Национальный исследовательский ядерный университет «МИФИ» (Московский Инженерно–Физический Институт)

Кафедра №42 «Криптология и кибербезопасность»

**Лабораторная работа №7:**

**«Совместное применение OpenMP и MPI»**

Антон Гатченко Б22-525

2024 г.

*Используемая рабочая среда:*

* Процессор - AMD Ryzen 5 5600H (laptop), 6c/12t
* Оперативная память – DDR4 16 ГБ
* ОС - Windows 10 Pro 22H2 19045.4780, 64 bit
* IDE – GCC/G++ 13.1, OpenMP 201511, Microsoft MPI Version 10.1.12498.16

*Ход работы:*

В ходе работы проводились замеры времени выполнения программы, написанной с использованием технологий MSMPI и OpenMP, по поиску простых чисел в заданном диапазоне. Используемый тип данных - целое беззнаковое число (unsigned int 4 byte). Верхняя граница поиска: . Потоки MPI изменялись от 1 до 6, для OpenMP выделялись 2 потока.

Поскольку MPI затрачивает довольно много времени на пересылку массивов, было выполнено таймирование конкретных частей, где происходил конкретно поиск простых чисел, чтобы сравнить плюсы от распараллеливания без влияния этой особенности реализации MPI.

*Теоретическая сложность алгоритма: .*

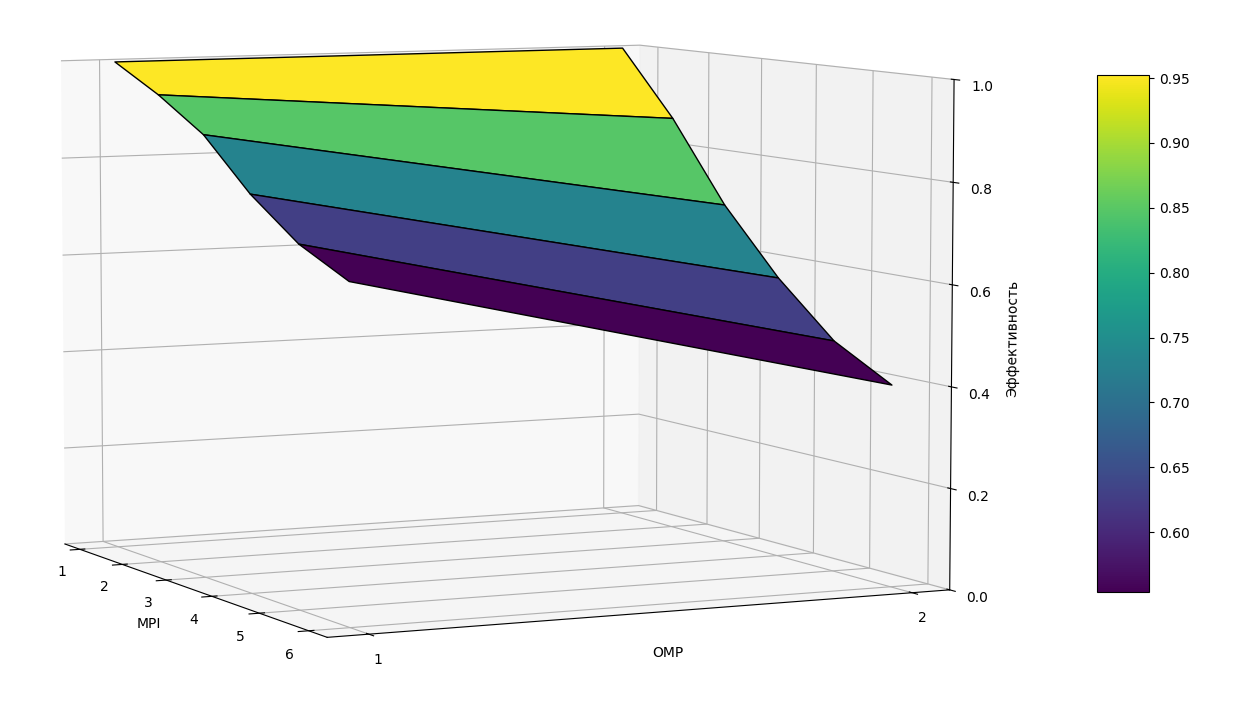
– верхняя граница диапазона, – количество предварительно посчитанных простых чисел до ; Количество простых чисел от до асимптотически приближаетсяк .

