Lomonosov Moscow State University

Faculty of Computational Mathematics and Cybernetics

Суперкомпьютерное моделирование и технологии

Отчет о выполнении задания

Разработка параллельной программы решения трехмерного гиперболического уравнения

Вариант 4

студент 624 группы факультета ВМК МГУ

Гладышев Глеб Юрьевич

Москва
2024

Содержание

1 Математическая постановка дифференциальной задачи

В трехмерной замкнутой области

$$\Omega = [0 \le x \le L_x] \times [0 \le y \le L_y] \times [0 \le z \le L_z]$$

для $0 < t \le T$ требуется найти решение $u(x, y, z, t)$ уравнения в частных производных

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

с начальными условиями

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

при условии, что на границах области заданы однородные граничные условия первого рода

$$u(0, y, z, t) = 0, \quad u(L_x, y, z, t) = 0,$$
$$u(x, 0, z, t) = 0, \quad u(x, L_y, z, t) = 0,$$
$$u(x, y, 0, t) = 0, \quad u(x, y, L_z, t) = 0,$$

либо периодические граничные условия

$$u(0, y, z, t) = u(L_x, y, z, t), \quad u_x(0, y, z, t) = u_x(L_x, y, z, t),$$
$$u(x, 0, z, t) = u(x, L_y, z, t), \quad u_y(x, 0, z, t) = u_y(x, L_y, z, t),$$
$$u(x, y, 0, t) = u(x, y, L_z, t), \quad u_z(x, y, 0, t) = u_z(x, y, L_z, t).$$

Конкретная комбинация граничных условий определяется индивидуальным вариантом задания.

2 Численный метод решения задачи

Для численного решения задачи введем на Ω сетку $\omega_{h\tau} = \overline{\omega}_h \times \omega_\tau$, где

$$T = T_0, \quad L_x = L_{x0}, \quad L_y = L_{y0}, \quad L_z = L_{z0},$$

$$\overline{\omega}_h = \{(x_i = ih_x, y_j = jh_y, z_k = kh_z),\ i, j, k = 0, 1, \ldots, N,\ h_x N = L_x,\ h_y N = L_y,\ h_z N = L_z\},$$
$$\omega_\tau = \{t_n = n\tau,\ n = 0, 1, \ldots, K,\ \tau K = T\}.$$

Через ω_h обозначим множество внутренних, а через γ_h — множество граничных узлов сетки $\overline{\omega}_h$.

Для аппроксимации исходного уравнения (1) с однородными граничными условиями (4)-(6) и начальными условиями (2)-(3) воспользуемся следующей системой уравнений:

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

где Δ_h — семиточечный разностный аналог оператора Лапласа:

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

Приведенная выше разностная схема является явной — значение u^{n+1}_{ijk} на $(n+1)$-м шаге можно явно выразить через значения на предыдущих слоях.

Для начала счета (т.е. для нахождения u^2_{ijk}) должны быть заданы значения u^0_{ijk} и u^1_{ijk}, $(x_i, y_j, z_k) \in \omega_h$. Из условия (2) имеем

$$u^0_{ijk} = \varphi(x_i, y_j, z_k), \quad (x_i, y_j, z_k) \in \omega_h.$$

Простейшая замена начального условия (3) уравнением $(u^1_{ijk} - u^0_{ijk})/\tau = 0$ имеет лишь первый порядок аппроксимации по τ. Аппроксимацию второго порядка по τ и h дает разностное уравнение

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

Откуда

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

Разностная аппроксимация для периодических граничных условий выглядит следующим образом:

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

где $i, j, k = 0, 1, \ldots, N$. Для вычисления значений $u^0, u^1 \in \gamma_h$ допускается использование аналитического значения u, которое задается в программе еще для вычисления погрешности решения задачи.

