Общий отчёт по СКМиТ.

Математическая постановка задачи:

Требуется написать программы для решения уравнения в частных производных

$$\frac{\partial^2 u}{\partial t^2} = a^2 \Delta u$$

$$u|_{t=0} = \varphi(x, y, z),$$
$$\left.\frac{\partial u}{\partial t}\right|_{t=0} = 0,$$

При следующих граничных условиях

6	П	1Р	П	$\sin(\frac{2\pi}{L_x}x) \cdot \sin(\frac{\pi}{L_y}y + \pi) \cdot \sin(\frac{2\pi}{L_z}z + 2\pi) \cdot \cos(a_t \cdot t + \pi), a_t = \frac{\pi}{3}\sqrt{\frac{4}{L_x^2} + \frac{1}{L_y^2} + \frac{4}{L_z^2}}, a^2 = \frac{1}{9}$

Методы численного решения:

Для численного решения на первом шаге используется следующая схема:

$$\frac{u^1_{ijk} - u^0_{ijk}}{\tau} = a^2 \frac{\tau}{2} \Delta_h \varphi(x_i, y_j, z_k), \quad (x_i, y_j, z_k) \in \omega_h.$$

$$u^1_{ijk} = u^0_{ijk} + a^2 \frac{\tau^2}{2} \Delta_h \varphi(x_i, y_j, z_k)$$

На следующих шагах используется схема, использующая два прошлых момента времени:

$$\frac{u^{n+1}_{ijk} - 2u^n_{ijk} + u^{n-1}_{ijk}}{\tau^2} = a^2 \Delta_h u^n, \quad (x_i, y_j, z_k) \in \omega_h, \quad n = 1, 2, \ldots, K-1,$$

Здесь Δ_h — семиточечный разностный аналог оператора Лапласа:

$$\Delta_h u^n = \frac{u^n_{i-1,j,k} - 2u^n_{i,j,k} + u^n_{i+1,j,k}}{h^2} + \frac{u^n_{i,j-1,k} - 2u^n_{i,j,k} + u^n_{i,j+1,k}}{h^2} + \frac{u^n_{i,j,k-1} - 2u^n_{i,j,k} + u^n_{i,j,k+1}}{h^2}.$$

Разностная аппроксимация для периодических граничных условий выглядит следующим образом

$$u^{n+1}_{0jk} = u^{n+1}_{Njk}, \qquad u^{n+1}_{1jk} = u^{n+1}_{N+1jk},$$
$$u^{n+1}_{i0k} = u^{n+1}_{iNk}, \qquad u^{n+1}_{i1k} = u^{n+1}_{iN+1k},$$
$$u^{n+1}_{ij0} = u^{n+1}_{ijN}, \qquad u^{n+1}_{ij1} = u^{n+1}_{ijN+1},$$

$i, j, k = 0, 1, \ldots, N.$

Краткое описание проделанной работы:

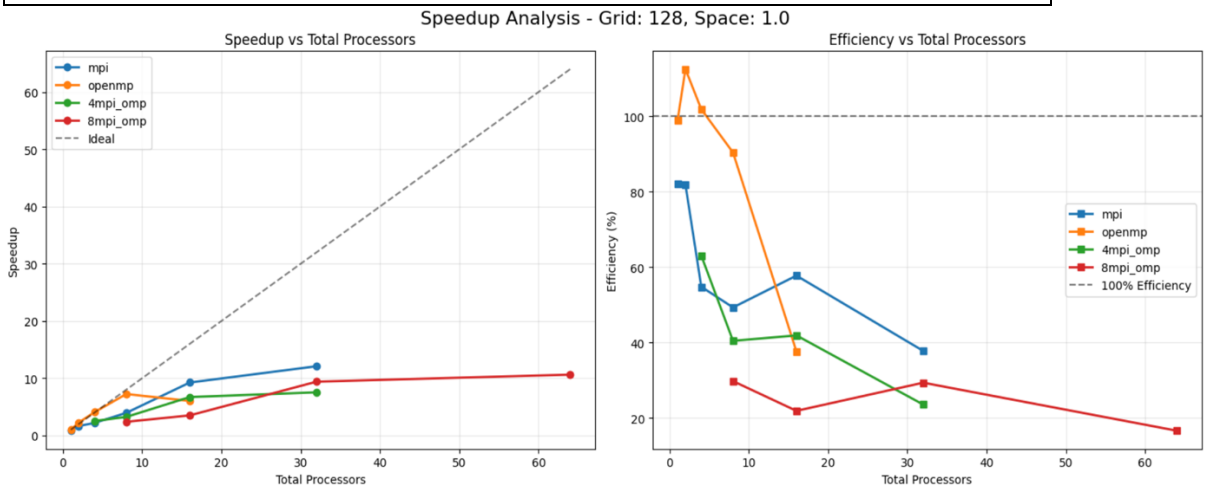
Были реализованы следующие вариации решения:
- Последовательное без параллелизма,
- OpenMP для параллелизма в циклах,
- MPI для пространственного параллелизма, где пространство бьётся на блоки по всем трём направлениям (в зависимости от числа доступных процессоров),
- MPI+OpenMP как комбинация достоинств обоих прошлых решений,
- MPI+CUDA для достижения наибольшего ускорения.
- Две программы с припиской timing для более подробного замера времени (нужно для GPU задачи).

Проведены эксперименты как для примеров из основного условия (сетка 128x128x128 или 256x256x256), так и из требований к MPI+CUDA реализации (сетка 512x512x512).

Далее идут результаты экспериментов.

