# Introduction to Parallel Computing

## Shared-memory programming with OpenMP

## The reduction clause

- What is wrong with this code? (it prints the correct result)

```
//gcc-14 -o main -fopenmp p5_5_trapz_parallel_bad.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

double f(double x);     /* Function we're integrating */
double local_trap(double a, double b, int n);

int main(int argc, char *argv[]) {
    double global_result = 0.0;  /* Store result in global_result */
    double a, b;                 /* Left and right endpoints      */
    int n;                       /* Total number of trapezoids    */

    int thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);

#   pragma omp parallel num_threads(thread_count)
    {
#  pragma omp critical
        global_result += local_trap(a, b, n);
    }

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n", a, b, global_result);
    return 0;
}  /* main */

double f(double x) {
    return x * x;
}  /* f */

double local_trap(double a, double b, int n) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    double x;

    double h = (b - a) / n;
    int local_n = n / thread_count;
    double local_a = a + my_rank * local_n * h;
    double local_b = local_a + local_n * h;
    double my_result = (f(local_a) + f(local_b)) / 2.0;
    for (int i = 1; i <= local_n - 1; i++) {
        x = local_a + i * h;
        my_result += f(x);
    }
    my_result = my_result * h;
    return my_result;
}  /* local_trap */
```

    - The call to $local\_trap$ can only be **executed by one thread** at a time.
    - Effectively, we're forcing the threads to execute the trapezoidal rule **sequentially**.
- How to fix the previous code?
    - Just move the critical section after the function call.

```c
//gcc-14 -o main -fopenmp p5_6_trapz_parallel_fix.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

double f(double x); /* Function we're integrating */
double local_trap(double a, double b, int n);

int main(int argc, char *argv[]) {
    double global_result = 0.0; /* Store result in global_result */
    double a, b; /* Left and right endpoints */
    int n; /* Total number of trapezoids */
    int thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);

# pragma omp parallel num_threads(thread_count)
    {
        double my_result = 0.0;     /* private */
        my_result += local_trap(a, b, n);
# pragma omp critical
        global_result += my_result;
    }
    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n", a, b, global_result);
    return 0;
} /* main */

double f(double x) {
    return x * x;
} /* f */

double local_trap(double a, double b, int n) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    double x;
    double h = (b - a) / n;
    int local_n = n / thread_count;
    double local_a = a + my_rank * local_n * h;
    double local_b = local_a + local_n * h;
    double my_result = (f(local_a) + f(local_b)) / 2.0;
    for (int i = 1; i <= local_n - 1; i++) {
        x = local_a + i * h;
        my_result += f(x);
    }
    my_result = my_result * h;
    return my_result;
} /* local_trap */
```

- Now the call to $local\_trap$ is outside the critical section, and the threads can execute their calls simultaneously.
- Note that $my\_result$ is **private** because it is declared in the parallel block.

- OpenMP provides a cleaner method to avoid serializing the execution of $local\_trap$ by using the **reduction clause**.
- A **reduction clause** consists of two elements:

**Reduction operator**

- An associative binary operation (+, *)

**Reduction variable**

- The variable in which the results of an operation (+, *) are stored

**Reduction**

- A computation that repeatedly applies the same **reduction operator** to a **sequence of operands** to get a single result

- For example, if $A$ is an array of $n$ integers, the computation

```
int sum = 0;
for (i = 0; i < n; i++)
   sum += A[i];
```

is a reduction in which the **reduction operator** is **+** and the **reduction variable** is $sum$.

