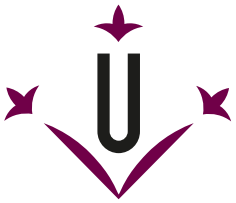
Universitat de Lleida
Màster en Enginyeria Informàtica
High Performance Computing
Prof. Francesc Gine De Sola
Prof. Jordi Ricard Onrubia Palacios



HPC Project

Heat Diffussion Equation

MPI + OMP

Jordi García Ventura
Christian López García

May 18, 2025

Contents

List of codes

List of figures

List of tables

1 Introduction

The purpose of this assignment is to make a hybrid implementation of the *Heat Diffusion* using OpenMP (OMP) and Message Passing Interface (MPI)[1] from the previous OMP implementation.

With the nature of these APIs, the program will be able to use the full capacity of the Moore cluster, doing so by delegating the execution in each node to OMP and the distribution of the work and communications to MPI. As we already had the OMP implementation from last submission, we will comment only the changes that we did to integrate MPI's communications.

First of all, as mentioned in the handed PDF, we decided to implement a **static mapping of tasks** that divides the virtual grid into rows and each node in MPI calculates one and communicates with its neighbors.

2 Code implementation

In this section we will comment the most important changes in the code that we made to make the OMP+MPI hybrid implementation, we will start function by function after commenting the MPI initialization comment.

2.1 Initialization

First, we will comment how the MPI has been initialized in the main function as in *Code 2*. This process involves the creation of two local variables to store the results of the initialization *rank* and *size*, representing a unique identifier and the number of processes respectively, which later will be used for the work split.

Then, we divide the number of rows in the grid by the number of processes, and gather the remaining ones (if any), having as result the 'total amount of work' that this process has to do in the *local_nx* variable, which represents the number of rows that the process has to do.

Lastly, we changed the initialization of the grids by adding two extra 'ghost' rows for the communication between processes, this is reflected in the memory allocation call, which has been updated to make use of the 'total amount of work' of the process instead of the full grid and adding these two extra rows.

2.1.1 Function: *initialize_grid*

Once MPI and everything is initialized, we have to do the 'starter' diagonal heat, to candle the light of the program, and be able to make the calculations. We don't expect much improvement in the overall execution time of the program by the optimization of this function since its only ran once, so we will go over it quickly. As you can see in the *Code 3*, we are now using the *rank* and *size* variables to know the offset that the process has to calculate and the *local_nx* to know how many rows each process has to compute, this will settle an example for how we will use it for the heat calculation. The rest of the function remains practically unchanged.

2.2 Heat calculations

Here comes the big deal. As we can see in *Code 4*, for every iteration every processes has to communicate to its neighbours the results that they may need to use in their calculations and to receive the same from them, this means, communicating these 'ghost' rows that are adjacent, having in mind the out-of-bounds. Once the processes have communicated, the formula is applied the same way as in the last assignment, just that the application of the Dirichlet boundaries is only done in the top and last row of the grid.

2.3 Gathering and finalization

Once the calculations are done, we communicate back the resulting heat grid as in *Code 5*, allocating a new grid with the full size and putting there the results of the calculations of the different processes.

2.3.1 Function: *print_grid*

Completely deleted. There was no need and it was slowing down the execution.

2.3.2 Function: *write_grid*

This function remained the same way since only one process had to do the parallelization using OMP, as commented in last assignment.

3 Results

Before commenting the execution times, we will point out the problem domain:

- Grid size: 100, 1000, 2000.

- Iterations: 100, 1000, 10000, 100000.
- MPI workers: 1, 2, 4, 8, 16, 32.
- OMP threads: 4.

Since we want to know the scalability of our problem with MPI, we have locked the number of OMP threads to 4, because in the last assignment it had some of the best results, and we will be able to compare our baseline (4/1 threads/worker) with the rest of the results.

We will do a comparison of the raw execution times between the hybrid implementations and its serial counterparts in *Figure 1*.