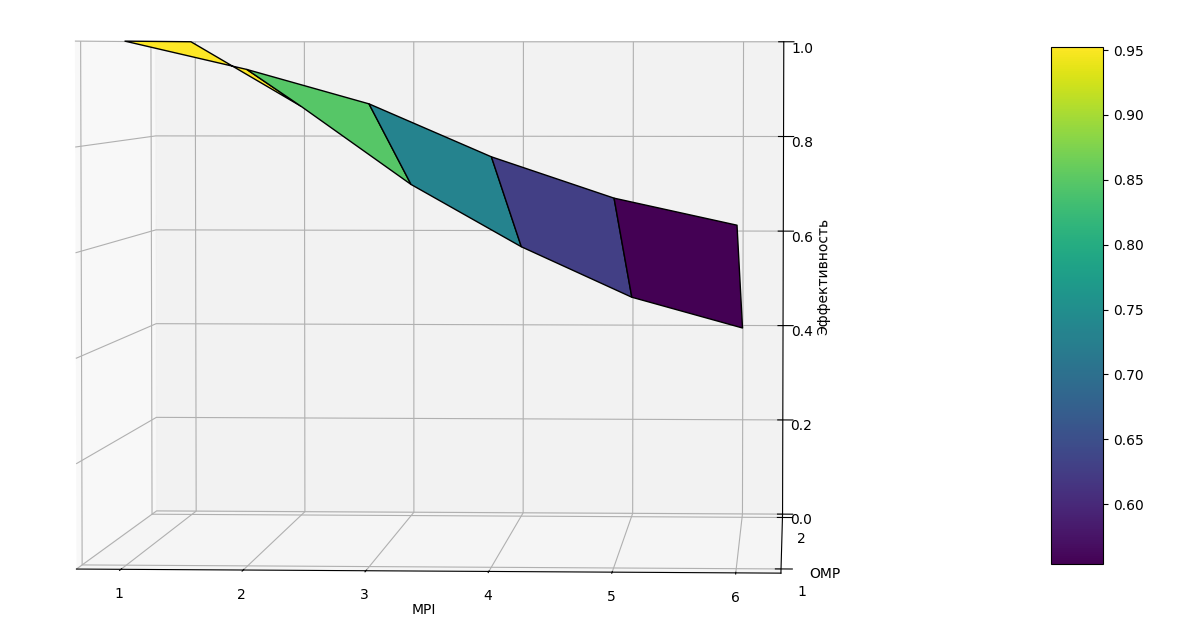
*Использованные функции MSMPI:*

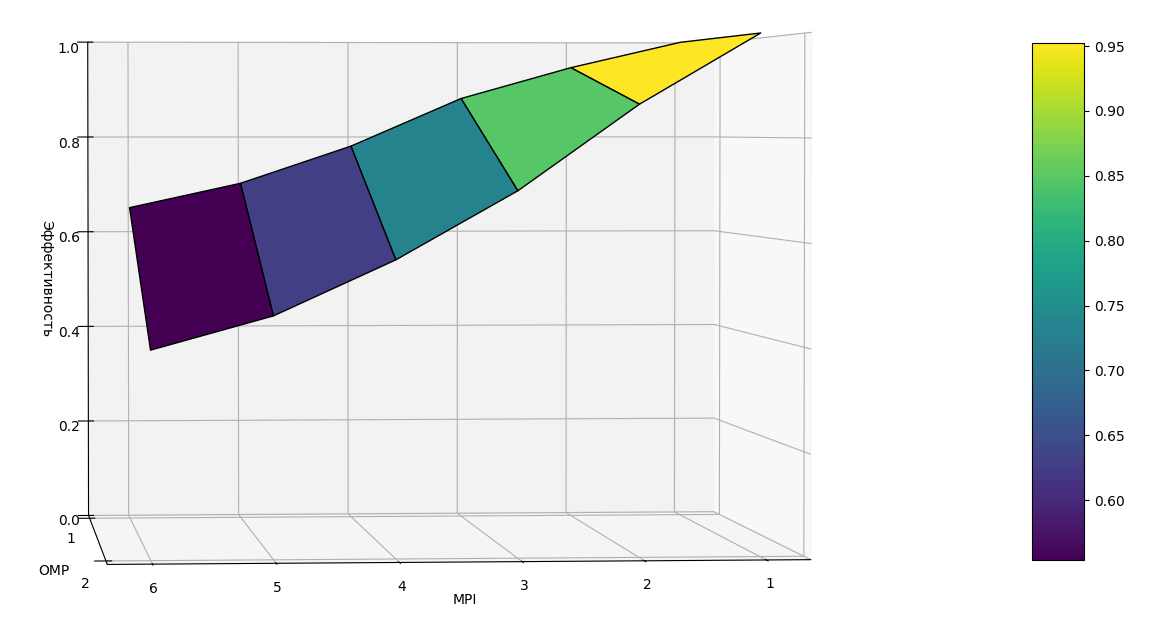
* MPI\_Init() – инициализирует среду MPI и должна быть первой функцией, вызываемой в любом MPI-приложении.
* MPI\_Comm\_size() – возвращает общее количество процессов в данном коммуникаторе (группе созданных MPI\_Init процессов), который чаще всего является глобальным коммуникатором MPI\_COMM\_WORLD.
* MPI\_Comm\_rank() – возвращает уникальный идентификатор процесса из заданного коммуникатора.
* MPI\_Reduce() – собирает данные от всех процессов и применяет к ним указанную операцию (максимум в данной программе), после чего результат передается указанному процессу.
* MPI\_Wtime() – возвращает текущее время в секундах, используется для измерения времени выполнения программы.
* MPI\_Finalize() – завершает работу MPI и освобождает ресурсы, связанные с MPI. Должна быть последней вызванной функцией в MPI-программе.
* MPI\_Gather() – используется для сбора данных в массив на заданном процессе.
* MPI\_Gatherv() - используется для сбора данных с различными размерами в массив на заданном процессе.

