004.032.24 **Информационные технологии**

***Титов Кирилл Евгеньевич**, аспирант,*

Национальный исследовательский ядерный университет «МИФИ»

ИССЛЕДОВАНИЕ ПОВЕДЕНИЯ ФРЕЙМВОРКА OPENMP В ПРОГРАММАХ С ПАРАЛЛЕЛЬНЫМИ ВЫЧИСЛЕНИЯМИ

Аннотация: в статье рассматриваются особенности распараллеливания программ при помощи технологии OpenMP. Распараллеливание программ средствами OpenMP определяется программистом явно на этапе написания программы при помощи предусмотренного набора специальных директив и вспомогательных функций фреймворка. Статья содержит анализ методов построения параллельного вычислительного процесса средствами OpenMP, а также способы повышения их эффективности.

Ключевые слова: параллельные вычисления, OpenMP, многопоточность, SPMD, балансировка данных.

Abstract: the article discusses the features of program parallelization using the OpenMP technology. OpenMP program parallelization is determined by the programmer explicitly at the stage of writing the program by using the provided set of special directives and additional functions of the framework. The article contains an analysis of OpenMP methods for constructing a parallel computational process, as well as ways to improve their efficiency.

Keywords: parallel computing, OpenMP, multithreading, SPMD, data balancing.

Введение

Современные вычислительные машины обладают колоссальной вычислительной мощностью и позволяют решать большинство предлагаемых инженерных и научных задач. Однако существует круг задач, для которых мощности существующих вычислительных систем не хватает, к таким задачам относятся изучение и предсказание климатических условий, обнаружение космических объектов, молекулярное моделирование белков, обнаружение простых чисел и многие другие. В связи с тем, что развитие вычислительных технологий имеет принципиальные физические ограничения, перспективным способом повышения производительности вычислительных систем являются параллельные вычисления.

Одним из наиболее распространенных средств обеспечения параллельных вычислений для компьютеров с общей памятью является технология OpenMP. Данная технология представляет из себя набор директив, функций и переменных окружения, позволяющих последовательную программу превратить в параллельную. Для обеспечения параллельных вычислений OpenMP использует свойство многопоточности, при котором программа распределяется между главным (master) потоком и набором подчиненных (slave) потоков, создаваемых им [1].

OpenMP предполагает SPMD-модель параллельного программирования, в рамках которой один и тот же блок кода обрабатывается несколькими процессами. В начале программы должна находиться последовательная область, то есть изначально запускается лишь один процесс, который порождает некоторое количество дочерних нитей при входе в параллельную область программы. Число процессов, выполняющих конкретную параллельную часть программы остается неизменным до момента завершения расчетов. После того, как нити отработают параллельную часть, дочерние нити завершаются, оставляя работать родительский процесс, причем в данный момент происходит неявная синхронизация данных программы. В программе может быть

предусмотрено любое количество последовательных и параллельных блоков кода, причем параллельные области могут находиться внутри друг у друга [2].

В отличие от UNIX-процессов, порождение нитей в OpenMP не является трудозатратной операцией, поэтому частое порождение и завершение дочерних нитей не сильно влияют на производительность программы. Необходимым условием для эффективной OpenMP программы является равномерная загрузка нитей полезной работой и отсутствие простоев вычислительных ресурсов машины. Для этого фреймворк предлагает различные средства для балансировки данных.

Исследование поведения технологии

Целью работы является анализ методов построения параллельного вычислительного процесса при помощи технологии OpenMP и способов повышения их эффективности. В рамках работы были выбраны алгоритмы и их параллельные реализации, которые были запущены на процессорах различных поколений с различным числом обрабатывающих потоков. Были получены данные по времени работы каждого алгоритма и определен коэффициент «распараллеливания» каждого алгоритма для различного числа потоков процессора.

В рамках исследования были использованы алгоритмы сортировки подсчетом, быстрой сортировки, поиск подстроки в тексте, умножение матриц, метод Гаусса – Зейделя решения системы линейных уравнений и алгоритм поиска простых чисел. Алгоритмы были запущены на трех процессорах Intel различных поколений с разным набором технологий, обеспечивающих распараллеливание вычислений: 5-е поколение Broadwell (Intel Core i5-5350U), 7-е поколение Kaby Lake (Intel Core i5-7200U), 9-е поколение Coffee Lake (Intel Core i7-9750H).

