Parallel Programming SS21 Final Project

Project 02 – Gaussian Elimination

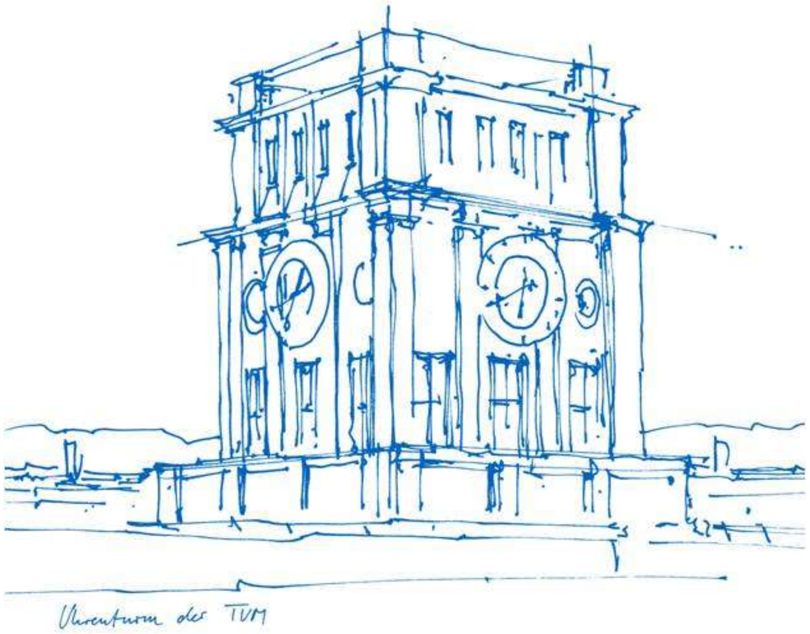
Group 215

09.07.2021

Zhelin Yang

Dian Yuan

Jing Xiong



Sequential code analysis - Profiling

size 1024x1024

```
Samples: 1K of event 'task-clock', Event count (approx.): 480750000
  Children      Self  Command   Shared Object          Symbol
    77,95%    77,95%  serialge  serialge               [.] Serial::ForwardElimination
+    7,07%     0,00%  serialge  serialge               [.] main
+    6,97%     0,10%  serialge  serialge               [.] Serial::SerialSolve
+    6,81%     0,00%  serialge  serialge               [.] Utility::InitializeArray
+    6,50%     0,73%  serialge  serialge               [.] ReadLine
+    5,82%     1,14%  serialge  libstdc++.so.6.0.28    [.] std::istream:: M extract<double>
+    5,41%     0,00%  serialge  [unknown]              [.] 0000000000000000
+    3,17%     3,17%  serialge  libstdc++.so.6.0.28    [.] std::num get<char, std::istreambuf
+    2,86%     2,86%  serialge  libc-2.31.so           [.]   GI     strtod l internal
+    2,03%     2,03%  serialge  libstdc++.so.6.0.28    [.] std::num get<char, std::istreambuf
+    1,51%     1,30%  serialge  libstdc++.so.6.0.28    [.] std::  convert to v<double>
     1,46%     1,46%  serialge  libc-2.31.so           [.] round and return
     1,14%     1,14%  serialge  libstdc++.so.6.0.28    [.] std::istream::sentry::sentry
+    1,14%     1,14%  serialge  libc-2.31.so           [.]   strlen avx2
```

size 2048x2048

```
Samples: 14K of event 'task-clock', Event count (approx.): 3709250000
  Children      Self  Command   Shared Object          Symbol
    89,00%    88,99%  serialge  serialge               [.] Serial::ForwardElimination
+    3,34%     0,00%  serialge  serialge               [.] main
+    3,31%     0,07%  serialge  serialge               [.] Serial::SerialSolve
+    3,24%     0,00%  serialge  serialge               [.] Utility::InitializeArray
+    3,21%     0,28%  serialge  serialge               [.] ReadLine
+    2,76%     0,41%  serialge  libstdc++.so.6.0.28    [.] std::istream:: M extract<double>
+    1,48%     1,48%  serialge  libc-2.31.so           [.]   GI     strtod l internal
+    1,46%     0,00%  serialge  [unknown]              [.] 0000000000000000
+    1,27%     1,27%  serialge  libstdc++.so.6.0.28    [.] std::num get<char, std::istreambuf
+    1,12%     0,00%  serialge  [unknown]              [.] 0x00007ffc69927ad4
+    0,86%     0,86%  serialge  libstdc++.so.6.0.28    [.] std::num get<char, std::istreambuf
     0,79%     0,79%  serialge  libstdc++.so.6.0.28    [.] std::istream::sentry::sentry
     0,72%     0,72%  serialge  libc-2.31.so           [.] round and return
```

