Юрий Зыков. 628

Задание 2 и 3 (CUDA)

Постановка задачи

В прямоугольной области П = [A1, A2] x [B1, B2] требуется найти дважды гладкую функцию u = u(x, y), удовлетворяющую дифференциальному уравнению
-Lu = F(x, y), A1 < x < A2, B1 < y < B2
и дополнительному граничному условию u(x, y) = f(x, y).
Оператор Лапласа L определен равенством: $Lu = \partial^2 u/\partial x^2 + \partial^2 u/\partial y^2$
Функции F(x, y) и f(x, y) определены 14 вариантом:
$F(x, y) = 4(1 - 2(x + y)(x + y))e^{1 - (x + y)(x + y)}$
$f(x, y) = e^{1 - (x + y)(x + y)}$ **(искомая функция p(x, y) оказалась идентичной f(x, y), достаточно дважды продифференцировать)**
П = [0, 2] x [0, 2]
Сетка:
$x_i = A2t(i / N1) + A1(1 – t(i / N2)), i = 0, …, N1,$
$y_i = A2t(i / N1) + A1(1 – t(i / N2)), i = 0, …, N2,$
$t(x) = ((1 + x)^q – 1) / (2^q – 1), 0 <= t <= 1,$
q = 1.5
Требуемая точность: $eps = 10^{-4}$
Норма: максимум-норма $||p|| = \max|p(x_i, y_j)|, 0 < i < N1, 0 < j < N2$

Разностная схема решения задачи

Для всех функций вводятся их дискретные аналоги, для оператора Лапласа – его 5-точечный разностный аналог. Приближенное решение поставленной задачи находится с помощью начальной итерации метода скорейшего спуска и последующих итераций метода сопряженных градиентов до получения требуемой точности. Все вычисления осуществляются только в узлах расчетной сетки, никакие промежуточные точки не используются. Более формально схема описана в формулировке задания.

Реализация MPI

При запуске на N ядрах (N – степень двух) создается декартова топология MPI-процессов, каждый из которых знает своих соседей в решетке, а также размеры своего блока (блок – подсетка узлов) и его смещение относительно (0, 0) глобальной расчетной сетки. Размеры топологии определяются исходя из значений N1, N2 – используется процедура partition(), поочередно уменьшающая оба параметра в два раза и увеличивающая их счетчики, пока сумма счетчиков не станет равна количеству выделенных ядер-процессов (такой алгоритм был описан на одной из лекций) – таким образом балансируется нагрузка между MPI-процессами, длина и ширина итоговых блоков отличаются друг от друга не более чем в два раза (сами параметры N1, N2 также должны удовлетворять этому требованию для хорошей балансировки). Для удобства введена структура Context, хранящая всю необходимую для вычислений информацию:

```
typedef struct Context {
  // problem params
  INDEXTYPE N1;
  INDEXTYPE N2;
  BASETYPE A1;
  BASETYPE A2;
  BASETYPE B1;
  BASETYPE B2;
  BASETYPE q;
  BASETYPE eps;

  // environment
  MPI_Comm comm;
  int rank;
  int size;
```

```
  int dims[NDIMS];  // размеры декартовой топологии по X, Y
  int coords[NDIMS];  // координаты процесса в топологии
  int up, down, left, right;  // соседи

  // block params, in local and global coords
  INDEXTYPE blockSize[NDIMS];  // размер блока по X, Y (кол-во узлов расчетной сетки)
  INDEXTYPE offsets[NDIMS];  // смещения относительно (0, 0) глобальной сетки
  INDEXTYPE begin[NDIMS];  // смещение в собственном блоке (для процессов на периметре, которые не должны пересчитывать граничные точки)
  INDEXTYPE end[NDIMS];  // смещение в собственном блоке (для процессов на периметре, которые не должны пересчитывать граничные точки)

  // computation data
  BASETYPE* grid[NDIMS];  // значения xi, yj, в виде двух массивов, а не матрицы (более компактно)
  BASETYPE* h[NDIMS];  // шаги сетки по X, Y
  BASETYPE* mh[NDIMS];  // половины шагов сетки по X, Y
  BASETYPE* shadow[NDIMS];  // массивы для передачи столбцов в теневые грани соседей (строки могут быть переданы без временного хранилища)
  BASETYPE* shadow_copy[NDIMS];  // массивы для приема столбцов в теневые грани от соседей (строки могут быть приняты без временного хранилища)
  BASETYPE* prod_coeff;  // матрица коэффициентов для скалярного произведения
  BASETYPE* f;  // матрица значений функции F
  BASETYPE* p;  // текущее приближение искомой функции
  BASETYPE* r;  // невязка
  BASETYPE* g;  // g, связывает текущую невязку с предыдущей итерацией (для МСГ)

  // GPU data
  // all pointers at host are device pointers
  struct Context* device_ctx;  // копия структуры в памяти устройства, все указатели (кроме теневых граней) – в память устройства
  INDEXTYPE blockDim, gridDim, threadDim;  // узлы в нити, нити в блоке, блоки в гриде
  BASETYPE* shadows[NDIMS][NDIMS * NDIMS];  // теневые грани - вынуждены хранить в хостовой памяти (отображена в адресное пространство устройства, по факту во время итерационного процесса вообще не будет вызовов cudaMemcpy между CPU и GPU) для передачи через MPI
  double* reduction;  // временное хранилище, в основном для редукции – как для MPI, так и для блоков GPU

  INDEXTYPE iteration;
  double current_eps;
} Context;
```

Теневые грани расчетного блока конкретного процесса – дополнительный периметр блока, хранящий данные соседних процессов (коммуникации через MPI), т.е. на самом деле на процессе выделена матрица узлов сетки размером (blockSize[X] + 2) * (blockSize[Y] + 2), проход в которой осуществляется только по «внутренностям» (такой подход позволяет не тратить такты на проверки, находимся ли мы на границе топологии, и избежать многократного дублирования кода для всех случаев (begin, end хранят реальные границы блоков)). Алгоритм в рамках одной итерации – последовательный, состоит из трех шагов и трех обновлений теневых граней:

```
computeR(&ctx);
actualizeShadows(&ctx, ctx.r);
computeG(&ctx, dg_g);
actualizeShadows(&ctx, ctx.g);
computeP(&ctx, &dg_g);
actualizeShadows(&ctx, ctx.p);
```

dg_g – скалярное произведение g и ее дискретной аппроксимации, может быть переиспользовано на следующей итерации (вместо «тяжеловесного» нового пересчета через 5-точечный оператор Лапласа (вообще говоря, самая времязатратная операция (кэш-промахи тоже дают о себе знать) (попытки ускорить ее через встроенные функции SSE/AVX ускорения не дали))).

Коммуникации – через асинхронные Isend/Irecv с последующим Waitall (время коммуникаций оказалось слишком мало, чтобы занять чем-нибудь процессор в период до подтвержденных отсылки/приема данных).

Итого на одну итерацию в худшем случае:

3 x 4 Isend (грани),

3 x 4 Irecv (грани),

3 Allreduce (частичные суммы скалярных произведений).

Примененные оптимизации:

- предрассчитать значения функции F в узлах блока текущего процесса (программно «кешировать»), но хранить их со знаком «минус» (F, которое ctx.f, используется только для расчета невязки, сложение (add) с отрицательным выполняется быстрее, чем вычитание (sub) положительного);
- в ущерб читабельности не реализовывать скалярное произведение отдельной функцией (в одном вложенном цикле в ComputeP можно сразу считать два произведения (а точнее, две частичные суммы));
- предрассчитать коэффициенты скалярного произведения для всех узлов текущего блока (формула для пространства H в формулировке задания);
- пересчитать шаги сетки и их полусуммы так, чтобы в 5-точечном операторе Лапласа не использовать операцию деления.

Реализация OMP

Все инициализирующие функции на старте программы используют #pragma omp parallel for (т.к. один виток пишет в одну ячейку памяти). Для вложенных циклов используется клауза collapse(2) (OMP будет рассматривать общее пространство витков, а не только внешний цикл). На BlueGene collapse не поддерживается, поэтому необходимо в main.h перед компиляцией сделать макрос COLLAPSE(x) пустым. Также на BlueGene не поддерживаются беззнаковые индексные переменные циклов, поэтому нужно также заменить макрос INDEXTYPE на int (тем не менее, unsigned int работает быстрее, например, на ПВС «Ломоносов»). Все основные вычислительные циклы в указанных выше трех шагах – также распараллеливаются, т.к. пишут в уникальные для витка места памяти. Редукция для суммы реализована посредством клаузы reduction(+:), редукция для максимума не поддерживается на BlueGene, сделана одной параллельной секцией с приватными максимумами нитей и завершающей критической секцией для нахождения глобального максимума для данного MPI-процесса. На BlueGene использовались три нити OMP на один MPI-процесс, на «Ломоносов» - 8 нитей. Асинхронные операции коммуникаций распараллеливать с OMP не имеет смысла – накладные расходы на создание секций оказались гораздо выше.

Реализация CUDA

Реализация максимально близка по структуре основной программе на C, имена функций те же, за исключением префикса cuda у хостовых функций и _cuda у ядер и девайсовых функций. Используется схема «один MPI-процесс – один графический ускоритель». Максимальное количество нитей в блоке установлено в 512 (Tesla X2070 позволяет 1024, но реализация использует больше ресурсов, тем не менее можно скомпилировать с параметром –maxrregcount 32, но прироста не будет, и смысла нет, так как больше 14SM * 1536 нитей запустить не получится, а загрузить видеокарту получится и с 512 нитями на блок (т.е. 3 параллельных блока по 512 нитей на 1 SM)). Вся динамическая память в структуре Context, кроме теневых граней – на устройстве, теневые грани отображены в адресное пространство графического ускорителя. Схема одной итерации:

```
cudaComputeR(&ctx);
cudaActualizeShadows(&ctx, ctx.r);
cudaComputeG(&ctx, dg_g);
cudaActualizeShadows(&ctx, ctx.g);
cudaComputeP(&ctx, &dg_g);
cudaActualizeShadows(&ctx, ctx.p);
```

Используется только один cuda-поток (cudaStream). Каждой нити устройства сопоставляется некоторое количество элементов сетки (ctx.threadDim), однако для максимально эффективного доступа к глобальной памяти устройства обращения должны быть выравнены, поэтому нулевая нить обрабатывает нулевой элемент, первая нить – первый и т.д., следующий элемент у нулевой нити станет 512 элементом массива, у первой – 513 и т.д. Таким образом, GPU сможет обеспечить чтение большого кол-ва данных (128(?) байт) одной транзакцией. Для операций редукции используется быстрая общая память, с последующей редукцией по блокам. Для каждого из трех вычислительных шагов выше реализовано собственное ядро (kernel). Посылка/загрузка вертикальных (столбцовых) теневых граней также реализована ядрами. Профилирование проводилось с помощью утилиты nvprof.

Выходные данные

Бинарный файл с именем argv[3] (argv[1], argv[2] – N1, N2 соответственно), первый байт – тип данных (0 – float, 1 – double), два натуральных числа N1, N2 по 4 байта, значения x_i сетки ((N1 + 1) * sizeof(тип данных) байт), значения y_i сетки ((N2 + 1) * sizeof(тип данных) байт), значения аппроксимации искомой функции p ((N1 + 1) * (N2 + 1) * sizeof(тип данных) байт). В стандартный поток вывода печатается время инициализации, время выполнения расчетов и абсолютная погрешность найденного решения.

Визуализация

Python-скрипт visualize.py (2.7, 3.5), два параметра – имя бинарного файла и порядок байт на платформе (le – Little-Endian, be – Big-Endian, по умолчанию le, на BlueGene Big-Endian), на выходе (может занять продолжительное время):

1. <имя бинарного файла>.png и <имя бинарного файла>.accur.png (будут, если установлен matplotlib; графики полученного и точного решения);
2. <имя бинарного файла>.dat и <имя бинарного файла>.plt – данные и скрипт для gnuplot, график полученного решения;
3. <имя бинарного файла>.accur.dat и <имя бинарного файла>.accur.plt – данные и скрипт для gnuplot, график точного решения;
4. <имя бинарного файла>.re.dat и <имя бинарного файла>.re.plt – данные и скрипт для gnuplot, график относительной погрешности решения (из-за специфики расчета относительной погрешности в областях с очень маленькими значениями функции могут возникать «всплески», в моем варианте – в окрестности точки (2, 2), при повышении точности проблема пропадает; можно было бы обнулять все значения меньше заданного eps, но это неестественное решение);
5. <имя бинарного файла>.ae.dat и <имя бинарного файла>.ae.plt – данные и скрипт для gnuplot, график относительной погрешности решения.

Кастомизация

Все данные, уникальные для варианта, хранятся в файле 14.h, включается в исходные коды с помощью #include VARIANT(14). Достаточно везде заменить 14 на любой другой № варианта, изменив нужные константы/функции F, f и функции нормы (приводятся примеры), чтобы программа начала решать поставленную задачу для другого варианта (полиморфизм в стиле C).

Make

Makefile:

SSH_KEY_L = # полный путь к приватному ключу
SSH_KEY_BG = $(SSH_KEY_L)
USER_L = # логин Lomonosov
USER_BG = # логин BlueGene
VARIANT = 14
+ различные константы для используемых компиляторов и т.п.

Цели:

all – собрать main (mpi), main_o (mpi+omp), main_c (mpi+cuda) (можно по отдельности, это подцели)
lmount – создать в текущей директории папку lomonosov и по ssh подмонтировать удаленную папку на ПВС «Ломоносов» (папка с именем-логином)
bgmount – аналогично для ПВС BlueGene
lupload – загрузить на ПВС «Ломоносов» исходники и Makefile
bgupload – аналогично для ПВС BlueGene (но без main.cu)
umount – размонтировать обе папки
clean – удалить собранные исполняемые программы
lcopy – на ПВС «Ломоносов» - скопировать собранные программы в папку _scratch (для узлов видна только она)
bgcustom – сборка программ на BlueGene
vis <имя бинарного файла> <le:be> - визуализировать данные
gnuplot <*.plt> - выполнить gnuplot над скриптом

Аппаратное и программное обеспечение

ПВС BlueGene/P:

На одном узле 4x PowerPC 450 850MHz, 2ГБ RAM, не более 4 нитей на узле (использовалось 3). Коммуникационная сеть – трехмерный тор, латентность до соседа не более 0,8 μs, пропускная способность 425МБ/с до каждого соседа в обе стороны. Используемые компиляторы: mpixlc, mpixlc_r.

ПВС «Ломоносов»:

На одном узле 2x Intel Xeon X5570 2.93GHz, 8 x86-ядер, 12ГБ RAM, с поддержкой HyperThreading – не более 16 параллельных нитей на узле (т.е. не более 16 нитей и 12ГБ RAM на 1 полноценный процесс) (использовалось 8). На узлах с GPU по два графических ускорителя Nvidia Tesla X2070. Коммуникационная сеть - QDR Infiniband 4x/10G Ethernet/Gigabit Ethernet. Используемые компиляторы: nvcc, mpiicc. Используемые библиотеки: Intel MPI.

Результаты

Ниже приводятся «сырые» (raw) данные запусков и их графики.

N – число используемых ядер (если ПВС BlueGene, то 3 нити в случае OMP; если ПВС «Ломоносов», то 8 нитей (два процесса на узел)). На BlueGene – по возможности используется отображение процессов на ядра «тор», на ПВС «Ломоносов» - линейное. Вызов MPI_Cart_create (с параметром reorder=1) может дополнительно их переупорядочить (а точнее, не сами процессы, а их номера) для оптимизации коммуникаций.

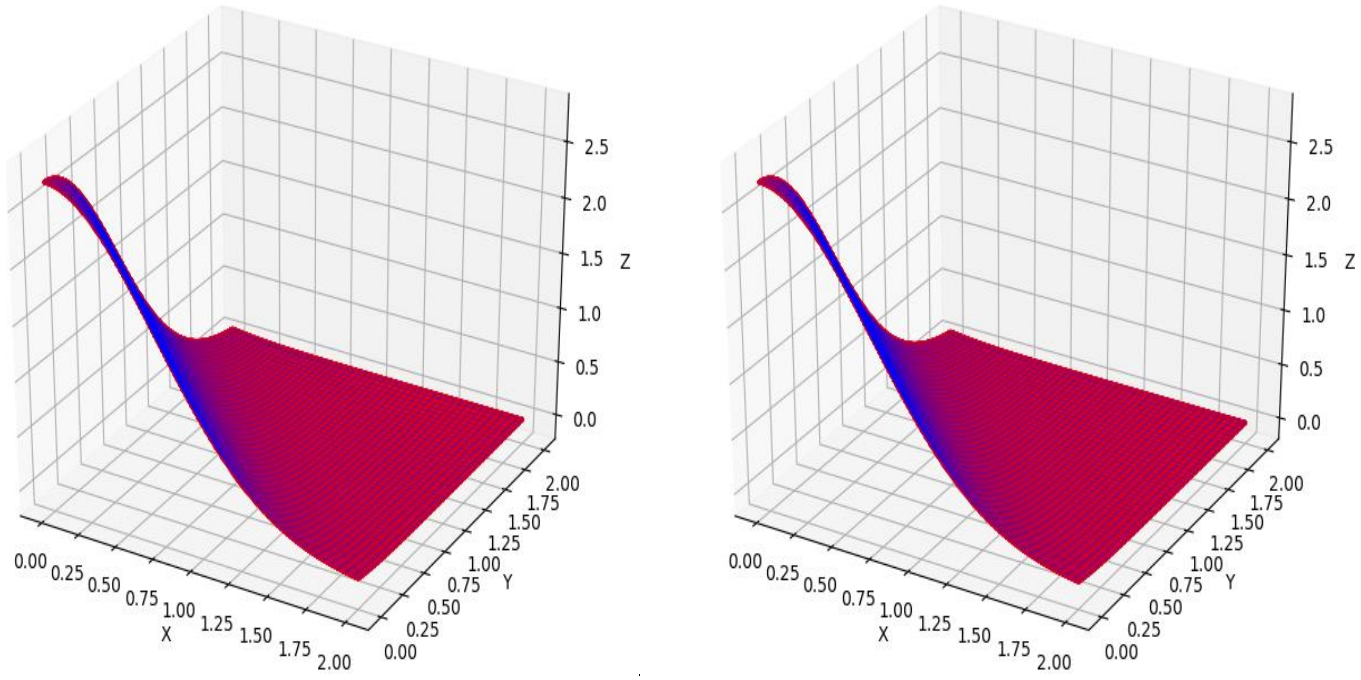
T – время расчета (только вычисления, без инициализации), секунды.

$S = T_1 / T_N$ – ускорение.

E = S / N – эффективность (от 0 до 1, иногда выше (объяснено ниже)).

