Parallel Programing

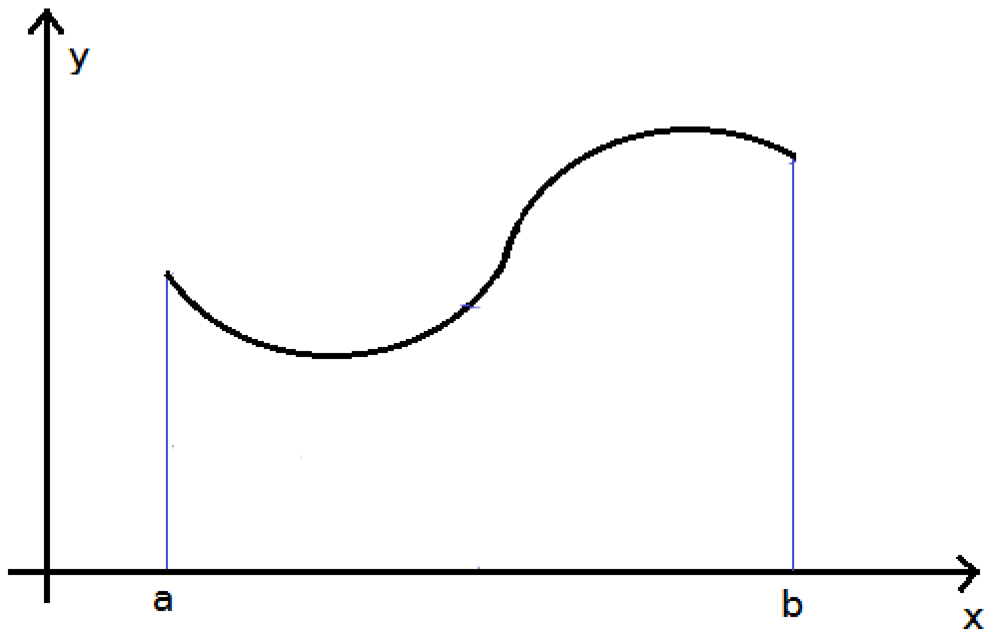
Trapeze method with MPI

Henrique Weber
João Luiz Grave Gross

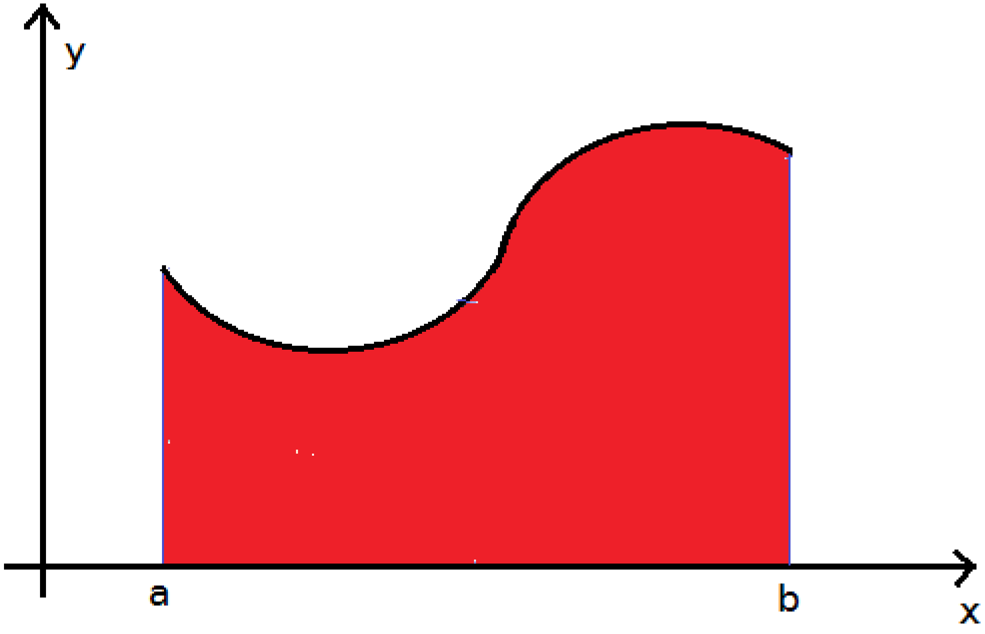
Professor Nicolas Mailard

December 3rd, 2010.

