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

PhD Proposal

Shahrzad Shirzad

December 4, 2019

Division of Computer Science and Engineering
School of Electrical Engineering and Computer Science
Louisiana State University

Center for Computation & Technology

STE||AR GROUP

Outline

Objective

Objective

- A compile-time and runtime solution to optimize the performance of a linear algebra library based on a specific application
- These parameters could be: Machine architecture, number of cores to run the program on, the expression to be evaluated: the type of operations, the number of matrices involved, the matrix sizes.

Introduction

Introduction

- AMT(Asynchronous Many-task) model and runtime systems
    - HPX, Charm++, Uintah, Legion
- Linear algebra libraries
    - Scalapack, ATLAS, SPIRAL

Background

HPX: Execution Model

- Parallex
  - SLOW

Blaze C++ Library

Linear Algebra Library based on Smart Expression Templates

- Expression Templates:
  - Creates a parse tree of the expression at compile time and postpone the actual evaluation to when the expression is assigned to a target
- Smart:
  - Creation of intermediate temporaries when needed
  - Integration with highly optimized compute kernels
  - Selecting optimal evaluation method automatically for compound expressions

Blaze: Parallelization

Depending on the operation and the size of operands, the assignment could be parallelized through four different backends

- HPX
- OpenMP
- C++ threads
- Boost

Blaze: Backend Implementation

In the current implementation, the work is equally divided between the cores at compile time.

- Parallel for loop

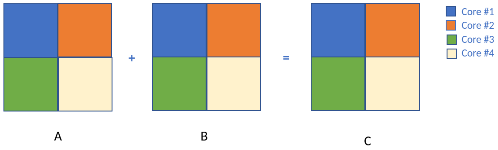


Figure 1: An example of how C=A+B is performed in parallel in Blaze with 4 cores

Loop Scheduling

Chunk size: Number of loop iterations executed by one thread

- Static
- Dynamic
- Other methods including

HPX Tasks

Task Granularity

Grain size: The amount of work performed by one HPX thread

- What causes performance degradation?
  - Overheads
  - Starvation

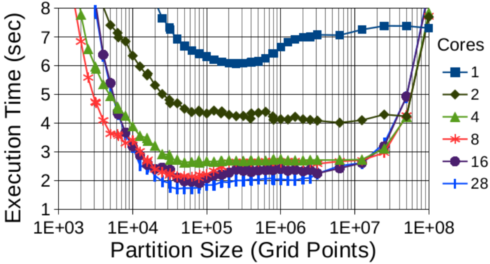


Figure 2: The effect of task size on execution time for Stencil application[1]

[1] Grubel, Patricia, et al. "The performance implication of task size for applications on the hpx runtime system." 2015 IEEE International Conference on Cluster Computing. IEEE, 2015.

Modeling Performance

- Amdahl's Law

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

- Universal Scalibility Law

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

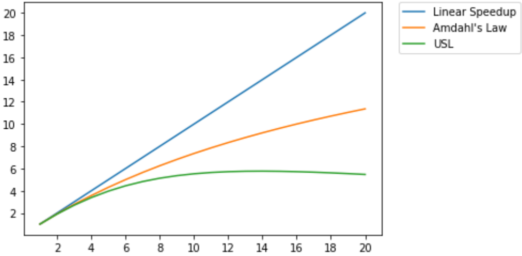


Figure 3: An example of the achievable speedup based on Amdahl's law and USL compared to the ideal linear speedup where $\sigma = 0.04$ and $\kappa = 0.005$.