size 4096x4096

```
Samples: 112K of event 'task-clock', Event count (approx.): 28046500000
  Children      Self  Command   Shared Object          Symbol
    94,27%    94,27%  serialge  serialge               [.] Serial::ForwardElimination
+    1,67%     0,00%  serialge  serialge               [.] main
+    1,66%     0,03%  serialge  serialge               [.] Serial::SerialSolve
+    1,62%     0,00%  serialge  serialge               [.] Utility::InitializeArray
+    1,60%     0,13%  serialge  serialge               [.] ReadLine
+    1,40%     0,00%  serialge  [unknown]              [k] 0000000000000000
+    1,39%     0,20%  serialge  libstdc++.so.6.0.28    [.] std::istream:: M extract<double>
+    0,82%     0,82%  serialge  libc-2.31.so           [.]   GI     strtod l internal
+    0,70%     0,70%  serialge  libstdc++.so.6.0.28    [.] std::num get<char, std::istreambuf
     0,46%     0,46%  serialge  libstdc++.so.6.0.28    [.] std::num get<char, std::istreambuf
     0,43%     0,43%  serialge  libstdc++.so.6.0.28    [.] std::istream::sentry::sentry
     0,39%     0,00%  serialge  [unknown]              [.] 0x0000000000000020
     0,36%     0,26%  serialge  libstdc++.so.6.0.28    [.] std::  convert to v<double>
     0,35%     0,35%  serialge  libc-2.31.so           [.] round and return
     0,28%     0,28%  serialge  libc-2.31.so           [.]  int free
```

size 8192x8192

```
Samples: 866K of event 'task-clock', Event count (approx.): 216643750000
  Children      Self  Command   Shared Object          Symbol
    97,05%    97,04%  serialge  serialge               [.] Serial::ForwardElimination
+    0,88%     0,00%  serialge  serialge               [.] main
+    0,87%     0,02%  serialge  serialge               [.] Serial::SerialSolve
+    0,85%     0,00%  serialge  serialge               [.] Utility::InitializeArray
+    0,84%     0,07%  serialge  serialge               [.] ReadLine
+    0,74%     0,12%  serialge  libstdc++.so.6.0.28    [.] std::istream:: M extract<double>
     0,43%     0,43%  serialge  libc-2.31.so           [.]   GI     strtod l internal
     0,42%     0,00%  serialge  [unknown]              [.] 0000000000000000
     0,38%     0,38%  serialge  libstdc++.so.6.0.28    [.] std::num get<char, std::istreambuf
     0,29%     0,00%  serialge  [unknown]              [.] 0x00007ffe546a5954
     0,24%     0,24%  serialge  libstdc++.so.6.0.28    [.] std::num get<char, std::istreambuf
     0,23%     0,23%  serialge  libstdc++.so.6.0.28    [.] std::istream::sentry::sentry
     0,17%     0,13%  serialge  libstdc++.so.6.0.28    [.] std::  convert to v<double>
```

Sequential code analysis - Profiling

```
Performance counter stats for './serialge ./ge data/size8192x8192':

       216.171,10 msec task-clock                #    1,000 CPUs utilized
            3.996      context-switches          #    0,018 K/sec
                2      cpu-migrations            #    0,000 K/sec
          131.257      page-faults               #    0,607 K/sec
  893.168.658.549      cycles                    #    4,132 GHz
2.459.203.457.971      instructions              #    2,75  insn per cycle
  565.625.107.143      branches                  # 2616,562 M/sec
       42.950.841      branch-misses             #    0,01% of all branches

    216,187621527 seconds time elapsed

    216,013328000 seconds user
      0,160030000 seconds sys
```

Sequential code analysis - Amdahl's law

Parameters:

f = fraction of parallel execution — f = 99.8%(size2048x2048)

p = number of parallel tasks/threads/processes — p = 4

Speedup refers to speedup of function “Solve”

Theoretical Speedup:

$$SU(p) = \frac{T}{T(p)} = \frac{T}{(1-f)*T+\frac{f*T}{p}} = \frac{1}{1-f+\frac{f}{p}} = \frac{1}{1-0.998+\frac{0,998}{4}} = 3.976$$