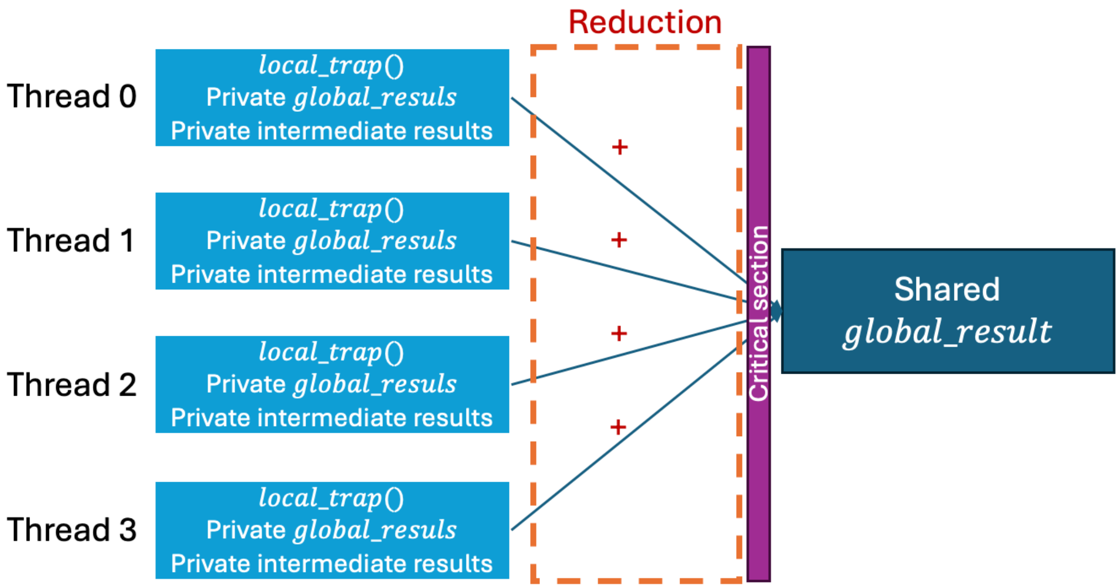
- Reduction clause in OpenMP is defined as follows

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

  - $< parallel\ directive >\ reduction(< operator >:< variable\ list >)$
  - Reduction variable → $global\_result$
  - Reduction operator → +

```c
//gcc-14 -o main -fopenmp p5_7_trapz_parallel_reduction.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

double f(double x);    /* Function we're integrating */
double local_trap(double a, double b, int n);

int main(int argc, char *argv[]) {
    double global_result = 0.0;  /* Store result in global_result */
    double a, b;                 /* Left and right endpoints      */
    int n;                     /* Total number of trapezoids    */
    int thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);

#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
    global_result += local_trap(a, b, n);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n",
           a, b, global_result);
    return 0;
}  /* main */

double f(double x) {
    double return_val;

    return_val = x * x;
    return return_val;
}  /* f */

double local_trap(double a, double b, int n) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    double x;
    double h = (b - a) / n;
    int local_n = n / thread_count;
    double local_a = a + my_rank * local_n * h;
    double local_b = local_a + local_n * h;
    double my_result = (f(local_a) + f(local_b)) / 2.0;
    for (int i = 1; i <= local_n - 1; i++) {
        x = local_a + i * h;
        my_result += f(x);
    }
    my_result = my_result * h;

    return my_result;
}  /* Trap */
```

- Behind the scenes,
  - OpenMP creates a private variable for each thread
  - The run-time system stores each thread's result in this private variable
  - OpenMP creates a critical section, and the values stored in the private variables are added in this critical section.




- A reduction operator can be $+, -, *, \&, |, \hat{}, \&\&$ $and$ $||$.
  - Why is **division not supported**? Because it is not commutative or associative.
  - Although **subtraction is not commutative or associative**, it's handled in a different way in OpenMP.
- $float$ and $double$ reduction variables may cause the results to be slightly different.
  - **Floating point arithmetic isn't associative**.
  - For example, if $a$, $b$, and $c$ are floats, then $(a + b) + c \neq a + (b + c)$.
- The **private variables** of each thread are initialized to a **default value** according to the datatype of the reduction variable

**Table 5.1** Identity values for the various reduction operators in OpenMP.

| Operator | Identity Value |
|---|---|
| + | 0 |
| * | 1 |
| - | 0 |
| & | ~0 |
| \| | 0 |
| ^ | 0 |
| && | 1 |
| \|\| | 0 |

## The *parallel for* directive

- Instead of using **explicit parallelization**, we can use the ***parallel for*** directive
- It's placed immediately before the *for* loop.
    - Serial program vs parallel program

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
# pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
for (i = 1; i <= n-1; i++)
      approx += f(a + i*h);
approx = h*approx;
```

- The system **divides** the iterations of the *for* loop among the threads.
- The **loop variable *i*** has a default **private** scope; each thread has its own copy of $i$.

