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

Scientific Computing Around Louisiana (SCALA)

Shahrzad Shirzad, Dr. Hartmut Kaiser

Feburary 8, 2020

Louisiana State University

Objective

A compile-time and runtime solution to optimize the performance of a linear algebra library based on

- Machine architecture
- Number of cores to run the program on
- Expression to be evaluated
    - Type of operations
    - Number of matrices involved
    - Matrix sizes

Introduction

- Current programming models would not be able to keep up with the advances toward exascale computing
  - More complex machine architectures, deeper memory hierarchies, heterogeneous nodes, complicated networks
- AMT(Asynchronous Many-Task) model and runtime systems
  - Examples: HPX, Charm++, Legion

Introduction

- Performance of HPC applications heavily rely on the linear algebra library they are using.
- Linear algebra libraries
  - BLAS(Basic Linear Algebra Subprograms) are the fundamental routines for basic vector and matrix operations.
  - Examples: ScaLAPACK, ATLAS, SPIRAL
- Our motivation:
  - Phylanx, a platform to run your python code in parallel and distributed with machine learning as the target application.

Background: HPX

- HPX is a general purpose C++ runtime system for parallel and distributed applications of any scale.
- Fine-grained parallelism instead of heavyweight threads.
- Four major factors for performance degradation: SLOW
  - Starvation
  - Latency
  - Overheads
  - Waiting for contention resolution

Blaze is a high performance C++ 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:
  - Integration with highly optimized compute kernels
  - Selecting optimal evaluation method automatically for compound expressions

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

- HPX, OpenMP, C++ threads, Boost

In the current implementation, the work is equally divided between the cores at compile time.
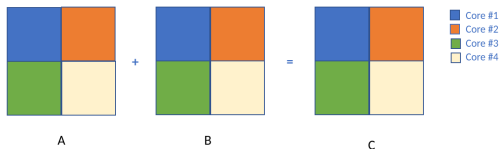
- Parallel for loop

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

Background: Task Granularity

Grain size: The amount of work performed by one task
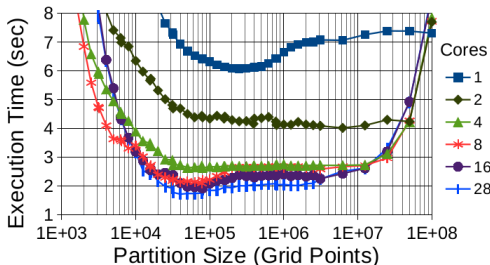
- What causes performance degradation?
    - Overheads
    - Starvation

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.

Method: 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
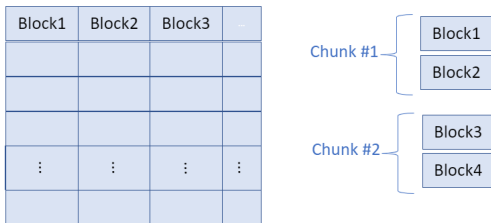- chunk_size: the number of loop iterations included in one task

Figure 3: An example of blocking a matrix 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

- For simplicity we look at each matrix size individually, one number
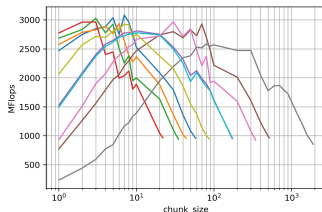 of core at a time
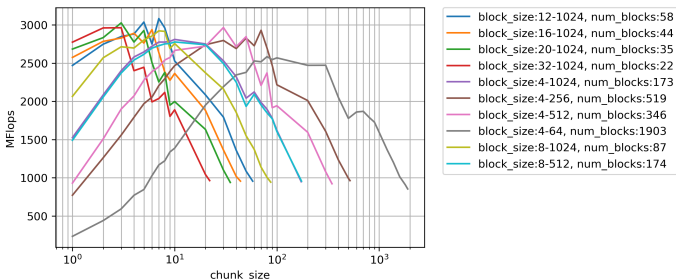
Figure 4: The results obtained from running *DMATDMATADD* benchmark for matrix sizes
690×690 with different combinations of block size and chunk size on 4 cores

Method: 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.