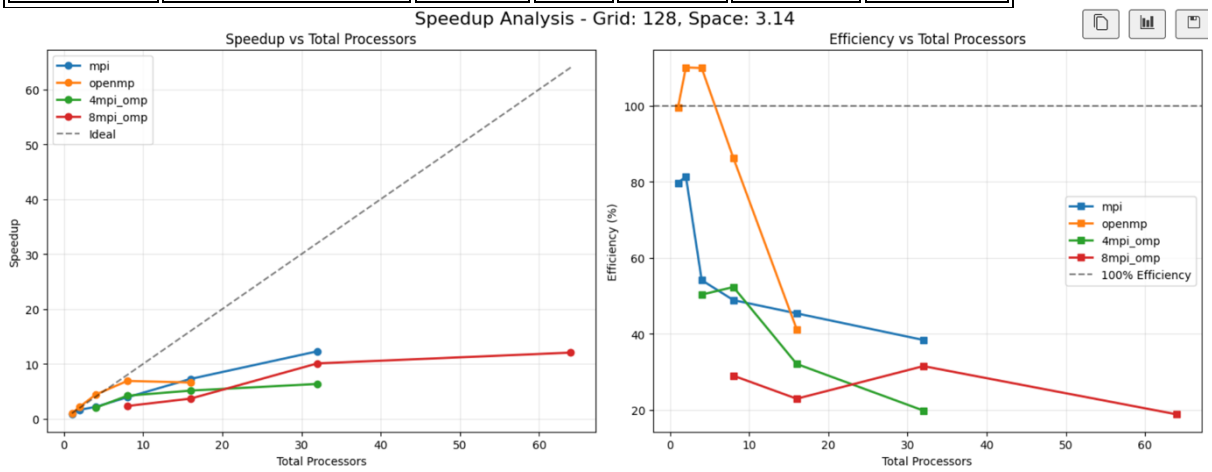
Сетка 128x128x128, L=1.0, T=1.0

Program	ompNumThreads	mpiSize	Time	Error	SpeedUp	Efficiency
no_parallel	1	1	6.295	0.00014	1.00	100.00%
openmp	1	1	6.465	0.00014	0.97	97.37%
openmp	2	1	2.841	0.00014	2.22	110.76%
openmp	4	1	1.569	0.00014	4.01	100.29%
openmp	8	1	0.884	0.00014	7.12	89.00%
openmp	16	1	1.062	0.00014	5.93	37.05%
mpi	1	1	7.798	0.00014	0.81	80.74%
mpi	1	2	3.905	0.00014	1.61	80.56%
mpi	1	4	2.918	0.00014	2.16	53.93%
mpi	1	8	1.621	0.00014	3.88	48.54%
mpi	1	16	0.692	0.00014	9.10	56.86%
mpi	1	32	0.529	0.00014	11.90	37.19%
mpi_omp	1	4	2.536	0.00014	2.48	62.04%
mpi_omp	2	4	1.976	0.00014	3.19	39.84%
mpi_omp	4	4	0.954	0.00014	6.60	41.24%
mpi_omp	8	4	0.849	0.00014	7.41	23.17%
mpi_omp	1	8	2.686	0.00014	2.34	29.29%
mpi_omp	2	8	1.827	0.00014	3.45	21.54%
mpi_omp	4	8	0.681	0.00014	9.24	28.88%
mpi_omp	8	8	0.602	0.00014	10.46	16.34%
mpi_cuda	1	1	0.127	0.00013	49.57	-



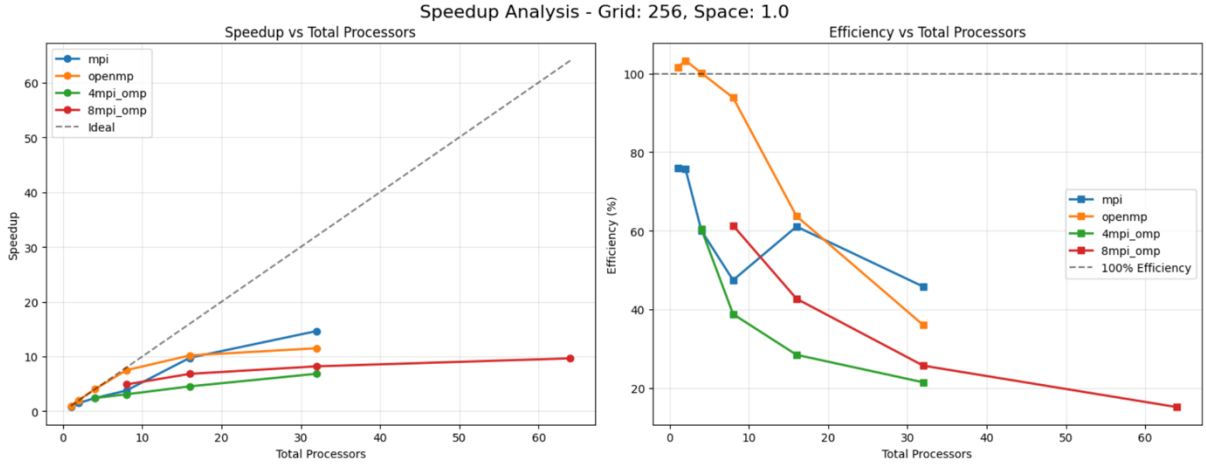
Сетка 128x128x128, L=π, T=π

Program	ompNumThreads	mpiSize	Time	Error	SpeedUp	Efficiency
no_parallel	1	1	6.053	0.00013	1.00	100.00%
openmp	1	1	6.283	0.00013	0.96	96.34%
openmp	2	1	2.840	0.00013	2.13	106.55%
openmp	4	1	1.422	0.00013	4.26	106.45%
openmp	8	1	0.906	0.00013	6.68	83.52%
openmp	16	1	0.948	0.00013	6.39	39.91%
mpi	1	1	7.841	0.00013	0.77	77.21%
mpi	1	2	3.841	0.00013	1.58	78.80%
mpi	1	4	2.888	0.00013	2.10	52.39%
mpi	1	8	1.599	0.00013	3.78	47.30%
mpi	1	16	0.861	0.00013	7.03	43.93%
mpi	1	32	0.509	0.00013	11.89	37.17%
mpi_omp	1	4	3.106	0.00013	1.95	48.72%
mpi_omp	2	4	1.494	0.00013	4.05	50.64%
mpi_omp	4	4	1.218	0.00013	4.97	31.07%
mpi_omp	8	4	0.986	0.00013	6.14	19.19%
mpi_omp	1	8	2.695	0.00013	2.25	28.09%
mpi_omp	2	8	1.704	0.00013	3.55	22.20%
mpi_omp	4	8	0.620	0.00013	9.76	30.51%
mpi_omp	8	8	0.519	0.00013	11.66	18.22%
mpi_cuda	1	1	0.125	0.00013	48.42	-



