Отчет по домашнему заданию 2

Капралов Степан Алексеевич

Задача 1. Метод k-средних (k-means)

Для начала нужно определить область программы которую можно эффективно распараллелить. Функция KMeans выполняет основную работу программы, в ней присутствуют несколько циклов, 1 цикл снаружи цикла while и 3 внутри. разберем каждый из них.

1. for (size_t i = 0; i < K; ++i) - выполняется вне while тоесть один раз за всю программу. Содержимое этого цикла выполняется за константное время. K - число кластеров, ожидается, что K значительно меньше N. Распараллеливание этого цикла не даст хорошего прироста производительности, оставлю его как есть.
2. while (!converged) - внутри этого цикла происходит основная работа программы, однако его итерации должны выполняться последовательно. Распараллелить его не получится.
3. for (size_t i = 0; i < data_size; ++i) - выполняется N раз, а каждая итерация выполняется за K, и того асимтотическая оценка вычислительной сложности этого цикла - O(N*K), этот цикл можно и нужно распараллеливать.
4. for (size_t i = 0; i < data_size; ++i)
   for (size_t d = 0; d < dimensions; ++d) - Вложеный цикл O(N*d). Выполняется значительно быстрее 3го цикла, так как ожидается, что d в большинстве случаев значительно меньше K. Однако N самая большая величина в программе, поэтому, распараллеливание должно дать положительный результат.
5. for (size_t i = 0; i < K; ++i) - O(K*d). На фоне 1го цикла выполняется очень быстро. Вряд ли есть смысл его распараллеливать.

Реализация:

Два цикла которые я буду распараллеливать находятся внутри while, поэтому создам параллельную область снаружи while, что бы избежать дорогостоящей операции создания потоков на каждой итерации цикла. Присваивание значения переменной converged должно производиться в одном процессе, помещу его в #pragma omp single. Следующий цикл помещаем в #pragma omp for reduction (+:converged) schedule(auto). Изменяем тип converged на int и применяем к нему reduction, что бы не использовать блокировок внутри цикла. В ходе тестов наилучшее распределение итераций показали schedule(static, 2) и schedule(auto), программа распределенная с auto работает быстрее чем со static. Поэтому оставим компилятору право выбирать распределение. Выносим инициализацию vector<size_t> clusters_sizes(K) за границу цикла while, а вместо нее подставляем clusters_sizes.assign(K, 0). В данном месте программы распределим

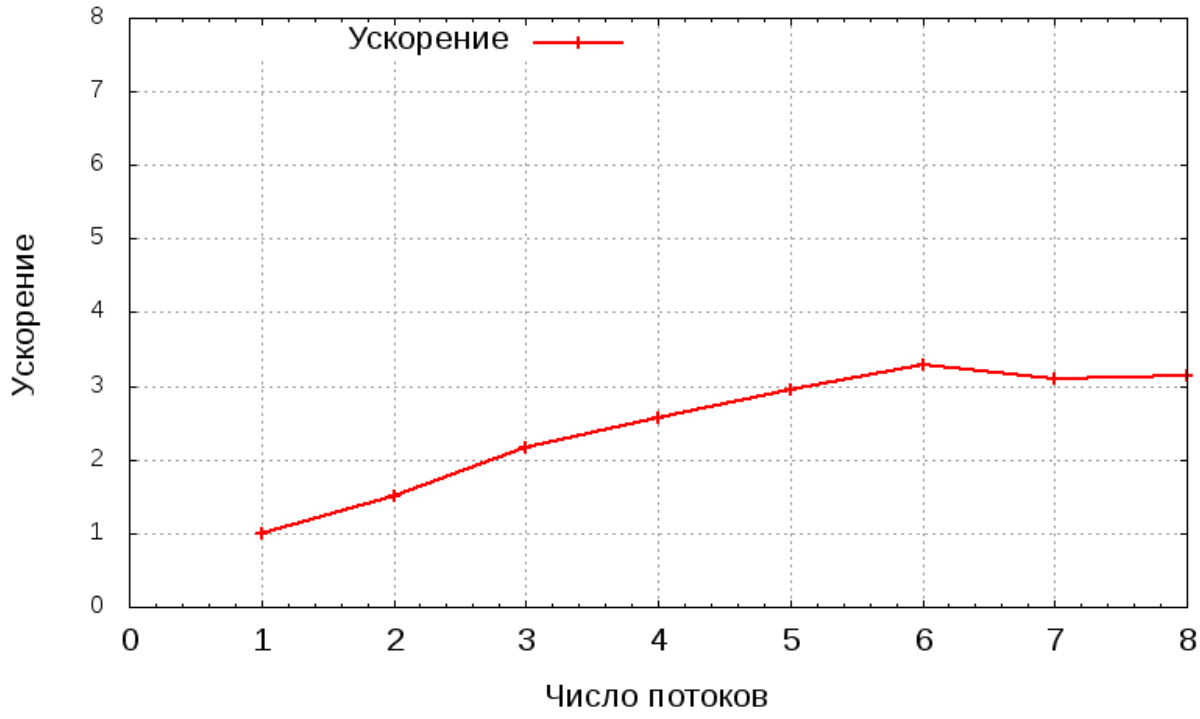
ассоциацию двух векторов по параллельным секция с помощью #pragma omp sections. Следующий цикл модифицируем так, чтобы все операции выполнялись во вложенном цикле, и применяем к нему #pragma omp for collapse(2) schedule(auto). Последний цикл я решил не распараллеливать, но так как его нельзя вынуть из параллельной области, нужно обеспечить ему правильную работу, применю для этого #pragma omp for nowait schedule(auto). Так как потоки нужно синхронизировать вначале цикла while, поставлю туда #pragma omp barrier, а к последнему циклу применю nowait.