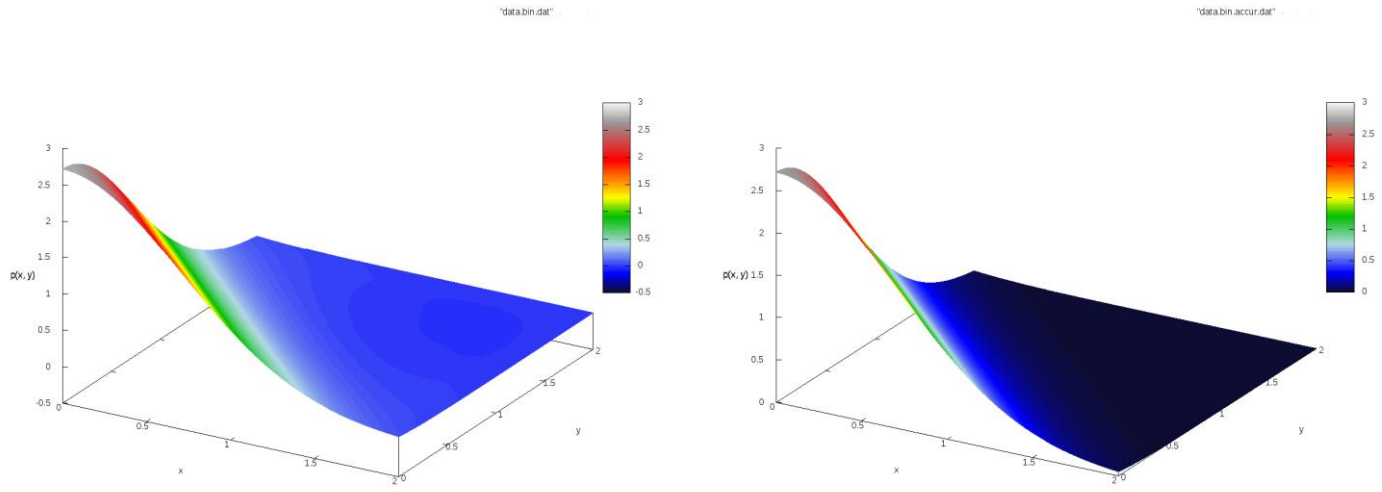
I – кол-во итераций (иногда незначительно расходится из-за погрешности вычислений).

Приближенное и точное решение 2000x2000 (matplotlib):



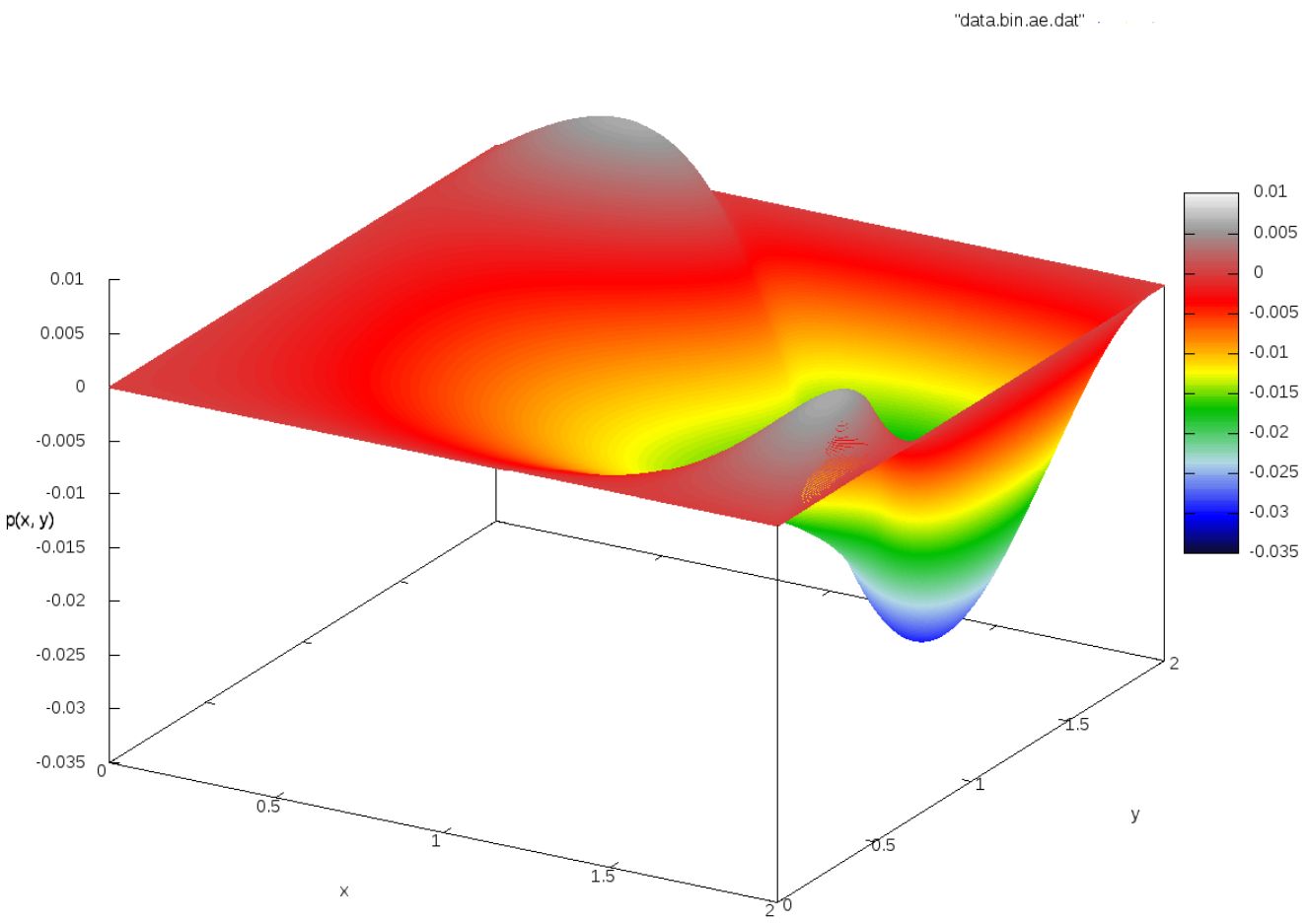
Изображения почти идентичны, но на правом (точное) отметка 0.0 по вертикали находится чуть ниже, чем на левом.

Приближенное и точное решение 2000x2000 (gnuplot):



У приближенного решения из-за погрешности часть значений ниже нуля, поэтому на графике отобразился интервал (-0.5, 0.0), точное решение по форме аналогично.

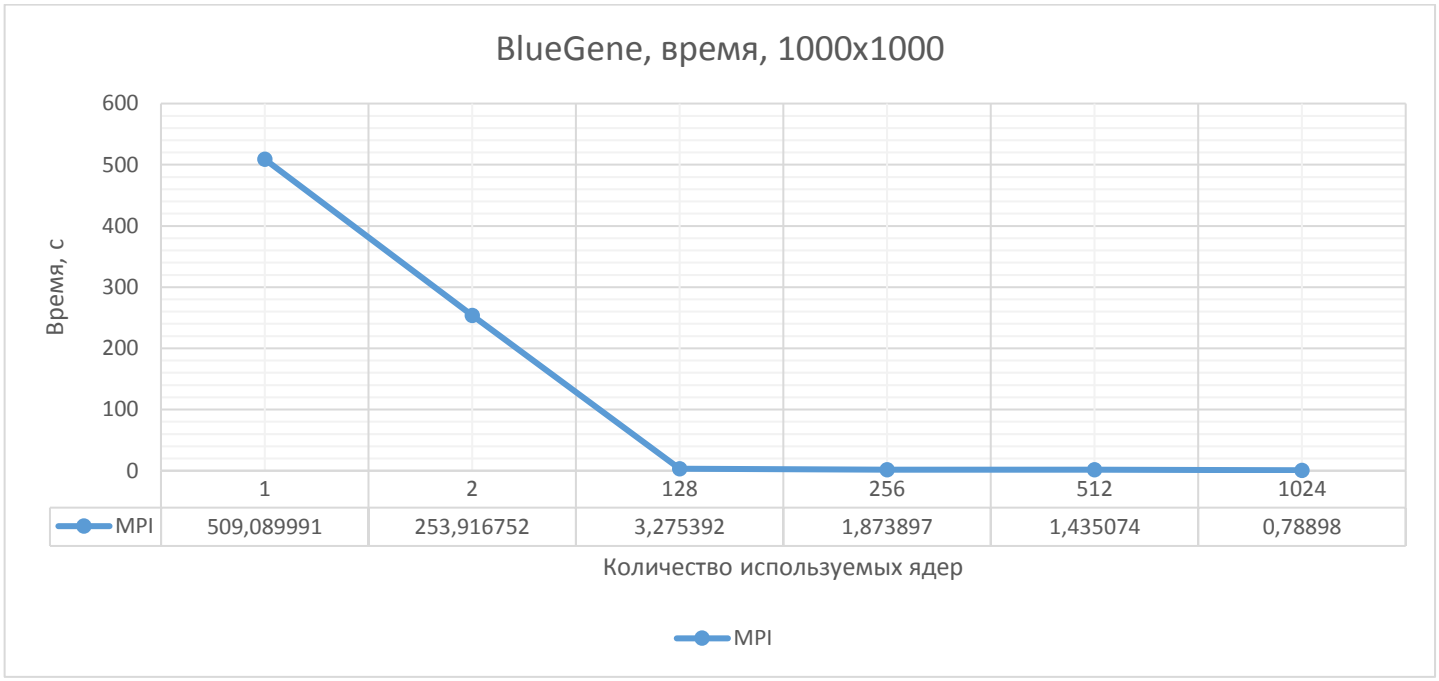
Абсолютная погрешность решения:

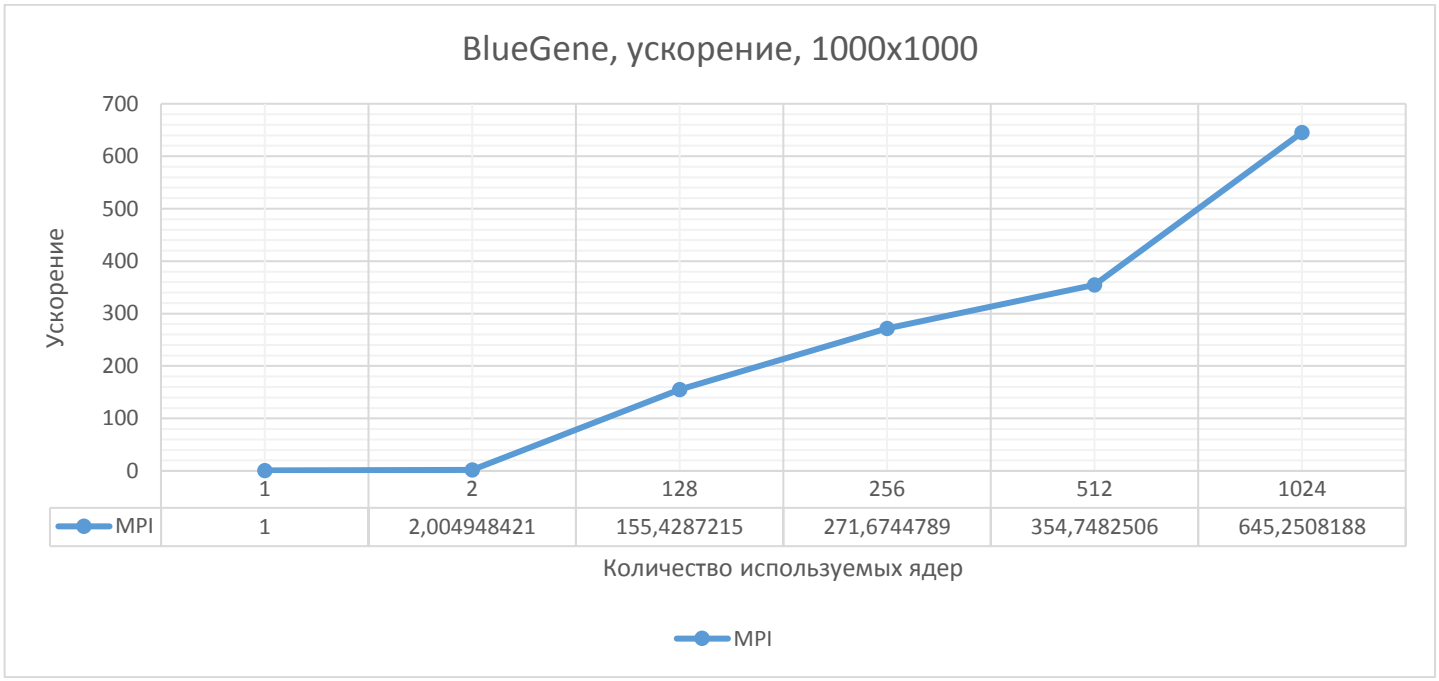


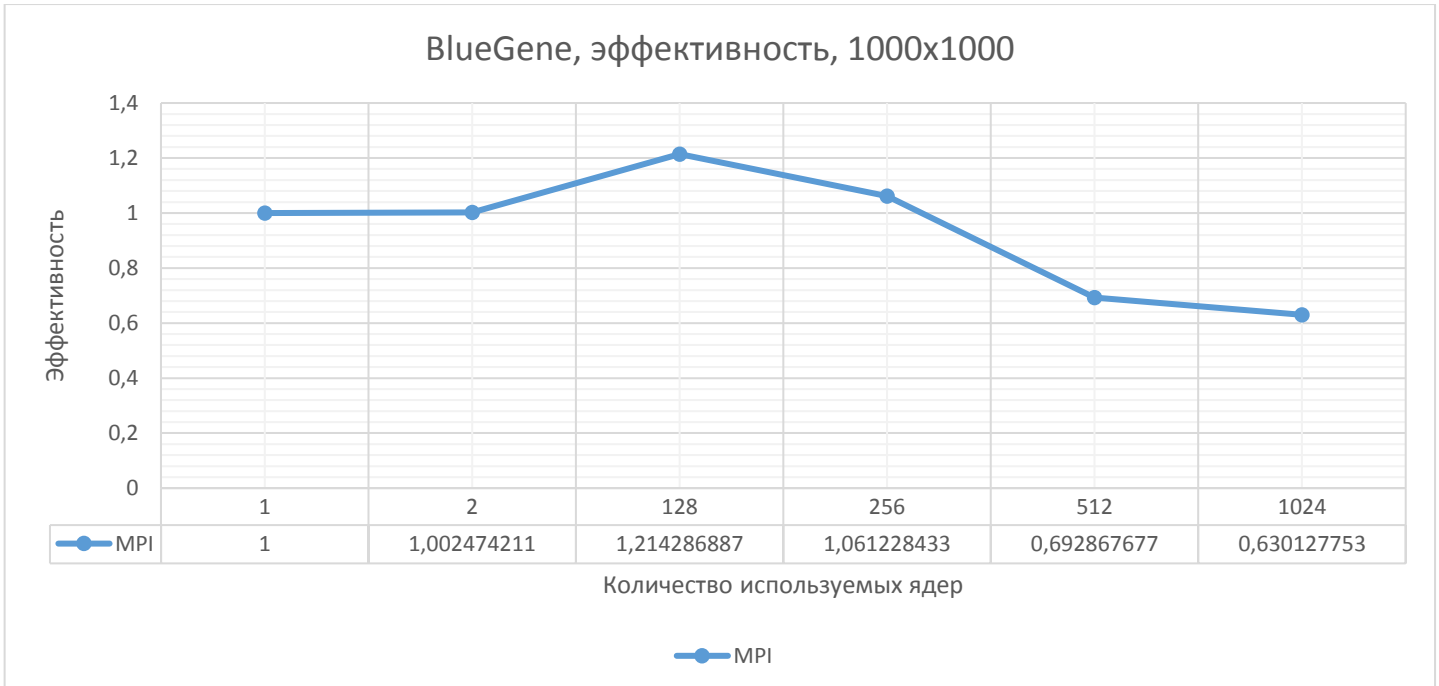
Bluegene(1000, MPI)				
N	T	S	E	I
1	509,089991	1	1	1701
2	253,916752	2,004948421	1,00247421	1701
128	3,275392	155,4287215	1,21428689	1701
256	1,873897	271,6744789	1,06122843	1701
512	1,435074	354,7482506	0,69286768	1701
1024	0,78898	645,2508188	0,63012775	1701
Bluegene(1000, MPI+OMP)				
N	T	S	E	I
3	171,510321	2,968276125	0,98942537	1701
6	86,227123	5,904058645	0,98400977	1701
384	1,807913	281,5898724	0,73330696	1701
768	1,310254	388,5429779	0,50591534	1701
1536	1,468752	346,61399	0,22566015	1701
3072	0,952575	534,4355993	0,17396992	1701
Bluegene(2000, MPI)				
N	T	S	E	I
1	4092,10743	1	1	3413
2	2053,95809	1,992303276	0,99615164	3413
128	23,449088	174,5103019	1,36336173	3412
256	12,112976	169,5667594	0,66237015	3412
512	7,247668	564,6102203	1,10275434	3412
1024	3,764589	1086,999783	1,06152323	3412
Bluegene(2000, MPI+OMP)				
N	T	S	E	I
3	1377,72089	2,970200615	0,99006687	3413
6	693,39252	5,901574228	0,9835957	3412
384	9,599318	426,2914747	1,11013405	3412
768	5,483225	746,295734	0,97173924	3412
1536	4,478257	913,7723507	0,59490387	3412
3072	2,653428	1542,196519	0,5020171	3412

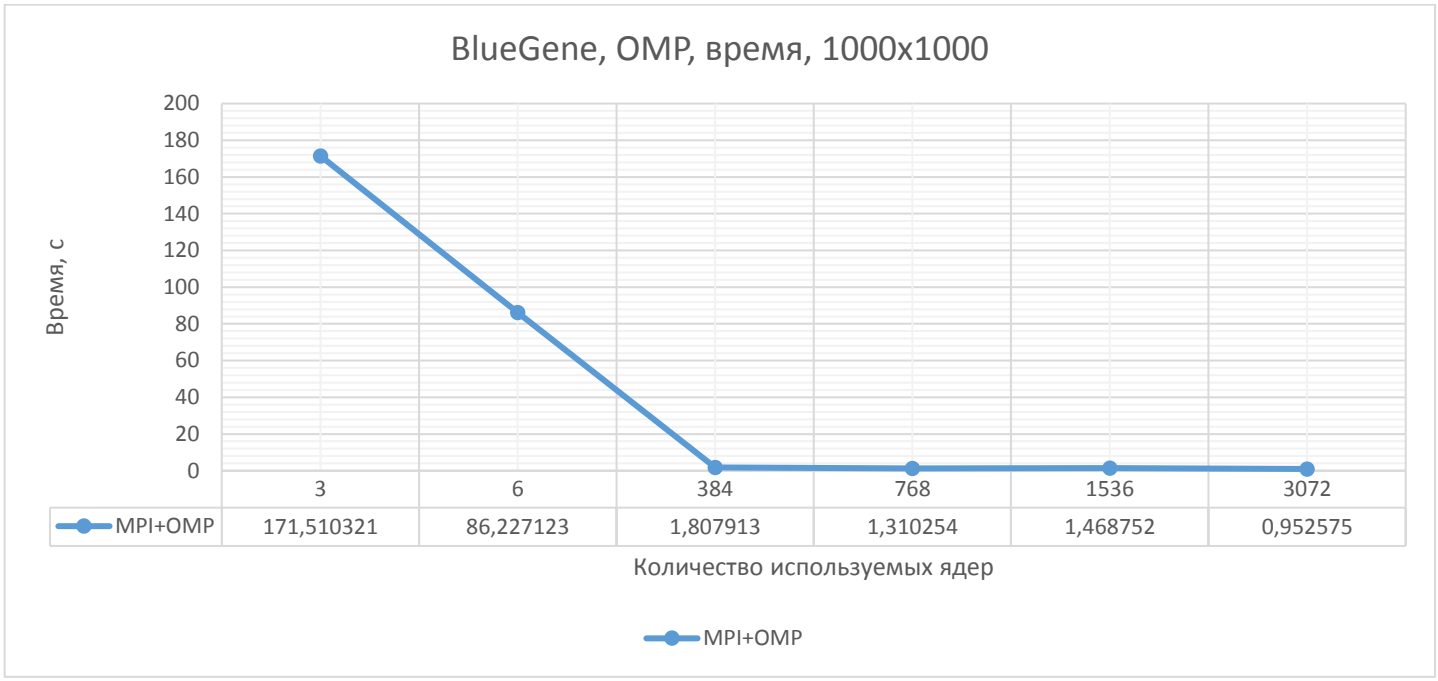
Lomonosov(1000, MPI)				
N	T	S	E	I
1	50,332994	1	1	1701
8	7,456423	6,750286833	0,84378585	1701
16	3,15029	15,97725733	0,99857858	1701
32	1,605576	31,34887044	0,9796522	1701
64	0,863473	58,29133511	0,91080211	1701
128	0,501998	100,2653278	0,78332287	1701
Lomonosov(2000, MPI)				
N	T	S	E	I
1	404,245819	1	1	3413
8	64,135427	6,303003471	0,78787543	3413
16	31,883225	12,67895011	0,79243438	3412
32	15,490663	26,09609537	0,81550298	3412
64	6,594275	61,30254183	0,95785222	3412
128	3,442407	117,4311518	0,91743087	3412
Lomonosov(1000, MPI+OMP)				
N	T	S	E	I
8	12,307352	4,089668842	0,51120861	1701
64	3,254604	15,46516688	0,24164323	1701
128	2,311736	21,77281229	0,1701001	1701
256	1,475359	34,1157603	0,13326469	1701
512	1,520123	33,11113245	0,06467018	1701
1024	1,597744	31,50253983	0,0307642	1701
Lomonosov(2000, MPI+OMP)				
N	T	S	E	I
8	97,769476	4,134683293	0,51683541	3412
64	25,296389	15,98037645	0,24969338	3412
128	13,21631	30,58688991	0,23896008	3412
256	7,032696	57,48091756	0,22453483	3412
512	4,827327	83,74113024	0,16355689	3412
1024	3,546889	113,9719396	0,11130072	3412

