10-22-2020

Optimizing the Performance of Multi-threaded Linear Algebra Libraries Based on Task Granularity

Shahrzad Shirzad

Follow this and additional works at: https://repository.lsu.edu/gradschool_dissertations

Part of the Other Computer Engineering Commons

Recommended Citation

https://repository.lsu.edu/gradschool_dissertations/5376

This Dissertation is brought to you for free and open access by the Graduate School at LSU Scholarly Repository. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Scholarly Repository. For more information, please contact gradetd@lsu.edu.
OPTIMIZING THE PERFORMANCE OF MULTI-THREADED LINEAR ALGEBRA LIBRARIES BASED ON TASK GRANULARITY

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

The Division of Electrical and Computer Engineering

by

Shahrzad Shirzad
B.Sc., Sharif University of Technology, 2006
M.Sc., K. N. Toosi University of Technology, 2009
December 2020
Acknowledgments

First of all I would like to thank my advisors Dr. Kaiser and Dr. Ramanujam for their help and support. I am and will always be beyond grateful for Dr. Kaiser’s care, support and guidance throughout these years.

I would like to express my gratitude and appreciation for Dr. Soysal for supporting me and helping me to grow.

I also want to thank my husband Alireza for his love and all the sacrifices he made to make this possible for me, and my little one, Damon, whose smile brings joy to my life.

Special thanks to my sister Sharareh, I was so lucky to have her here with me, it certainly made life easier and more pleasant for me.

I have to thank my parents, my sister Shohreh and her lovely family including my dearests Kian and Karen for their love and support, I couldn’t go on without it.

Special thanks to Maziar, Mobina, Kasra, Raysan with their little one Zhina, Melika, Elika and Zahra for their invaluable friendship throughout these years.

Finally I want to express my deep gratitude to my friends at CCT especially Adrian, Rod, and Bibek for spending hours listening to me, guiding and motivating me, and Katie for all the time she spent proofreading my dissertation.
**Table of Contents**

Acknowledgements .............................................................. ii

Abstract ................................................................................ v

Chapter

1. Introduction ................................................................. 1
   1.1. Dissertation Statement .............................................. 1
   1.2. Research Contributions ............................................ 2
   1.3. Dissertation Outline ............................................... 3

2. Background ................................................................. 5
   2.1. Asynchronous Many-task Runtime Systems ..................... 5
   2.2. Task Granularity .................................................... 6
   2.3. Analytical Modeling of the Execution Time of the Parallel Programs .... 7
   2.4. Loop Scheduling .................................................... 10
   2.5. Blaze ............................................................... 12

3. Data Collection and Analysis .............................................. 18
   3.1. Configurations ..................................................... 18
   3.2. Experiments ....................................................... 19
   3.3. Observation ....................................................... 22

4. Polynomial Modeling of the Throughput in Terms of Grain Size ........... 26
   4.1. Generalizing the Fitted Function to Include Number of Cores ........... 27
   4.2. Finding the Range of Grain Sizes for Maximum Performance ........... 30
   4.3. Estimating the Optimum Range of Chunk Sizes .................... 31
   4.4. Conclusions ....................................................... 33

5. An Analytical Model for Predicting the Optimum Range of Grain Size for Minimum Execution Time ............................................. 35
   5.1. Evaluating the Proposed Model .................................... 42
   5.2. Testing the Proposed Model on Other Architectures ................. 49
   5.3. Identifying the Optimal Range of Grain Sizes .................... 56
   5.4. Identifying the Range of Grain Size for Minimum Execution Time .... 64
   5.5. Locating the Optimum Range of Grain Sizes for the for-loop Benchmark .... 65
   5.6. Applying the Proposed Model to the Blazemark Data ............... 69
   5.7. Predicting the Range of Chunk Size for Minimum Execution Time .... 80
   5.8. Conclusions ....................................................... 88

6. Runtime Task Granularity Control through Splittable Tasks ................ 92
   6.1. Splittable Tasks ................................................... 92
   6.2. Implementation .................................................. 94
   6.3. Using the Proposed Analytical Model to Estimate the Stop Splitting Threshold .................................................. 100
Abstract

Linear algebra libraries play a very important role in many HPC applications. As larger datasets are created everyday, it also becomes crucial for the multi-threaded linear algebra libraries to utilize the compute resources properly.

Moving toward exascale computing, the current programming models would not be able to fully take advantage of the advances in memory hierarchies, computer architectures, and networks. Asynchronous Many-Task(AMT) Runtime systems would be the solution to help the developers to manage the available parallelism.

In this Dissertation we propose an adaptive solution to improve 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 get as close as possible to the highest performance. Our focus is on machine learning applications, where we are potentially dealing with very large matrices, which could make creating temporaries very expensive.

For this purpose we selected 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.

HPX, as an AMT runtime system, offers scalability and fine-grained parallelism through creating lightweight threads for fast context switching between the threads. Finding the optimum task granularity is a challenge in AMTs. Creating too many small tasks would result in performance degradation due to scheduling overhead of the tasks, and creating too few tasks would lead to under-utilization of the resources. Our work focuses on finding the optimum task granularity for each specific problem.

We tried two different approaches to model the relationship between the performance and the grain size, in order to find a range of grain sizes that could lead us to the maximum performance.

First, we used polynomial functions to model how throughput changes in terms of the
grain size, and number of cores. Although this method was successful in finding the range of grain sizes for maximum throughput, it was not physical. This motivated us to go deeper and try to develop an analytical model for execution time in terms of grain size for balanced parallel for loops. Based on the analytical model we propose a method to predict the range of grain size for minimum execution time.

Moreover, since the parameters of the proposed model only depend on the system architecture, we suggest to use a parallel for-loop benchmark to find these parameters on a system, and use it to find the range of grain size for minimum execution time for arbitrary balanced parallel for-loop applications ran on the same machine.

Having the mentioned models, we changed the current implementation of the HPX back-end for Blaze by adding two parameters to represent the unit of work, and the number of 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 to estimate the number of floating point operations occurring in each unit of work.

The model parameters estimated through the parallel for-loop benchmark could also be plugged into Blaze at compile time, in order to find the optimum range of grain size at runtime based on the matrix sizes and complexity of the operations.

In the next step, we used the identified range of grain size to extend the previous implementation of splittable tasks, as an algorithm to control task granularity. We modified the current implementation by scheduling the tasks on idle cores directly instead of waiting for them to be stolen, and integrating the lower-bound of the analytical model as the threshold to stop splitting, in order to adapt the threshold to the system architecture and the application being executed.
Chapter 1. Introduction

The path toward exascale computing would include more complex machine architectures, deeper memory hierarchies, heterogeneous nodes, and complicated networks[1]. 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[2].

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[3].

On the other hand, the core element of many high performance computing applications is the linear algebra library. The performance of these applications significantly 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 requires a deep understanding of memory hierarchy and registers from the programmer[4]. Linear algebra libraries like ATLAS[4], SPIRAL[5] use hardware-specific optimizations to improve their performance.

In this work, we study 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 in order to optimize the execution time. A number of these parameters can be extracted at compile time, while extraction of the others must be postponed to runtime.

1.1. Dissertation Statement

The main objective of this dissertation 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 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 many-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. 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 not to use our resources properly. So in every 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. Research Contributions

The contributions of this dissertation could be summarized into:

- Developing an analytical model to predict the total execution time of a balanced parallel for-loop. To our knowledge, this is the first analytical model in terms of both grain size and number of cores.

- A method has been offered to predict the range of grain sizes to achieve minimum execution time for a particular number of cores.

- Building a benchmark on top of HPX to evaluate the model. We utilized this benchmark to estimate the model parameters on each machine architecture. The obtained parameters could then be used to estimate the range of grain sizes for minimum exe-
cution time, and consequently improve the performance of any other balanced parallel for-loop application executed on the same system.

- Modifying the implementation of HPX backend for Blaze in order to be able to impose fined grain parallelism and provide a tool to control task granularity.

- Modifying the current implementation of splittable tasks so that 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, in order to avoid the work stealing overhead originated from unsuccessful steal attempts.

- We utilized our proposed method to identify the optimum range of grain sizes, and use the lower-bound of the range as a cutoff value to stop splitting in order to avoid creating too fine tasks. This threshold is specific to the machine architecture and the application being executed.

- Our work on runtime adaptivity through splittable tasks is integrated as splittable executor into HPX, an open source C++ Standard Library for Concurrency and Parallelism.

1.3. Dissertation Outline

In Chapter 2, we will explain briefly the background needed for this dissertation, including Blaze and HPX libraries, effects of task granularity, and the Universal Scaling Law (USL) method for modeling the throughput based on number of cores. In an effort to understand the association between execution time and grain size we ran a set of experiments. The experiment environment and the obtained results are explained in Chapter 3.

In Chapter 4 and Chapter 5 we investigate the possibility of using polynomial regression and an analytical model respectively to express how execution time changes in terms of grain size and number of cores for the purpose of identifying the optimum range of grain size. Chapter 5 shows how the identified range of grain size could help a previously implemented algorithm. Finally Chapter 7 discusses other works that have been done in our area of focus, and
the conclusions of this dissertation is presented in Chapter 8.
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[6, 3]. Some of the common existing execution models include fork-join, Communicating Sequential Processes(CSP), event-based models and actor model[3]. Some of the current parallel programming frameworks include, accelerator-based programming models like OpenACC[7], OpenCL[8], and CUDA[9], shared-memory programming models like Intel TBB[10], Cilk[11], OpenMP[12], and distributed-memory programming models like MPI[13], Charm++[14], ParalleX[15], UPC[6].

A runtime system is in charge of creating and managing the concurrency, through implementing parts of an execution model[3]. 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[3].

Some of well known AMT runtimes include: HPX[2], Charm++[14], Uintah[16], Legion[17].

2.1.1. HPX

HPX[2] is a C++ runtime system for parallel and distributed applications based on ParalleX execution model[15]. HPX creates lightweight user-level threads with fast context switching[6]. 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[6].

HPX provides work sharing and work stealing for automatic balancing of the workload among the resources [18].
2.1.1.1. Execution Model

The Parallex [15] execution model identifies four potential sources for performance degradation as "SLOW": Starvation, Latency, Overheads, and Waiting or contention[15].

Starvation refers to a situation where there is an insufficient amount of work for the computing resource. This could be due to insufficient total amount of work available, or unbalanced distribution of work among resources[6].

Latency is the time distance, usually measured in processor clock, of accessing remote data or services[19].

Overhead refers to the effort that needs to be taken to manage parallel resources and actions on the critical path[6].

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[19]. This could happen due to limited network bandwidth, shared communication channels, memory bank conflicts[6],[19].

2.2. Task Granularity

Defining the grain size as the amount of work assigned to one HPX thread, Grubel[20] 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[20]. When grain size is small, to perform same amount of work, a 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 [20]

2.3. Analytical Modeling of the Execution Time of the 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[21].

2.3.1. Universal Scalibility Law

Amdahl’s law[22], 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 $\sigma$ is the serial fraction of the execution time, based on Amdahl’s law[23].

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

However, Gunther[23] extends Amdahl’s law by incorporating the effect of three factors, namely concurrency, contention, and coherency, as shown in Equation 2.2.

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

Concurrency($p$) represents the linear speedup that could have been achieved if no interaction existed among the processors, contention($\sigma$) represents the serialization effect of shared writable data, and finally coherency or data consistency($\kappa$) represents the effort that needs to be made for keeping shared writable data consistent[23].

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($\gamma$) to represent the serial throughput.

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

Universal scalability law also suggests that for some values of $\sigma$ and $\kappa$ there could be a certain number of processors that yield to maximum performance[23]. 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 $\kappa = 0.005$.

2.3.2. Other Models

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

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

The parameter $\phi$, where $0 < \phi \leq 1$, is called the MP factor, and it represents the remaining section of the processor capacity after deducting overheads.

The geometric model is nonphysical for large number of processors, due to inconsistency with Coxian queuing model[24].

Quadratic model[25], with overhead parameter $\gamma$, where $0 \leq \gamma < 1$, is represented in Equation 2.5.

$$S(p) = p - \gamma p(p - 1)$$ (2.5)
The quadratic model has a critical point at:

$$p^* = \left\lfloor \frac{1 + \gamma}{2\gamma} \right\rfloor$$ (2.6)

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

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

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

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

2.4. 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 [26].

