30 we are in a safe region.

In Figure 5.15 the identified regions for minimum execution time is shown in green, for $problem\_size$ = 10000, 100000, 1000000, 10000000, 100000000, executed on 8 cores. The gray dashed line represents the grain size where work is equally divided among the cores,

$\frac{problem\_size}{N}$.

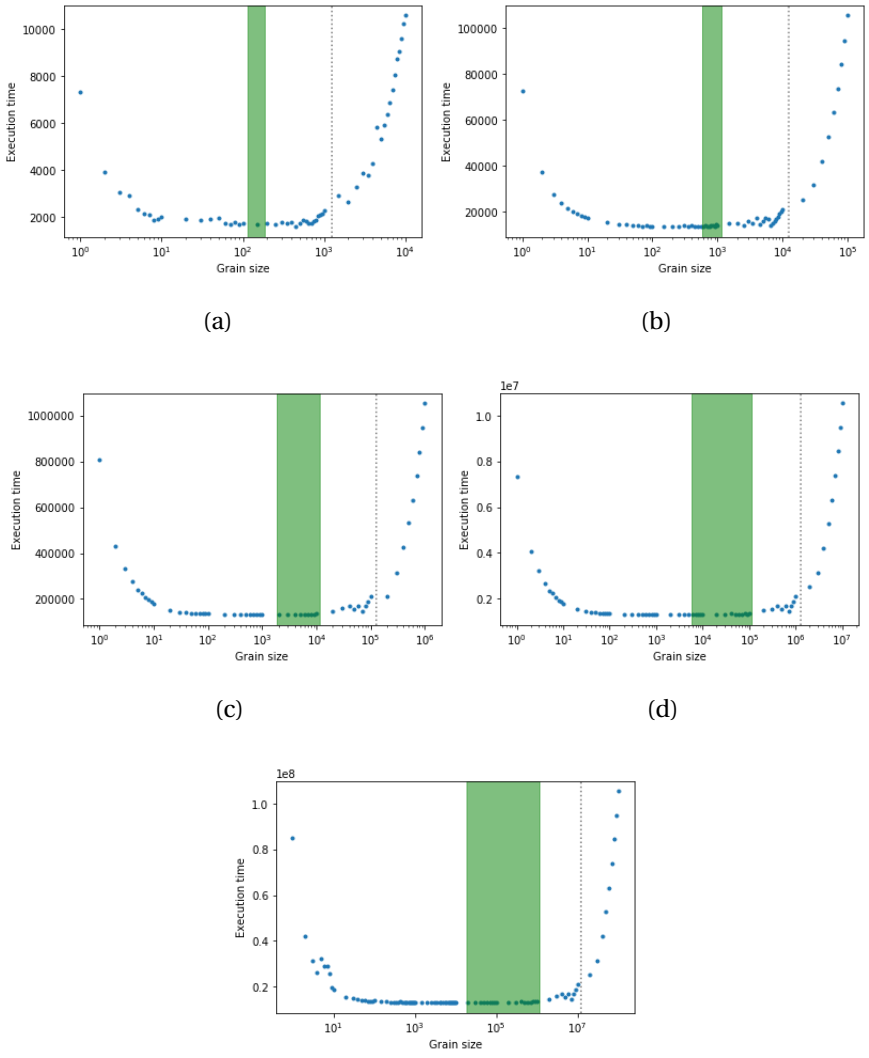


Figure 5.15. The identified range of grain size for minimum execution time for $problem\_size$ = (a) 10000, (b) 100000, (c) 1000000, (d) 10000000, (e) 100000000, on 8 cores, with $\lambda_s = 0.1$ and $\lambda_b = 0.1$. The gray dashed line represents the grain size where work is equally divided among the cores, $\frac{problem\_size}{N}$. The unit for execution time is microseconds.

Selecting a greater value for λ_b would move the left border of the region to left, for a larger acceptable slope of change of execution time in terms of grain size. On the other hand, selecting a smaller value for λ_s would result in shifting the right border of the region to left, imposing a higher restriction on $imbalance\_ratio$, as shown in Figure 5.16.

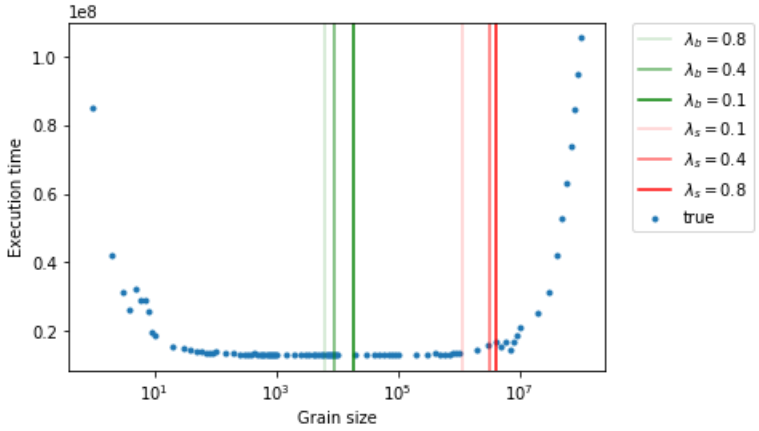


Figure 5.16. An example of the effect of λ_b and λ_s on the borders of the identified region for minimum execution time due to change for $problem\_size = 100000000$ and $N = 8$.