Тестирование:

Тестирование производилось на кластере Jet. Для замеров времени использовалась функция omp_get_wtime(), замеры производились с учетом ввода и вывода данных.

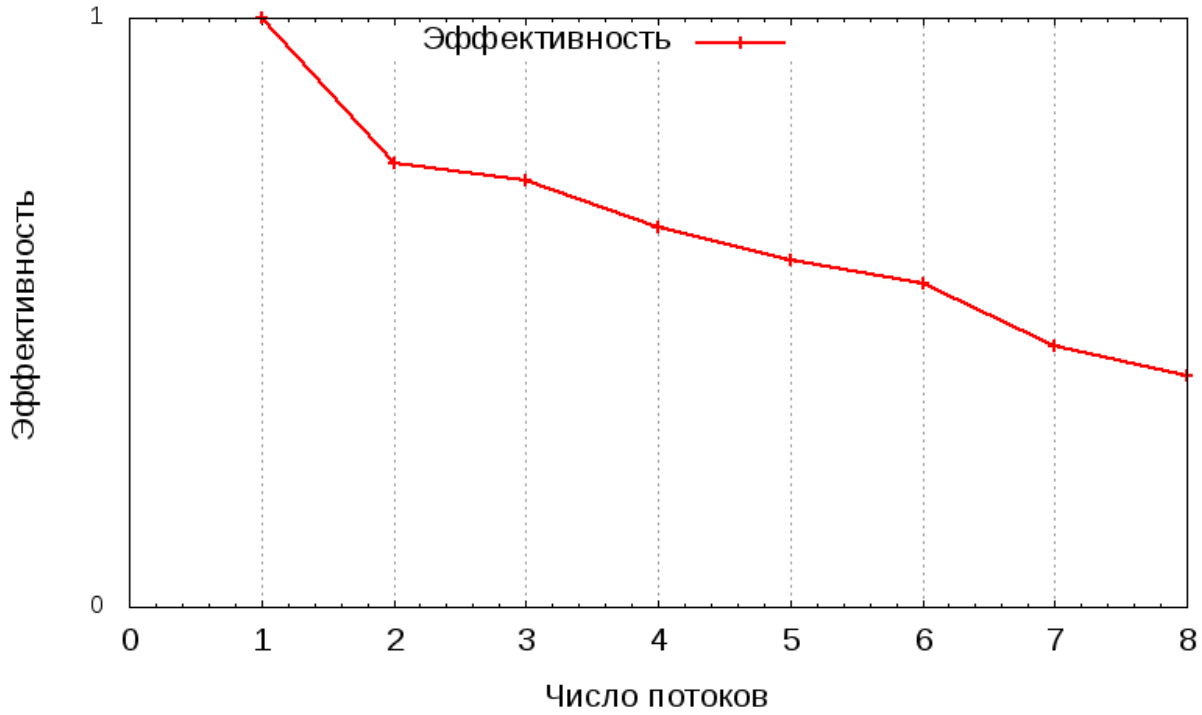
Ускорение:

1 000 000 точек 10 кластеров



Ускорение растет линейно, присутствует масштабируемость. Максимальное ускорение достигается шестью потоками и равно 3.3раз.

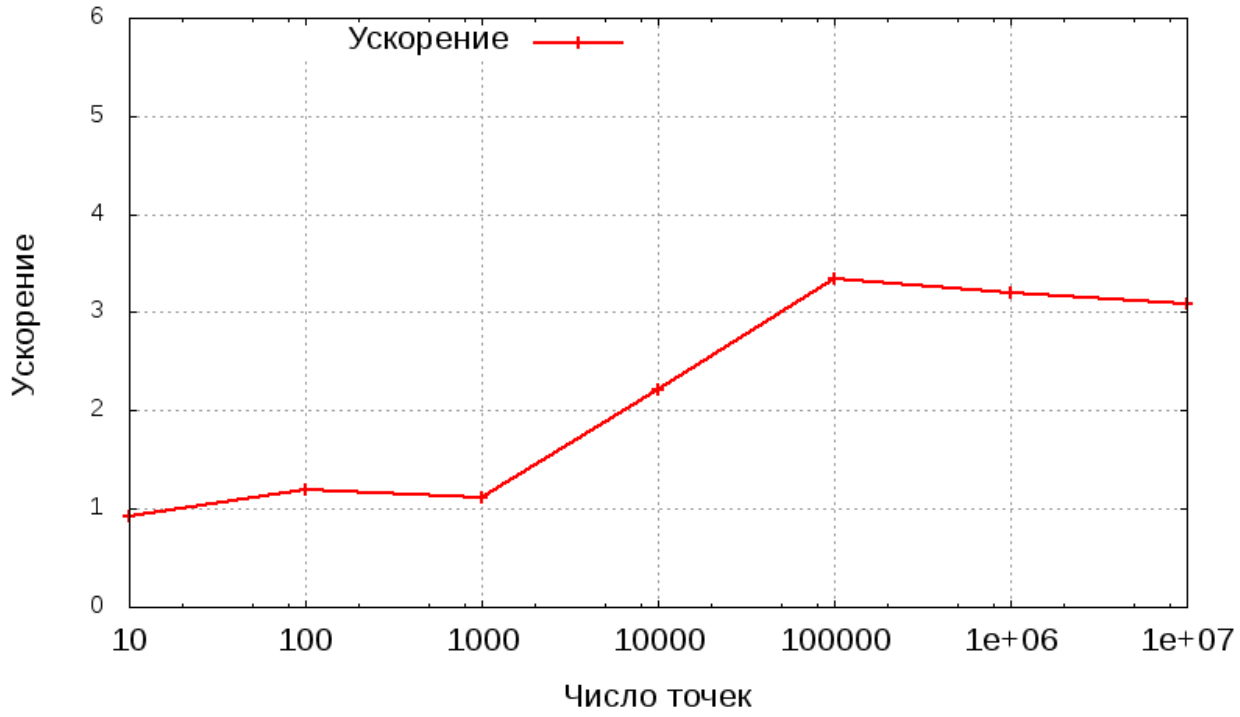
Эффективность:



Эффективность падает из за увеличения накладных расходов при росте числа процессов.