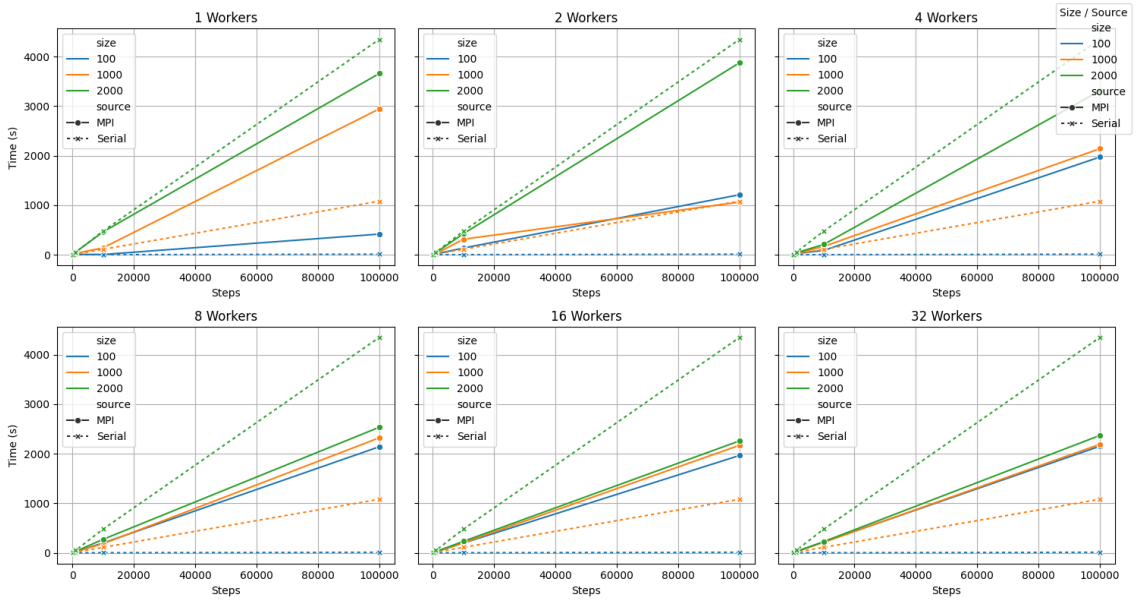


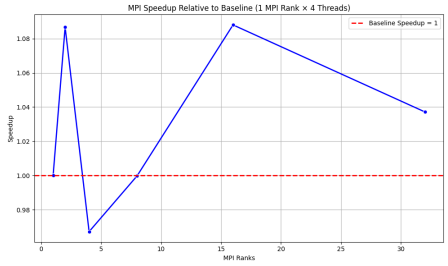
Figure 1: Time results of the executions

With these times and the following formulas, we can calculate the speedups in *Figure 2a* and *Figure 2b*, by comparing the execution time of the MPI baseline, and with N workers as in *Formula 1*.

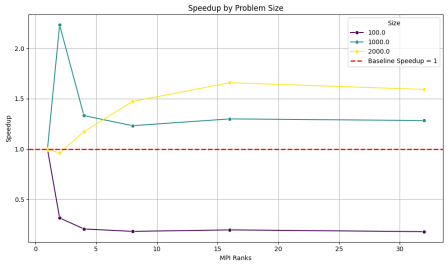
$$Speedup = \frac{T_{1x4}}{T_{N_{MPI\_ranks}}x4} \tag{1}$$

With speedups calculated, we can also calculate the efficiency and the overhead. To do so, we will use *Formula 2*, *Formula 3* and *Formula 4*. Begin T_{ideal} the baseline time divided by the number of cores.

