Решаване на задачи

Курс „Паралелно програмиране“

ИНСТИТУТ *за* СЪВРЕМЕННИ
ФИЗИЧЕСКИ ИЗСЛЕДВАНИЯ

Стоян Мишев

Задачи

Запълване на масив

Синхронизиране на нишки с critical

Reduction

Master и Single

Запълване на елементи на масив без OpenMP

```
1   #define TOTAL 2048 // 100000
2
3       int A[TOTAL];
4       clock_t start, end;
5       double cpu_time_used;
6
7       // Start timing
8       start = clock();
9
10      for (int i = 0; i < TOTAL; ++i)
11      {
12          A[i] = i * i;
13          // Comment out printf to reduce I/O overhead
14          // printf("%02d=%03d\n", i, A[i]);
15      }
16
17      // End timing
18      end = clock();
19      cpu_time_used = ((double)(end - start)) /
            CLOCKS_PER_SEC;
20
```

```
1  #define TOTAL 2048 // 100000
2       int A[TOTAL];
3       double start, end;
4       double cpu_time_used;
5       // Set the number of threads
6       omp_set_num_threads(4);
7       // Start timing
8       start = omp_get_wtime();
9
10 #pragma omp parallel for
11      for (int i = 0; i < TOTAL; ++i)
12      {
13          A[i] = i * i;
14          // Comment out printf to reduce I/O overhead
15          // printf("Th[%d]: %02d=%03d\n",
                omp_get_thread_num(), i, A[i]);
16      }
17      // End timing
18      end = omp_get_wtime();
19      cpu_time_used = end - start;
```

N = 2048
Non-OpenMP Execution Time: 0.000003 seconds
OpenMP Execution Time: 0.000102 seconds
N = 1000000
Non-OpenMP Execution Time: 0.001654 seconds
OpenMP Execution Time: 0.000623 seconds

Синхронизиране на нишки 1/2

```
1   #define TOTAL 1000
2 float funcao_complexa_1(int i) {
3     return (float)(i * 2.0); // Simple function
4 }
5 float funcao_complexa_2(float B) {
6     return (float)(B + 2.0); // Simple function
7 }
8 int main() {
9     float res = 0.0; // Shared variable to accumulate
          results
```

Синхронизиране на нишки 2/2

```
1  #pragma omp parallel
2      {
3          float B; // Private variable for each thread
4          int i, id, nthreads;
5          id = omp_get_thread_num(); // Get the thread ID
6          nthreads = omp_get_num_threads(); // Get the
               number of threads
7          // Divide the work among threads
8          for (i = id; i < TOTAL; i += nthreads)
9          {
10             B = funcao_complexa_1(i); // Perform a
                   complex computation
11 // Critical section: only one thread can execute this
       at a time
12 #pragma omp critical
13             {
14                 res += funcao_complexa_2(B); // Update
                       the shared variable
15             }
16         }
17     }
```

Reduction без OpenMP 1/2


```
1 #include <random>
2 #include <vector>
3
4 static void baseline() {
5   // Create a random number generator
6   std::random_device rd;
7   std::mt19937 mt(rd());
8   std::uniform_real_distribution dist(0.0f, 1.0f);
9
10  // Create vectors of random numbers
11  const int num_elements = 1 << 20;
12  std::vector<float> v_in;
13  std::generate_n(std::back_inserter(v_in),
        num_elements, [&]{return dist(mt);});
```

Reduction без OpenMP 2/2


```
1     // Timing loop
2     for (auto _ : s) {
3       // Create our variable to accumulate into
4       float sink = 0;
5
6       // Run the sum of squares
7       for (int i = 0; i < num_elements; i++) {
8         // Square v_in and add to sink
9         sink += v_in[i] * v_in[i];
10      }
11    }
12  }
```

Reduction c OpenMP 1/2


```
1   #include <random>
2 #include <vector>
3
4 static void baseline() {
5   // Create a random number generator
6   std::random_device rd;
7   std::mt19937 mt(rd());
8   std::uniform_real_distribution dist(0.0f, 1.0f);
9
10  // Create vectors of random numbers
11  const int num_elements = 1 << 20;
12  std::vector<float> v_in;
13  std::generate_n(std::back_inserter(v_in),
        num_elements, [&]{return dist(mt);});
```

Reduction c OpenMP 2/2

```
    // Timing loop
    for (auto _ : s) {
      // Create our variable to accumulate into
      float sink = 0;

      // Run the sum of squares
      // Parallelize the for loop
      #pragma omp parallel for reduction(+:sink)
      for (int i = 0; i < num_elements; i++) {
        // Square v_in and add to sink
        sink += v_in[i] * v_in[i];
      }
    }
}
```

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(void)
5  {
6      #pragma omp parallel
7      {
8          #pragma omp master
9          {
10             printf("Thread %d is executing master
                   construct\n", omp_get_thread_num());
11         }
12         printf("Thread %d is executing non-master
               construct\n", omp_get_thread_num());
13     }
14
15     return 0;
16 }
```

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main(void)
5 {
6     #pragma omp parallel
7     {
8         #pragma omp single
9         {
10            printf("Thread %d is executing single
                  construct\n", omp_get_thread_num());
11        }
12        printf("Thread %d is executing non-single
              construct\n", omp_get_thread_num());
13    }
14
15    return 0;
16 }
```