Method: Observation
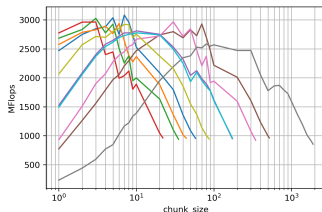
- For each selected block size, there is a range of chunk sizes that gives us the best performance.
- Except for some uncommon cases, no matter which block size we choose, we are able to achieve the maximum performance if we select the right chunk size.
- Instead of looking at block size and chunk size individually, look at grain size.

Grain size: The number of floating point operations performed by one thread

- Grain size represents the complexity of the expression
- For *DMATDMATADD*, with *block_size* $= r \times c$ and *chunk_size* $= ch$
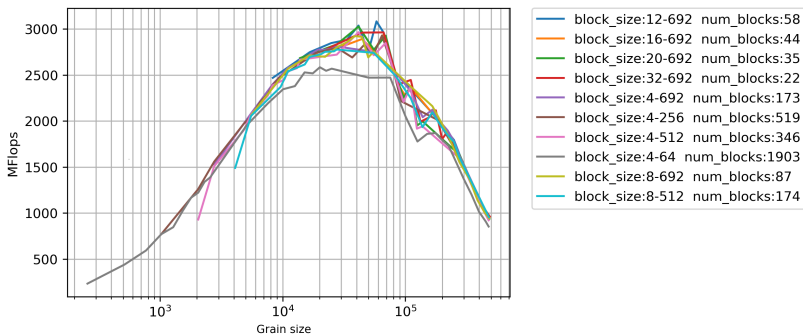 Grain_size$= r \times c \times ch$
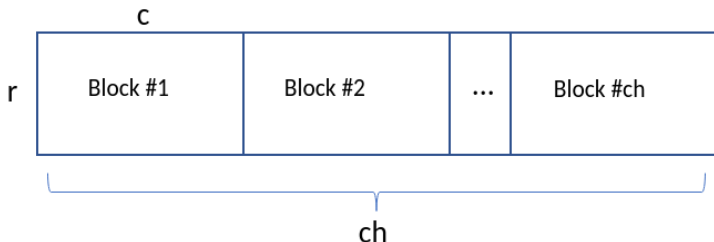
Method: Throughput vs. Grain Size

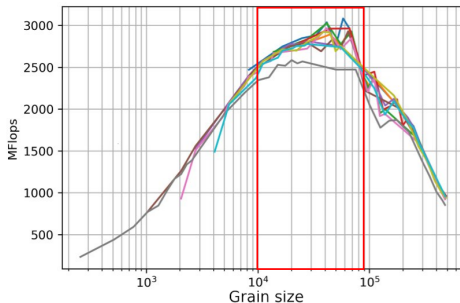**Figure 5:** The results obtained from running *DMATDMATADD* benchmark through Blazemark for matrix size 690×690 on 4 cores.

The range of grain size for maximum performance

Figure 6: The results obtained from running *DMATDMATADD* benchmark through Blazemark for matrix size 690×690 on 4 cores.
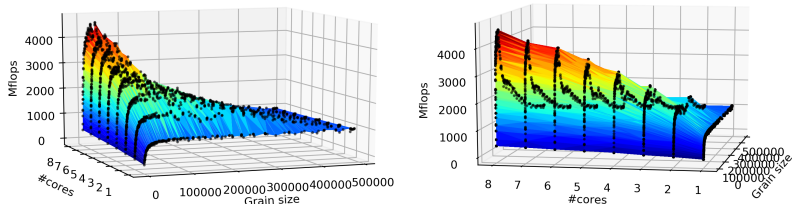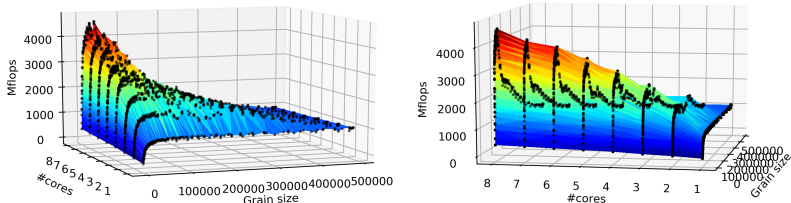
Figure 7: The results obtained from running *DMATDMATADD* benchmark through Blazemark for matrix size 690×690 based on grain size and number of cores.

