HPC final project:
Parallelized Ewald summation of stokes potential

Zhe Chen, Guanchun Li

May 28, 2019

1 Problem background

Periodic Stokes potentials Given the Stokes equation, the Green's function for the velocity field for the free-space problem is given by the Oseen-Burgers tensor:

$$\mathbf{S}(\mathbf{x}) = \frac{\mathbf{1}}{|\mathbf{x}|} + \frac{\mathbf{x}\mathbf{x}}{|\mathbf{x}|^3}. \tag{1.1}$$

Now we consider a system of N point sources at location $\mathbf{x}_n$ with strength $\mathbf{f}_n$, in the periodic setting, the velocity field is given as

$$\mathbf{u}(\mathbf{x}) = \sum_{n=1}^{N} \sum_{\mathbf{p}} \mathbf{S}(\mathbf{x} - \mathbf{x_n} + \mathbf{p})\mathbf{f}_n, \tag{1.2}$$

where $\mathbf{p}$ form the discrete set $\{[iL_x \; jL_y \; kL_z] : (i, j, k) \in \mathbb{Z}^3\}$ and L_x, L_y and L_z are the periodic lengths in the three directions.

However, it turns out that this method is not realistic since Eq.(1.1) decays slowly as $1/|\mathbf{x}|$, which means that we can not use affordable many image box layers to compute Eq.(1.1) correctly. This, Ewald summation method, which split this summation into fast-decaying parts, is required.

2 Algorithm description

2.1 Ewald summation for Stokes

Applying the idea of Ewald decomposition, the Eq.(1.2) can be split as following:

$$\mathbf{u}(\mathbf{x}_m) = \sum_{n=1}^{N} \sum_{\mathbf{p}} \mathbf{A}(\xi, \mathbf{x_m} - \mathbf{x_n} + \mathbf{p})\mathbf{f}_n + \frac{1}{V} \sum_{\mathbf{k} \neq 0} \mathbf{B}(\xi, \mathbf{k}) e^{-k^2/4\xi^2} \sum_{n=1}^{N} \mathbf{f}_n e^{-i\mathbf{k}\cdot(\mathbf{x}_m - \mathbf{x}_n)} - \mathbf{u}_{\text{self}} \tag{2.1}$$

where $\mathbf{k} \in \{[2\pi k_i/L_i] : k_i \in \mathbb{Z}, i = 1, 2, 3\}, k = |\mathbf{k}|, V = L_x L_y L_z$ and ξ is a positive constant known as the Ewald parameter. From the Eq.(2.1) we can see that the velocity field has been split into three parts: one sum in real space ($\mathbf{u}^R$), one sum in frequency domain ($\mathbf{u}^F$), also know as k-space, and a self-contribution $\mathbf{u}_{\text{self}}$. There are many ways for this kind of splitting. Here we choose Hasimoto's formulation[2], which is reported better than Beenakker's [3].

From the formulation by Hasimoto, we have

$$\mathbf{A}(\xi, \mathbf{x}) = 2\left(\frac{\xi e^{-\xi^2 r^2}}{\sqrt{\pi} r^2} + \frac{\text{erfc}(\xi r)}{2r^3}\right)(r^2\mathbf{I} + \mathbf{x}\mathbf{x}) - \frac{4\xi}{\sqrt{\pi}} e^{-\xi^2 r^2}\mathbf{I} \tag{2.2}$$

with $r = |\mathbf{X}|$ and

$$B(\xi, \mathbf{k}) = 8\pi\left(1 + \frac{k^2}{4\xi^2}\right)\frac{1}{k^4}(k^2\mathbf{I} - \mathbf{k}\mathbf{k}) \tag{2.3}$$

and

$$\mathbf{u}_{\text{self}}(\mathbf{x}_m) = \frac{4\xi}{\sqrt{\pi}}\mathbf{f}_m \tag{2.4}$$

2.2 Numerical schemes

Since self-contribution part is easy to get, here we are going to discuss efficient scheme to compute real space and k-space summation.

2.2.1 Real space

For summation in real space, it now decays exponentially as $\exp(-r^2)$. Thus, it's reasonable to truncate it within p_∞ layers. Error would decays exponentially too, which would be discussed later in 2.4.1. Ewald parameter ξ would be key parameter to decide p_∞, which is shown in Eq.(2.20). The only trick here is that we could compute particle j's effect on particle i at the same time as particle i's effect on particle j since it's even kernel and $\mathbf{A}(\xi, \mathbf{x})$ is the same. This trick would help us save 50% computational cost.

