Programowanie równoległe
Laboratorium 8

Sprawozdanie

Zadania 3-7

Zadania brzmią:

- Stworzenie programu mnożącego dwa wektory o zmiennych typu double (losowych) i długości 1000000000.
- Powielenie pętli sekwencyjnej i jej zrównoleglenie w openmp.
- Powielenie pętli sekwencyjnej i jej zwektoryzowanie przez dodanie instrukcji #pragma omp simd.
- Dodanie procedur pomiaru czasu i porównanie czasów wykonania poszczególnych pętli

Kod źródłowy rozwiązania:

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

void fillVector(size_t n, double vector[n]) {
    // srand(time(NULL));

    for (; n > 0; --n) {
      vector[n - 1] = (double)rand() / RAND_MAX * 2.0 - 1.0;
    }
}

void scalarMultiplyVectors(size_t n, double vector1[n], double
vector2[n], double *result) {
    for (; n > 0; --n) {
      *result += vector1[n - 1] * vector2[n - 1];
    }
}

void scalarMultiplyVectorsOmp(size_t n, double vector1[n],
double vector2[n], double *result) {
    #pragma omp parallel for shared(vector1, vector2, result)
num_threads(8) schedule(static)
    for (int i = n; i > 0; --i) {
      double tmp = vector1[i - 1] * vector2[i - 1];
```

```
      #pragma omp critical
      {
           *result += tmp;
      }
    }
}

void scalarMultiplyVectorsSimd(size_t n, double vector1[n],
double vector2[n], double *result) {
    #pragma omp simd
    for (int i = n; i > 0; --i) {
      *result += vector1[i - 1] * vector2[i - 1];
    }
}

int main() {
    int n = 1000000000;
    double start, end;
    double vector1[n], vector2[n], result;

    fillVector(n, vector1);
    fillVector(n, vector2);

    result = 0;
    start = omp_get_wtime();
    scalarMultiplyVectors(n, vector1, vector2, &result);
    end = omp_get_wtime();
    printf("Vector multiplication result is:\t\t\t%lf\nTime
taken:\t%f\n", result, end - start);

    result = 0;
    start = omp_get_wtime();
    scalarMultiplyVectorsOmp(n, vector1, vector2, &result);
    end = omp_get_wtime();
    printf("Vector multiplication with the use of omp result
is:\t%lf\nTime taken:\t%f\n", result, end - start);

    result = 0;
    start = omp_get_wtime();
    scalarMultiplyVectorsSimd(n, vector1, vector2, &result);
    end = omp_get_wtime();
    printf("Vector multiplication with the use of simd result
is:\t%lf\nTime taken:\t%f\n", result, end - start);
```

```
    return 0;
}
```

Wydruk programu:

```
Vector multiplication result is:          73.762826
Time taken:    0.000282
Vector multiplication with the use of omp result is:
73.762826
Time taken:    0.009852
Vector multiplication with the use of simd result is:
73.762826
Time taken:    0.000348
```

Ze względu na to że mój komputer nie posiada wystarczającej ilości pamięci RAM dla alokacji dwóch wektorów typu double o długości 1000000000, przedstawiony wydruk jest dla n = 100000.
Jak widać z wydruku, jednowątkowa wersja funkcji skalarnego mnożenia wektorów zajmuje najmniej czasu; ale wykorzystanie simd jest prawie tak samo skuteczne jak i jednego wątku.

Użyte nowe dyrektywy:
#pragma omp simd — wektoryzacja części kodu.

Zadania 8-10

Zadania brzmią:

- Napisz program, który rekurencyjnie implementuje obliczanie funkcji Fibonacciego bez użycia dyrektywy task.
- Używając dyrektywy task zapisz ten sam kod co powyżej -przy większej liczbie, kod który napisałeś tworzy bardzo dużo zadań, zastanów się jak to ograniczyć i zmodyfikuj kod.

Kod źródłowy rozwiązania:

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

int fibbonacci(int n) {
    if (0 == n || 1 == n) {
      return n;
    } else {
```

```
      return fibbonacci(n - 1) + fibbonacci(n - 2);
    }
}

int fibbonacciTask(int n) {
    if (0 == n || 1 == n) {
      return n;
    } else {
      int tmp[2];

      #pragma omp task shared(n, tmp) if(n < 20)
      tmp[0] = fibbonacci(n - 1);

      #pragma omp task shared(n, tmp) if(n < 20)
      tmp[1] = fibbonacci(n - 2);

      #pragma omp taskwait
      return tmp[0] + tmp[1];
    }
}

int main() {
    for(int i = 0; i < 15; ++i) {
      printf("%d ", fibbonacci(i));
    }

    printf("\n");

    for(int i = 0; i < 15; ++i) {
      printf("%d ", fibbonacciTask(i));
    }

    printf("\n");

    return 0;
}
```

Wydruk programu:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Wydruk jest zgodny z ciągiem Fibonacciego z Wikipedii więc wynik jest poprawny.