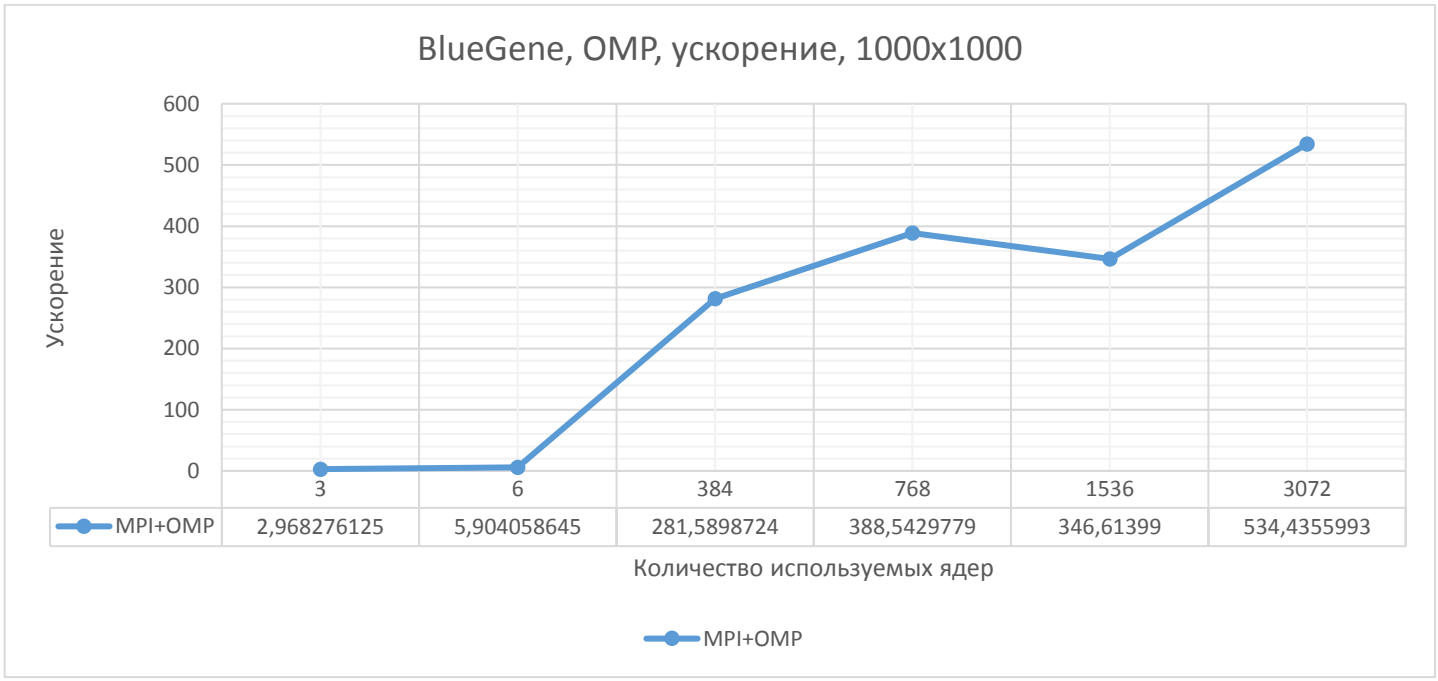
Lomonosov(1000, MPI+CUDA)				
N	T	S	E	I
1	3,853147	1	1	1701
2	2,427794	1,587097999	0,793549	1701
4	1,832925	2,102184759	0,52554619	1701
8	1,537529	2,506064601	0,31325808	1701
16	1,473122	2,615633328	0,16347708	1701
32	1,37461	2,803083784	0,08759637	1701
Lomonosov(2000, MPI+CUDA)				
N	T	S	E	I
1	29,117193	1	1	3412
2	15,786998	1,844378076	0,92218904	3412
4	8,755417	3,325620356	0,83140509	3412
8	5,686837	5,120103319	0,64001291	3412
16	4,273648	6,813194021	0,42582463	3412
32	3,666345	7,941749344	0,24817967	3412
Lomonosov(5000, MPI+CUDA)				
N	T	S	E	I
1	413,90291	1	1	6650
2	213,489814	1,938747813	0,96937391	6648
4	96,832007	4,27444316	1,06861079	6650
8	49,784135	8,313952025	1,039244	6641
16	26,12795	15,8413848	0,99008655	6650
32	15,764556	26,25528496	0,82047765	6650

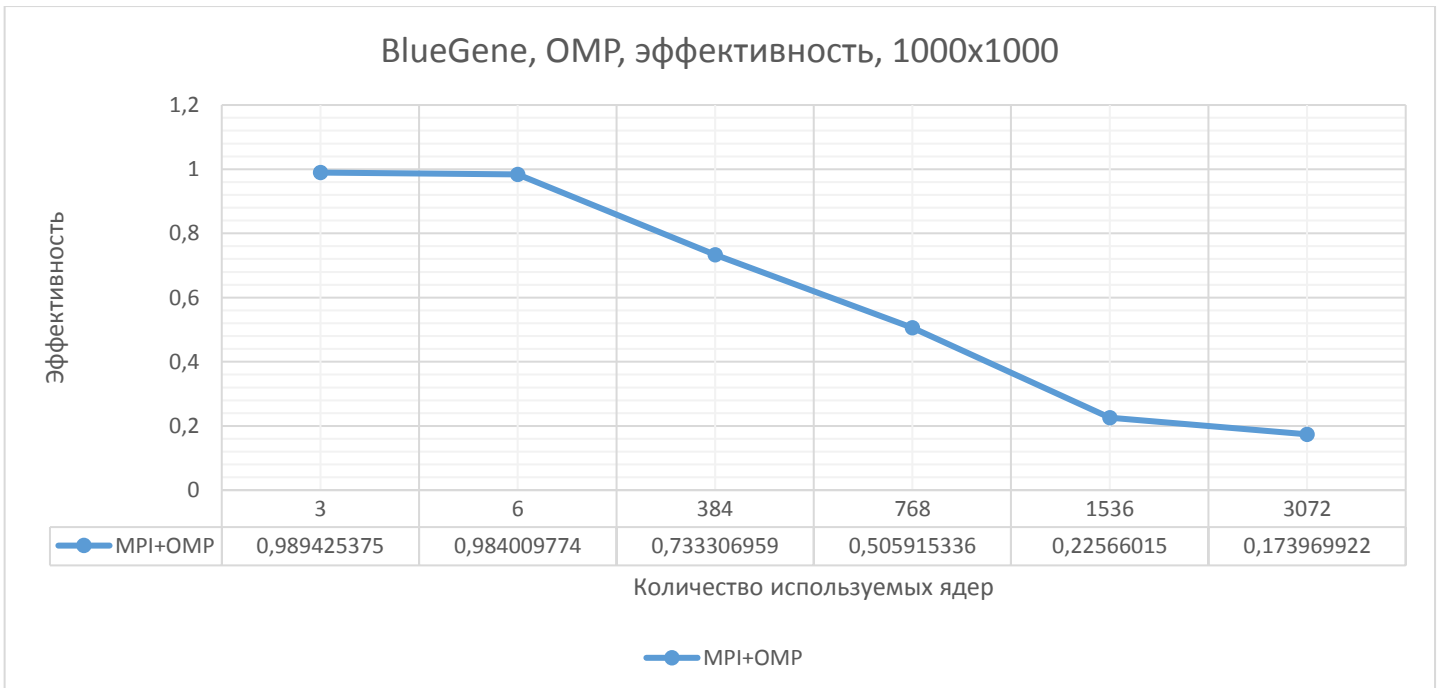

BlueGene, время, 1000x1000
Время, с
600
500
400
300
200
100
0
1
2
128
256
512
1024
MPI
509,089991
253,916752
3,275392
1,873897
1,435074
0,78898
Количество используемых ядер
MPI

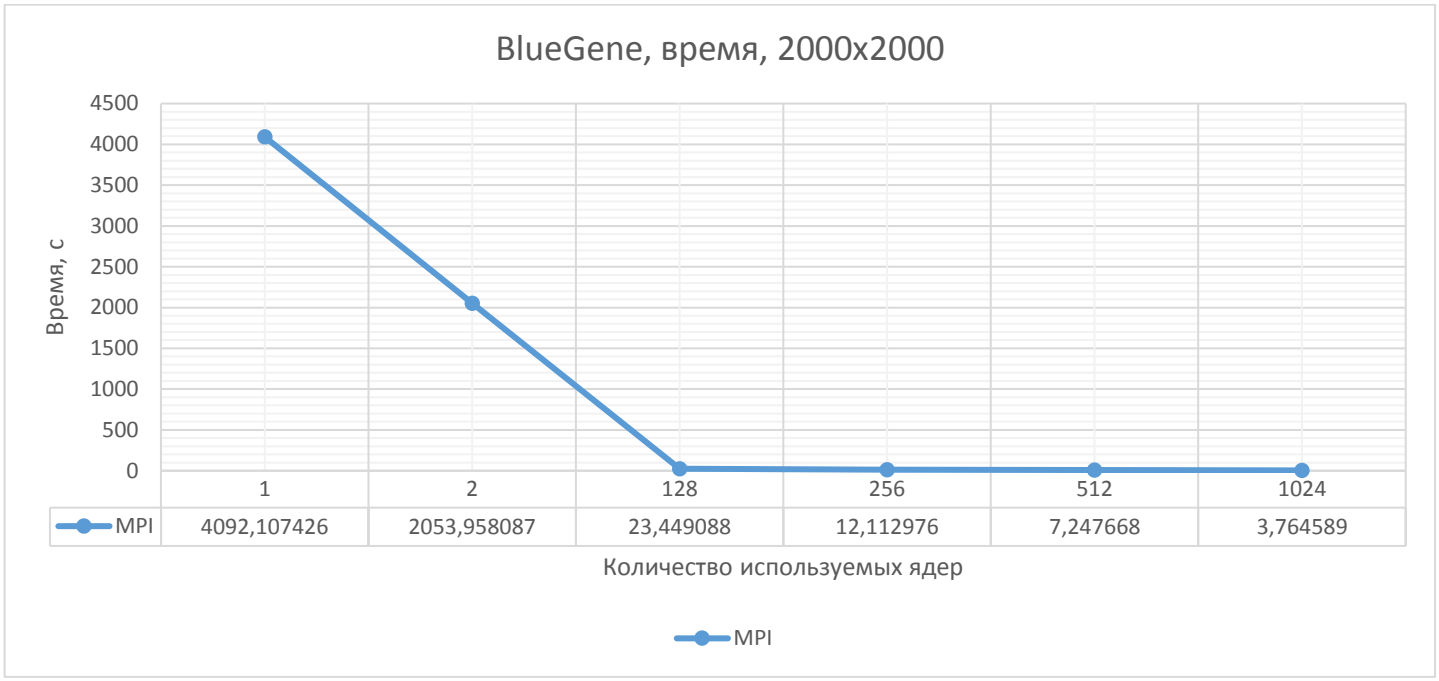

BlueGene, ускорение, 1000x1000
Ускорение
700
600
500
400
300
200
100
0
1
2
128
256
512
1024
MPI
1
2,004948421
155,4287215
271,6744789
354,7482506
645,2508188
Количество используемых ядер
MPI

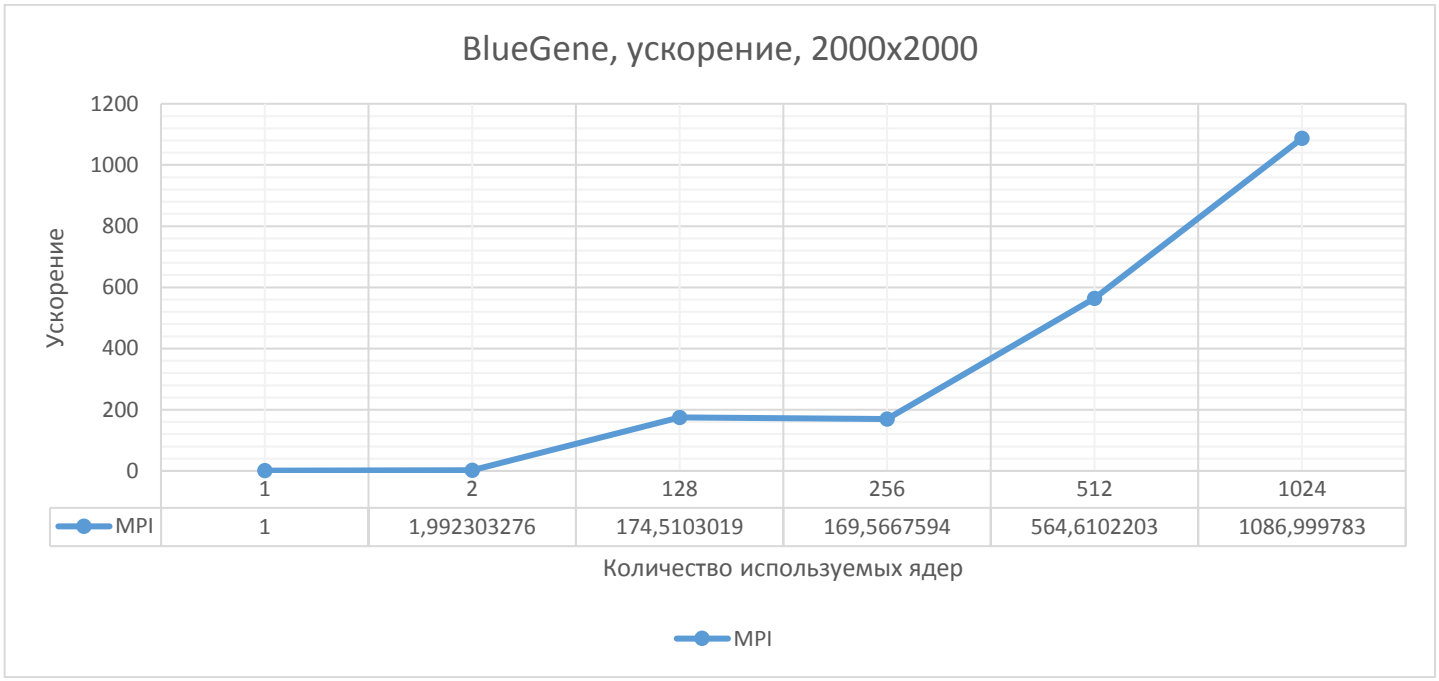

BlueGene, эффективность, 1000x1000
Эффективность
1,4
1,2
1
0,8
0,6
0,4
0,2
0
1
2
128
256
512
1024
MPI
1
1,002474211
1,214286887
1,061228433
0,692867677
0,630127753
Количество используемых ядер
MPI

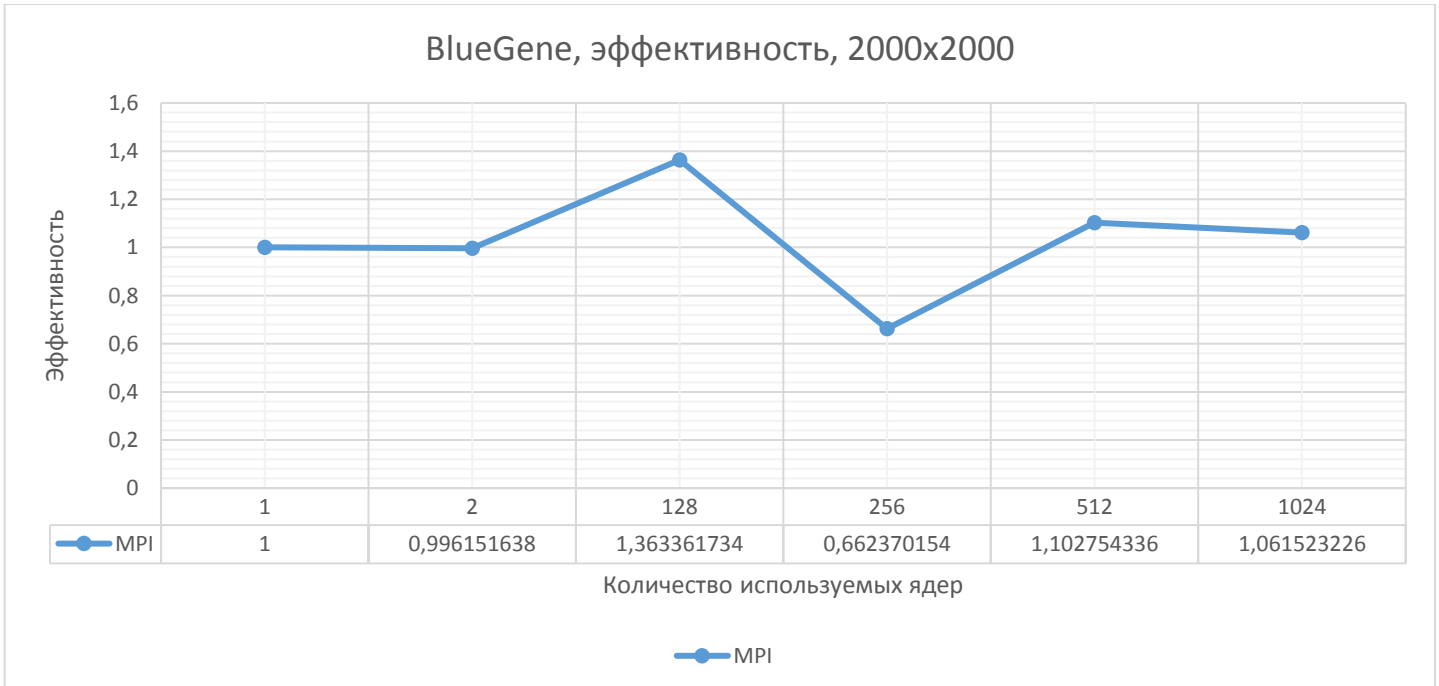

BlueGene, OMP, время, 1000x1000
Время, с
200
180
160
140
120
100
80
60
40
20
0
3
6
384
768
1536
3072
MPI+OMP
171,510321
86,227123
1,807913
1,310254
1,468752
0,952575
Количество используемых ядер
MPI+OMP