*Графики выполнения программы при различном количестве потоков:*

*Поверхности (X – MPI, Y – OMP, Z – Эффективность):*







*Заключение:*

В ходе данной лабораторной работы был изучен процесс совместного использования технологий MPI и OpenMP для организации параллельных вычислений на примере функции поиска простых чисел в заданном диапазоне. Программа была запущена при разном количестве потоков, что позволило оценить влияние распараллеливания на производительность.

Ускорение программы с помощью OpenMP ожидаемо показало хороший результат, практически двукратное ускорение при 2 потоках OpenMP. Ускорение с помощью увеличения потоков MPI показало меньший прирост производительности, самый большой в пересчете на количество ядер – на 2 потоках, до 1.8 ускорения. После 5 потоков производительность уже почти не растет.

Нагрузка на ядра процессора распределена равномерно. Нулевой процессор почти всегда оказывался самым медленным, что объясняется его нагрузкой системой и иными процессами.

При учете времени, затрачиваемого на пересылку данных, разница между отдельными потоками становится еще меньше, т.к. они дожидаются готовности корневого потока.

Был также реализован более производительный, однако плохо поддающийся распараллеливанию алгоритм с сегментированным решетом Эратосфена (Приложение 2). Его результаты с 1 потоком MPI и 2 потоками OpenMP были примерно в 10 раз лучше, чем у базового варианта (3.12 с / 31.52 с). Также при уменьшении диапазона в 10 раз, до элементов, он показал результат 0.30 с против 1.4 с. Таким образом, при увеличении диапазона он показывает более приятные результаты (увеличение времени составило 10.4 раза против 22.5 раз у базового варианта при увеличении диапазона от до ). Его временная сложность составляет ; у базового варианта: .

Замедление роста производительности при увеличении числа потоков также может быть вызвано сторонними процессами системы и влиянием планировщика ОС на выделяемые программе ресурсы.

*Приложение:*

1. Исходный код программы с измерением времени работы программы:

#include <stdlib.h>  
#include <stdio.h>  
#include <mpi.h>  
#include <omp.h>  
#include <math.h>  
#include <stdbool.h>  
  
#define RUNS\_NUM 1  
#define NUMBER (unsigned int) 3e8  
  
void save\_many\_res(const double \*times, const int size){  
 FILE \*file = fopen("C:\\C++\\5sem\\PPD\\lab7\\res.txt", "a");  
  
 if (!file){  
 perror("Error opening file");  
 }  
 for (int i = 0; i < size; i++){  
 fprintf(file, "%lf ", times[i]);  
 }  
 fprintf(file, "\n");  
  
 fclose(file);  
}  
  
int \*findPrimesInRangePre(unsigned int S, unsigned int N, int \*primeCount){  
 if (N < 2 || S > N){  
 \*primeCount = 0;  
 return NULL;  
 }  
  
 bool \*isPrime = (bool \*) malloc((N - S + 1) \* sizeof(bool));  
  
 isPrime[0] = isPrime[1] = false;  
  