```
#include <stdio.h>

double f(double x);
double Trap(double a, double b, int n, double h);

int main(void) {
    double integral;
    double a, b;
    int n;
    double h;

    printf("Enter a, b, and n\n");
    scanf("%lf", &a);
    scanf("%lf", &b);
    scanf("%d", &n);

    h = (b - a) / n;
    integral = Trap(a, b, n, h);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.15f\n", a, b, integral);

    return 0;
}  /* main */

double Trap(double a, double b, int n, double h) {
    double integral = (f(a) + f(b)) / 2.0;

#pragma omp parallel for num_threads(4) reduction(+:integral)
    for (int k = 1; k <= n - 1; k++) {
        integral += f(a + k * h);
    }
    integral = integral * h;

    return integral;
}  /* Trap */

double f(double x) {
    return x * x;
}  /* f */
```

- Note the following:
  1. OpenMP only parallelize $for$ loop – **no parallelization for $while$ and $do-while$ loops.**
  2. The number of iterations of the $for$ loop **must be determined**
     a. Infinite loops cannot be parallelized

```
for ( ; ; ) {
    . . .
}
```

     b. $for$ loops with $break$ cannot be parallelized

```
for (i = 0; i < n; i++) {
    if ( . . . ) break;
    . . .
}
```

- Notice the error in this code

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

int search(int key, int arr[], int n) {
    int i;
# pragma omp parallel for
    for (i = 0; i < 5; i++) {
        if (key == arr[i])
            return i;
    }
    return -1;

}

int main(int argc, char *argv[]) {
    int n;
    int arr[] = {1, 2, 3, 4, 5};
    int key = 4;

    scanf("%d", &n);
    int res = search(key, arr, n);
    printf("%d\n", res);
    return 0;
}
```

  - The error is at the $return\ i;$ statement.
  - OpenMP cannot exit a loop that is parallelized.

- Try this Fibonacci program **without** and **with** parallelization:

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

int main(int argc, char *argv[]) {
    int fib[20];
    fib[0] = fib[1] = 1;

#pragma omp parallel for
    for (int i = 2; i < 20; ++i) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }

    for (int i = 0; i < 20; ++i) {
        printf("%d ", fib[i]);
    }
    return 0;
}
```

  - When parallelization is used, the **output is unpredictable**.
  - This is because the Fibonacci program includes **dependencies** – each iteration is dependent on the previous iterations. Hence, we cannot compute the value at a specific index without computing the value at lower indices.
- In summary, **OpenMP doesn't check for dependences** among iterations in a loop.

## Estimating $\pi$

- To compute a numerical approximation to $\pi$, we use the following formula

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}.$$

- The serial program

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    double factor = 1.0;
    double sum = 0.0;
    for (int i = 0; i < 1000; ++i) {
        sum += factor / (2 * i + 1);
        factor = -factor;
    }
    double pi = 4 * sum;
    printf("PI = %lf", pi);
    return 0;
}
```

- The parallel program

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    double factor = 1.0;
    double sum = 0.0;
#pragma omp parallel for
    for (int i = 0; i < 1000; ++i) {
        sum += factor / (2 * i + 1);
        factor = -factor;
    }
    double pi = 4 * sum;
    printf("PI = %lf", pi);
    return 0;
}
```

  - This program has a problem: there is a **data dependency** between ***factor*** and ***sum***, thus it gives **incorrect** results.

- We can fix the previous program by replacing

```
sum += factor/(2*k+1);
factor = -factor;
```

  by

```
if (k % 2 == 0)
    factor = 1.0;
else
    factor = -1.0;
sum += factor/(2*k+1);
```

  or

```
factor = (k % 2 == 0) ? 1.0 : -1.0;
sum += factor/(2*k+1);
```

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    double factor = 1.0;
    double sum = 0.0;
#pragma omp parallel for
    for (int i = 0; i < 1000; ++i) {
        factor = (i % 2 == 0) ? 1.0 : -1.0;
        sum += factor / (2 * i + 1);
    }
    double pi = 4 * sum;
    printf("PI = %lf", pi);
    return 0;
}
```

