ЛАБОРАТОРНАЯ РАБОТА № 1

ПАРАЛЛЕЛЬНОЕ ВЫЧИСЛЕНИЕ СУММЫ РЯДА СРЕДСТВАМИ АВТОМАТИЧЕСКОГО РАСПАРАЛЛЕЛИВАНИЯ И OPENMP

1. Цель работы

Научиться использовать основные директивы, функции и переменные окружения интерфейса OpenMP на примерах написания простейших параллельных программ для многоядерных вычислительных систем.

2. Постановка задачи и описание алгоритма

В настоящее время использование многопроцессорных вычислительных систем (МВС) с распределенной памятью представляется оправданным в двух основных случаях:

1) решаемая задача настолько сложна и ресурсоемка, что существующие персональные компьютеры и рабочие станции в принципе не позволяют ее решить с требуемой точностью;
2) задача решаема на персональных компьютерах и рабочих станциях, но затрачиваемое на ее решение время неприемлемо велико.

Использование МВС требует перехода от использования общепринятых и хорошо апробированных последовательных алгоритмов к алгоритмам параллельным, что в свою очередь ведет к смене всей парадигмы программирования. Одним из негативных факторов такого перехода является, в частности, существенное (иногда на порядок) увеличение времени разработки и отладки параллельной программы по сравнению с соответствующей последовательной программой.

Из любой достаточно сложной задачи всегда можно выделить ряд существенно более простых, большинство которых решается с помощью типовых алгоритмов. Классическим примером такой типовой, с точки зрения программирования, задачи является нахождение суммы элементов числового ряда:

$$S = \sum_{n=1}^{N} a_n = a_1 + a_2 + \ldots + a_N, \quad a_n \in \boldsymbol{R},\ n \in \boldsymbol{N}. \qquad (1.1)$$

Очевидно, что решение этой задачи на обычном персональном компьютере не представляет никакой сложности. Поэтому, с учетом сказанного вначале, решение этой задачи на МВС и разработка соответствующего параллельного алгоритма может иметь единственной целью сокращение времени вычисления. Если данная задача должна решаться многократно (например, в рамках какой-либо более общей задачи), то положительный эффект от ее распараллеливания может оказаться весьма значительным.

Для принципиальной возможности решения на МВС задача должна обладать внутренним параллелизмом, то есть допускать декомпозицию на ряд относительно независимых подзадач. Если для решения каждой подзадачи выделить отдельный процессор, то в силу независимости эти задачи могут решаться одновременно, то есть параллельно. Максимальный положительный эффект от такой декомпозиции достигается в том случае, когда все подзадачи требуют одинакового процессорного времени для своего решения.

Задача (1.1) обладает большим внутренним параллелизмом в силу ассоциативности операции сложения. Пусть в нашем распоряжении имеется МВС с количеством процессоров равном P. Будем также полагать, что число суммируемых членов ряда много больше количества процессоров, то есть $P << N$. Тогда на первом этапе задача (1.1) может быть разделена на P независимых подзадач вычисления отдельных частей суммы ряда:

$$S = \sum_{n=1}^{N_1} a_n + \sum_{n=N_1+1}^{N_2} a_n \ldots + \sum_{n=N_{P-1}+1}^{N} a_n = \sum_{p=1}^{P} S_p, \qquad (1.2)$$

где $S_p = \sum_{n=N_{p-1}+1}^{N_p} a_n$, $N_0 = 0$, $N_P = N$. Каждая сумма S_p может вычисляться своим процессором с номером $p = 1, 2, \ldots, P$.

Второй этап решения задачи заключается в вычислении окончательной суммы ряда S как суммы всех найденных на первом этапе частичных сумм S_p. Оба этапа проиллюстрированы на рис. 1.1.

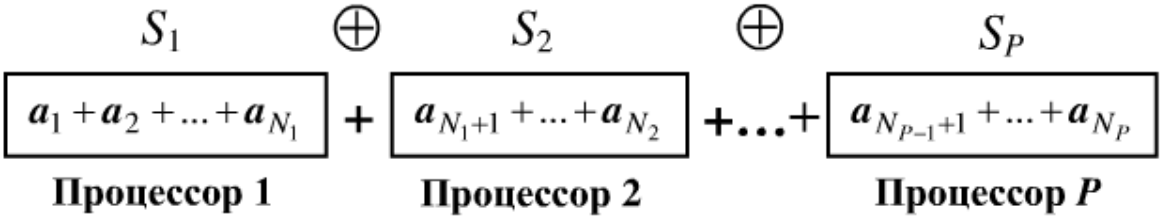

Рис.1.1. Параллельный алгоритм решения задачи нахождения суммы числового ряда

При переходе от последовательной схемы решения задачи к параллельной важнейшим является вопрос эффективности такого перехода. Для ее численной оценки традиционно используются два параметра – ускорение и эффективность.

