Практикум по курсу
“Суперкомпьютеры и параллельная обработка данных”

Разработка параллельной версии программы с реализацией метода конечных областей во временной области для уравнений в двумерной области на языке C.

ОТЧЕТ

о выполненном задании

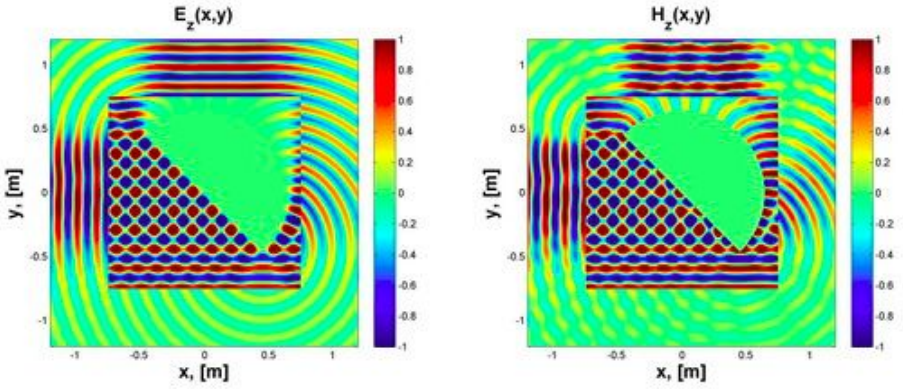
студента 327 группы ВМК МГУ
Мотриченко Дмитрия Олеговича

Обзор задания

Целью данного задания было усовершенствование реализации метода конечных областей во временной области для уравнений в двумерной области на языке C. В ходе выполнения задания были получены две реализации данного метода – на OpenMP(«fdtd-2d.c») и на MPI(«fdtd_mpi.c»).

Описание задачи

Метод FDTD относится к общему классу сеточных методов решения дифференциальных уравнений. В уравнениях Максвелла изменение электрического поля E (частная производная) зависит от распределения в пространстве магнитного поля H (ротор). Аналогично, изменение поля H зависит от распределения в пространстве поля E. Ниже приведена иллюстрация двумерного случая, рассматриваемого в задании:



Данный алгоритм представляет собой последовательное обновление значений матриц электрического и магнитного полей в соответствии с уравнениями Максвелла:

$$-\mu \frac{\partial H_x}{\partial t} = \frac{\partial E_z}{\partial y},$$

$$\mu \frac{\partial H_y}{\partial t} = \frac{\partial E_z}{\partial x},$$

$$\frac{\partial E_z}{\partial t} = \frac{1}{\varepsilon}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma E_z\right)$$

Для элементов матриц данные уравнения принимают вид:

$$x_i = i\Delta x,\ x_{i+\frac{1}{2}} = x_i + \frac{1}{2}\Delta x,\ i = 0,1,2,...,I-1,\ x_I = a,$$
$$y_j = j\Delta y,\ y_{j+\frac{1}{2}} = y_j + \frac{1}{2}\Delta y,\ j = 0,1,2,...,J-1,\ y_J = b,$$
$$t^n = n\Delta t,\ t^{n+\frac{1}{2}} = t^n + \frac{1}{2}\Delta t,\ n = 0,1,2,...,N-1,\ N\Delta t = T,$$

Реализация программ

Обе реализации алгоритма основаны на разделении между потоками матриц, которые требуется обновить.

