Conjugated Gradient Solver

Shi-Lin WANG

SCIPER: 294925
URL: https://c4science.ch/diffusion/9010/

- Final Project -

2020, Spring

Content

Introduction/ 1

Consider minimizing a quadratic function,

Introduction/ 1

Consider minimizing a quadratic function,

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c$$

Introduction/ 1

Consider minimizing a quadratic function,

$$f(x) = \frac{1}{2}x^T A x - b^T x + c$$

- Conjugated gradient solver (CG)
    - Residual & *A*-orthogonal vector
    - Fewer steps during searching

Introduction/ 2

Algorithm of CG:

```
• $x_0$
• $p_0 = r_0 = b - Ax_0$
• For $k = 0, 1, 2, \dots n$ :
    1. $x_{k+1} = x_k + \alpha_k \; p_k$    where    $\alpha_k = (r_k^T r_k)/(p_k^T A p_k)$
    2. $r_{k+1} = b - Ax_{k+1} = r_k - Ap_k$
    3. if $r_{k+1} \leq \epsilon$ then break
    4. $p_{k+1} = r_{k+1} + \beta_k p_k$    where    $\beta_k = (r_{k+1}^T r_{k+1})/(r_k^T r_k)$
    5. $k = k + 1$
• End
```

Introduction/ 2

Algorithm of CG:

```
• $x_0$
• $p_0 = r_0 = b - Ax_0$
• For $k = 0, 1, 2, \dots n$ :
    1. $x_{k+1} = x_k + \alpha_k \; p_k$    where    $\alpha_k = (r_k^T r_k)/(p_k^T A p_k)$
    2. $r_{k+1} = b - Ax_{k+1} = r_k - Ap_k$
    3. if $r_{k+1} \leq \epsilon$ then break
    4. $p_{k+1} = r_{k+1} + \beta_k p_k$    where    $\beta_k = (r_{k+1}^T r_{k+1})/(r_k^T r_k)$
    5. $k = k + 1$
• End
```

Let's see how does computation/storage scales with problem size.

Theoretical Analysis/ Complexity/ 1

Consider problem size as *n* and the number of total iterations as κ ,

Theoretical Analysis/ Complexity/ 1

Consider problem size as n and the number of total iterations as κ ,

Operation	Representation	Flops
Matrix-times-vector	$c = A \times a$	n^2
Vector-plus-scalar-times-vector	$c = a + \alpha \times b$	$2n$
Vector addition	$c = a + b$	n
Inner product	$\alpha = b \cdot c$	n
Division	$\alpha = \alpha / \beta$	1
Assuming A(n×n), a(n), b(n), c(n), α(scalar), β(scalar)		

Table: Summary of operations vs. number of flops

Theoretical Analysis/ Complexity/ 1

Consider problem size as n and the number of total iterations as κ ,

Operation	Representation	Flops
Matrix-times-vector	$c = A \times a$	n^2
Vector-plus-scalar-times-vector	$c = a + \alpha \times b$	$2n$
Vector addition	$c = a + b$	n
Inner product	$\alpha = b \cdot c$	n
Division	$\alpha = \alpha / \beta$	1
Assuming A(n×n), a(n), b(n), c(n), α(scalar), β(scalar)		

Table: Summary of operations vs. number of flops

- The whole program has the total number of flops,

$$Flops = (n + n^2) + \kappa \left((n^2) + 3(2n) + 3(n) + 2(1) \right)$$

Theoretical Analysis/ Complexity/ 2

- Thus, **time complexity** is obtained,

Theoretical Analysis/ Complexity/ 2

- Thus, **time complexity** is obtained,

$$\mathcal{O}(\kappa n^2)$$

Theoretical Analysis/ Complexity/ 2

- Thus, **time complexity** is obtained,

$$\mathcal{O}(\kappa n^2)$$

- With similar method, **space complexity** can also be calculated as,

Theoretical Analysis/ Complexity/ 2

- Thus, **time complexity** is obtained,

$$\mathcal{O}(\kappa n^2)$$

- With similar method, **space complexity** can also be calculated as,

$$\mathcal{O}(n^2)$$

Theoretical Analysis/ Complexity/ 2

- Thus, **time complexity** is obtained,

$$\mathcal{O}(\kappa n^2)$$

- With similar method, **space complexity** can also be calculated as,

$$\mathcal{O}(n^2)$$

- Based on simple observation, it shows both complexities scale in the fashion of n^2. It seems that the dominating parts are where matrix operation occurs.