Определение 1.1. Отношение времени выполнения последовательной программы T_1 ко времени выполнения параллельной программы на p процессорах T_p называется *ускорением* при использовании p процессоров:

$$S_p = \frac{T_1}{T_p}.$$

Отметим, что время T_1 известно не всегда, так как программно может реализовываться только параллельный алгоритм. В этом случае может использоваться оценка ускорения вида

$$S_p^* = \frac{T_1^*}{T_p}, \qquad (1.3)$$

где T_1^* – время выполнения параллельной программы на одном процессоре.

Определение 1.2. Отношение ускорения S_p к количеству процессоров p называется *эффективностью* при использовании p процессоров:

$$E_p = \frac{S_p}{p}.$$

Эффективность показывает среднюю загруженность процессоров при работе параллельной программы. Аналогично ускорению можно использовать оценку эффективности в виде

$$E_p^* = \frac{S_p^*}{p}. \qquad (1.4)$$

3. Основы интерфейса OpenMP

За прошедшее десятилетие с момента стандартизации интерфейс OpenMP Application Program Interface стал наиболее популярным высокоуровневым средством разработки многопоточных приложений для систем с общей памятью. Основными преимуществами данной

технологии считаются переносимость, возможность постепенного («инкрементного») распараллеливания, возможность совмещения в разрабатываемом исходном коде последовательной и параллельной версий программы. В настоящее время поддержка OpenMP вложена в большинство современных компиляторов (Microsoft Visual C++, Intel, Sun Studio, PathScale, Portland Group International, IBM XL, GNU Compiler Collection), работающих на различных платформах под управлением Windows, Linux, Unix, Solaris и других операционных систем. В связи с массовым распространением многоядерных систем технология OpenMP становится еще более привлекательной.

Интерфейс OpenMP включает в себя директивы компилятора, функции библиотеки времени исполнения (RTL) и переменные окружения. В данной лабораторной работе рассматриваются основные директивы OpenMP, используемые при распараллеливании последовательных программ, способы контроля и управления количеством потоков, а также различные варианты синхронизации.

Директива parallel

Начнём знакомство с интерфейсом OpenMP с распараллеливания простейшей последовательной программы, написанной на языке C, которая осуществляет вывод на экран приветствия «*Hello, World!*»:

```
#include <stdio.h>
int main()
{
    printf ("Hello, world!\n");
    return 0;
}
```

Для получения многопоточного варианта программы воспользуемся основной директивой OpenMP - `parallel`. После ее выполнения становятся доступными несколько потоков, которые могут параллельно выполнять участок программного кода, следующего за директивой – параллельную область. Программа, в которой приветствие «*Hello, World!*» выводится каждым из потоков, может выглядеть следующим образом:

```
#include <omp.h>
#include <stdio.h>
int main()
{
    #pragma omp parallel
        printf ("Hello, world!\n");
    return 0;
}
```

Функции omp_get_thread_num и omp_get_num_threads

Количество потоков, доступных в параллельной области, будет по умолчанию равно количеству логических процессоров, «видимых» операционной системой. Во время существования у каждого потока имеется свой порядковый номер, который можно получить при помощи функции библиотеки OpenMP `omp_get_thread_num`. Определение количества потоков, выделенного для выполнения параллельной секции, производится функцией `omp_get_num_threads`. Для использования функций библиотеки OpenMP обязательным является подключение заголовочного файла `omp.h`. Пользуясь вышеприведенными функциями, модернизируем параллельную программу так, чтобы в строке приветствия каждый поток выводил свой порядковый номер и общее число порожденных потоков.

Спецификатор private

Если для хранения порядкового номера и количества потоков используются переменные, объявленные в области последовательного кода, то во избежание конкурентного доступа к ним как общим (`shared`) переменным несколькими потоками одновременно, необходимо применить спецификатор `private` директивы `parallel`. Данный спецификатор позволяет задать список переменных `list`, которые в параллельной области программы будут локальными переменными потоков. При этом начальные значения этих переменных при входе в параллельную область не определены, даже если перед директивой одноименные переменные в мастер-потоке были инициализированы. При выходе из

параллельной области значения всех этих локальных переменных теряются, включая и значения в мастер-потоке. Переменные в списке `list` указываются через запятую.