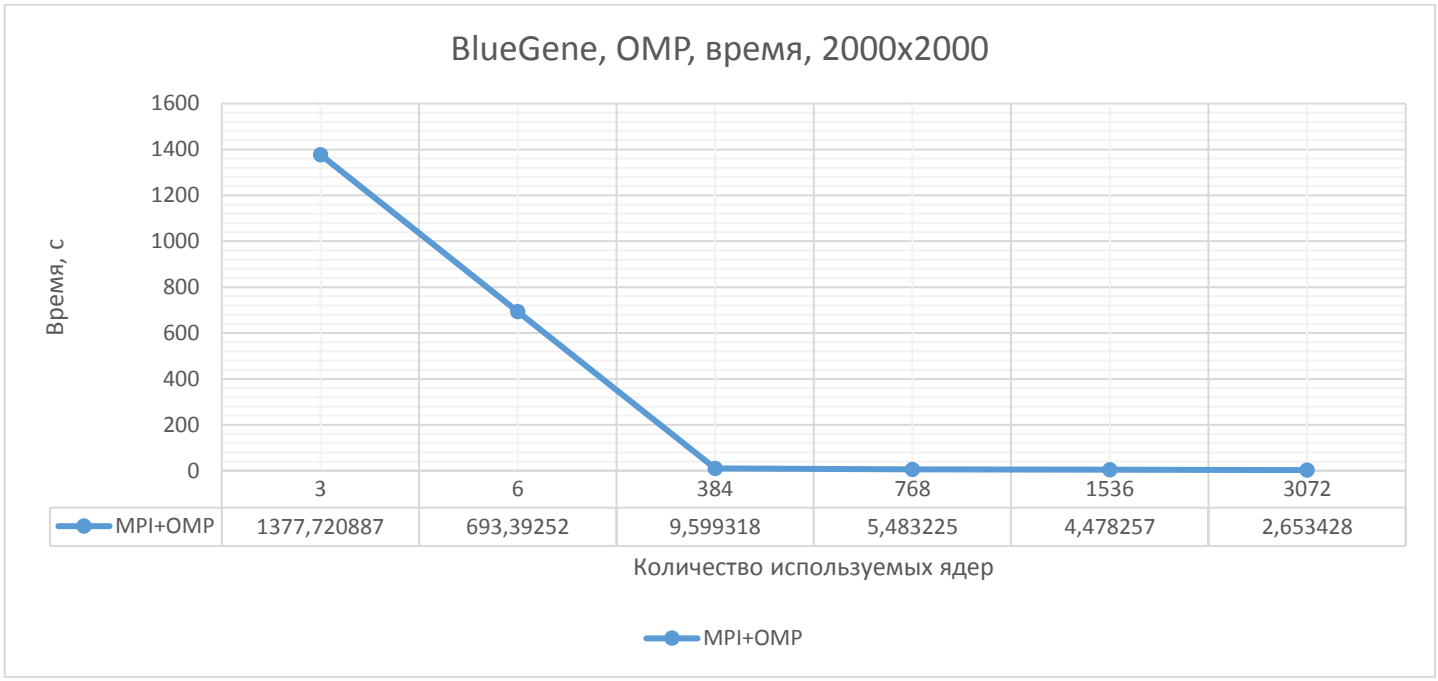

BlueGene, OMP, ускорение, 1000x1000
Ускорение
600
500
400
300
200
100
0
3
6
384
768
1536
3072
MPI+OMP
2,968276125
5,904058645
281,5898724
388,5429779
346,61399
534,4355993
Количество используемых ядер
MPI+OMP

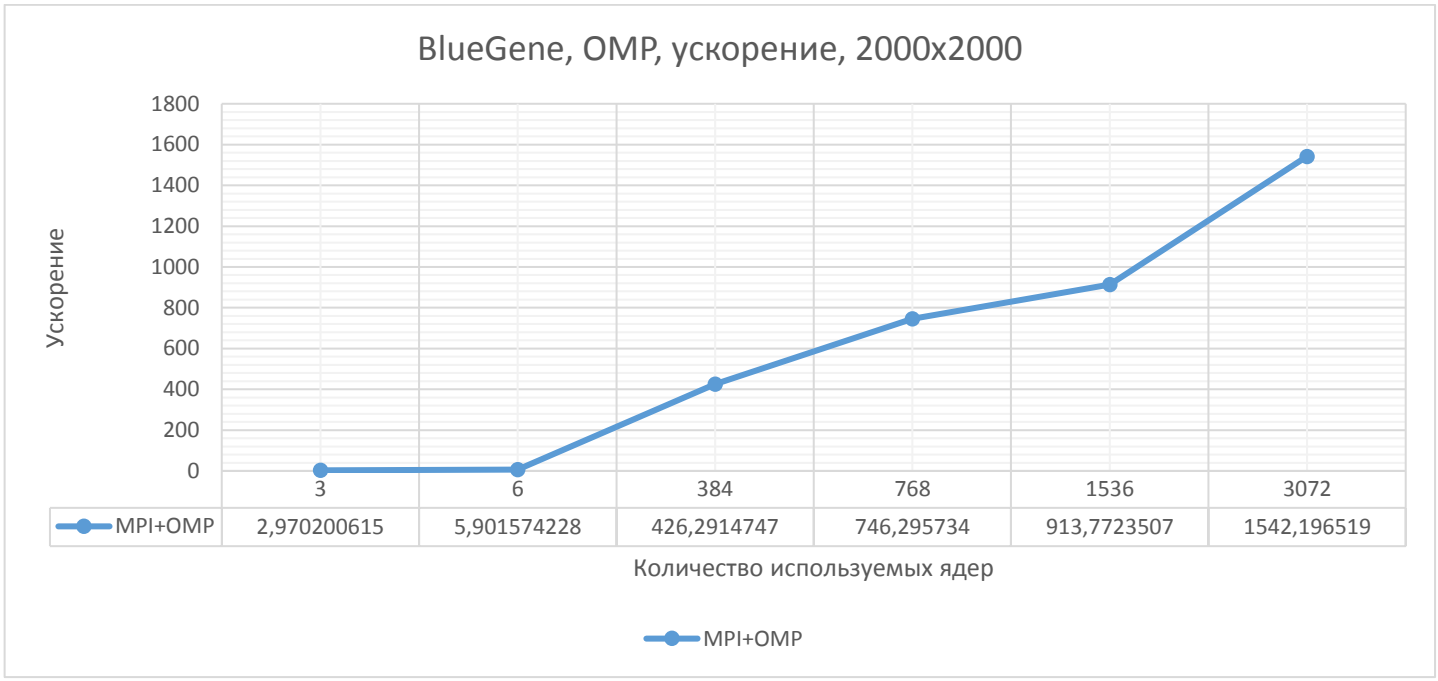

BlueGene, OMP, эффективность, 1000x1000
Эффективность
1,2
1
0,8
0,6
0,4
0,2
0
3
6
384
768
1536
3072
MPI+OMP
0,989425375
0,984009774
0,733306959
0,505915336
0,22566015
0,173969922
Количество используемых ядер
MPI+OMP

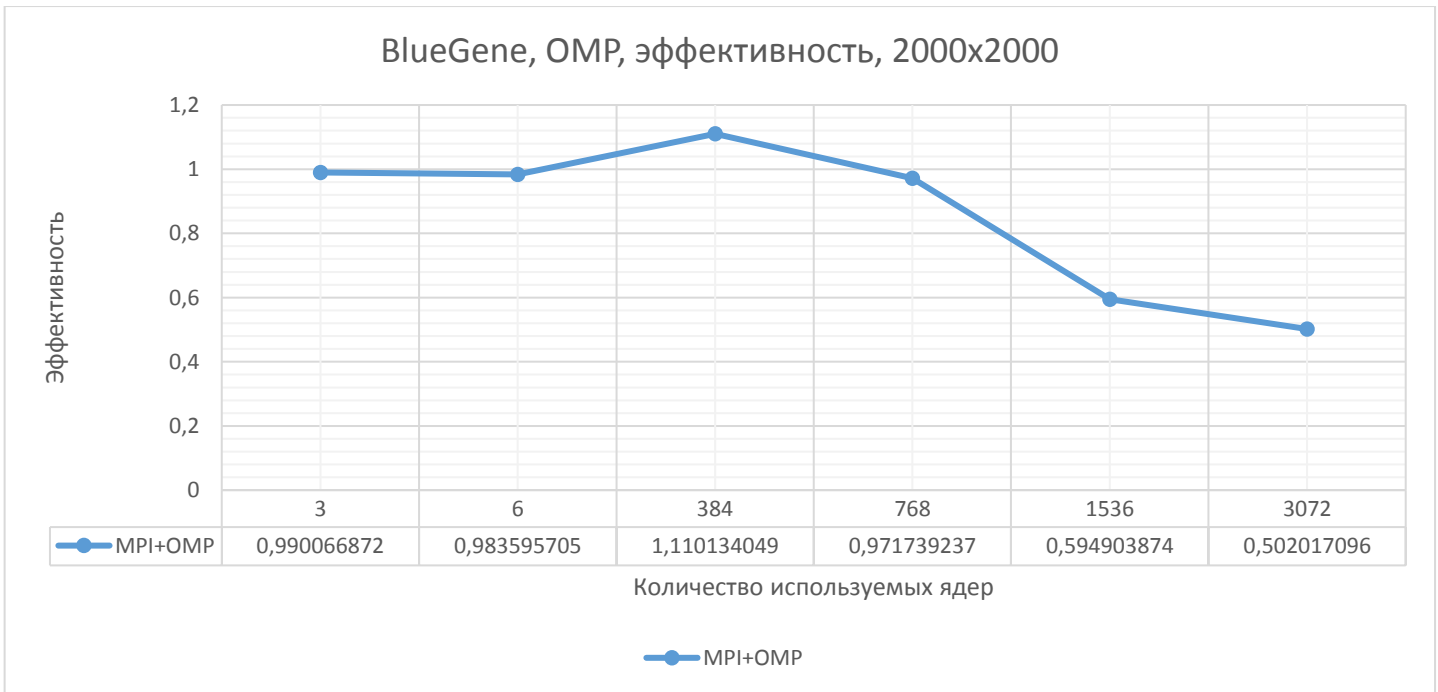

BlueGene, время, 2000x2000
Время, с
4500
4000
3500
3000
2500
2000
1500
1000
500
0
1
2
128
256
512
1024
MPI
4092,107426
2053,958087
23,449088
12,112976
7,247668
3,764589
Количество используемых ядер
MPI

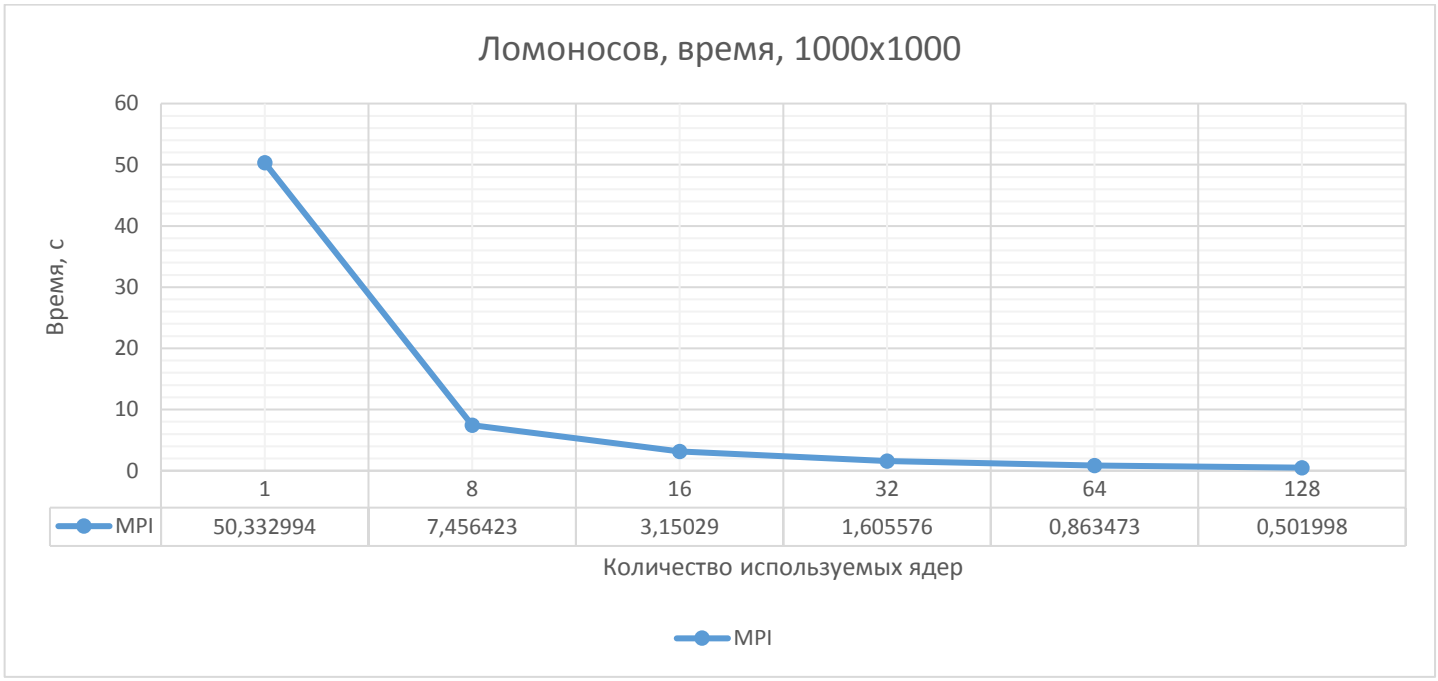

BlueGene, ускорение, 2000x2000
Ускорение
1200
1000
800
600
400
200
0
1
2
128
256
512
1024
MPI
1
1,992303276
174,5103019
169,5667594
564,6102203
1086,999783
Количество используемых ядер
MPI


BlueGene, эффективность, 2000x2000
Эффективность
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
1
2
128
256
512
1024
MPI
1
0,996151638
1,363361734
0,662370154
1,102754336
1,061523226
Количество используемых ядер
MPI

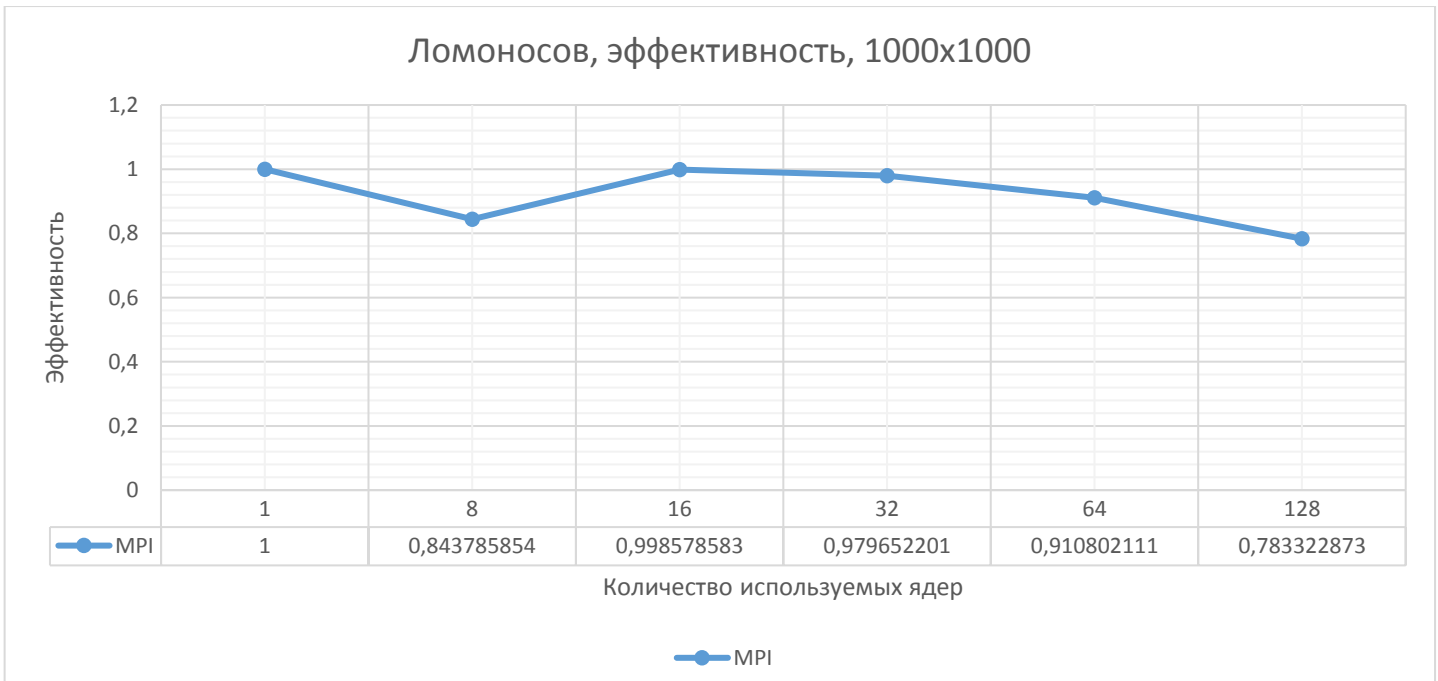

BlueGene, OMP, время, 2000x2000
Время, с
1600
1400
1200
1000
800
600
400
200
0
3
6
384
768
1536
3072
MPI+OMP
1377,720887
693,39252
9,599318
5,483225
4,478257
2,653428
Количество используемых ядер
MPI+OMP

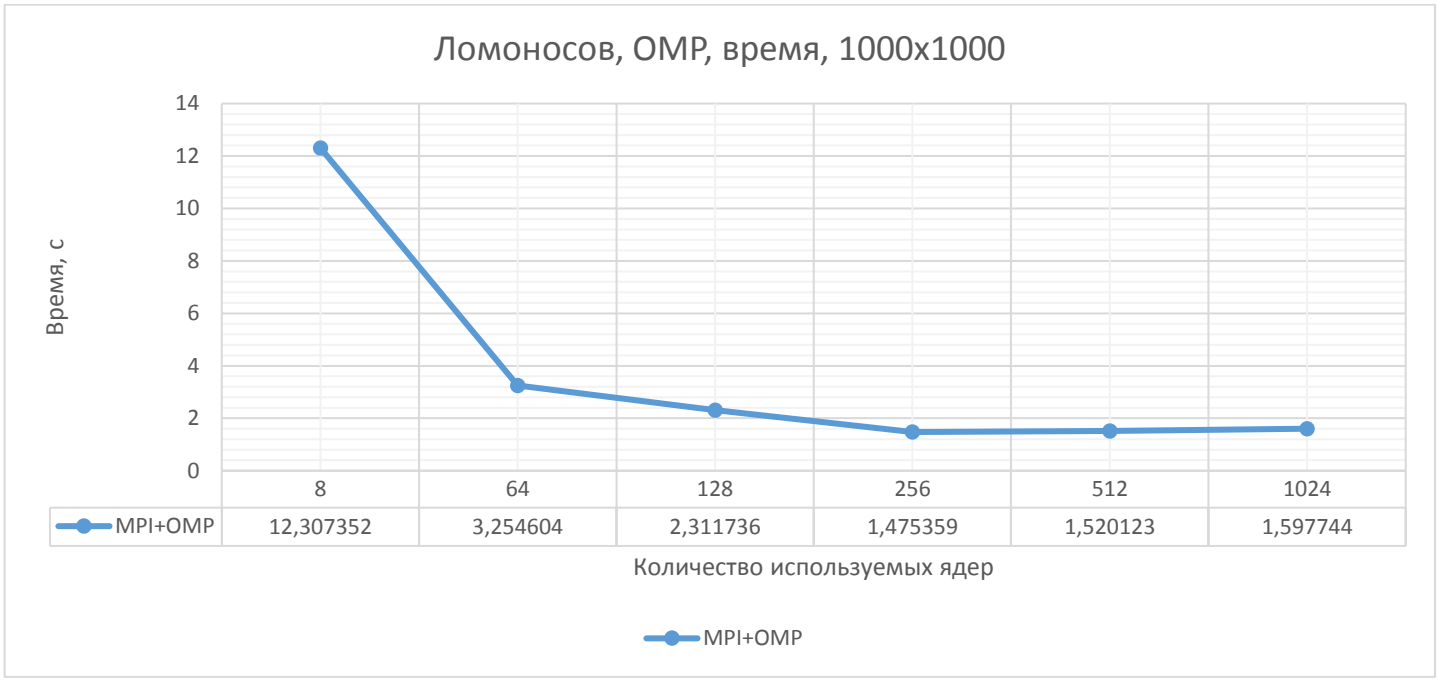

BlueGene, OMP, ускорение, 2000x2000
Ускорение
1800
1600
1400
1200
1000
800
600
400
200
0
3
6
384
768
1536
3072
MPI+OMP
2,970200615
5,901574228
426,2914747
746,295734
913,7723507
1542,196519
Количество используемых ядер
MPI+OMP