5.1.8. Applying the Proposed Model to the Blazemark Data

In the previous section we proposed an analytical model to predict how execution time changes with grain size, and suggested a method to estimate the range of grain size for a specific $problem\_size$ ran on a certain number of cores to stay in the region of minimum execution time. We then tested the proposed model on a simple parallel for-loop benchmark, an the results seemed promising. In this section we revisit our original problem of finding the range of chunk size for minimum execution time to calculate an expression using the Blaze linear algebra library. We start by looking at the data collected from $DMATDMATADD$ benchmark, which represents addition of two row major square dense matrices.

As we previously stated in Chapter 4, for the collected data, the *DMATDMATADD* benchmark was executed with different configurations of $block\_size$, $chunk\_size$, and number of cores, for different matrix sizes. As discussed earlier the collected data suggested that instead of looking at $block\_size$ and then $chunk\_size$, we can study the effect of grain size on execution time. By identifying the range of grain size that ensures staying in the flat region of the bathtub, we can then select a reasonable value for $block\_size$, we can find the range of $chunk\_sizes$ to generate the mentioned range of grain size. We define a reasonable value for block size for an operation on row major matrices, az the size of a matrix with larger number of columns than number of rows, where number of columns is divisible by cache line, and

the number of elements in the block multiplied by number of matrices involved in an operation, is smaller than L2 cache size. With these assumptions we selected the value of 4×256 for $block\_size$ for this benchmark.

In the previous section we suggested Formula 5.8 as an analytical model to explain the relationship between execution time and grain size in a balanced parallel for-loop.

$$execution\_time = \alpha \times \left\lceil \frac{num\_tasks}{N} \right\rceil + t\_seq \times \frac{w\_c}{problem\_size} \times (1 + \sigma \times (M-1)) \tag{5.31}$$

$problem\_size$ in Formula 5.8 refers to the total amount of work available. For $matrix\_size = m$, performing $DMATDMATADD$ benchmark, $problem\_size = 2 \times m^2$, which is the total amount of floating point operations needed to be performed.

Previously, using the benchmark we were able to find the value for the parameters of the model on $Marvin$ node as $\alpha = 2.674$ and $\sigma = 0.0268$.

Grain size, as the amount of work executed by one thread, for this problem is represented by the number of floating point operations performed by one thread. As an example, if $block\_size = 4 \times 256$, and $chunk\_size = 3$ performing $DMATDMATADD$ benchmark, would result in a $grain\_size$ of $2 \times 4 \times 256 \times 3 = 6144$.

Figure 5.17 and Figure 5.18 show the results of applying the proposed analytical model with the parameters identified through the benchmark, to $DMATDMATADD$ benchmark, alongside the true values for the execution time, for $matrix\_size = 690$ and $matrix\_size = 4222$ respectively, for different number of cores.