  - **This program gives correct results, but not accurate!**

- The previous program has two issues:
  1. We must tell OpenMP that $sum$ is a **reduction** variable, thus the final result is accumulated into it
  2. The $\boldsymbol{factor}$ variable is **shared** because it's defined before the $parallel\ for$ directive. We must **tell OpenMP that $\boldsymbol{factor}$ is private**.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    double factor = 1.0;
    double sum = 0.0;
#pragma omp parallel for reduction(+:sum) private(factor)
    for (int i = 0; i < 1000; ++i) {
        factor = (i % 2 == 0) ? 1.0 : -1.0;
        sum += factor / (2 * i + 1);
    }
    double pi = 4 * sum;
    printf("PI = %lf", pi);
    return 0;
}
```

- Instead of letting OpenMP to choose the scope of the variables, we can explicitly specify the scope using the $\boldsymbol{default}$ clause

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    double factor = 1.0;
    double sum = 0.0;
    int i;
    int n = 1000;
#pragma omp parallel for default(none) reduction(+:sum)\
        private(i, factor) shared(n)
    for (i = 0; i < n; ++i) {
        factor = (i % 2 == 0) ? 1.0 : -1.0;
        sum += factor / (2 * i + 1);
    }
    double pi = 4 * sum;
    printf("PI = %lf", pi);
    return 0;
}
```

## Sorting

### Bubble sort

- The serial bubble sort algorithm:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int n = 10;
    int a[] = {10, -5, 9, 1, 0, 2, 4, 3, 6, 11};

    for (int i = n-1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (a[j] > a[j + 1]) {
                int tmp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = tmp;
            }}}

    for (int i = 0; i < n; i++) {
        printf("a[%d] = %d\n", i, a[i]);
    }
    return 0;
}
```

  - The outer loop handles all the n-element array
  - The inner loop handles the first n-1 element array
  - The inner loop compares consecutive pairs of elements in the current list. When a pair is out of order ($a[i] > a[i+1]$) it swaps them.
- **The inner loop depends on the outer loop.**
- **The inner loop itself also has dependence on previous iterations**:
  - Say an iteration $j$ will swap the values at index $a[j]$ and $a[j+1]$, this will impact iteration $j+1$.
  - So, iteration $j+1$ cannot decide whether to swap the elements or not, until iteration $j$ finishes.
- We cannot remove the **loop-carried dependence** without completely rewriting the algorithm.

### Odd-even transposition sort

- A sorting algorithm similar to bubble sort but can be parallelized.
- The serial algorithm is as follows

```
for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
    else
        for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

**Table 5.2** Serial odd-even transposition sort.

| Phase | Subscript in Array | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | | 1 | | 2 | | 3 |
| 0 | 9 | ⇔ | 7 | | 8 | ⇔ | 6 |
| | 7 | | 9 | | 6 | | 8 |
| 1 | 7 | | 9 | ⇔ | 6 | | 8 |
| | 7 | | 6 | | 9 | | 8 |
| 2 | 7 | ⇔ | 6 | | 9 | ⇔ | 8 |
| | 6 | | 7 | | 8 | | 9 |
| 3 | 6 | | 7 | ⇔ | 8 | | 9 |
| | 6 | | 7 | | 8 | | 9 |

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int phase, i, tmp;
    int n = 10;
    int a[] = {10, -5, 9, 1, 0, 2, 4, 3, 6, 11};

    for (phase = 0; phase < n; phase++) {
        if (phase % 2 == 0) {
            for (i = 1; i < n; i += 2) {
                if (a[i - 1] > a[i]) {
                    tmp = a[i - 1];
                    a[i - 1] = a[i];
                    a[i] = tmp;
                }}
        } else {
            for (i = 1; i < n - 1; i += 2) {
                if (a[i] > a[i + 1]) {
                    tmp = a[i + 1];
                    a[i + 1] = a[i];
                    a[i] = tmp;
           }}}}
    for (i = 0; i < n; i++) {
        printf("a[%d] = %d\n", i, a[i]);}
    return 0;}
```

- There is a **carried dependence** in the outer loop; two iterations cannot be executed simultaneously.
- The inner loops do not have dependence; each of the inner loops can be executed simultaneously.
    - This because, say the even-phase loop, compares two adjacent elements. So, for two distinct values of $i$, say $i = j$ and $i = k$, the paris $\{j-1, j\}$ and $\{k-1, k\}$ will be disjoint.
    - Hence, the comparison and possible swaps of the pairs $(a[j-1], a[j])$ and $(a[k-1], a[k])$ can proceed simultaneously.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int phase, i, tmp;
    int n = 10;
    int a[] = {10, -5, 9, 1, 0, 2, 4, 3, 6, 11};

    for (phase = 0; phase < n; phase++) {
        if (phase % 2 == 0) {
#pragma omp parallel for num_threads(4) shared(a, n) private(i, tmp)
            for (i = 1; i < n; i += 2) {
                if (a[i - 1] > a[i]) {
                    tmp = a[i - 1];
                    a[i - 1] = a[i];
                    a[i] = tmp;
                }
            }
        } else {
#pragma omp parallel for num_threads(4) shared(a, n) private(i, tmp)
            for (i = 1; i < n - 1; i += 2) {
                if (a[i] > a[i + 1]) {
                    tmp = a[i + 1];
                    a[i + 1] = a[i];
                    a[i] = tmp;
                }
            }
        }
    }
    for (i = 0; i < n; i++) {
        printf("a[%d] = %d\n", i, a[i]);
    }
    return 0;
}
```