Пример получения информации о потоках

Ниже приведен исходный код, в котором переменные `num_ths` и `th_num`, объявленные в последовательной области, используются каждым потоком для хранения уникального порядкового номера и количества потоков, поэтому для корректной работы программы должны задаваться в спецификаторе `private`:

```
…
int num_ths, th_num;
#pragma omp parallel private (num_ths, th_num)
{
    th_num = omp_get_thread_num();
    num_ths = omp_get_num_threads();
    printf("Hello, world! I am %d from %d\n",
        th_num, num_ths);
}
…
```

Управление количеством потоков

Рассмотрим различные варианты управления количеством потоков в OpenMP программе. Как было сказано выше, по умолчанию в параллельной области будет порождено столько потоков, сколько логических процессоров «видит» операционная система. Более приоритетной является настройка, определяемая переменной окружения `OMP_NUM_THREADS`. Преимущество управления потоками через данную переменную заключается в том, что в программу не требуется вносить изменения. При каждом новом запуске программы можно модифицировать значение данной переменной окружения и получать желаемое число потоков.

Команды установки переменных окружения зависят от используемой операционной системы и командной оболочки. В Windows для установки значения `8` в переменную `OMP_NUM_THREADS` потребуется выполнить команду

```
set OMP_NUM_THREADS = 8
```

В Linux/Unix – в зависимости от командной оболочки

cshell(csh): `setenv OMP_NUM_THREADS 8`

bash: `export OMP_NUM_THREADS = 8`

Две другие еще более приоритетные настройки должны располагаться непосредственно в исходном коде: функция `omp_set_num_threads` (вызывается в последовательной области) и спецификатор `num_threads` директивы `parallel`, позволяющий задать количество рабочих потоков непосредственно для каждой конкретной параллельной области.

Директива for

Основным способом распараллеливания в OpenMP (для языков C, C++) является разделение по потокам итерационных циклов for. Предназначенная для этого директива `for` обязана располагаться внутри параллельной области, предварительно выделенной с помощью директивы `parallel`. При использовании директивы `for` распараллеливанию подвергается только один цикл, следующий непосредственно за ней. Директива допускает использование семи различных спецификаторов. В данной лабораторной работе для решения поставленной задачи потребуются два из них:

1) `reduction (operator : list);`
2) `schedule (kind [, chunk]).`

Спецификатор reduction

В качестве параметра `operator` спецификатора `reduction` указывается один из операторов, приведенных в табл. 2.1. Параметр `list` – список из одной или нескольких перечисленных через запятую переменных. При входе в параллельную область для каждого потока создаются локальные переменные, определенные в `list`. Начальные значения локальных переменных выбираются в зависимости от используемого оператора, как показано в табл. 2.1. При выходе из параллельной области выполняется обновление одноименных переменных мастер-потока финальными значениями локальных переменных с использованием редукционного оператора.

В приведенном ниже примере, для того чтобы не возникало конфликтных ситуаций при доступе к общей переменной, хранящей значение произведения, спецификатор `reduction` применяется к переменной `pr` с использованием оператора умножения:

```
    double pr = 2;
#pragma omp parallel
#pragma omp for reduction (* : pr)
    for (i = 1; i <= N; i++)
        pr *= i / (1. + i);
```

Таблица 2.1

Оператор	Назначение	Начальное значение
+	Сумма	0
*	Произведение	1
&	Побитовое «И»	~0
|	Побитовое «ИЛИ»	0
^	Побитовый сдвиг	0
&&	Логическое «И»	1
||	Логическое «ИЛИ»	0

Тогда на время работы цикла в каждом потоке будет создаваться локальная переменная `pr`, инициализированная единицей, а при выходе из цикла все локальные произведения будут перемножены со значением общей переменной `pr`.

