HPX Backend for Blaze

Shahrzad Shirzad

Nov 7, 2019

STE||AR GROUP

Blaze

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

Parallelization

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

- HPX
- OpenMP
- C++ threads
- Boost

HPX Backend

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

- HPX for-loop with static chunking and chunk size=1

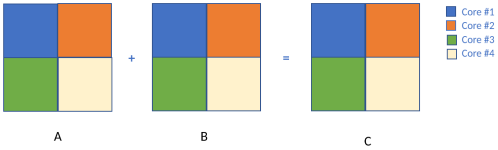


Figure 1: An example of how C=A+B is performed in HPX Backend with 4 cores

Objective

Dynamically divide the work among the cores based on number of cores, matrix size, operation, etc. 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

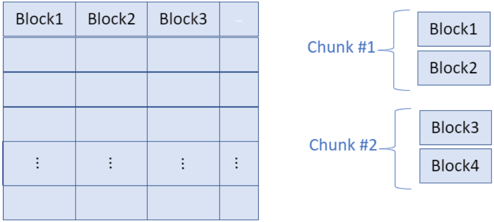


Figure 2: An example of blocking and creating chunks for chunk_size = 2

Background

- Effect of Task Granularity on execution time
- Universal Scalibility Law

Task Granularity

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

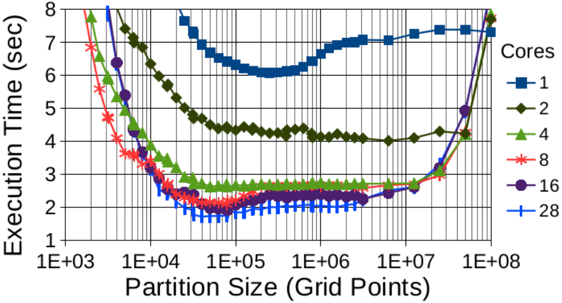


Figure 3: The effect of task size on execution time for Stencil application[1]

[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.

Universal Scalibility Law

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

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1) + \kappa N(N-1)}$$

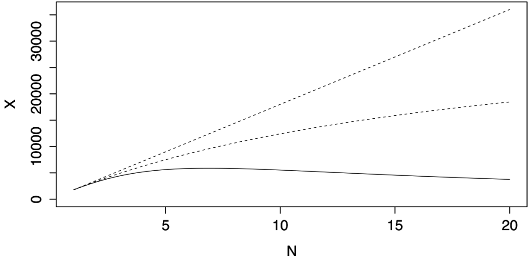


Figure 4: Throughput vs. number of cores[2]

[2] Schwarz, B. "Practical Scalability Analysis with the Universal Scalability Law." (2015).

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 5: An example of results obtained from Blazemark

Method

- Starting from *DMATDMATADD* benchmark, C = A + B where A, B, and C are N by N matrices
- Collect data with different configurations such az matrix size, number of cores, block_size, chunk_size.
  - matrix sizes: 200, 230, 264, 300, 396, 455, 523, 600, 690, 793, 912, 1048, 1200, 1380, 1587
  - number of cores: 1, 2, 3, ..., 8
  - block_sizes: [4, 8, 12, 16, 20, 32] by [64, 128, 256, 512, 1024] blocks
  - chunk_sizes: between 1 and total number of blocks (logarithmic increase)

Results

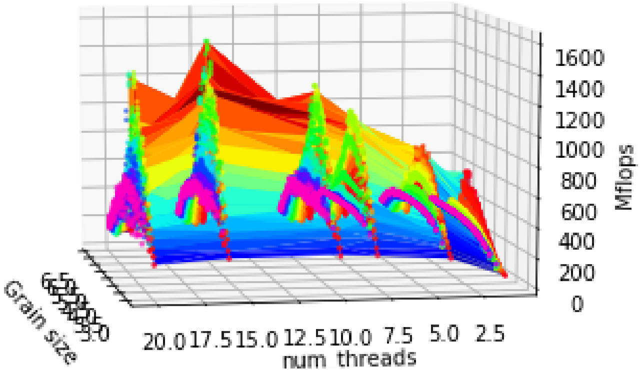


Figure 6: Results of running the *DMATDMATADD* benchmark for different matrix sizes with different block_size and chunk_size combinations

Method

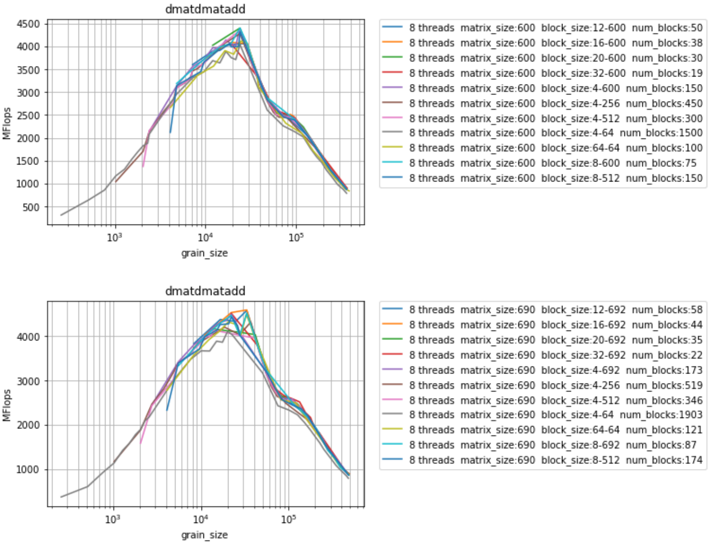


Figure 7: Results of running the *DMATDMATADD* benchmark on 8 cores for different block sizes