Sequential code analysis - Amdahl's law

Parameters:

f = fraction of parallel execution f ≈ 100%

p = number of parallel tasks/threads/processes p = 4

Speedup refers to speedup of function "Solve"

$$SU(p) = \frac{T}{T(p)} = \frac{T}{(1-f)*T + \frac{f*T}{p}} = \frac{1}{1-f+\frac{f}{p}} = \frac{1}{1-1+\frac{1}{4}} = 4$$

Maximal Speedup: Around 4

OpenMP - Parallelized implementation and approach

```
void ForwardElimination(double *matrix, double *rhs, int rows, int columns){
    for(int row = 0; row < rows; row++){
        // Extract Diagonal element
        int diag_idx = row*rows + row;
        double diag_elem = matrix[diag_idx];
        #pragma omp parallel for schedule(dynamic)
        for (int lower_rows=row+1; lower_rows<rows; lower_rows++){
            int below_diag_idx = lower_rows*rows + row;
            assert(diag_elem!=0);
            // Compute the factor
            double elimination_factor = matrix[below_diag_idx]/diag_elem;
            int element_idx;
            for (int column=row+1; column<columns; column++){
                // set the column index of the entry to be operated
                element_idx = lower_rows*rows + column;
                // subtract the row
                matrix[element_idx] -= elimination_factor*matrix[row*rows+column];
            }
            rhs[lower_rows] -= elimination_factor*rhs[row];
            // set below diagonal elements to 0
            // matrix[below_diag_idx] = 0.;
        }
    }
}
```

First try

-Totally 3 „for loops“ in this function

-Parallel region: 2nd „for loop“

-Schedule:dynamic

OpenMP - Intermediate Speed-up results, profiling

size	Overhead of ForwardElimination	Overhead of vmovapd	Speedup (on my PC)	Speedup (on server)
2048	80.99%	69.15%	1.39	≈10
4096	90.44%	73.9%	1.23	
8192	95.06%	75.82%	1.14	

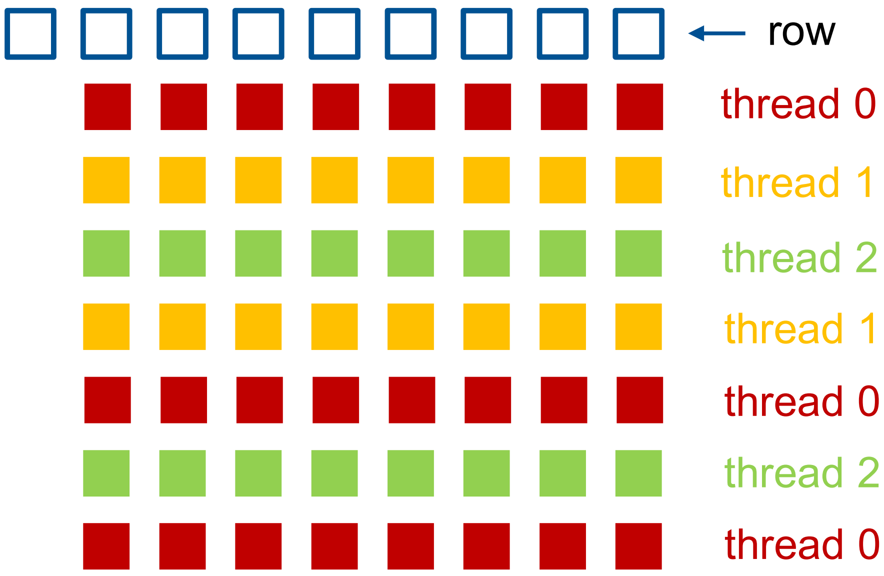
PS: speedup on PC is a speedup for function “Solve” compared with the optimized sequential code

Result of first try

-Bottleneck: ForwardElimination--Data Transfer

-Speedup: Not bad on my PC, but on the server it is not fast enough

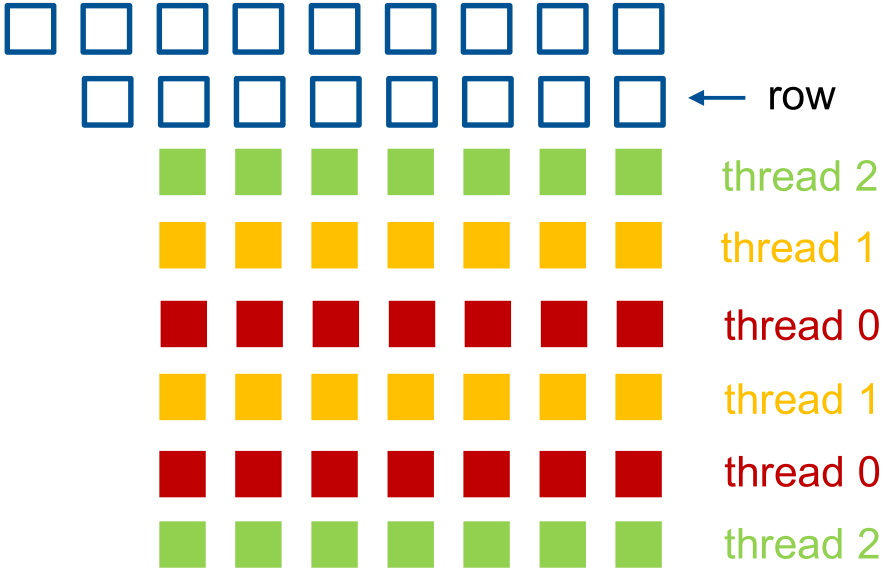
OpenMP - Parallelized implementation and approach