3 Особенности варианта

x	y	z	$u_{\text{analytical}}$
1Р	П	П	$\sin\left(\frac{3\pi}{L_x}x\right) \cdot \sin\left(\frac{2\pi}{L_y}y\right) \cdot \sin\left(\frac{2\pi}{L_z}z\right) \cdot \cos(a_t \cdot t + 4\pi), \quad a_t = \pi\sqrt{\frac{9}{L_x^2} + \frac{4}{L_y^2} + \frac{4}{L_z^2}}$

4 Программная реализация

Алгоритм решения задачи выглядит следующим образом:

1. Исходя из варианта, рассчитывается точное аналитическое решение $u_{\text{analytical}}$ в узлах сетки.
2. Проводим разбиение области Ω между процессами.
3. Фиксируем временной слой (начиная с $t = 0$).
4. Используя формулы (10) и (12), находим значения u^0 и u^1.
5. Пользуясь найденными u^0 и u^1 и разностным представлением уравнения (1), находим значения u в локальной области разбиения.
6. Передаем посчитанные граничные значения блокам-соседям.
7. Повторяем шаги 5-6 для внутренних блоков.
8. Определяем максимальную погрешность на сетке между посчитанным и аналитическим решением.
9. Переходим на следующий слой по времени и повторяем шаги 2-9.

В параллельной версии программы с помощью технологии OpenMP осуществляется распараллеливание следующих фрагментов кода:

- цикл по узлам сетки, вычисляющий начальное условие задачи в момент времени $t = 0$.
- цикл по внутренним узлам сетки, вычисляющий приближенное значение решения в момент времени $t = \tau$.
- цикл по внутренним узлам сетки, вычисляющий приближенное значение решения на каждом временном слое.

- цикл по всем узлам сетки, вычисляющий максимальную разницу между численным решением и аналитическим решением.

Использована директива OpenMP `reduction(max:max_error)`, которая обеспечивает корректное объединение локальных максимальных значений от каждого потока в глобальное максимальное значение.

В представленном коде реализована параллельная программа с использованием библиотеки MPI. Работа начинается с инициализации среды MPI при помощи `MPI_Init`, где каждому процессу присваивается уникальный идентификатор (`rank`) и определяется общее количество процессов (`size`) с помощью `MPI_Comm_rank` и `MPI_Comm_size`.

Для распределения процессов в трёхмерной сетке используется функция `MPI_Dims_create`, которая автоматически определяет разбиение по осям X, Y, Z. Затем `MPI_Cart_create` создаёт декартову топологию, упорядочивая процессы в трёхмерной структуре. Каждый процесс определяет свои координаты в этой решётке с помощью `MPI_Cart_coords`.

Чтобы организовать обмен данными между соседними процессами, `MPI_Cart_shift` определяет соседей каждого процесса по каждой из осей, задавая их ранги в `neighborRanksPrev` и `neighborRanksNext`. Эти данные используются для передачи и получения граничных слоёв между процессами.

Глобальные размеры сетки задаются переменными `Nx`, `Ny`, `Nz`, а локальные размеры вычисляются для каждого процесса в зависимости от количества процессов по оси (`gridDimensions`) и координат процесса в сетке (`gridCoordinates`). Границы локального блока данных (`xCoordinates`, `yCoordinates`, `zCoordinates`) определяются функцией `initCoordinates`, которая рассчитывает индексы начала и конца блока на основе шагов сетки (`hx`, `hy`, `hz`).

Функция `communicateBoundaryLayers` обеспечивает обмен граничными слоями между соседними процессами, используя буферы `sendLeftX`, `sendRightX` и их аналоги для всех осей. Передача и приём данных выполняются с помощью `MPI_Sendrecv`.

Начальные значения для вычислений задаются функцией `updateField`, которая рассчитывает значение поля `u` на основе аналитического решения. На каждом временном шаге данные обновляются, а вычисления производятся с использованием оператора Лапласа (`computeLaplacian`) и метода конечных разностей.

Время выполнения программы измеряется через `MPI_Wtime`. Максимальное время среди всех процессов определяется при помощи `MPI_Reduce` и выводится процессом с нулевым рангом. Завершение работы программы и освобождение ресурсов происходит с вызовом `MPI_Finalize`.