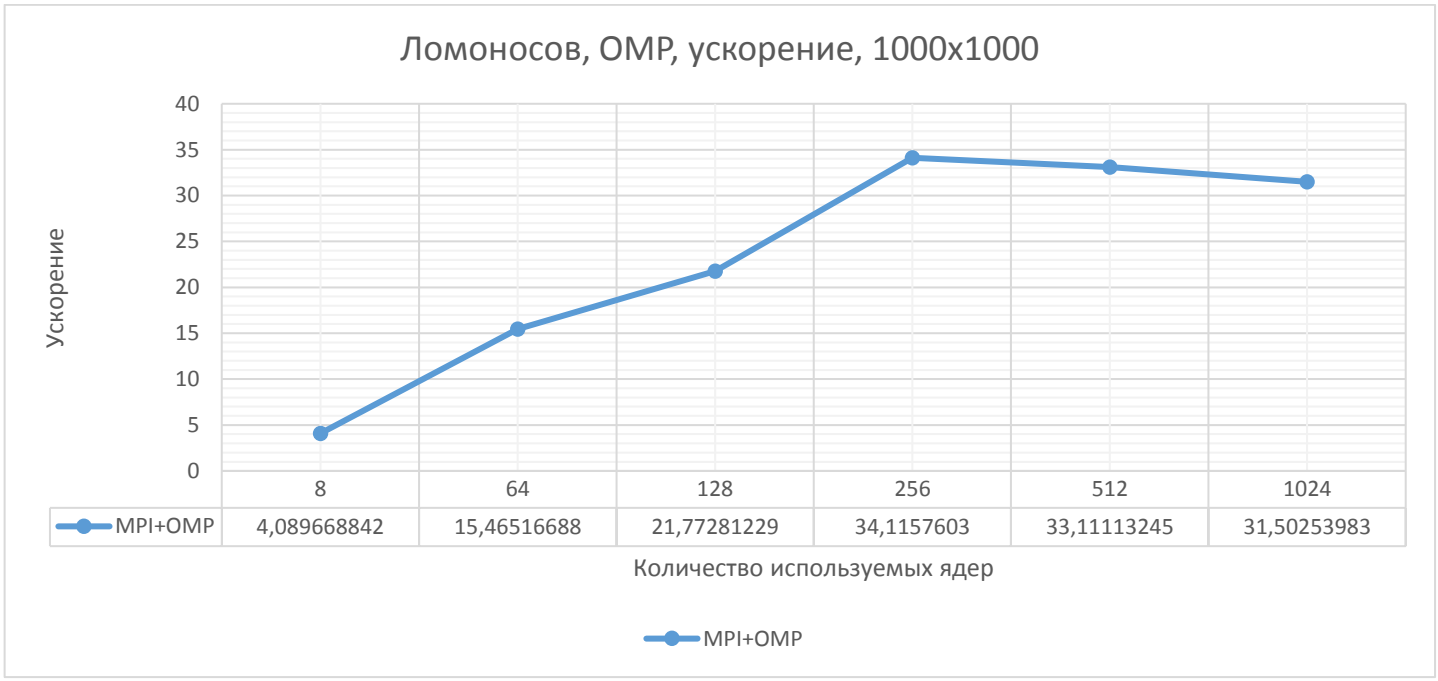

BlueGene, OMP, эффективность, 2000x2000
Эффективность
1,2
1
0,8
0,6
0,4
0,2
0
3
6
384
768
1536
3072
MPI+OMP
0,990066872
0,983595705
1,110134049
0,971739237
0,594903874
0,502017096
Количество используемых ядер
MPI+OMP

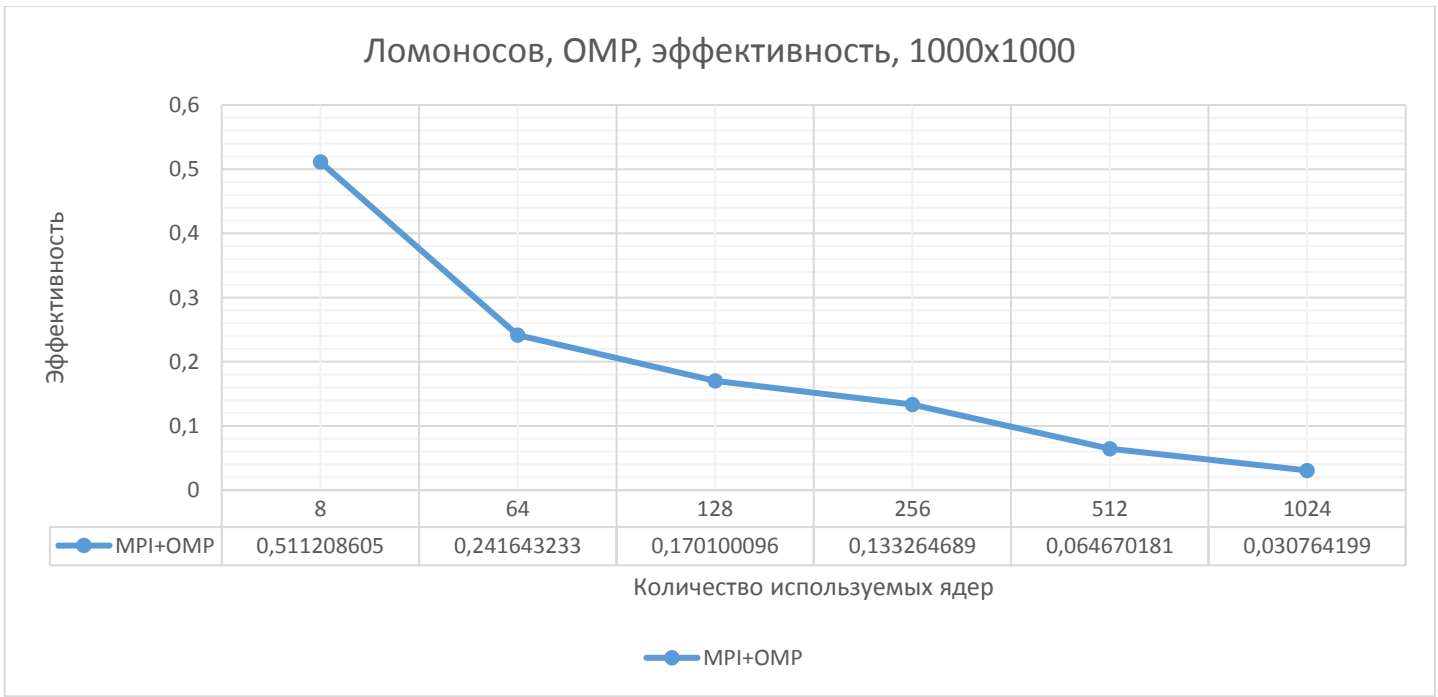

Ломоносов, время, 1000x1000
Время, с
60
50
40
30
20
10
0
1
8
16
32
64
128
MPI
50,332994
7,456423
3,15029
1,605576
0,863473
0,501998
Количество используемых ядер
MPI


Ломоносов, ускорение, 1000x1000
Ускорение
120
100
80
60
40
20
0
1
8
16
32
64
128
MPI
1
6,750286833
15,97725733
31,34887044
58,29133511
100,2653278
Количество используемых ядер
MPI

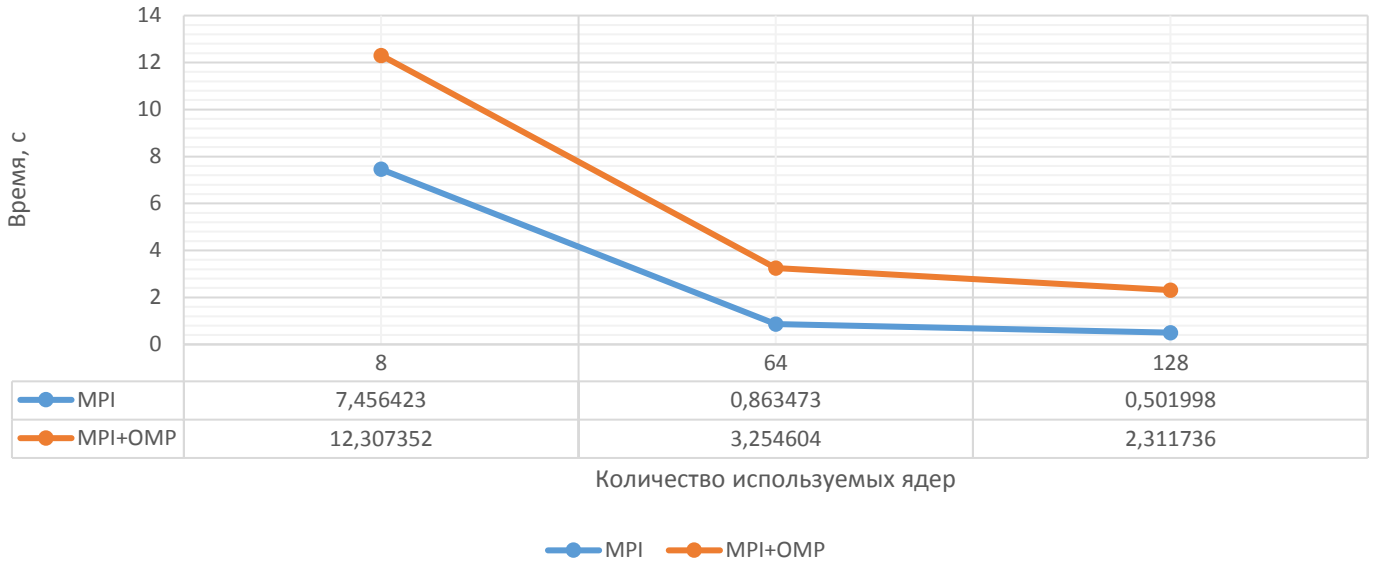

Ломоносов, эффективность, 1000x1000
Эффективность
1,2
1
0,8
0,6
0,4
0,2
0
1
8
16
32
64
128
MPI
1
0,843785854
0,998578583
0,979652201
0,910802111
0,783322873
Количество используемых ядер
MPI


Ломоносов, OMP, время, 1000x1000
Время, с
14
12
10
8
6
4
2
0
8
64
128
256
512
1024
MPI+OMP
12,307352
3,254604
2,311736
1,475359
1,520123
1,597744
Количество используемых ядер
MPI+OMP


Ломоносов, OMP, ускорение, 1000x1000
Ускорение
40
35
30
25
20
15
10
5
0
8
64
128
256
512
1024
MPI+OMP
4,089668842
15,46516688
21,77281229
34,1157603
33,11113245
31,50253983
Количество используемых ядер
MPI+OMP

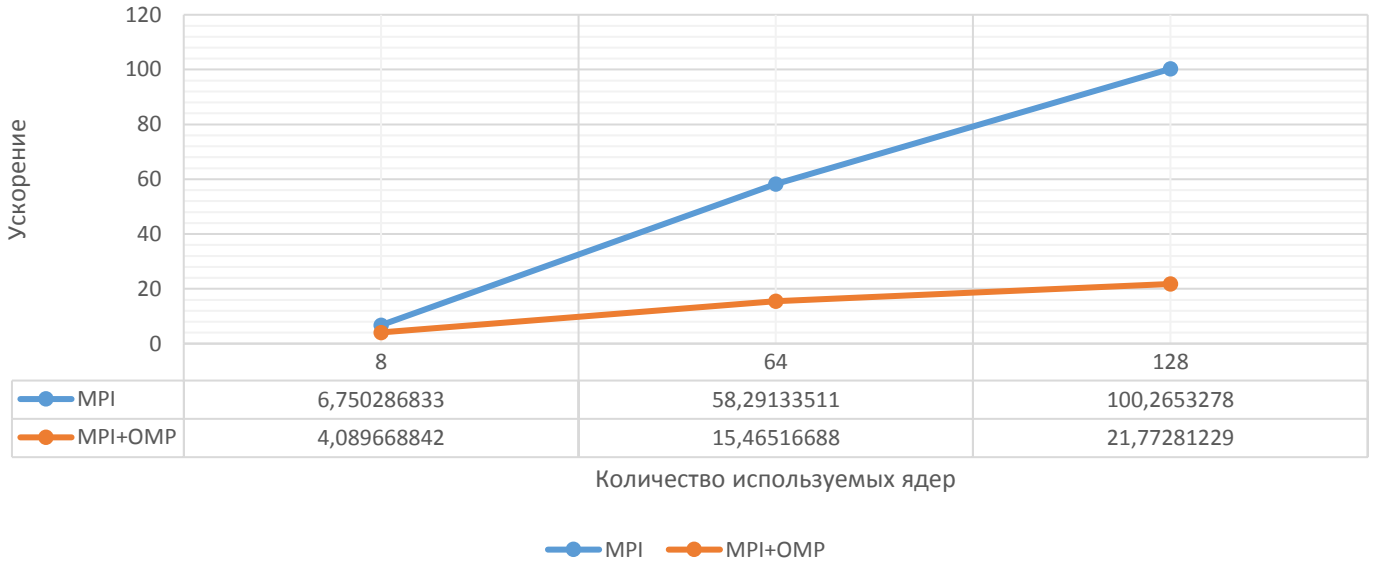

Ломоносов, OMP, эффективность, 1000x1000
Эффективность
0,6
0,5
0,4
0,3
0,2
0,1
0
8
64
128
256
512
1024
MPI+OMP
0,511208605
0,241643233
0,170100096
0,133264689
0,064670181
0,030764199
Количество используемых ядер
MPI+OMP

Ломоносов, время, 1000x1000



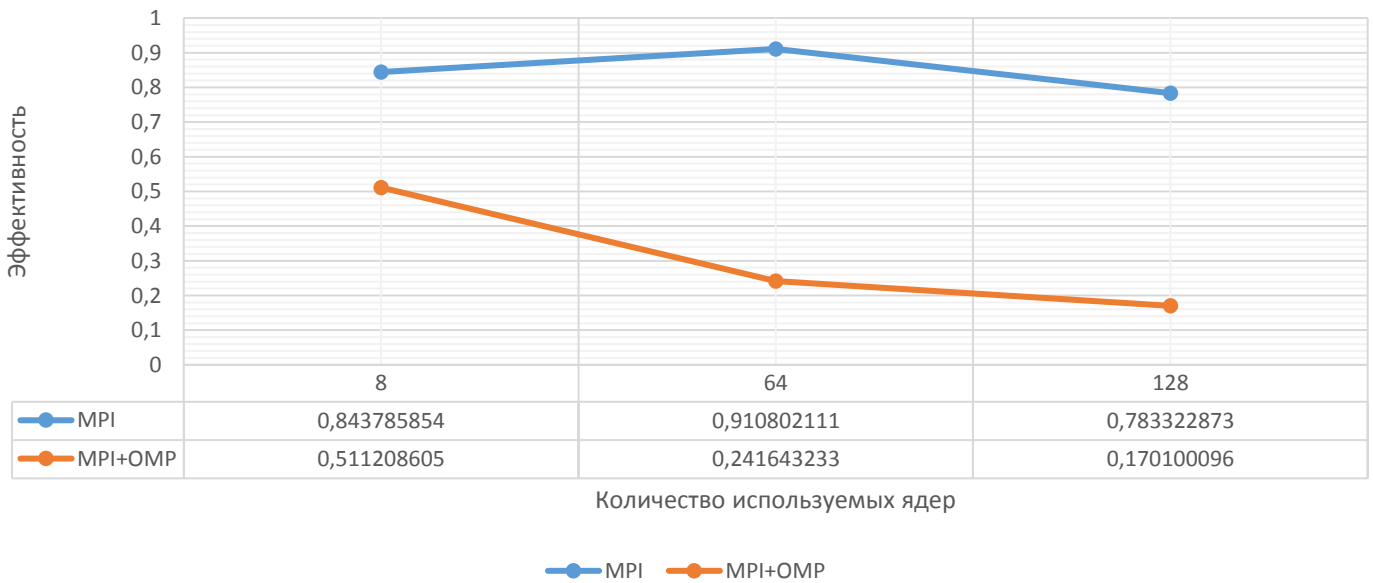
	8	64	128
MPI	7,456423	0,863473	0,501998
MPI+OMP	12,307352	3,254604	2,311736

Ломоносов, ускорение, 1000x1000



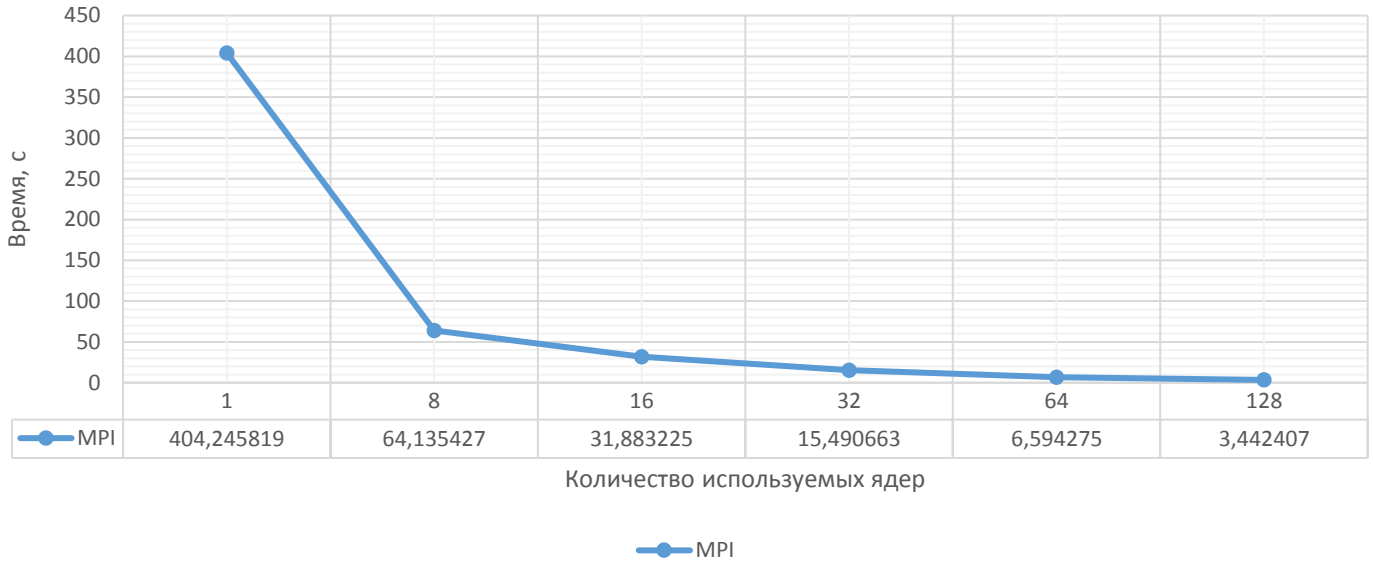
	8	64	128
MPI	6,750286833	58,29133511	100,2653278
MPI+OMP	4,089668842	15,46516688	21,77281229