Ускорение от числа точек:

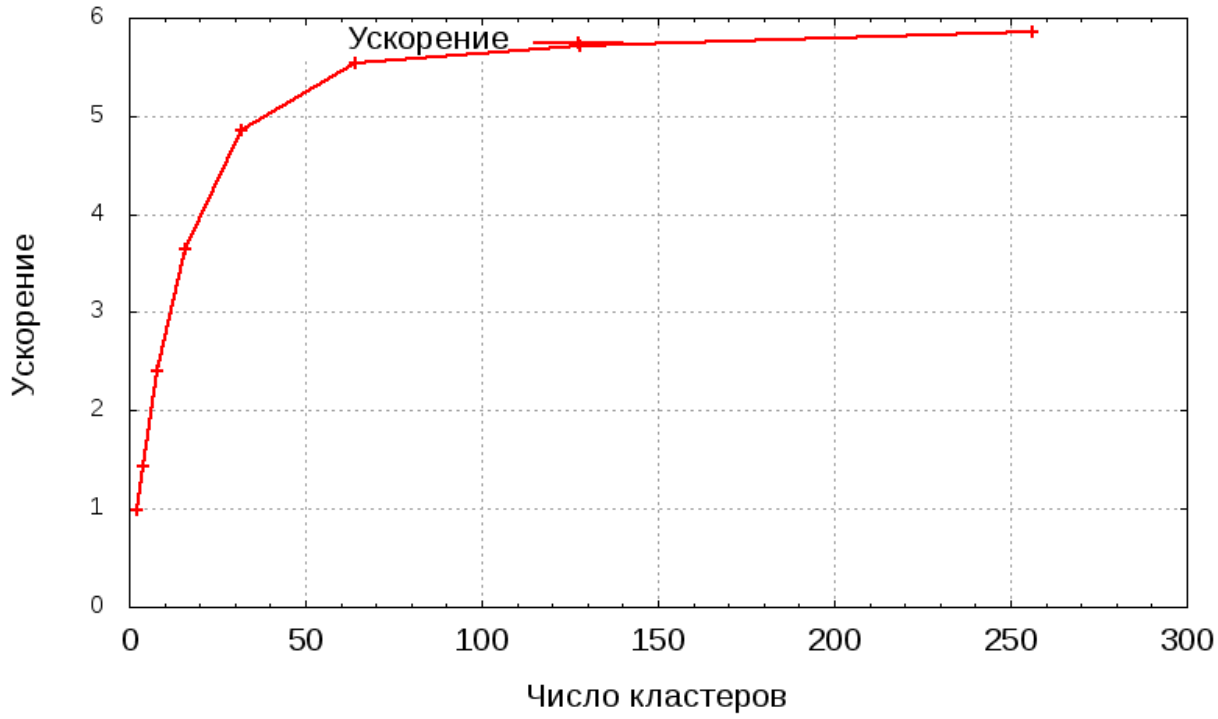
10 кластеров 6 потоков



При увеличении числа точек наблюдается рост ускорения, так как сглаживается влияние накладных расходов.

Ускорение от числа кластеров:

100 000 точек 6 потоков



Задача 2. Игра «Жизнь» (Conway's Game of Life)

Идея:

Возьму для последующей работы вторую реализацию игры жизнь, с дополнительным массивом front размером N*N, служащим для определения изменений на предыдущем шаге. Для начала нужно определиться с методом распределения массива по процессам. Выбираю распределение по полосам N*M, так как оно позволяет использовать меньше пересылок сообщений между процессами, чем распределение на блоки M*M. Имея квадратное поле игры, можно не заботиться о выборе оптимальной ширины полосы. Поэтому распределение всегда будет происходить по горизонтальным полосам, так удобнее делить память на буферы. Ширина полосы рассчитывается как M=N/p (p - число процессов), если N%p<>0 то оставшийся хвост распределяем равномерно, по одной строке на каждую полосу. Так же для корректной работы алгоритма полосам нужно видеть крайние строки своих соседей. Буду передавать их с помощью сообщений непосредственно внутри итераций главного цикла. Каждый процесс проводит изменения только внутри своей полосы не изменяя общих границ. Массив front в начальном распределении не нуждается так как всегда будет заполнен единицами, инициализирую его локально в каждом процессе. Осталось решить еще одну проблему. При изменении ячейки в крайней строке полосы, изменение массива front придется производить и в общих строках. Эти изменения должны быть видны в

соседней области. Первый способ решения - организовать пересылку общих частей массива front и совместить эти части в полосе владеющей общей строкой. Второй способ - при выполнении итерации просматривать все ячейки крайних строк, независимо от массива front. Выберу второй способ так как он не требует дополнительного обмена сообщениями.