2.3 Frequency space

In k-space, things are much more difficult. By observation, the formula

$$\mathbf{u}^F(\mathbf{x}_m) = \frac{1}{V}\sum_{\mathbf{k}\neq 0}\mathbf{B}(\xi,\mathbf{k})e^{-k^2/4\xi^2}\sum_{n=1}^{N}\mathbf{f}_n e^{-i\mathbf{k}\cdot(\mathbf{x}_m-\mathbf{x}_n)} \tag{2.5}$$

can be split into

$$\mathbf{u}^F(\mathbf{x}_m) = \frac{1}{V}\underbrace{\sum_{\mathbf{k}\neq 0}\underbrace{\mathbf{B}(\xi,\mathbf{k})e^{-k^2/4\xi^2}}_{\text{scaling}}\underbrace{\left(\sum_{n=1}^{N}\mathbf{f}_n e^{i\mathbf{k}\cdot\mathbf{x}_n}\right)}_{\text{NuFFT type 1}}e^{-i\mathbf{k}\cdot\mathbf{x}_m}}_{\text{NuFFT type 2}}, \tag{2.6}$$

which is composed by Non-uniform-FFT (NuFFT) (type 1) combined with scaling and another NuFFT (type 2). The computation of scaling is straightforward. Thus to compute this efficiently, we have to find an efficient way to compute NuFFT.

2.3.1 Nonuniform fast Fourier transform

The nonuniform discrete Fourier transform (NuFFT) of type 1 and 2 is defined as:

$$F(\mathbf{k}) = \frac{1}{N}\sum_{n=1}^{N} f_j e^{-i\mathbf{k}\cdot\mathbf{x}_n} \quad \text{(type 1)} \tag{2.7}$$

and

$$f(x_n) = \sum_{\mathbf{k}} F(\mathbf{k})e^{i\mathbf{k}\cdot\mathbf{x}_n} \quad \text{(type 2)} \tag{2.8}$$

Now we will show in the following paragraphs that these can be computed efficiently by **Gaussian Gridding and uniform FFT**.

Reformulation by Gaussian Gridding & FFT The basic idea here is to consider the point source as delta function and convolve it with Gaussian function to smooth it in the real space (it's named as **'Gaussian Gridding'**), which makes it possible to apply the uniform FFT. Then we conduct deconvolution in the frequency space by scaling (element-wise multiplication) and get the finally result.The details are shown below.

For NuFFT of type 1, introducing a free parameter η, we have that

$$\sum_{n=1}^{N}\mathbf{f}_n e^{i\mathbf{k}\cdot\mathbf{x}_m} = e^{\eta k^2/8\xi^2}\sum_{n=1}^{N}\mathbf{f}_n e^{-\eta k^2/8\xi^2}e^{-i(-\mathbf{k})\cdot\mathbf{x}_m} := e^{\eta k^2/8\xi^2}\hat{H}_{-\mathbf{k}} \tag{2.9}$$

while $\hat{H}_{\mathbf{k}}$ is the Fourier transform of

$$H(\mathbf{x}) = \left(\frac{2\xi^2}{\pi\eta}\right)^{3/2} \sum_{n=1}^{N} \mathbf{f}_n e^{-2\xi^2|\mathbf{x}-\mathbf{x}_n|_*^2/\eta} \quad \text{(Gaussian gridding type 1)} \tag{2.10}$$

where $|\cdot|_*$ denotes distance to closest periodic image.

For the part of type 2 NuFFT, under the same parameter η, first we can simplify

$$\mathbf{u}^F(\mathbf{x}_m) = \frac{1}{V}\sum_{\mathbf{k}\neq 0} \mathbf{B}(\xi,\mathbf{k}) e^{-k^2/4\xi^2} e^{\eta k^2/8\xi^2} \hat{H}_{-\mathbf{k}} e^{-i\mathbf{k}\cdot\mathbf{x}_m} \tag{2.11}$$

$$= \frac{1}{V}\sum_{\mathbf{k}\neq 0} \hat{\hat{H}}_{\mathbf{k}} e^{-\eta k^2/8\xi^2} e^{i\mathbf{k}\cdot\mathbf{x}_m} \tag{2.12}$$

with

$$\hat{\hat{H}}_{\mathbf{k}} := \mathbf{B}(\xi,-\mathbf{k})\hat{H}_{\mathbf{k}} e^{-(1-\eta)k^2/4\xi^2} \tag{2.13}$$

Then we denote $\tilde{H}(\mathbf{x}_i)$ as the inverse Fourier transform of $\hat{\hat{H}}_{\mathbf{k}}$.