First try

-Distribution: The task row addresses distributed to one thread are not continuous

OpenMP - Parallelized implementation and approach



First try

-Distribution: The task row addresses distributed to one thread are not continuous and randomly

OpenMP - Final Implementation improvements and new speed-up

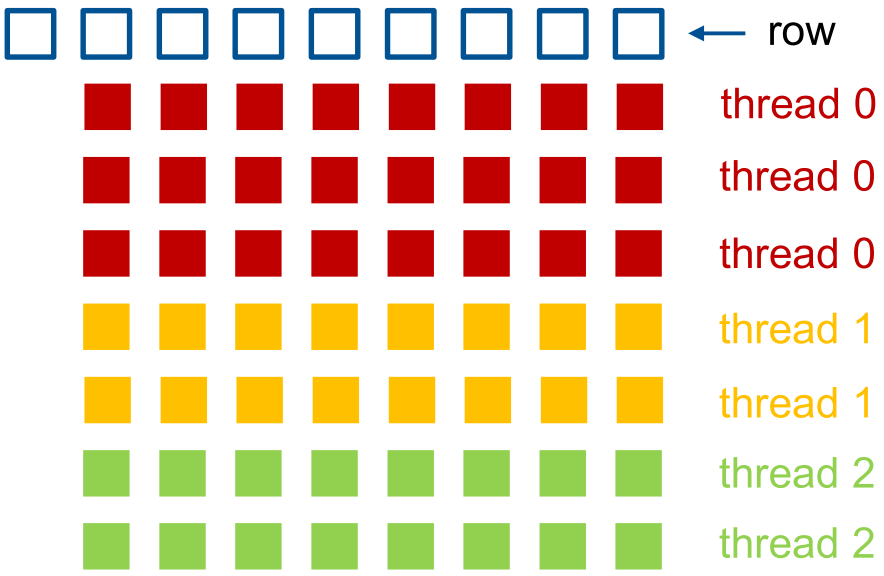
```
void ForwardElimination(double *matrix, double *rhs, int rows, int columns){
    for(int row = 0; row < rows; row++){
        // Extract Diagonal element
        int diag_idx = row*rows + row;
        double diag_elem = matrix[diag_idx];
        #pragma omp parallel for
        for (int lower_rows=row+1; lower_rows<rows; lower_rows++){
            int below_diag_idx = lower_rows*rows + row;
            assert(diag_elem!=0);
            // Compute the factor
            double elimination_factor = matrix[below_diag_idx]/diag_elem;
            int element_idx;
            for (int column=row+1; column<columns; column++){
                // set the column index of the entry to be operated
                element_idx = lower_rows*rows + column;
                // subtract the row
                matrix[element_idx] -= elimination_factor*matrix[row*rows+column];
            }
            rhs[lower_rows] -= elimination_factor*rhs[row];
            // set below diagonal elements to 0
            // matrix[below_diag_idx] = 0.;
        }
    }
}
```