Modeling Performance: Other Models

- Quadratic model

$$S(p) = p - \gamma p(p-1)$$

- Exponential model

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

- Geometric model

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

Objective

Dynamically divide the work among the cores based on number of cores, matrix size, complexity of the operation, machine architecture. For this purpose two parameters have been introduced:

- block_size: at each loop iteration the assignment is performed on one block
- chunk_size: the number of loop iterations included in one task

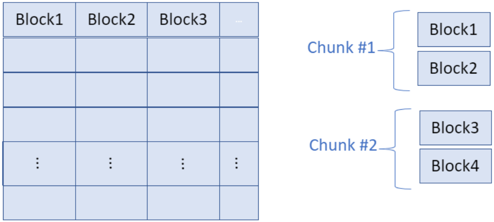


Figure 4: An example of blocking and creating chunks for chunk_size = 2

Method

Data Collection

- Starting from DMATDMATADD benchmark: $C = A + B$

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

Table 1: List of different values used for each variable for running the *DMATDMATADD* benchmark

Data

- For simplicity we look at one matrix size at a time

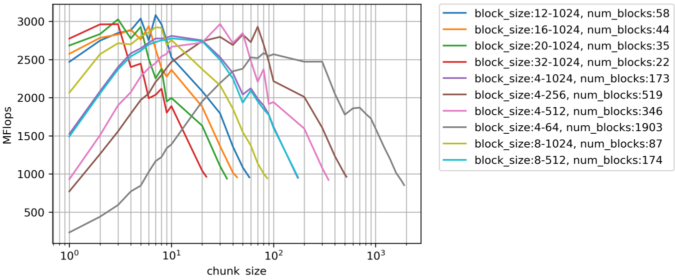


Figure 5: The results obtained from running *DMATDMATADD* benchmark through Blazemark for matrix sizes from 690×690 with different combinations of block size and chunk size on 4 cores

Throughput vs. Grain Size

Grain size: The amount of work performed by one HPX thread

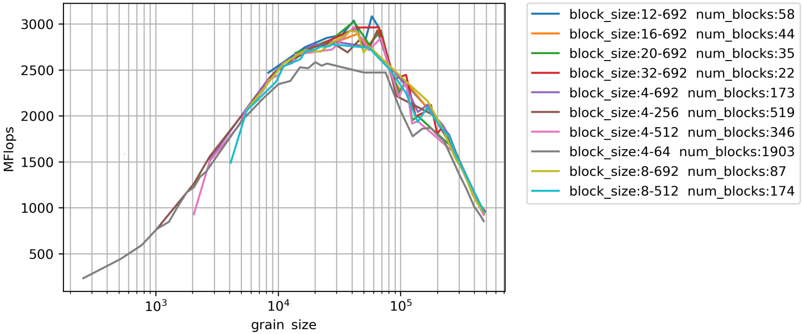
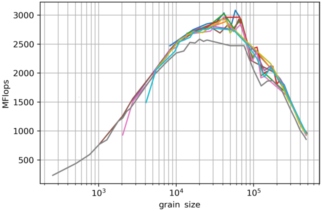


Figure 6: The results obtained from running *DMATDMATADD* benchmark through Blazemark for matrix size 690×690 on 4 cores.

Observation



- 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.

Can we model the relationship between the throughput and the grain size?

Modeling

1- Quadratic Model

2- Bathtub Model

Method: Quadratic Model

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.

- Used a second order polynomial to model the relationship between the throughput and the grain size when number of cores is fixed.

$$P = ag^2 + bg + c$$

- Divide the data into training(60%) and test(40%)