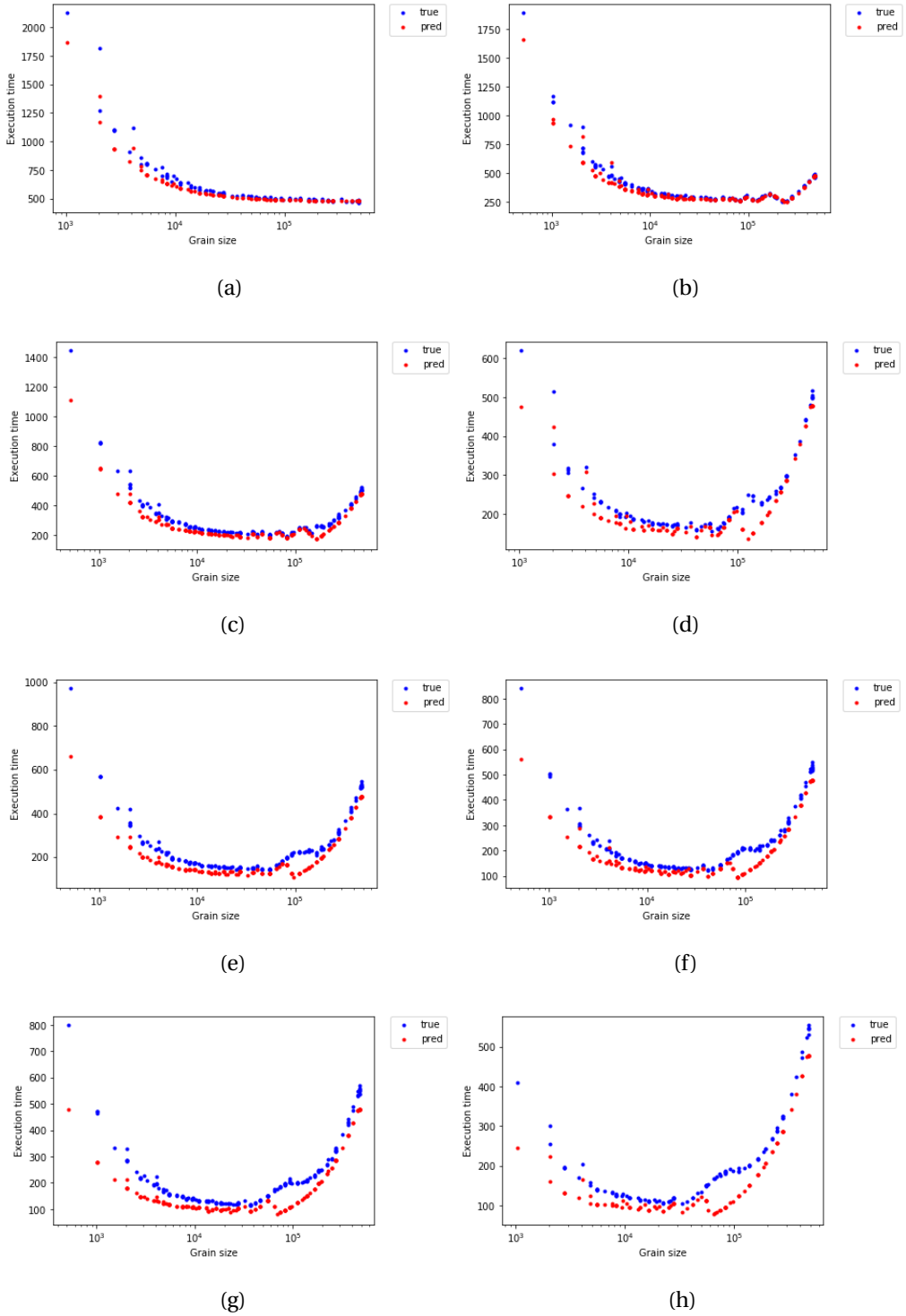


Figure 5.17. The results of predicting execution time using the suggested analytical model with parameters identified using the parallel for-loop benchmark compared to the original values for $matrix\_size = 690$, ran on (a) 1 core, (b) 2 cores, (c) 3 cores, (d) 4 cores, (e) 5 cores, (f) 6 cores, (g) 7 cores, and (h) 8 cores. The unit for execution time is microseconds.

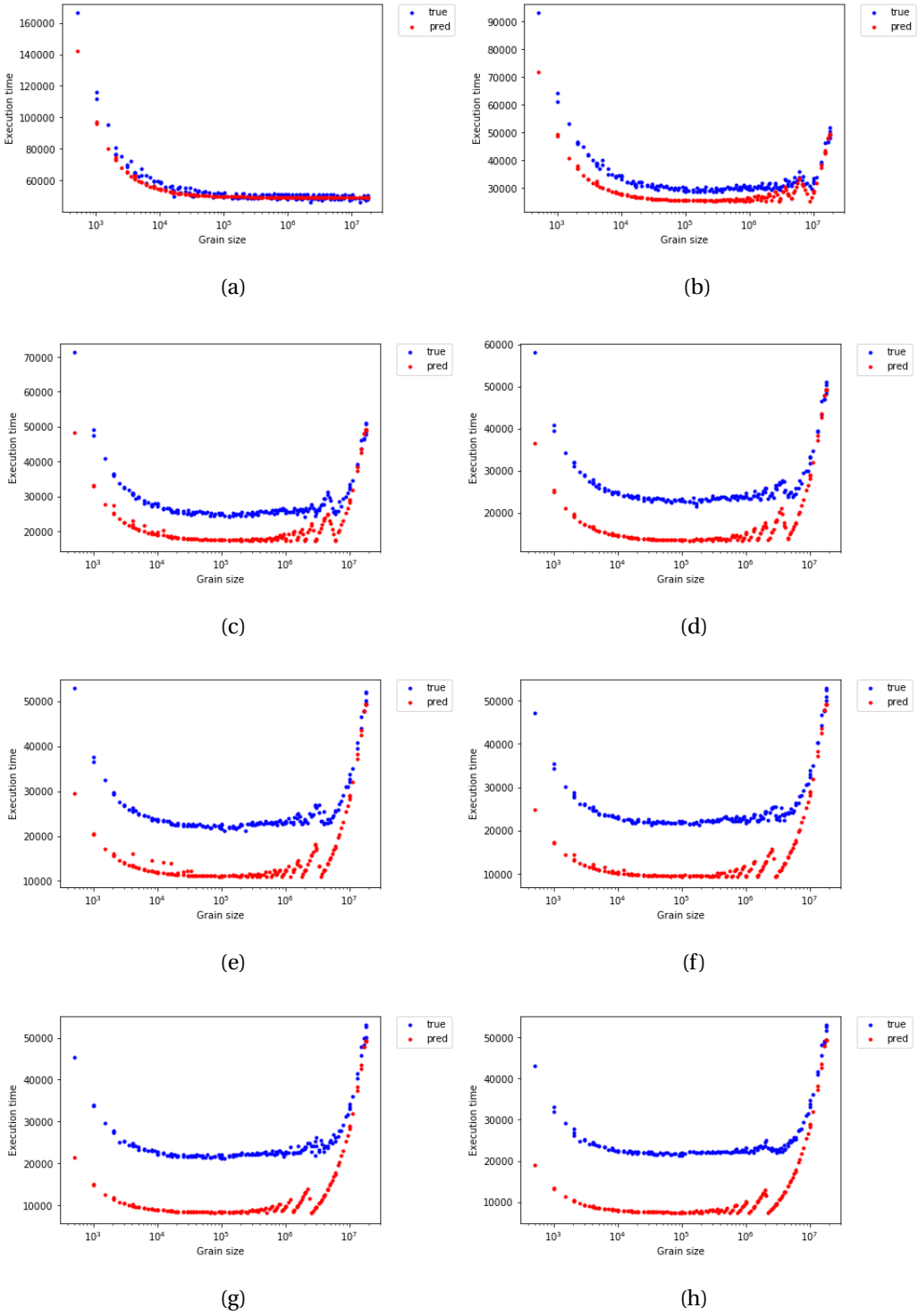


Figure 5.18. The results of predicting execution time using the suggested analytical model with parameters identified using the benchmark compared to the original values for $matrix\_size = 4222$, ran on (a) 1 core, (b) 2 cores, (c) 3 cores, (d) 4 cores, (e) 5 cores, (f) 6 cores, (g) 7 cores, and (h) 8 cores. The unit for execution time is microseconds.

