Отчет о выполнении 2 задания практикума кафедры СКИ

Р.М. Куприй, 423 группа

Факультет ВМК МГУ имени М.В. Ломоносова

1. Задание

Задание состоит в освящении проблемы масштабируемости при программировании MPI-приложений. В частности, организации упорядоченного точного порядка приема данных при пересылках. Основной причиной снижения производительности в таком случае является ожидание на приёме данных. В качестве примера рассматривается наивная реализация алгоритма параллельной сортировки слиянием. Представлено два варианта расспалалеливания, отличающиеся алгоритмом слияния.

2. Реализация оптимизации

В первом варианте программы 1 в рассматриваемой проблемной области слияние происходит в порядке возрастания номеров процессов. Итоговый отсортированный вектор сортируется и накапливается на процессе с рангом 0 из отсортированных кусочков, получаемых от остальных процессов. В результате работы такого алгоритма слияние происходит фактически последовательно, поскольку всю работу на финальном шаге алгоритмов выполняет нулевой процесс.

Во втором варианте программы 2 слияние происходит параллельно. Исходный вектор, изначально распределенный между всеми процессами поровну, постепенно собирается на выбранных процессах путём слияния кусочков между собой. На каждой итерации выбранный процесс получает от другого процесса часть вектора и организовывает слияние. Процесс, который отправил свой отсортированный вектор другому процессу больше не участвует в сортировке. Таким образом, часть слияния исходного вектора происходит параллельно.

```
for (int i = 1; i < world_size; i++) {
  if (rank == i) {
    MPI_Send(local_x_ptr, local_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  } else if (rank == 0) {
    MPI_Recv(local_x_ptr, local_size, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
   &Stat);
    merge(tmp, x, local_x, current_size);
    current_size += local_size;
  }
}
```

Рис. 1. Базовый вариант программы

```
for (int i = world_size / 2; i >= 1; i /= 2) {
  if (rank < i) {
    MPI_Recv(local_x_ptr, local_size, MPI_DOUBLE, rank + i, 0,
   MPI_COMM_WORLD, &Stat);
    merge(tmp, x, local_x, local_size);
    local_size *= 2;
  } else if (rank < 2 * i) {
    MPI_Send(x_ptr, local_size, MPI_DOUBLE, rank - i, 0, MPI_COMM_WORLD)
   ;
  }
}
```

Рис. 2. Оптимизированный вариант программы

3. Методика тестирования

Для оценки эффективности оптимизированного варианта программы сравнивается время сортировки целевого вектора. Проведены эксперимены для векторов разных размеров.

Для тестирования производительности использовалась параллельная вычислительная система Polus, с 2 вычислительными узлами, в каждом из которых 2 10 ядерных процессора IBM POWER8. Для компиляции использовался компилятор `mpixlC` с соответствующими флагами опимизации: `-O5 -qarch=pwr8`.

Для исключения выбросов и получения равномерной оценки по запускам, каждая версия программы запускалась до 6 раз, среди всех запусков выбиралось минимальное время работы. Для привязки процессам к физическим ядрам использовались ключи запуска `-map-by core -bind-to core`. Далее представлены результаты сортировки векторов размером 64000, 160000 и 480000.

4. Оценки эффективности OpenMP программ

Для рассматриваемой проблемы ожидается нарастание проблемы с масштабируемостью, поскольку при росте числа нитей увеличиваются накладные расходы на их создание и уничтожение. Это подтверждается проведенными экспериментами и наглядно видно в таблицах 3, **??**. В таблицах приведено общее время выполнения всех итераций, для базового и для оптимизированного варианта программы соответственно, а также значение ускорения времени расчётов для улучшенной версии программы.

5. Анализ полученных результатов

По результатам видно, что разичия между подходами проявляются только при максимальном числе процессов. Кроме того, чем больше размер вектора, тем менее заметен эффект от оптимизации. Так, для вектора размера 64000 элементов, приведенная оптимизация позволяет ускорить расчёт на 15% для 32 процессов. В то время как для 160000 элементов достигаемое ускорение снижается до 8% и продолжает линейно снижаться далее.

Крайне важной особенностью представленной программы является то, что перед слиянием каждый процесс сортирует свою часть вектора самым наивным методом – методом пузырька. В результате этого, общее время сортировки меняется нелинейно

Таблица 1. Сравнение времени выполнения базовой (base time) и оптимизированной (optimized time) программы для вектора размером 64000

nprocs	base time, sec	optimized time, sec	speedup
1	7.715	7.714	0%
2	1.84	1.836	0.2%
4	0.423	0.425	-0.5%
8	0.97	0.97	-0.1%
16	0.023	0.023	-0.1%
32	0.007	0.006	15.3%

Таблица 2. Сравнение времени выполнения базовой (base time) и оптимизированной (optimized time) программы для вектора размером 160000

nprocs	base time, sec	optimized time, sec	speedup
1	49.8	49.8	0%
2	12.2	12.2	0%
4	2.96	2.96	-0.1%
8	0.7	0.7	0%
16	0.164	0.163	0.3%
32	0.04	0.037	8%

Таблица 3. Сравнение времени выполнения базовой (base time) и оптимизированной (optimized time) программы для вектора размером 480000

nprocs	base time, sec	optimized time, sec	speedup
1	449	449	0%
2	113.3	113.4	-0.01%
4	30.5	30.4	0.2%
8	7.74	7.738	0%
16	1.725	1.725	0%
32	0.4	0.39	1.8%

при разбиении вектора по процессам, поскольку используемая сортировка обладает квадратичной временной сложностью. Эта особенность предположительно влияет и на показываемое ускорение таким образом, что время затраченное на сортировку отдельный частей вектора непропорционально больше времени, затраченного на слияние. Тем не менее, при использовании достаточно большого числа процессов, времена достаточно выравниваются, чтобы был заметен эффект от оптимизации.

Применение данной оптимизации показывает сразу два интересных и практически важных момента. Упорядоченные обмены данных между процессами могут приводить к потере производительности. Особенно, когда ппроцессы обладают разным объемом данных. Доработка алгоритмов таким образом, чтобы задействовать процессы, которые простаивают в ожидании обменов, позволяет значительно ускорить выполнение программы в ряде случаев. Второй особенностью является то, что при оптимизации определенных участков, получаемый эффект может быть не виден из-за других участков программы, время выполнения которых нелинейно зависит от числа используемых процессов. Это может привести к ошибочным выводам при проверке имплементированных оптимизаций, поэтому в первую очередь необходимо найти и/или устранить такие проблемные места.