Библиотека “OpenMP”

Презентацию составил: Кремер Илья
Группа: НК-401

Презентация является приложением к реферату.

Распараллеливание программ

Существуют две основные причины, вызывающие необходимость распараллеливать программы:

- Выполнение программой двух и более одновременных действий, задач
- Ускорение работы программы на современных компьютерах (с 2005 на ПК, с 2001 на серверах) и эффективное использование суперкомпьютеров (с 1962)

Основные способы распараллеливания

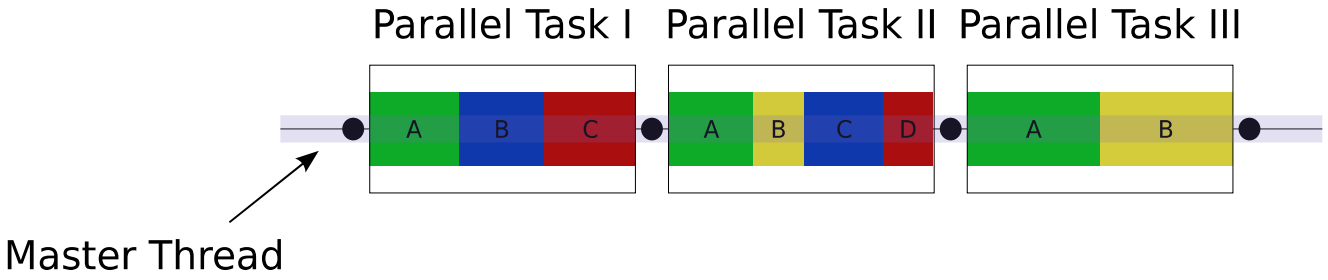
- Встроенные в язык средства создания новых потоков (threads). Например `fork()` в C, класс `Thread` в Java
- Дополнительные инструменты, обеспечивающие возможность передачи сообщений между процессами, (напр. MPI)
- Внедрение специальных «распараллеливающих» конструкций в уже написанную программу (напр. OpenMP)

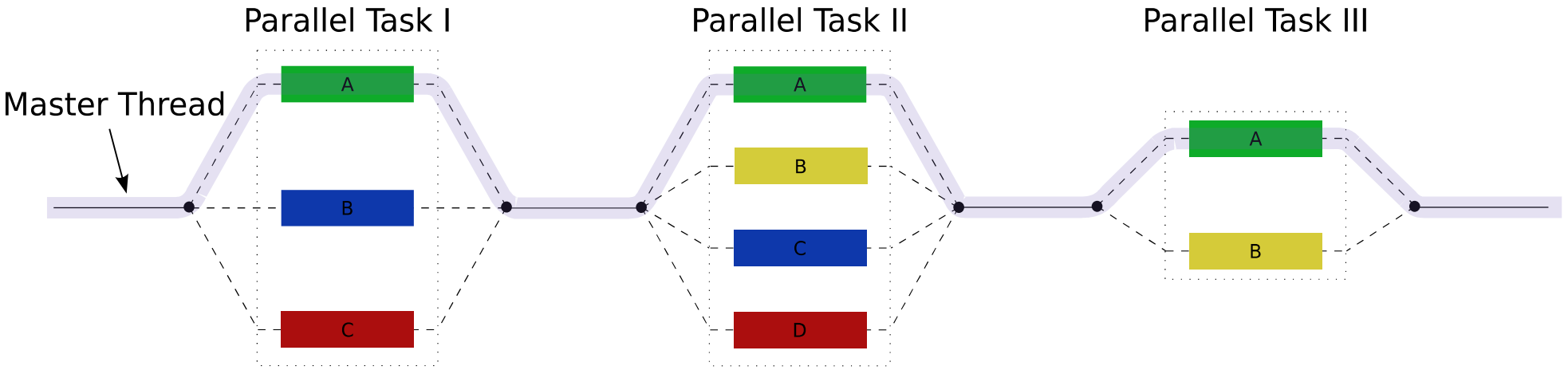
OpenMP – распараллеливающие конструкции

OMP – очень распространённый инструмент для распараллеливания, имеет ряд достоинств:

- Сначала пишем, потом распараллеливаем (в основном)
- Очень маленький, следовательно, простой набор директив
- Поддерживается многими компиляторами
- Не портит код, если не поддерживается

Визуализация распараллеливания





Ключевые директивы

Управление параллелилизмом ведётся с помощью директив (напр. в C это `#pragma omp ...`, в Fortran это `!$omp ...`) и нескольких функций:

- Директива `parallel` (начало параллельной области)
- Директивы `do/for` (Fortran/C)
- Директивы управления данными `shared/private`
- Директивы для синхронизации и управления потоками: `barrier, master`
- Дополнения к директивам

- Функции для управления свойствами параллельных областей (`set_nested(), get_thread_num(), set_dynamic()` и др.)