In static loop scheduling, each processor is assigned an equal number of consecutive iterations. The iterations included in one task, are either could also be selected in a round-robin way, which would not be helpful from cache coherency point of view [27]. This scheduling method includes almost no runtime overhead due to scheduling, due to assignments occurring at compile time. Although iterations are evenly assigned to the processors, several factors can still contribute to difference in execution time of the iterations, resulting in load imbalance between the processors. These factors include, cache misses, page faults, and interprocessor communication [28].
On the other hand, dynamic scheduling approaches leave the assignments to the runtime. Although this concludes higher scheduling overhead, these approaches could help with load balancing. These methods include Pure Self-scheduling, Chunk Self-scheduling, Guided Self-scheduling[29], Trapezoid Self-scheduling[30],[27], and Factoring[31].

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[27].

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[27].

As an adaptive loop scheduling technique, Guided Self-scheduling[29] 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[32].

Very similar to Guided Self-scheduling, Factoring[31] 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, Fac-
toring performs better than Guided scheduling[33].

Along with the mentioned loop scheduling approaches, two other scheduling techniques can be utilized for load balancing. Work stealing[34] 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[34].

Each of the mentioned methods performs well for a specific problem. In this dissertation we are interested in finding a general solution which can automatically decide on the grain size or chunk size parameter for each application in order to achieve the best performance.

2.5. Blaze

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

Although able to achieve promising performances for element-wise operations, these methods are not suitable for high performance computing for the following reasons. Due to their abstraction from both the data type and also the operation itself, they do not allow optimizations specific to the type of the arrays, alongside the operation[35]. As a solution, Blaze proposes smart ETs with these 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[35].

Some of the ET-based linear algebra libraries are: Blitz++[37], Boost uBLAS[38], MTL[39], and Eigen[40]. Among these libraries, Eigen, MTL, alongside Blaze, impose different conceptual changes to ETs in order to make them suitable for HPC. Blaze also makes it possible
for programmers to utilize SIMD (Single Instruction Multiple Data) vectorization simply by adding a compile time flag.

2.5.1. Parallelization in Blaze

Depending on the operation and the size of operands, this assignment could be parallelized through four different backends, namely, HPX, OpenMP[12], C++ threads, and Boost[41].

Table 2.1 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.

Table 2.1. List of some of the thresholds applied to the operations performed by Blaze, starting from which the operation is executed in parallel

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Array size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DVecDVecADD, DVecDVecMULT$</td>
<td>38000</td>
</tr>
<tr>
<td>$DMatDMatADD$</td>
<td>36100 elements equivalent to a $175 \times 175$ matrix</td>
</tr>
<tr>
<td>$DMatDMatMULT$</td>
<td>3025 elements equivalent to a $55 \times 55$ matrix</td>
</tr>
</tbody>
</table>

2.5.2. 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.

Listing 2.1 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 \( \text{block\_size} \) and \( \text{chunk\_size} \). Instead of evaluating the expression on the whole matrix, we divide the matrices into blocks of size \( \text{block\_size} \) and perform the operations on each of these blocks at each iteration. Since these blocks are independent from each other, a number of these blocks could be assigned to each core in order to exploit parallelism. \( \text{chunk\_size} \) denotes the number of iteration, or the number of blocks of size \( \text{block\_size} \) that are included in one task. Listing 2.2 shows the new implementation of the HPX backend for Blaze based on these new definition.

### 2.5.3. Blazemark

Blazemark is a benchmark suite provided by Blaze to compare the performance of Blaze with other linear algebra libraries including Blitz++[37], Boost uBLAS[41], GMM++[42], Armadillo[43], MTL4[39], and Eigen3[44], alongside plain BLAS libraries like Atlas[45], Goto[46], and Intel MKL[47]
Listing 2.1: Previous implementation of Assign function for HPX backend in Blaze.

```cpp
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 hp::::parallel::for_loop;
    using hp::::parallel::execution::par;
    BLAZE_FUNCTION_TRACE;

    constexpr bool simdEnabled( MT1::simdEnabled && MT2::simdEnabled && IsSIMDCombinable_v<MT1,MT2> );
    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 rowsPerThread ( ( simdEnabled && addon1 ) ? (~ rhs ).rows() / threadmap.first + addon1 ) ;
    const size_t rest1 ( rowsPerThread & ( 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 colsPerThread ( ( simdEnabled && addon2 ) ? (~ rhs ).columns() / threadmap.second + addon2 ) ;
    const size_t rest2 ( colsPerThread & ( 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 ) ) ;
            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 ) ) ;
            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 ) ) ;
            auto source( submatrix<aligned>(~rhs , row , column , m , n ) ) ;
            op( target , source ) ;
        }
        else
        {
            auto target( submatrix<unaligned>(~lhs , row , column , m , n ) ) ;
            auto source( submatrix<unaligned>(~rhs , row , column , m , n ) ) ;
            op( target , source ) ;
        }
    }
}
```
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<size_t>(BLAZE_HPX_MATRIX_BLOCK_SIZE_ROW ), (~rhs).rows() ) );
  const size_t numCols ( min( static_cast<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 && rhsAligned ) {
      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 && lhsAligned ) {
      auto target( submatrix<aligned>(~lhs, row, column, m, n ) );
      const auto source( submatrix<unaligned>(~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 );
    }
  }
}
Figure 2.3. An example of the results obtained from running `DVecDVecADD` benchmark through Blazemark
Chapter 3. Data Collection and Analysis

In order to understand 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.

3.1. Configurations

Our experiments were run on Marvin and Medusa nodes of Rostam cluster at Center for Computation and Technology (CCT) at Louisiana State University. Table 3.1-3.3 show some of the specifications of these nodes.

Table 3.1. Specifications of the Marvin and Medusa node from Rostam cluster at CCT.

<table>
<thead>
<tr>
<th>Node</th>
<th>CPU details</th>
<th>RAM</th>
<th>Number of Cores</th>
<th>Hyperthreading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marvin</td>
<td>2 x Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz</td>
<td>48 GB</td>
<td>16</td>
<td>Off</td>
</tr>
<tr>
<td>Medusa</td>
<td>2 x Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz</td>
<td>96 GB</td>
<td>40</td>
<td>Off</td>
</tr>
</tbody>
</table>

Table 3.2. Cache specifications of the Marvin node from Rostam cluster at CCT.

<table>
<thead>
<tr>
<th>Cache Level</th>
<th>Coherency Line Size</th>
<th>Number of Sets</th>
<th>Ways of Associativity</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>64</td>
<td>512</td>
<td>8</td>
<td>32KB</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>512</td>
<td>8</td>
<td>256KB</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>512</td>
<td>20</td>
<td>20480KB</td>
</tr>
</tbody>
</table>

Table 3.3. Cache specifications of the Medusa node from Rostam cluster at CCT.

<table>
<thead>
<tr>
<th>Cache Level</th>
<th>Coherency Line Size</th>
<th>Number of Sets</th>
<th>Ways of Associativity</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>64</td>
<td>512</td>
<td>8</td>
<td>32KB</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>1024</td>
<td>11</td>
<td>1024KB</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>40960</td>
<td>16</td>
<td>28160KB</td>
</tr>
</tbody>
</table>
Table 3.4. Specifications of the libraries used to run our experiments.

<table>
<thead>
<tr>
<th>Library</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPX</td>
<td>1.5.0</td>
</tr>
<tr>
<td>Blaze</td>
<td>3.5</td>
</tr>
<tr>
<td>clang</td>
<td>10.0.0</td>
</tr>
<tr>
<td>boost</td>
<td>1.73.0</td>
</tr>
</tbody>
</table>

3.2. Experiments

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. Throughout this dissertation matrix size of $m$ represents a square matrix of size $m \times m$.

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) Blaze uses the configured backend to parallelize the assignment operation. For this reason, we start our experiments with matrix size of 200 and gradually increase the size to 1587.

Table 3.5 show the matrix sizes and the number of cores chosen for our experiments with *DMatDMatADD* benchmark.

Table 3.5. List of different values used for each variable for running the *DMatDMatADD* benchmark

<table>
<thead>
<tr>
<th>Category</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix sizes</td>
<td>200, 230, 264, 300, 396, 455, 523, 600, 690, 793, 912, 1048, 1200, 1380, 1587</td>
</tr>
<tr>
<td>Number of cores</td>
<td>1, 2, 3, 4, 5, 6, 7, 8</td>
</tr>
<tr>
<td>Number of rows in the block</td>
<td>4, 8, 12, 16, 20, 32</td>
</tr>
<tr>
<td>Number of columns in the block</td>
<td>64, 128, 256, 512, 1024</td>
</tr>
<tr>
<td>Chunk size</td>
<td>Between 1 and total number of blocks (logarithmic increase)</td>
</tr>
</tbody>
</table>
Figure 3.2 shows the results of running \textit{DMatDMatADD} benchmark for matrix sizes and number of cores listed in Table 3.5 based on grain size.

On the other hand, Figure 3.3 integrates the results obtained from running the same benchmark with different matrix sizes. Each color in this graph represents a specific matrix size.

Figure 3.1. The results obtained from running \textit{DMatDMatADD} benchmark through Blaze-mark for matrix size 690 on different number of cores.
Figure 3.2. The results obtained from running *DMatDMat ADD* benchmark through Blaze-mark for matrix of size 690 from two different angles
Figure 3.3. The results obtained from running \textit{DMatDMat ADD} benchmark through Blaze-mark for matrix sizes from 200×200 to 1587×1587

3.3. 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 3.5.

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.
The collected data, as seen in Figure 3.4, 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 $4 \times 64$ 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 3.5 shows how the throughput changes with regards to the grain size for the $DMatDMatADD$ benchmark through Blaze-mark for matrix size 690 with different combinations of block size and chunk size on 4 cores.
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 3.1 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 3.5. The results obtained from running \texttt{DMatDMatADD} benchmark through BlazeMark for matrix size 690 on 4 cores.
Figure 3.6. Throughput vs. grain size graph obtained from running $DMatDMatADD$ benchmark on 4 cores.
Chapter 4. Polynomial Modeling of the Throughput in Terms of Grain Size

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.

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 3.6 to a second degree polynomial.

Figure 4.3 shows the results of using a quadratic function to fit the data for one matrix size with different number of threads. We used the 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 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 4.1 for training and test set. In this equation, $t_i$ and $p_i$ denote the true value and the predicted value of the $i$th sample respectively, where $n$ is the number of samples with the particular number of cores.

$$Relative\_error = \frac{1}{n} \sum_{i=1}^{n} |1 - \frac{p_i}{t_i}|$$  \hspace{1cm} (4.1)
4.1. 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 $P = a g^2 + b g + c$, 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 regard 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.
Figure 4.3. The results of fitting the throughput vs grain size data into a 2d polynomial for $DMatDMatADD$ benchmark for matrix size 690 with different number of cores on the test data set.
Using this 3rd degree polynomial to fit the parameters, we can generalize the relationship between throughput and grain size in the following equation:

\[ P = a_{11} g^2 N^3 + a_{10} g^2 N^2 + ... + a_1 N + a_0 \]  

(4.2)

where \( P \) is the throughput, \( g \) is the grain size, and \( N \) is the number of cores and coefficients \( a_{11}, \ldots, a_0 \) 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 *DMatDMat ADD* benchmark, the fitted polynomial based on equation 4.2, is represented in Figure 4.5, for 2, 4, and 8 cores for a matrix of size 690.
Figure 4.6. The training and test error obtained from 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, for each number of cores.

(a) all data points included  
(b) leftmost sample excluded

Figure 4.6a 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 4.6b.

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.

4.2. Finding the Range of Grain Sizes for Maximum 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. In 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.

![Figure 4.7](image1)

(a) 2 cores  
(b) 4 cores  
(c) 8 cores

Figure 4.7. The range of grain size (shown as the red line) that leads to a performance within 10% of the maximum performance.

![Figure 4.8](image2)

(a) matrix size 690  
(b) matrix size 523 to 912

Figure 4.8. The range of grain size within 10% of the maximum performance of the fitted polynomial function for $DMat$ benchmark for different number of cores.

Figure 4.8a shows the calculated range for matrix size 690 for each specific number of threads, while Figure 4.8b compares the range for different matrix sizes.