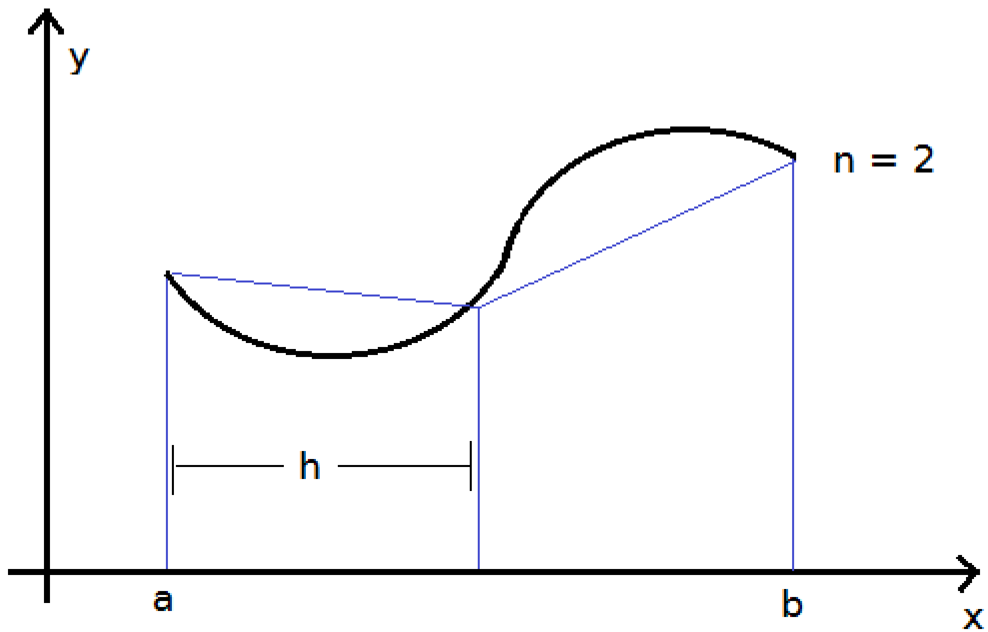
Area under a function



Area under a function



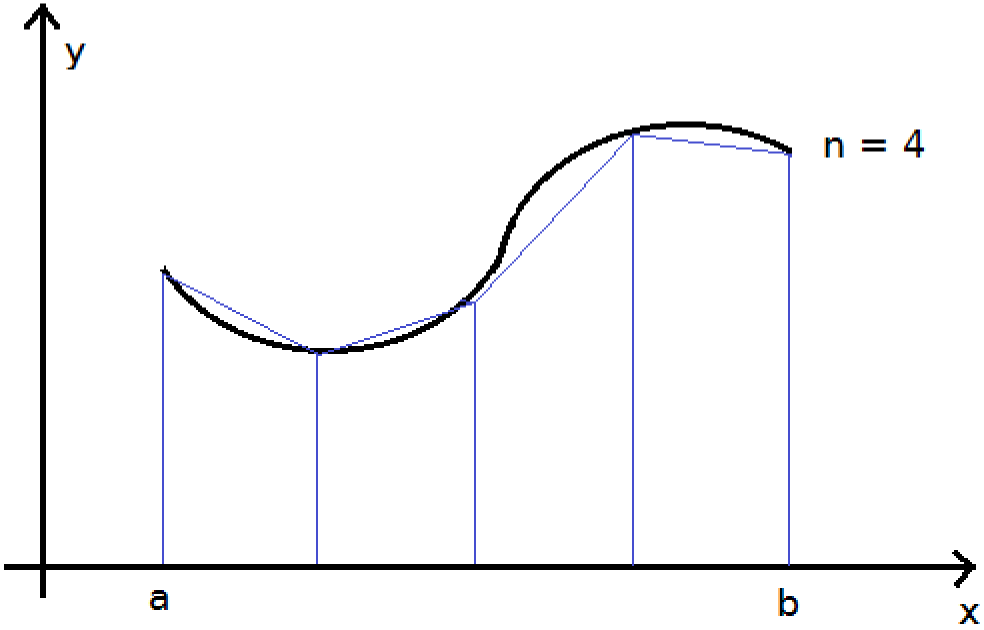
Trapeze method



$$A = \sum_{0}^{n-1} A_i$$

f(x) = ([(b – a) / n] / 2) * (f0 + fn + sum fj, from j=1 to n-1)

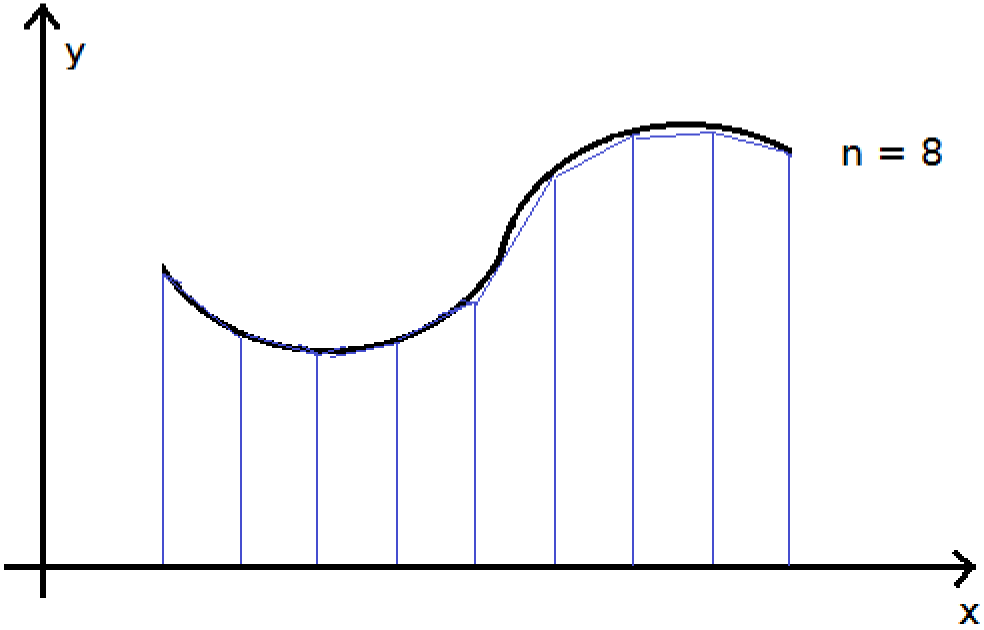
Trapeze method



$$A = \sum_{0}^{n-1} A_i$$

f(x) = ([(b – a) / n] / 2) * (f0 + fn + sum fj, from j=1 to n-1)

Trapeze method



$$A = \sum_{0}^{n-1} A_i$$

f(x) = ([(b – a) / n] / 2) * (f0 + fn + sum fj, from j=1 to n-1)

Sequential Program

Functions

```
int main ()
{
      /* Variable Declaration */

      /* Time Counting Begin */
      CalculateInteger( interval, polynomial, &subintervals, &integer, &polynomial_grade );
      /* Time Counting End */

      PrintResult( interval, polynomial, &subintervals, &integer, &t_total, &polynomial_grade );

      return 0;
}
```

Simplified main function of the program.