According to the convolution argument, we have that

$$\mathbf{u}^F(\mathbf{x}_m) = \left(\frac{2\xi^2}{\pi\eta}\right)^{3/2} \int_\Omega \tilde{H}(\mathbf{x}) e^{-2\xi^2|\mathbf{x}-\mathbf{x}_m|_*^2/\eta} \mathbf{dx} \tag{2.14}$$

$$\approx \frac{V}{N_{\text{grid}}}\left(\frac{2\xi^2}{\pi\eta}\right)^{3/2} \sum_{\mathbf{x}_{(i)} \text{ in equi-space grid}} \tilde{H}(\mathbf{x}_{(i)}) e^{-2\xi^2|\mathbf{x}_{(i)}-\mathbf{x}_m|_*^2/\eta} \tag{2.15}$$

The summation (Gaussian gridding type 2) to approximate integral is spectrally accurate since the integral is periodic.

Since the Gaussian function has the infinite support, we have to truncate the Gaussian function to get the numerical value in Eq.(2.10) and Eq.(2.15). Denote the grid size of real space (in one dimension) as h and number of grids in the support of truncated Gaussian as P, then the value of Gaussian at the truncated points is $e^{-\xi^2 P^2 h^2/2\eta}$. We choose the free parameter η as

$$\eta = \left(\frac{Ph\xi}{m}\right)^2 \tag{2.16}$$

then we know value of the Gaussian kernel at the truncated points is $e^{-m^2/2}$. Later we will see the advantage of choosing η like this is then the error from Eq.(2.10) and Eq.(2.15) is decoupled with Ewald parameter ξ, which makes it easier to determine all the free parameters.

Now the only left problem is how to compute Eq.(2.10) and Eq.(2.15) in high efficency, which are known as **Fast Gaussian gridding**.

Fast Gaussian Gridding Denote N as the number of particles and M the number of layers in k-space, then if we compute Eq.(2.10) and Eq.(2.15) directly, the computation cost (each) is NM^3 exponential computation, $3NM^3$ multiplications and additions. Since exponential takes much more time than multiplication ($15 \sim 20$ time clocks than $1 \sim 2$ time clocks), a natural idea is to substitute exponential operation with multiplication. **The idea of fast Gaussian gridding is to pre-compute and store the exponential and only do necessary multiplications afterwards**, which is described in detail in Greengard's paper[1].

Roughly speaking, taking 1D situation for example, we have that ($x_{(i)} = ih, x_n = \tilde{x}_n + jh, 0 \leq \tilde{x}_n < h$)

$$e^{-\alpha|x_{(i)}-x_n|^2} = e^{-\alpha|ih-jh-\tilde{x}_n|^2} = e^{-\alpha((i-j)h)^2}\left(e^{2\alpha h\tilde{x}_n}\right)^i e^{-\alpha\tilde{x}_n^2} \tag{2.17}$$

All $e^{-\alpha((i-j)h)^2}, e^{2\alpha h\tilde{x}_n}, e^{-\alpha\tilde{x}_n^2}$ can be precomputed with only $\mathbf{2(M+N)}$ exponential computations (also that much storage cost) and then we only need to do $2NM$ more multiplications to get all $e^{-\alpha|x_{(i)}-x_n|^2}$. For 3D situation we need $\mathbf{6(M+N)}$ exponential computations and $2NM^3$ more multiplications. **As long as the exponential takes significantly more time than multiplication, the fast Gaussian gridding will improve the performance and it will benefit more with higher dimension.**

2.4 Error and complexity analysis

By analyzing error bound and complexity, we could have a clear view on how much speed-up we get from this scheme and how to choose parameters with trade-off between accuracy and computing cost.

2.4.1 Error bound

The overall error comes from three parts: 1) the truncation error in the k-space; 2) the quadrature error in the k-space (Eq.(2.15)); 3) the truncation error in the real space.