4.3. **Estimating the Optimum Range of Chunk Sizes**

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 3.5 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 $4 \times 256$. 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 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 $4 \times 256$, 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 4.9, for matrix size 690 and block size $4 \times 256$. 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) block size $4 \times 256$, 2 cores  
(b) block size $4 \times 256$, 4 cores  
(c) block size $4 \times 256$, 8 cores  
(d) block size $4 \times 512$, 2 cores  
(e) block size $4 \times 512$, 4 cores  
(f) block size $4 \times 512$, 8 cores

Figure 4.9. 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 with block size of $4 \times 256$ and $4 \times 512$ on 2, 4, and 8 cores. Silver points denotes the detected range of chunk size, and the red star shows the median point.
4.4. Conclusions

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 4.10) , while the minimum possible grain size is the same.

![Figure 4.10. Throughput vs grain size graph obtained from running DMatDMat ADD benchmark on 4 cores.](image)

Moreover, Figure 4.11 shows how the predicted grain size range changes for different ma-
trix sizes for each number of cores.

Figure 4.11. The range of grain size within 10% of the maximum performance of the fitted polynomial function for \textit{DMatDMat ADD} benchmark for different number of cores for matrix size 523 to 912.

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

In the next chapter 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. An Analytical Model for Predicting the Optimum Range of Grain Size for Minimum Execution Time

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 only at the data and trying to find a function to fit, we studied 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 create 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.

Table 5.1 shows the notations used throughout this section. It should be noted that the amount of work is measured in terms of execution time or floating point operations depending on the application. Chunk size is defined as the number of iterations included in one task, while grain size is the amount of work contained in a task, to be executed by one user-level thread.

In an attempt to find this analytical model, we started by looking into two factors that
Table 5.1. Table of Notations

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Number of cores</td>
</tr>
<tr>
<td>$n_t$</td>
<td>Total number of tasks created</td>
</tr>
<tr>
<td>$p_s$</td>
<td>Total amount of work</td>
</tr>
<tr>
<td>$w_{\text{max}}$</td>
<td>Maximum amount of work assigned to a single core</td>
</tr>
<tr>
<td>$t_{\text{max}}$</td>
<td>Execution time of $w_{\text{max}}$</td>
</tr>
<tr>
<td>$M$</td>
<td>Number of utilized cores</td>
</tr>
<tr>
<td>$t_{\text{seq}}$</td>
<td>Sequential execution time</td>
</tr>
<tr>
<td>$t_i$</td>
<td>Execute time of one iteration</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of loop iterations</td>
</tr>
<tr>
<td>$c_s$</td>
<td>Chunk size</td>
</tr>
<tr>
<td>$g$</td>
<td>Grain size</td>
</tr>
</tbody>
</table>

we believe play the most important role in determining the execution time: the overhead of scheduling the tasks, and the maximum amount of work assigned to one core. In order to study the effects of these factors on execution time of a balanced parallel for-loop, we define iteration length, $t_i$, as the time it takes to execute one iteration, $n$ as total number of loop iterations, $c_s$ as the chunk size which is the number of iterations to be executed by one user-level thread. The grain size, $g$, then refers to the amount of work assigned to one task. Depending on the parallel program being executed, this parameter has to be quantified somehow.

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 execute the tasks it has been assigned to. Here we denote the core with maximum expected amount of work as $\text{core}_0$, and the amount of work that has been assigned to it as $w_{\text{max}}$.

With this assumption, the main factors contributing to the execution time would be the overhead of scheduling tasks on $\text{core}_0$, the time it takes to run $w_{\text{max}}$ amount of work on $\text{core}_0$, denoted with $t_{\text{max}}$, 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.
Equation 5.1 shows the expected formula in its simplest form.

\[
\text{execution\_time} = t_{\text{overhead}} + t_{\text{max}}
\]  \hspace{1cm} (5.1)

In equation 5.1, \(t_{\text{overhead}}\) represents the penalty that we have to pay for running the program in parallel. We hold three major factors accountable for this overhead.

The first factor is the overhead of scheduling the tasks. Although this overhead is negligible for a small number of tasks, it becomes significant as the number of created tasks becomes larger.

In an ideal case, when \(n_t\) tasks are created, \(\lceil \frac{n_t}{N} \rceil\) tasks would be created on core_0. If we represent the overhead of scheduling a task on one core with \(\alpha\), then \(\alpha \lceil \frac{n_t}{N} \rceil\) would be the portion of \(t_{\text{overhead}}\) associated with \(n_t\) tasks.

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. Work stealing 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 unsuccessful steal efforts becomes 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 its 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 \(--\ hpx: \text{numa-sensitive}--\) 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.
Figure 5.1. The results of running the parallel for-loop benchmark with $p_s = 3000$, on 4 and 8 cores, with and without work stealing. The vertical gray line shows the grain size that would generate the same number of tasks as the number of cores, with the same amount of work for all the cores.
Figure 5.1 shows the results of running a parallel for-loop benchmark with and without work stealing. The benchmark consists of an 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 section.

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.

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.

The third factor is the overhead due to contention and coherency based on USL. Equation 5.2 shows how USL models the overheads due to contention($\sigma$) and coherency($\kappa$) in overall execution time $t$, with sequential execution time of $t_{seq}$, on $N$ cores, when $\frac{t_{seq}}{N}$ is the expected execution time on $N$ cores in ideal case when the mentioned overheads did not exist.

$$speedup = \frac{t_{seq}}{t} = \frac{N}{1 + \sigma(N-1) + \kappa N(N-1)} \Rightarrow$$

$$t = \frac{t_{seq}}{N} + \sigma(N-1) + \kappa N(N-1)$$

(5.2)

Based on equation 5.2, we added the term $\sigma(N-1)t_{max}$ to equation 5.6 to represent the overhead observed due to the contention effect, assuming $t_{max}$ 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.

$$t_{\_overhead} = a \left\lceil \frac{n_t}{N} \right\rceil + \sigma t_{max}(N-1)$$

(5.3)

Moreover, we adjusted equation 5.3 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 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 $g$, $n_t$ tasks are being created, and $M$ cores are actually doing the work. If $n_t < N$, $M$ would be equal to $n_t$, otherwise $M = N$.

$$M = \begin{cases} n_t & \text{if } n_t < N \\ N & \text{otherwise} \end{cases}$$  \hspace{1cm} (5.4)

With these assumptions 5.3 then converts into:

$$\text{execution} = \alpha \left\lceil \frac{n_t}{N} \right\rceil + t_{\text{max}} + \sigma t_{\text{max}}(M-1)$$  \hspace{1cm} (5.5)

With this assumption equation 5.1 then becomes:

$$\text{execution} = \alpha \left\lceil \frac{n_t}{N} \right\rceil + t_{\text{max}}$$  \hspace{1cm} (5.6)

We had previously defined $t_{\text{max}}$ as the time it takes to perform $w_{\text{max}}$ amount of work, where $w_{\text{max}}$ is the maximum amount of work assigned to the cores.

For a balanced parallel for-loop, $w_{\text{max}}$ can be calculated as:

$$w_{\text{max}} = \begin{cases} p_s & \text{if } N = 1 \\ p_s - g(N - 1)(\left\lceil \frac{n_t}{N} \right\rceil - 1) & \text{if } n_t \% N = 1 \text{ and } n \% c_s \neq 0 \\ g \left\lceil \frac{n_t}{N} \right\rceil & \text{otherwise} \end{cases}$$  \hspace{1cm} (5.7)

When $N = 1$, all the work will be assigned to the only available core, causing $w_{\text{max}}$ to be equal to all the available work, $p_s$. When $N > 1$, in the general case at most $\left\lceil \frac{n_t}{N} \right\rceil$ 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 $g \left\lceil \frac{n_t}{N} \right\rceil$ assigned to a core.
It should be noted that the last chunk of work created could contain a smaller amount of work than the $c_s$, and this is the case when $n \% c_s \neq 0$. This, although causing a smaller 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 $n_t \% N = 1$. For these special cases where $n_t \% N = 1$ and $n \% c_s \neq 0$ the maximum amount of work assigned to the cores would be $p_s - g(N - 1)(\lceil \frac{n_t}{N} \rceil - 1)$.

Figure 5.2 illustrates how $w_{max}$ is calculated for this case. Each square represents a chunk, or one task, containing $g$ amount of work generating $n_t$ chunks in total, and each column represents one core. The last chunk contains $g' = n \% c_s$ amount of work, where $g' < g$ and $gN(\lceil \frac{n_t}{N} \rceil - 1) + g' = p_s$.

As for $t_{max}$, the time to execute $w_c$ amount of work, where total amount of work available is $p_s$, since we are looking into balanced parallel for-loops with almost same amount of work
at each iteration, we can estimate \( t_{\text{max}} \) as:

\[
t_{\text{max}} = t_{\text{seq}} \frac{w_{\text{max}}}{p_s}
\]  

(5.8)

Where \( t_{\text{seq}} \) is the to run the whole amount of work, \( p_s \), sequentially. Equation 5.3 is then simplified into:

\[
\text{execution\_time} = \alpha \left\lceil \frac{n_t}{N} \right\rceil + t_{\text{seq}} \frac{w_{\text{max}}}{p_s} (1 + \sigma (M - 1))
\]  

(5.9)

We refer to equation 5.9 as our proposed analytical model in the next sections.

In summary, we deducted the following list as determining factors of the execution time in a balanced parallel for-loop:

- number of tasks created
- number of cores
- 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.

In addition, the two factors \( \alpha \) and \( \sigma \) represent the system-architecture that is constant between runs. Invariability of the architecture assists us to better predict the behavior of the same program with different input data as well as other new programs.

5.1. Evaluating the Proposed Model

In order to evaluate the model, we developed a benchmark of HPX parallel for-loops as shown in Listing 5.1 and ran it on a wide range of iteration lengths, chunk sizes, and number of threads. We refer to this benchmark as the for-loop benchmark. In each iteration, a while loop keeps the core busy for a certain amount of time denoted as \( t_i \).
Listing 5.1: For-loop benchmark, a simple hpx for_loop used to study the effect of grain size on the achieved parallelism.

```cpp
void measure_function_futures_for_loop(std::uint64_t count, bool csv, std::uint64_t chunk_size, std::uint64_t iter_length)
{
    // start the clock
    high_resolution_timer walltime;
    hpx::parallel::for_loop(hpx::parallel::execution::par.with(
        hpx::parallel::execution::dynamic_chunk_size(chunk_size)),
    0, count, &[](std::uint64_t) { worker_timed(iter_length*1000); });
    // stop the clock
    const double duration = walltime.elapsed();
    print_stats("for_loop", "par", "parallel_executor", count, duration, csv);
}

int hpx_main(variables_map& vm)
{
    const int repetitions = vm["repetitions"].as<int>();
    num_threads = hpx::get_num_worker_threads();
    const std::uint64_t chunk_size = vm["chunk_size"].as<std::uint64_t>();
    const std::uint64_t iter_length = vm["iter_length"].as<std::uint64_t>();
    const std::uint64_t count = vm["num_iterations"].as<std::uint64_t>();
    bool csv = vm.count("csv") != 0;
    if (HPX_UNLIKELY(0 == count))
        throw std::logic_error("error: count of 0 futures specified\n");
    for (int i = 0; i < repetitions; i++)
    {
        measure_function_futures_for_loop(count, csv, chunk_size, iter_length);
    }
    return hpx::finalize();
}

inline void worker_timed(std::uint64_t delay_ns)
{
    if (delay_ns == 0)
        return;
    std::uint64_t start = hpx::util::high_resolution_clock::now();
    while (true)
    {
        if ((hpx::util::high_resolution_clock::now() - start).seconds() >= delay_ns)
            break;
    }
}

int main(int argc, char * argv[])
{
    // Configure application-specific options.
    options_description cmdline("usage: " HPX_APPLICATION_STRING " [options]");
    cmdline.add_options()("num_iterations", value<std::uint64_t>()-->default_value(500000), "number of iterations to invoke")
    ("repetitions", value<int>()-->default_value(1), "number of repetitions of the full benchmark")
    ("iter_length",value<std::uint64_t>()-->default_value(1), "length of each iteration")
    ("chunk_size",value<std::uint64_t>()-->default_value(1), "chunk size");
    // Initialize and run HPX.
    return init(cmdline, argc, argv);
}
```
By setting \( t_i = 1\mu\text{sec} \), and changing number of iterations \( n \), and chunk sizes \( c_s \), we can see how the execution time changes when the benchmark is executed on different number of cores.