Method: Modeling Performance based on Grain Size

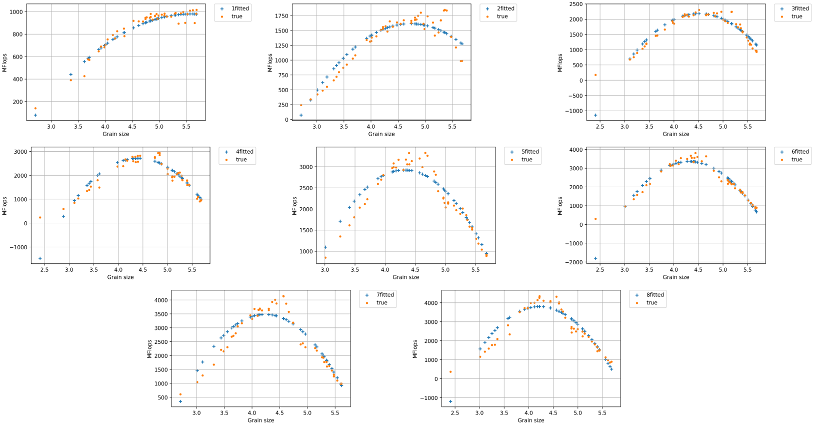


Figure 7: The results of fitting the throughput vs grain size data into a 2d polynomial for *DMATDMATADD* benchmark for matrix size 690×690 with different number of cores on the test data set (a) 1 core, (b) 2 cores, (c) 3 cores, (d) 4 cores, (e) 5 cores, (f) 6 cores, (g) 7 cores, (h) 8 cores.

Method: Modeling Performance based on Grain Size

$$Relative_error = \frac{1}{n}\sum_{i=1}^{n} 1 - p_i/t_i$$

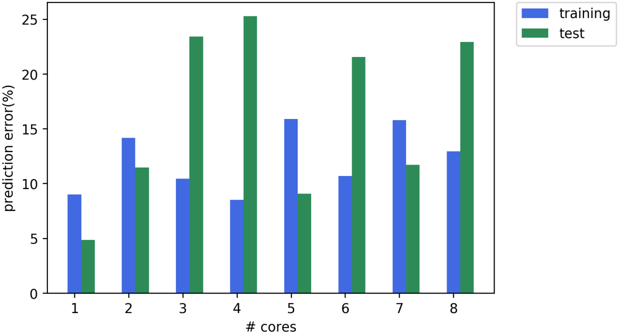


Figure 8: The training and test error for fitting data obtained from the *DMATDMATADD* benchmark for matrix size 690 × 690 against different number of cores cores.

Method: Modeling Performance based on Grain Size

Can we somehow integrate number of cores into the model?

Method: Modeling Performance based on Grain Size

- For $P = ag^2 + bg + c$, see how a, b, and c change with the number of cores
- Model the relationship with a 3rd degree polynomial

$$a, b, c = a_0 N^3 + a_1 N^2 + a_2 N + a_3$$

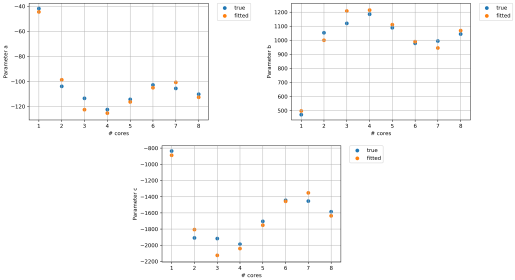


Figure 9: Fitting the parameters of the quadratic function with a 3rd degree polynomial from the *DMATDMATADD* benchmark for matrix size 690×690 against different number of cores.

Method: Modeling Performance based on Grain Size

The final model:

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

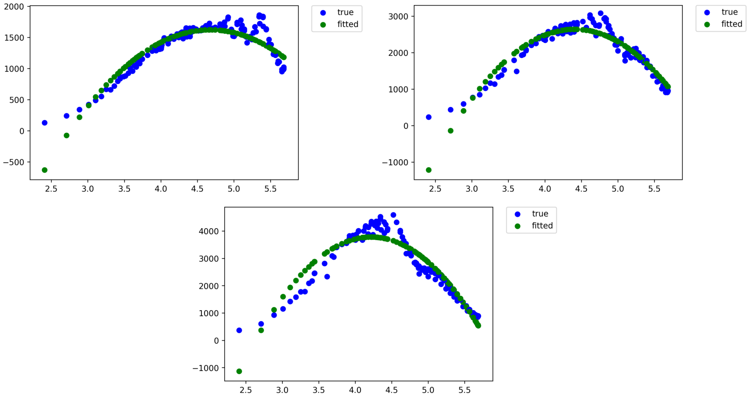
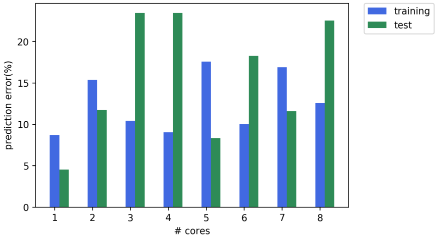


Figure 10: matrix size 690 × 690 for (a) 2 core, (b) 4 cores, (c) 8 cores.