Given the parameters as

- L: the length of grid in each dimension
- ξ : Ewald parameter
- M: Number of layers of k-space
- p_∞: Number of layers of real-space
- P, m: Parameter of Gaussian gridding (Eq.(2.16))

we have the following estimation of error bound [3]:

$$E = E^F + E^Q + E^R \tag{2.18}$$

$$E^F \le C_F e^{-\frac{M^2\pi^2}{4L^2\xi^2}} \quad \text{(truncation of k-space)} \tag{2.19}$$

$$E^R \le C_R\left(\frac{1}{\xi^2} + \frac{p_\infty}{\xi}\right)e^{-p_\infty^2\xi^2}. \quad \text{(truncation of real space)} \tag{2.20}$$

$$E^Q \le 4e^{-\frac{\pi^2P^2}{2m^2L^2}} + \text{erfc}(m/\sqrt{2}) \quad \text{(quadrature)} \tag{2.21}$$

We can see that all three parts of error decay exponentially with the corresponding parameter (E^F with M, E^Q with P and E^R with p_∞). It's also been shown numerically.

Besides, we can see that **Ewald parameter ξ controls the trade off between real space and k-space.** Specifically, with larger ξ, we will need larger M to get the same accuracy, but smaller p_∞ is needed. With smaller ξ, we need smaller M but larger p_∞. In practice, we can choose ξ to balance the work load between the real space and the k-space.

Note that by choosing the free parameter η as Eq.(2.16), the quadrature error E^Q is independent with the Ewald parameter ξ, which makes it easier to determine the parameters.

2.5 Time complexity

The total computational cost of the method is

$$\underbrace{O(NP^3)}_{\text{Gaussian gridding}} + \underbrace{O(M^3\log(M^3))}_{\text{FFT + iFFT}} + \underbrace{O(M^3)}_{\text{Scaling on k-space}} + \underbrace{O(N^2p_\infty^3)}_{\text{Real space}} + \underbrace{O(N)}_{self}. \tag{2.22}$$

Since P can be regarded as an accuracy parameter, which is independent of the mesh and the number of source points, the complexity of the method could be stated as

$$O(N) + O(M^3\log(M^3)) + O(N^2p_\infty^3). \tag{2.23}$$

If we don't use the NuFFT method (computing k-space by Eq.(2.5)), the total cost is

$$\underbrace{O(N^2M^3)}_{k-space} + \underbrace{O(N^2p_\infty^3)}_{\text{Real space}} + \underbrace{O(N)}_{self} \tag{2.24}$$

which is a sharp contrast.

To verify this theoretically, we change the size of the problem (the number of particles

N) and see how the time of each part changes. Notice that when increasing the number of particles, we increase the number of layers in k-space $M^3 \sim N$ (to keep the same accuracy). The dashed line is the fit for the time ($O(N^2)$ for Real space, $O(N)$ for Gaussian Gridding and $O(N \log N)$ for FFT), which verifies the theoretic time complexity.

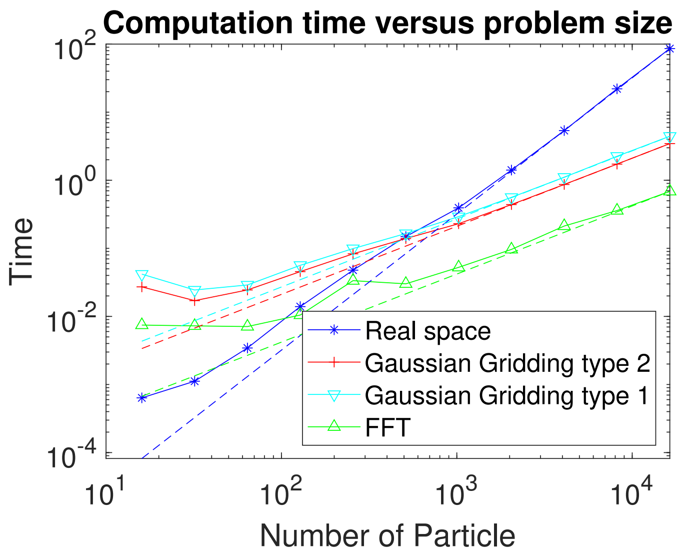

Figure 1: The plot of the time cost of each part versus the size of the problem (the number of particles N). When increasing the number of particles, we increase the number of layers in k-space $M^3 \sim N$. The dashed line is the fit for the time ($O(N^2)$ for Real space, $O(N)$ for Gaussian Gridding and $O(N \log N)$ for FFT). It verifies the theoretic time complexity.

