Computer Science МФТИ

Add features in OpenMP

Субботина Анна
МФТИ

Содержание лекции

- Задачи (Tasking)
- Hybrid OpenMP/MPI

Понятие задания

Задачи появились в OpenMP 3.0

Каждая задача:

- Представляет собой последовательность операторов, которые необходимо выполнить
- Включает в себя данные, которые используются при выполнении этих операторов
- Выполняется некоторой нитью

Понятие задания

- В OpenMP 3.0 каждый оператор программы является частью одной из задач
- При входе в параллельную область создаются неявные задачи (implicit task), по одной задаче для каждой нити
- Создается группа нитей
- Каждая нить из группы выполняет одну из задач
- По завершении выполнения параллельной области, master-thread ожидает, пока не будут завершены все неявные задачи

Директива "task"

```
#pragma omp task
```

```
#pragma omp task
```

```
#pragma omp taskwait
```

```
!$omp flush taskwait
```

Пример "HelloWorld"

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
        printf("Hello ");
        printf("World ");
        printf("\n");
        return(0);
}
```

Что программа напечатает?

Пример "HelloWorld"

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
        printf("Hello ");
        printf("World ");
        printf("\n");
        return(0);
}
```

```
$ cc hello.c
$ ./a.out
Hello World
$
```

Пример "HelloWorld"

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
        #pragma omp parallel
        {
                printf("Hello ");
                printf("World ");
        } // конец параллельной области
        printf("\n");
        return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export
OMP_NUM_THREADS=2
$ ./a.out
```

Что программа напечатает?

Пример "HelloWorld"

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World Hello World
```

Заметим, что программа может также напечатать “Hello Hello World World”

Пример "HelloWorld"

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
        #pragma omp parallel
        {
                #pragma omp single
                {
                        printf("Hello ");
                        printf("World ");
                }
        } // конец параллельной области
        printf("\n");
        return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export
OMP_NUM_THREADS=2
$ ./a.out
```

Что программа напечатает?

Пример "HelloWorld"

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
```

... но теперь код выполняет только ОДИН поток

Пример "HelloWorld"

```
int main(int argc, char *argv[]) {

        #pragma omp parallel
        {
                #pragma omp single
                {
                        #pragma omp task
                                {printf("Hello ");}
                        #pragma omp task
                                {printf("World ");}
                }
        } // конец параллельной области
        printf("\n");
        return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export
OMP_NUM_THREADS=2
$ ./a.out
```

Что программа напечатает?

Пример "HelloWorld"

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
$ ./a.out
Hello World
$ ./a.out
World Hello
```

задания могут выполняться в произвольном порядке

Пример "HelloWorld"

```
int main(int argc, char *argv[]) {
        #pragma omp parallel
        {
                #pragma omp single
                {
                        #pragma omp task
                                {printf("Hello ");}
                        #pragma omp task
                                {printf("World ");}
                        printf(“\nThank You “);
                }
        } // конец параллельной области
        printf("\n");
        return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export
OMP_NUM_THREADS=2
$ ./a.out
```

Что программа напечатает?

Пример "HelloWorld"

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Thank You World Hello
$ ./a.out
Thank You Hello World
$ ./a.out
Thank You World Hello
```

задания выполняются в некоторой "точке исполнения заданий"

Пример "HelloWorld"

```
int main(int argc, char *argv[]) {
      #pragma omp parallel
      {
            #pragma omp single
            {
                  #pragma omp task
                        {printf("Hello ");}
                  #pragma omp task
                        {printf("World ");}
                  #pragma omp taskwait
                  printf("\nThank You ");
            }
      } // конец параллельной области
printf("\n");return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export
OMP_NUM_THREADS=2
$ ./a.out
```

Что программа напечатает?