Theoretical Analysis/ Complexity/ 3

In order to check it, a simple profiling is done by performace analyzaing tool **perf** with the problem size set to $n = 1e4$.

Theoretical Analysis/ Complexity/ 3

In order to check it, a simple profiling is done by performace analyzaing tool **perf** with the problem size set to $n = 1e4$.

Indeed,

- Around 99.75% of execution counts are spent on function cblas_dgemv.
- It does the matrix-vector-multiplication and the vector addition with double precision.

Theoretical Analysis/ Complexity/ 3

In order to check it, a simple profiling is done by performace analyzaing tool **perf** with the problem size set to $n = 1e4$.

Indeed,

- Around 99.75% of execution counts are spent on function cblas_dgemv.
- It does the matrix-vector-multiplication and the vector addition with double precision.

In the following, instead of the whole program, for simplicity, I will then focus on cblas_dgemv only.

Theoretical Analysis/ Roofline Model/ 1

Roofline model provides an easy way to assess the performance of program running on certain machine.

Theoretical Analysis/ Roofline Model/ 1

Roofline model provides an easy way to assess the performance of program running on certain machine.

$$P(AI) = min\begin{cases} \pi \\ AI \times \beta \end{cases}$$

Theoretical Analysis/ Roofline Model/ 1

Roofline model provides an easy way to assess the performance of program running on certain machine.

$$P(AI) = min \begin{cases} \pi \\ AI \times \beta \end{cases}$$

- Given π, β, by observing AI, one can estimate whether program is bounded by bandwidth or CPU on that machine.

Theoretical Analysis/ Roofline Model/ 1

Roofline model provides an easy way to assess the performance of program running on certain machine.

$$P(AI) = min \begin{cases} \pi \\ AI \times \beta \end{cases}$$

- Given π, β, by observing AI, one can estimate whether program is bounded by bandwidth or CPU on that machine.
- I test CG on the debug node of EPFL cluster FIDIS.
  - Processor: **Xeon E5-2690 v4 processors**
  - Memory: **DDR4-2400**

Theoretical Analysis/ Roofline Model/ 2

- Peak band width $\beta = 71.53$ Gbytes/sec.

Theoretical Analysis/ Roofline Model/ 2

- Peak band width $\beta = 71.53$ Gbytes/sec.
- To estimate peak performance π, I use the equation,

$$\begin{aligned} \pi &\approx \text{Number of FP ports} \times \\ &\quad \text{Flops/cycle} \times \\ &\quad \text{Vector length} \times \\ &\quad \text{Frequency} \times \\ &\quad \text{Number of cores} \\ &= 2 \times 16 \times 32 \times 2.6 \times 1 \quad \text{Gflops/sec} \end{aligned}$$

Theoretical Analysis/ Roofline Model/ 2

- Peak band width $\beta = 71.53$ Gbytes/sec.
- To estimate peak performance π, I use the equation,

$$\begin{aligned} \pi &\approx \text{Number of FP ports} \times \\ &\quad \text{Flops/cycle} \times \\ &\quad \text{Vector length} \times \\ &\quad \text{Frequency} \times \\ &\quad \text{Number of cores} \\ &= 2 \times 16 \times 32 \times 2.6 \times 1 \quad \text{Gflops/sec} \end{aligned}$$

- To estimate AI, I use the equation,

$$AI = \frac{W}{Q}$$

Theoretical Analysis/ Roofline Model/ 3

- For simplicity, I consider cblas_dgemv as overall process.

Theoretical Analysis/ Roofline Model/ 3

- For simplicity, I consider cblas_dgemv as overall process.

$$AI = (2n^2)/(n^2 + 3n) = 1.9994 \approx 2 \quad \text{flops/bytes}$$

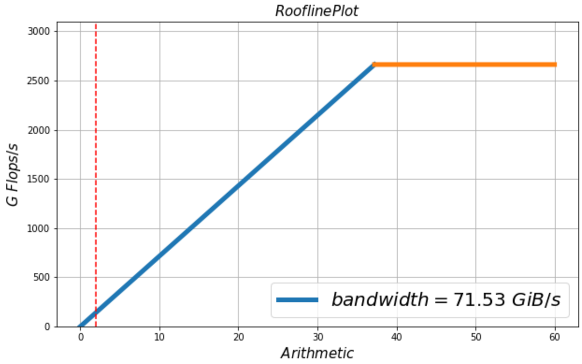
- Thus, the Roofline plot is obtained,

Theoretical Analysis/ Roofline Model/ 3

- For simplicity, I consider cblas_dgemv as overall process.

$$AI = (2n^2)/(n^2 + 3n) = 1.9994 \approx 2 \quad \text{flops/bytes}$$

- Thus, the Roofline plot is obtained,



Parallel Implementation/ 1

Based on Roofline plot, although the program is already bounded by bandwidth, one can still parallelize it to boost performance.