2.6 Parallelization

Basically, we use GPU-based FFT library cuFFT to parallelize FFT and cpu-based openMP to parallelize real space and gaussian griding.

2.6.1 Real space

In real space, computation of each particle's velocity is independent with each other, which is ideal to parallelize the for-loop using shared memory OpenMP[6].

2.6.2 Gaussian Griding

We also use openMP for Gaussian Griding.

The parallelization of Eq.(2.15) (Gaussian gridding type 2) is simple since we can compute the velocity filed $\mathbf{u}^F(\mathbf{x}_m)$ at each point source $\mathbf{x}_m$ separately , thus parallelly.

Since we usually have a large number of source points and there's no communication between threads, it's a parallelization with high-efficiency.

However, the parallelization of Eq.(2.10) is not that ideal since (almost) every source point contributes to the grid points and it will need great memory cost if we still parallelize on the source points. Thus *instead we parallelize on the grid points to avoid the repeating IO for grid points.* To be more specific, for each source point we parallelize on the grid points that will be affected by the source (since we use a truncated Gaussian) and do the computation serially on the source points. *Since each source point affects only a limited number of grid points (actually P points in one dimension),* we will see later it's not as efficient as the parallelization of Eq.(2.15), but it did reduce the computation time with more cores.

2.6.3 cuFFT

Here we use cuda version of FFT on GPU called cuFFT[5] to parallelize FFT. Basically, for large problem size, it would reach a speed-up up to 10 for 2D FFT and up to 3-4 for 3D FFT compared with serial FFTW[4].

2.7 Conclusion

Generally, we should choose big ξ so that p_∞ for real space is small and computational cost is small for real space. For k-space, since we use fft and gaussian gridding, which reduces greatly the complexity, large ξ is affordable.

The Description of Algorithm The algorithm is conducted in the following way:

(1) Choose the free parameter ξ and η wisely and construct the uniform grid according to the problem.

(2) Evaluate $H(\mathbf{x})$ on the grid according to Eq.(2.10) with fast Gaussian gridding.

(3) Conduct FFT on $H(\mathbf{x})$ to get $\hat{H}_\mathbf{k}$.

(4) Apply the scaling according to Eq.(2.13) to get $\hat{\tilde{H}}_\mathbf{k}$.

(5) Conduct iFFT on $\hat{\tilde{H}}_\mathbf{k}$ to get $\tilde{H}(\mathbf{x})$.

(6) Evaluate the $\mathbf{u}^F(\mathbf{x}_m)$ according to Eq.(2.15) with fast Gaussian gridding.

(7) Evaluate the $\mathbf{u}^R(\mathbf{x}_m)$ in real space and get the final result.

3 Numerical results

3.1 Convergence

To test our method's convergence, which is supposed to be exponential, we examine real space and k-space seperately.

a. Real space

As Eq.(2.20) shows, truncation error of real space would decays exponentially to p_∞. We set $L = 1,\ N = 100,\ \mathbf{f} = [1, 1, 1]$ and change $p_\infty = [2, 4, 6, 8, 10],\ \xi = [0.5, 0.8, 1.0]$, and the successive error is shown in fig.2. We see clear exponential decay and well-match with theory. Moreover, ξ serves well to adjusting error.

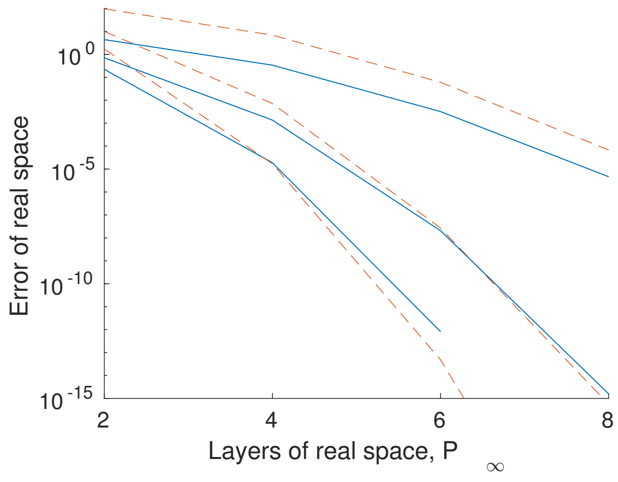

Figure 2: Error of real space vs p_∞, $L = 1,\ N = 100,\ \mathbf{f} = [1, 1, 1]$, blue solid line is numerical successive error, red dashed line is theoretical estimated error bound from Eq.(2.20). From above to bottom, three lines are accordingly $\xi = [0.5, 0.8, 1.0]$. We see clear exponential decay and well-match with theory