Figure 7: The results obtained from running *DMATDMATADD* benchmark through Blazemark for matrix size 690×690 based on grain size and number of cores.

Can we model the relationship between the **throughput** and the **grain size** and the **number of cores**?
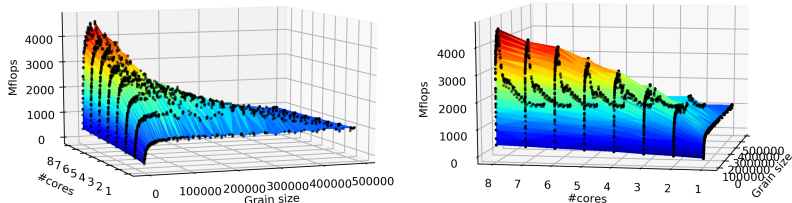
Figure 7: The results obtained from running *DMATDMATADD* benchmark through Blazemark for matrix size 690×690 based on grain size and number of cores.

Can we model the relationship between the **throughput** and the **grain size** and the **number of cores**?

- First we try to model the relationship between throughput and grain size.
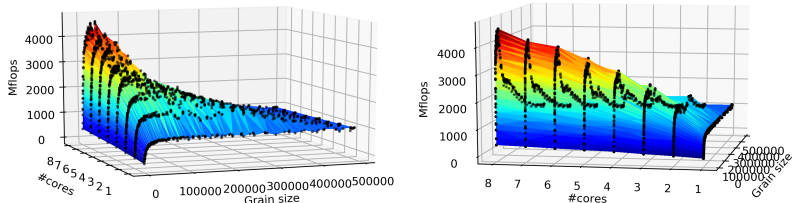
Figure 7: The results obtained from running *DMATDMATADD* benchmark through Blazemark for matrix size 690×690 based on grain size and number of cores.

Can we model the relationship between the **throughput** and the **grain size** and the **number of cores**?

- First we try to model the relationship between throughput and grain size.
  - Polynomial 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.
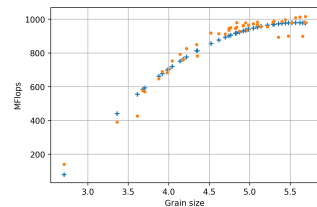
- Used a second order polynomial to model the relationship between the throughput and the grain size when number of cores is fixed.
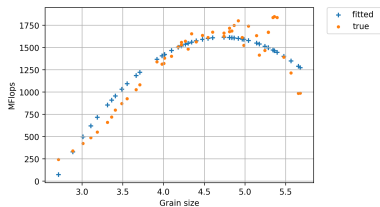
$$P = ag^2 + bg + c$$
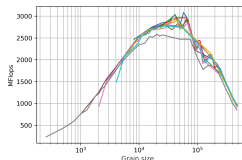
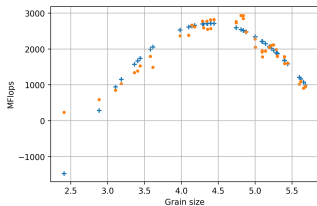- Divide the data into training(60%) and test(40%)

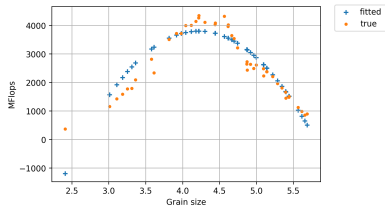# Method: Modeling Performance based on Grain Size

(a) 1 core

(b) 2 cores

(c) 4 cores

(d) 8 cores

Figure 8: 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.