Last Version

-Schedule:default

-More server friendly

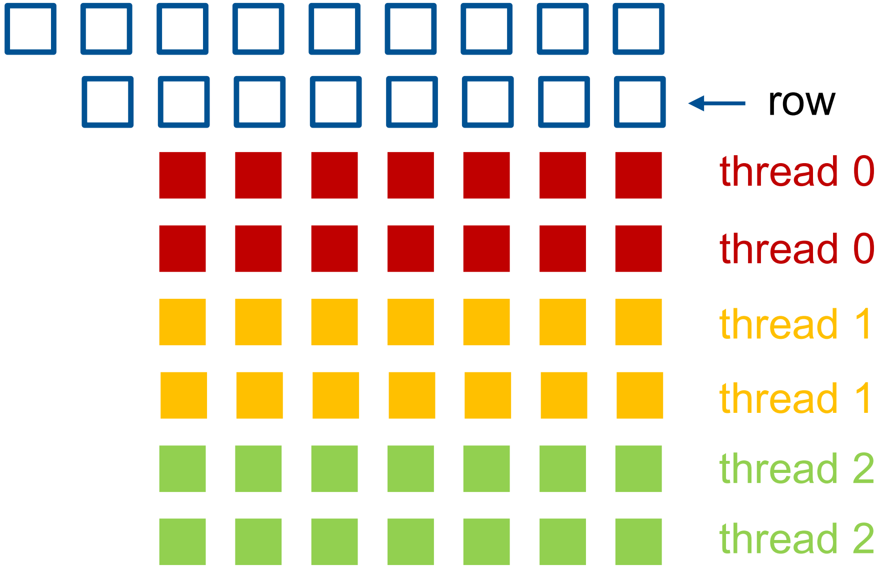
OpenMP - Parallelized implementation and approach



Last version

-The thread will access similar address in the next round

OpenMP - Parallelized implementation and approach



Last version

-The thread will access similar address in the next round

OpenMP - Final Speed-up results, profiling

size	Cache misses	
	dynamic	default
2048	57%	57%
4096	84%	83%
8192	90%	86%

OpenMP - Final Speed-up results

size	Overhead of ForwardElimination	Overhead of vmovapd	Speedup (on my PC)	Speedup (on server)
2048	80.99%	69.15%	1.39	≈10(dynamic) ≈14(default)
4096	90.44%	73.9%	1.23	
8192	95.06%	75.82%	1.14	

Result of Final

-Speedup: Similar as the previous version on my PC, but better on server

OpenMP - Final Implementation

Theoretical speedup: 3~4(with 4 threads) according to Amdahl's law
- Almost all of ForwardElimination is parallized

Bottleneck: Large cost from data transfer
- The bigger the matrix is, the larger the cost from data transfer would be.

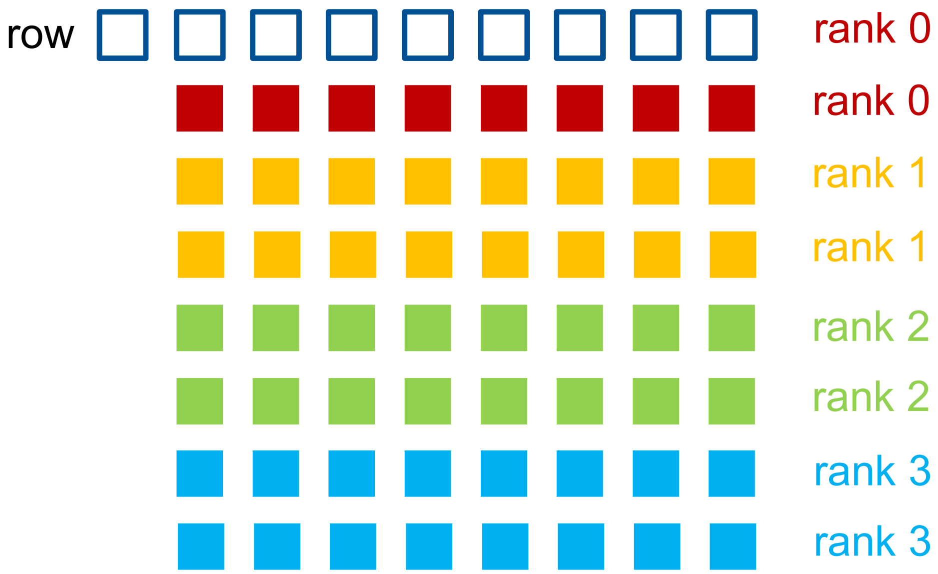
MPI - Parallelized implementation and approach