Method: Modeling Performance based on Grain Size



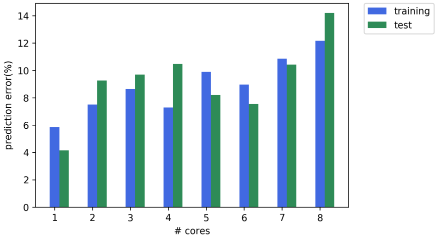


Figure 11: (a) All the data points are include in calculation of error, (b) the leftmost sample was removed from error calculation.

Method: Finding the Grain Size Range for Maximum Performance

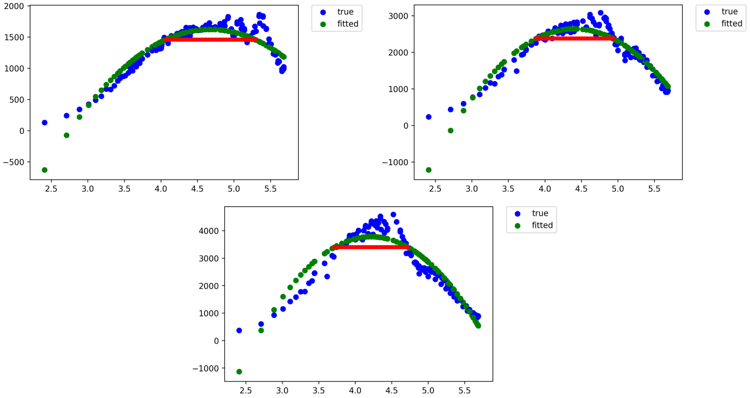


Figure 12: The range of grain size (shown as the red line) that leads to a performance within 10% of the maximum performance for (a) 2 cores, (b) 4 cores and (b) 8 cores.

Method: Finding the Grain Size Range for Maximum Performance

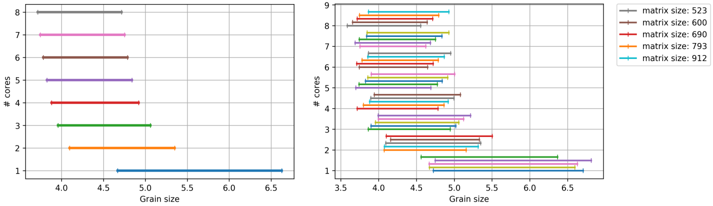


Figure 13: The range of grain size within 10% of the maximum performance of the fitted polynomial function for *DMATDMATADD* benchmark for different number of cores for (a) matrix size 690×690 (b)matrix size 523×523 to 912×912.

Method: Finding the Grain Size Range for Maximum Performance

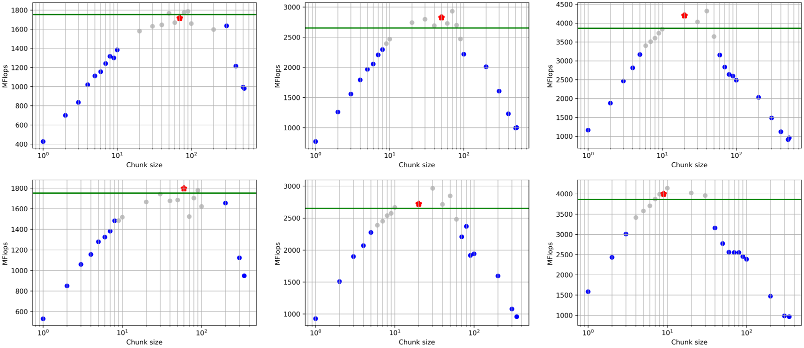


Figure 14: matrix size 690 × 690 with block size of 4 × 256 on (a) 2 cores, (b) 4 cores, and (c) 8 cores, and block size of 4 × 512 on (d) 2 cores, (e) 4 cores, and (f) 8 cores.