Пример "HelloWorld"

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
$
World Hello
Thank You
$ ./a.out
World Hello
Thank You
$ ./a.out
Hello World
Thank You
$
```

теперь СНАЧАЛА выполняются задания

Завершение задания

- Неявная барьерная синхронизация потоков
- Явная барьерная синхронизация потоков
  - C/C++: #pragma omp barrier
  - Fortran: !$omp barrier
- Собственная барьерная синхронизация (task barrier)
  - C/C++: #pragma omp taskwait
  - Fortran: !$omp taskwait

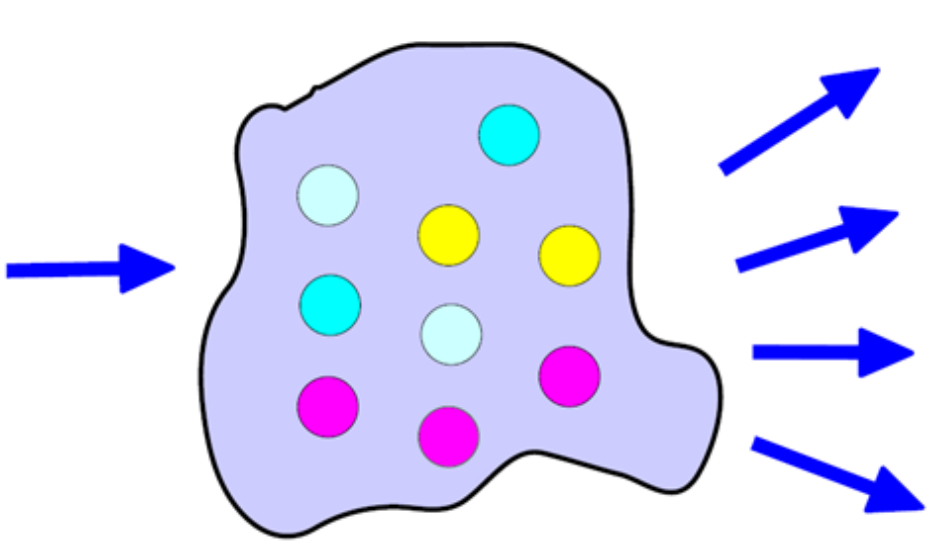
Связанный список (пример)

```
.....
while(my_pointer) {
        (void) do_independent_work (my_pointer);
        my_pointer = my_pointer->next;
} // конец цикла
.....
```

До OpenMP 3.0 раелизовать связанный список было тяжело:
сначала надо было посчитать количество итераций, затем переписать цикл while() в цикл for()

Модель исполнения заданий

входной поток добавляет задание(я) в пул заданий



потоки исполняют задания из пула

Разработчик определяет задания в приложении
Runtime система выполняет задания

Связанный список (пример)

```
my_pointer = listhead;
#pragma omp parallel
{
      #pragma omp single nowait
      {
            while(my_pointer) {
                  #pragma omp task firstprivate(my_pointer)
                  {
                        (void) do_independent_work (my_pointer);
                  }
            my_pointer = my_pointer->next ;
            }
      } // нет барьерной синхронизации (nowait)
} // конец параллельной области -
  // подразумевается барьерная синхронизация
```

Числа Фибоначчи (пример)

Последовательность Фибоначчи определяется следующим образом:

F(0) = 1
F(1) = 1
F(n) = F(n-1) + F(n-2) (n=2,3,4,.....)

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Числа Фибоначчи (пример)

Рекурсивный алгоритм:
(не самый быстрый, взят для примера)

```
long comp_fib_numbers(int n){
        // Basic algorithm: f(n) = f(n-1) + f(n-2)
        long fnm1, fnm2, fn;
        if ( n == 0 || n == 1 ) return(n);
        fnm1 = comp_fib_numbers(n-1);
        fnm2 = comp_fib_numbers(n-2);
        fn = fnm1 + fnm2;
        return(fn);
}
```

Числа Фибоначчи (пример)

```
// основная программа
#pragma omp parallel
shared(nthreads)
{
#pragma omp single nowait
{
result = comp_fib_numbers(n);
} // конец single области
} // конец параллельной
  // области
```

```
long comp_fib_numbers(int n){
    // алгоритм: f(n) = f(n-1) + f(n-2)
    long fnm1, fnm2, fn;
    if ( n == 0 || n == 1 ) return(n);
    #pragma omp task shared(fnm1)
        {fnm1 = comp_fib_numbers(n-1);}
    #pragma omp task shared(fnm2)
        {fnm2 = comp_fib_numbers(n-2);}
    #pragma omp taskwait
    fn = fnm1 + fnm2;
    return(fn);
```

Числа Фибоначчи (пример)