Метод расчета затраченного времени

Для подсчета затраченного времени на выполнение программы использовалась функция gettimeofday(timeval*, timezone*), которая позволяет получать текущее значение времени в секундах в переданную переменную [3].

Важно отметить, что для целей работы не подходят функции, измеряющие процессорное время, такие как clock_gettime, getrusage или times, так как было важно измерить скорость работы алгоритма в зависимости от количества доступных потоков.

Сортировка подсчетом (counting sort)

Алгоритм сортировки подсчетом состоит из цикла, который проходит по всем элементам массива, запуская вложенный цикл, подсчитывающий число таких элементов [4].

Как видно из графика, приведенного на рисунке 1, двухъядерные процессоры i5-5350U и i5-7200U благодаря технологии Hyper-Threading ускоряют выполнение задачи при установленных режимах с 2мя, 3мя и 4мя потоками. Также у процессора i-7200U наблюдается незначительное снижение производительности при числе потоков больше четырех. Это говорит о том, что технология OpenMP не производит первоначальную проверку на число поддерживаемых процессором потоков. В работе процессора i7-9750 наблюдается аномалия на 6 потоках.

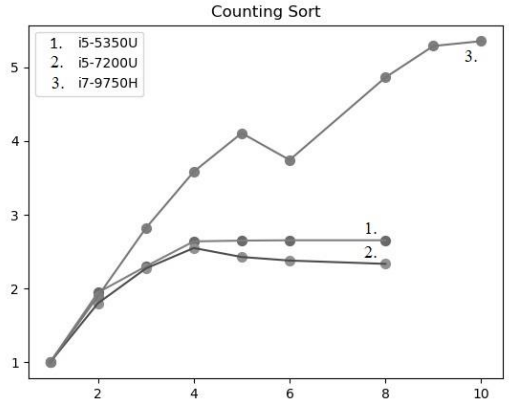

Рисунок 1. График для алгоритма Сортировка подсчетом

Алгоритм сортировки подсчетом состоит из двух циклов, вложенных друг в друга. Такие алгоритмы хорошо параллелятся при помощи OpenMP, что подтверждается полученными результатами.

Быстрая сортировка (quicksort)

Алгоритм быстрой сортировки также состоит из циклов, вложенных друг в друга [5]. Такие алгоритмы достаточно хорошо параллелятся. На рисунке 2

приведен график зависимости коэффициента параллелизации от количества потоков для процессоров Intel i5-5450U, i5-7200U и i7-5350H.

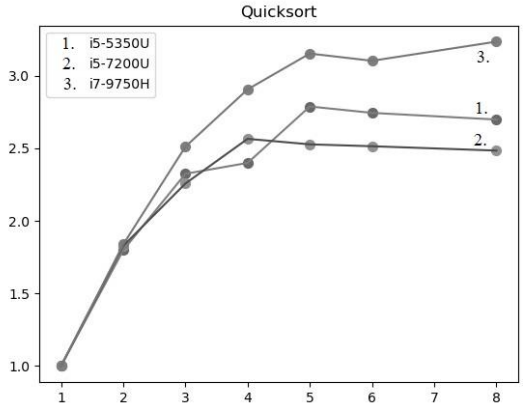

Рисунок 2. График для алгоритма Быстрая сортировка

Данные графика показывают, что алгоритм быстрой сортировки на 6 потоках и больше перестает поддаваться дальнейшему распараллеливанию. Дело в том, что в программе используется директива фреймворка OpenMP omp single, которая сигнализирует программе выполнять секцию кода полностью одним потоком. Таким образом, следует аккуратно относиться к данной директиве для избежания падения производительности программы. Кроме того, можно заметить, что на графике процессора i7-9750H при 6 потоках также наблюдается проседание производительности.

Поиск подстроки в тексте (Naive Pattern Searching)

Алгоритм Naive Pattern Searching относится к семейству алгоритмов поиска подстроки в строке. Данный алгоритм пробегается по тексту, последовательно проверяя символы текста на совпадение с символами паттерна поиска. При несовпадении любого из символов паттерна и символа текста «каретка» поиска сдвигается на одну позицию и поиск совпадений происходит заново [6].

Алгоритм поиска подстроки в тексте поддерживает распараллеливание между двумя потоками, при увеличении числа потоков производительность резко падает. Кроме того, на графике 3 видно, что коэффициент распараллеливания у процессора i5-5350U приблизительно равен коэффициенту

процессора i7-9750H, который показал лучшие результаты в остальных методах. Коэффициент процессора i5-7200U также незначительно уступает остальным. Возможно, данный факт связан с незначительным объемом тестовых данных.