Спецификатор schedule

Спецификатор `schedule` задает правило распределения итераций цикла между параллельными потоками. С помощью этого спецификатора можно улучшить балансировку потоков и оптимизировать работу с кэш-памятью. Допускается четыре значения параметра `kind`: `static`, `dynamic`, `guided`, `runtime`, которые определяют характер распределения итераций цикла:

а) `schedule(static,chunk)` – разделяет итерации цикла на равные части размером `chunk`, которые статически распределяются между доступными потоками, как правило, с чередованием (см. рис. 2.1, а). Если размер `chunk` не задан, цикл разделяется на равные части, число которых равно числу потоков (это распределение активно по умолчанию);

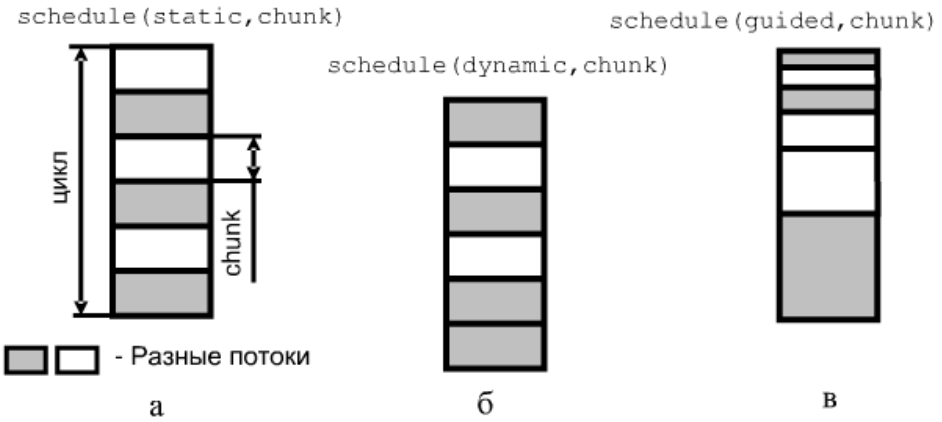

Рис. 2.1. Возможные формы спецификатора **schedule**

б) `schedule(dynamic,chunk)` – также разделяет итерации цикла на равные части размером `chunk`, но при этом распределение частей между потоками происходит динамически, в процессе выполнения программы: если поток закончил обработку своей части, он переходит к следующей, еще не обработанной, части (см. рис. 2.1, б);

в) `schedule(guided,chunk)` – размеры частей определяются динамически, пропорционально количеству оставшихся итераций (см. рис. 2.1, в). Размер части не может быть меньше `chunk`. Пусть `chunk=f`, число потоков `p`. Тогда текущий размер части `q` определяется как

`q = [n/p], n = max(n-q,p*f).`

Данный расчет является итерационным по `n`. На первой итерации `n` равно числу итераций цикла;

г) `schedule(runtime)` – вид распределения задается при запуске программы.

Комбинированная директива parallel for

Предыдущий пример исходного кода можно записать более лаконично, используя комбинированную директиву OpenMP `parallel for`:

```
    double pr = 2;
#pragma omp parallel for reduction (+ : pr)
    for (i = 1; i <= N; i++)
        pr *= i / (1. + i);
```

4. Выполнение работы

1. Выполнить программную реализацию на языке C / C++ последовательного алгоритма суммирования функционального ряда согласно варианту. Предусмотреть вывод на экран времени работы программы. Для компиляции использовать компилятор Intel.

Для замера времени использовать функцию `clock()`, описанную в заголовочном файле `time.h`, по аналогии с нижеприведенным примером:

```
#include <time.h>
…
clock_t t1, t2;
double time;
t1 = clock();
…// вычислительный блок
t2 = clock();
time = (t2 - t1) / (double) CLOCKS_PER_SEC;
…
```

2. Произвести распараллеливание программы путем добавления директив OpenMP в текст последовательной программы. Предусмотреть вывод на экран нулевым потоком количества потоков, доступных в параллельной области программы.

Для поддержки OpenMP необходимо подключить поддержку компилятором OpenMP (OpenMP support) в свойствах проекта.

3. Отладить написанные программы при небольшом числе членов ряда. Убедиться, что результаты последовательной и параллельной программ корректны и стабильны при многократных запусках. Убедиться, что количество порождаемых потоков равно числу ядер вычислительной системы.

4. Добавить опцию задания числа порождаемых потоков и запустить программу при различном числе потоков.

5. Подключить автоматическое распараллеливание к последовательной версии программы (ключ /QParallel). Убедиться, что количество порождаемых потоков равно числу ядер вычислительной системы.