$$Total\ cores = N_{MPI\_ranks}x4 \tag{2}$$



(a) Speedup compared with baseline

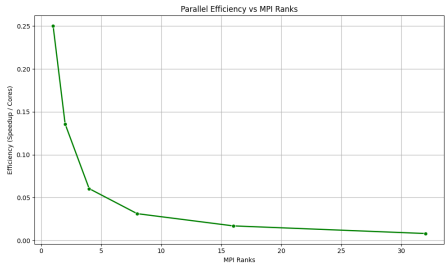


(b) Speedup compared with baseline by problem size

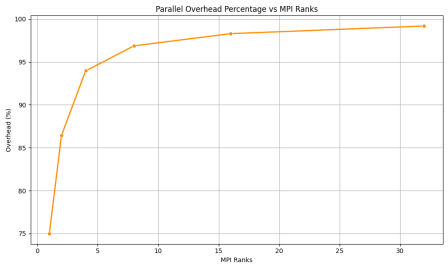
$$Effiency = \frac{Speedup}{Total\ cores} \tag{3}$$

$$Overhead(percentage) = (1 - \frac{T_{ideal}}{T_P})x100 \tag{4}$$

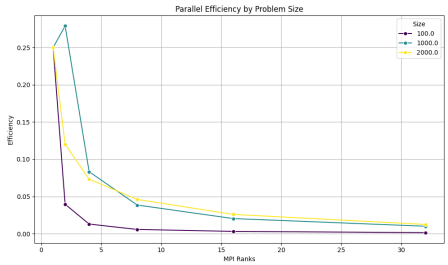
And having as result these metrics in *Figure 3a* and *Figure 3b*, and by problem size *Figure 4a* and *Figure 4b*.



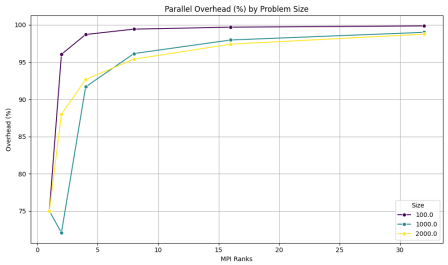
(a) Efficiency of the number of nodes



(b) Overhead of the communications



(a) Efficiency of the number of nodes by problem size



(b) Overhead of the communications by problem size

4 Conclusions

First of all we will discuss about imbalance, since its one of the assignment points. We split the work in a way that every node has *N* or *N+1* number of rows which, in our opinion is decent, although some rows may contain more workload than others and lead to imbalance either way. To quantify that, we could gather information of the runtime of every node

and gather it at the end, but we did not implement that. Either way, we can think there is imbalance since the nodes have to communicate every iteration, they must wait for each other. Because execution time remains nearly constant when using more than 4 workers, we can infer that the bottleneck is not computation but synchronization, nodes are likely waiting for others to complete before proceeding.

There is a weird pattern in speedup, worsening at 4 nodes but improving in the rest of workers number, although that we can see that it's due to the rapid decrease in size (100x100). In the other hand, we can see a rapid improvement in size (1000x1000) that steadies around *1.3*, while in larger sizes is slower and steadies around *1.6*.

From these results, we can already see that the scalability is not that great, at least not with our implementation, we can further see that with the following conclusions.

Efficiency drops rapidly as we increase the number of MPI ranks, it's likely that each node is not doing enough computation to justify the communication overhead. This is typical when the problem size is too small relative to the number of workers. We can confirm this by looking at the overhead percentage, which approaches 100% as the number of workers increases

In conclusion, we see that the resources are not used well and its more pronounced when the problem size is smaller.

5 Extra work

Just as in last submission, we kept the extra work we did and now we are just going to make an overview in the following *Code 1*.

Code 1 Some examples of Makefile commands

```
$ make all      # Compiles the different C files and runs the MPI program
$ make test     # Compares the '.bmp' files in the results directory
$ make time     # Extracts the execution times from the log in the output files
$ make clean    # Removes all resulting files
#... There are some others but they are irrelevant for the purpose of this assignment
```

5.1 Testing

As we talked about in the last assignment, we did some automatic testing that compared the resulting BMP files of the serial version with the rest (depending on size and iterations, obviously). In the last submit, it was not that needed since the solution was just adding a

few lines of code, but in this one, which was harder, really shined since we had to do some iterations and it was easier to track some errors.