First try

-Data are equally distributed to each progress in block

-Addresses of data are continuous in each progress

MPI - Parallelized implementation and approach

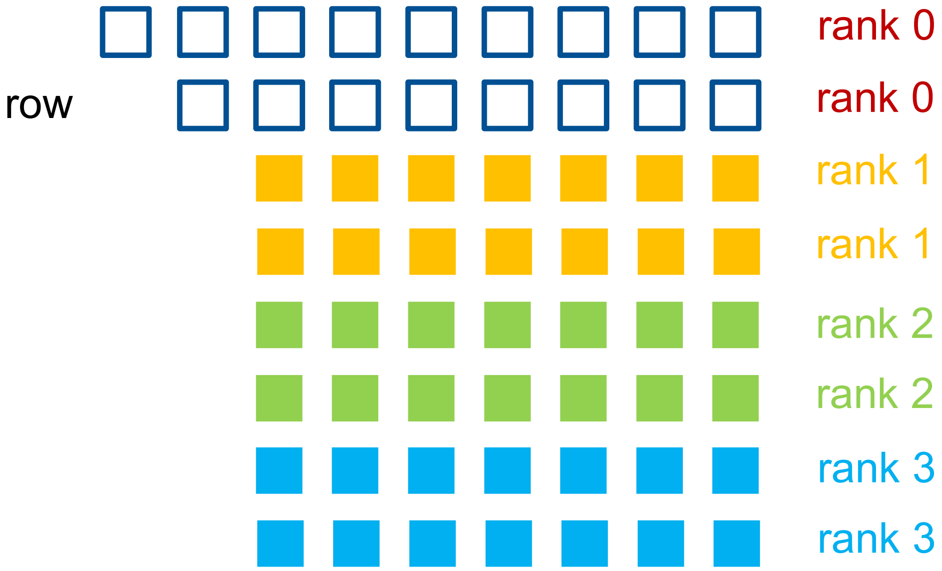


First try

-Data are equally distributed to each progress in block

-Addresses of data are continuous in each progress

MPI - Parallelized implementation and approach



First try

-Data are equally distributed to each progress in block

-Addresses of data are continuous in each progress

-Not balanced

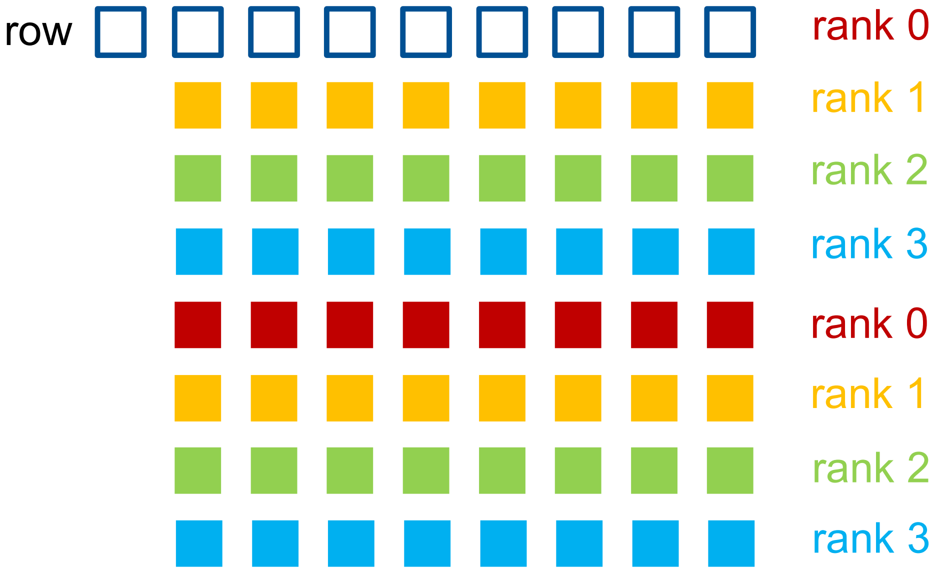
MPI - Final Implementation improvements and new speed-up