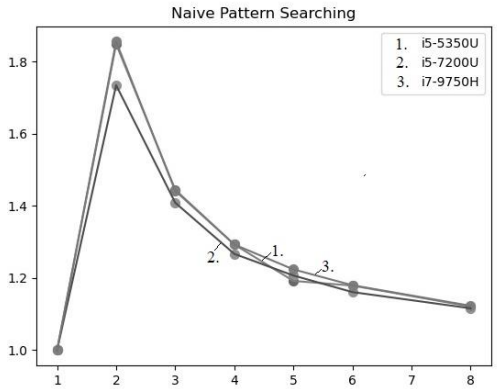

Рисунок 3. График для алгоритма поиска подстроки

Далее приведен листинг параллельной части программы. Из листинга видно, что программа использует директиву omp critical, которая позволяет исполнять определенную секцию кода только одному потоку в определенный момент времени. Падение производительности программы при увеличении числа потоков обусловлено структурой программы, которая использует выражение условного перехода if .. else. В случае использования трех потоков и больше, секция кода if(tid==0) будет исполняться первым потоком, а секция кода else по очереди будет выполняться оставшимися потоками. В итоге, чем больше потоков заявлено в программе, тем меньший объем данных будет обработан каждым потоком по отдельности и тем больше времени займет простой потоков, ожидающих выполнения секции else.

```
#pragma omp parallel num_threads(NUM_THREADS) private(tid, start, end) shared(text, pat, rem, bs, m)
{
	tid=omp_get_thread_num();
	if(tid==0) {
		#pragma omp critical (part1) {
			start=tid;
```

```
                end=bs-1;
                search(text, start, end, pat);
        }
    } else {
        #pragma omp critical (part2) {
                start=(tid*bs)-lenp;
                end=(tid*bs)+bs-1;
                search(text, start, end, pat);
        }
    }
}
```

Умножение матриц

Алгоритм перемножения матриц содержит 3 цикла, вложенных друг в друга. Ниже приведен его листинг.

```
#pragma omp parallel for private(i,j,k) shared(A,B,C)
for (i = 0; i < N; ++i) {
    for (j = 0; j < N; ++j) {
        for (k = 0; k < N; ++k) {
            C[i][j] += A[i][k] * B[k][j];
} } }
```

По графику 4 видно, что алгоритм умножения матриц не поддается распараллеливанию, причем в случае процессора i7-9750H при увеличении числа потоков наблюдается уменьшение производительности до 90 процентов от однопоточного режима.

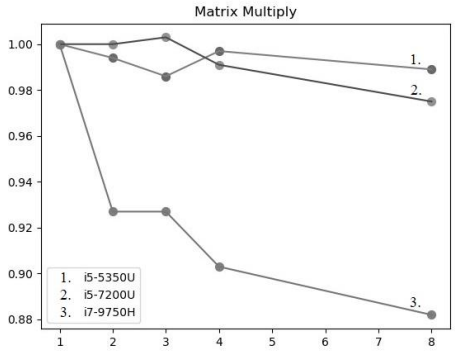

Рисунок 4. График для алгоритма умножения матриц

В данном случае OpenMP оказался неэффективен потому, что в программе используются 3 вложенных цикла, которые планировщику не удается распределить по потокам.

Метод Гаусса – Зейделя решения системы линейных уравнений

Метод Гаусса — Зейделя является классическим итерационным методом решения линейных уравнений. В реализации метода на языке Си используется 2 цикла, один из которых вложен в другой, с накоплением суммы в переменной при помощи директивы reduction. На графике 5 приведена зависимость коэффициента параллелизации от количества потоков для данного алгоритма.

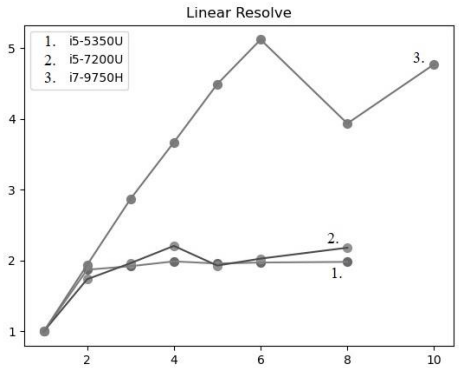

Рисунок 5. График для алгоритма Гаусса – Зейделя

По приведенному графику можно сделать вывод, что процессоры i5-5350U и i5-7200U позволяют достичь увеличения производительности в 2 раза при числе потоков, равным двум, однако при дальнейшем увеличении потоков коэффициент меняется незначительно. Иная картина наблюдается у процессора i7-9750H. График зависимости коэффициента от числа потоков линейно

увеличивается до 6 потоков. Тот факт, что коэффициент распараллеливания увеличивается только до числа физических ядер процессора, а не до числа логических потоков, обусловлен необходимостью использования директивы reduction, которая предназначена для выполнения конкретной операции над значениями переменной в разных потоках.