6. Для последовательной программы подобрать три значения N так, чтобы время ее работы на многоядерной системе составляло 10 $(N = N_1)$, 20 $(N = N_2)$ и 40 $(N = N_3)$ секунд. Запустить многопоточные программы (OpenMP и автораспараллеливание) при аналогичных значениях N. Количество потоков должно равняться числу ядер вычислительной системы. Время работы каждой программы занести в табл. 2.2.

Таблица 2.2

Программа	T_p, с		
	N_1	N_2	N_3
Послед.			
OpenMP			
Авторасп.			

7. Для параллельных программ (автоматическое распараллеливание и OpenMP) вычислить ускорение и эффективность по формулам (1.3) и (1.4) лабораторной работы №1 для каждого N, полученные значения занести в табл. 2.3. Объяснить полученные результаты.

Таблица 2.3

Программа	S_p^*			E_p^*		
	N_1	N_2	N_3	N_1	N_2	N_3
OpenMP						
Авторасп.						

8. Провести запуски параллельной программы $(N = N_1)$, предварительно модифицируя исходный код для установки различных видов (`kind`) балансировки итераций параллельного цикла между потоками: `static`, `dynamic`, `guided`. Для каждого вида взять три значения параметра `chunk` (C): $C_1 = 1$, $C_2 = 100$, $C_3 = 10000$. Времена работы программ внести в табл. 2.4. Объяснить полученные результаты.

Таблица 2.4

`kind`	T, с		
	C_1	C_2	C_3
`static`			
`dynamic`			
`guided`			

9. Оформить отчет по результатам работы.

Приложение А
(обязательное)

Индивидуальные задания к лабораторным работам

А.1. ***Индивидуальные задания к лабораторной работе №1***

№	Ряд	№	Ряд	№	Ряд
1.	$\sum_{n=1}^{N}\frac{(-1)^n}{(3n-1)^2}$	10.	$\sum_{n=1}^{N}\frac{(-1)^n}{(3n-2)(3n+1)}$	19.	$\sum_{n=1}^{N}\frac{(-1)^{n-1}}{\sqrt{n}}$
2.	$\sum_{n=1}^{N}\frac{\sqrt[3]{n}}{(n+1)\sqrt{n}}$	11.	$\sum_{n=1}^{N}(-1)^{n-1}\frac{2n+1}{n(n+1)}$	20.	$\sum_{n=1}^{N}\frac{(-1)^n}{(2n+1)^3-1}$
3.	$\sum_{n=1}^{N}\frac{(-1)^n}{(n+1)^2-1}$	12.	$\sum_{n=1}^{N}(-1)^n\frac{n+1}{(n+1)\sqrt{n+1}-1}$	21.	$\sum_{n=1}^{N}\frac{(-1)^{n+1}}{2n-\sqrt{n}}$
4.	$\sum_{n=1}^{N}\frac{(-1)^{n-1}}{n^2}$	13.	$\sum_{n=1}^{N}\frac{\sin(2n+1)}{(n+1)^2(n+2)^2}$	22.	$\sum_{n=1}^{N}\frac{(-1)^{n-1}}{\ln(n+1)}$
5.	$\sum_{n=1}^{N}(-1)^n\frac{\ln n}{n}$	14.	$\sum_{n=1}^{N}(-1)^{n-1}\operatorname{tg}\left(\frac{1}{n\sqrt{n}}\right)$	23.	$\sum_{n=2}^{N}\frac{(-1)^n}{n\sqrt[3]{n}-\sqrt{n}}$
6.	$\sum_{n=1}^{N}\frac{\sin(1/n^2)}{(5n-1)^2}$	15.	$\sum_{n=1}^{N}\frac{\sin(5n+1)}{(6n+4)^2(7n-1)^3}$	24.	$\sum_{n=1}^{N}\frac{\sin(2n-1)}{(2n-1)^2}$
7.	$\sum_{n=1}^{N}\frac{(-1)^{n-1}}{(2n-1)^2}$	16.	$\sum_{n=1}^{N}(-1)^n\ln\left(\frac{n^2+1}{n^2}\right)$	25.	$\sum_{n=1}^{N}\frac{(-1)^n}{n-\ln n}$
8.	$\sum_{n=2}^{N}\frac{1}{n\ln^2 n}$	17.	$\sum_{n=1}^{N}\frac{(-1)^n}{n(n+1)(n+2)}$		
9.	$\sum_{n=2}^{N}\frac{(-1)^{n-1}}{n^2-n}$	18.	$\sum_{n=1}^{N}\left(1-\cos\left(\frac{\pi}{n}\right)\right)$		