Sequential Program

Functions

1) **double** * CaptureEntries(**double** *interval, **long int** *subintervals, **long int** *polynomial_grade)

2) **void** CalculateInteger(**double** *interval, **double** *polynomial, **long int** *subintervals, **double** *integer, **long int** *polynomial_grade)

3) **double** CalculatePolynomial(**double** *polynomial, **double** *xi, *polynomial_grade)

4) **void** PrintResult(**double** *interval, **double** *polynomial, **long int** *subintervals, **double** *integer, **double** *t_total, **long int** *polynomial_grade)

Sequential Program

Functions

1) **double** * CaptureEntries(**double** *interval, **long int** *subintervals, **long int** *polynomial_grade)

This function just capture all the entries needed for the execution of the trapeze method.

Sequential Program

Functions

```
2) void CalculateInteger( double *interval, double *polynomial, long int *subintervals,
double *integer, long int *polynomial_grade )

{
    double interval_h, interval_aux;
    long int i;

    interval_h = (interval[1] - interval[0]) / *subintervals;
    *integer = *integer + CalculatePolynomial( polynomial, &interval[0], polynomial_grade );
    *integer = *integer + CalculatePolynomial( polynomial, &interval[1], polynomial_grade );

    for(i = 1; i < *subintervals; i++){
        interval_aux = interval[0] + i*interval_h;
        *integer = *integer + 2*CalculatePolynomial( polynomial, &interval_aux,
polynomial_grade );
    }
    *integer = *integer * interval_h / 2;
    return;
}
```

Sequential Program

Functions

```
3) double CalculatePolynomial( double *polynomial, double *xi, long int
*polynomial_grade )

{
    double trapeze = 0;
    int i;

    for(i = 0; i <= *polynomial_grade; i++)
        trapeze = trapeze + polynomial[i]*pow(*xi,i);

    return trapeze;
}
```

Sequential Program

Functions

4) **void** PrintResult(**double** *interval, **double** *polynomial, **long int** *subintervals, **double** *integer, **double** *t_total, **long int** *polynomial_grade)

Just prints the result of the program.

Sequential Program

Time Measurement

```
...
#include <sys/time.h>

int main()
{
        …
        struct timeval tv_begin, tv_end;
        double t_begin, t_end, t_total;

        …
        gettimeofday(&tv_begin, NULL);
        t_begin = (tv_begin.tv_sec)*1000000 + tv_begin.tv_usec;
        CalculateInteger( interval, polynomial, &subintervals, &integer, &polynomial_grade );
        gettimeofday(&tv_end, NULL);
        t_end = (tv_end.tv_sec)*1000000 + tv_end.tv_usec;

        t_total = t_end - t_begin; //The value is in micro seconds
        …
}
```

Sequential Program

Execution Enviroment: two quad-core Intel Xeon E5530 (Nehalem arch)
64KB L1 cache, 12MB L2 cache, 2GHz

How to compile: gcc trapeze_sequential.c -o trapeze_sequential
How to run: ./trapeze_sequential

Input: Polynomial: $0.625x^4 - 4x^3 + 2x + 1$
Interval: [0,8]
Subintervals: 10^7

Output: 72,00000

Average Time = 2.841356 s

Parallel Program

Differences from the sequential version

1) More than one process in the program execution

2) One root process

3) n slave processes

4) Task division

5) MPI functions

Similarities from the sequential version

1) Same algorithm

2) Same functions (just adapted)

3) Just one program written (just adapted for parallel uses)

Parallel Program

Functions

```
...
#include <mpi.h>

int main ()
{
    /* Variable declaration */
    /* MPI initializing */

    if (rank == 0){ /* Root process */
        /* Capture entries */

        /* Time counting begin */
        /* Divide tasks */
        /* Receive and sum results */
        /* Time counting end */
    }
    ...
```

Parallel Program

Functions

```
    …
    else{ /* Slave process */
        /* Receive information */
        /* Compute part of the problem */
        /* Send results to root */
    }

    /* MPI ending */
    return 0;
}
```