The $DMATDMATADD$ benchmark was run on $Marvin$ node with $SandyBridge$ architecture, with $256KB$ of level 2 cache, and $20MB$ of level 3 cache. In order for the 3 same sized square matrices involved in $DMATDMATADD$ benchmark ($C = A + B$) to fit into level3 cache, they must be smaller than 935×935 in size.

For matrix sizes greater than 935, we observe an increase in execution time from our prediction using the proposed analytical model, as represented in Figure 5.18. We have to pay a penalty for loading data from memory instead of cache.

Figure 5.19 shows the relative error, while Figure 5.20 represents the R^2 score both calculated for each specific number of cores individually, and for matrix sizes smaller than 935.

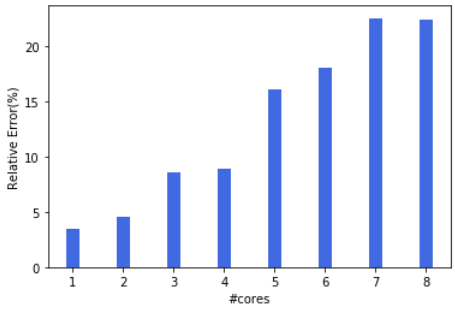


Figure 5.19. The relative error of using the predicting the execution time based on grain size for the $DMATDMATADD$ benchmark from $Blazemark$ suite. The relative error of all matrix sizes smaller than 953 was averaged for each specific number of cores.

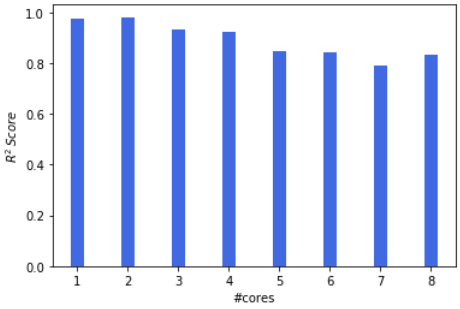


Figure 5.20. The R^2 score of using the predicting the execution time based on grain size for the $DMATDMATADD$ benchmark from $Blazemark$ suite. The relative error of all matrix sizes smaller than 953 was averaged for each specific number of cores.

Although we had completely ignored cache effects up until here for simplification, Figure 5.18 suggests that these effects do not influence the location of the flat region in the graph, which is the area with very small changes in execution time over which we observe the lowest

execution time.

Figure 5.21 shows the identified range for minimum execution time based on Formula 5.13, for different matrix sizes ran on 8 cores.

