Hw1 P2-P3, HPC

Zhe Chen*[1]

[1]Courant Institute of Mathematical Sciences, New York University

February 2019

1 P2: Matrix-matrix multiplication.

1.1 Computing Environment

The two problems, P2 and P3, are running on my office's machine 'blob'. It has a 4 core CPU, Intel Xeon E5-1603, 2.80GHz. Max memory size is 375GB, max memory bandwidth is 31.4GB/s. It uses Sandy Bridge EP, so operations per cycle is 8. Thus, theoretic FLOPS/s is $4Cores * 2.80GHz * 8operations/cycle = 89.6GFLOPs/s$

1.2 Results

As for the test[1], we set $N = 100, 200, ..., 600$, and tested with different optimization flag $O0, ..., O3$. Results are shown in table.4 It's clear that bandwidth and FLOPs are greater when use higher optimization flag. Moreover, it never reaches maxium theoretic quantities and decays as system get larger as a result of bandwidth-bounded.

2 P3: Laplace equation in one space dimension.

2.1 Problem Discription

For a given function $f : [0, 1] \to \mathbf{R}$, we attempt to solve the linear differential equation

$$-u'' = f \text{ in } (0, 1), \text{ and } u(0) = 0, u(1) = 0 \tag{1}$$

for a function u.

$$\begin{aligned} -\Delta u &= f \text{ on } \Omega, \\ u &= 0 \text{ on } \partial\Omega, \end{aligned}$$

which is one of the most important partial differential equations in mathematical physics.

2.2 Numerical Implementation

We use a finite number of grid points in $[0, 1]$ and finite-difference approximations for the second derivative to approximate the solution to (1). We choose the uniformly spaced points $\{x_i = ih : i = 0, 1, \ldots, N, N + 1\} \subset [0, 1]$, with $h = 1/(N + 1)$, and approximate $u(x_i) \approx u_i$ and $f(x_i) \approx f_i$, for $i = 0, \ldots, N + 1$. Using central scheme of Laplacian operator to get second derivatives,

$$-u''(x_i) = \frac{-u(x_i - h) + 2u(x_i) - u(x_i + h)}{h^2} + \text{h.o.t.},$$

*zc1291@nyu.edu

[1]https://github.com/NYU-HPC19/lecture1

Dimension	Time	Gflops/s	GB/s
100	0.015123	1.322519	26.450373
200	0.126933	1.260506	25.210114
300	0.448755	1.203331	24.066611
400	1.257350	1.018014	20.360287
500	2.500023	0.999991	19.999816
600	12.616200	0.342417	6.848338

Table 1: O3 flag

Dimension	Time	Gflops/s	GB/s
100	0.017980	1.112378	22.247552
200	0.139210	1.149342	22.986839
300	0.482602	1.118934	22.378670
400	1.358481	0.942229	18.844574
500	2.603811	0.960131	19.202619
600	12.549510	0.344237	6.884731

Table 2: O2 flag

Dimension	Time	Gflops/s	GB/s
100	0.030117	0.664072	13.281439
200	0.254674	0.628253	12.565059
300	0.866379	0.623283	12.465669
400	2.130831	0.600705	12.014091
500	4.157889	0.601267	12.025332
600	12.659443	0.341247	6.824945

Table 3: O1 flag

Dimension	Time	Gflops/s	GB/s
100	0.085294	0.234482	4.689649
200	0.658981	0.242799	4.855985
300	2.238380	0.241246	4.824917
400	5.549473	0.230653	4.613050
500	10.833987	0.230755	4.615106
600	20.392939	0.211838	4.236761

Table 4: O0 flag

	Total time/s	Iteration	Relative Err.
N=100; GS	0.03	14175	1e-6
N=100; Jacobi	0.05	28348	1e-6
N=10000; GS	1093.68	5000000	0.5496
N=10000; Jacobi	939.93	5000000	0.7043

Table 5: Linear system solvers of $N = 100, 10000$

where h.o.t. stands for a remainder term that is of higher order in h, i.e., becomes small as h becomes small.

finite-dimensional approximation of (1):

$$Au = f, \tag{2}$$

where

$$A = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}, \qquad u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-1} \\ u_N \end{bmatrix}, \qquad f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N \end{bmatrix}.$$

Here we use Gauss-Seidel method and Jacobi method to solve (2), which start from an initial vector $u^0 \in \mathbb{R}^N$ and compute approximate solution vectors u^k, $k = 1, 2, \ldots$.

The component-wise formula for the Jacobi method is

$$u_i^{k+1} = \frac{1}{a_{ii}}\left(f_i - \sum_{j \neq i} a_{ij} u_j^k\right),$$

where a_{ij} are the entries of the matrix A.

The Gauss-Seidel algorithm is given by

$$u_i^{k+1} = \frac{1}{a_{ii}}\left(f_i - \sum_{j<i} a_{ij} u_j^{k+1} - \sum_{j>i} a_{ij} u_j^k\right).$$

2.3 Numerical Results

All codes are written in C programming, available and updated on my Github[1]. I use the property that it's 3-diagonal matrix, i.e. sparsity, or it's too slow for large system.

$f(x) = 1$, i.e., the right hand side vector f is a vector of all ones. And we initiate the iteration with zeros vector, i.e., u^0 is the zero vector.

The iteration stops when residual $\|Au^k - f\|$ decreases by a factor of $10e6$ or it reaches maxium iteration.

We compared the situations where $N = 100$ and $N = 10000$. Iterations needed and total running time is display in table.5 As we can see, Gauss-Seidel method is generally faster in convergence, which results from that GS updates vectors more smartly that uses previous updated information at each iteration while Jacobi does not. And these two method is far too slow as system grows larger, For system N=10000, it is really hard to reach solution even though we did some acceleration using its 3-diagonal property and it's super fast at each iteration. The convergence speed is so slow that we only reach $iterations = 50000000$ and stops. But this somewhat shows already how these two methods perform.

For different optimization, computing speed varies too. We tested $N = 100000, iter = 100$ and set up optimization flag $O0, O1, O2, O3$. Results in table.6 shows that speed does not get quicker for higher optimization level, which only shows acceleration from O0 to O1. I guess this is because it's bandwidth bounded problem for $N = 10000$.

[1] https://github.com/CecilMartin/HPC

Optimizing flag	Time/s	N	Iter
O0	0.03	10000	100
O1	0.01	10000	100
O2	0.01	10000	100
O3	0.01	10000	100

Table 6: Different optimization flag.