Użyte nowe dyrektywy:
#pragma omp task — tworzy zadanie które jest wykonywane w osobnym wątku.
#pragma omp taskwait — oczekiwanie zakończenia wykonywania wszystkich wątków
if(n < 20) — tworzenie nowych wątków jest wykonywane tylko jeśli n jest mniejsze niż 20

Zadania 11-13

Zadania brzmią:

- Rozpakowanie i uruchomienie programu wyszukiwania wartości maksymalnej w tablicy
- Uzupełnienie programu o definicje zadań (tasks) –dla wersji równoległej openmp wyszukiwania liniowego –w dyrektywie task użyć klauzuli default(none) i ustalić jak poprawnie i optymalnie przeprowadzać obliczenia (jakich zmiennych użyć?).

Kod źródłowy rozwiązania:

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

#include "search_max_openmp.h"

/********************** linear search ************************/
double search_max(double *A, int p, int k) {
    double a_max = A[p];

    for(int i = p + 1; i <= k; ++i) {
      if(a_max < A[i]){
           a_max = A[i];
      }
    }

    return a_max;
}

/************* parallel linear search - openmp ***************/
double search_max_openmp_simple(double *A, int p, int k) {
    double a_max = A[p];
    double a_max_local = a_max;

    #pragma omp parallel default(none) firstprivate(A, p, k,
a_max_local) shared(a_max)
```

```
    {
      #pragma omp for
      for(int i = p + 1; i <= k; ++i) {
           if(a_max_local < A[i]) {
                a_max_local = A[i];
           }
      }

      #pragma omp critical (cs_a_max)
      {
           if(a_max < a_max_local) {
                a_max = a_max_local;
           }
      }

    }

    return a_max;
}

/************* parallel linear search - openmp ****************/
double search_max_openmp_task(double *A, int p, int k) {
    double a_max = A[p];

    #pragma omp parallel default(none) firstprivate(A, p, k)
shared(a_max)
    {
      #pragma omp single
      {
           int num_threads = omp_get_num_threads();
           int n = k - p + 1;

           int num_tasks = num_threads + 1;
           int n_loc = ceil(n / num_tasks);

           for(int itask = 0; itask < num_tasks; ++itask){
                int p_task = p + itask * n_loc;

                if(p_task > k) {
                     printf("Error in task decomposition!
Exiting.\n");
                     exit(0);
                }

                int k_task = p + (itask + 1) * n_loc - 1;
```

```
                if(itask == num_tasks - 1) {
                     k_task = k;
                }

                #pragma omp task
                {
                     for(int i = p + 1; i <= k; ++i) {
                          if(a_max < A[i]) {
                               a_max = A[i];
                          }
                     }
                } // end task definition
           } // end loop over tasks
      } // end single region
    } // end parallel region

    return a_max;
}


/************ binary search (array not sorted) ****************/
double bin_search_max(double *A, int p, int k) {
    if(p < k) {
      int s = (p + k) / 2;
      double a_max_1 = bin_search_max(A, p, s);
      double a_max_2 = bin_search_max(A, s + 1, k);

      //printf("p %d  k %d, maximal elements %lf, %lf\n", p, k,
a_max_1, a_max_2);

      if(a_max_1 < a_max_2) {
           return a_max_2;
      } else {
           return a_max_1;
      }

    } else {
      return A[p];
    }
}

/*** single task for parallel binary search (array not sorted) -
openmp ***/
#define max_level 4
```

```
double bin_search_max_task(double* A, int p, int r, int level) {
    if (level <= max_level)    {
      if(p < r) {
           int s = (p + r) / 2;
           double a_max_1, a_max_2;

           #pragma omp task shared(A, p, s, level, a_max_1)
           a_max_1 = bin_search_max_task(A, p, s, level + 1);

           #pragma omp task shared(A, p, s, level, a_max_2)
           a_max_2 = bin_search_max_task(A, s + 1, r, level +
1);

           //printf("p %d  r %d, maximal elements %lf, %lf\n",
p, r, a_max_1, a_max_2);

           #pragma omp taskwait
           if(a_max_1 < a_max_2) {
                return a_max_2;
           } else {
                return a_max_1;
           }

      } else {
           return A[p];
      }
    } else {
      return bin_search_max(A, p, r);
    }
}

/********** parallel binary search (array not sorted) - openmp
***********/

double bin_search_max_openmp(double *A, int p, int k) {
    double a_max;

    #pragma omp parallel default(none) firstprivate(A, p, k)
shared(a_max)
    {
      #pragma omp single
      {
           #pragma omp task
           {
                a_max = bin_search_max_task(A, p, k, 0);
```

```
            }
        }
    }

    return a_max;
}
```

Wydruk programu:

```
maximal element 2499.995000
time for sequential linear search: 0.001297
maximal element 2499.995000
time for parallel linear search: 0.000586
maximal element 2499.995000
time for parallel linear search: 0.003211
maximal element 2499.995000
time for sequential binary search: 0.004317
maximal element 2499.995000
time for parallel binary search: 0.001689
```

Najszybsza funkcja wyszukiwania maksymalnej liczby — to search_max_openmp_simple z wykorzystaniem #pragma omp for i #pragma omp critical.

Wyniki różnych funkcji zgadzają się z wynikami funkcji przykładowych więc są one poprawne.

Użyte nowe dyrektywy:
#pragma omp single — zezwolenie na wykonywanie bloku kodu tylko dla jednego wątku