[1] MATH-454 PHPC lecture-01

Parallel Implementation/ 1

Based on Roofline plot, although the program is already bounded by bandwidth, one can still parallelize it to boost performance.

- Given a fixed problem size, Amdahl's law [1] is usually used to estimate the speed-up $\mathcal{S}$,

$$\mathcal{S}(p) \leq \frac{t_s + t_p}{t_s + t_p/p} = \frac{1}{s + a/p}$$

[1] MATH-454 PHPC lecture-01

Parallel Implementation/ 1

Based on Roofline plot, although the program is already bounded by bandwidth, one can still parallelize it to boost performance.

- Given a fixed problem size, Amdahl's law [1] is usually used to estimate the speed-up $\mathcal{S}$,

$$\mathcal{S}(p) \leq \frac{t_s + t_p}{t_s + t_p/p} = \frac{1}{s + a/p}$$

- To take the problem size into consideration, Gustafson's law [1] is then used,

$$\mathcal{S}(p) \leq \frac{t_s + t_p}{t_s + t_p/p} = \frac{s + a \times n}{s + a \times \frac{n}{p}}$$

[1] MATH-454 PHPC lecture-01

Parallel Implementation/ 2

Assuming $a \propto$ flops(cblas_dgemv), I can calculate perf result: 99.75%

$$(s, a) = (0.07\%,\ 99.93\%)$$

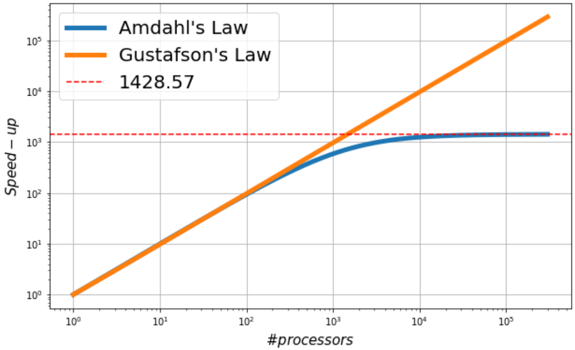
Therefore, given $n = 1e4$,

Parallel Implementation/ 2

Assuming $a \propto$ flops(cblas_dgemv), I can calculate perf result: 99.75%

$$(s, a) = (0.07\%,\ 99.93\%)$$

Therefore, given $n = 1e4$,



Then, how to implement parallelization for CG? MPI & CUDA

Parallel Implementation/ Strategy/ MPI/ 1

Consider a matrix-vector multiplication $y = A \times p$,

$$y = A \times p = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

Parallel Implementation/ Strategy/ MPI/ 1

Consider a matrix-vector multiplication $y = A \times p$,

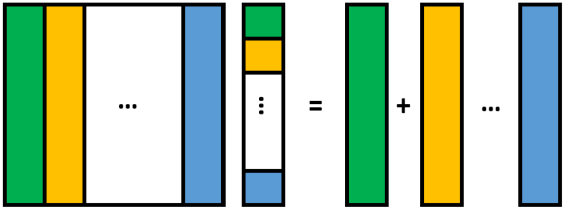
$$y = A \times p = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

For example, given 2 processors, one can distribute tasks as follow,

$$y = \sum_{i=1}^{2} A_i \times p_i \quad = \quad \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \times \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} + \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \\ a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} p_3 \\ p_4 \end{bmatrix}$$

Parallel Implementation/ Strategy/ MPI/ 2

A simple way to visualize such method is as follow,



Let's take a look at part of my code.