#pragma omp parallel for num\_threads(6) default(shared)  
 for (unsigned int i = 2; i <= N; ++i){  
 isPrime[i] = true;  
 }  
  
 unsigned int sqrtN = sqrt(N);  
  
 for (unsigned int i = 2; i <= sqrtN; ++i){  
 if (isPrime[i]){  
 for (unsigned int j = i \* i; j <= N; j += i){  
 isPrime[j] = false;  
 }  
 }  
 }  
  
 unsigned int \*primes = (unsigned int \*) malloc((N / log(N) \* 1.5) \* sizeof(unsigned int));  
 \*primeCount = 0;  
  
 for (unsigned int i = (S < 2 ? 2 : S); i <= N; ++i){  
 if (isPrime[i]){  
 primes[(\*primeCount)++] = i;  
 }  
 }  
 primes = realloc(primes, \*primeCount \* sizeof(unsigned int));  
  
 free(isPrime);  
  
 return primes;  
}  
  
int \*findPrimesInRange(unsigned int S, unsigned int N, const int \*divisors, int divisorCount, int \*primeCount, double \*time){  
 if (N < 2 || S > N){  
 \*primeCount = 0;  
 return NULL;  
 }  
  
 unsigned int \*primes = (unsigned int \*) malloc((N / log(N) \* 2) \* sizeof(unsigned int));  
 \*primeCount = 0;  
  
 double s1 = MPI\_Wtime();  
#pragma omp parallel for num\_threads(2) default(shared)  
 for (unsigned int n = S; n <= N; n++){  
 bool isPrime = true;  
 for (unsigned int i = 0; i < divisorCount; i++){  
 if (!(n % divisors[i])){  
 isPrime = false;  
 break;  
 }  
 }  
 if (isPrime){  
 primes[(\*primeCount)++] = n;  
 }  
 }  
 double e1 = MPI\_Wtime();  
  
 (\*time) += e1 - s1;  
  
 primes = realloc(primes, \*primeCount \* sizeof(unsigned int));  
  
 return primes;  
}  
  
double combineResults(const unsigned int S, const unsigned int N, const int \*sharedPrimes, int sharedPrimesCount, const int rank,  
 const int proc\_num){  
 double time = 0;  
 int localPrimesCount = 0;  
 int \*localPrimes = findPrimesInRange(S, N, sharedPrimes, sharedPrimesCount, &localPrimesCount, &time);  
  
 int totalCount;  
 int \*array = NULL;  
 int \*recvcounts = NULL;  
 int \*displs = NULL;  
  
 double s1 = MPI\_Wtime();  
 if (!rank){  
 recvcounts = (int \*) malloc(proc\_num \* sizeof(int));  
 displs = (int \*) malloc(proc\_num \* sizeof(int));  
 }  
  
 MPI\_Gather(&localPrimesCount, 1, MPI\_INT, recvcounts, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  
  
 if (!rank){  
 totalCount = 0;  
 for (int i = 0; i < proc\_num; ++i){  
 displs[i] = totalCount;  
 totalCount += recvcounts[i];  
 }  
  
 array = (int \*) malloc(totalCount \* sizeof(int));  
 }  
  
 MPI\_Gatherv(localPrimes, localPrimesCount, MPI\_INT, array, recvcounts, displs, MPI\_INT, 0, MPI\_COMM\_WORLD);  
 double e1 = MPI\_Wtime();  
  
 if (!rank){  
 free(recvcounts);  
 free(displs);  
 free(array);  
 }  
  
// time += e1 - s1;  
  
 free(localPrimes);  
 return time;  
}  
  
void time\_algorithm(){  
 double timeSpent = 0;  
 double maxTimeSpent = 0;  
 int proc\_num = -1;  
 int rank = -1;  
  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);  
  
 int sharedPrimesCount = 0;  
 int \*sharedPrimes = NULL;  
 unsigned int sqrtN = sqrt(NUMBER);  
 unsigned int workingRange = NUMBER - sqrtN;  
  
 if (!rank){  
 sharedPrimes = findPrimesInRangePre(0, sqrtN, &sharedPrimesCount);  
 }  
  
 double s1 = MPI\_Wtime();  
 MPI\_Bcast(&sharedPrimesCount, 1, MPI\_INTEGER, 0, MPI\_COMM\_WORLD);  
 double e1 = MPI\_Wtime();  
  
 if (rank){  
 sharedPrimes = malloc(sharedPrimesCount \* sizeof(int));  
 }  
 double s2 = MPI\_Wtime();  
 MPI\_Bcast(sharedPrimes, sharedPrimesCount, MPI\_INTEGER, 0, MPI\_COMM\_WORLD);  
 double e2 = MPI\_Wtime();  
  