Method: Modeling Performance based on Grain Size

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

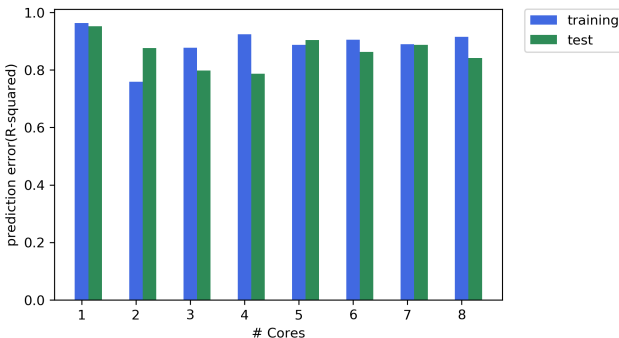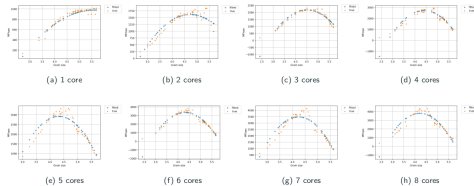**Figure 9:** The training and test error for fitting data obtained from the *DMATDMATADD* benchmark for matrix size 690 × 690 against different number of cores cores.

Method: Modeling Performance based on Grain Size

(a) 1 core   (b) 2 cores   (c) 3 cores   (d) 4 cores

(e) 5 cores   (f) 6 cores   (g) 7 cores   (h) 8 cores

- We have developed a model for each number of cores:1, 2,.., 8
  $$P = ag^2 + bg + c$$

Method: Modeling Performance based on Grain Size

(a) 1 core   (b) 2 cores   (c) 3 cores   (d) 4 cores

(e) 5 cores   (f) 6 cores   (g) 7 cores   (h) 8 cores
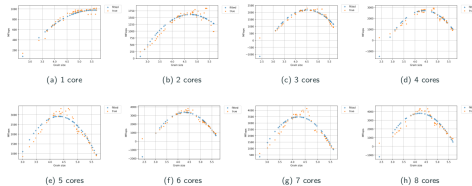
- We have developed a model for each number of cores: 1, 2,.., 8
  $$P = ag^2 + bg + c$$

- Can we somehow integrate number of cores into the model?

Method: Modeling Performance based on Grain Size

- For $P = ag^2 + bg + c$, how do a, b, and c change with the number of cores?
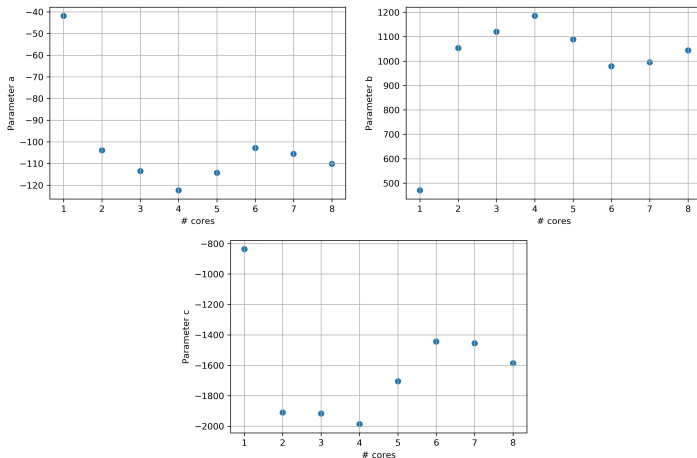

Figure 10: The parameters of the polynomial fit from the *DMATDMATADD* benchmark for matrix size 1587 × 1587 against different number of cores.

Method: Modeling Performance based on Grain Size

- Model the relationship with a 3rd degree polynomial

$$a = a_0 N^3 + a_1 N^2 + a_2 N + a_3, \quad b = b_0 N^3 + b_1 N^2 + b_2 N + b_3,$$

$$c = c_0 N^3 + c_1 N^2 + c_2 N + c_3$$

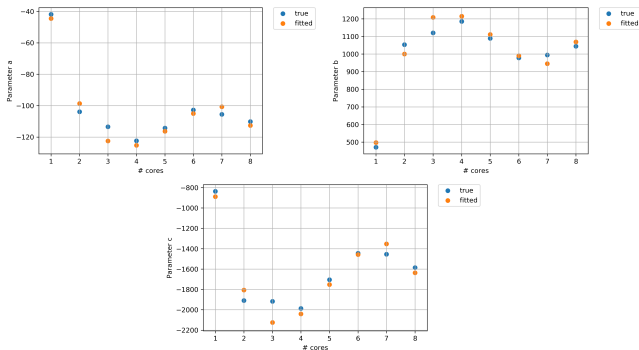**Figure 11:** Fitting the parameters of the polynomial function with a 3rd degree polynomial from the *DMATDMATADD* benchmark for matrix size 1587 × 1587 against different number of cores.