Having defined \( p_s \) as the total amount of work that has to be performed, for this benchmark \( p_s \) would be the time it takes to execute all the iterations, which is \( t_i \times n \). Since we have set \( t_i = 1\mu\text{sec} \) in this benchmark, \( p_s = n \). On the other hand, for this specific problem \( w_{max} = t_{max} \), and \( t_{seq} = p_s \).

In order to test how well the suggested model captures the relationship between execution time and grain size, we ran the benchmark with different configurations of number of cores \( (N) \), number of iterations \( (n) \), and chunk sizes \( (c_s) \). For each \( n \), we change \( c_s \) from 1 to \( n \) in logarithmic scale. Each of these runs was executed on 1, 2, 3, ..., 8 cores.

Figure 5.3 shows the measured execution time in terms of grain size for \( p_s = 100000 \), on different number of cores.

![Figure 5.3. The results of running the parallel for-loop benchmark with \( p_s = 100000 \), on different number of cores.](image)

Using the collected data points for each problem size \( (p_s) \), we utilized the \texttt{optimize.curve-fit} package from \textit{scipy} library in Python to fit our model to the collected data. Figure 5.4 shows
the fitted model and the original data for \( p_s = 100000 \), on 8 cores.

![Graph](image)

Figure 5.4. The results of predicted values of execution time through curve fitting vs the real data for \( p_s = 100000 \), for 8 cores.

To evaluate our model, we measured the relative error and \( R^2 \) score of the prediction for each problem size based on equation 5.10 and 5.11. In these equations \( p_k \) is the predicted value for sample \( k \), \( t_k \) is true measured value, and \( K \) is the total number of samples.

\[
\text{Relative_error} = \frac{1}{K} \sum_{k=1}^{K} |1 - \frac{p_k}{t_k}|
\]

(5.10)

\[
R^2 = 1 - \frac{\sum_{k=1}^{K} (t_k - p_k)^2}{\text{Var}(t)}
\]

(5.11)

Figure 5.5 illustrates the calculated relative error and \( R^2 \) score for problem sizes (\( p_s \)) of 10000, 100000, 1000000, 10000000, 100000000 iterations. For each of the \( p_s \)s, 440, 512, 440, 512, and 728 data points were collected respectively, by running the benchmark with a range of different chunk sizes to cover different regions of grain size. The relative error and \( R^2 \) score are calculated for each specific number of cores individually.

As discussed earlier, this model depends on the number of cores, tasks created on them, and the ratio of the maximum workload assigned to a single core and the problem size. Model parameters, \( \alpha \) and \( \sigma \), depend on the system architecture and type of the parallel application,
and we wouldn't expect them to change for different problem sizes.

Therefore, it would be sufficient to rely on collected data from one problem size rather than all measured data to find parameters’ $\alpha$ and $\sigma$ values.

In the next step, we used the parameters identified through curve-fitting based on the data collected from one specific problem size, to estimate the predicted execution time of other problem sizes, and measured the relative error and $R^2$ score. The obtained results are illustrated in Figure 5.6.

As it can be seen, using the data collected for $p_s = 1000$ to estimate the model parameters would generate higher prediction error on other problem sizes, but for other problem sizes we don’t see a considerable change in prediction error when data from different problem sizes were used to estimate the model parameters. Since larger problem sizes require more data to be collected, we selected $p_s = 100000$ as a reasonable problem size to estimate the model parameters.
Figure 5.5. The relative error, and $R^2$ score of fitting the measured data for different problem sizes, calculated for each number of cores.
Fitting our model to all the 512 data points collected resulted in model parameters $\alpha = 2.416$ and $\sigma = 0.0251$, where $\alpha$ represents the overhead of scheduling the tasks in $\mu$secs, and $\sigma$ represents the contention defined in the Universal Scalibility Law.

We run the for-loop benchmark on the system we want to run our parallel application on for $p_s = 100000$ to estimate $\alpha$ and $\sigma$. Plugging the estimated parameters into (5.9) would create the analytical model that could be used for other balanced parallel for-loop applications on
the same system.

Figure 5.7 shows the fitted curves along with the original data for different number of cores.

5.2. Testing the Proposed Model on Other Architectures

On the next step we repeated the same process for the data collected from another architecture, Medusa node with a Skylake CPU. We used $p_s = 100000$ as the base $p_s$ on both architectures. On Medusa, with $p_s = 100000$, the model parameters were found to be $\alpha = 1.673$ and $\sigma = 0.023$. The calculated relative error and $R^2$ score for each individual number of cores for $p_s = 100000$ is demonstrated in Figure 5.9, and Figure 5.10, while the relative error and $R^2$ score calculated for all other problem sizes is shown in Figure 5.9, and Figure 5.10.
Figure 5.7. The results of predicted values of execution time through curve fitting vs the real data for $p_3 = 100000$, for different number of cores.
Figure 5.8. The results of predicted values of execution time through curve fitting vs the real data for $p_3 = 100000000$, for different number of cores.
Figure 5.9. The relative error, and $R^2$ score of fitting the collected data to the proposed analytical model for different problem sizes, calculated for each number of cores on Medusa node.
Figure 5.10. The relative error and $R^2$ score of using the model parameters found from each base $p_s$, fitting the proposed analytical model to the collected data for different $p_s$, calculated over each number of cores on Medusa node.
Figure 5.11. The results of predicted values of execution time through curve fitting vs the real data for $p_s = 100000$, for different number of cores on Medusa node.
Figure 5.12. The results of predicted values of execution time through curve fitting vs the real data for $p_s = 100000000$, for different number of cores on Medusa node.
5.3. **Identifying the Optimal Range of Grain Sizes**

The proposed model suggests a range of grain sizes resulting in the optimum execution time for a given application running on a specific architecture. As illustrated earlier in Figure 2.1, the execution time versus grain size graph in logarithmic scale, denoted as the *bathtub curve*, can be divided into three regions. We refer to these regions as the left side, the right side, and the flat regions of the graph. Figure 5.13 shows an example of the flat region of the execution time versus grain size graph in both linear and logarithmic scales. As it can be observed, the flat region contains a small section of range of different grain sizes.

At the right side of the graph with large grain sizes, the number of tasks created is smaller than the number of cores which results in idling at least one of the cores, while the other cores are assigned a rather big chunk of work. The performance degradation we observe at those points is associated with starvation. 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 ($t_{max}$) is the dominant factor in determining the execution time.

On the other hand, on the left 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 a performance degradation due to starvation.

Our intention in this dissertation is to find this flat region of the bathtub curve, as shown
in Figure 5.13a, in order to ensure that the effect of the two major sources of performance
degradation (overheads of scheduling tasks, and starvation) in execution time is minimized.

![Graph](image)

(a) logarithmic scale

![Graph](image)

(b) linear scale

Figure 5.13. The results of running the for-loop benchmark with \( p_s = 100000 \), on 8 cores in
logarithmic and linear scale.

5.3.1. Left side of the Graph

As stated earlier, in equation 5.9, for small grain sizes the first term(\( \alpha \lceil \frac{n_t}{N} \rceil \)) is the dominant
factor while the second term(\( t_{sed} \frac{w_{max}}{p_s} (1 + \sigma (M - 1)) \)) roughly stays constant. Likewise, 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 can assume that the second factor is constant in that region. Also we can change \( N \) to \( M \), knowing that our concern is on the left side of the graph, where \( n_t \) is definitely greater than the number of cores. Taking the derivative of the function based on the grain size then leads to:

\[
\frac{\partial \text{execution time}}{\partial g} = \frac{\alpha}{N} \times \frac{\partial n_t}{\partial g} = \frac{\alpha}{N} \times \frac{\partial (\frac{p_s}{g})}{\partial g} = \frac{\alpha}{N} \times p_s \times \frac{-1}{g^2}
\]

From Formula 5.12, it can be observed that for the left 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 \( \lambda_b \), where \( \lambda_b \ll 1 \), for this purpose).

\[
\frac{\alpha}{N} \times p_s \times \frac{1}{g^2} \leq \lambda_b
\]

\[
g^2 \geq \frac{\alpha}{N} \times p_s \frac{1}{\lambda_b}
\]

\[
g \geq \sqrt{\frac{\alpha}{N} \times p_s} \frac{1}{\lambda_b}
\]

Formula 5.13 can also be represented as shown in Formula 5.14. This representation shows that when the ratio of the time it takes to execute one task to the total overhead of creating and managing \( n_t \) tasks on \( N \) core, is greater than a threshold, we will end up in the flat region of the graph, close to the left side.

\[
\frac{\alpha}{N} \times p_s \times \frac{1}{g} \leq \lambda_b \Rightarrow \frac{\alpha}{N} \times n_t \times \frac{1}{g} \leq \lambda_b
\]

\[
\frac{\alpha}{N} \times n_t \leq g \times \lambda_b \Rightarrow \frac{g}{\frac{\alpha}{N} \times n_t} \geq \frac{1}{\lambda_b}
\]
5.3.2. **Right side of the graph**

Now looking at the right 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 \((w_{max})\) and consequently the time to perform this amount of work \((t_{max})\) is the dominant factor.

Formula 5.7 shows how \(w_{max}\) is calculated for different cases, but in general we can estimate \(w_{max}\) with \(g \times \lceil \frac{n_t}{N} \rceil\).

\[
w_{max} \approx g \times \left\lceil \frac{n_t}{N} \right\rceil \\
\approx g \times \left\lceil \frac{p_s}{g} \frac{N}{N} \right\rceil \\
\approx \frac{p_s}{N} \quad \text{if} \quad n_t \geq N
\] (5.15)

What happens here is that as the grain size changes, there are points for which \(\left\lceil \frac{n_t}{N} \right\rceil\) is the same but since the grain size is different, a different \(w_{max}\) would be resulted. For all the values of \(g\) that create the same \(\left\lceil \frac{n_t}{N} \right\rceil\), as \(g\) increases the difference between \(w_{max}\) and \(\frac{p_s}{N}\) increases.

For example, considering a case where \(p_s = 100000\), and \(N = 8\), for the grain sizes in range of \([4167, 6249]\) would result in creating between \(\left\lceil \frac{100000}{6249} \right\rceil = 17\) and \(\left\lceil \frac{100000}{4167} \right\rceil = 24\) tasks. This amount of tasks created itself would result in \(\left\lceil \frac{17}{8} \right\rceil = 3\) and \(\left\lceil \frac{24}{8} \right\rceil = 3\) tasks. On the other hand, \(w_{max} = g \times \left\lceil \frac{n_t}{N} \right\rceil = 3 \times g\), would have a value in range of \([3 \times 4167, 3 \times 6249] = [12501, 18747]\), where the average amount of work per core is \(\frac{p_s}{N} = 12500\). 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 \( \left\lceil \frac{n_t}{N} \right\rceil \) as \( k \), then:

\[
\begin{align*}
 k - 1 & < \frac{n_t}{N} \leq k \\
(k - 1) \times N & < n_t \leq k \times N \\
(k - 1) \times N & < \left\lfloor \frac{p_s}{g} \right\rfloor \leq k \times N \\
(k - 1) \times N & < \frac{p_s}{g} \leq k \times N
\end{align*}
\]  

(5.16)

If \( k = 1 \), then,

\[
\begin{align*}
0 & < \frac{p_s}{g} \leq N \\
\frac{p_s}{N} & \leq g \leq p_s.
\end{align*}
\]  

(5.17)

Otherwise, when \( k > 1 \),

\[
\frac{p_s}{k \times N} \leq g < \frac{p_s}{(k - 1) \times N}.
\]  

(5.18)

Since \( \left\lceil \frac{n_t}{N} \right\rceil = k \), and \( w_{\text{max}} = g \times \left\lceil \frac{n_t}{N} \right\rceil = k \times g \) if \( n_t \% N \neq 1 \), we can conclude for \( k > 1 \):

\[
\begin{align*}
 k \times \frac{p_s}{k \times N} & \leq w_{\text{max}} < k \times \frac{p_s}{(k - 1) \times N} \\
\frac{p_s}{N} & \leq w_{\text{max}} < \frac{k}{k - 1} \times \frac{p_s}{N} \\
0 & \leq w_{\text{max}} - \frac{p_s}{N} < \frac{1}{k - 1} \times \frac{p_s}{N}
\end{align*}
\]  