В случае с OpenMP разделение происходит автоматически с помощью директив

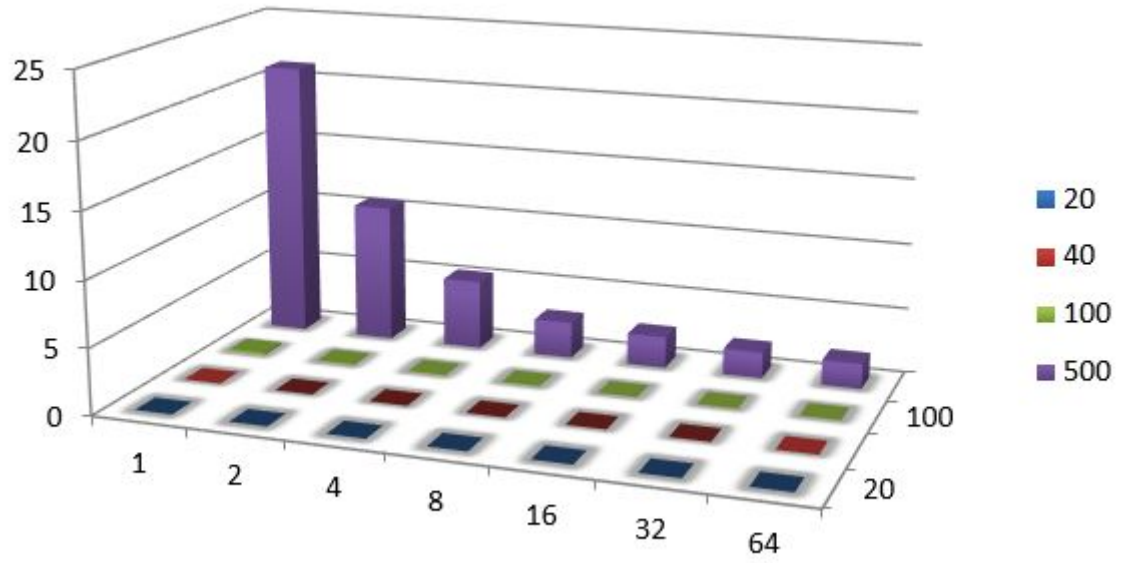
#pragma omp parallel for
и
#pragma omp parallel for collapse(2) .

В случае с MPI матрицы делятся построчно в зависимости от числа потоков. Затем поток 0 отправляет всем остальным потокам необходимые части матриц, и после вычислений принимает обновленные части.

Результаты работы программ и их масштабируемость

Ниже приводятся данные об измерении скорости работы реализованных алгоритмов на вычислительном комплексе IBM Polus и соответствующие графики.

График результатов работы параллельного алгоритма, реализованного с помощью библиотеки OpenMP:



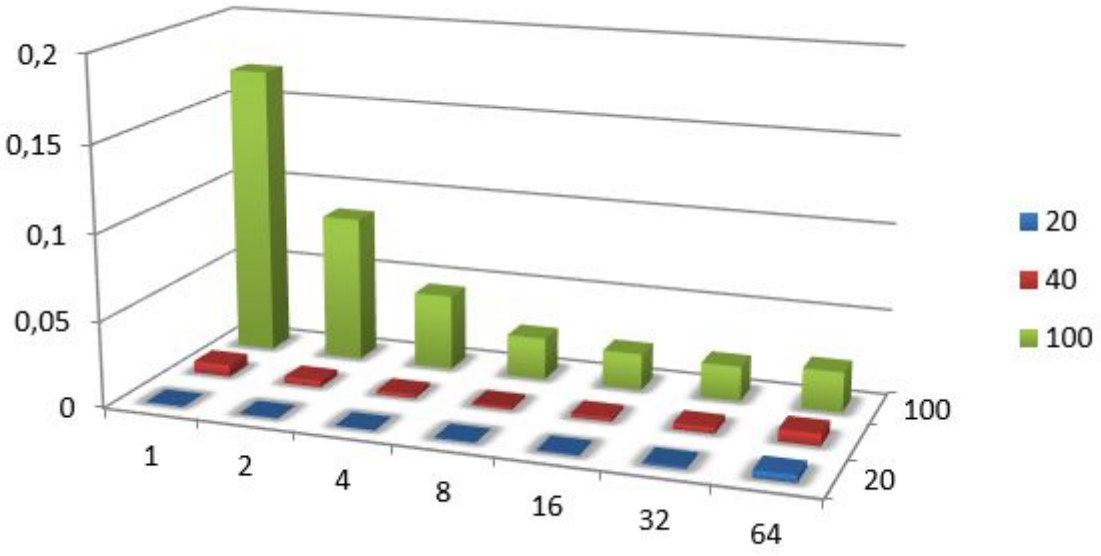
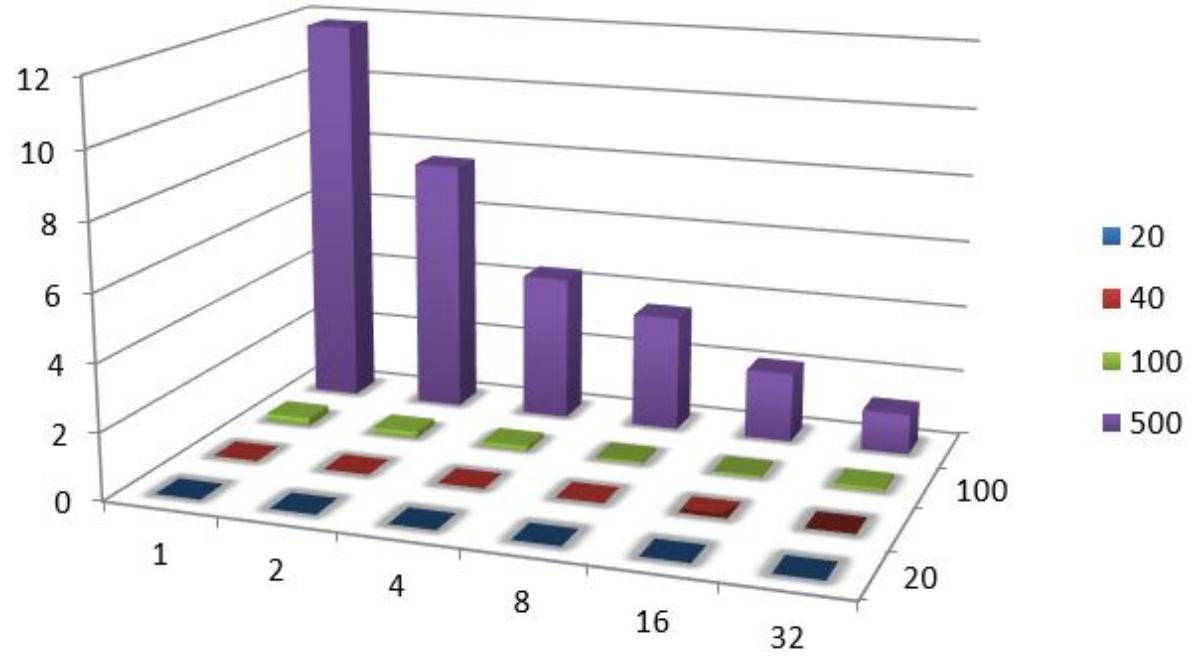
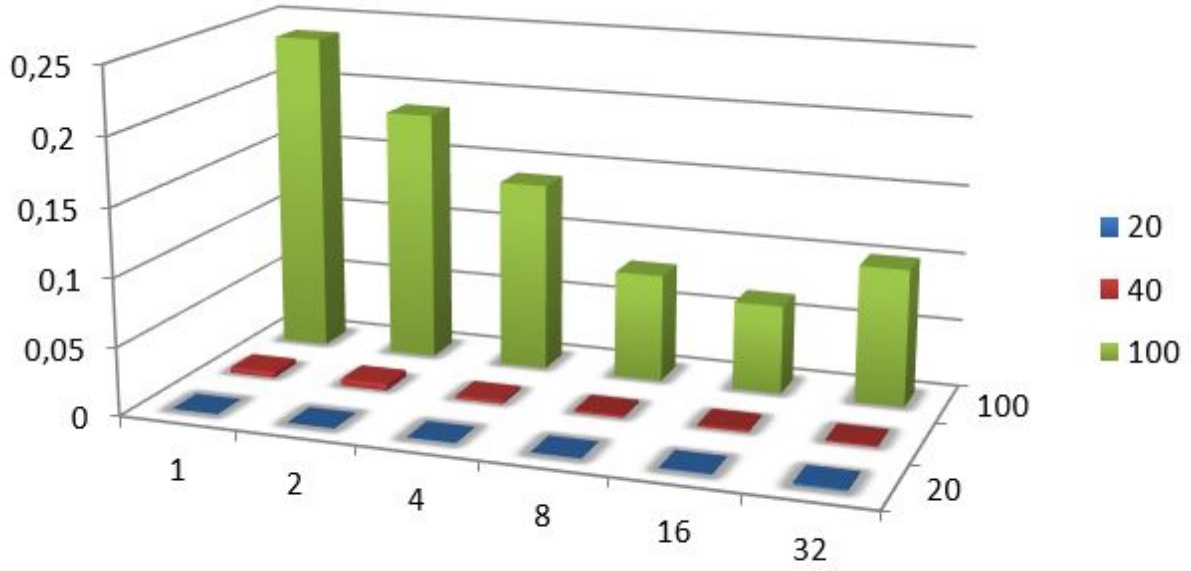


График результатов работы параллельного алгоритма, реализованного с помощью библиотеки MPI:





Вывод

Таким образом, анализируя полученные графики, можно отметить, что реализация алгоритма с помощью OpenMP в большинстве случаев работает быстрее реализации на MPI. Также стоит отметить, что при использовании программ на данных большого размера, то есть на максимально приближенных к реальным данным, время работы алгоритма уменьшается на порядок по сравнению со временем работы его не

параллельной версии.