Ломоносов, эффективность, 1000x1000

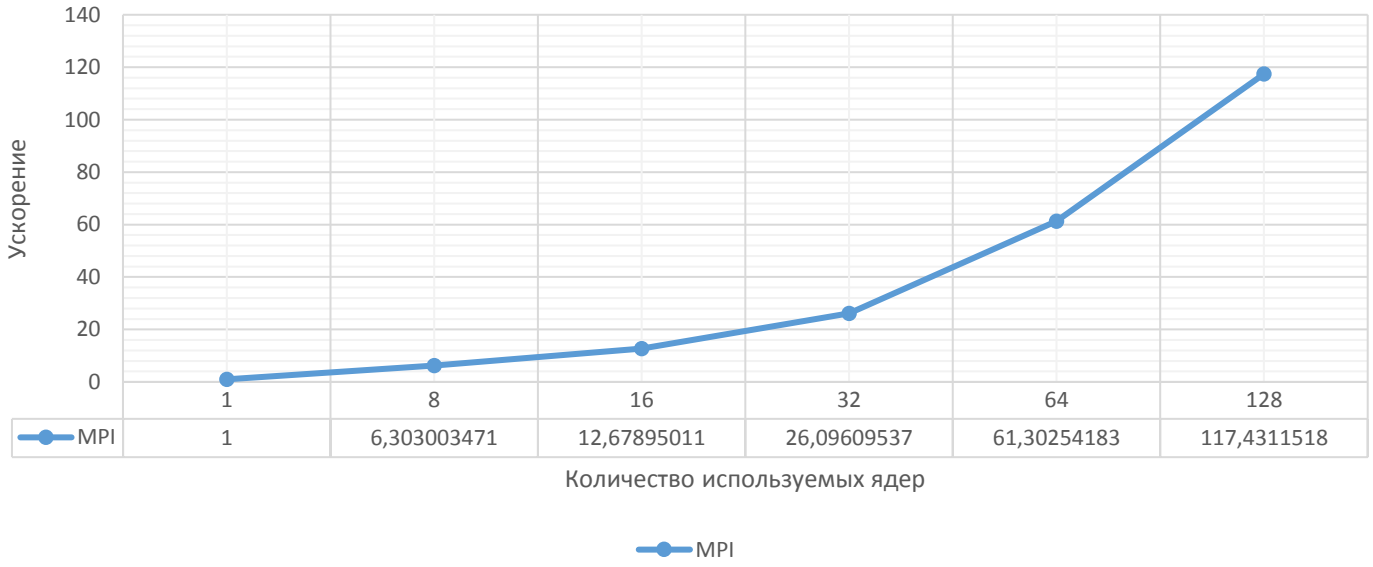


	8	64	128
MPI	0,843785854	0,910802111	0,783322873
MPI+OMP	0,511208605	0,241643233	0,170100096

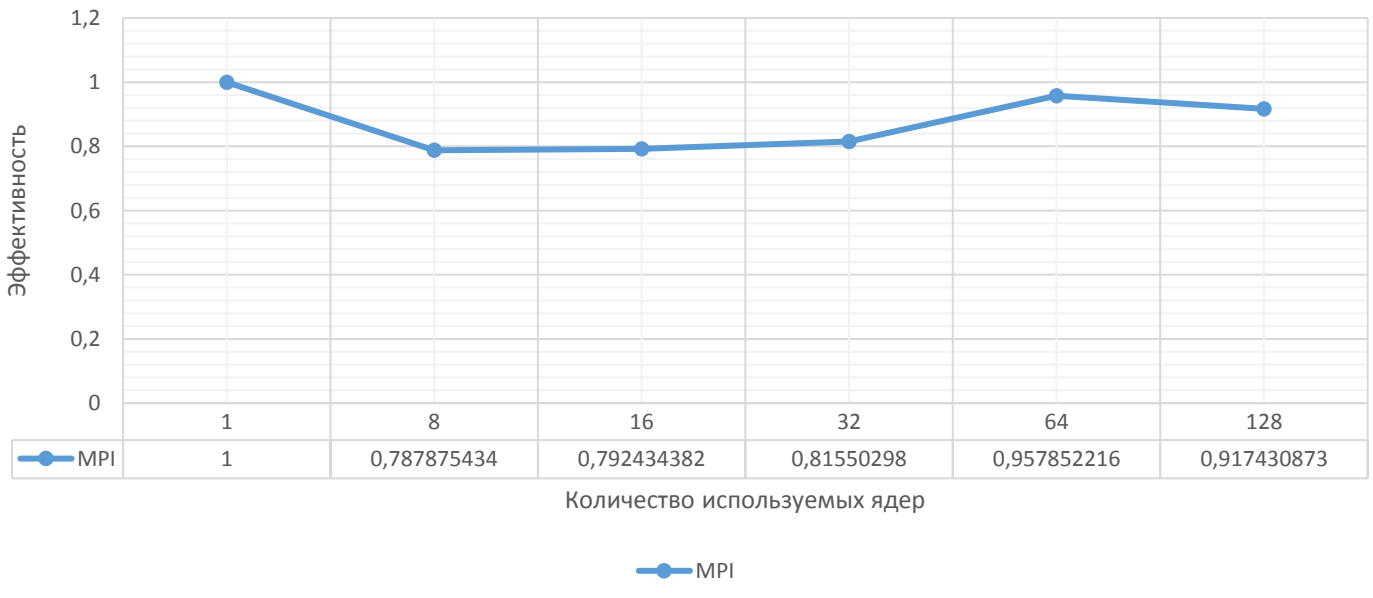
Ломоносов, время, 2000x2000

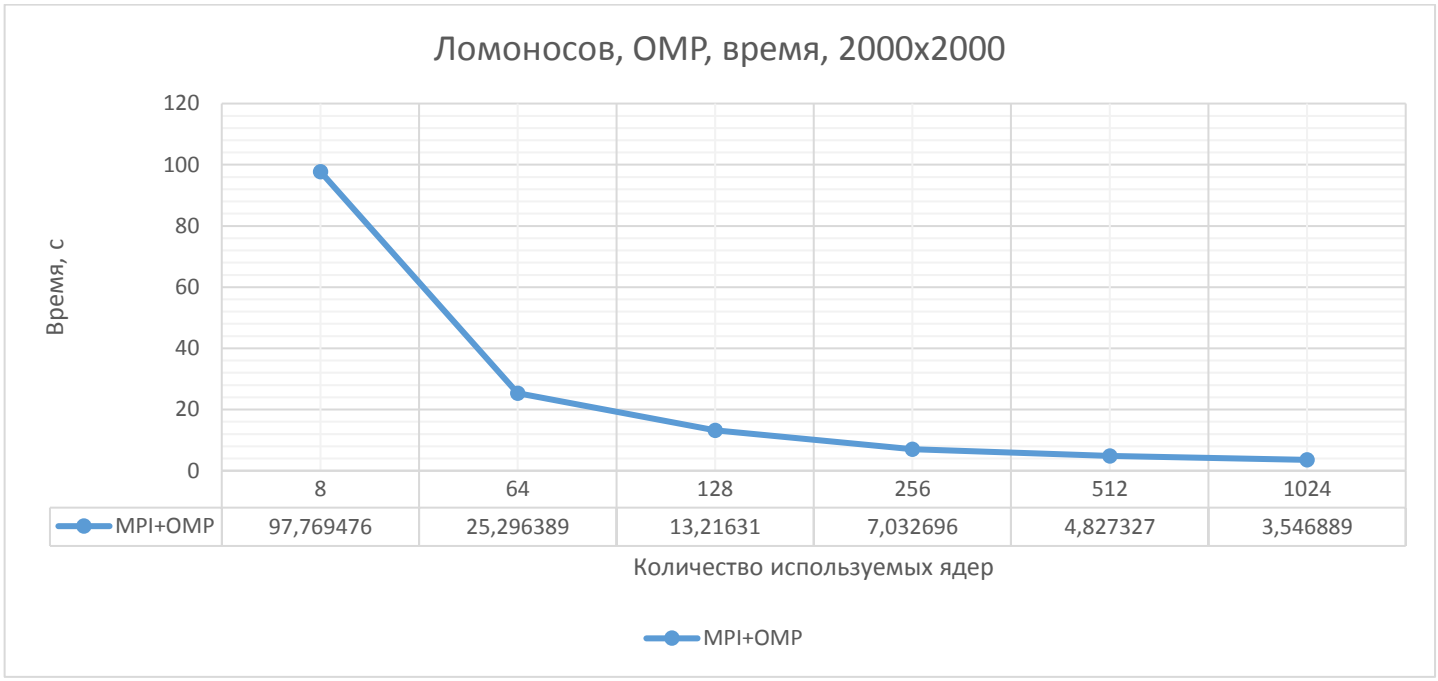


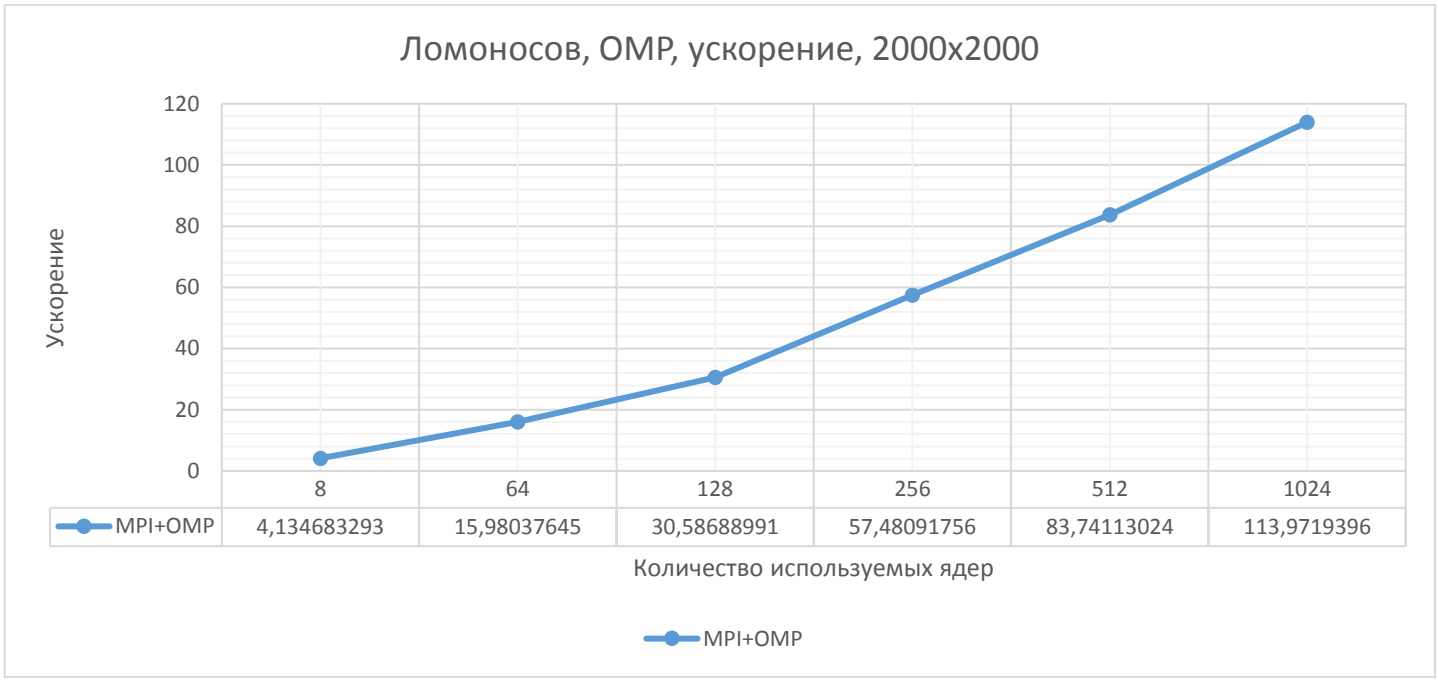
Ломоносов, ускорение, 2000x2000

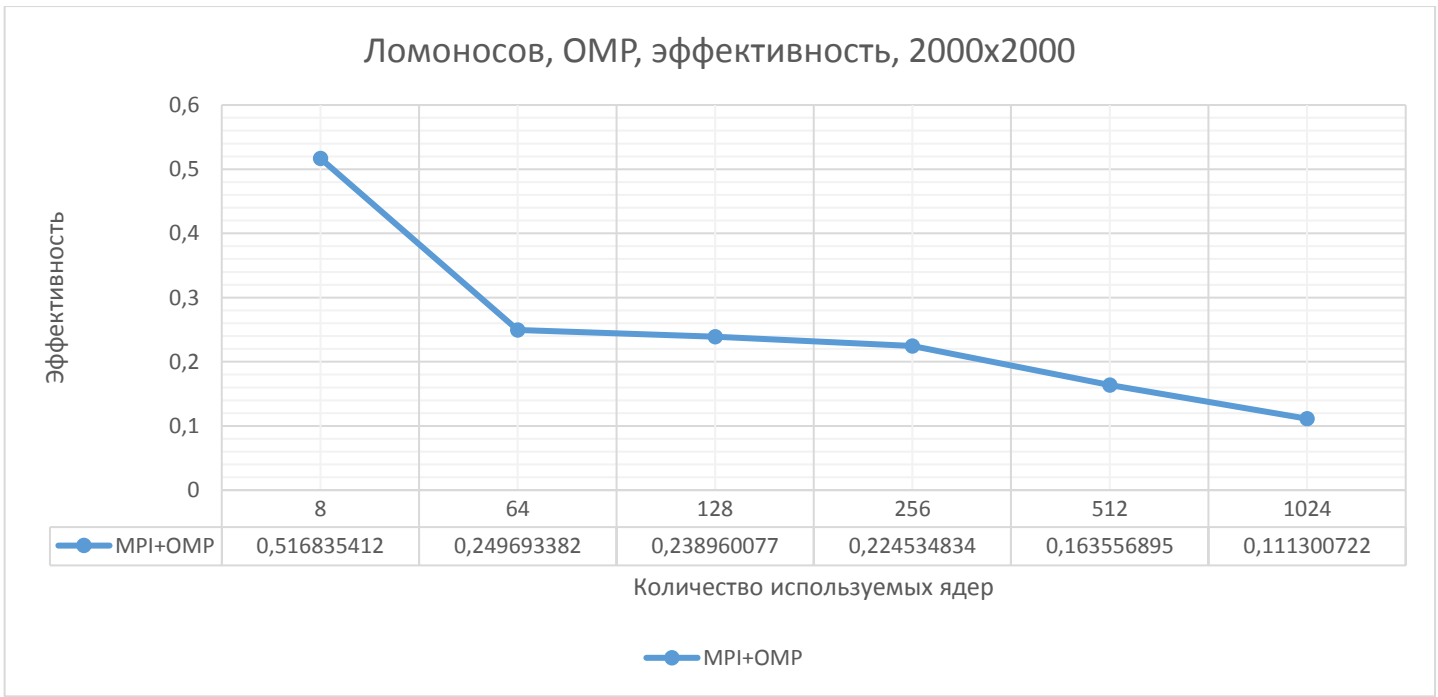


Ломоносов, эффективность, 2000x2000

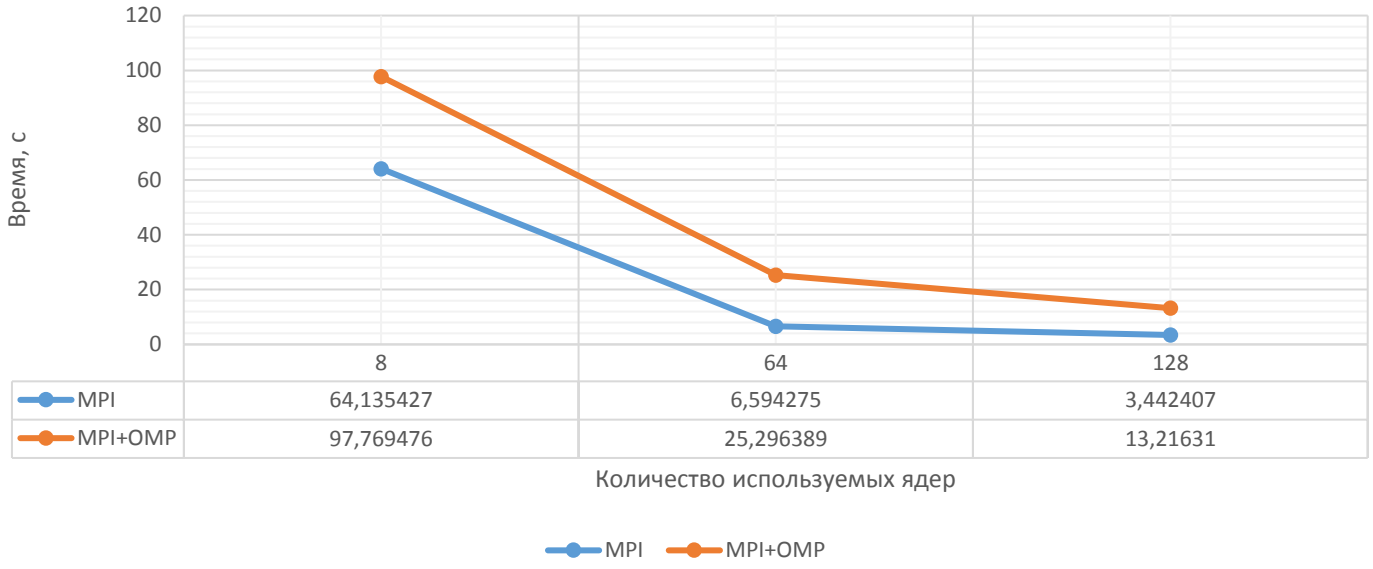



Ломоносов, OMP, время, 2000x2000
Время, с
120
100
80
60
40
20
0
8
64
128
256
512
1024
MPI+OMP
97,769476
25,296389
13,21631
7,032696
4,827327
3,546889
Количество используемых ядер
MPI+OMP