(5.19)

For the cases where \( k > 1 \) and \( n_t \% N = 1 \), there could be a change in \( w_{\text{max}} \) if \( p_s \% g \neq 0 \). For these cases:

\[
\begin{align*}
\left\lceil \frac{n_t}{N} \right\rceil & = k \hspace{1em} \& \hspace{1em} n_t \% N = 1 \Rightarrow \\
n_t & = (k - 1) \times N + 1 \Rightarrow \\
(k - 1) \times N & < \frac{p_s}{g} \leq (k - 1) \times N + 1 \Rightarrow \\
\frac{p_s}{(k - 1) \times N + 1} & \leq g < \frac{p_s}{(k - 1) \times N}
\end{align*}
\]  

(5.20)
From Formula 5.7 we know,

\[ w_{\text{max}} = p_s - (k - 1) \times (N - 1) \times g. \tag{5.21} \]

Therefore,

\[
(k - 1)(N - 1) \frac{ps}{(k - 1)N + 1} \leq (k - 1)(N - 1)g < (k - 1)(N - 1) \frac{ps}{(k - 1)N} \Rightarrow \\
\frac{ps}{N} < ps - (k - 1)(N - 1)g \leq k \frac{ps}{(k - 1)N + 1} \tag{5.22}
\]

\[
\frac{ps}{N} < w_{\text{max}} \leq k \times \frac{ps}{(k - 1)N + 1}
\]

And for \( k = 1 \), where \( n_t \leq N \),

\[ w_{\text{max}} = g, \quad \frac{p_s}{N} \leq g \leq p_s \Rightarrow \]

\[ 0 \leq w_{\text{max}} - \frac{p_s}{N} = g - \frac{p_s}{N} \leq (N - 1) \times \frac{p_s}{N} \tag{5.23} \]

Defining \( \text{imbalance\_ratio} = \frac{w_{\text{max}} - \frac{p_s}{N}}{\frac{p_s}{N}} \), then,

\[ 0 \leq \text{imbalance\_ratio} \leq N - 1 \quad \text{for} \quad k = 1 \]

\[ 0 \leq \text{imbalance\_ratio} < \frac{1}{k - 1} \quad \text{for} \quad k > 1 \text{ and } n_t \% N \neq 1 \tag{5.24} \]

\[ 0 \leq \text{imbalance\_ratio} < \frac{N - 1}{N(k - 1) + 1} = \frac{1}{k - 1 + \frac{k}{N-1}} \quad \text{otherwise} \]

Formula 5.24 shows that as number of created tasks increases, as long as number of tasks per core is the same, the imbalance factor decreases.

Figure 5.14 shows the imbalance ratio calculated for different grain sizes for \( p_s = 10000 \), on 8 cores. Each of the regions between two dashed green lines correspond to a specific value for \( k = \left\lceil \frac{n_t}{N} \right\rceil \).

At each of the regions with \( k > 1 \), \( \left\lceil \frac{n_t}{N} \right\rceil = k \), \( \text{imbalance\_ratio} \) starts from 0 and ap-
approaches \( \frac{1}{k-1} \left( \frac{1}{k-1} + \frac{k}{N-1} \right) \) for regions where \( n_i \% N \neq 1 \) at the end of the region. When \( k = 1 \), \textit{imbalance\_ratio} increases linearly starting from 0 and reaching the maximum of \( N - 1 \) when \( g = p_s \). As we move to larger grain sizes, \( \left\lceil \frac{n_i}{N} \right\rceil \) decreases, therefore the upper-bound for \textit{imbalance\_ratio} increases.

![Figure 5.14](image)

Figure 5.14. The imbalance ratio calculated for different grain sizes for \( p_s = 100000 \), on 8 cores. At the area between each two green lines \( k = \left\lceil \frac{n_i}{N} \right\rceil \) is constant.

Figure 5.15 represents the imbalance ratio, along with the ratio of the sequential execution time over execution time (speed-up) against grain size for \( p_s = 100000 \), ran on 8 cores. As it can be observed, as the \textit{imbalance\_ratio} increases, the speed-up decreases.
Figure 5.15. The imbalance ratio alongside speedup for different grain sizes for $p_s = 100000$, on 8 cores, where $k = \lceil \frac{n_i}{N} \rceil$.

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, $\lambda_s$ ($0 < \lambda_s < 1$), where for imbalance ratios smaller than this threshold the imbalance effect is not significant. As we get close to this threshold, we are likely to reach the right 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 5.23 states that for grain sizes greater than $p_s$, $imbalance\_ratio$ increases linearly with grain size from 0 to $N - 1$. While for grain sizes smaller than $p_s$, the maximum $imbalance\_ratio$ depends on $k = \lceil \frac{n_i}{N} \rceil$. So, in order to ensure $imbalance\_ratio$ is smaller than or equal to a threshold ($\lambda_s$), first we search the grain sizes smaller than $\frac{p_s}{N}$. 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 $k_{\text{min}}$ (creating an imbalance ratio between 0 and $\frac{1}{k_{\text{min}}-1}$), where $\frac{1}{k_{\text{min}}-1} \leq \lambda_s$, $\forall k < k_{\text{min}}$ maximum value
of $imbalance\_ratio$ would be greater than $\lambda_s$. So in order to find the grain size that would create maximum $imbalance\_ratio$ of $\lambda_s$:

$$imbalance\_ratio \leq \lambda_s \Rightarrow$$

$$\frac{1}{k-1} \leq \lambda_s$$

$$k \geq 1 + \frac{1}{\lambda_s}$$

$$k_{min} = \left\lceil 1 + \frac{1}{\lambda_s} \right\rceil + 1$$

$$g < \frac{p_s}{(k_{min} - 1) \times N}$$

$$g_{max} = \frac{p_s}{(k_{min} - 1) \times N} - 1 = \frac{p_s}{(1 + \left\lceil \frac{1}{\lambda_s} \right\rceil) \times N}$$

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

5.4. 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. Integrating Formula 5.13 and Formula 5.25 suggests the following range for minimum execution time:

$$\sqrt{\left(\frac{g}{N}\right) \times \frac{p_s}{\lambda_b}} \leq g \leq \frac{p_s}{(1 + \left\lceil \frac{1}{\lambda_s} \right\rceil) \times N}$$

(5.26)

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

Note that based on Formula 5.26 we can state,

$$\sqrt{\left(\frac{g}{N}\right) \times \frac{p_s}{\lambda_b}} \leq g \leq \frac{p_s}{(1 + \left\lceil \frac{1}{\lambda_s} \right\rceil) \times N} < \frac{p_s}{N}.$$ 

(5.27)

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, $\frac{p_s}{N}$. 

64
5.5. Locating the Optimum Range of Grain Sizes for the for-loop Benchmark

In this section we used Formula 5.26 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 \( \lambda_b \) to 0.01 and \( \lambda_s \) 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 5.27. \( \lambda_b \) indicates the slope of the graph at the left side of the graph where overhead of tasks is the dominant factor. \( \lambda_b = 0.01 \) shows an angle of around 0.5°, which seems to be a reasonable threshold. Grain sizes smaller than \( \sqrt{\frac{(\frac{p}{N})\times p_s}{\lambda_b}} \) would create a slope of more than \( \lambda_b \). As for \( \lambda_s \), a grain size greater than \( \frac{p_s}{(1+\frac{1}{\lambda_s})\times N} \) could generate an imbalance_ratio of greater than \( \lambda_s \). Therefore, for grain sizes within the range in Formula 5.27 we are in a safe region.

Figure 5.16 and Figure 5.17 show the identified regions for minimum execution time, for \( p_s = 10000, 100000, 1000000, 10000000, 100000000 \), with 8 cores on two different architectures. The gray dashed line represents the grain size where work is equally divided among the cores, \( \frac{p_s}{N} \).

Selecting a greater value for \( \lambda_b \) 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 \( \lambda_s \) would result in shifting the right border of the region to left, imposing a higher restriction on imbalance_ratio, as shown in Figure 5.18(ran on Marvin node) and Figure 5.27(ran on Medusa node).
Figure 5.16. The identified range of grain size for minimum execution time for $p_s = 10000, 100000, 1000000, 10000000, 100000000$, on 8 cores, with $\lambda_b = 0.01$ and $\lambda_s = 0.1$ on Marvin node. The gray dashed line represents the grain size where work is equally divided among the cores, $p_s/N$. 
Figure 5.17. The identified range of grain size for minimum execution time for $p_s = 10000, 100000, 1000000, 10000000, 100000000$, on 8 cores, with $\lambda_b = 0.01$ and $\lambda_s = 0.1$ on Medusa node. The gray dashed line represents the grain size where work is equally divided among the cores, $\frac{p_s}{N}$.
Figure 5.18. An example of the effect of $\lambda_b$ and $\lambda_s$ on the borders of the identified region for minimum execution time due to change for $p_s = 100000000$ and $N = 8$ on Marvin node.

Figure 5.19. An example of the effect of $\lambda_b$ and $\lambda_s$ on the borders of the identified region for minimum execution time due to change for $p_s = 100000000$ and $N = 8$ on Medusa node.
5.6. 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 $p_s$ 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, and 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 from the Blazemark suite. The benchmark calculates $C = A + B$ where $A$, $B$, and $C$ are row major square dense matrices for 60 different matrix sizes from 1 to 7000. Since the parallelization threshold for $DMatDMatADD$ is set to 36100 elements which is equivalent to a matrix of size 193 by default, we used matrix sizes from 230 to 7000 in our experiments. For each matrix size the expression is evaluated 6 times and the average execution time of the last 5 runs is returned as the output.

As we previously stated in Chapter 5, 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, as 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 $4 \times 256$ for block_size for this benchmark.
In the previous section we provided Formula 5.5 as an analytical model to explain the relationship between execution time and grain size in a balanced parallel for-loop.

\[
\text{execution\_time} = \alpha \times \left\lceil \frac{n_t}{N} \right\rceil + t_{seq} \times \frac{w_{\text{max}}}{p_s} \times (1 + \sigma \times (M - 1)) \tag{5.28}
\]

In Formula 5.5, \(p_s\) refers to the total amount of work available. If we denote matrix size with \(m\), performing \textit{DMatDMatADD} benchmark would result in \(p_s = 2 \times m^2\), which is the total number 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 \textit{Marvin} node as \(\alpha = 2.416\) and \(\sigma = 0.0251\). 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 \textit{block\_size} = \(4 \times 256\), and \textit{chunk\_size} = \(3\) performing \textit{DMatDMatADD} benchmark, would result in a grain size of \(2 \times 4 \times 256 \times 3 = 6144\).

Figure 5.20 and Figure 5.21 show the results of applying the proposed analytical model with the parameters identified through the benchmark, to \textit{DMatDMatADD} benchmark, alongside the true values for the execution time, for \(m = 690\) and \(m = 4222\) respectively, for different number of cores.

The \textit{DMatDMatADD} benchmark was run on \textit{Marvin} node with \textit{Sandy Bridge} architecture, with \(256\)\(KB\) of level 2 cache, and \(20\)\(MB\) of level 3 cache. In order for the 3 same sized square matrices of type double involved in \textit{DMatDMatADD} benchmark \((C = A + B)\) to fit into level3 cache, they must be smaller than \(935 \times 935\) 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 5.21. We have to pay a penalty for loading data from memory instead of cache.

Figure 5.22 shows the relative error, while Figure 5.23 represents the \(R^2\) score both calculated for each specific number of cores individually, and among matrix sizes smaller than 953.
Figure 5.20. The results predicting the execution time in terms of grain size based on the proposed model compared to the original values for $DMatDMatADD$ benchmark with $m = 690$ on $Marvin$ node, ran on different number of cores.
Figure 5.21. The results predicting the execution time in terms of grain size based on the proposed model compared to the original values for $DMatDMatADD$ benchmark with $m = 4222$ on $Marvin$ node, ran on different number of cores.
Figure 5.22. The relative error of predicting the execution time in terms of grain size based on the proposed model for $DMatDMatADD$ benchmark on $Marvin$ node. The relative error of matrix sizes smaller than 953 was averaged for each specific number of cores.