Parallel Program

Functions

1) **double** * CaptureEntries(**double** *interval, **long int** *subintervals, **long int** *polynomial_grade)

2) **void** CalculateInteger(**int** rank, **int** numtasks, **double** interval_h, **double** *interval, **double** *polynomial, **long int** *subintervals, **double** *parcial_integer, **long int** *polynomial_grade, **long int** init_subint, **long int** end_subint)

3) **double** CalculatePolynomial(**double** *polynomial, **double** *xi, *polynomial_grade)

4) **void** PrintResult(**double** *interval, **double** *polynomial, **long int** *subintervals, **double** *integer, **double** *t_total, **long int** *polynomial_grade)

All the 4 functions works just like the functions from the sequential code.

Parallel Program

Functions

```
2) void CalculateInteger(int rank, int numtasks, double interval_h, double *interval,
double *polynomial, long int *subintervals, double *parcial_integer, long int  *polynomial_grade,
long int  init_subint, long int end_subint )

{
        double interval_aux;
        long int i;
        *parcial_integer = 0;

        for(i = init_subint; i < end_subint; i++){
                interval_aux = interval[0] + i*interval_h;
                *parcial_integer = *parcial_integer + 2*CalculatePolynomial( polynomial,
&interval_aux, polynomial_grade );
        }

        return;
}
```

Parallel Program

Time Measurement

The same as in the sequential code.

Implemented in the root process.

Parallel Program

Data transfer through processes

```
int main ()
{
    ...
    if (rank == 0){ /* Root process */
        …
        /* Time counting begin */

        MPI_Bcast( &polynomial_grade, 1, MPI_LONG, root, MPI_COMM_WORLD );
        /* Broascast other variables */

        …

        for(i = 1; i < numtasks; i++){
            MPI_Send(&init_subint, 1, MPI_DOUBLE, i, tag, MPI_COMM_WORLD);
            MPI_Send(&end_subint, 1, MPI_DOUBLE, i, tag, MPI_COMM_WORLD);
            ...
        }

        ...
```

Parallel Program

Data transfer through processes

```
        …
        for(i = 1; i < numtasks; i++){
                ...
                MPI_Recv(&parcial_integer, 1, MPI_DOUBLE, i, tag, MPI_COMM_WORLD, &Stat);
                integer += parcial_integer;
        }

        /* Calculate final integer */
        /* Finish time counting */
    }

    ...
```

Parallel Program

Data transfer through processes

```
else{ /* Slave process */
      MPI_Bcast( &polynomial_grade, 1, MPI_LONG, root, MPI_COMM_WORLD );
      /* Receive other data via broadcast */
      ...

      MPI_Recv(&init_subint, 1, MPI_DOUBLE, root, tag, MPI_COMM_WORLD, &Stat);

      MPI_Recv(&end_subint, 1, MPI_DOUBLE, root, tag, MPI_COMM_WORLD, &Stat);

      CalculateInteger( rank, numtasks, interval_h, interval, polynomial, &subintervals,
&parcial_integer, &polynomial_grade, init_subint, end_subint );

      MPI_Send(&parcial_integer, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
}

MPI_Finalize();

return 0;
}
```

Parallel Program

How to compile: mpicc trapeze_parallel.c -o trapeze_parallel

How to run: mpirun -np 2 ./trapeze_parallel (here the number 2 is just an example, but it can be higher than 2)

Input: Polynomial: $0.625x^4 - 4x^3 + 2x + 1$

Interval: [0,8]