Далее приведен листинг программы. Из листинга видно, что при помощи reduction происходит общее суммирование переменной diff по всем потокам. При отсутствии данной директивы, что безусловно нарушает принцип работы алгоритма, наблюдается линейное увеличение коэффициента распараллеливания до 12 потоков на процессоре i7-9750H и до 4 на процессорах i5-5350U и i5-7200U.

```
#pragma omp parallel for num_threads(num_ths) schedule(static, max_cells_per_th) collapse(2) reduction(+:diff)
for (int i = 1; i < n-1; i++) {
	for (int j = 1; j < m-1; j++) {
		const float temp = (*mat)[pos];
		(*mat)[pos] =
			0.2f * ((*mat)[pos] + (*mat)[pos - 1] +
			(*mat)[pos — n] + (*mat)[pos + 1] + (*mat)[pos + n]);
		diff += abs((*mat)[pos] - temp);
	}
}
```

Алгоритм поиска простых чисел

Алгоритм поиска простых чисел, использующийся в программе, поочередно перебирает числа. Метод в цикле пытается обнаружить делитель исследуемого числа N, пока не переберет числа от 2-х до (N+1)/2. Таким образом, в программе используются два цикла, один вложен в другой.

В случае алгоритма поиска простых чисел график процессоров i5-5350U и i5-7200U идут практически симметрично. Процессор i5-5350U позволяет достичь максимальной производительности при 3х потоках, при дальнейшем

увеличении потоков коэффициент остается неизменным. Максимальная производительность у процессора i5-7200U достигается при 5ти потоках. По графику (рисунок 6) процессора i7-9750H видно, что максимальная производительность достигается при 7ми потоках. У процессоров i5-7200U и i7-9750H при 6ти потоках наблюдается спад производительности до коэффициентов 1.7 и 1.9 соответственно.

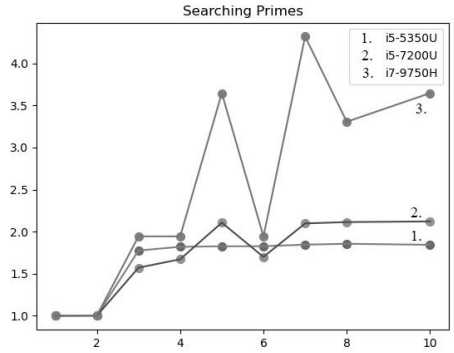

Рисунок 6. График для алгоритма поиска простых чисел

У всех трех процессоров при выполнении алгоритма с использованием двух потоков время выполнения остается тем же, что и в однопоточном режиме. Это связано с использованием директивы schedule(static, 1), которая указывает программе, каким образом поделить блоки данных между потоками [7]. В алгоритме поиска простых чисел происходит поочередная проверка чисел от 2 до N путем деления проверяемого числа на все меньшие числа и сравнения частного с 0. В связи с этим сложность обработки разных блоков, данных неравномерна. На рисунке 7 показан пример разделения данных между потоками без использования данной директивы.

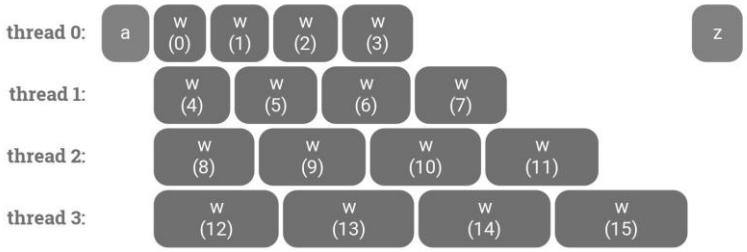

Рисунок 7. Пример разделения данных между потоками без schedule

По рисунку видно, что последний поток работает существенно дольше первого. Поэтому в таких случаях целесообразно добавлять директиву schedule(static, 1), в которой число 1 указывает, что программе необходимо распределять блоки данных между потоками циклически с шагом 1, то есть первый блок данных первому потоку, второй — второму и так далее. На рисунке 8 приведена временная диаграмма работы программы с использованием директивы schedule. Видно, что данные распределены более равномерно.