a. k-space

As Eq.(2.19) and Eq.(2.21) show, error of k-space would decays exponentially to P and M. We set $L = 1,\ ,m = 20,\ N = 100,\ \mathbf{f} = [1, 1, 1]$ and change $P = M = [24, 34, 44, 54, 64],\ \xi = [5, 10, 15]$, and the successive error is shown in fig.3. We see clear exponential decay and well-match with theory. Moreover, ξ serves well to adjusting error. The larger ξ is, the greater the error of k-space is.

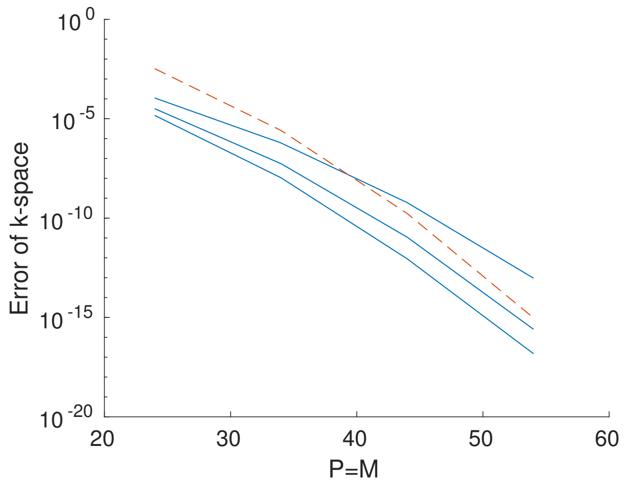

Figure 3: Error of k-space vs $P = M,\ L = 1,\ ,m = 20,\ N = 100,\ \mathbf{f} = [1, 1, 1]$, blue solid line is numerical successive error, red dashed line is theoretical estimated error bound from Eq.(2.19) and Eq.(2.21). From bottom to above, three lines are accordingly $\xi = [5, 10, 150]$. We see clear exponential decay and well-match with theory

3.2 Speed up

The result in this part is run on a Prince server with 8 CPU cores and one GPU (P40).

To see how the parallelization of real space and Gaussian gridding and CuFFT performs, we compare the run time of 1 thread with 8 threads and compute the speed up under different problem sizes. The result is shown in Fig.4. We can see that with increasing problem size, the parallelization of real space shows the idealistic $8\times$ speed up, while Gaussian Gridding shows $\sim 7\times$ speed up and type 2 performs better than type 1. For FFT, with a large enough problem size M^3, CuFFT achieves $2 \sim 4\times$ speed up than serial FFTW[4].

3.3 Scalability

The result in this part is run on a Prince server with 32 CPU cores.

Now we focus on the scalability (scaling efficiency) of our method, which indicates how efficient our method is with increasing numbers of parallel processing threads. Here we focus on the real space and Gaussian gridding part.

To verify it numerically, we run the program under the setting of 8192 particles, which usually takes 3 minutes overall for one thread. The strong scaling efficiency of each part is shown in Fig.5. For large problem size, the result is similar to Fig.5. When problem size