Acronyms

MPI Message Passing Interface. , 1–4

OMP OpenMP. 1–3

PDF Portable Document File. 1

Bibliography

[1] *Open MPI v5.0.x — Open MPI 5.0.x documentation*. [Online; accessed 17. May 2025]. May 2025. URL: *https://docs.open-mpi.org/en/v5.0.x*.

A Code changes

Code 2 MPI initialization

```
int main(int argc, char *argv[]) {
    // ... Other initializations
    double start_time, end_time;
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc != 4)
    {
        printf("Usage: heat_mpi <size> <steps> <name_output_file>\n");
        MPI_Finalize();
        return 1;
    }

    if (rank == 0) start_time = MPI_Wtime();

    // Local size of the process
    int base_rows = nx / size;
    int remainder = nx % size;
    int local_nx = base_rows + (rank < remainder ? 1 : 0);

    // Allocate grid with 2 extra rows for ghost rows
    double *grid = (double *)calloc((local_nx + 2) * ny, sizeof(double));
    double *new_grid = (double *)calloc((local_nx + 2) * ny, sizeof(double));
    // ... Heat solving
}
```

Code 3 Heat grid initialization

```
void initialize_grid(double *grid, int nx, int ny, int rank, int size, int local_nx)
{
    int base_rows = nx / size;
    int extra = nx % size;
    int row_offset = rank * base_rows + (rank < extra ? rank : extra);
    int i, j, global_i;
    double value;

#pragma omp parallel for collapse(2) private(i, j, global_i, value)
    for (i = 1; i <= local_nx; i++)
    {
        for (j = 0; j < ny; j++)
        {
            global_i = row_offset + i - 1;
            value = 0.0;

            if (global_i != 0 && global_i != nx - 1 && j != 0 && j != ny - 1)
            {
                if (global_i == j || global_i == nx - 1 - j)
                {
                    value = T;
                }
            }

            grid[i * ny + j] = value;
        }
    }
}
```

Code 4 Heat calculations

```
void solve_heat_equation(double *grid, double *new_grid, int steps, double r, int ny,
↪  int local_nx, int rank, int size)
{
    double *temp;
    int i, j, step, global_i;
    int is_first = (rank == 0);
    int is_last = (rank == size - 1);
    int idx;

    for (step = 1; step < steps; step++)
    {
        // Exchange ghost rows
        if (!is_first)
        {
            MPI_Sendrecv(&grid[1 * ny], ny, MPI_DOUBLE, rank - 1, 0,
                         &grid[0 * ny], ny, MPI_DOUBLE, rank - 1, 1,
                         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        if (!is_last)
        {
            MPI_Sendrecv(&grid[local_nx * ny], ny, MPI_DOUBLE, rank + 1, 1,
                         &grid[(local_nx + 1) * ny], ny, MPI_DOUBLE, rank + 1, 0,
                         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

#pragma omp parallel for private(i, j) collapse(2)
        for (i = 1; i <= local_nx; i++)
        {
            for (j = 1; j < ny - 1; j++)
            {
                idx = i * ny + j;
                new_grid[idx] = grid[idx]
                                + r * (grid[idx + ny] + grid[idx - ny] - 2 *
                                 ↪  grid[idx])
                                + r * (grid[idx + 1] + grid[idx - 1] - 2 * grid[idx]);
            }
        }

        temp = grid;
        grid = new_grid;
        new_grid = temp;
    }
}
```

Code 5 MPI gathering

```
int main(int argc, char *argv[]) {
    // ... Heat solving

    double *full_grid = NULL;
    int *recvcounts = NULL, *displs = NULL;

    if (rank == 0)
    {
        full_grid = malloc(nx * ny * sizeof(double));
        recvcounts = malloc(size * sizeof(int));
        displs = malloc(size * sizeof(int));

        int offset = 0;
        int i;
        for (i = 0; i < size; i++)
        {
            int rows = base_rows + (i < remainder ? 1 : 0);
            recvcounts[i] = rows * ny;
            displs[i] = offset;
            offset += rows * ny;
        }
    }

    MPI_Gatherv(&grid[1 * ny], local_nx * ny, MPI_DOUBLE,
                full_grid, recvcounts, displs, MPI_DOUBLE,
                0, MPI_COMM_WORLD);
    // ...
}
```