- Переменные окружения

Пример

Демонстрация работы параллельной области на C:

```
#include <stdio.h>
#include <omp.h>/* omp_lib.h в Fortran */

int main() {
    int pid;
    printf("До распараллеливания...\n");
#pragma omp parallel num_threads(3), private(pid)
  {
    pid = omp_get_thread_num();
    printf("Hello, World! (pid: %d)\n", pid);
  }
}
```

Пример

Можно задавать количество процессов по требованию пользователя или как-либо в ходе выполнения программы:

```
    printf("Количество процессов: ");
    scanf ("%d", &numthreads);
    if (numthreads > 128)
        numthreads = 128;
    omp_set_num_threads(numthreads);
#pragma omp parallel private(...)
/* Опускаем num_threads(3) из пред. примера */
```

По умолчанию количество потоков соответствует количеству логических ядер. Это делает распараллеливание ещё более лёгким, тем более, что их количество можно узнать через `omp_get_num_procs()`.

Кроме того, можно использовать переменные среды:

```
$ env OMP_NUM_THREADS=4 ./a.out
```

Элементарный способ распараллеливания

Как было видно из пред. примеров, все порождённые процессы нумерованы от `0` до `numprocs - 1`, следовательно самый простой способ распределить выполнение задачи, это:

```
#pragma omp parallel private(pid)
  {
    pid = omp_get_thread_num();
    if (pid == 0) {
        //часть задачи 1
    } else if (pid == 1) {
        //часть задачи 2
    } //...
  }
```

Циклы

Однако, более всего востребованно распараллеливание циклов. В OMP есть специальные директивы, которые всё сделают сами.

```
#define N 4096

int main() {
    double a[N][N], b[N][N], c[N][N];
    int i, j, k;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            //инициализация матриц:
            a[i][j] = b[i][j] = i + j;
#pragma omp parallel for shared(a, b, c) private(i, j, k)
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) {
            c[i][j] = 0;
            for (k = 0; k < N; k++)
                c[i][j] += a[i][k] * a[k][j];
        }
}
```

Циклы

В коде приведённом на пред. слайде итерации цикла автоматически распрелятся между процессами (количество процессов – по умолчанию). Однако возникает вопрос управления данными. В пред. примерах мы писали `private(pid)`, это означает, что в каждой нити будет своя копия переменной `pid`. Если в пред. циклах индексные переменные были бы общими для всех процессов, то каждое выполнение `c[i][j] += a[i][k] * a[k][j]` смогло бы не оказаться уникальным – процессы перебивали бы друг друга во время присваивания значений `i`,`j` и `k`.

В то же время сами матрицы могут и должны быть общими для всех, во-первых не произойдёт конфликта, во-вторых это сэкономит память.

Если `private`/`shared` не указано, то что-то подставится по умолчанию (зависит от реализации).

Эффективность

Пример с перемножением двух матриц отлично показывает эффективность OpenMP. Измерим время выполнения цикла с помощью omp-функции замера времени:

```
double t1 = omp_get_wtime();
/* перемножение */
printf("Прошло %lf с.\n", omp_get_wtime() - t1);
```

Тест, проведённый на современном 4-ядерном процессоре показывает:

Нитей	Время, сек	Загрузка CPU, %
1	2500	13
2	900	25
4	600	50
8	500	100

Простота

Использование класса Thread в Java показывает сопоставимые результаты – от 2900 до 650 секунд, однако сложность программирования такой элементарной задачи возростает в разы.

С помощью расширения `schedule` директивы `for` можно указать, как именно будет происходить распределение итераций. Общий вид записи: `schedule(type, chunk)`:

```
#pragma omp parallel for schedule(static, 10)
```

`static` означает, что итерации раздаются поровну, `dynamic` – динамически (след. порцию получает тот, кто закончил выполнять предыдущую), `guided` – `dynamic` с уменьшением до `chunk`, `auto` – выбирается компилятором или системой (без `chunk`), `runtime` – автоматически во время выполнения (без `chunk`).

Более сложные случаи

Вначале было сказано, что OMP позволяет заниматься распараллеливанием после написания самой программы. К сожалению это не всегда так. Рассмотрим две типовых задачи.

Первая – поиск минимального или максимального элемента.

Вторая – вычисление последовательности, каждый член которой зависит от предудыщуего (например, факториала).

Проблема первой состоит в правильном управлении данных – минимум не может быть ни частным, ни общим.

Проблема второй состоит в невозможности распараллелить цикл (как можно вычислить n-ый элемент из середины, когда его предшественник к этому моменту может быть ещё не вычислен?).