- Compare the performance of the bubble sort vs the odd-even sort on an array of 100,000 integers.
    - *Comment either the bubble sort or the odd-even sort block, when testing.*
- You will notice that the parallel sort algorithm is almost 2x faster than the serial one.

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

#define ARRAY_SIZE 100000

void generate_random_array(int *array, int size) {
    // Seed the random number generator
    srand(time(NULL));

    for (int i = 0; i < size; i++) {
        array[i] = rand();
    }
}
```

```c
void bubble_sort(int *array, int size) {
    for (int i = size - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (array[j] > array[j + 1]) {
                // Compare adjacent elements
                int tmp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = tmp;
            }}}}

void odd_even_sort(int *array, int size) {
    int phase, i, tmp;

    for (phase = 0; phase < size; phase++) {
        if (phase % 2 == 0) {
#pragma omp parallel for num_threads(4) shared(array, size) private(i, tmp)
            for (i = 1; i < size; i += 2) {
                if (array[i - 1] > array[i]) {
                    tmp = array[i - 1];
                    array[i - 1] = array[i];
                    array[i] = tmp;
                }
            }
        } else {
#pragma omp parallel for num_threads(4)shared(array, size) private(i, tmp)
            for (i = 1; i < size - 1; i += 2) {
                if (array[i] > array[i + 1]) {
                    tmp = array[i + 1];
                    array[i + 1] = array[i];
                    array[i] = tmp;
                }}}}}

int main(int argc, char *argv[]) {
    int array[ARRAY_SIZE];
    // Generate the random array
    generate_random_array(array, ARRAY_SIZE);

    time_t start, end;

    time(&start);
    bubble_sort(array, ARRAY_SIZE);
    time(&end);
    printf("%f\n", difftime(end, start));

    time(&start);
    odd_even_sort(array, ARRAY_SIZE);
    time(&end);
    printf("%f\n", difftime(end, start));

    return 0;
}
```

- **The previous parallel odd-even sort has one issue: $omp$ forks the threads at the beginning of the first $for$ loop and joins them at the end. The team of threads is forked at the beginning of the second $for$ loop and joined again at the end.**
- This fork-join process costs time.
- We can solve this issue by forking the threads once at the beginning of the outer loop and only join them at the end of the second inner loop.

```
#define ARRAY_SIZE 200000

void enhanced_odd_even_sort(int *array, int size) {
    int phase, i, tmp;
#pragma omp parallel num_threads(4) shared(array, size)
private(i, tmp, phase)
    for (phase = 0; phase < size; phase++) {
        if (phase % 2 == 0) {
#pragma omp for
            for (i = 1; i < size; i += 2) {
                if (array[i - 1] > array[i]) {
                    tmp = array[i - 1];
                    array[i - 1] = array[i];
                    array[i] = tmp;
                }
            }
        } else {
#pragma omp for
            for (i = 1; i < size - 1; i += 2) {
                if (array[i] > array[i + 1]) {
                    tmp = array[i + 1];
                    array[i + 1] = array[i];
                    array[i] = tmp;
                }
            }
        }
    }
}
```

- Run this enhanced odd-even sort in comparison with the basic odd-sort function on an array of size 200,000.
- You will notice that this new function is slightly faster than the basic function.
- In the enhanced function:
  1. We defined the outer loop as a parallel structure, but without using $parallel\ for$ directive. So, this will fork the threads.
  2. At the beginning of each of the inner loops, we used the $for$ directive. This will let the loops to execute in parallel using the forked threads, without forking another team of threads.