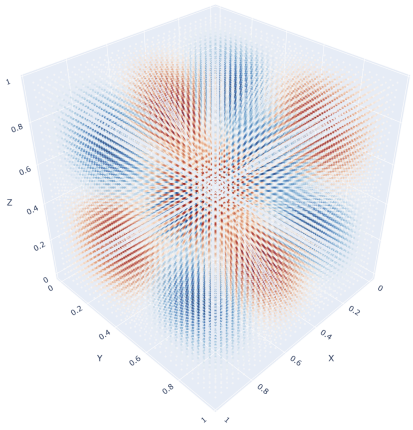
В параллельной версии MPI+CUDA для работы с памятью и реализации редукции использовалась библиотека thrust. С помощью nvprof было выполнено профилирование и подсчитано время, затраченное на работу различных функций программы.

Проверка корректности выполнения параллельных версий программ проверялась сопоставлением результатов работы программ с результатами работы последовательной версии. Ошибки вычисления функции на каждом временном слое выводились в файл мастер-процессом и сравнивались с ошибками последовательной версии. Полученные одинаковые значения позволяют предположить правильность работы программ.

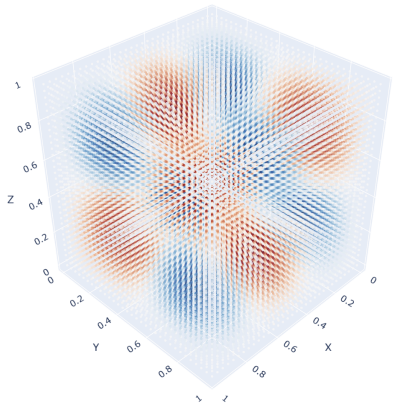
5 Графики решений и погрешности

Графики для аналитической функции, вычисленной функции и погрешности вычисления для $L = 1$, $N = 128$

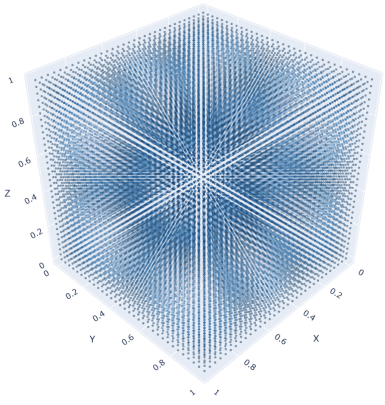
Численное решение:



Аналитическое решение:



Погрешность:



6 Расчеты времени выполнения и погрешности

6.1 OpenMP

Таблица 1: Анализ производительности с различными параметрами сетки

Входные данные L	Число OpenMP нитей в процессе	Число точек сетки N^3	Время решения T	Ускорение S	Погрешность δ
1	1	128^3	3.34371	1	3.98742e-06
1	2	128^3	1.91444	1.74657	3.98742e-06
1	4	128^3	1.08232	3.08939	3.98742e-06
1	8	128^3	0.622881	5.36814	3.98742e-06
1	16	128^3	0.304624	10.97651	3.98742e-06
1	1	256^3	26.4043	1	8.00743e-06
1	2	256^3	13.9152	1.89751	8.00743e-06
1	4	256^3	7.14958	3.69312	8.00743e-06
1	8	256^3	4.98064	5.30139	8.00743e-06
1	16	256^3	3.03546	8.69861	8.00743e-06
π	1	128^3	3.32794	1	4.04025e-07
π	2	128^3	1.72477	1.92949	4.04025e-07
π	4	128^3	0.919064	3.62101	4.04025e-07
π	8	128^3	0.608638	5.46785	4.04025e-07
π	16	128^3	0.304545	10.9275805	4.04025e-07
π	1	256^3	26.4894	1	8.11441e-07
π	2	256^3	13.2944	1.99252	8.11441e-07
π	4	256^3	6.74646	3.92641	8.11441e-07
π	8	256^3	3.65854	7.24042	8.11441e-07
π	16	256^3	3.14987	8.40968	8.11441e-07