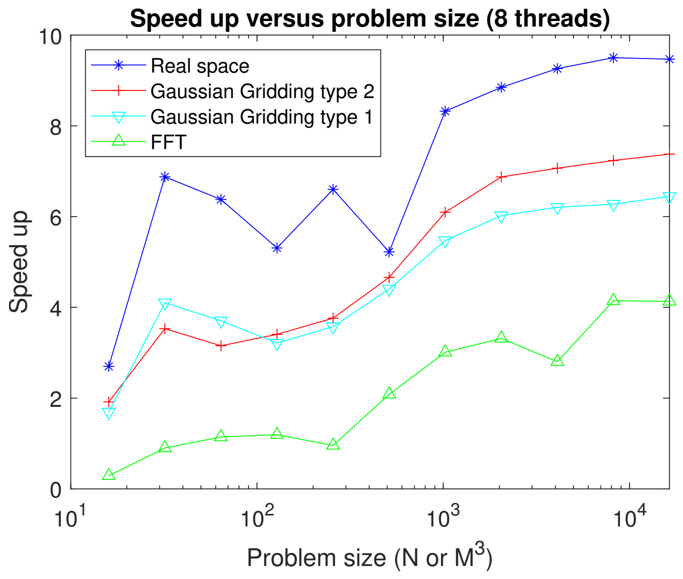

Figure 4: The plot of the speed up (with 8 threads) of each part versus the size of the problem (the number of particles N). With increasing problem size, the parallelization of real space shows the idealistic $8\times$ speed up, while Gaussian Gridding shows $\sim 7\times$ speed up and type 2 performs better than type 1. For FFT, with a large enough problem size M^3, CuFFT achieves $2 \sim 4\times$ speed up than serial FFTW.

is too small, (like less than 128 particles), the scalability is not good, which is reasonable since then the communication cost dominates instead of the computation.

We can see that with increasing number of threads N, the parallelization of real space shows the perfect strong-scaling efficiency (~ 1) when $N \leq 16$ and drops a little when N is large. For Gaussian Gridding (type 2), it drops to almost 80%, which is still not bad. However, for Gaussian Gridding (type 1), the scalability decays to almost 50% when $N > 8$, which partly due to *we can only parallelize it on the grid points instead of source points and thus involves more communication and parallelization overhead.*

For the weak scaling, the problem size assigned to each processing threads keeps the same and with increasing threads we can solve a larger size of problem. Denote the amount of time to complete a work unit with 1 processing element is t_1 and the amount of time to complete N of the same work with N threads is t_N, the weak scaling efficiency is defined as:

$$p_N = \frac{t_1}{t_N} \tag{3.1}$$

The weak scaling efficiency of each part is shown in Fig.6.

We can see that the parallelization of real space shows the good weak-scaling effi-

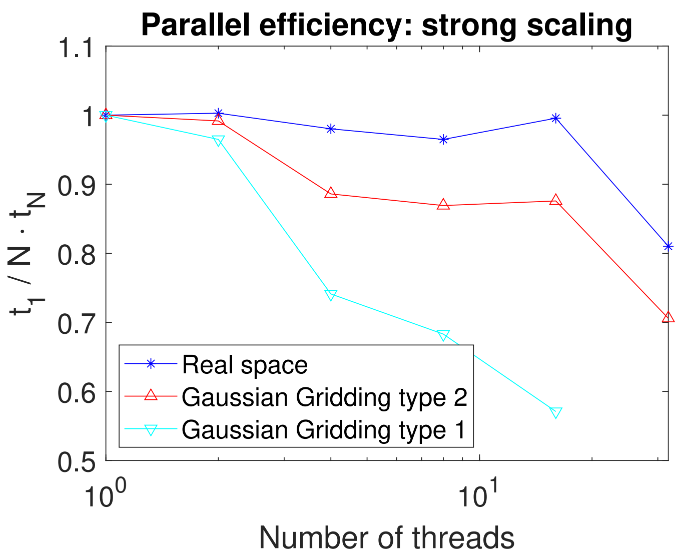

Figure 5: The plot of the strong scaling efficiency of each part versus the number if threads. With increasing number of threads N, the parallelization of real space shows the good strong-scaling efficiency ($\sim 90\%$) while Gaussian Gridding (type 2) drops to almost 80%. For Gaussian Gridding (type 1), it's worse than type 2 and the scalability drops to almost 50% when $N > 8$.