Modeling Execution Time based on Grain Size

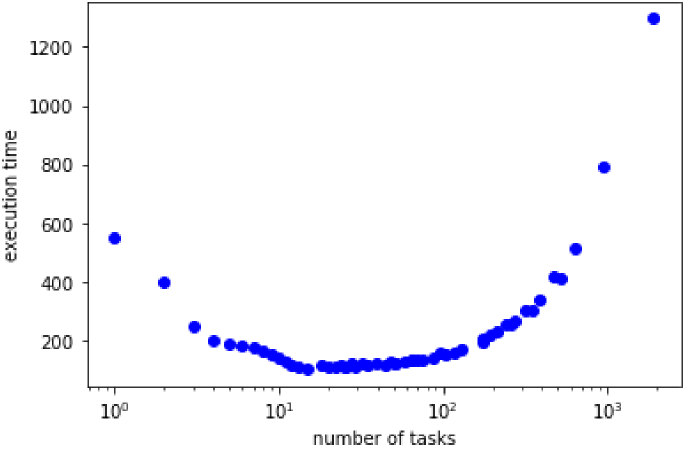


Figure 8: Results of running the *DMATDMATADD* benchmark on 8 cores matrix size 690(time unit is microseconds)

Modeling Execution Time based on Grain Size

- Overheads of creating tasks
- Starvation

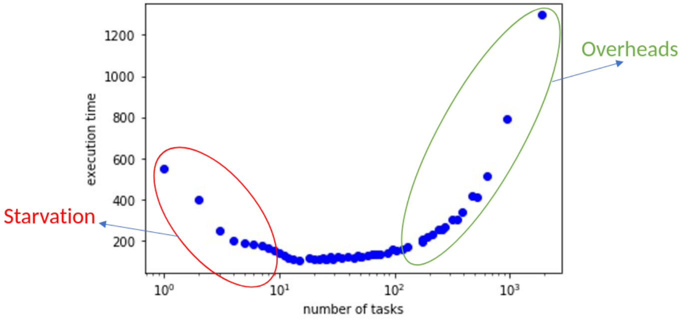


Figure 9: Results of running the *DMATDMATADD* benchmark on 8 cores matrix size 690(time unit is microseconds)

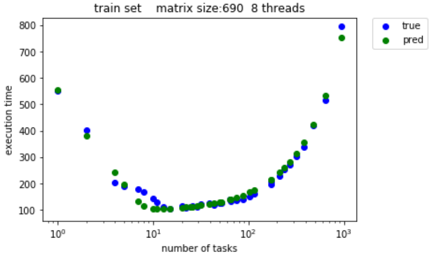
Modeling Execution Time based on Grain Size

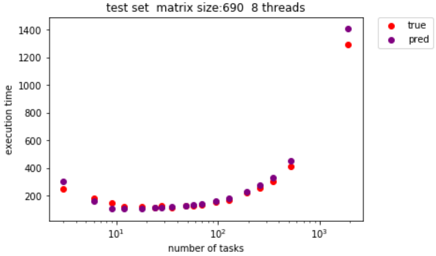
$$Execution\_time =$$

$$\begin{cases} \dfrac{\alpha \times num\_tasks + \ t_s}{num\_tasks} + \ \beta \times num\_tasks + \ \gamma & num\_tasks < N \\ \dfrac{\alpha \times num\_tasks + \ t_s}{N} + \ \beta \times num\_tasks + \ \gamma & num\_tasks \geq N \end{cases}$$

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

- For a fixed matrix size, and number of cores we need 4 parameters to estimate execution time based on number of tasks
- How does these four parameters change for different number of cores?
  - used USL to model each of these parameters

Modeling Execution Time based on Grain Size

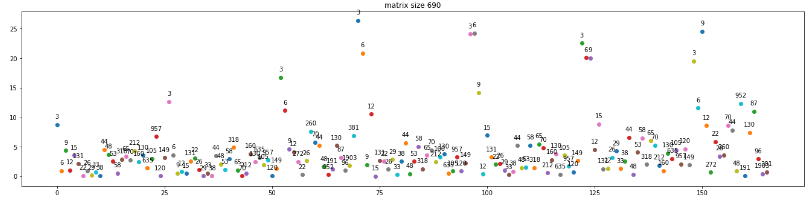


Figure 11: Relative error of predicting execution time

Next Step

- Find the range of the flat region of grain size
- Choosing a small block_size while number of columns is divisible by cache line
- Find the range of chunk_sizes for the given range of grain size
- Generalize the model to integrate the matrix size

Thank you!