Лабораторная работа №5 (Распределение вычислительной нагрузки)

Лабораторная работа №6 (Замки и барьеры)

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

#include <bitset>
#include <iostream>

const int N = 1000000;
using namespace std;

// функция для получения случайного числа между start и end
int getRandomNum(int start, int end) {
  return rand() % (end - start + 1) + start;
}

// функция для параллельного расчёта
int parallelTask(int arr_A[N], int arr_B[N]) {
  int* arr_C = new int[N];
  int answer = 0;
#pragma omp parallel
  {
#pragma omp for reduction(+ : answer)
    for (int i = 0; i < N; i++) {
      if (arr_A[i] >= arr_B[i]) {
        arr_C[i] = arr_A[i];
      } else {
        arr_C[i] = arr_B[i];
      }
      answer = answer + arr_C[i];
    }
  }
  return answer;
}

// функция для параллельного расчёта
```

```
int parallelAtomicTask(int arr_A[N], int arr_B[N]) {
  int* arr_C = new int[N];
  int answer = 0;
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < N; i++) {
      if (arr_A[i] >= arr_B[i]) {
        arr_C[i] = arr_A[i];
      } else {
        arr_C[i] = arr_B[i];
      }
#pragma omp atomic
      answer += arr_C[i];
    }
  }
  return answer;
}

// функция для параллельного расчёта
int parallelCriticalTask(int arr_A[N], int arr_B[N]) {
  int* arr_C = new int[N];
  int answer = 0;
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < N; i++) {
      if (arr_A[i] >= arr_B[i]) {
        arr_C[i] = arr_A[i];
      } else {
        arr_C[i] = arr_B[i];
      }
#pragma omp critical
      answer = answer + arr_C[i];
    }
  }
  return answer;
}
```

```
// функция для последовательного расчёта
int sequentialTask(int arr_A[N], int arr_B[N]) {
  int* arr_C = new int[N];
  int answer = 0;
  for (int i = 0; i < N; i++) {
    if (arr_A[i] >= arr_B[i]) {
      arr_C[i] = arr_A[i];
    } else {
      arr_C[i] = arr_B[i];
    }
    answer = answer + arr_C[i];
  }
  return answer;
}

// функция для расчёта с помощью двух секций
int sectionTwoTask(int arr_A[N], int arr_B[N]) {
  int* arr_C = new int[N];
  int answer = 0;

#pragma omp parallel sections
  {
#pragma omp section
    {
      for (int i = 0; i < N / 2; i++) {
        if (arr_A[i] >= arr_B[i]) {
          arr_C[i] = arr_A[i];
        } else {
          arr_C[i] = arr_B[i];
        }
#pragma omp atomic
        answer += arr_C[i];
      }
    }
#pragma omp section
    {
      for (int i = N / 2; i < N; i++) {
```

```
        if (arr_A[i] >= arr_B[i]) {
          arr_C[i] = arr_A[i];
        } else {
          arr_C[i] = arr_B[i];
        }
#pragma omp atomic
        answer += arr_C[i];
      }
    }
  }
  return answer;
}

// функция для расчёта с помощью четырёх секций
int sectionFourTask(int arr_A[N], int arr_B[N]) {
  int* arr_C = new int[N];
  int answer = 0;

#pragma omp parallel sections
  {
#pragma omp section
    {
      for (int i = 0; i < N / 4; i++) {
        if (arr_A[i] >= arr_B[i]) {
          arr_C[i] = arr_A[i];
        } else {
          arr_C[i] = arr_B[i];
        }
#pragma omp atomic
        answer += arr_C[i];
      }
    }
#pragma omp section
    {
      for (int i = N / 4; i < N / 2; i++) {
        if (arr_A[i] >= arr_B[i]) {
          arr_C[i] = arr_A[i];
        } else {
```