// timeSpent += e2 - s2 + e1 - s1;  
  
 unsigned int S = sqrtN + workingRange / proc\_num \* rank;  
 unsigned int N = sqrtN + workingRange / proc\_num \* (rank + 1);  
  
 for (int i = 0; i < RUNS\_NUM; i++){  
 timeSpent += combineResults(S, N, sharedPrimes, sharedPrimesCount, rank, proc\_num);  
 }  
  
 double \*timeArray = NULL;  
  
 if (!rank){  
 timeArray = malloc(proc\_num \* sizeof(double));  
 }  
  
 MPI\_Gather(&timeSpent, 1, MPI\_DOUBLE, timeArray, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  
 MPI\_Reduce(&timeSpent, &maxTimeSpent, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);  
  
 if (!rank){  
 printf("%lf\n\n", maxTimeSpent / RUNS\_NUM);  
  
 for (int i = 0; i < proc\_num; i++){  
 printf("%lf ", timeArray[i] / RUNS\_NUM);  
 }  
 save\_many\_res(timeArray, proc\_num);  
 free(timeArray);  
 }  
 free(sharedPrimes);  
}  
  
int main(int argc, char \*\*argv){  
 int ret = -1; ///< For return values  
 int rank = -1; ///< This processor's number  
  
 ret = MPI\_Init(&argc, &argv);  
 if (!rank){ printf("MPI Init returned (%d)\n", ret); }  
  
 time\_algorithm();  
  
 ret = MPI\_Finalize();  
  
 return 0;  
}

1. Исходный код алгоритма сегментированного решета Эратосфена

int \*findPrimesInRange(unsigned int S, unsigned int N, const int \*divisors, int divisorCount, int \*primeCount, double \*time){  
 if (N < 2 || S > N){  
 \*primeCount = 0;  
 return NULL;  
 }  
  
 bool \*isPrime = (bool \*) malloc((N - S + 1) \* sizeof(bool));  
  
 double s1 = omp\_get\_wtime();  
 isPrime[0] = isPrime[1] = false;  
  
#pragma omp parallel for num\_threads(2) default(shared)  
 for (unsigned int i = 2; i < N - S + 1; ++i){  
 isPrime[i] = true;  
 }  
  
 for (unsigned int i = 0; i < divisorCount; ++i){  
 unsigned int p = divisors[i];  
 unsigned int start = fmax(p \* p, ((S + p - 1) / p) \* p);  
#pragma omp parallel for num\_threads(2) default(shared)  
 for (unsigned int j = start - S; j <= N - S; j += p){  
 isPrime[j] = false;  
 }  
 }  
 double e1 = omp\_get\_wtime();  
  
 unsigned int \*primes = (unsigned int \*) malloc((N / log(N) \* 1.5) \* sizeof(unsigned int));  
 \*primeCount = 0;  
  
 double s2 = MPI\_Wtime();  
 for (unsigned int i = (S < 2 ? 2 : S); i <= N; ++i){  
 if (isPrime[i - S]){  
 primes[(\*primeCount)++] = i;  
 }  
 }  
 double e2 = MPI\_Wtime();  
  
 (\*time) = e2 - s2 + e1 - s1;  
  
 primes = realloc(primes, \*primeCount \* sizeof(unsigned int));  
 free(isPrime);  
  