Final version

-Data are equally distributed to each progress

-Addresses of data are not continuous in each progress

MPI - Final Implementation improvements and new speed-up

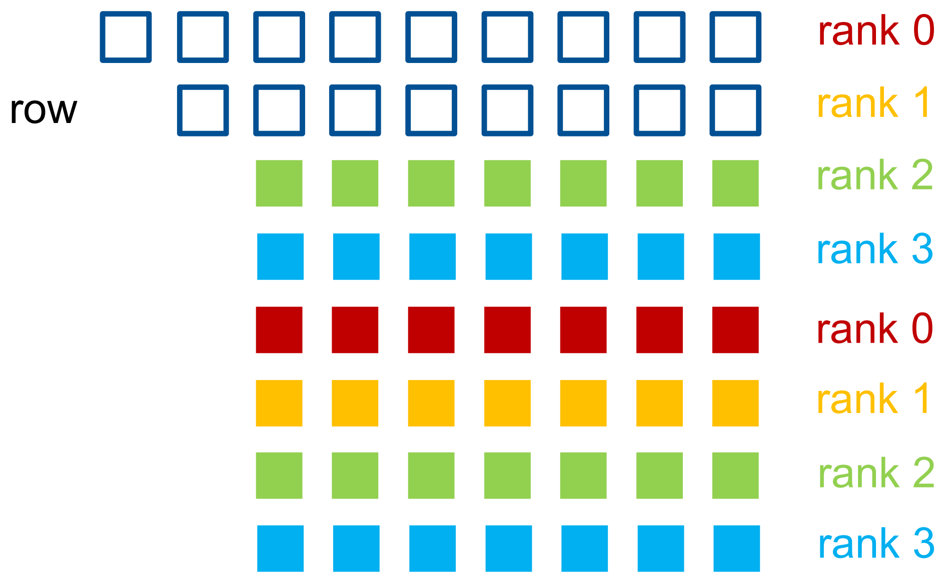


Final version

-Data are equally distributed to each progress

-Addresses of data are not continuous in each progress

MPI - Final Implementation improvements and new speed-up



Final version

-Data are equally distributed to each progress

-Addresses of data are not continuous in each progress

-Almost balanced

MPI – Speedup

size	Speedup on my PC		Speedup Server	
	First try	Final version	First try	Final version
2048	1.39	1.39	10.54	10.89
4096	1.21	1.18		
8192	1.13	1.12		

Comparison

-Speedup: Very close both on PC and server

-The impact of imbalance is not so obvious

MPI - Parallelized implementation and approach

```
# Total Lost Samples: 0
#
# Samples: 2M of event 'cycles'
# Event count (approx.): 1016468430965
#
# Overhead      Pid:Command
# ........  ......................
#
    25.05%     7329:mpige
    25.04%     7330:mpige
    25.03%     7332:mpige
    24.88%     7331:mpige
     0.00%     7328:hydra_pmi_proxy
     0.00%     7326:mpirun
```

Comparison

- The impact of imbalance is not so obvious

- The load seems balanced because there is a barrier at each end of the first “for loop” to update “row+1”

```
MPI_Bcast(matrix+(row+1)*rows, columns, MPI_DOUBLE, r, MPI_COMM_WORLD);
MPI_Bcast(rhs+row+1, 1, MPI_DOUBLE, r, MPI_COMM_WORLD);
```

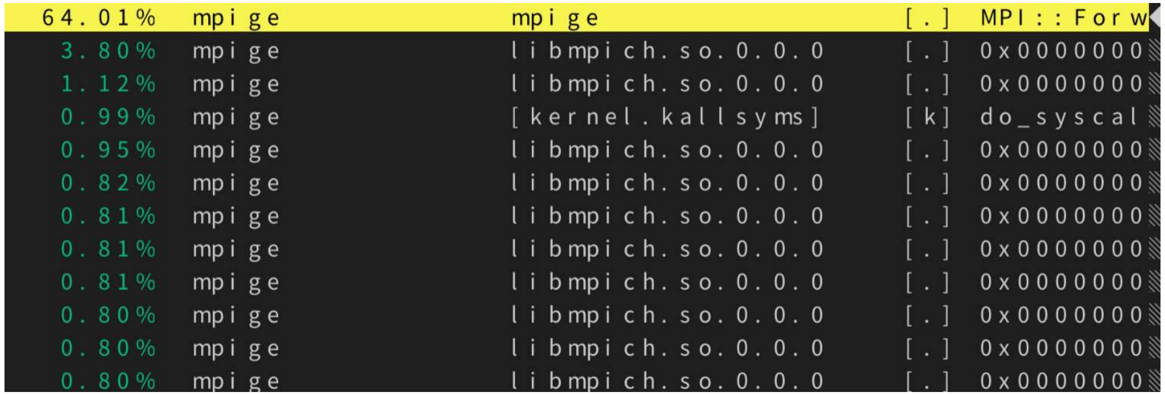
MPI - Profiling

size	First try		Final version	
	ForwardElimination	vmovapd	ForwardElimination	vmovapd
2048	45.13%	52.3%	53.37%	67.27%
4096	57.31%	56.98%	73.14%	72.68%
8192	64.01%	57.75%	85.03%	74.77%

Comparison

- The data address in the first try are continuous, so that it would be easier to access

MPI - Final Speed-up results, profiling



Cost from MPI

-Using MPI might cause some communication cost

MPI - Final Implementation

Theoretical speedup: 3~4(with 4 threads) according to Amdahl’s law
- Almost all of ForwardElimination is parallized

Bottleneck: Large cost from data transfer, MPI communication
- The bigger the matrix is, the larger the cost from data transfer would be
- MPI communication would cost some time