```
long comp_fib_numbers(int n){
// параллельный рекурсивный алгоритм: f(n) = f(n-1) + f(n-2)
long fnm1, fnm2, fn;
if ( n == 0 || n == 1 ) return(n);  // k - некоторое число, до которого
// считать параллельно не эффективно (например, k = 20)
if ( n > k ) return(comp_fib_numbers(n-1) + comp_fib_numbers(n-2));
#pragma omp task shared(fnm1)
      {fnm1 = comp_fib_numbers(n-1);}
#pragma omp task shared(fnm2)
      {fnm2 = comp_fib_numbers(n-2);}
#pragma omp taskwait
fn = fnm1 + fnm2;
return(fn);}
```

Числа Фибоначчи (пример)

```
$ export OMP_NUM_THREADS=1
$ ./fibonacci-omp.exe 40
Parallel result for n = 40: 102334155
(1 threads needed 5.63 seconds)
$ export OMP_NUM_THREADS=2
$ ./fibonacci-omp.exe 40
Parallel result for n = 40: 102334155
(2 threads needed 3.03 seconds)
$
```

Hybrid OpenMP/MPI

Порядок действий:

- Сначала программа параллелится с помощью MPI – на сокетах
- Далее внутри сокета (по ядрам процессоров) программа параллелится с помощью OpenMP

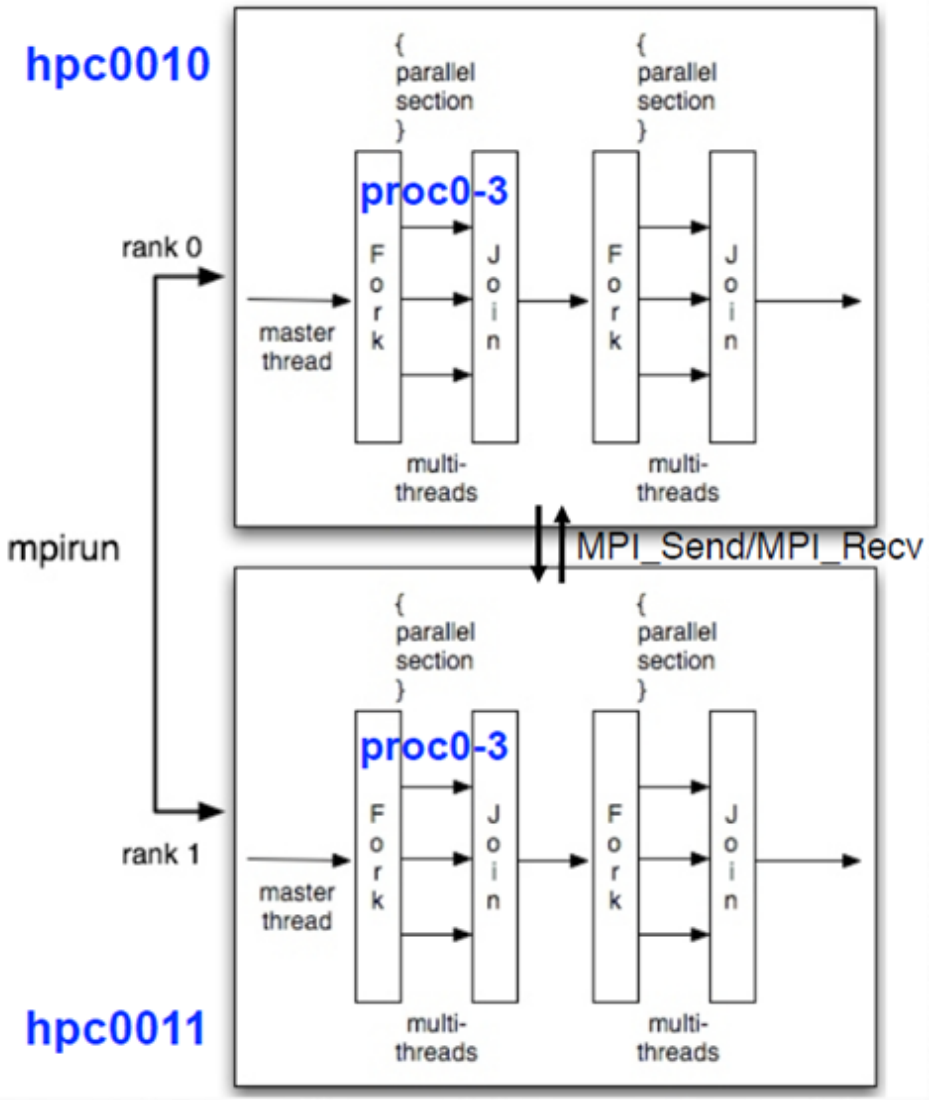
Смысл:

- можно перенести часть обменов в кэш процессора без потерь времени на пересылку (аналог общей памяти) для OpenMP, что в конечном счете приводит к ускорению расчетов

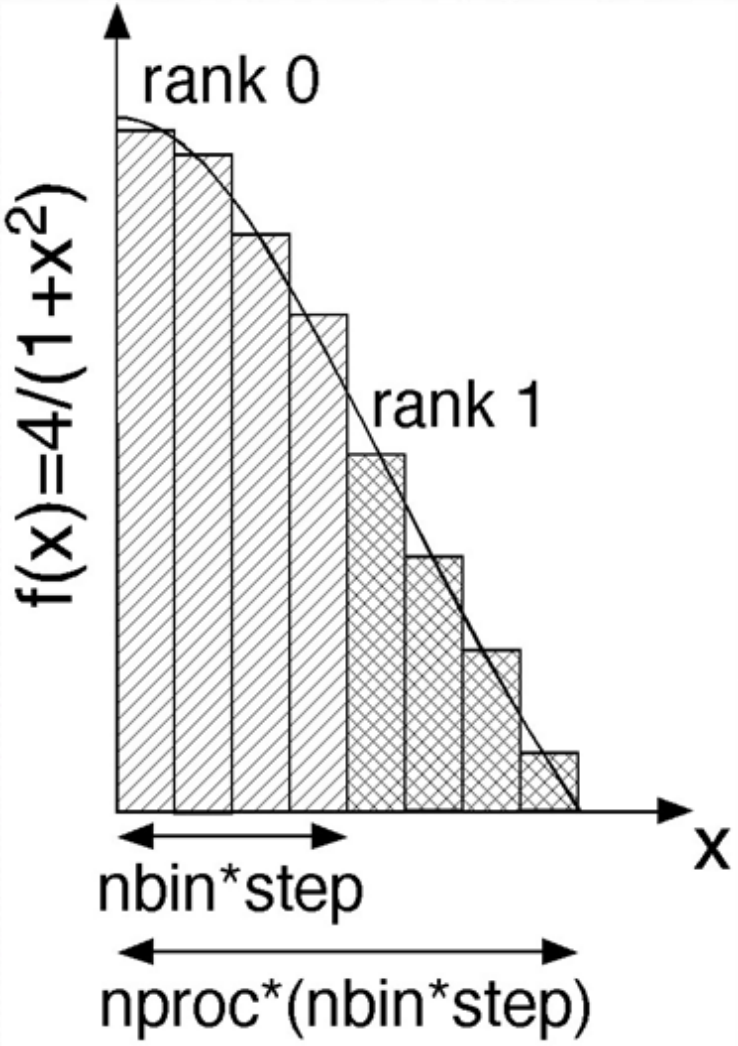
Hybrid OpenMP/MPI

каждый процесс порождает множество потоков

для запуска:
mpirun –np 2
в коде:
omp_set_num_threads(3);



Вычисление числа pi



- **Каждый MPI процесс интегрирует по отрезку 1/nproc, считая площадь каждого из nbin прямоугольников шириной step**
- **В тоже время внутри каждого MPI процесса, nthreads потоков рассчитывают частичные суммы на OpenMP**