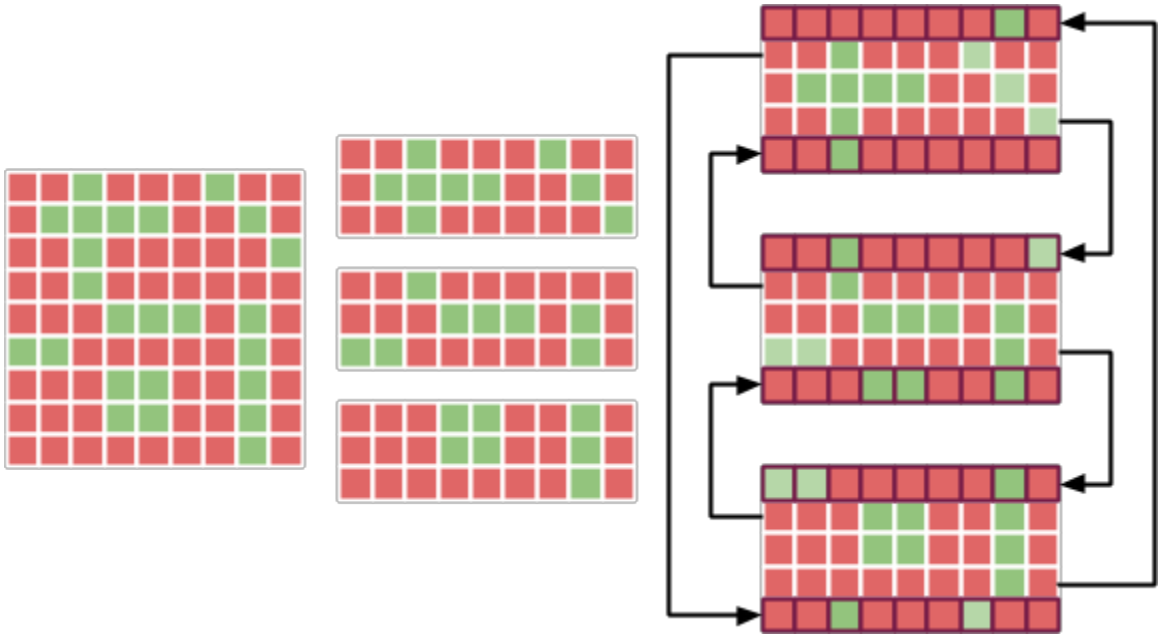


Схема распределения элементов поля 9x9 на 3 процесса.

Реализация:

После инициализации MPI области (MPI_Init), главный процесс выделяет память под общий буфер и считывает его значения из файла. Для предварительной рассылки полос по процессам использую функцию MPI_Scatterv, так как она позволяет рассылать буферы разного размера. Для рассылки граничных строк внутри итерации применил функции без ожидания передачи сообщения MPI_Isend и MPI_Irecv. Исходя из этого, внутри итераций применил следующую схему работы: процесс отправляет и принимает пограничные строки, не дожидаясь окончания приема и отправки сообщений обрабатывает внутренние строки, для которых не нужно знать значения границ, потом происходит ожидание завершения приема и передачи при помощи функции MPI_Waitall и последующая обработка крайних строк. После завершения итераций все полосы собираются в буфер находящийся в главном процессе с помощью функции MPI_Gatherv. Главный процесс выводит буфер в файл.

Тестирование:

Тестирование производилось на кластере Jet. Для замеров времени использовалась утилита time с параметром -p, откуда бралось реальное время от запуска программы до ее завершения, так как требуется замерять время выполнения программы целиком.