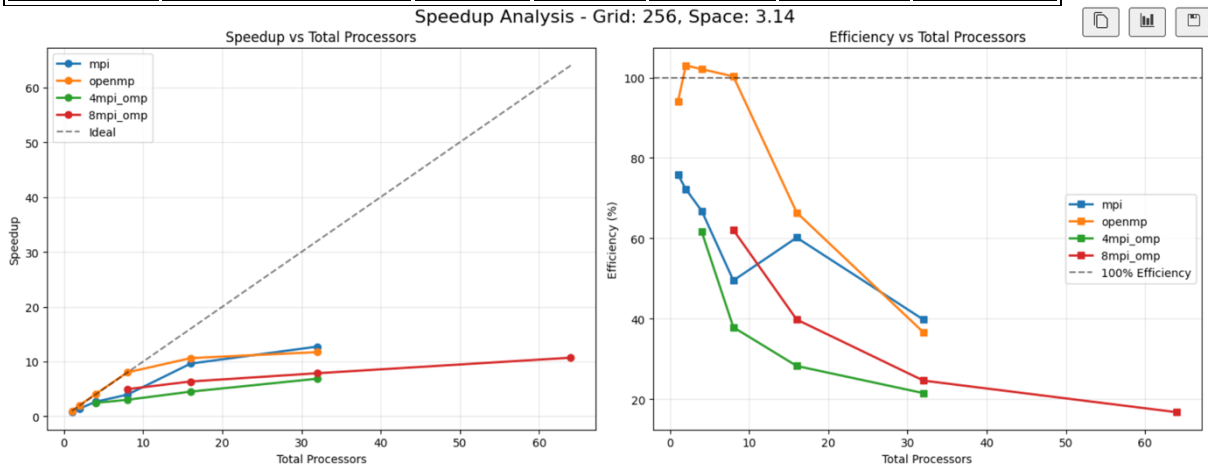
Сетка 256x256x256, L=1.0, T=1.0

Program	ompNumThreads	mpiSize	Time	Error	SpeedUp	Efficiency
no_parallel	1	1	90.421	0.000048	1.00	100.00%
openmp	1	1	91.962	0.000048	0.98	98.32%
openmp	2	1	45.223	0.000048	2.00	100.00%
openmp	4	1	23.326	0.000048	3.88	96.93%
openmp	8	1	12.445	0.000048	7.27	90.84%
openmp	16	1	9.166	0.000048	9.86	61.65%
openmp	32	1	8.119	0.000048	11.14	34.80%
mpi	1	1	122.984	0.000048	0.74	73.52%
mpi	1	2	61.737	0.000048	1.46	73.19%
mpi	1	4	38.909	0.000048	2.32	58.08%
mpi	1	8	24.635	0.000048	3.67	45.88%
mpi	1	16	9.566	0.000048	9.45	59.08%
mpi	1	32	6.381	0.000048	14.17	44.28%
mpi_omp	1	4	38.689	0.000048	2.34	58.44%
mpi_omp	2	4	30.130	0.000048	3.00	37.52%
mpi_omp	4	4	20.560	0.000048	4.40	27.49%
mpi_omp	8	4	13.643	0.000048	6.63	20.72%
mpi_omp	1	8	19.048	0.000048	4.75	59.34%
mpi_omp	2	8	13.681	0.000048	6.61	41.33%
mpi_omp	4	8	11.388	0.000048	7.94	24.81%
mpi_omp	8	8	9.675	0.000048	9.35	14.60%
mpi_cuda	1	1	3.215	0.000015	28.13	-
mpi_cuda	1	2	1.685	0.000015	53.66	-