```
#define NBIN 100000
#define MAX_THREADS 8
void main(int argc,char **argv) {
int nbin, myid, nproc, nthreads, tid;
double step, sum[MAX_THREADS] = {0.0}, pi = 0.0, pig;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);
nbin = NBIN/nproc; step = 1.0/(nbin*nproc);
#pragma omp parallel private(tid) {
int i; double x;
nthreads = omp_get_num_threads(); tid = omp_get_thread_num();
for (i=nbin*myid+tid; i<nbin*(myid+1); i+=nthreads) {
x = (i + 0.5) * step; sum[tid] += 4.0 / (1.0 + x * x);}
printf("rank tid sum = %d %d %e\n",myid, tid, sum[tid]); }
for (tid = 0; tid < nthreads; tid++) pi += sum[tid] * step;
MPI_Allreduce(&pi,&pig,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WOR
LD);
if (myid==0) printf("PI = %f\n",pig);
MPI_Finalize();}
```

Вычисление числа pi

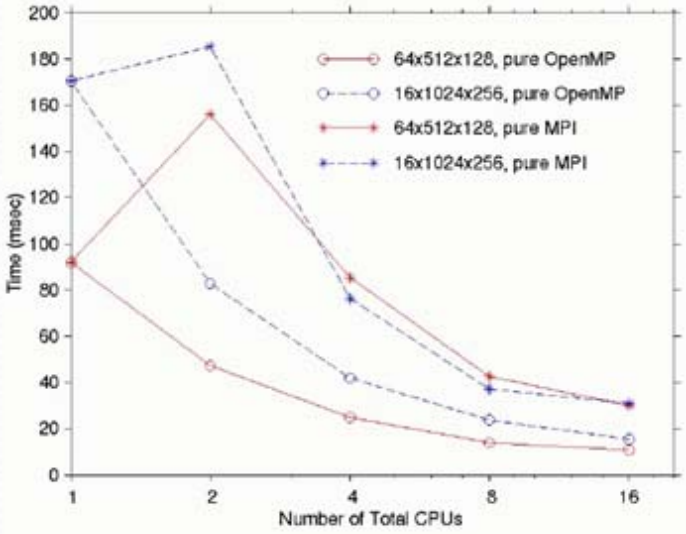
- **Компиляция**

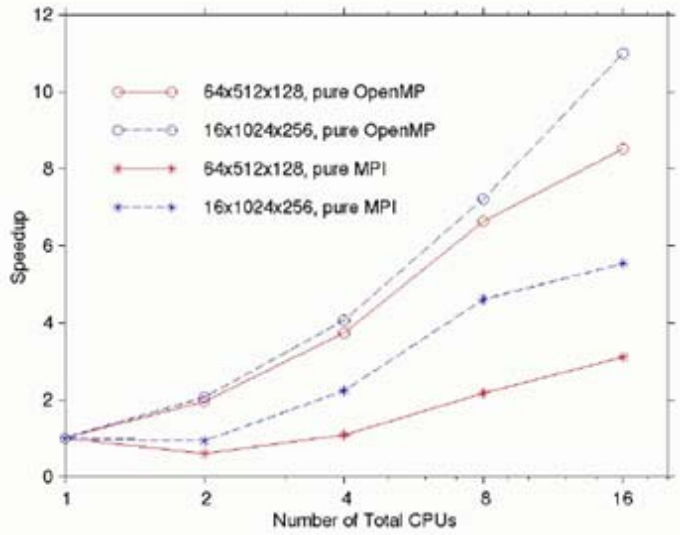
```
mpicc -o hpi hpi.c -openmp
```

- **Вывод**

```
rank tid sum = 1 1 6.434981e+04
rank tid sum = 1 0 6.435041e+04
rank tid sum = 0 0 9.272972e+04
rank tid sum = 0 1 9.272932e+04
PI = 3.141593
```

Сравнение производительности





OpenMP vs. MPI (16 CPUs)
64x512x128: 2.76 быстрее
16x1024x256:1.99 быстрее

MM5 Regional Weather Prediction Model

Метод	время на коммуникацию (сек)	общее время счета (сек)
64 MPI tasks	494	1755
16 MPI tasks with 4 threads/task	281	1505

время на коммуникацию уменьшилось на 85%
так же уменьшилось время расчета программы

Вопросы?