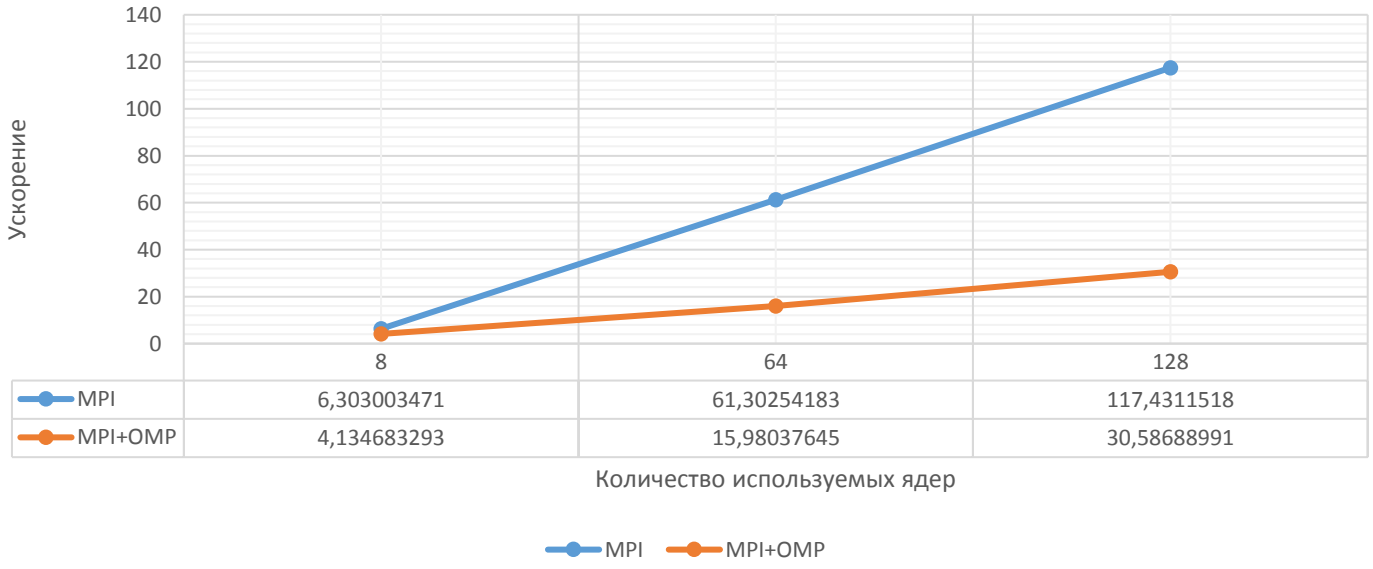

Ломоносов, OMP, ускорение, 2000x2000
Ускорение
120
100
80
60
40
20
0
8
64
128
256
512
1024
MPI+OMP
4,134683293
15,98037645
30,58688991
57,48091756
83,74113024
113,9719396
Количество используемых ядер
MPI+OMP


Ломоносов, OMP, эффективность, 2000x2000
Эффективность
0,6
0,5
0,4
0,3
0,2
0,1
0
8
64
128
256
512
1024
MPI+OMP
0,516835412
0,249693382
0,238960077
0,224534834
0,163556895
0,111300722
Количество используемых ядер
MPI+OMP