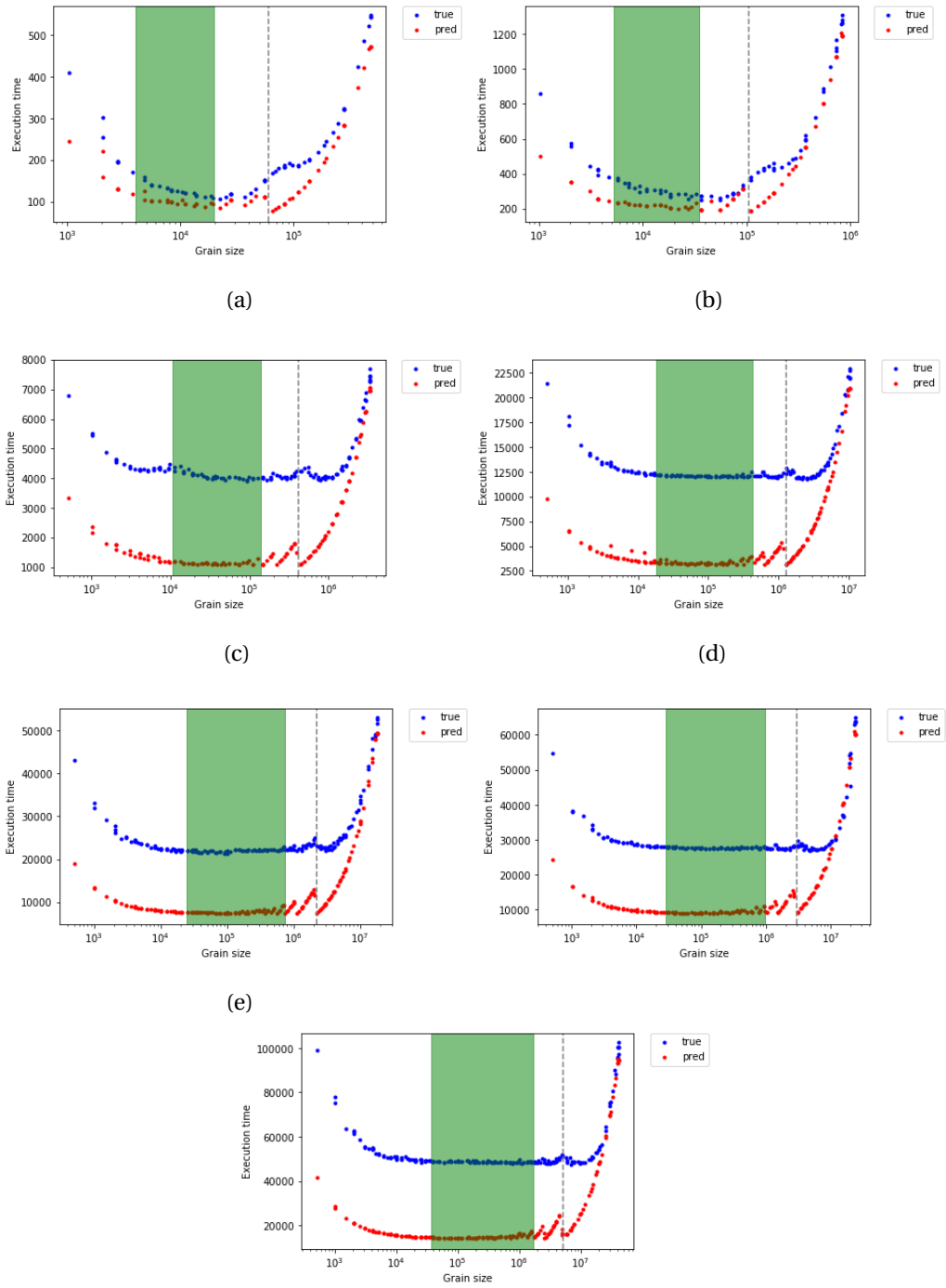


Figure 5.21. The results of predicted values of execution time through curve fitting vs the real data for $matrix\_size =$, for (a) 690, (b) 912, (c) 1825, (d) 3193, (e) 4222, (f) 4855, (g) 6420, on 8 cores, with $\lambda_b = 0.01$ and $\lambda_s = 0.5$. The unit for execution time is microseconds.

5.1.9. Predicting the Range of Chunk Size for Minimum Execution Time

Having identified the range of minimum execution time for each matrix size, we can find the range of chunk size that would result in that range of grain size, for a specific block size. As stated before, block size of 4×256 was selected as a safe choice for $DMATDMATADD$ benchmark.

Figure 5.22 shows the identified range of chunk size for minimum execution time based on Formula 5.13, for different matrix sizes ran on 8 cores, when $block\_size = 4 \times 256$. Each point on the graph is the real data collected from running the $DMATDMATADD$ benchmarks. The red area denotes the proposed range for chunk size, and the red points are the data points that lie in this region. As it can be seen the predicted range for chunk size are reasonable.