The final model:

$$P = \sum_{i=0}^{2} \sum_{j=0}^{3} a_{ij} g^i N^j$$
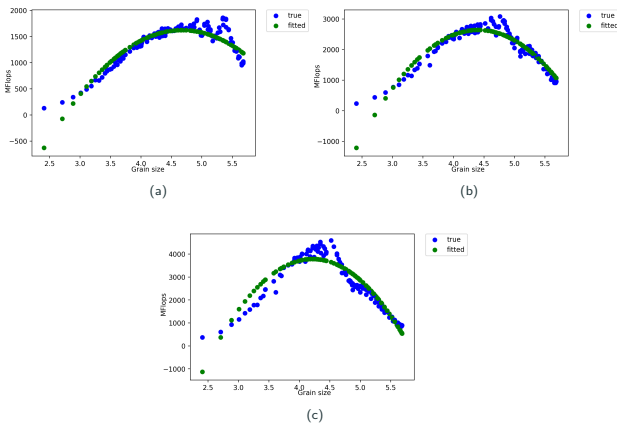
Figure 12: matrix size 690 × 690 for (a) 2 core, (b) 4 cores, (c) 8 cores.
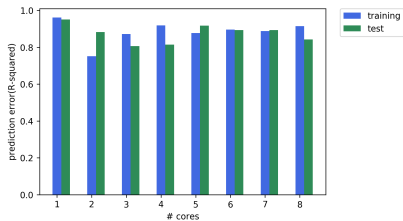
Figure 13: All the data points are included in calculation of error, R_squared error.

Method: Finding the Grain Size Range for Maximum Performance
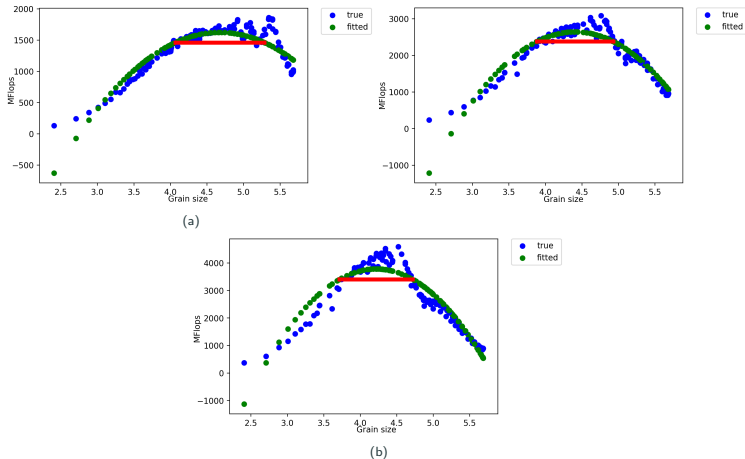
Figure 14: The range of grain size (shown as the red line) that leads to a performance within 10% of the maximum performance for (a) 2 cores, (b) 4 cores and (b) 8 cores.

How do we use the calculated range?

- Select a reasonable block size, e.g. 4×512
- Find the range of chunk size that results in the calculated range of grain size, Grain_size=$r \times c \times ch$

Method: Finding the Grain Size Range for Maximum Performance
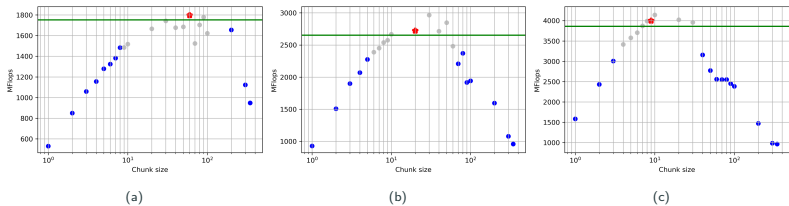
Figure 15: matrix size 690 × 690 with block size of 4 × 512 on (a) 2 cores, (b) 4 cores, and (c) 8 cores.

Method: Polynomial Model, Wrap up

Strength:

- Simple model, can easily find the maximum.