Code 6 BMP file generation call and program finalization

```
int main(int argc, char *argv[]) {
    // ... MPI gathering
    if (rank == 0)
    {
        FILE *file = fopen(argv[3], "wb");
        if (!file)
        {
            fprintf(stderr, "Failed to open output file\n");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        write_bmp_header(file, nx, ny);
        write_grid(file, full_grid, nx, ny);
        fclose(file);

        end_time = MPI_Wtime();
        printf("Execution Time = %fs for %dx%d grid and %d steps\n", end_time -
         ↪  start_time, nx, ny, steps);

        free(full_grid);
        free(recvcounts);
        free(displs);
    }

    // Free allocated memory
    free(grid);
    free(new_grid);
    MPI_Finalize();

    return 0;
}
```

B Data

Table 1: Raw data

program	threads	steps	size	time(s)
mpi	*1*	*100000*	*1000*	*2943.4*

Continued on next page

Table 1: Raw data (Continued)

program	threads	steps	size	time(s)
mpi	*1*	*100000*	*100*	*416.4*
mpi	*1*	*100000*	*2000*	*3657.3*
mpi	*1*	*10000*	*1000*	*139.5*
mpi	*1*	*10000*	*100*	*4.4*
mpi	*1*	*10000*	*2000*	*461.1*
mpi	*1*	*1000*	*1000*	*25.5*
mpi	*1*	*1000*	*100*	*7.5*
mpi	*1*	*1000*	*2000*	*50.5*
mpi	*1*	*100*	*1000*	*3.9*
mpi	*1*	*100*	*100*	*2.8*
mpi	*1*	*100*	*2000*	*5*
mpi	*16*	*100000*	*1000*	*2169.7*
mpi	*16*	*100000*	*100*	*1964.1*
mpi	*16*	*100000*	*2000*	*2258.8*
mpi	*16*	*10000*	*1000*	*202.5*
mpi	*16*	*10000*	*100*	*196.6*
mpi	*16*	*10000*	*2000*	*231*
mpi	*16*	*1000*	*1000*	*20.2*
mpi	*16*	*1000*	*100*	*19.7*
mpi	*16*	*1000*	*2000*	*23.2*
mpi	*16*	*100*	*1000*	*2.5*
mpi	*16*	*100*	*100*	*1.9*
mpi	*16*	*100*	*2000*	*3.2*
mpi	*2*	*100000*	*1000*	*1061.1*
mpi	*2*	*100000*	*100*	*1210.9*
mpi	*2*	*100000*	*2000*	*3875*

Continued on next page

Table 1: Raw data (Continued)

program	threads	steps	size	time(s)
mpi	*2*	*10000*	*1000*	*313.1*
mpi	*2*	*10000*	*100*	*135*
mpi	*2*	*10000*	*2000*	*424.8*
mpi	*2*	*1000*	*1000*	*18*
mpi	*2*	*1000*	*100*	*14.5*
mpi	*2*	*1000*	*2000*	*41.3*
mpi	*2*	*100*	*1000*	*1.6*
mpi	*2*	*100*	*100*	*1.7*
mpi	*2*	*100*	*2000*	*4.8*
mpi	*32*	*100000*	*1000*	*2189.3*
mpi	*32*	*100000*	*100*	*2151.2*
mpi	*32*	*100000*	*2000*	*2368.7*
mpi	*32*	*10000*	*1000*	*211.3*
mpi	*32*	*10000*	*100*	*218.6*
mpi	*32*	*10000*	*2000*	*225.6*
mpi	*32*	*1000*	*1000*	*22.2*
mpi	*32*	*1000*	*100*	*22.2*
mpi	*32*	*1000*	*2000*	*23.6*
mpi	*32*	*100*	*1000*	*2.4*
mpi	*32*	*100*	*100*	*2.3*
mpi	*32*	*100*	*2000*	*3.6*
mpi	*4*	*100000*	*1000*	*2139.5*
mpi	*4*	*100000*	*100*	*1971*
mpi	*4*	*100000*	*2000*	*3308.7*
mpi	*4*	*10000*	*1000*	*166.9*
mpi	*4*	*10000*	*100*	*86.4*