6.2 MPI

Таблица 2: Анализ производительности с различными параметрами сетки

Входные данные L	Число MPI процессов	Число точек сетки N^3	Время решения T	Ускорение S	Погрешность δ
1	1	128^3	8.27669	1	3.98744e-06
1	2	128^3	4.31723	1.91712	3.98744e-06
1	4	128^3	2.29078	3.61304	3.98744e-06
1	8	128^3	1.30512	6.34170	3.98744e-06
1	16	128^3	0.87159	9.49608	3.98744e-06
1	32	128^3	0.65704	12.59693	3.98744e-06
1	1	256^3	64.04690	1	8.00759e-06
1	2	256^3	32.62930	1.96286	8.00759e-06
1	4	256^3	16.99500	3.76857	8.00759e-06
1	8	256^3	9.45677	6.7726	8.00759e-06
1	16	256^3	7.21225	8.88029	8.00759e-06
1	32	256^3	4.85620	13.18869	8.00759e-06
π	1	128^3	8.08089	1	4.04025e-07
π	2	128^3	4.23362	1.90874	4.04025e-07
π	4	128^3	3.70684	2.17999	4.04025e-07
π	8	128^3	1.26281	6.39913	4.04025e-07
π	16	128^3	1.09307	7.39284	4.04025e-07
π	32	128^3	0.499441	16.17987	4.04025e-07
π	1	256^3	64.45800	1	8.11442e-07
π	2	256^3	32.57080	1.97901	8.11441e-07
π	4	256^3	17.02980	3.78501	8.11441e-07
π	8	256^3	9.52857	6.76471	8.11441e-07
π	16	256^3	4.82382	13.36244	8.11441e-07
π	32	256^3	2.97843	21.6416	8.11441e-07

6.3 MPI + OpenMP

Везде число нитей OpenMP = 4

Таблица 3: Анализ производительности с различными параметрами сетки

Входные данные L	Число MPI процессов	Число точек сетки N^3	Время решения T	Ускорение S	Погрешность δ
1	1	128^3	6.79814	1	3.98744e-06
1	2	128^3	9.62519	0.70629	3.98744e-06
1	4	128^3	11.3649	0.59817	3.98744e-06
1	8	128^3	14.0331	0.48444	3.98744e-06
1	16	128^3	10.2732	0.66174	3.98744e-06
1	32	128^3	25.3954	0.26769	3.98744e-06
1	1	256^3	34.9094	1	8.00759e-06
1	2	256^3	22.8497	1.52778	8.00759e-06
1	4	256^3	35.8495	0.97378	8.00759e-06
1	8	256^3	13.4166	2.60196	8.00759e-06
1	16	256^3	9.2506	3.77374	8.00759e-06
1	32	256^3	19.3432	1.80474	8.00759e-06
π	1	128^3	2.22027	1	4.04025e-07
π	2	128^3	7.59362	0.29239	4.04025e-07
π	4	128^3	9.08707	0.24433	4.04025e-07
π	8	128^3	6.70878	0.33095	4.04025e-07
π	16	128^3	16.8566	0.13172	4.04025e-07
π	32	128^3	13.104	0.16943	4.04025e-07
π	1	256^3	16.9434	1	8.11442e-07
π	2	256^3	33.0304	0.51296	8.11441e-07
π	4	256^3	26.9937	0.62768	8.11441e-07
π	8	256^3	32.7482	0.51738	8.11441e-07
π	16	256^3	23.2874	0.72758	8.11441e-07
π	32	256^3	17.2685	0.98117	8.11441e-07