Weakness:

- It is not physical.

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 16: An example of results obtained from Blazemark

Setup: 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.

Our Contributions

- To our knowledge, there has not been a work to create a 3D model of the throughput, grain size, and number of cores.
- We are proposing a method to apply the developed model to a linear algebra library, in a way specific to our application, and the machine architecture.

Future Work

- Generalizing the current model to different matrix sizes
- Looking into more complex expressions
- Finding an analytical model to explain the observed behavior between execution time, grain size, and number of cores.

Thank you!

BLAS operations

Level 1	addition/scaling	$\alpha x, \quad \alpha x + y$
	dot products, norms	$x^T y, \quad \|x\|_2, \quad \|x\|_1$

Level 2	matrix/vector products	$\alpha A x + \beta y, \quad \alpha A^T x + \beta y$
	rank 1 updates	$A + \alpha x y^T, \quad A + \alpha x x^T$
	rank 2 updates	$A + \alpha x y^T + \alpha y x^T$
	triangular solves	$\alpha T^{-1} x, \quad \alpha T^{-T} x$

Level 3	matrix/matrix products	$\alpha AB + \beta C, \quad \alpha AB^T + \beta C$
		$\alpha A^T B + \beta C, \quad \alpha A^T B^T + \beta C$
	rank-k updates	$\alpha AA^T + \beta C, \quad \alpha A^T A + \beta C$
	rank-$2k$ updates	$\alpha A^T B + \alpha B^T A + \beta C$
	triangular solves	$\alpha T^{-1} C, \quad \alpha T^{-T} C$

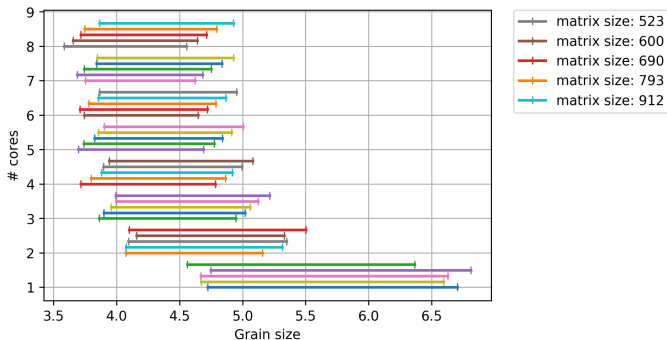**Figure 17:** https://web.stanford.edu/class/ee392o/nlas-foils.pdf

Figure 18: The range of grain size within 10% of the maximum performance of the fitted polynomial function for *DMATDMATADD* benchmark for different number of cores for matrix size 523 × 523 to 912 × 912.

Method: Modeling Performance based on Grain Size

$$Mean\_relative\_absolute\_error = \frac{1}{n} \sum_{i=1}^{n} abs(1 - p_i/t_i)$$
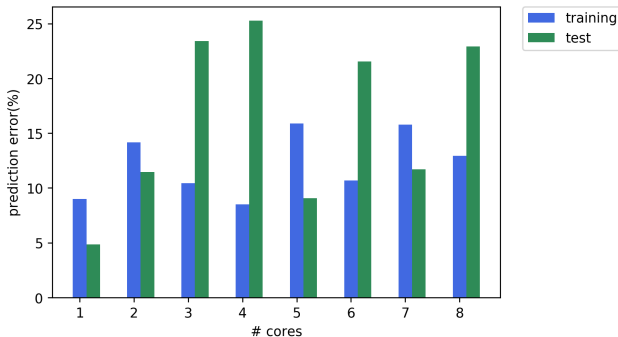
n is the number of samples