Subintervals: 10^7

Output: 72,00000

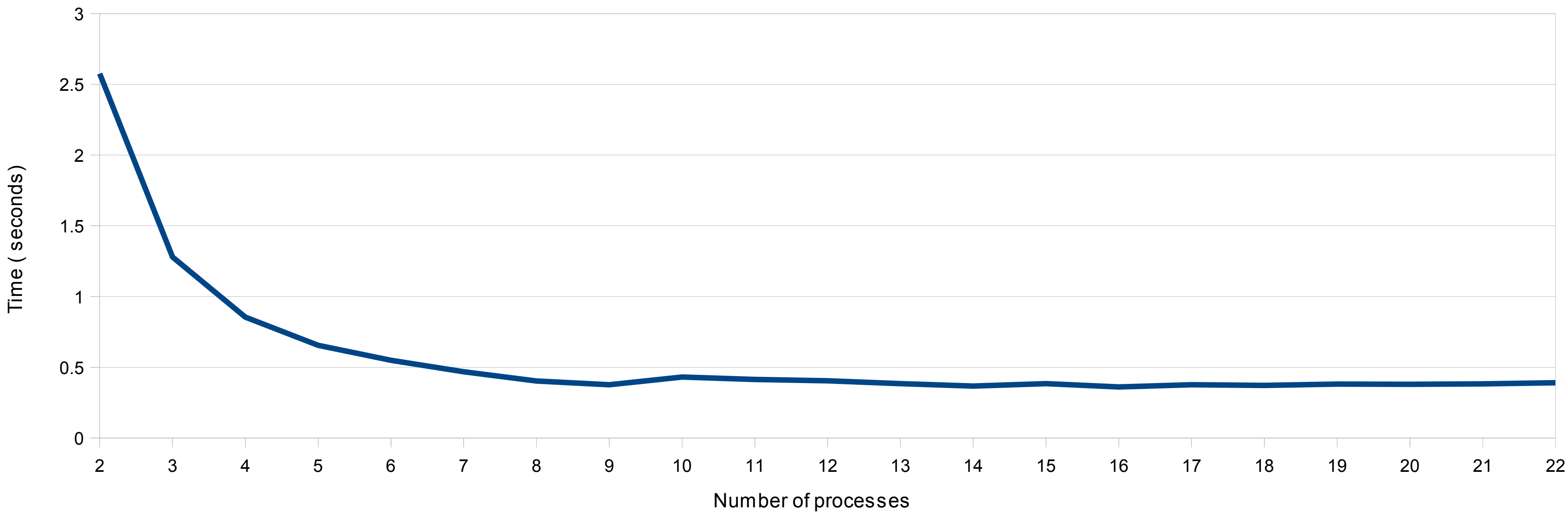
Number of processes	Average time (sec)	Number of processes	Average time (sec)
2	2.58	13	0.38
3	1.28	14	0.37
4	0.85	15	0.38
5	0.65	16	0.36
6	0.55	17	0.38
7	0.47	18	0.37
8	0.4	19	0.38
9	0.38	20	0.38
10	0.43	21	0.38
11	0.41	22	0.39
12	0.4		

Parallel Program

How to compile: mpicc trapeze_parallel.c -o trapeze_parallel
How to run: mpirun -np 2 ./trapeze_parallel (here the number 2 is just an example, but it can be higher than 2)

Input: Polynomial: $0.625x^4 - 4x^3 + 2x + 1$
Interval: [0,8]
Subintervals: 10^7

Output: 72,00000



Sequential Program x Parallel algorithm

Sequential Algorithm	Parallel Algorithm	Gain
2.841356 sec	Worst case (2.58 sec) – 2 processes	9,19%
	Medium case (0.55 sec)	516%
	Better case (0.36 sec) – 16 processes	780%

Conclusion

- Parallel applications can provide a huge gain in performance.

- Problems can be fully or parcially parallelized, we just have to SEE and THINK how this can be done.

- Parallel programming is growing fast so is important for us to master it.

- Its even more important due to the increase of cores inside processors, in other words, parallel application will be seen more frequently.

Thanks!

Questions?!

Henrique Weber - hweber@inf.ufrgs.br
João Gross - jlggross@inf.ufrgs.br