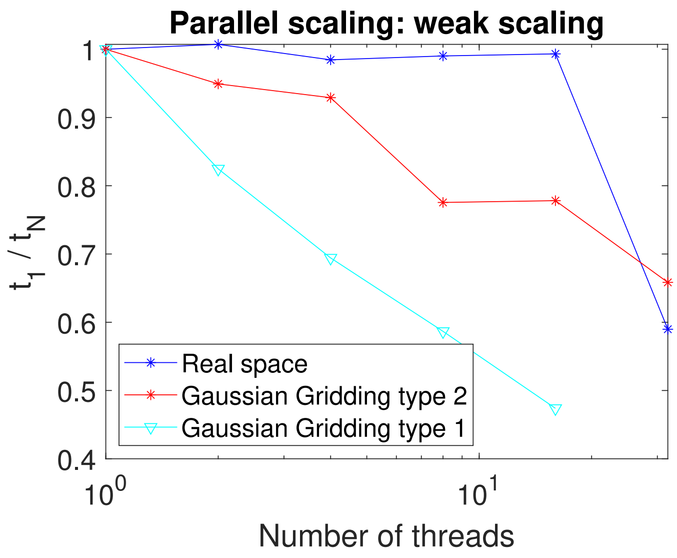

Figure 6: The plot of the weak scaling efficiency of each part versus the number if threads. With increasing number of threads N, the parallelization of real space shows the good weak-scaling efficiency when $N \leq 16$ but drops when $N = 32$, while Gaussian Gridding (type 2) also drops to almost 60%. For Gaussian Gridding (type 1), the scalability drops to almost 50%.

ciency when $N \leq 16$ but drops a little when $N = 32$, while Gaussian Gridding (type 2) also decays to almost 60%. For Gaussian Gridding (type 1), the scalability monotonously drops to 50% with more and more threads. All three methods are not so perfect *since they all involve global communication patterns to some extent, which increases the communication overhead when increasing the number of threads (also the size of the problem), especially for the type 1 Gaussian gridding since the paralleilzation is not that efficient.*

4 Additional Information about the project

4.1 Project info

- Link of the project:

  https://github.com/CecilMartin/Ewald_Summation

- Team work:
  - Zhe Chen: Write codes, Convergence test, presentation
  - Guanchun Li: Develop algorithm into pseudo-code, run codes, write slides
  - Together: Literature research, write reports

4.2 Experience and thinking of this project

From the numerical perspective, we learn a lot in this fast algorithm Ewald method, which is new to both of us. There are many subtle tricks to implement a faster numerical scheme. For a complicated method, it's important to figure the meaning of each free parameters and how it affects the accuracy and efficiency of the method, especially when trying to reproduce the result of the others' works. We should pay attention to error estimation and trade-off between accuracy and computational cost.

From the HPC perspective, we learn how to use OpemMP, cufft, fftw. It's much easier to learn only the idea of those parallelizations we used. However, there is much more to learn when you implement it by your own hands, most of them are subtle things easy to be ignored and only can be noticed through practice. For example, "nvcc" does some weird thing as it pretends to cover function of "gcc", especially when linked with fftw3 library. We need to hack into fftw3 library to solve this problem. It's tricky to compile a code that is both gpu and cpu parallelized. And we should notice private and shared variables of openMP. Moreover, we learn a lesson that timing a code is not easy at all, especially for parallel code. Not only timing accuracy is important, one should also notice the difference of host time and device time. In particular, one should not use "clock()" to timing a openMP block since it will add all threads' time cost in total, which gives you no way to see scaling of OpenMp code, etc... Things like these are often not possible to cover in a course, but only to learn by practice in something like final project. Besides, when designing and implementing the paralleled algorithm, the communication cost can't be neglected. A method performs well in serial running doesn't guarantee the performance in parallelization, nevertheless the scalability. We need to modify the algorithm and codes to balance the cost of computation and communication.

References

[1] Greengard, Leslie, and June-Yub Lee. "Accelerating the nonuniform fast Fourier transform." SIAM review 46.3 (2004): 443-454.

[2] Hasimoto, Hidenori. "On the periodic fundamental solutions of the Stokes equations and their application to viscous flow past a cubic array of spheres." Journal of Fluid Mechanics 5.2 (1959): 317-328.

[3] Lindbo, Dag, and Anna-Karin Tornberg. "Spectrally accurate fast summation for periodic Stokes potentials." Journal of Computational Physics 229.23 (2010): 8994-9010.

[4] Frigo, Matteo, and Steven G. Johnson. "The design and implementation of FFTW3." Proceedings of the IEEE 93.2 (2005): 216-231.

[5] https://developer.nvidia.com/cufft

[6] Dagum, Leonardo, and Ramesh Menon. "OpenMP: An industry-standard API for shared-memory programming." Computing in Science & Engineering 1 (1998): 46-55.