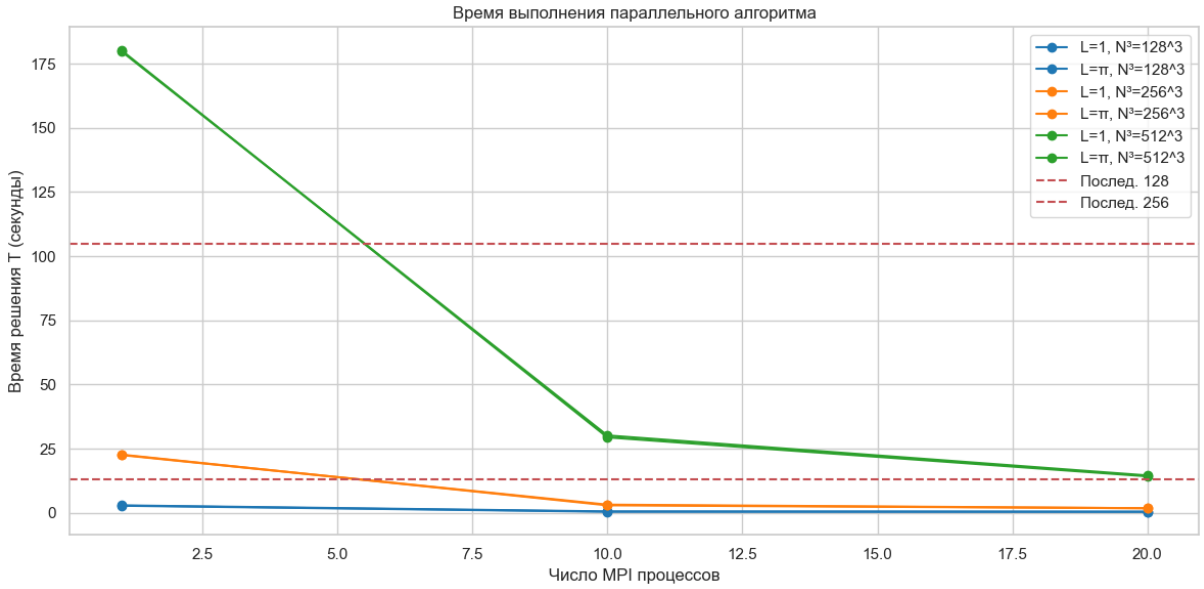
Гибридная программа MPI+OpenMP демонстрирует значительно худшую производительность по сравнению с программой, использующей только MPI. Это связано с существенными накладными расходами. Помимо затрат на межпроцессные коммуникации, использование OpenMP увеличивает нагрузку, так как потоки обрабатывают относительно небольшую часть данных, выделенных процессу. В результате на графиках при увеличении числа процессов часто наблюдается замедление работы программы.