Continued on next page

Table 1: Raw data (Continued)

program	threads	steps	size	time(s)
mpi	*4*	*10000*	*2000*	*208*
mpi	*4*	*1000*	*1000*	*27*
mpi	*4*	*1000*	*100*	*23.8*
mpi	*4*	*1000*	*2000*	*38.5*
mpi	*4*	*100*	*1000*	*2.7*
mpi	*4*	*100*	*100*	*1.9*
mpi	*4*	*100*	*2000*	*4.2*
mpi	*8*	*100000*	*1000*	*2319.5*
mpi	*8*	*100000*	*100*	*2139.6*
mpi	*8*	*100000*	*2000*	*2533.6*
mpi	*8*	*10000*	*1000*	*182.7*
mpi	*8*	*10000*	*100*	*197.8*
mpi	*8*	*10000*	*2000*	*271.6*
mpi	*8*	*1000*	*1000*	*21.4*
mpi	*8*	*1000*	*100*	*19.7*
mpi	*8*	*1000*	*2000*	*26.2*
mpi	*8*	*100*	*1000*	*2.1*
mpi	*8*	*100*	*100*	*1.9*
mpi	*8*	*100*	*2000*	*2.8*
serial	*1*	*100000*	*1000*	*1100.4*
serial	*1*	*100000*	*100*	*10.4*
serial	*1*	*100000*	*2000*	*4306.3*
serial	*1*	*10000*	*1000*	*109.2*
serial	*1*	*10000*	*100*	*1*
serial	*1*	*10000*	*2000*	*475.3*
serial	*1*	*1000*	*1000*	*12.1*

Continued on next page

Table 1: Raw data (Continued)

program	threads	steps	size	time(s)
serial	*1*	*1000*	*100*	*0.1*
serial	*1*	*1000*	*2000*	*47.6*
serial	*1*	*100*	*1000*	*1.1*
serial	*1*	*100*	*100*	*0*
serial	*1*	*100*	*2000*	*4.5*
serial	*16*	*100000*	*1000*	*1070.2*
serial	*16*	*100000*	*100*	*10.1*
serial	*16*	*100000*	*2000*	*4305.9*
serial	*16*	*10000*	*1000*	*109.2*
serial	*16*	*10000*	*100*	*1*
serial	*16*	*10000*	*2000*	*475.4*
serial	*16*	*1000*	*1000*	*12.1*
serial	*16*	*1000*	*100*	*0.1*
serial	*16*	*1000*	*2000*	*47.6*
serial	*16*	*100*	*1000*	*1.1*
serial	*16*	*100*	*100*	*0*
serial	*16*	*100*	*2000*	*4.5*
serial	*2*	*100000*	*1000*	*1098.8*
serial	*2*	*100000*	*100*	*10.4*
serial	*2*	*100000*	*2000*	*4307.5*
serial	*2*	*10000*	*1000*	*109.2*
serial	*2*	*10000*	*100*	*1*
serial	*2*	*10000*	*2000*	*475.6*
serial	*2*	*1000*	*1000*	*12.4*
serial	*2*	*1000*	*100*	*0.1*
serial	*2*	*1000*	*2000*	*48.9*