 return primes;  
}

1. Таблицы времени работы программы

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| С учетом времени на пересылку данных | | | | | | |
| **Потоки MPI (OMP = 1)** | **Поток 0** | **Поток 1** | **Поток 2** | **Поток 3** | **Поток 4** | **Поток 5** |
| 1 | 59,776266 |  |  |  |  |  |
| 2 | 31,586804 | 31,586934 |  |  |  |  |
| 3 | 22,615581 | 22,610625 | 22,615147 |  |  |  |
| 4 | 19,139364 | 19,13197 | 19,135746 | 19,139006 |  |  |
| 5 | 17,010842 | 17,002156 | 17,005044 | 17,008542 | 17,010784 |  |
| 6 | 15,280921 | 15,270885 | 15,273348 | 15,275892 | 15,278224 | 15,280712 |
|  |  |  |  |  |  |  |
| **Потоки MPI (OMP = 2)** | **Поток 0** | **Поток 1** | **Поток 2** | **Поток 3** | **Поток 4** | **Поток 5** |
| 1 | 31,469211 |  |  |  |  |  |
| 2 | 18,207126 | 18,207184 |  |  |  |  |
| 3 | 14,994969 | 14,989841 | 14,994861 |  |  |  |
| 4 | 13,833756 | 13,82623 | 13,829807 | 13,833672 |  |  |
| 5 | 13,593872 | 13,581465 | 13,585519 | 13,589599 | 13,593733 |  |
| 6 | 13,147482 | 13,13474 | 13,137991 | 13,141408 | 13,144111 | 13,147441 |
|  |  |  |  |  |  |  |
| Без учета времени на пересылку данных | | | | | | |
| **Потоки MPI (OMP = 1)** | **Поток 0** | **Поток 1** | **Поток 2** | **Поток 3** | **Поток 4** | **Поток 5** |
| 1 | 60,595774 |  |  |  |  |  |
| 2 | 31,872068 | 32,048682 |  |  |  |  |
| 3 | 22,592003 | 22,409679 | 22,646509 |  |  |  |
| 4 | 19,221724 | 18,822383 | 19,10157 | 19,22338 |  |  |
| 5 | 16,606664 | 16,412357 | 16,231513 | 16,312475 | 16,287893 |  |
| 6 | 15,256937 | 15,132593 | 14,876964 | 15,142383 | 14,952649 | 15,19314 |
|  |  |  |  |  |  |  |
| **Потоки MPI (OMP = 2)** | **Поток 0** | **Поток 1** | **Поток 2** | **Поток 3** | **Поток 4** | **Поток 5** |
| 1 | 31,549174 |  |  |  |  |  |
| 2 | 18,161188 | 17,831057 |  |  |  |  |
| 3 | 14,639066 | 14,653091 | 14,640661 |  |  |  |
| 4 | 13,741249 | 13,489363 | 13,508407 | 13,578518 |  |  |
| 5 | 13,118208 | 12,915696 | 12,741888 | 12,709362 | 12,896554 |  |
| 6 | 12,910406 | 12,671457 | 12,513654 | 12,635316 | 12,619663 | 12,675729 |

1. Таблицы ускорения программы

OMP = 1

|  |  |  |
| --- | --- | --- |
| Ускорение программы | |  |
| **Количество потоков** | **С учетом пересылки данных** | **Без учета пересылки данных** |
| 1 | 1 | 1 |
| 2 | 1,892444262 | 1,901218773 |
| 3 | 2,643145272 | 2,682178026 |
| 4 | 3,123210677 | 3,152463015 |
| 5 | 3,51400983 | 3,648883003 |
| 6 | 3,911823509 | 3,971686715 |

OMP = 2

|  |  |  |
| --- | --- | --- |
| **Количество потоков** | **С учетом пересылки данных** | **Без учета пересылки данных** |
| 1 | 1 | 1 |
| 2 | 1,728400792 | 1,737175674 |
| 3 | 2,098651288 | 2,155135717 |
| 4 | 2,274813218 | 2,295946606 |
| 5 | 2,314955665 | 2,404991139 |
| 6 | 2,393554218 | 2,443701151 |

1. Таблицы эффективности программы

OMP = 1

|  |  |  |
| --- | --- | --- |
| Эффективность программы | |  |
| **Количество потоков** | **С учетом пересылки данных** | **Без учета пересылки данных** |
| 1 | 1 | 1 |
| 2 | 0,946222131 | 0,950609386 |
| 3 | 0,881048424 | 0,894059342 |
| 4 | 0,780802669 | 0,788115754 |
| 5 | 0,702801966 | 0,729776601 |
| 6 | 0,651970585 | 0,661947786 |

OMP = 2

|  |  |  |
| --- | --- | --- |
| Эффективность программы | |  |
| **Количество потоков** | **С учетом пересылки данных** | **Без учета пересылки данных** |
| 1 | 1 | 1 |
| 2 | 0,864200396 | 0,868587837 |
| 3 | 0,699550429 | 0,718378572 |
| 4 | 0,568703304 | 0,573986651 |
| 5 | 0,462991133 | 0,480998228 |
| 6 | 0,398925703 | 0,407283525 |