Figure 5.23. The $R^2$ score of predicting the execution time in terms of grain size based on the proposed model for $DMatDMatADD$ benchmark on $Marvin$ node. The relative error of matrix sizes smaller than 953 was averaged for each specific number of cores.

For $Medusa$ node with $Skylake$ architecture, with 1024KB of level 2 cache, and 28MB of level 3 cache, in order for the matrices involved in the $DMatDMatADD$ benchmark to fit into level3 cache, they must be smaller than $1097 \times 1097$ in size.

For matrix sizes greater than 1097, we observe an increase in execution time from our prediction using the proposed analytical model, as represented in Figure 5.21. We have to pay a penalty for loading data from memory instead of cache.

Figure 5.22 shows the relative error, while Figure 5.23 represents the $R^2$ score both cal-
culated for each specific number of cores individually, among for matrix sizes smaller than 1097.
Figure 5.24. The results predicting the execution time in terms of grain size based on the proposed model compared to the original values for *DMat* benchmark with $m = 690$ on Medusa node, ran on different number of cores.
Figure 5.25. The results of predicting the execution time in terms of grain size based on the proposed model compared to the original values for $DMatDMatADD$ benchmark with $m = 4222$ on $Medusa$ node, ran on different number of cores.
Figure 5.26. The relative error of predicting the execution time in terms of grain size based on the proposed model for DMatDMatADD benchmark on Medusa node. The relative error of matrix sizes smaller than 10^9 was averaged for each specific number of cores.

Figure 5.27. The $R^2$ score of predicting the execution time in terms of grain size based on the proposed model for DMatDMatADD benchmark on Medusa node. The relative error of matrix sizes smaller than 10^9 was averaged for each specific number of cores.

Although we had completely ignored cache effects up until here for simplification, Figure 5.21 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 5.28 and Figure 5.29 show the identified range for minimum execution time based on Formula 5.10 for different matrix sizes ran on 8 cores on Marvin and Medusa nodes respectively.
Figure 5.28. The identified range for minimum execution time based on the analytical model vs the real data for *DMatDMatADD* benchmark for $m = 690, 912, 1825, 3193, 4222, 4855, 6420$ on 8 cores on *Marvin* node, with $\lambda_b = 0.01$ and $\lambda_s = 0.1$. 78
Figure 5.29. The identified range for minimum execution time based on the analytical model vs the real data for $DMatDMatADD$ benchmark for $m = $, for $m = 690, 912, 1825, 3193, 4222, 4855, 6420$, on 8 cores on Medusa node, with $\lambda_b = 0.01$ and $\lambda_s = 0.1$. 
5.7. 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 $4 \times 256$ was selected as a safe choice for $DMatDMatADD$ benchmark. For each grain size in the identified range, the equivalent chunk size is calculated. Figure 5.30 shows the identified range of chunk size for minimum execution time based on Formula 5.10, 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 5.30. The suggested range of chunk size for minimum execution time with \texttt{block\_size} = 4 \times 256, for \texttt{DMatDMatADD} benchmark for \( m = 690, 912, 1825, 3193, 4222, 4855, 6420 \), on 8 cores on \texttt{Marvin} node, when \( \lambda_p = 0.1 \) and \( \lambda_s = 0.01 \).
Figure 5.31. The suggested range of chunk size for minimum execution time with \( \text{block\_size} = 4 \times 256 \), for \( \text{DMatDMatADD} \) benchmark for \( m = 690, 912, 1825, 3193, 4222, 4855, 6420 \), on 8 cores on \textit{Medusa} node, when \( \lambda_b = 0.1 \) and \( \lambda_s = 0.01 \).
In the next step we tested the proposed approach on *DMatDMatDMatADD* benchmark. The benchmark calculates $D = A + B + C$, where $A$, $B$, $C$, and $D$ are dense row-major square matrices. The main difference between this benchmark and the previously studied *DMatDMatADD* benchmark is in the number of floating point operations executed. For this problem $p_s = 2m^2$ instead of $m^2$ observed for *DMatDMatADD* benchmark. The other issue that should be noted is that since 4 matrices are involved in this benchmark the requirement for matrix sizes to fit into cache would be different. Running *DMatDMatDMatADD* benchmark, matrix sizes greater than 810 for *Marvin* node and 950 for *Medusa* node, would experience performance degradation due to cache effects.
Figure 5.32. The identified range for minimum execution time based on the analytical model vs the real data for $DMatDMatDMatADD$ benchmark for $m = 690, 912, 1825, 3193, 4222, 4855, 6420$, on 8 cores on $Marvin$ node, with $\lambda_p = 0.01$ and $\lambda_s = 0.1$. 
Figure 5.33. The identified range for minimum execution time based on the analytical model vs the real data for DMatDMatDMatADD benchmark for $m = 690, 912, 1825, 3193, 4222, 4855, 6420$, on 8 cores on Medusa node, with $\lambda_b = 0.01$ and $\lambda_s = 0.1$. 
Figure 5.34. The suggested range of chunk size for minimum execution time with \( block\_size = 4 \times 256 \) for \( DMatDMatDMatADD \) benchmark, for \( m = 690, 912, 1825, 3193, 4222, 4855, 6420 \), on 8 cores on \( Marvin \) node, when \( \lambda_b = 0.1 \) and \( \lambda_s = 0.01 \).
Figure 5.35. The suggested range of chunk size for minimum execution time with $\text{block\_size} = 4 \times 256$ for $\text{DMatDMatDMatADD}$ benchmark, for $m = 690, 912, 1825, 3193, 4222, 4855, 6420$, on 8 cores on $\text{Medusa}$ node, when $\lambda_b = 0.1$ and $\lambda_s = 0.01$. 
5.8. Conclusions

In this section 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 $DMatDMatADD$ and $DMatDMatDMatADD$ benchmark from Blazemark suite. We were also able to find the range of grain sizes 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 $p_s = 100000$, with different values for chunk size on different number of cores, and use the collected data to estimate the $\alpha$ and $\sigma$ 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 should 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($\lambda_b$ and $\lambda_s$).

In order to evaluate our proposed approach we chose the middle point of the collected chunk sizes within the range as our suggested chunk size, and compared the execution time
obtained with this grain size with the minimum achievable execution time among with all different chunk sizes, and the chunk size that would create as many tasks as the number of cores (represented with equal). Speedup is calculated as the ratio of equal or min execution time to the execution time of the proposed chunk size.

Table 5.2 shows the percentage of the individual runs (a specific matrix size and number of cores) for which an improvement from the execution time when as many tasks as the number of cores as created, along with the percentage of runs for which the observed execution time is within the 10% range of the minimum execution time. The collected data shows that using the analytical model to find the optimum grain size helps us improve the achievable performance through creating as many tasks as the number of cores, in most of the cases. Results of running the benchmark on 1 core was excluded from the calculations.

Table 5.2. Comparing the obtained results from using the analytical model to find the optimum grain size with minimum execution time on *Marvin* node

<table>
<thead>
<tr>
<th>Node</th>
<th>Benchmark</th>
<th>Improvement from equal(%)</th>
<th>Within 10% of the minimum execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Marvin</em></td>
<td>DMatDMatADD</td>
<td>57%</td>
<td>68%</td>
</tr>
<tr>
<td><em>Marvin</em></td>
<td>DMatDMatDMatADD</td>
<td>58%</td>
<td>70%</td>
</tr>
<tr>
<td>Medusa</td>
<td>DMatDMatADD</td>
<td>63%</td>
<td>62%</td>
</tr>
<tr>
<td>Medusa</td>
<td>DMatDMatDMatADD</td>
<td>64%</td>
<td>60%</td>
</tr>
</tbody>
</table>
Figure 5.36. The average speedup compared to creating as many tasks as number of cores (equal) and the minimum execution time (min) for \textit{DMatDMatADD} benchmark on \textit{Marvin} and \textit{Medusa} node. The speedup was averaged over all matrix sizes for each specific number of cores.
Figure 5.37. The average speedup compared to creating as many tasks as number of cores (equal) and the minimum execution time (min) for DMatDMatDMatADD benchmark on Marvin and Medusa node. The speedup was averaged over all matrix sizes for each specific number of cores.
Chapter 6. Runtime Task Granularity Control through Splittable Tasks

6.1. Splittable Tasks

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 sizes 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 sizes 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 utilize our knowledge from identifying the optimum range of grain size based on the analytical model to extend a previously implemented algorithm for controlling task granularity.

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 [48].

Prell [48] intensively studies using splittable tasks for runtime adaptivity. 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 [48].

Next, based on lazy task scheduling [49], they suggest instead of creating at the tasks and
deciding 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, the original task is split into two parts. One part would be executed by the current worker, while the other part would be added to the current worker's deque as a splittable task to be stolen by idle workers. The split would only occur if the number of iterations assigned to the task is greater than a parameter $K$, which is a factor of the total number of iterations divided by the number of the workers [48].

There are different strategies for splitting, namely, Binary, Guided, and Adaptive splitting. In binary splitting [49], the task is split into two same sized tasks, while in guided splitting $\frac{1}{\text{chunks}}$ of the task is executed by the current worker and the rest is added to the worker's deque. The $\text{chunks}$ parameter is initialized to the number of workers and is decremented at each split. This type of split would be beneficial if many of the workers are idle. Adaptive splitting on the other hand, initializes $\text{chunks}$ parameter to number of workers like guided splitting, but adaptively changes it to the number of idle workers at each split [48].

Our work here is based on Prell [48]'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 Utilizing our proposed method for identifying the optimum range of grain size for minimum execution time, and use the lower-bound of the range as a cutoff value specific to system architecture and the application being executed, to stop splitting.
6.2. 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, \(T_1\) containing iterations in the range of \([a, c]\) and \(T_2\) would contain iterations from \([c, b]\), where \(c\) is the split point. \(T_1\) would be executed by the current worker while \(T_2\), 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.

The main difference between guided and adaptive splitting is the size of the task to be executed by the current thread and how the generated task is scheduled.

In guided splitting the split iteration index \(c\) is calculated as \(c = \frac{b - a}{N}\), where \(N\) is the total number of cores. The generated task, which is also splittable, would then be assigned to the next available worker. On the other hand, in adaptive splitting \(c = \frac{b - a}{M + 1}\) where \(M\) is the number of idle cores at the time split occurred. The created splittable task is then scheduled to be executed by the first identified idle core. The idle cores can be identified through an API function in HPX and are queried at the time of split. The API function returns a bitmask with the size of the number of cores, where a 1 at \(i\)th bit shows that the \(i\)th core is idle while a 0 represents a busy core.

In Adaptive splitting, after querying the idle cores, starting from the least significant bit, the first bit with value of 1 would be the core the task would be scheduled on. In order to make implicit scheduling of the work on the cores possible, and consequently to avoid the unnecessary overhead associated with work stealing, the splittable executor turns work stealing off automatically. Table 6.1 summarizes these two splitting strategies.
Table 6.1. Task split and scheduling in Guided and Adaptive splitting. \(a\) is the index of the first iteration, \(b\) is the last iteration, and \(c\) is the split location.

<table>
<thead>
<tr>
<th>Guided Splitting</th>
<th>Adaptive Splitting</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c = \frac{b-a}{N}) where (N) is the number of cores</td>
<td>(c = \frac{b-a}{M+1}) where (M) is the number of idle cores</td>
</tr>
<tr>
<td>Task (T_1) contains range ([a, c)) executed by the current worker</td>
<td>Task (T_1) contains range ([a, c)) executed by the current worker</td>
</tr>
<tr>
<td>Task (T_2) contains range ([c, b)) executed by the next available worker</td>
<td>Task (T_2) contains range ([c, b)) scheduled to be executed by the first identified idle worker</td>
</tr>
</tbody>
</table>

Figure 6.1 and Figure 6.2 illustrate the results of running the for-loop benchmark in Listing 5.1 using the splittable executor for \(p_s = 100000\) and \(p_s = 100000000\), in two modes guided and adaptive as shown in Table 6.1 on Marvin node. The results show that the splittable executor was successful in controlling the task granularity, in the sense that the obtained execution time was in the range of the optimum achievable execution time by changing the grain size. As it can be seen, the results for guided and adaptive splitting are very close to each other, since the benchmark forces the execution time of each iteration to be fixed.