Continued on next page

Table 1: Raw data (Continued)

program	threads	steps	size	time(s)
serial	*2*	*100*	*1000*	*1.1*
serial	*2*	*100*	*100*	*0*
serial	*2*	*100*	*2000*	*4.6*
serial	*32*	*100000*	*1000*	*1070*
serial	*32*	*100000*	*100*	*10.1*
serial	*32*	*100000*	*2000*	*4305.1*
serial	*32*	*10000*	*1000*	*109.4*
serial	*32*	*10000*	*100*	*1*
serial	*32*	*10000*	*2000*	*475.7*
serial	*32*	*1000*	*1000*	*12.1*
serial	*32*	*1000*	*100*	*0.1*
serial	*32*	*1000*	*2000*	*47.6*
serial	*32*	*100*	*1000*	*1.1*
serial	*32*	*100*	*100*	*0*
serial	*32*	*100*	*2000*	*4.5*
serial	*4*	*100000*	*1000*	*1069.9*
serial	*4*	*100000*	*100*	*10.1*
serial	*4*	*100000*	*2000*	*4427.6*
serial	*4*	*10000*	*1000*	*109.3*
serial	*4*	*10000*	*100*	*1*
serial	*4*	*10000*	*2000*	*488.8*
serial	*4*	*1000*	*1000*	*12.1*
serial	*4*	*1000*	*100*	*0.1*
serial	*4*	*1000*	*2000*	*47.6*
serial	*4*	*100*	*1000*	*1.1*
serial	*4*	*100*	*100*	*0*

Continued on next page

Table 1: Raw data (Continued)

program	threads	steps	size	time(s)
serial	*4*	*100*	*2000*	*4.5*
serial	*8*	*100000*	*1000*	*1072.7*
serial	*8*	*100000*	*100*	*10.1*
serial	*8*	*100000*	*2000*	*4427.2*
serial	*8*	*10000*	*1000*	*111.9*
serial	*8*	*10000*	*100*	*1*
serial	*8*	*10000*	*2000*	*475.4*
serial	*8*	*1000*	*1000*	*12.4*
serial	*8*	*1000*	*100*	*0.1*
serial	*8*	*1000*	*2000*	*48.9*
serial	*8*	*100*	*1000*	*1.1*
serial	*8*	*100*	*100*	*0*
serial	*8*	*100*	*2000*	*4.6*

Table 2: Calculated metrics

size	ranks	cores	mean time (s)	speedup	efficiency	Overhead	
						s	%
100	*1*	*4*	*107.8*	*1*	*0.25*	*80.8*	*74.9*
100	*2*	*8*	*340.5*	*0.3*	*0.03*	*327*	*96*
100	*4*	*16*	*520.8*	*0.2*	*0.01*	*514*	*98.7*
100	*8*	*32*	*589.7*	*0.1*	*0*	*586.4*	*99.4*
100	*16*	*64*	*545.6*	*0.1*	*0*	*543.9*	*99.6*
100	*32*	*128*	*598.6*	*0.1*	*0*	*597.7*	*99.8*
1000	*1*	*4*	*778.1*	*1*	*0.25*	*583.6*	*75*
1000	*2*	*8*	*348.5*	*2.2*	*0.27*	*251.2*	*72*

Continued on next page

Table 2: Calculated metrics (Continued)

size	ranks	cores	mean time (s)	speedup	efficiency	Overhead	
						s	%
1000	*4*	*16*	*584*	*1.3*	*0.08*	*535.4*	*91.6*
1000	*8*	*32*	*631.4*	*1.2*	*0.03*	*607.1*	*96.1*
1000	*16*	*64*	*598.7*	*1.2*	*0.02*	*586.6*	*97.9*
1000	*32*	*128*	*606.3*	*1.2*	*0.01*	*600.2*	*98.9*
2000	*1*	*4*	*1043.5*	*1*	*0.25*	*782.6*	*75*
2000	*2*	*8*	*1086.5*	*0.9*	*0.12*	*956*	*87.9*
2000	*4*	*16*	*889.9*	*1.1*	*0.07*	*824.7*	*92.6*
2000	*8*	*32*	*708.6*	*1.4*	*0.04*	*675.9*	*95.3*
2000	*16*	*64*	*629.1*	*1.6*	*0.02*	*612.8*	*97.4*
2000	*32*	*128*	*655.3*	*1.5*	*0.01*	*647.2*	*98.7*