Parallel Implementation/ Strategy/ MPI/ 3

In the file *cg.c*,

```
...
intv_A = m*n/size;
intv_x = n/size;
...

while ( k < n ){
    /* 1. Root broadcasts p to all */
    /* 2. Calculate yi=Ai*pi */
    /* 3. Sum yi to root */
    MPI_Bcast(p, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    cblas_dgemv(CblasColMajor,CblasNoTrans, m, n/size,\
                al , &A[intv_A*rank], m, &p[intv_x*rank],\
                incx,  be, App, incy);
    MPI_Reduce(App, Ap, m, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    /* 4. Root does serial task */
    if (rank==0) {...}
}
...
```

Parallel Implementation/ Strategy/ CUDA/ 1

For CUDA, unlike MPI, the parallelization is massive due to a large number of processors of GPU (threads).

Parallel Implementation/ Strategy/ CUDA/ 1

For CUDA, unlike MPI, the parallelization is massive due to a large number of processors of GPU (threads).

To parallelize cblas_dgemv,

- **NVIDIA** has provided a parallelized function for cblas_dgemv called cublas_Dgemv.
- However, the number of threads is already optimized for given matrix and vector size.
- Therefore, I will use it as benchmark and propose a naive implementation which does the same job.

Parallel Implementation/ Strategy/ CUDA/ 2

Consider a matrix-vector multiplication $y = A \times p$,

$$y = A \times p = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

Parallel Implementation/ Strategy/ CUDA/ 2

Consider a matrix-vector multiplication $y = A \times p$,

$$y = A \times p = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

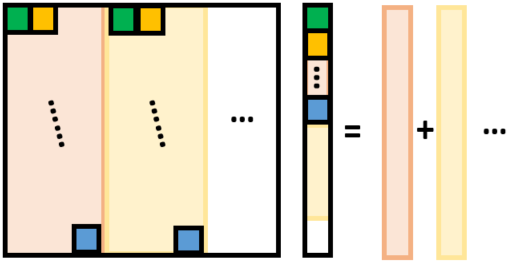
Given 16 threads, one can distribute task to each thread as,

$$y_i = Aij \times p_j$$

To prevent interference during update, it is important to use atomic operation, which guarantees that a race condition won't occur.

Parallel Implementation/ Strategy/ CUDA/ 3

A simple way to visualize such method is as follow,



Let's take a look at part of my code.

Parallel Implementation/ Strategy/ CUDA/ 4

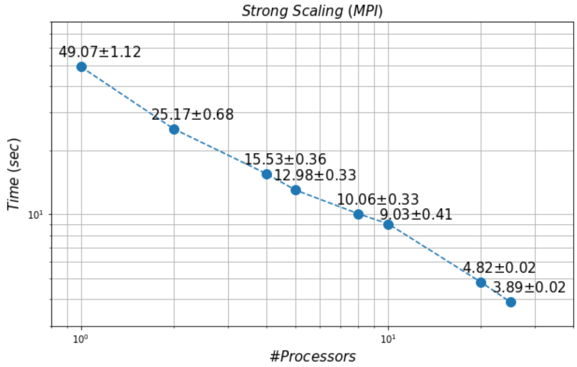
In the file cg.cu for naive implementation,

```
__global__ void kernel_mv(const double * const A, const double * const X, double * const AA, int m) {
    int i,j,k,l;
    int t = blockDim.x*gridDim.x;
    int id = threadIdx.x + blockIdx.x*blockDim.x;
    double temp;

    for (i=0;i<(m*m/t);i++){
        j=(id+i*t);
        k=(id+i*t)/m;
        l=(id+i*t)%m;
        if (j<m*m){
            temp=A[j]*X[k];
            atomic_Add(&AA[l], temp);
        }
    }
}
...
kernel_mv<<<10000,1000>>>(d_A, d_x, d_Ap, m);
...
```