Figure 19: The training and test error for fitting data obtained from the *DMATDMATADD* benchmark for matrix size 690 × 690 against different number of cores cores.
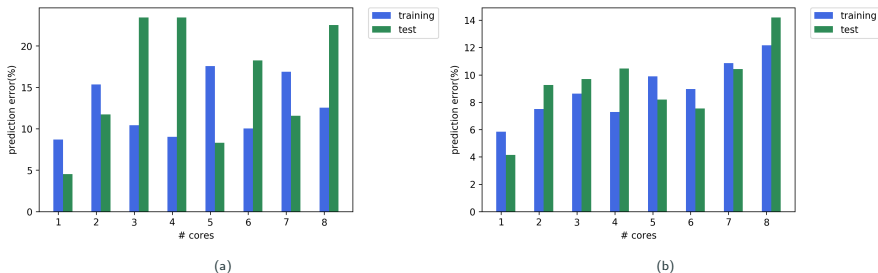
Figure 20: (a) All the data points are include in calculation of error, (b) the leftmost sample was removed from error calculation.
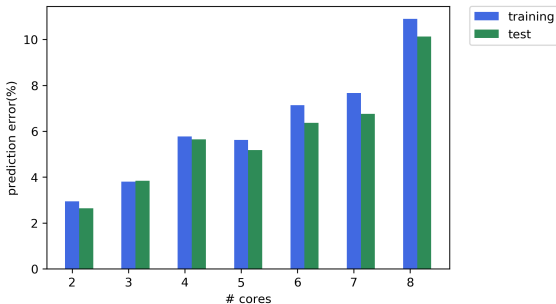
Figure 21: The error in fitting execution time with the bathtub formula for *DMATDMATADD* benchmark for matrix size 690 × 690 with different number of cores.

Related Work

- Liu et al. estimated the optimal number of cores to run the program on based on cache specific traces.
- Khatami et al. used logistic regression to find the best chunk size based on some static and dynamic features of the loop.
- Thoman et al. proposed a compile-time and runtime solution, using an effort estimation function set the chunk size.

- Laberge et al. used machine learning to find the best chunk size to get the maximum performance, while block size was fixed statistically.

- Features included: matrix size, number of cores, number of floating point operations, number of iterations.
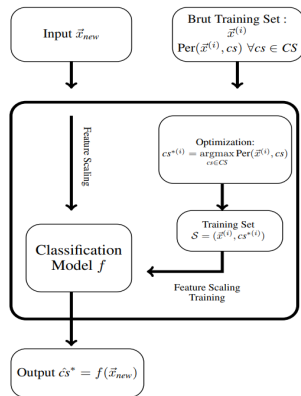

Figure 22: Laberge, G., Shirzad, S., Diehl, P., Kaiser, H., Prudhomme, S., Lemoine, A. (2019). Scheduling optimization of parallel linear algebra algorithms using Supervised Learning. arXiv preprint arXiv:1909.03947.

Method: Modeling Performance based on Grain Size

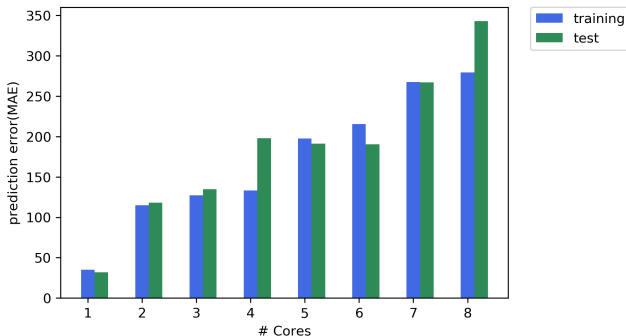$$Mean\_Absolute\_Error = \frac{1}{n} \sum_{i=1}^{n} abs(t_i - p_i)$$

Figure 23: The training and test error for fitting data obtained from the *DMATDMATADD* benchmark for matrix size 690 × 690 against different number of cores cores.

Background: Modeling Performance based on number of cores

- Amdahl's Law

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

- Universal Scalability Law(USL)

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

  - Models the effects of linear speedup, contention delay, and coherency delay due to crosstalk
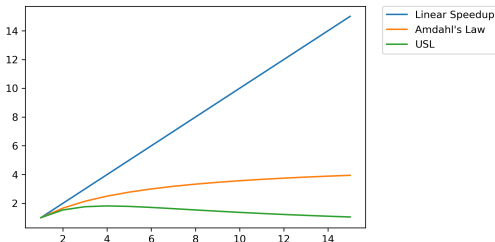

Figure 24: An example of speedup based on Amdahl's law and USL compared to the ideal linear speedup where $\sigma = 0.2$ and $\kappa = 0.05$.