```
          arr_C[i] = arr_B[i];
        }
#pragma omp atomic
        answer += arr_C[i];
      }
    }
#pragma omp section
    {
      for (int i = N / 2; i < 3 * N / 4; i++) {
        if (arr_A[i] >= arr_B[i]) {
          arr_C[i] = arr_A[i];
        } else {
          arr_C[i] = arr_B[i];
        }
#pragma omp atomic
        answer += arr_C[i];
      }
    }
#pragma omp section
    {
      for (int i = 3 * N / 4; i < N; i++) {
        if (arr_A[i] >= arr_B[i]) {
          arr_C[i] = arr_A[i];
        } else {
          arr_C[i] = arr_B[i];
        }
#pragma omp atomic
        answer += arr_C[i];
      }
    }
  }
  return answer;
}

// функция для расчёта с помощью семафора
int semaphoreTask(int arr_A[N], int arr_B[N]) {
  int* arr_C = new int[N];
  int answer = 0;
```

```
  omp_lock_t lock;
  omp_init_lock(&lock);

#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < N; i++) {
      if (arr_A[i] >= arr_B[i]) {
        arr_C[i] = arr_A[i];
      } else {
        arr_C[i] = arr_B[i];
      }
      omp_set_lock(&lock);
      answer = answer + arr_C[i];
      omp_unset_lock(&lock);
    }
  }
  omp_destroy_lock(&lock);
  return answer;
}

// функция для расчёта с помощью барьера
int barrierTask(int arr_A[N], int arr_B[N]) {
  int* arr_C = new int[N];
  int answer = 0;
#pragma omp parallel
  {
#pragma omp for nowait
    for (int i = 0; i < N; i++) {
      if (arr_A[i] >= arr_B[i]) {
        arr_C[i] = arr_A[i];
      } else {
        arr_C[i] = arr_B[i];
      }
    }
#pragma omp barrier
#pragma omp master
```

```
    for (int i = 0; i < N; i++) {
      answer += arr_C[i];
    }
  }
  return answer;
}

int main() {
  // русификация программы
  setlocale(LC_ALL, "Russian");
  srand((unsigned int)time(NULL));
  int arr_A[N];
  int arr_B[N];

  // инициализация массивов  случайными данными
  for (int i = 0; i < N; i++) {
    arr_A[i] = getRandomNum(1, 100);
    arr_B[i] = getRandomNum(1, 100);
  }

  double start;
  double end;

  start = omp_get_wtime();
  sequentialTask(arr_A, arr_B);
  end = omp_get_wtime();
  printf("Regular work took %f seconds\n", end - start);

  start = omp_get_wtime();
  parallelTask(arr_A, arr_B);
  end = omp_get_wtime();
  printf("Parallel reduction work took %f seconds\n", end - start);

  start = omp_get_wtime();
  parallelAtomicTask(arr_A, arr_B);
  end = omp_get_wtime();
  printf("Parallel atomic work took %f seconds\n", end - start);
```

```
    start = omp_get_wtime();
    parallelCriticalTask(arr_A, arr_B);
    end = omp_get_wtime();
    printf("Parallel critical work took %f seconds\n", end - start);

    start = omp_get_wtime();
    sectionTwoTask(arr_A, arr_B);
    end = omp_get_wtime();
    printf("Two sections work took %f seconds\n", end - start);

    start = omp_get_wtime();
    sectionFourTask(arr_A, arr_B);
    end = omp_get_wtime();
    printf("Four sections work took %f seconds\n", end - start);

    start = omp_get_wtime();
    semaphoreTask(arr_A, arr_B);
    end = omp_get_wtime();
    printf("Semaphore work took %f seconds\n", end - start);

    start = omp_get_wtime();
    barrierTask(arr_A, arr_B);
    end = omp_get_wtime();
    printf("Barrier work took %f seconds\n", end - start);
    return 1;
}
```

```
Regular work took 0.002548 seconds
Parallel reduction work took 0.001370 seconds
Parallel atomic work took 0.020770 seconds
Parallel critical work took 0.027776 seconds
Two sections work took 0.016666 seconds
Four sections work took 0.006736 seconds
Semaphore work took 0.021147 seconds
Barrier work took 0.001759 seconds
```