Ускорение:

Поле 5000*5000, 500 итераций

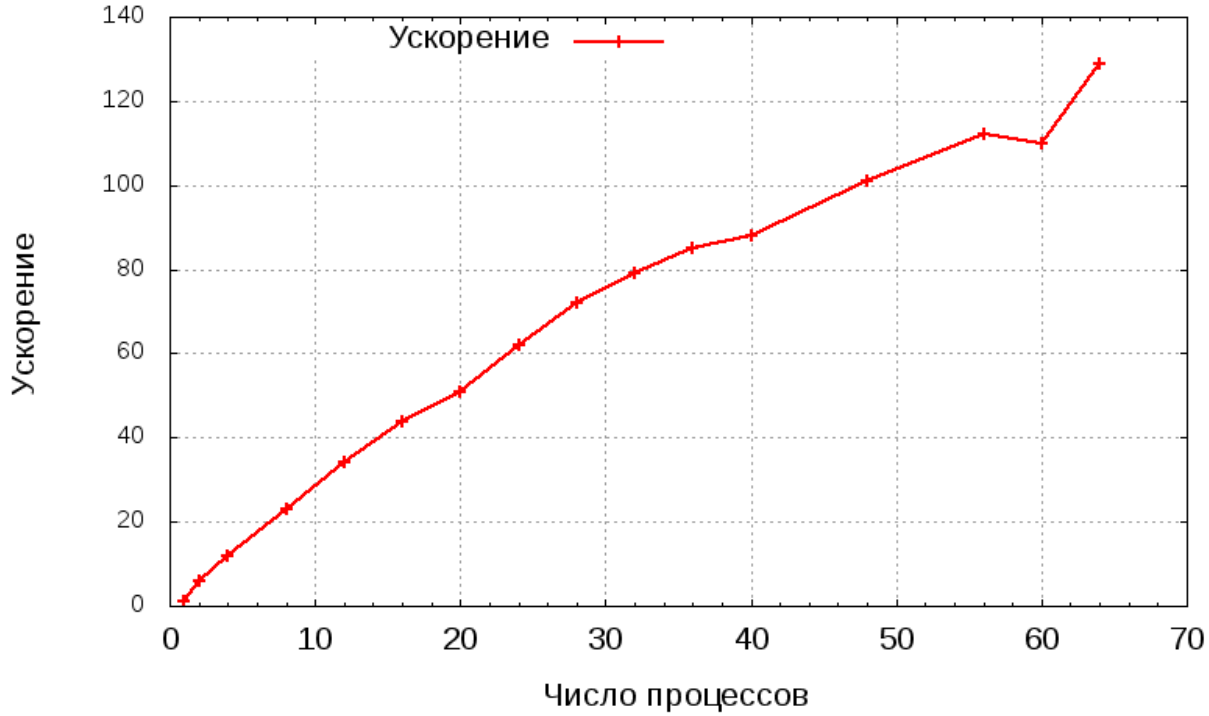
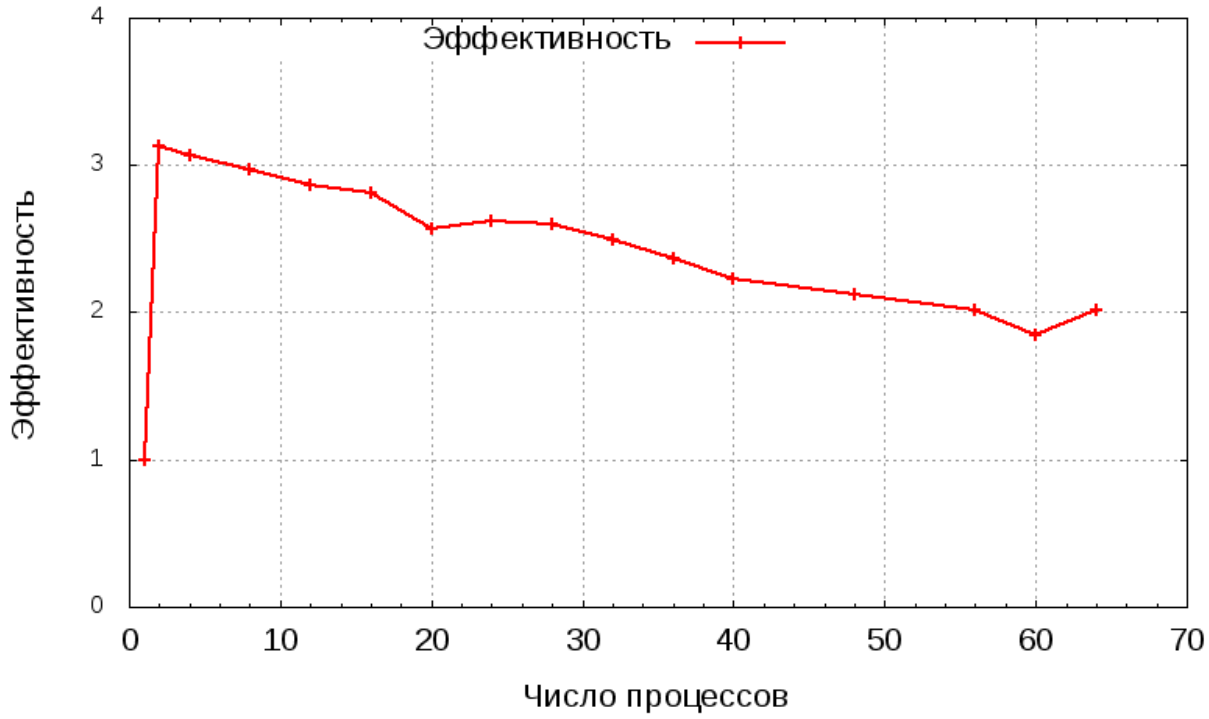