Первый тип задач

Рассмотрим проблему первой задачи. Поиск минимума идёт следующим образом: 1) Полагается, что первый элемент – минимальный. 2) В цикле сравается следующий с предыдущим, и если он ещё меньше, то минимум меняется на него.

Почему переменная, содержащая текущий минимум не может быть private? Потому что тогда в каждом процессе найдётся свой минимум. В то же время, она не может быть и общей для всех, ведь тогда может начаться гонка данных (как в случае с индексными переменнами в 9-ом слайде).

Однако, выход из этой ситуации, всё-таки прост: завести столько переменных для поиска минимума, сколько действует процессов т.е. создать массив, например:

```
int mins[numthreads]
```

Каждый процесс найдёт свой минимум, а после останется только сравнить полученные числа.

Второй тип задач

Вычисление последовательностей тоже имеет решение. В случае с факториалом всё просто – для двух нитей вычисляем отдельно произведение чётных чисел в одной и нечётных в другой, а затем умножаем результаты. Для трёх – шагаем через 2 и т.д. Более сложной представляется задача такого вида:

Есть массив A[N]. Необходимо вычислить S[N]:
S[i] = Произведение(A[0] … A[i]) / Сумма(A[0] … A[i])

Вычислять задачу строго по формуле чересчур затратно и неправильно – с каждым следующим элементом S[i] будет всё больше и больше вычислений. Поэтому линейное программное решение выглядит след. образом:

```
Sn = Sd = A[0];
for (int i = 1; i < N; i++) {
    S[i] = Sn / Sd;
    Sn = Sn * A[i];
    Sd = Sd + A[i];
}
```

Второй тип задач

Можно использовать два процесса, с тем, чтобы один считал только числители, а другой – только знаменатели:

```
    nums[0] = denoms[0] = A[0];
    omp_set_num_threads(2);
#pragma omp parallel private(i, pid) shared(n, td, tn)
  {
    pid = omp_get_thread_num();
    if (pid == 0) {
        for (i = 1; i < N; i++)
            nums[i] = nums[i - 1] * A[i];
    } else {
        for (i = 1; i < N; i++)
            denoms[i] = denoms[i - 1] + A[i];
    }
  }
/* далее -  вычисление S[i] в распараллеленом цикле */
```

Если при вычислении S использовать 8 процессов, то общая скорость решения увеличится в 1,5 раза.

Второй тип задач, усложнение

Однако, на стадии вычисления числителей и знаменателей мы используем всего два процесса. Попробуем с большей эффективность использовать большее количество процессов.

Вычислим двойки сумм для A: $(A_0 + A_1)$, $(A_1 + A_2)$, $(A_2 + A_3)$ … и аналогично двойки произведений. Поскольку их можно вычислять независимо друг от друга, сделаем это в таком цикле:

```
#pragma omp for schedule(static, 50)
  {
    tn[i] = a[i] * a[i+1];
    td[i] = a[i] + a[i+1];
  }
#prarma omp barrier
```

Остановка необходима, т.к. далее необходимы все подсчитанные элементы tn и td. Затем пусть каждая нить делает одно из заданий: 1) Считать нечётные числители 2) Считать чётные числители 3) Чётные знаменатели 4) Нечётные знаменатели.

Второй тип задач, усложнение

```
pid = omp_get_thread_num();
if (pid == 0) {
    nums[0] = A[0];
    for (i = 2; i < N; i+= 2)
        nums[i] = nums[i - 2] * tn[i - 1];
} else if (pid == 1) {
    denoms[0] = A[0];
    for (i = 2; i < N; i+= 2)
            denoms[i] = denoms[i - 2] + td[i - 1];
} else if (pid == 2) {
    nums[1] = tn[0];
    for (i = 3; i < N; i+= 2)
        nums[i] = nums[i - 2] * tn[i - 1];
} else if (pid == 3) {
    denoms[1] = td[0];
    for (i = 3; i < N; i+= 2)
        denoms[i] = denoms[i - 2] + td[i - 1];
}
/* далее -  вычисление S[i] в распараллеленом цикле */
```

Второй тип задач, усложнение

Такое решение в среднем на 30% быстрее, чем предыдущее (все три варианта проводились с 2 млн элементами), т.е. же в два раза быстрее нераспараллеленого варианта. Можно проделать такую же работу для троек, четвёрок и т.д. сумм и произведений, ещё сильнее увеличив производительность программы.

Существуют куда более сложные задачи на распараллеливание, но всё же OMP является одним из первых инструментов, применимых для таких задач.

Литература и др. источники

Антонов А.С. «Параллельное программирование с использованием технологии OpenMP», – Издательство Московского Университета, 2009.

`http://parallel.ru/` – Информационно-аналитический центр.

Спасибо за внимание!