Figure 6.3 and Figure 6.4 shows the results of running the for-loop benchmark using the splittable executor for \(p_s = 100000\) and \(p_s = 100000000\), on Medusa node.
Figure 6.1. The results of running the for-loop benchmark using splittable tasks with guided and adaptive splitting for $p_s = 100000$ on Marvin node, for different number of cores.
Figure 6.2. The results of running the for-loop benchmark using splittable tasks with guided and adaptive splitting for \( p_s = 100000000 \) on Marvin node, for different number of cores.
Figure 6.3. The results of running the for-loop benchmark using splittable tasks with guided and adaptive splitting for $p_s = 100000$ on Medusa node, for different number of cores.
Figure 6.4. The results of running the for-loop benchmark using splittable tasks with guided and adaptive splitting for $p_s = 100000000$ on Medusa node, for different number of cores.

99
6.3. Using the Proposed Analytical Model to Estimate the Stop Splitting Threshold

Intel’s TBB (threading Building Blocks) [50] utilizes two strategies for work stealing, *Single Partitioner* (SP) and *Adaptive Partitioner* (AP). SP introduces a stop splitting threshold (sst) which uses a parameter identified by the programmer to stop splitting. A task would not split if the number of iterations in the task is smaller than this threshold [49].

Auto partitioner, on the other hand, adds runtime adaptivity by creating $K \times P$ chunks, where $P$ is the number of cores [51]. They assign a variable $n$ to each range, initializing it to $K \times P$, every time a split occurs $n$ is halved. They also add another constant parameter $V$, each time a task is stolen the value of $n$ for that task is required to be at least $V$. Currently both $K$ and $V$ are set to 4 in their implementation [51].

Instead of setting one static threshold for splitting for every application, [48] suggest to stop splitting when the number of the remaining iterations gets smaller than a threshold $K$ which is a fraction of the number of iterations divided by the number of cores, for runtime adaptivity.

Here we propose to utilize the identified optimum range of grain sizes through our analytical model to adjust the split threshold based on the system architecture and the specific application being executed. For this purpose, a member variable with default value of 0, was added to the splittable task implementation denoting the minimum number of iterations each created task should include. On the other side, the implementation of the HPX backend for Blaze was changed to create splittable tasks with a minimum size (number of iterations) calculated at runtime for each expression being evaluated specific to the system architecture. Listing 6.1 shows the modified implementation of the HPX backend for Blaze.

Figures 6.5 and 6.6 show the results of applying the threshold calculated through the analytical model for two matrix sizes, 264, 1825, for *DMatDMatAdd* benchmark from Blaze-mark. As it can be seen, the effect of applying the threshold is more noticeable for smaller problem sizes, where the penalty of creating too many tasks is considerable compared to the total execution time.
Listing 6.1: 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<size_t>( BLAZE_HPX_MATRIX_BLOCK_SIZE_ROW ), (~rhs).rows() ) ) ;
    const size_t numCols ( min( static_cast<size_t>( BLAZE_HPX_MATRIX_BLOCK_SIZE_COLUMN ), (~rhs).columns() ) ) ;
    const size_t rest1 ( numRows & ( SIMDSIZE-1UL ) ) ;
    const size_t rowsPerIter ( 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 equalShare2 ( (~rhs).columns() / colsPerIter + addon2 ) ;
    hpx::parallel::execution::splittable_mode sptMode = hpx::parallel::execution::splittable_mode::all;
    if ( (BLAZE_HPX_SPLIT_TYPE_IDLE == 1) )
    { sptMode = hpx::parallel::execution::splittable_mode::idle; }
    hpx::parallel::execution::splittable_executor spt( sptMode, minChunkSize );
    for_loop( par.on( spt ), 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 ) ;
    } } ;
}
Figure 6.5. The results of running the $DMatDMatAdd$ benchmark using splittable tasks with adaptive and thresholded adaptive splitting for matrix size 264 on $Marvin$ node, for different number of cores.
Figure 6.6. The results of running the $DMatDMatAdd$ benchmark using splittable tasks with adaptive and thresholded adaptive splitting for matrix size 1825 on $Marvin$ node, for different number of cores.
Figure 6.7. The results of running the *DMatAdd* benchmark using splittable tasks with adaptive and thresholded adaptive splitting for matrix size 264 on *Medusa* node, for different number of cores.
Figure 6.8. The results of running the $DMatDMatAdd$ benchmark using splittable tasks with adaptive and thresholded adaptive splitting for matrix size 1825 on $Medusa$ node, for different number of cores.

Figure 6.9 compares the obtained results using the splittable executor in guided, adaptive,
guided with threshold, adaptive with threshold, and creating as many tasks as the number of cores by dividing the total amount of work equally among the cores (denoted with "equal"), while Figure 6.10 compares the results with the minimum execution time observed from the collected data with different grain sizes. In Figure 6.9 the speedup denotes the ratio of the execution time in the corresponding mode and the observed execution time when the same number of tasks as the number of cores are created, while in Figure 6.10 speedup is the ratio of the execution time to the minimum execution time collected. In the plots, the speedup is averaged over all the matrix sizes for each specific number of cores.
Figure 6.9. The average speedup compared to creating as many tasks as number of cores (equal mode) for different modes for $DMatDMatADD$ benchmark on $Marvin$ and $Medusa$ node. The speedup was averaged over all matrix sizes for each specific number of cores.
Figure 6.10. The average speedup compared to the minimum execution time for different modes for $DMatDMatADD$ benchmark on $Marvin$ and $Medusa$ node. The speedup was averaged over all matrix sizes for each specific number of cores.
Figure 6.11. The average speedup compared to creating as many tasks as number of cores (equal mode) for different modes for $D\text{Mat}D\text{Mat}D\text{Mat}ADD$ benchmark on $Marvin$ and $Medusa$ node. The speedup was averaged over all matrix sizes for each specific number of cores.
Figure 6.12. The average speedup compared to the minimum execution time for different modes for $DMatDMatDMatADD$ benchmark on $Marvin$ and $Medusa$ node. The speedup was averaged over all matrix sizes for each specific number of cores.

Based on the collected data, using splittable tasks helps us improve the performance achieved through creating as many tasks as the number of cores, in most of the cases, for all the guided, guided with threshold, adaptive, and adaptive with threshold modes, as shown in Table 6.2 and Table 6.3 for $DMatDMatADD$ benchmark and Table 6.4 and Table 6.5 for $DMatDMatDMatADD$ benchmark. Results of running the benchmark on 1 core was excluded from the calculations.
Table 6.2. Comparing the obtained results from splittable executor with minimum execution time for DM at ADD benchmark on Marvin node

<table>
<thead>
<tr>
<th>mode</th>
<th>Improvement from equal(%)</th>
<th>Within 10% of the minimum execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptive</td>
<td>86%</td>
<td>90%</td>
</tr>
<tr>
<td>adaptive with threshold</td>
<td>73%</td>
<td>84%</td>
</tr>
<tr>
<td>guided</td>
<td>94%</td>
<td>90%</td>
</tr>
<tr>
<td>guided with threshold</td>
<td>83%</td>
<td>87%</td>
</tr>
</tbody>
</table>

Table 6.3. Comparing the obtained results from splittable executor with minimum execution time for DM at ADD benchmark on Medusa node

<table>
<thead>
<tr>
<th>mode</th>
<th>Improvement from equal(%)</th>
<th>Within 10% of the minimum execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptive</td>
<td>83%</td>
<td>69%</td>
</tr>
<tr>
<td>adaptive with threshold</td>
<td>83%</td>
<td>66%</td>
</tr>
<tr>
<td>guided</td>
<td>93%</td>
<td>66%</td>
</tr>
<tr>
<td>guided with threshold</td>
<td>93%</td>
<td>68%</td>
</tr>
</tbody>
</table>

Table 6.4. Comparing the obtained results from splittable executor with minimum execution time for DM at ADD benchmark on Marvin node

<table>
<thead>
<tr>
<th>mode</th>
<th>Improvement from equal(%)</th>
<th>Within 10% of the minimum execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptive</td>
<td>91%</td>
<td>87%</td>
</tr>
<tr>
<td>adaptive with threshold</td>
<td>92%</td>
<td>87%</td>
</tr>
<tr>
<td>guided</td>
<td>97%</td>
<td>91%</td>
</tr>
<tr>
<td>guided with threshold</td>
<td>97%</td>
<td>91%</td>
</tr>
</tbody>
</table>

Table 6.5. Comparing the obtained results from splittable executor with minimum execution time for DM at ADD benchmark on Medusa node

<table>
<thead>
<tr>
<th>mode</th>
<th>Improvement from equal(%)</th>
<th>Within 10% of the minimum execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptive</td>
<td>84%</td>
<td>71%</td>
</tr>
<tr>
<td>adaptive with threshold</td>
<td>89%</td>
<td>70%</td>
</tr>
<tr>
<td>guided</td>
<td>93%</td>
<td>72%</td>
</tr>
<tr>
<td>guided with threshold</td>
<td>93%</td>
<td>72%</td>
</tr>
</tbody>
</table>
Chapter 7. Related Work

In this chapter we briefly study some of the related research efforts in this dissertation’s area of focus.

Loop scheduling techniques have been extensively studied by different researchers. In [52] 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 [52], 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[52]. They were able to improve data locality and scheduling overhead, while creating a more balanced workload.

Recently there have been efforts to use machine learning to predict the execution time or throughput in parallel applications. Khatami et al. [53] defined a set of features that could be extracted from the compiler including deepest loop level, total number of operations per iteration, number of floating and overall operations per iteration, and a set of runtime features including number of threads, and number of iterations. They developed a logistic regression model to predict the optimum chunk size of a parallel for-loop, from the options 0.1%, 1%, 10%, and 50% of the total number of iterations.

In an attempt similar to our work, Laberge et al. [54] use machine learning to estimate the optimum chunk size for evaluation of different array operations and array sizes in Blaze library. Defining the features as array size, number of cores, operation type, and machine architecture, the authors developed two random forest models. One, as a classification model, would predict the optimum chunk size; while the other one, as a regression model, predicts the achievable performance. The optimum chunk size, would then be identified through finding the predicted performance for each chunk size and selecting the chunk size that would result in the maximum performance. In the classification model, the random forest model was modified to use the difference in real and predicted values for throughput as the loss func-
tion, which was helpful in improving the achieved accuracy. The authors also integrated their model into a fork of Blaze library in order to use the model predictions to set the chunk size at runtime. The drawbacks of this method were that the predicted chunk size was limited to the set 1, 2, 3, ..., 10 which is not always a good choice, and also their method required collecting a considerable amount of data from different architectures in order to be generalized their model [54].

As another field to use machine learning, [55] 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 [56] 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 types 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 $L_1$ data cache miss rate. The authors 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
In [57], 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[57]. Their solution relies completely on the compiler and the run-time, and eliminates the need for manual support.

In [58], the authors try to find the optimum task granularity to keep the scheduling overheads and resource utilization in balance in an asynchronous many-task runtime system. They utilize a system emulator to study the effect of task granularity in system performance, and conclude that the optimum task size with representative schedulers is between $1.2 \times 10^4$ and $10 \times 10^4$ cycles on a system with up to 1024 cores. They also provide an automatic algorithm to aggregate tasks into larger tasks based on the concluded task granularity in order to improve the performance.

Grubel et al. [20] also studies the effect of the task size on performance of the HPX applications. They introduce some performance metrics in order to help them identify the optimum grain size which could be very helpful for runtime adaptivity.

In [59], the authors use thresholds to decide on whether to inline a task at runtime. The imposed threshold for task inlining on a specific architecture then converts into the problem to what portion of the execution time of the task should be spent for scheduling the task, so that it would be worth to be executed as a separate task. This is in compliance with our findings in this paper for $\lambda_b$, as shown in (5.14), where we suggest in order to land in the flat region of the execution time versus grain size graph, the ratio of the grain size over the scheduling overhead of one task on one core should be greater than a threshold.