График ускорения получился весьма неожиданным. Ускорение линейно, присутствует хорошая масштабируемость, но располагается оно выше чем ожидаемое ускорение(ускорение равное числу процессов). Это может быть обусловленно тем, что поле равномерно распределено по процессам, и у каждого из них есть своя область оперативной памяти, свой процессорный кэш, что позволяет более эффективно обращаться к памяти.

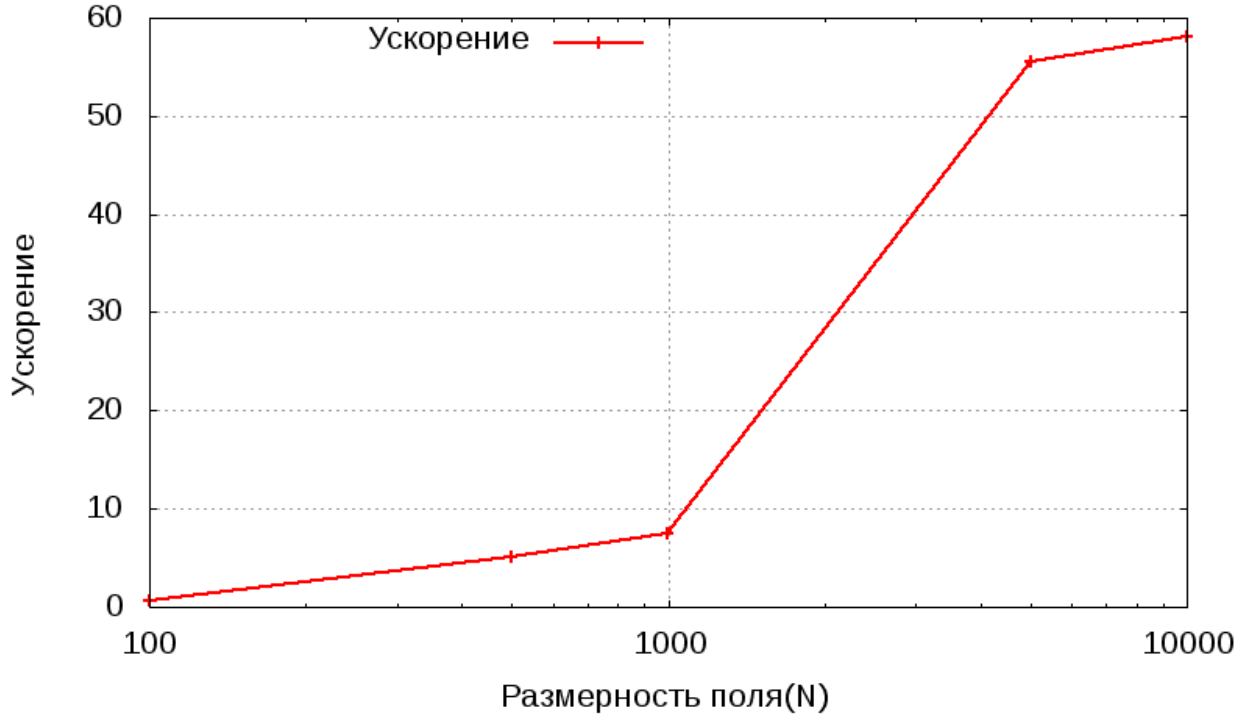
Эффективность:



Эффективность снижается с увеличением числа процессов, так как возрастает число пересылок сообщений.

Ускорение от размера поля:

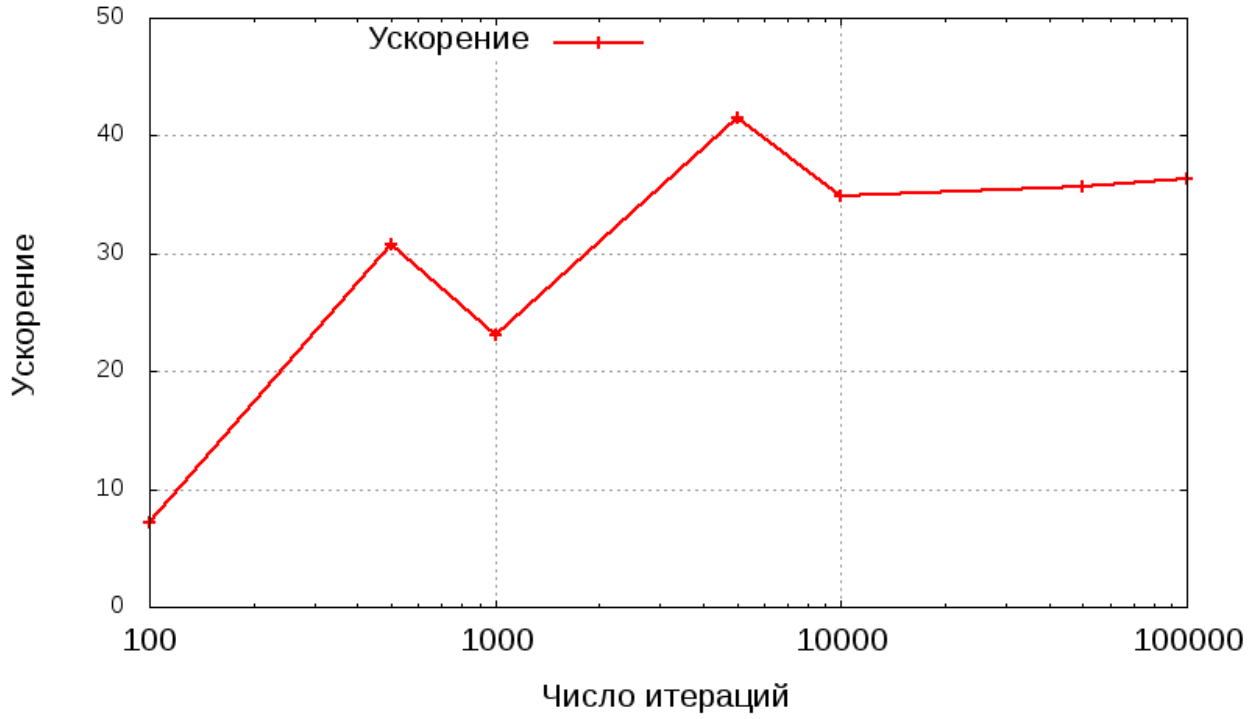
32 процесса, 100 итераций
Размерность поля(100, 500, 1000, 5000, 10000)



Ускорение увеличивается с увеличением размеров поля, так как разбиение происходит на все более крупные части, и обработка их середин компенсирует врем затраченное на передачу крайних строк между процессами.

Ускорение от числа итераций:

Поле 1000*1000, 32 процесса



С увеличение числа итераций растет и ускорение, так как сглаживается влияние накладных расходов затраченных на параллелизацию алгоритма.