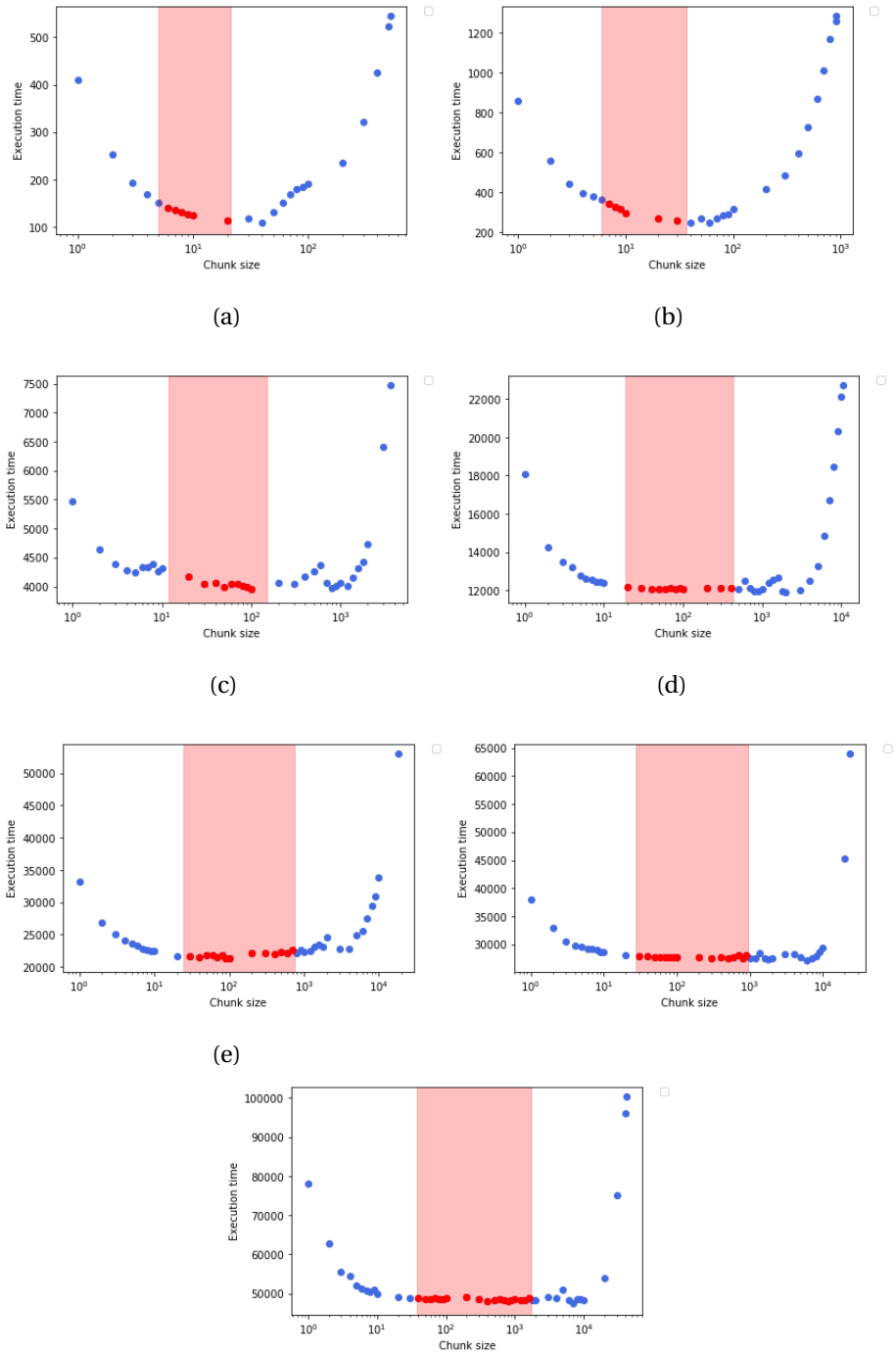


Figure 5.22. The suggested range of chunk size for minimum execution time with $block\_size = 4 \times 256$, for $matrix\_size =$, for (a) 690, (b) 912, (c) 1825, (d) 3193, (e) 4222, (f) 4855, (g) 6420, on 8 cores, when $\lambda_s = 0.5$ and $\lambda_b = 0.01$. The unit for execution time is microseconds.

5.2. Conclusion

In the previous sections we offered an analytical model to predict the behavior of execution time in terms grain size for a balanced parallel loop, and based on that model suggested a method to estimate the range of grain size to ensure staying in the flat region of the bathtub curve of execution time. The proposed model was then validated using a simple implemented benchmark. The parameters found through the benchmark was then utilized to predict the execution time of the $DMATDMATADD$ benchmark from $Blazemark$ suite. We were also able to find the range of grain size to achieve minimum execution time. At this point by selecting a reasonable value for block size, the range of chunk size to ensure staying in that range of grain size could be identified.

We propose to run the parallel for-loop benchmark in Listing 5.1 for a reasonable *problem_size = 100000*, with different values for chunk size on different number of cores, and use the collected data to estimate the α and σ parameters through curve fitting. This process could be added as an extra step to building process of Blaze. The complexity of the expression to be evaluated is then calculated at compile time as the number of floating point operations. A block size should also be selected based on the available cache size, the cache line, and the number of matrices involved in the operation. This part could also be done at compile time. At runtime, based on the matrix size and number of cores chosen to run the program on, the range of chunk size to achieve the lowest execution time for each specific expression is identified.

It has to be mentioned that although the proposed formula for execution time vs grain size depends on the sequential execution time, the proposed range for minimum execution time does not rely on sequential execution time, but depends on total amount of work, number of cores, overhead of creating and managing tasks, and the values we choose for the threshold on both sides(λ_b and λ_s).

Chapter 6. Splittable Task

6.1. splittable Task

In the previous chapter we proposed an analytical model to estimate the execution time of a balanced parallel for loop in terms of the grain size. Based on this model, we offered an approach to find the range of grain size to achieve minimum execution time. The parameters of the proposed model are identified through a benchmark and are exposed to the Blaze library to predict the range of grain size for minimum execution time of a problem at run-time. So the proposed method is a combination of compile-time and run-time solution to improve the performance.

In this chapter we choose another direction and look into a run-time adaptive solution to control task granularity in order to achieve the minimum execution time. Why? unbalanced work load.

Utilizing splittable tasks is a runtime adaptive method for managing task granularity, to avoid the large overhead of creating and managing too many tasks due to the fine grain parallelism on one hand, and the starvation resulted from creating less tasks than the available parallelism on the other hand.

Splittable tasks are tasks that could be partitioned into smaller tasks, when sufficient parallelism is available [60].

[61]

Prell intensively studies using the splittable tasks for runtime adaptivity in [60]. They start by offering steal-half strategy for work stealing in order to steal half of the tasks from a worker's queue at each steal attempt instead of just one task. This could help to avoid creating too much work stealing overhead. But depending on the application, steal-one or steal-half might be the preferable method for work stealing. They propose an adaptive method to decide on whether to use steal-one or steal-half strategy at runtime, based on the current selected strategy and the ratio of the number of executed tasks(M) within the last N steals, where N is considered the evaluation interval [60].

Next, basing on lazy task scheduling [55], they suggest to instead of creating at the tasks and decide on how to the workers should steal them, we can create one task and let it split if needed. This way you wouldn't have to deal with the overhead of scheduling tasks along with the overhead of stealing the tasks.
At each split, a portion of the task would be added to the current worker's deque as a splittble task to be stolen by free workers, while the rest of the task would be executed by the current worker. They propose two splitting strategies namely, and adaptive.

Our work here is based on Prell [60]'s definition of splittable tasks and their split strategies. We have implemented an executor within HPX which would create splittable tasks to execute the work, and we improved upon their work in following ways:

- At each split, instead of allowing the created splittable task to be stolen by free workers, we turn work stealing off, identify the idle workers and explicitly assign the work to one of them. This way we avoid the work stealing overhead originated from unsuccessful steal attempts.