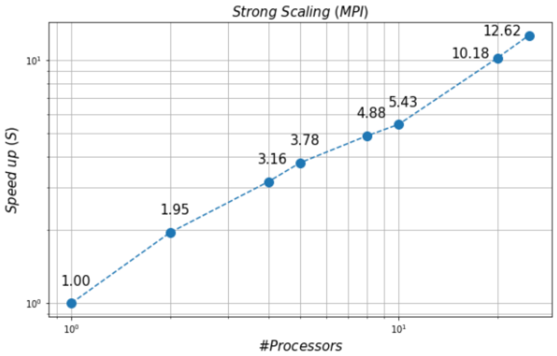
Parallel Implementation/ Result/ MPI/ 1

- Strong scaling, $p = \{1, 2, 4, 5, 8, 10, 20, 25\}$ & $n = 10000$



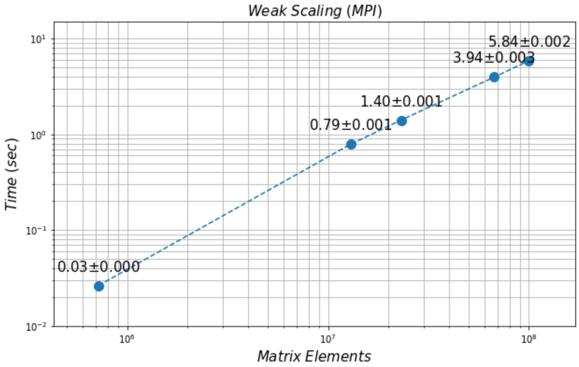
Parallel Implementation/ Result/ MPI/ 2

- Speed up, $p = \{1, 2, 4, 5, 8, 10, 20, 25\}$ & $n = 10000$ Gustafson: 1



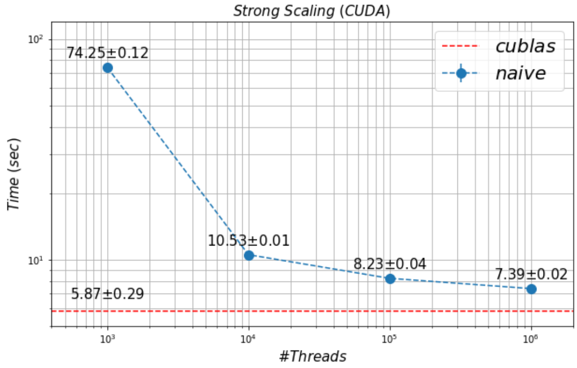
Parallel Implementation/ Result/ MPI/ 3

- Weak scaling, $p = 16$ & $n = \{848, 3600, 4800, 8192, 10000\}$



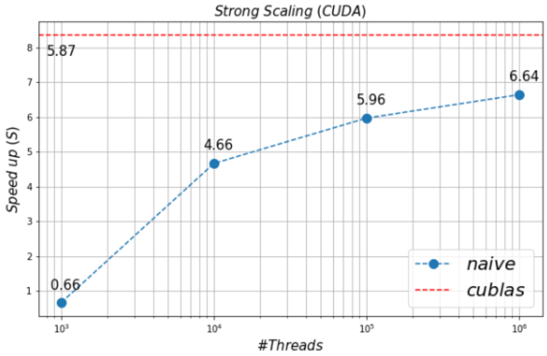
Parallel Implementation/ Result/ CUDA/ 1

- Strong scaling, $p = \{10^3, 10^4, 10^5, 10^6\}$ & $n = 10000$



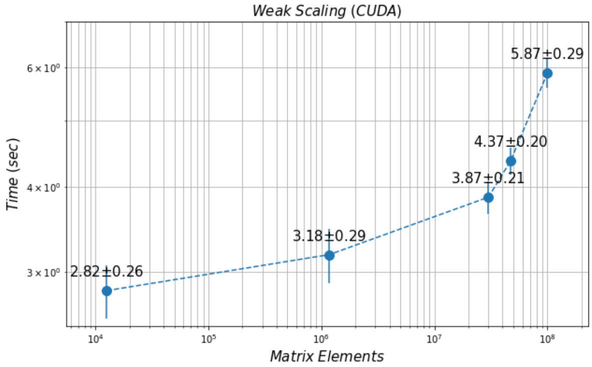
Parallel Implementation/ Result/ CUDA/ 2

- Speed up, $p = \{10^3, 10^4, 10^5, 10^6\}$ & $n = 10000$



Parallel Implementation/ Result/ CUDA/ 3

- Weak scaling, $p = 10^6$ & $n = \{848, 3600, 4800, 8192, 10000\}$



Conclusion

- Indeed, the program is speed-up by paralleization. Speed-up of MPI goes linearly while speed-up of CUDA does not.

Conclusion

- Indeed, the program is speed-up by paralleization. Speed-up of MPI goes linearly while speed-up of CUDA does not.
- The measured speed-up doesn't follow the prediction. It may arise from data transfer, processors' communication and cache size.

Conclusion

- Indeed, the program is speed-up by paralleization. Speed-up of MPI goes linearly while speed-up of CUDA does not.
- The measured speed-up doesn't follow the prediction. It may arise from data transfer, processors' communication and cache size.
- Nevertheless, the linear behavior of MPI still provides a means of extrapolation for one to predict the budget for a bigger project.

Thank you!