Tzannes et al.[49] proposed Lazy Binary Splitting(LBS) that adjusts the available parallelism based on the inferred load at run-time, to avoid the unnecessary parallelism overhead. They work improve upon Intel’s TBB(threading Building Blocks) [50] implementation of E-
ger Binary Splitting (EBS), by first postponing splitting to when parallelism is available, and second applying a runtime adaptive threshold to stop splitting.
Chapter 8. Conclusions

In this dissertation, our focus is on using task granularity for runtime adaptivity in asynchronous many-task runtime systems, in order to improve the performance of multi-threaded linear algebra libraries. We selected HPX, the C++ Standard library for parallelism and concurrency, and Blaze, a high performance C++ math library, for this purpose.

At the first step, using polynomial regression, we were able to provide an estimate for the range of grain size for maximum throughput with a specific number cores. But this model was not physical, so in the next step we provided an analytical model for execution time of balanced parallel for-loops in terms of grain size and number of cores for the purpose of locating the range of grain sizes for minimum execution time. Some of the contributing factors were ignored to simplify the model since it wouldn't contribute to the located optimum range, which was our main goal.

This model depends on two mostly architecture-specific parameters, the overhead of scheduling tasks($\alpha$) and contention($\sigma$). A simple parallel for-loop benchmark was developed to validate the proposed model. We suggest that we can utilize this benchmark to find the model parameters on a specific architecture by collecting a small set of data. Once the parameters are identified, the analytical model could be used to estimate the optimum range of grain sizes for any other balanced parallel for-loop application ran on the same machine.

The proposed approach was then applied to Blaze, a high performance C++ math library. By defining grain size as the number of floating point operations we implemented a function API to estimate the number of floating point operations for an expression at compile time. This allows us to offer a runtime and compile-time solution to find the optimum range of grain size on a specific architecture while evaluating a specific expression. The obtained results showed that the identified chunk size had improved the performance from the equal case(where the same number of tasks as the number of cores are created) in more than half of the cases.

Having tested and validated our proposed model, we extended a previously implemented
algorithm to apply the information extracted through the model in order to avoid the unnecessary scheduling overhead. We implemented a splittable executor with two modes, guided and adaptive, in HPX. This executor would launch splittable tasks instead of a regular task, which would generate more splittable tasks if enough parallelism is available. This helps to avoid both unnecessary scheduling overhead and under-utilization of our resources. The results show an improvement from the equal case, and also reaching an execution time within 10% of the minimum execution time in majority of the cases. In order to adjust the provided solution to the running application, we used the optimum range identified using the analytical model as a threshold to stop splitting. The task splitting would stop if the grain size of the created task gets smaller than the lower-bound of the identified range. This helped decreasing the scheduling overhead of tasks for smaller problem sizes.
References


Listing A.1: Implementation of splittable task in HPX

```cpp
// Copyright (c) 2007–2020 Hartmut Kaiser
// Copyright (c) 2020 Shahrzad Shirzad
// SPDX-License-Identifier: BSL-1.0
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#ifndef HPX_SPLITTABLE_TASK_HPP
#define HPX_SPLITTABLE_TASK_HPP

#include <hpx/execution/detail/post_policy_dispatch.hpp>
#include <hpx/modules/datastructures.hpp>
#include <hpx/modules/execution.hpp>
#include <hpx/modules/runtime_local.hpp>
#include <hpx/modules/synchronization.hpp>
#include <hpx/threading_base/thread_description.hpp>
#include <cstdint>
#include <utility>
namespace hpx { namespace parallel { namespace execution {

enum class splittable_mode {
  all = 0,
  idle = 1,
};

inline char const* const get_splittable_mode_name(splittable_mode mode) {
  static constexpr char const* const splittable_mode_names[] = {
    "all", "idle"};
  return splittable_mode_names[static_cast<std::size_t>(mode)];
}

namespace hpx { namespace parallel { namespace execution {

enum class splittable_mode {
  all = 0,
  idle = 1,
};

inline char const* const get_splittable_mode_name(splittable_mode mode) {
  static constexpr char const* const splittable_mode_names[] = {
    "all", "idle"};
  return splittable_mode_names[static_cast<std::size_t>(mode)];
}

template <typename Executor, typename F>
struct splittable_task
{
  template <typename F_, typename Shape>
  splittable_task(Executor& exec, F_& f, Shape const& elem,
                   std::size_t num_free, splittable_mode split_type,
                   std::size_t min_task_size) :
                   start_(hpx::util::get<0>(elem)),
                   stop_(hpx::util::get<1>(elem)),
                   index_(hpx::util::get<2>(elem)),
                   num_free_(num_free),
                   f_(std::forward<F_>(f)),
                   exec_(exec),
                   split_type_(split_type),
                   min_task_size_(min_task_size)
  { }

  void operator() (hpx::latch* outer_latch = nullptr)
  {
    if (split_type_ == splittable_mode::idle)
      { call_idle(); }
    else
      { call(); }

    // notify outer waiting task
    if (outer_latch != nullptr)
      { outer_latch->count_down(1); }
  }

private:
  void call_idle()
  { auto mask = hpx::threads::get_idle_core_mask();
    num_free_ = hpx::threads::count(mask);
  }

}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}
std::size_t task_size = std::ceil(float(stop_ - start_) / (num_free_ + 1));
std::size_t remainder = stop_ - start_ - task_size;

if ((num_free_ != 0 && task_size > min_task_size_) && remainder > 0) {
    // split the current task, create one for the first identified idle core
    for (std::size_t i = 0, j = 0; i != 1; ++j)
        if (hpx::threads::test(mask, j)) {
            // pass schedule hint to place new task on empty core
            using policy = hpx::launch::async_policy;
            detail::post_policy_dispatch<policy>::call(policy(),
                desc, exec_.get_priority(), exec_.get_stacksize(),
                hpx::threads::thread_schedule_hint(std::int16_t(j)),
                splittable_task(exec_, f_,
                    hpx::util::make_tuple(
                        stop_ - remainder, stop_, index_ + i + 1),
                        num_free_, split_type_, min_task_size_),
                &l);
            stop_ = stop_ - remainder;
            ++i;
        }
}

hpx::latch 1(2);

std::size_t task_size = std::ceil(float(stop_ - start_) / (num_free_ + 1));
std::size_t remainder = stop_ - start_ - task_size;

if ((num_free_ != 0 && task_size > min_task_size_) && remainder > 0) {
    // split the current task
    exec_.post(splittable_task(exec_, f_,
        hpx::util::make_tuple(
            stop_ - remainder, stop_, index_ + 1),
            num_free_ - 1, split_type_, min_task_size_),
        &l);
    stop_ = stop_ - remainder;
}
else {
    l.count_down(1);
}

f_(hpx::util::make_tuple(start_, stop_ - start_, index_));

// wait for task scheduled above
l.arrive_and_wait(1);

private:
    std::size_t start_;
    std::size_t stop_;
    std::size_t index_;
    std::size_t num_free_;
    F f_;
    Executor& exec_;
    splittable_mode split_type_;
template <typename Executor, typename F, typename Shape>
splittable_task<Executor, typename std::decay<F>::type>
make_splittable_task(Executor& exec, F&& f, Shape const& s,
splittable_mode split_type, std::size_t min_task_size)
{
  std::size_t num_free = hpx::get_os_thread_count() - 1;
  return splittable_task<Executor, typename std::decay<F>::type>(
    exec, std::forward<F>(f), s, num_free, split_type, min_task_size);
}
Listing A.2: Implementation of splittable executor in HPX

// Copyright (c) 2007-2020 Hartmut Kaiser
// Copyright (c) 2020 Shahrzad Shirzad
// SPDX-License-Identifier: BSL-1.0
// Distributed under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

#include <hpx/config.hpp>
#include <hpx/execution/trait/is_executor.hpp>
#include <hpx/execution/detail/splittable_task.hpp>
#include <hpx/include/async.hpp>
#include <hpx/modules/executors.hpp>
#include <hpx/modules/iterator_support.hpp>
#include <hpx/modules/serialization.hpp>
#include <hpx/modules/timing.hpp>
#include <algorithm>
#include <cstdint>
#include <type_traits>

namespace hpx { namespace parallel { namespace execution {

/// Loop iterations are divided into pieces and then assigned to threads.
/// The number of loop iterations combined is determined based on
/// measurements of how long the execution of 1% of the overall number of
/// iterations takes.
/// This executor parameters type makes sure that as many loop iterations
/// are combined as necessary to run for the amount of time specified.

struct splittable_executor : parallel_policy_executor<hpx::launch::async_policy>
{
    using base_type = parallel_policy_executor<hpx::launch::async_policy>;

public:
    /// Construct an splittable_executor parameters object
    /// Note Default constructed splittable_executor parameters object
    /// types will use 80 microseconds as the minimal time for which
    /// any of the scheduled chunks should run.
    splittable_executor() : split_type_(splittable_mode::all)
    , min_task_size_(0)
    {
    }

    /// Construct a splittable_executor parameters object
    /// param rel_time [in] The time duration to use as the minimum
    /// to decide how many loop iterations should be
    /// combined.
    splittable_executor(splittable_mode split_type)
    : split_type_(split_type)
    , min_task_size_(0)
    {
        if (split_type_ != splittable_mode::all &&
            split_type_ != splittable_mode::idle &&
            split_type_ != splittable_mode::idle_mask &&
            split_type_ != splittable_mode::all_multiple_tasks)
        {
            HPX_THROW_EXCEPTION(hpx::bad_parameter,
                "splittable_executor::splittable_executor",
                "unknown type. type should be either all, idle, "
                "idle_mask, or all_multiple_tasks");
        }
    }

    splittable_executor(splittable_mode split_type, std::size_t min_task_size)
    : split_type_(split_type)
    , min_task_size_(min_task_size)
    {
        if (split_type_ != splittable_mode::all &&
            split_type_ != splittable_mode::idle &&
            split_type_ != splittable_mode::idle_mask &&
            split_type_ != splittable_mode::all_multiple_tasks)
        {
            HPX_THROW_EXCEPTION(hpx::bad_parameter,
                "splittable_executor::splittable_executor",
                "unknown type. type should be either all, idle, "
                "idle_mask, or all_multiple_tasks");
        }
    }

}}};
HPX_THROW_EXCEPTION(hpx::bad_parameter,
   "splittable_executor::splittable_executor",
   "unknown type, type should be either all, idle, "
   "idle_mask, all_multiple_tasks");
}

HPX_FORCEINLINE static std::size_t processing_units_count()
{
    return hpx::get_os_thread_count();
}

template <typename Parameters, typename F>
static std::size_t get_chunk_size(
    Parameters&&, F&&, std::size_t, std::size_t count)
{
    return count;
}

template <typename F, typename S, typename... Ts>
std::vector<hpx::future<typename detail::bulk_function_result<F, S, Ts...>::type>>
bulk_async_execute(F&& f, S const& shape, Ts&&... ts)
{
    std::vector<hpx::future<typename detail::bulk_function_result<F, S, Ts...>::type>>
    results;
    results.reserve(hpx::util::size(shape));
    for (auto const& elem : shape)
    {
        results.push_back(hpx::async(make_splittable_task(
            static_cast<base_type&>(*this), std::forward<F>(f), elem,
            split_type_, min_task_size_));
    }
    return results;
}

private:
friend class hpx::serialization::access;
splittable_mode split_type_;
std::size_t min_task_size_;}

// workaround for older HPX versions
template <typename Param, typename Executor, typename F>
std::size_t get_chunk_size(Param&& param, Executor&& exec, F&& f,
    std::size_t core, std::size_t count)
{
    return count;
}

template <typename Param, typename Executor>
HPX_FORCEINLINE static std::size_t processing_units_count(
    Param&& params, Executor&&& exec)
{
    return hpx::get_os_thread_count();
}

// namespace hpx::parallel::execution
Vita

Shahrzad Shirzad got her Bachelor of Science degree in Electrical Engineering from Sharif University of Technology in Iran in 2006, and her Master's degree in Biomedical Engineering from K.N. Toosi University of Technology in Iran in 2009. After moving to Baton Rouge in 2013 she started her PhD studies in Computer Engineering. After getting her Master's degree in Engineering Science, IT concentration, she joined the Stellar group at Center for Computation and Technology at LSU, where she mostly worked on Phylanx, a distributed array processing toolkit, under Dr. Kaiser's supervision.

A.1. Publications