- We suggest to utilize our proposed method for identifying the range of grain size for minimum execution time, and use the lower-bound of the range as a cutoff value to stop splitting in order to avoid creating too fine tasks.

- We used Apex, an Autonomic Performance Environment for Exascale [62], as a profiling tool to study how tasks are being scheduled.

6.1.1. Implementation

For a parallel for-loop with the range of $[a, b)$, one splittable task containing all the iterations from a to $(b-1)$ would be created. Depending on the splitting strategy when a certain condition is met this task would be split into two tasks, $task_1$ containing iterations in the range of $[a, c)$ and $task_2$ would contain iterations from $[c, b)$, where c is the split point. $task_1$ would be executed by the current worker while $task_2$, which is also a splittable task, would

be scheduled to run on another core. This allows the runtime to adaptively decide whether to split the current task into smaller tasks or just run it.

6.1.2. Splitting Strategies

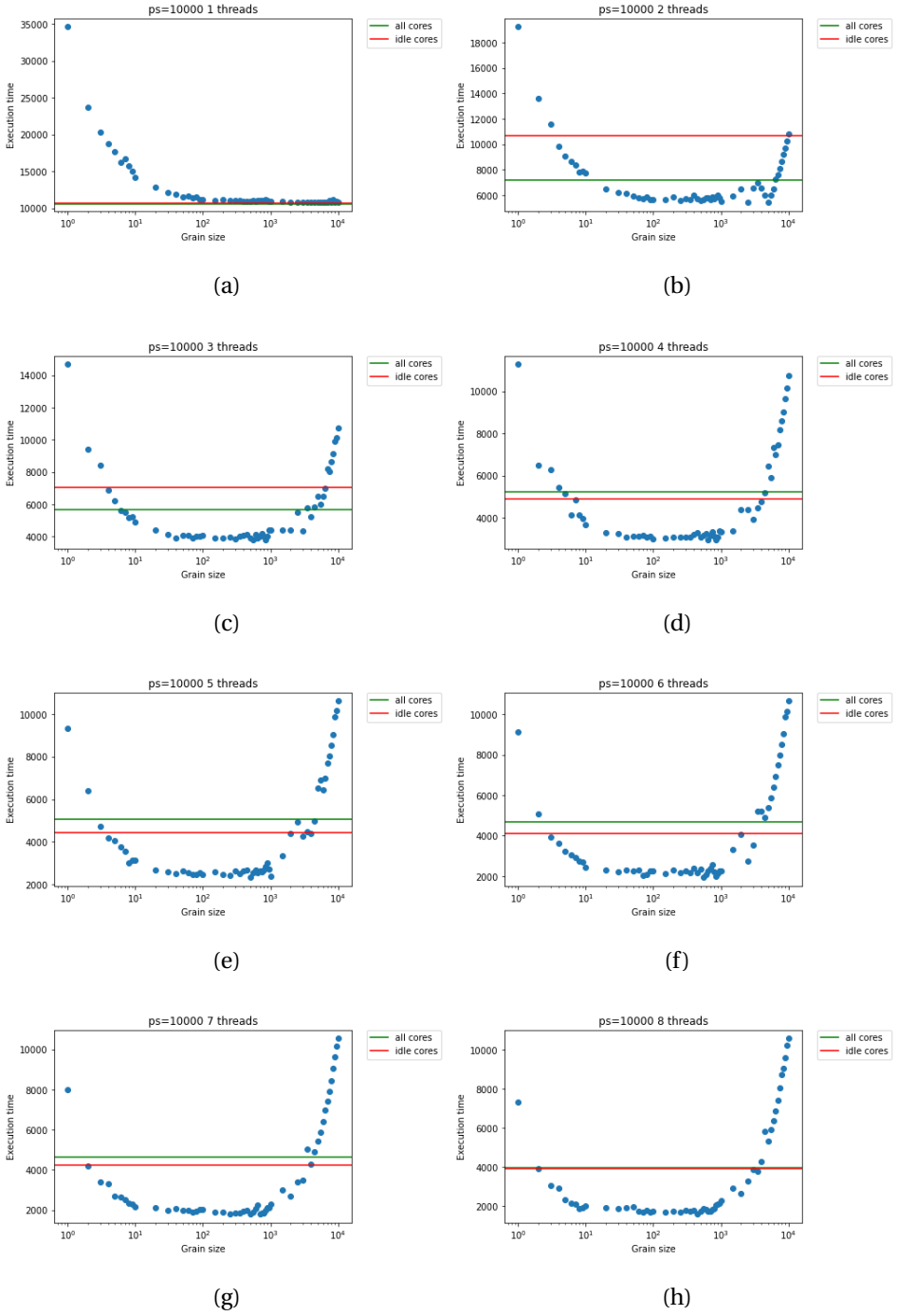
The splittable task could be split either based on total number of cores available, or number of idle cores at the time of split.

6.1.2.1. Splitting based on total number of cores

6.1.2.2. Splitting based on number of idle cores

6.1.3. Results

6.1.3.1. For-loop Benchmark



(a) (b)

(c) (d)

(e) (f)

(g) (h)

Figure 6.1. The results of running the hpx for_loop using splittable tasks with all-cores and idle-cores split types compared with different grain sizes, for $problem\_size = 10000$, for (a) 1 core, (b) 2 cores, (c) 3 cores, (d) 4 cores, (e) 5 cores, (f) 6 cores, (g) 7 cores, (h) 8 cores. The unit for execution time is microseconds.