Данные измерений

Результаты работы параллельного алгоритма, реализованного с помощью библиотеки OpenMP:

MINI_DATASET:

1 thread:
Time in seconds = 0.000478
2 thread:
Time in seconds = 0.000386
4 thread:
Time in seconds = 0.000419
8 thread:
Time in seconds = 0.000635
16 thread:
Time in seconds = 0.000833
32 thread:
Time in seconds = 0.001425
64 thread:
Time in seconds = 0.004161
128 thread:
Time in seconds = 0.235966
160 thread:
Time in seconds = 0.238254

SMALL_DATASET:

1 thread:
Time in seconds = 0.006877
2 thread:
Time in seconds = 0.003706
4 thread:
Time in seconds = 0.002230
8 thread:
Time in seconds = 0.001696
16 thread:
Time in seconds = 0.002183
32 thread:
Time in seconds = 0.003340
64 thread:
Time in seconds = 0.006984

128 thread:
Time in seconds = 0.487337
160 thread:
Time in seconds = 0.503410

MEDIUM_DATASET:

1 thread:
Time in seconds = 0.169514
2 thread:
Time in seconds = 0.085252
4 thread:
Time in seconds = 0.043954
8 thread:
Time in seconds = 0.024331
16 thread:
Time in seconds = 0.021206
32 thread:
Time in seconds = 0.020176
64 thread:
Time in seconds = 0.023357
128 thread:
Time in seconds = 1.237062
160 thread:
Time in seconds = 1.304231

LARGE_DATASET:

1 thread:
Time in seconds = 21.191872
2 thread:
Time in seconds = 10.592099
4 thread:
Time in seconds = 5.346072
8 thread:
Time in seconds = 2.790215
16 thread:
Time in seconds = 2.421704
32 thread:
Time in seconds = 2.011578
64 thread:
Time in seconds = 1.930073
128 thread:
Time in seconds = 1.493244
160 thread:
Time in seconds = 1.527564

EXTRALARGE_DATASET:

1 thread:
Time in seconds = 190.291178
2 thread:
Time in seconds = 235.452396
4 thread:
Time in seconds = 46.620240
8 thread:
Time in seconds = 26.431466
16 thread:
Time in seconds = 21.904601
32 thread:
Time in seconds = 22.094486
64 thread:
Time in seconds = 14.595478
128 thread:
Time in seconds = 9.826704
160 thread:
Time in seconds = 8.116231

Результаты работы параллельного алгоритма, реализованного с помощью библиотеки MPI:

MINI_DATASET:

1 thread:
Time in seconds = 0.000127
2 thread:
Time in seconds = 0.000313
4 thread:
Time in seconds = 0.000397
8 thread:
Time in seconds = 0.000454
16 thread:
Time in seconds = 0.000901
32 thread:
Time in seconds = 0.001428

SMALL_DATASET:

1 thread:
Time in seconds = 0.004053
2 thread:
Time in seconds = 0.004951
4 thread:
Time in seconds = 0.002386
8 thread:
Time in seconds = 0.001989

16 thread:
Time in seconds = 0.001832
32 thread:
Time in seconds = 0.001445

MEDIUM_DATASET:

1 thread:
Time in seconds = 0.237467
2 thread:
Time in seconds = 0.185609
4 thread:
Time in seconds = 0.139447
8 thread:
Time in seconds = 0.079453
16 thread:
Time in seconds = 0.065244
32 thread:
Time in seconds = 0.040751

LARGE_DATASET:

1 thread:
Time in seconds = 11.786230
2 thread:
Time in seconds = 7.647928
4 thread:
Time in seconds = 4.373725
8 thread:
Time in seconds = 4.486958
16 thread:
Time in seconds = 2.100764
32 thread:
Time in seconds = 1.618721