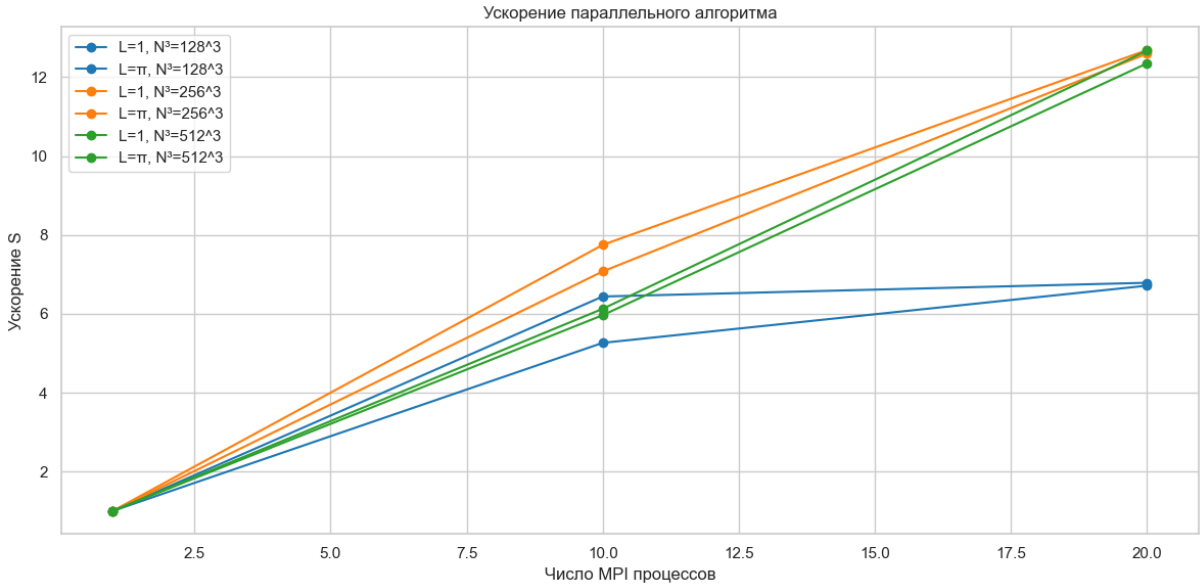
6.4 MPI + CUDA

CUDA со 128 нитями на процесс

Таблица 4: Анализ производительности с различными параметрами сетки

Входные данные L	Число MPI процессов	Число точек сетки N^3	Время решения T	Ускорение S	Погрешность δ
1	1	128^3	2.90145	1	3.98744e-06
1	10	128^3	0.551218	5.263	3.98744e-06
1	20	128^3	0.432163	6.7137	3.98744e-06
1	1	256^3	22.6893	1	8.00759e-06
1	10	256^3	2.92874	7.748	8.00759e-06
1	20	256^3	1.78786	12.69	8.00759e-06
1	1	512^3	180.259	1	1.60425e-05
1	10	512^3	29.4216	6.1267	1.60425e-05
1	20	512^3	14.2223	12.6744	1.60425e-05
π	1	128^3	2.76558	1	4.04025e-07
π	10	128^3	0.429524	6.4387	4.04025e-07
π	20	128^3	0.407486	6.7869	4.04025e-07
π	1	256^3	22.5277	1	8.11442e-07
π	10	256^3	3.18431	7.07459	8.11441e-07
π	20	256^3	1.78786	12.6	8.11441e-07
π	1	512^3	179.892	1	1.6264e-06
π	10	512^3	30.1561	5.9654	1.6264e-06
π	20	512^3	14.5633	12.3524	1.6264e-06





7 Профилирование MPI+CUDA

Для оценки времени выполнения различных частей программы были проведены замеры с использованием утилиты `nvprof`. Программа запускалась на одном MPI-процессе с различными размерами сетки $N = 128$, 256 и 512. Общее время, затраченное на выполнение основных функций, представлено в таблицах.

Таблица 5: Времена выполнения основных функций при $L = 1$, $p = 1$

Имя функции	**Время выполнения**
N = 128	
updateFieldKernel(...)	23.675 ms
computeLaplacianKernel(...)	15.041 ms
initializeU1Kernel(...)	0.722 ms
initializeField(...)	0.167 ms
N = 256	
computeLaplacianKernel(...)	343.79 ms
updateFieldKernel(...)	273.12 ms
initializeU1Kernel(...)	11.106 ms
N = 512	
computeLaplacianKernel(...)	3.07276 s
updateFieldKernel(...)	2.29396 s
initializeU1Kernel(...)	93.576 ms

Таблица 6: Времена выполнения вспомогательных функций при $L = 1$, $p = 1$

Имя функции	**Время выполнения**
N = 128	
cudaMemcpyAsync	46.096 ms
cudaDeviceSynchronize	29.777 ms
cudaMalloc	774.59 ms
cudaLaunchKernel	3.6855 ms
cudaFree	0.9552 ms
N = 256	
cudaMemcpyAsync	737.59 ms
cudaDeviceSynchronize	228.19 ms
cudaMalloc	227.72 ms
cudaLaunchKernel	17.640 ms
cudaFree	0.76297 ms
N = 512	
cudaMemcpyAsync	4.70999 s
cudaDeviceSynchronize	1.76144 s
cudaMalloc	238.10 ms
cudaLaunchKernel	16.218 ms
cudaFree	3.5114 ms

Исходя из полученных результатов, наблюдается, что значительная часть времени выполнения программы уходит на операции передачи данных между хостом и GPU. Также существенное время затрачивается на выполнение основных вычислительных ядер, таких как `computeLaplacianKernel` и `updateFieldKernel`, особенно при увеличении размера сетки.

Вспомогательные функции, такие как `cudaMalloc` и `cudaMemcpyAsync`, также занимают значительную долю времени, особенно при увеличении размера сетки. Например, для $N = 512$ `cudaMemcpyAsync` отвечает за 43.23% времени выполнения вспомогательных функций.

8 Выводы

В ходе работы были разработаны последовательная и четыре параллельные программы (OpenMP, MPI, MPI+OpenMP, MPI+CUDA) для численного решения трехмерного гиперболического уравнения с использованием явной разностной схемы. Для вычисления максимальной погрешности и предотвращения состояния гонки применялся механизм редукции с функцией «максимум».

Несмотря на неизбежные дополнительные затраты при выполнении параллельных программ, можно заключить, что использование параллельного алгоритма для решения этой задачи существенно сокращает время вычисления значений функции и обеспечивает более высокую точность результатов.