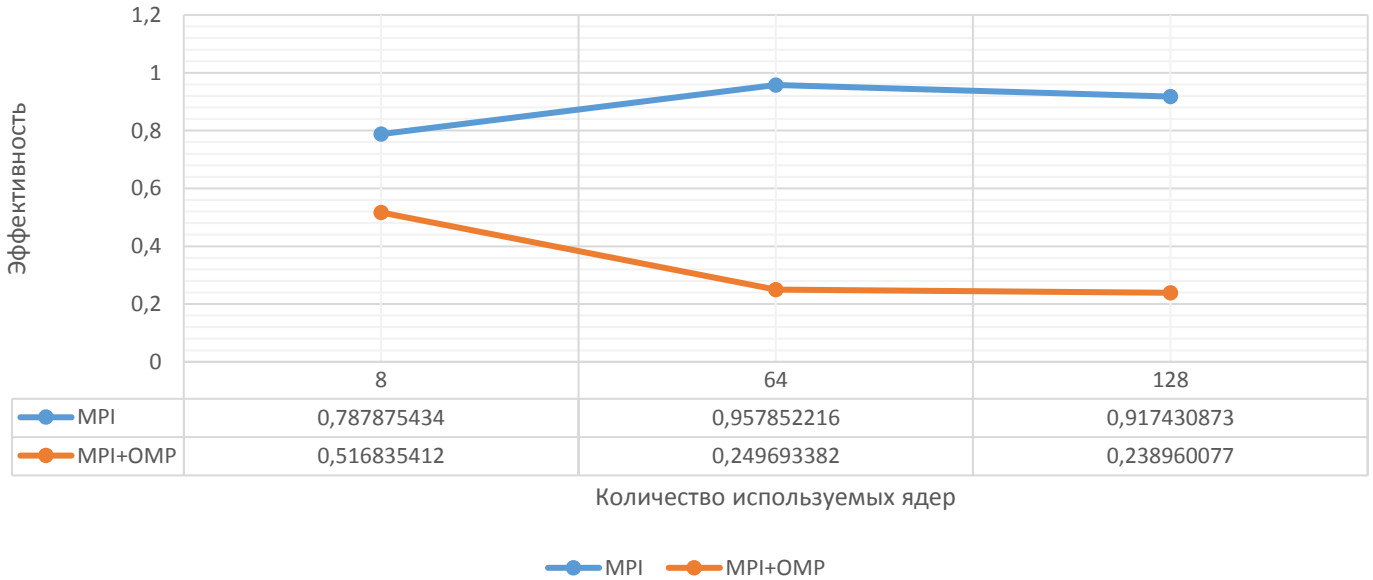
Ломоносов, время, 2000x2000

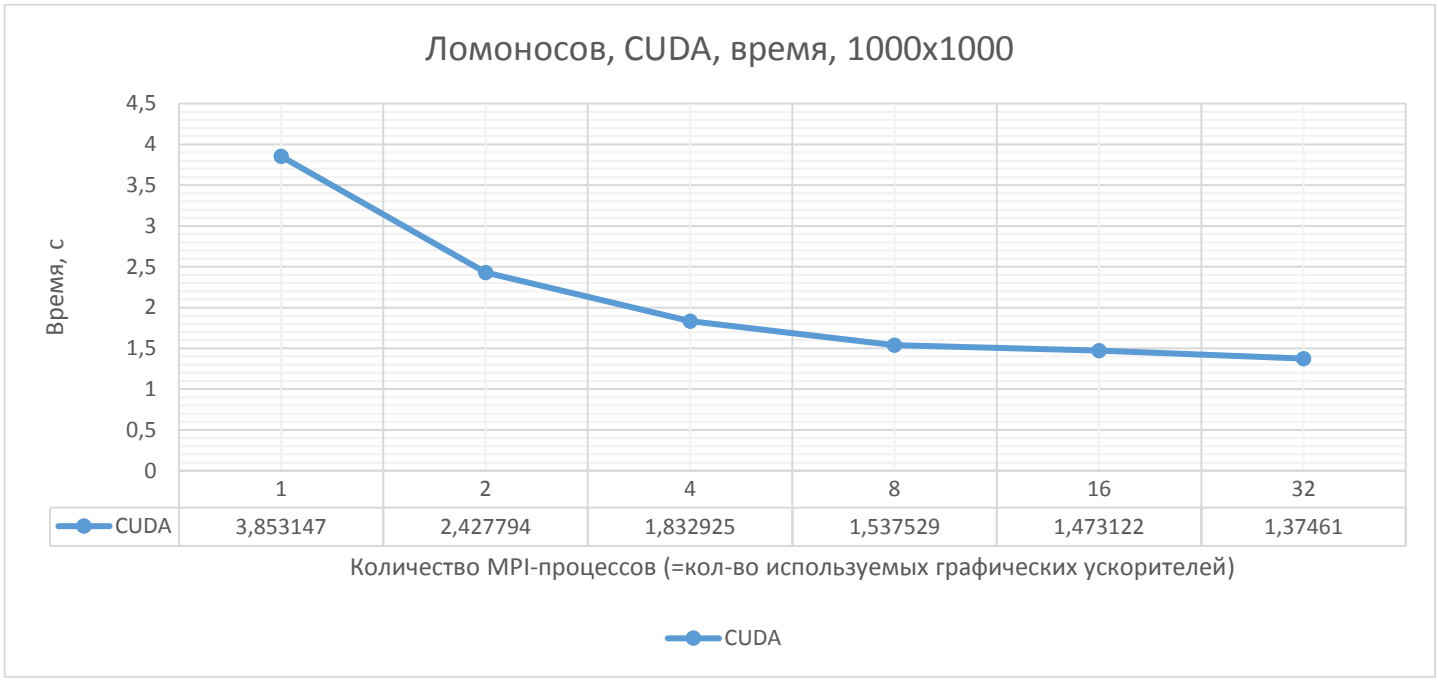


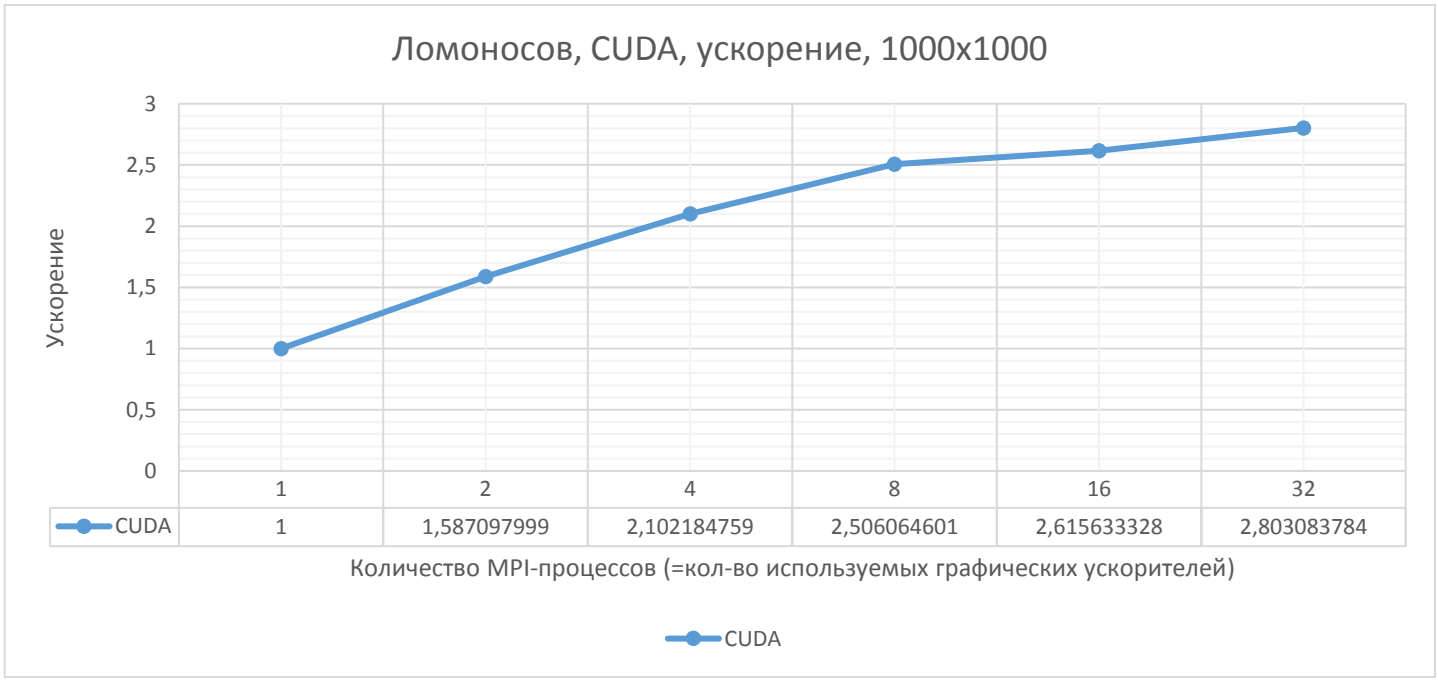
Ломоносов, ускорение, 2000x2000

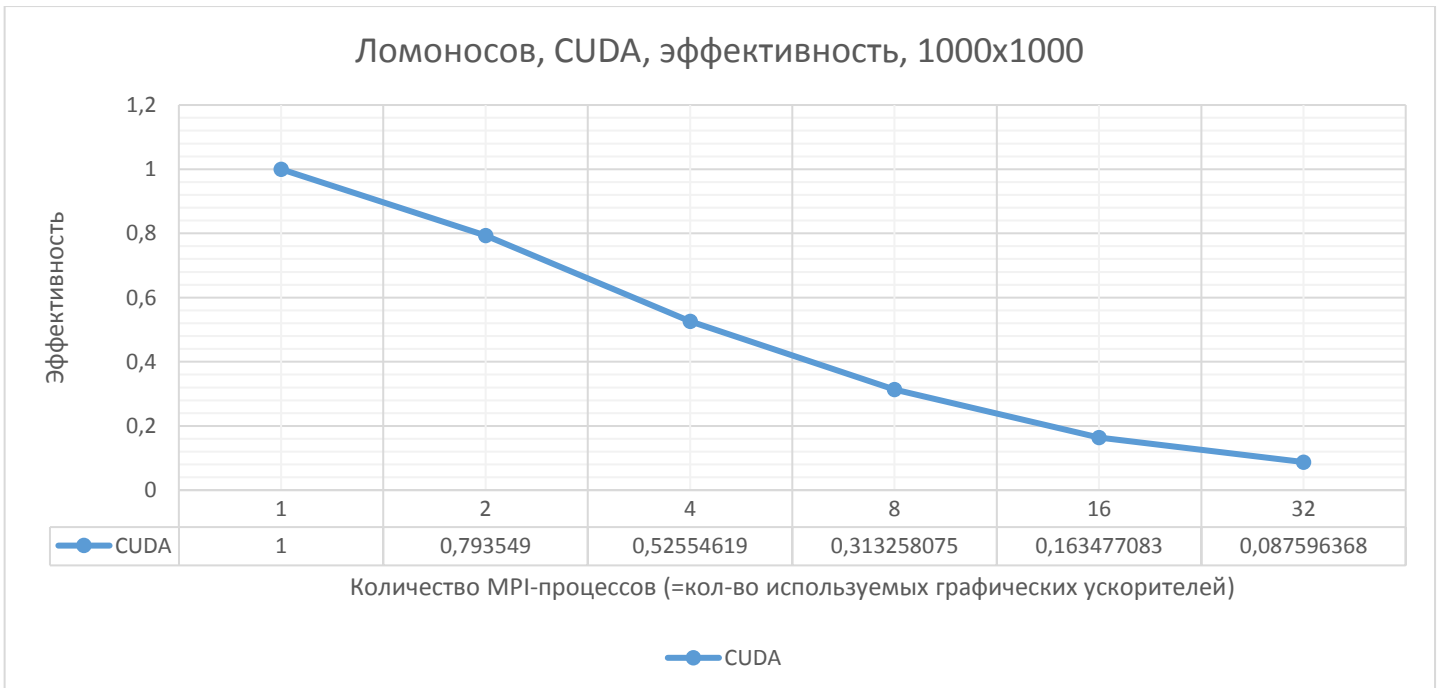


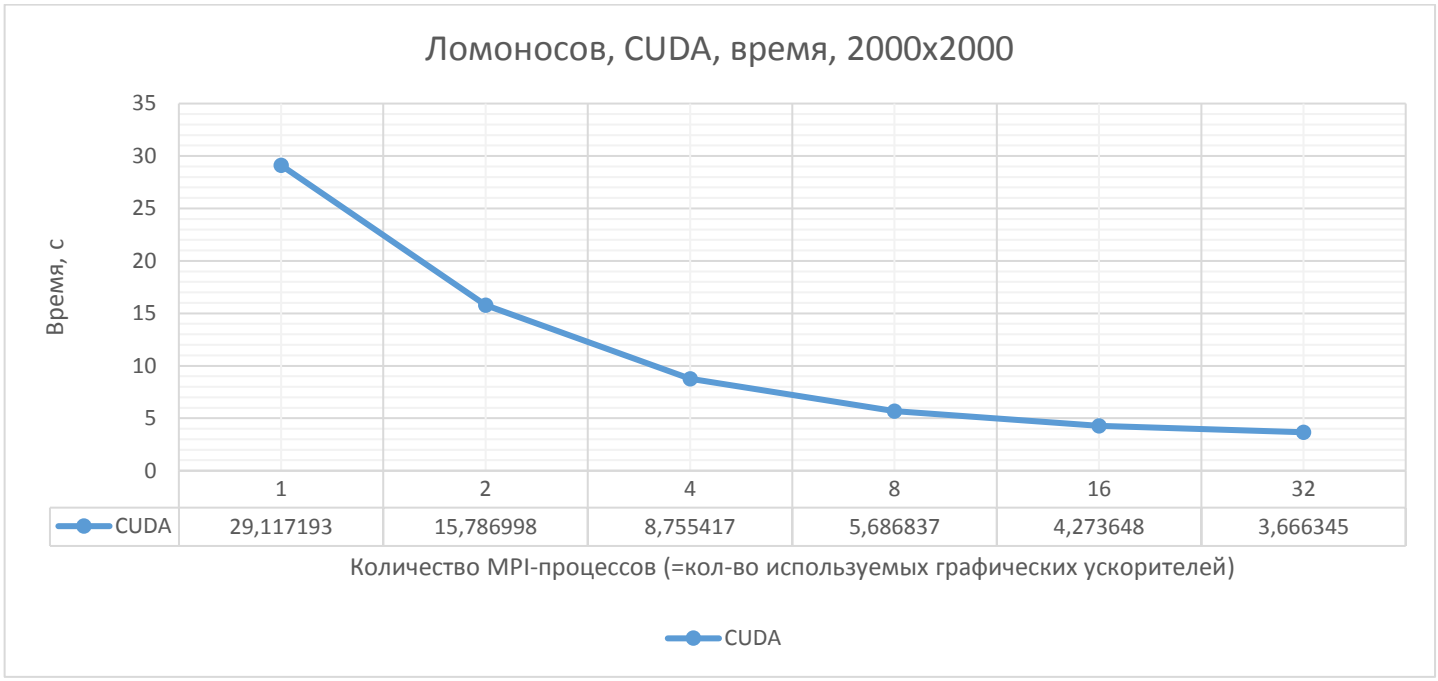
Ломоносов, эффективность, 2000x2000

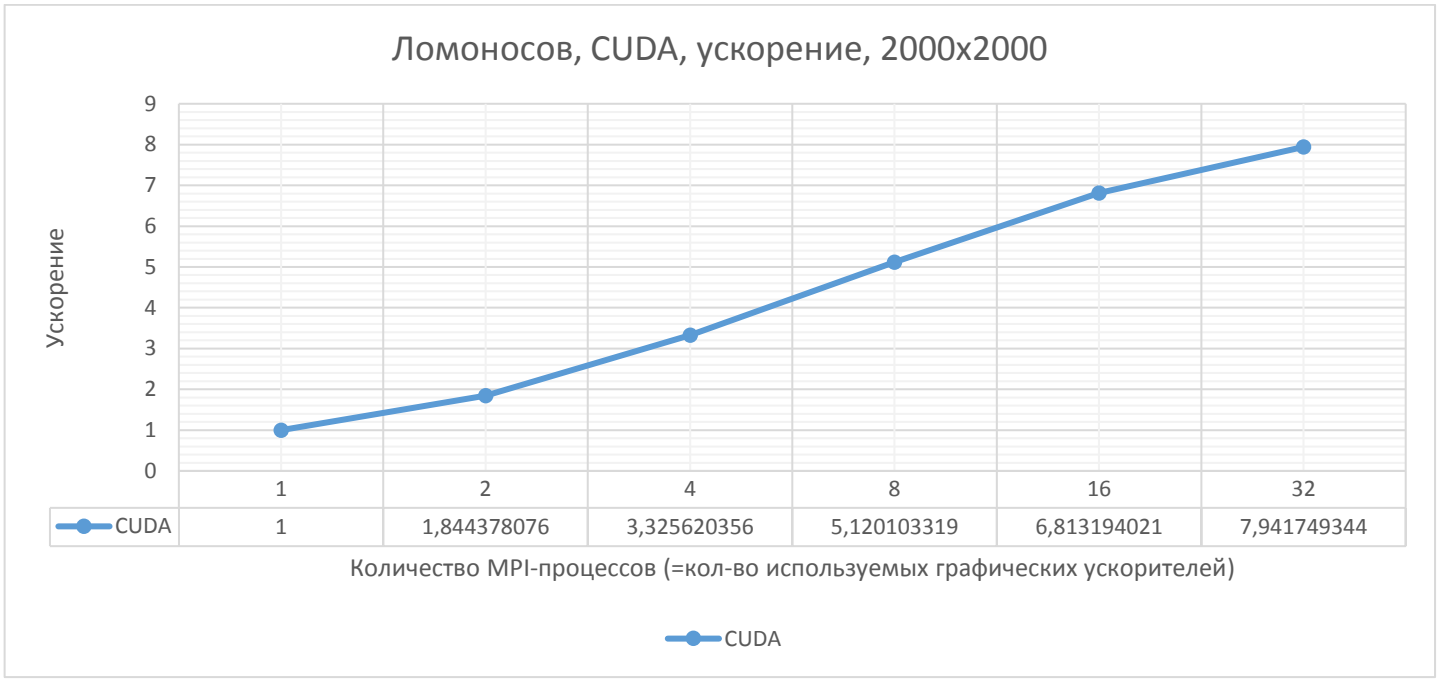


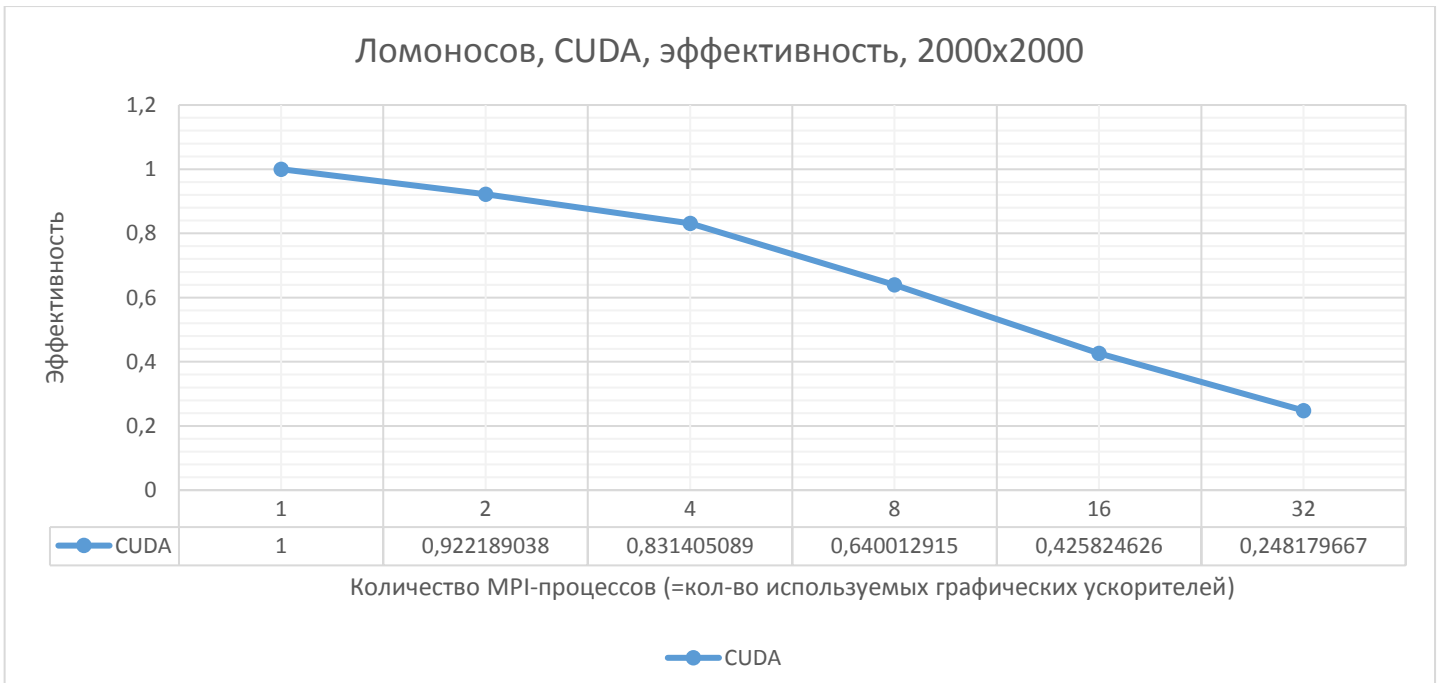

Ломоносов, CUDA, время, 1000x1000
Время, с
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
1
2
4
8
16
32
CUDA
3,853147
2,427794
1,832925
1,537529
1,473122
1,37461
Количество MPI-процессов (=кол-во используемых графических ускорителей)
CUDA

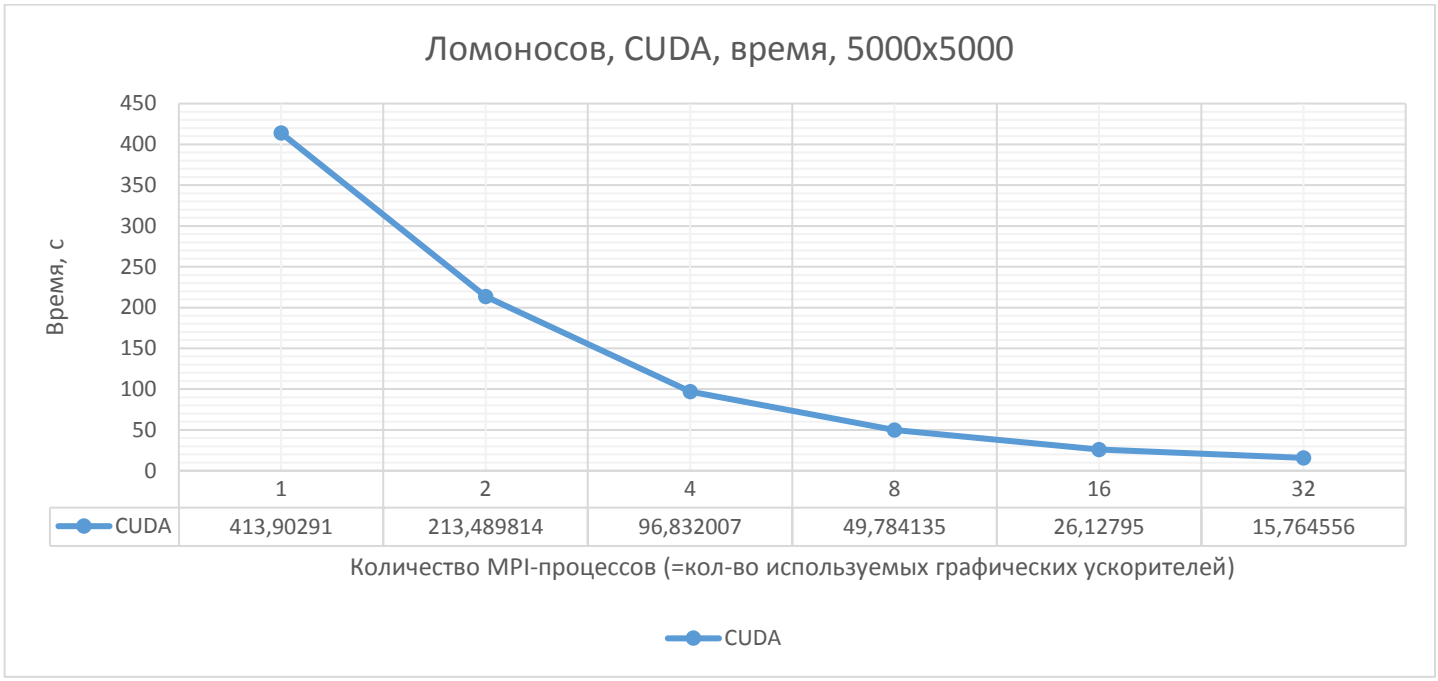

Ломоносов, CUDA, ускорение, 1000x1000
Ускорение
3
2,5
2
1,5
1
0,5
0
1
2
4
8
16
32
CUDA
1
1,587097999
2,102184759
2,506064601
2,615633328
2,803083784
Количество MPI-процессов (=кол-во используемых графических ускорителей)
CUDA

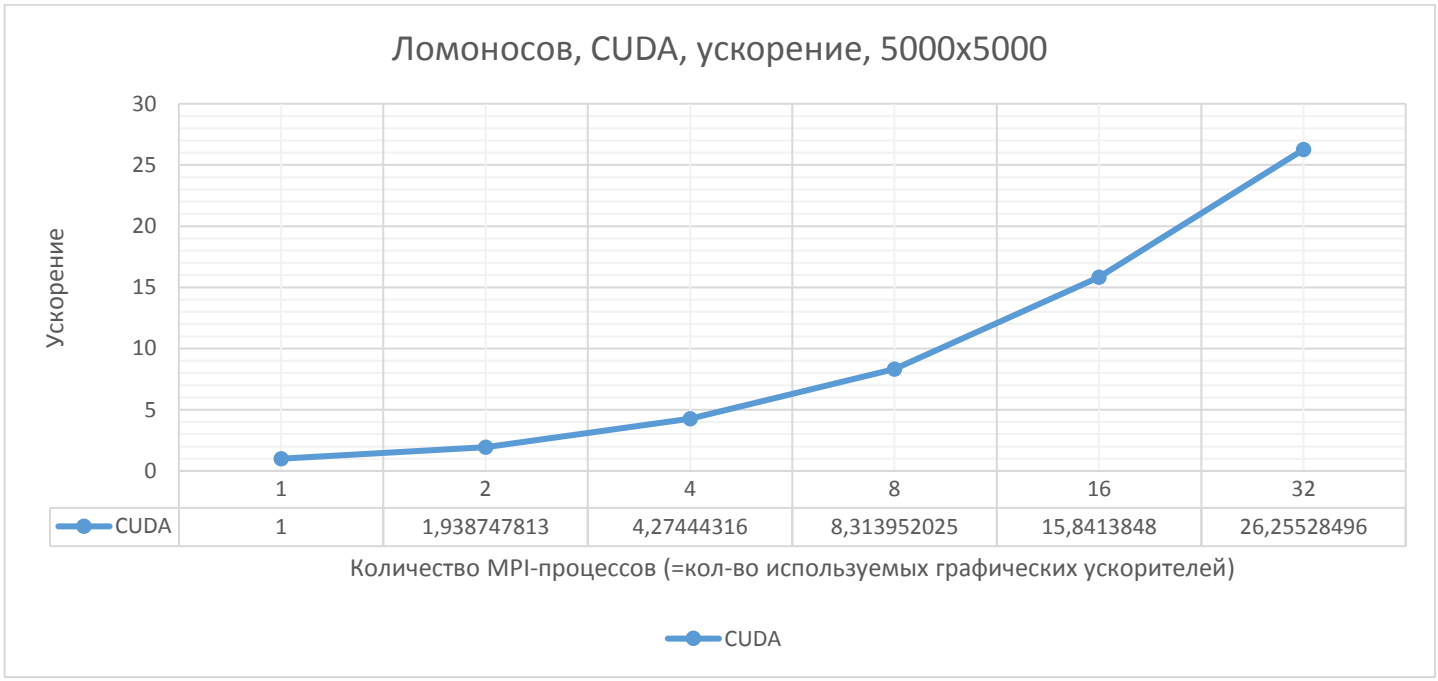

Ломоносов, CUDA, эффективность, 1000x1000
Эффективность
1,2
1
0,8
0,6
0,4
0,2
0
1
2
4
8
16
32
CUDA
1
0,793549
0,52554619
0,313258075
0,163477083
0,087596368
Количество MPI-процессов (=кол-во используемых графических ускорителей)
CUDA

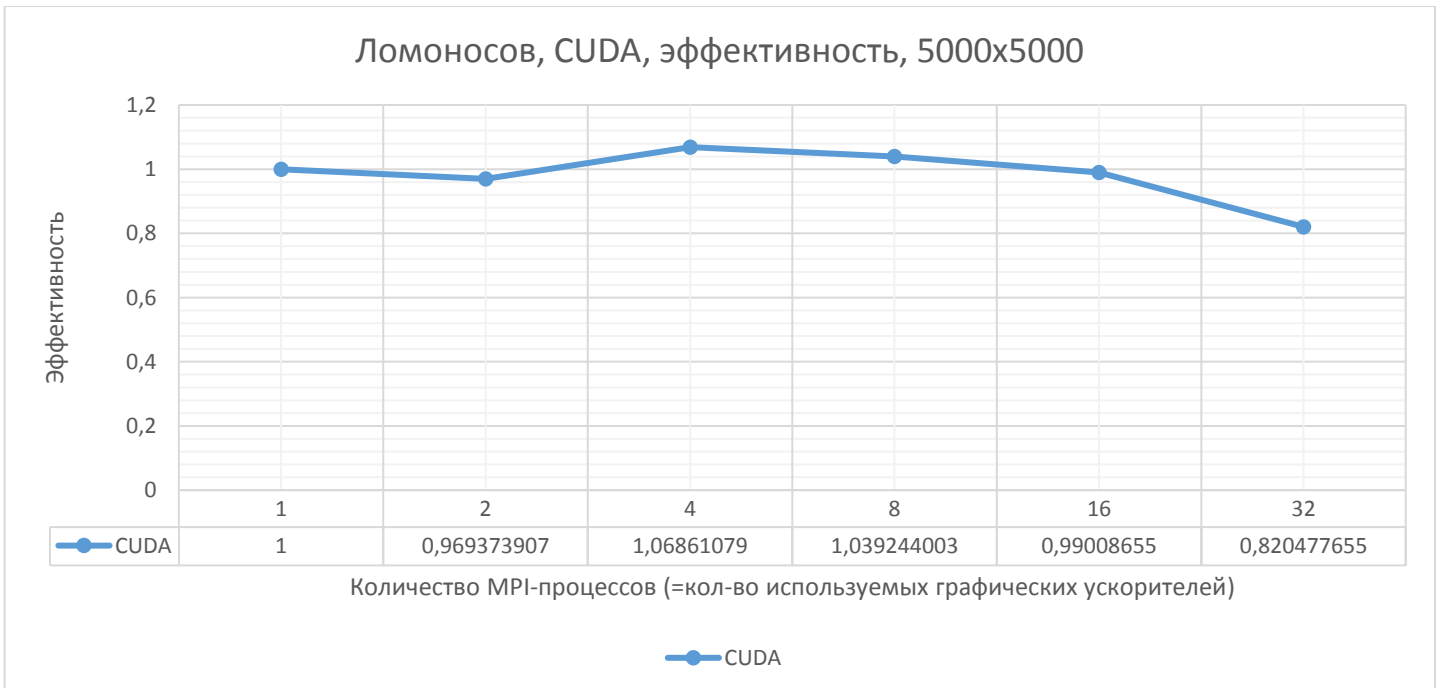

Ломоносов, CUDA, время, 2000x2000
Время, с
35
30
25
20
15
10
5
0
1
2
4
8
16
32
CUDA
29,117193
15,786998
8,755417
5,686837
4,273648
3,666345
Количество MPI-процессов (=кол-во используемых графических ускорителей)
CUDA


Ломоносов, CUDA, ускорение, 2000x2000
Ускорение
9
8
7
6
5
4
3
2
1
0
1
2
4
8
16
32
CUDA
1
1,844378076
3,325620356
5,120103319
6,813194021
7,941749344
Количество MPI-процессов (=кол-во используемых графических ускорителей)
CUDA


Ломоносов, CUDA, эффективность, 2000x2000
Эффективность
1,2
1
0,8
0,6
0,4
0,2
0
1
2
4
8
16
32
CUDA
1
0,922189038
0,831405089
0,640012915
0,425824626
0,248179667
Количество MPI-процессов (=кол-во используемых графических ускорителей)
CUDA


Ломоносов, CUDA, время, 5000x5000
Время, с
450
400
350
300
250
200
150
100
50
0
1
2
4
8
16
32
CUDA
413,90291
213,489814
96,832007
49,784135
26,12795
15,764556
Количество MPI-процессов (=кол-во используемых графических ускорителей)
CUDA


Ломоносов, CUDA, ускорение, 5000x5000
Ускорение
30
25
20
15
10
5
0
1
2
4
8
16
32
CUDA
1
1,938747813
4,27444316
8,313952025
15,8413848
26,25528496
Количество MPI-процессов (=кол-во используемых графических ускорителей)
CUDA


Ломоносов, CUDA, эффективность, 5000x5000
Эффективность
1,2
1
0,8
0,6
0,4
0,2
0
1
2
4
8
16
32
CUDA
1
0,969373907
1,06861079
1,039244003
0,99008655
0,820477655
Количество MPI-процессов (=кол-во используемых графических ускорителей)
CUDA

Выводы

BlueGene:

Версия без OMP показывает значительное, почти линейное увеличение ускорения с ростом количества используемых процессоров. Эффективность выше 100% означает, что блоки обрабатываемых данных начали полностью помещаться в кэш процессоров, либо значительно уменьшилось (стало редким) количество обращений в оперативную память, что привело к суперлинейному ускорению программы. Наибольшую эффективность демонстрирует конфигурация с 128 процессами, при дальнейшем росте количества используемых ядер накладные расходы на коммуникации превосходят полезное время (пересылаются очень маленькие объемы данных, но для каждой пересылки существует ненулевая латентность сети).

Версия с OMP имеет тенденцию к падению эффективности, в основном, из-за двух причин: накладные расходы для создания нитей, когда блок данных уже достаточно мал для быстрой обработки в один поток, и отсутствие клаузы collapse, из-за чего рассматривается неполное пространство витков (параллелится только внешний цикл, а внутренний выполняется целиком одной нитью, при этом внутренний может быть затратным по времени). Можно было бы переписать циклы, сделав один цикл по количеству элементов в блоке, но в этом случае нам потребуется дополнительно находить матричные координаты в блоке (т.к. в линейном участке памяти будут периодически попадаться два пустых элемента – фрагменты вертикальных теневых граней, которые нужно пропускать) посредством операций деления и взятия остатка, что дополнительно замедлит программу (в версии с CUDA, тем не менее, ядра без вложенных циклов и на каждом шаге выполняются эти операции для нахождения матричных координат элемента в массиве, но быстрота выровненного доступа к памяти по сравнению с невыровненным более чем полностью компенсирует эти целочисленные операции (вообще говоря, не очень дружелюбные операции для GPU)).

Lomonosov:

Версия без OMP показывает высокую эффективность, однако суперлинейное ускорение отсутствует, вероятно, из-за нестабильности коммуникационной среды (параллельно с выполнением расчетов сеть могла быть нагружена другими задачами и пересылками, в отличие от BlueGene, реализующего (имитирующего) монопольный доступ программы как к узлу, так и к сети). При увеличении количества процессов значение ускорения стабильно растет, но, судя по эффективности на 128 процессах, данные, обрабатываемые одним процессом, уже недостаточно велики, чтобы загрузить ядро работой, поэтому накладные расходы на пересылки будут расти, а эффективность использования ресурсов – падать.

Версия с OMP неэффективна на большом количестве MPI-процессов, т.к. количество обрабатываемых одним процессом данных слишком мало для продолжительных вычислений, гораздо быстрее обработать их в один поток.

Версия с CUDA показывает удивительное быстродействие по сравнению с CPU-версией, но эффективность невелика из-за малого количества обрабатываемых данных. Дополнительный расчет на сетке 5000x5000 показывает достаточно высокую эффективность по сравнению с меньшими сетками. Так как на одном графическом ускорителе может быть запущено 1536 * 14 = 21504 параллельных нитей, имеем 47 элементов для 1000x1000, 187 для 2000x2000 и 1163 для 5000x5000. На последнем графике видно, что эффективность достигает максимума при 4 MPI-процессах (=4 графических ускорителях). Разделив сетку 5000x5000 на 4, а затем на 21504, мы получим 291 элемент на нить, т.е. для эффективного использования MPI и CUDA 187 элементов на нить мало, а 291 достаточно, значит, существует «золотая середина» либо на этом отрезке, либо >291, на которой достигается максимум эффективности для данной конфигурации аппаратуры.