Сетка 256x256x256, L=π, T=π

Program	ompNumThreads	mpiSize	Time	Error	SpeedUp	Efficiency
no_parallel	1	1	89.172	0.000045	1.00	100.00%
openmp	1	1	99.095	0.000045	0.90	90.00%
openmp	2	1	45.221	0.000045	1.97	98.60%
openmp	4	1	22.815	0.000045	3.91	97.72%
openmp	8	1	11.614	0.000045	7.68	95.98%
openmp	16	1	8.770	0.000045	10.17	63.54%
openmp	32	1	7.953	0.000045	11.21	35.03%
mpi	1	1	122.829	0.000045	0.73	72.60%
mpi	1	2	64.513	0.000045	1.38	69.13%
mpi	1	4	34.854	0.000045	2.56	63.97%
mpi	1	8	23.528	0.000045	3.79	47.39%
mpi	1	16	9.677	0.000045	9.21	57.58%
mpi	1	32	7.322	0.000045	12.18	38.06%
mpi_omp	1	4	37.782	0.000045	2.36	59.02%
mpi_omp	2	4	30.764	0.000045	2.90	36.24%
mpi_omp	4	4	20.631	0.000045	4.32	27.02%
mpi_omp	8	4	13.558	0.000045	6.58	20.56%
mpi_omp	1	8	18.763	0.000045	4.75	59.42%
mpi_omp	2	8	14.639	0.000045	6.09	38.07%
mpi_omp	4	8	11.840	0.000045	7.53	23.54%
mpi_omp	8	8	8.708	0.000045	10.24	16.00%
mpi_cuda	1	1	3.213	0.000051	27.75	-
mpi_cuda	1	2	1.673	0.000051	53.30	-



Сетка 512x512x512, L=1.0, T=1.0

Program	omp	mpi	T Full	T Loop	T steps	T exchanges	T init	T U0	T U1	Error
mpi_omp	8	20	**51.31**	50.82	22.21	5.345	0.00469	0.437	0.053	0.00013
mpi_cuda	1	1	**63.92**	63.81	36.36	0.4089	0.0869	0.0087	0.021	0.00011
mpi_cuda	1	2	**31.03**	30.95	15.2	0.3157	0.0655	0.0054	0.009	0.00011

Сетка 512x512x512, L=π, T=π

Program	omp	mpi	T Full	T Loop	T steps	T exchanges	T init	T U0	T U1	Error
mpi_omp	8	20	**51.203**	50.634	22.475	5.2944	0.00527	0.51154	0.053	0.00013
mpi_cuda	1	1	**59.532**	59.418	32.165	0.4058	0.08717	0.00876	0.018	0.000112
mpi_cuda	1	2	**31.919**	31.838	17.226	0.3336	0.0661	0.00447	0.01	0.000112

Комментарий к результатам (Общий для всех экспериментов основного задания):

Эффективность убывает с ростом числа потоков, так как на каждый поток начинает приходиться меньше вычислений. Сторонние затраты начинают занимать большую долю от общего времени.

На сетке 128 решение OpenMP с 16 потоками работает медленнее, чем с 8 потоками. Причина в коммуникациях между 4 процессорами и малом размере задачи.

Для OpenMP версии эффективность скорее всего превышает 100% по причине недостаточного числа запусков для схождения среднего к истине (От запуска к запуску время выполнения меняется примерно в окне 10% от истинного времени). Здесь среднее время по результатам 3-ёх запусков.

Комментарий к дополнительному заданию (GPU):

Отметим, что MPI+GPU решение работает кратно быстрее остальных в основной части.

Ещё большего ускорения (в 1.5 раза) можно добиться при использовании разделяемой памяти в редукции, но это запрещено условием задания.

То, что MPI+GPU с одним процессом работает медленнее, чем MPI+OMP с 20 процессами объясняется затратами на синхронный замер времени на каждой итерации цикла по времени. Без замеров времени MPI+GPU работает быстрее даже с одним процессом.

Комментарий по оценке корректности работы заданий:

Для обеспечения корректности работы программ сначала была отлажена последовательная версия (3D визуализация погрешностей и вывод в stdout максимальной ошибки).

Корректность остальных версий обеспечивалась следующим процессом написания кода:

- Берём функцию из последовательного решения,
- Строим её параллельный аналог,
- Сравниваем результаты параллельной функции с выверенной последовательной,
- Повторить для всех функций.