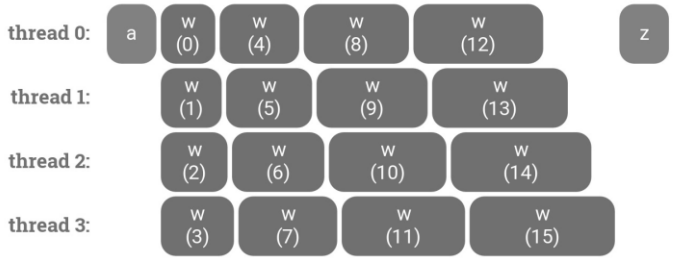

Рисунок 8. Пример разделения данных между потоками с schedule

График зависимости коэффициента ускорения от количества потоков, приведенный выше, соответствует программе с директивой schedule (static, 1). Примерная структура программы приведена в листинге.

```
#pragma omp parallel num_threads(t)
{
    …
    #pragma omp for nowait schedule(static, 1)
    for (i = 2; i <= N; ++i)
    {local_primes[j++] = i;}
    for (i = 0; i < j; ++i)
    {...}
    for (i = 0; i < j; ++i)
    {...}
}
```

Если в программе директиву schedule(static, 1) заменить на schedule(static, 2), график изменится. График для процессора i7-9750H при schedule(static, 2) приведен на рисунке 9. Как видно из графика, горизонтальная область переместилась из промежутка [1,2] в [2,3], что говорит о том, что для разного числа активных потоков целесообразно указывать различные значения в директиве schedule для достижения большей производительности программы.

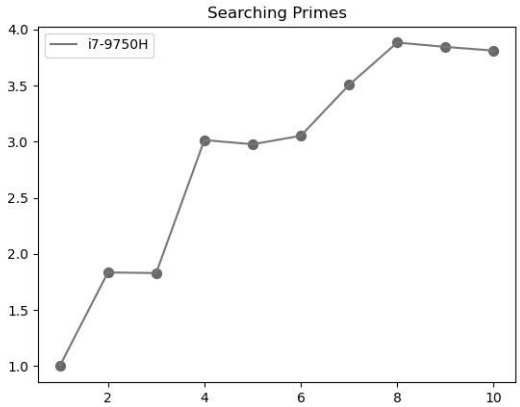

Рисунок 9. График для алгоритма поиска простых чисел на процессоре i7-9750H

Заключение

По приведенным выше данным можно сделать вывод, что фреймворк OpenMP хорошо подходит для решения задач, преимущественно состоящих из циклов. В таких задачах у процессора существует возможность выполнять один и тот же кусок кода одновременно для разного набора данных. Также стоит отметить, что разработчику необходимо аккуратно использовать директиву omp single, которая может существенно снизить производительность программы при некорректном использовании.

Также существуют программы, в которых использование рассматриваемого фреймворка может привести к падению производительности, пропорциональном количеству ядер процессора. Алгоритм умножения матриц служит примером такой программы. В силу специфики алгоритма, планировщику не удается эффективно распределить данные по потокам.

Следует отметить, что использование директивы schedule (static, N) может привести к существенному увеличению производительности в случае программ с равномерно нарастающей сложностью последующих итераций цикла.

Библиографический список:

1. S. Salvini, Unlocking the Power of OpenMP. Fifth European Workshop OpenMP (EWOMP '03), 2003.
2. J. M. Bull, Measuring Synchronization and Scheduling Overheads in OpenMP, The European Workshop on OpenMP (EWOMP ’99), 1999.
3. 8 Ways to Measure Execution Time in C/C++ [Электронный ресурс]. – URL: https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9 (дата обращения: 28.03.2021).
4. [Back to basics] Counting sort [Электронный ресурс]. – URL: https://medium.com/programming-basics/counting-sort-461cf29c9407 (дата обращения: 30.03.2021).
5. QuickSort [Электронный ресурс]. – URL: https://www.geeksforgeeks.org/quick-sort/ (дата обращения: 31.03.2021).
6. Naive algorithm for Pattern Searching [Электронный ресурс]. – URL: https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/ (дата обращения: 03.04.2021).
7. Multithreading with OpenMP [Электронный ресурс]. – URL: https://ppc.cs.aalto.fi/ch3/schedule/ (дата обращения: 09.04.2021).