Hybrid - Parallelized implementation and approach

Combine of OMP and MPI approach

- Two threads for MPI(distributed nodes) and two for OMP(multiple cores)
- The structure is almost the same as MPI approach
- OMP is nested within the MPI process

```c
void ForwardElimination(double *matrix, double *rhs, int rows, int columns, int rank, int size){
    for(int row = 0; row < rows-1; row++){
        int block_idx = row / size;
        int l_rows_start;

        if((row%size)<rank){
            l_rows_start = block_idx*size + rank;
        }

        else{
            l_rows_start = (block_idx+1)*size + rank;
        }

        ...

        #pragma omp parallel for num_threads(2)
        for (){
            ...
        }

        ...
        int r = (row+1)%size;
        MPI_Bcast(matrix+(row+1)*rows, columns, MPI_DOUBLE, r, MPI_COMM_WORLD);
        MPI_Bcast(rhs+row+1, 1, MPI_DOUBLE, r, MPI_COMM_WORLD);
    }
}
```

Master thread

Worker threads of the master processor

End OpenMP

Hybrid - Final Performance Results

size	Speedup
2048	1.39
4096	1.20
8192	1.13

**with 4 threads on own PC*

```
#  Overhead        Pid:Command
#  ........  ......................
#
      27.61%       4122:hybridge
      26.99%       4123:hybridge
      18.20%       4125:hybridge
      18.00%       4124:hybridge
```

The two threads responsible for the MPI process share more load

Theoretical speedup: 3~4 (with 4 threads) according to Amdahl's law
- Almost all of ForwardElimination is parallized

Bottleneck: Large cost from data transfer, MPI communication
- The bigger the matrix is, the larger the cost from data transfer would be
- MPI communication would cost some time
- Load balance

-But this is a simulation on a single machine, it should perform better on real distribution nodes

Bonus (SIMD) - Parallelized implementation and approach

```
for (int column=row+1; column<columns; column++){
    // Set the column index of the entry to be operated
    element_idx = lower_rows*rows + column;
    // Subtract the row
    matrix[element_idx] -= elimination_factor*matrix[row*rows+column];
}
```

This is operation on the row and can be vectorized.

AVX Intrinsics

_m256d (four 64-bit float)

The rest do the normal operation

```
int vectorLen = 4;
int end = columns - columns % vectorLen;

__m256d elimination_factor_vec=_mm256_set1_pd(elimination_factor);
__m256d element_row, select_row;

for(int column = row + 1;column < end;column += vectorLen){
    element_row = _mm256_loadu_pd(matrix + lower_rows*rows + column);
    select_row = _mm256_loadu_pd(matrix + row*rows + column);
    __m256d temp_row = _mm256_mul_pd(elimination_factor_vec, select_row);
    element_row = _mm256_sub_pd(element_row, temp_row);
    _mm256_storeu_pd(matrix + lower_rows*rows + column, element_row);
}

//Eliminate the rest elements in the row
for(int column = end; column < columns; column++){
    matrix[lower_rows*rows + column] -= elimination_factor*matrix[row*rows+column];
}
rhs[lower_rows] -= elimination_factor*rhs[row];
}
```

Bonus (SIMD) – Final Performance Results

size	Hybrid Speedup	Bonus Speedup
2048	1.39	1.40
4096	1.20	1.20
8192	1.13	1.13

*with 4 threads on own PC

No significant improvement on speedup.

Let's look at assembly of *hybridge.cpp*

```
.L8:
    vmovupd (%rax,%r12), %xmm6
    vinsertf128 $0x1, 16(%rax,%r12), %ymm6, %ymm1
    vmovupd 0(%r13,%r12), %xmm5
    vinsertf128 $0x1, 16(%r13,%r12), %ymm5, %ymm0
    vmulpd  %ymm3, %ymm1, %ymm1
    vsubpd  %ymm1, %ymm0, %ymm0
    vmovups %xmm0, 0(%r13,%r12)
    vextractf128    $0x1, %ymm0, 16(%r13,%r12)
    addq    $32, %r12
    cmpq    %rsi, %r12
    jne .L8
```

Many vectorized AVX/SSE instructions have been generated after compilation.
It's the characteristic of gcc -o3

Conclusion

**Speedup with 4 threads on own PC*

size	OpenMP	MPI	OMP+MPI	OMP+MPI+SIMD
2048	1.39	1.39	1.39	1.40
4096	1.23	1.18	1.20	1.20
8192	1.14	1.12	1.13	1.13

- Speedup are close among the four approaches.
- Speedup doesn't reach the theoretical speedup mostly due to cost of data transfer
- It would be more appropriate to use MPI and hybrid method for distributed system