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 два 10 ядерных процессора IBM POWER8. Для компиляции использовался компилятор `mpixlC` с соответствующими флагами опимизации: `-O5 -unroll-loops -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 $1.6e+8$, $8e+8$ и $1.6e+7$.

4. Оценки эффективности OpenMP программ

Для рассматриваемой проблемы ожидается нарастание проблемы с масштабируемостью, поскольку при росте числа нитей увеличиваются не только накладные расходы на их создание и уничтожение, но и требования к памяти (на выделение буферов у каждого процесса). Это подтверждается проведенными экспериментами и наглядно видно в таблицах 1, 2 и 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 различия между подходами проявляются при использовании больше 4 процессов. Кроме того, эффективность оптимизации растёт нелинейно с увеличением числа процессов. Так, для векторов разных размеров, приведенная оптимизация позволяет ускорить расчёт в среднем на 5% для 8 процессов и уже на 30% для 16 процессов. При использовании 32 процессах ускорение достигает 99%. Это происходит за счёт того, что реализованная оптимизация позволяет значительно улучшить масштабируемость.

Применение данной оптимизации показывает сразу два интересных и практи-

Таблица 1. Сравнение времени выполнения базовой (base time) и оптимизированной (optimized time) программы для вектора размером $1.6e+7$

nprocs	base time, sec	optimized time, sec	speedup
1	4.03	4.03	0%
2	2.02	2.04	-0.08%
4	1.1	1.1	0.1%
8	0.7	0.67	5.2%
16	0.59	0.46	29.8%
32	0.7	0.35	99.4%

Таблица 2. Сравнение времени выполнения базовой (base time) и оптимизированной (optimized time) программы для вектора размером $1.6e+8$

nprocs	base time, sec	optimized time, sec	speedup
1	45.97	45.95	0%
2	23.09	23.09	0%
4	12.43	12.44	0%
8	7.82	7.47	4.7%
16	6.39	5	27.7%
32	7.36	3.78	94.6%

Таблица 3. Сравнение времени выполнения базовой (base time) и оптимизированной (optimized time) программы для вектора размером $8e+8$

nprocs	base time, sec	optimized time, sec	speedup
1	247.6	247.59	0%
2	124.27	125.01	-0.6%
4	66.59	66.61	0%
8	41.6	39.61	5%
16	33.1	26.15	26.5%
32	37.33	-	-

чески важных момента. Упорядоченные обмены данных между процессами могут приводить к потере производительности. Особенно, когда процессы 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 наблюдаться не сразу, а для некоторого количества процессов. Поэтому необходимо исследовать поведение программы на всём доступном множестве процессов. Кроме того, важный нюанс использования большого числа процессов – ограничения на использование памяти, выделение буферов при большом числе процессов.