Method: Bathtub Model

Creating a analytic model for execution time based on grain size

Method: Bathtub Model

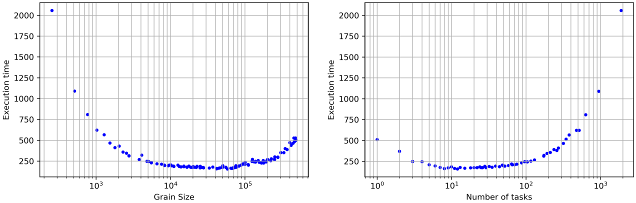


Figure 15: (a)The execution time vs. grain size graph, and (b) execution time vs. number fo tasks graph for *DMATDMATADD* benchmark for matrix size 690×690 ran on 4 cores.

Modeling Execution Time based on Grain Size

- Overheads of creating tasks
- Starvation

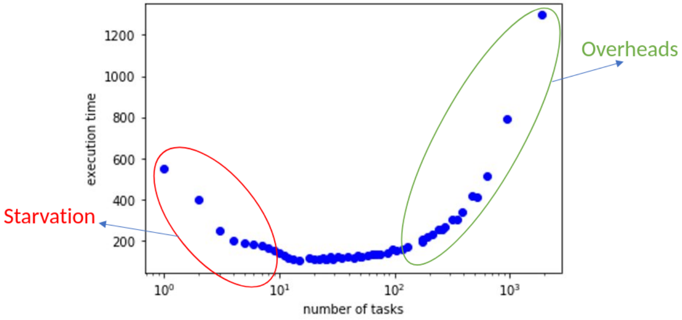


Figure 16: Results of running the *DMATDMATADD* benchmark on 8 cores matrix size 690(time unit is microseconds)

Modeling Execution Time based on Grain Size

$$t = \begin{cases} \alpha + \dfrac{t_s}{n_t} + \gamma & \text{if } n_t < N \\ \dfrac{\alpha n_t + t_s}{N} + \gamma & \text{otherwise} \end{cases}$$

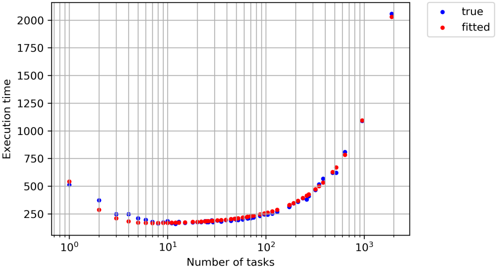
n_t: number of tasks

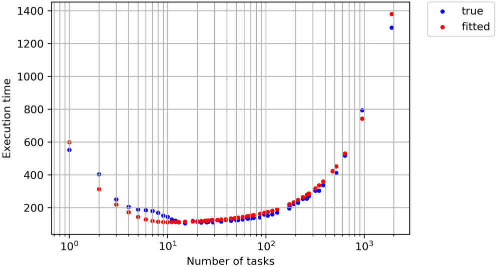
N: number of cores t_s: sequential execution time γ: parallelization constant Softplus function:

$$f(x) = Ln(1 + e^x)$$

Modeling Execution Time based on Grain Size

- Fixed matrix size, and number of cores
- Training set and test set (%60, %40)





Modeling Execution Time based on Grain Size

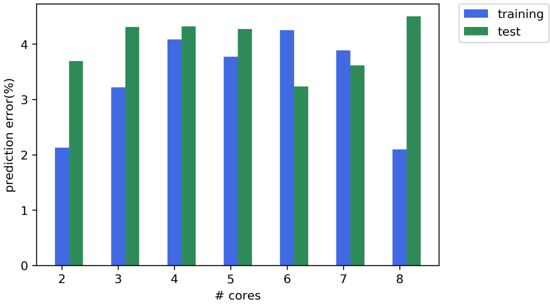


Figure 18: The error in fitting execution time with the bathtub formula for *DMATDMATADD* benchmark for matrix size 690×690 with different number of cores.