6.1.3.2. Blazemark

6.1.3.3. Apex

Chapter 7. Setup

7.0.1. Blazemark

Blazemark is a benchmark suite provided by Blaze to compare the performance of Blaze with other linear algebra libraries including Blitz++[26], Boost uBLAS[59], GMM++[63], Armadillo[64], MTL4[28], and Eigen3[65], alongside plain BLAS libraries like Atlas[66], Goto[67], and Intel MKL.[68]

```
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 7.1. An example of the results obtained from running $DVECDVECADD$ benchmark through Blazemark

7.0.2. Configurations

Our experiments were run on Marvin nodes of Rostam cluster at Center for Computation and Technology(CCT) at Louisiana State University. Table 7.1 and Table 7.2 show some of the specifications of this node.

Category	Specification
CPU	2 x Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz
RAM	48 GB
Number of Cores	16
Hyperthreading	Off

Table 7.1. Specifications of the Marvin node from Rostam cluster at CCT.

Cache Level	Coherency Line Size	Number of Sets	Ways of Associativity	Size
1	64	512	8	32KB
2	64	512	8	256KB
3	64	512	20	20480KB

Table 7.2. Cache specifications of the Marvin node from Rostam cluster at CCT.

Library	Version
HPX	1.3.0
Blaze	3.5

Table 7.3. Specifications of the libraries used to run our experiments.

References

[1] 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.

[2] Patricia Grubel, Hartmut Kaiser, Kevin Huck, and Jeanine Cook. Using intrinsic performance counters to assess efficiency in task-based parallel applications. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1692–1701. IEEE, 2016.

[3] 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.

[4] J Bennett, R Clay, G Baker, M Gamell, D Hollman, S Knight, H Kolla, G Sjaardema, N Slattengren, K Teranishi, et al. Asynchronous many-task runtime system analysis and assessment for next generation platforms. *US Department of Energy, Sandia National Laboratories Report, Rep. no. SAND2015-8312*, 2015.

[5] R Clinton Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 38–38. IEEE, 1998.

[6] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[7] Neil J Gunther. The practical performance analyst. iuniverse. com inc. *Lincoln, Nebraska*, 2000.

[8] Neil J Gunther. A new interpretation of amdahl's law and geometric scalability. *arXiv preprint cs/0210017*, 2002.

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

[10] Neil J Gunther. A new interpretation of amdahl's law and geometric scaling. *arXiv preprint cs/0210017*, 2011.

[11] Abhishek Kulkarni and Andrew Lumsdaine. A comparative study of asynchronous many-tasking runtimes: Cilk, charm++, parallex and am++. *arXiv preprint arXiv:1904.00518*, 2019.

[12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[13] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[14] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[15] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[16] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

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

[18] MPI Mpi. A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[19] Laxmikant V Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *OOPSLA*, volume 93, pages 91–108. Citeseer, 1993.

[20] 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.

[21] J Davison de St Germain, John McCorquodale, Steven G Parker, and Christopher R Johnson. Uintah: A massively parallel problem solving environment. In *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, pages 33–41. IEEE, 2000.

[22] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[23] Guang R Gao, Thomas Sterling, Rick Stevens, Mark Hereld, and Weirong Zhu. Parallex: A study of a new parallel computation model. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–6. IEEE, 2007.

[24] 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.

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

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

[27] Jörg Walter and Mathias Koch. The boost ublas library, 2002.

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

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

[30] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. *Introduction to parallel computing*. Pearson Education, 2003.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

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

[39] Ali Mohammed, Ahmed Eleliemy, Florina M Ciorba, Franziska Kasielke, and Ioana Banicescu. Experimental verification and analysis of dynamic loop scheduling in scientific applications. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 141–148. IEEE, 2018.

[40] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] Peter Thoman, Herbert Jordan, and Thomas Fahringer. Adaptive granularity control in task parallel programs using multiversioning. In *European Conference on Parallel Processing*, pages 164–177. Springer, 2013.

[55] Alexandros Tzannes, George C Caragea, Uzi Vishkin, and Rajeev Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(3):1–51, 2014.

[56] José E Moreira, Dale Schouten, and Constantine Polychronopoulos. The performance impact of granularity control and functional parallelism. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 581–597. Springer, 1995.

[57] Xavier Aguilar, Herbert Jordan, Thomas Heller, Alexander Hirsch, Thomas Fahringer, and Erwin Laure. An on-line performance introspection framework for task-based runtime systems. In *International Conference on Computational Science*, pages 238–252. Springer, 2019.

[58] Bibek Wagle, Mohammad Alaul Haque Monil, Kevin Huck, Allen D Malony, Adrian Serio, and Hartmut Kaiser. Runtime adaptive task inlining on asynchronous multitasking runtime systems. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

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

[60] Andreas Prell. *Embracing explicit communication in work-stealing runtime systems*. PhD thesis, 2016.

[61] Arch Robison, Michael Voss, and Alexey Kukanov. Optimization via reflection on work stealing in tbb. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[62] Kevin A Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen D Malony, Thomas Sterling, and Rob Fowler. An autonomic performance environment for exascale. *Supercomputing frontiers and innovations*, 2(3):49–66, 2015.

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

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

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

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

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

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