Modeling Execution Time based on Grain Size

How do α, t_s, and γ change with number of cores?

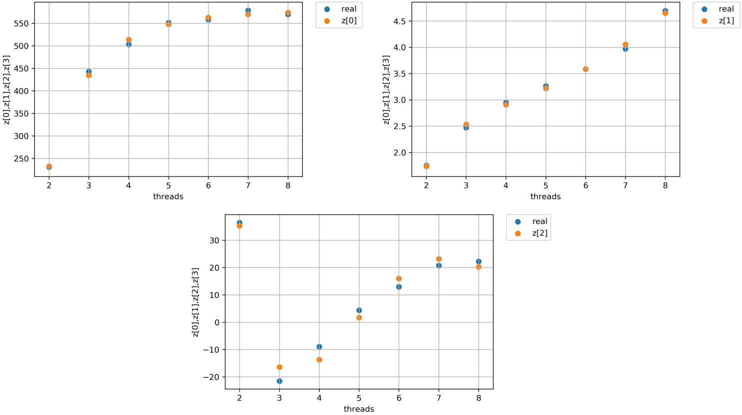


Figure 19: Fitting the three parameters (a)α, (b)t_s, and (c)γ for *DMATDMATADD* benchmark for matrix size 690×690.

Modeling Execution Time based on Grain Size

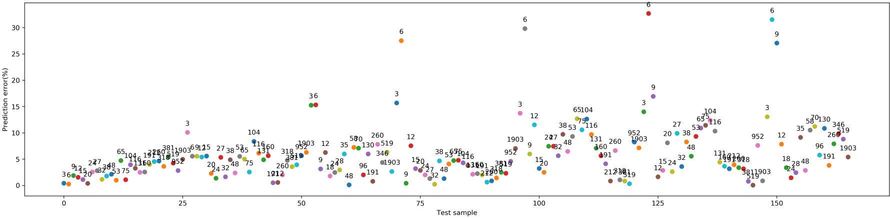


Figure 20: The error in fitting execution time with the bathtub formula for *DMATDMATADD* benchmark for matrix size 690×690 with different number of cores.

Modeling Execution Time based on Grain Size

The problem with the current model is that with this formula we know that the minimum occurs at $n_t = N$, but that's not usually the case. This inspires us to check for a missing factor. This model still needs to be studied. The estimate that we have is for execution time, which is in our experiments

Setup

Blazemark

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

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

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

Figure 21: An example of results obtained from Blazemark

Configuration

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

Table 2: Specifications of the Marvin node from Rostam cluster at CCT.

Library	Version
HPX	1.3.0
Blaze	3.5

Table 3: Specifications of the libraries used to run our experiments.

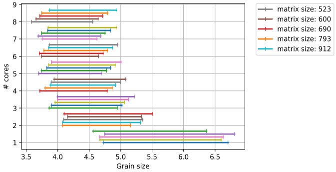
Related Work

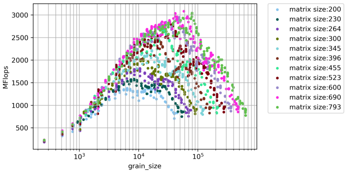
HPX

- Zahra
- Gabriel
- Peter thoman

Proposed Study

Proposed Study





- Studying the bathtub model
- Generalization for matrix size
- Generalization for complex expressions
- Generalization for different architectures

Thank you!