High-**P**erformance **C**omputing 20**24**

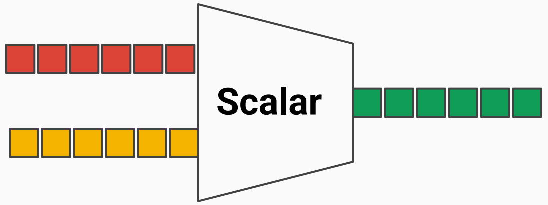
Parallel Computing, Distributed/Shared-Memory Parallelism and **OpenMP**

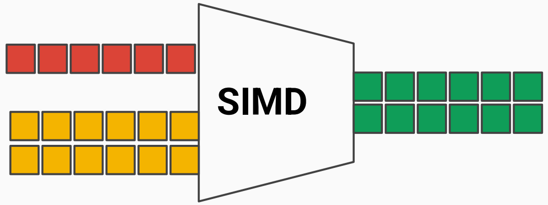
Aryan Eftekhari, Olaf Schenk | **U**niversità della **S**vizzera **i**taliana (USI) | **I**nstitute of **C**omputing (CI)

Review

SIMD–Single Instruction Multiple Data

An **instruction** is carried out on (multiple) **inputs** resulting in (multiple) **outputs**.





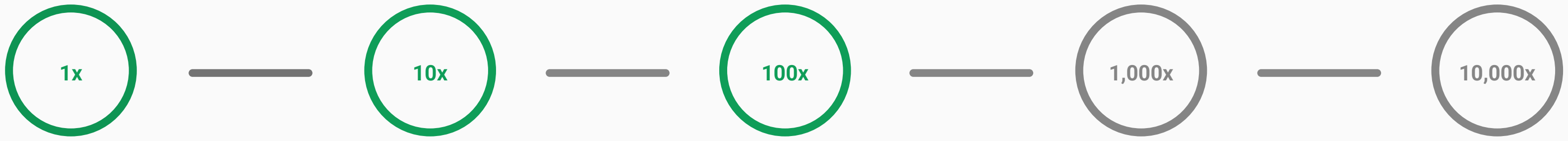
E.g., *Advanced Vector Extension* (AVX512[1]) (*NEON* for ARM[2]).

There are other levels of parallelism, such as instruction-level, thread-level, process-level, etc.

[1] https://www.intel.com/content/dam/develop/external/us/en/documents/31848-compilerautovectorizationguide.pdf
[2] https://developer.arm.com/Architectures/Neon

Review
Bandwidth–A sense of scale

The rate of read/write/transfer.

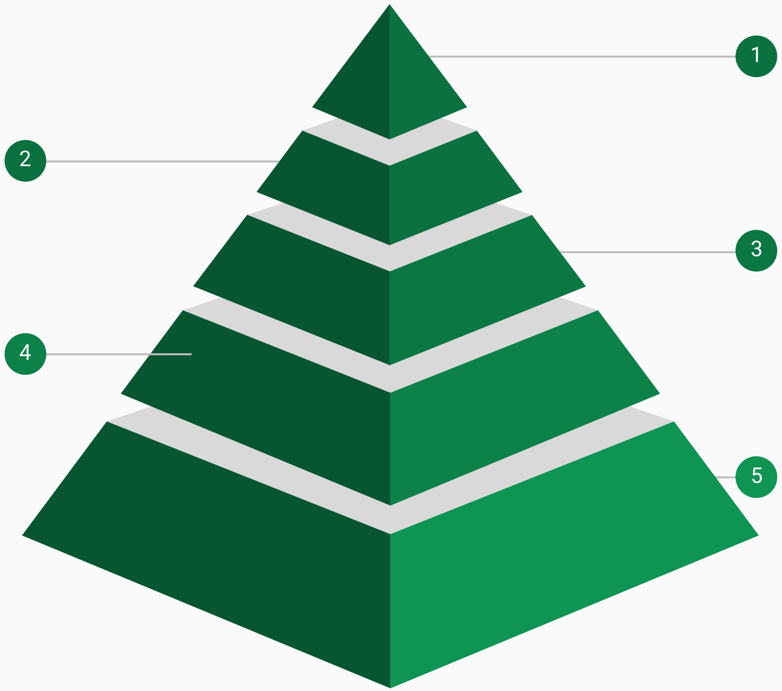


Cache	Main Memory	Local Network	Disk	Internet
L1/L2/L3 cache has a bandwidth in the ~100s of GB/s.	The bandwidth of ram is on the order of ~10 GB/s.	High-performance networks in data centers have a bandwidth of ~1GB/s.	The bandwidth of disks can vary depending on type of SSD ~0.1GB/s.	4G is about ~0.01 GB/s.

Think about bottlenecks ...

Review

Memory Hierarchy



1 **Registers** ~ 64–512 bit
Integral to the CPU, lowest latency. Size varies: x86-64 general-purpose registers are 64-bit, Apple's M1 SIMD registers are 128-bit, and some Intel CPUs have 512-bit SIMD registers.

2 **L1 Cache** ~16–128 KB
Extremely fast with only few cycles latency.

3 **L2 Cache** ~256KB–1MB
Very fast, slower than L1 but much faster than main memory.

4 **L3 Cache** ~1–8 MB
Fast, slower than L2 but much faster than main memory. *Shared with all cores.*

5 **Main Memory** ~32–128 GB
Compared to the L3 this is slow.

No free lunch...fast memory is expensive and limited in capacity.

The values above are for reference and can vary significantly between products and over time.

Review
Multi or Manycore Parallelism

Computers have stopped getting “faster” since the mid 2000’s !

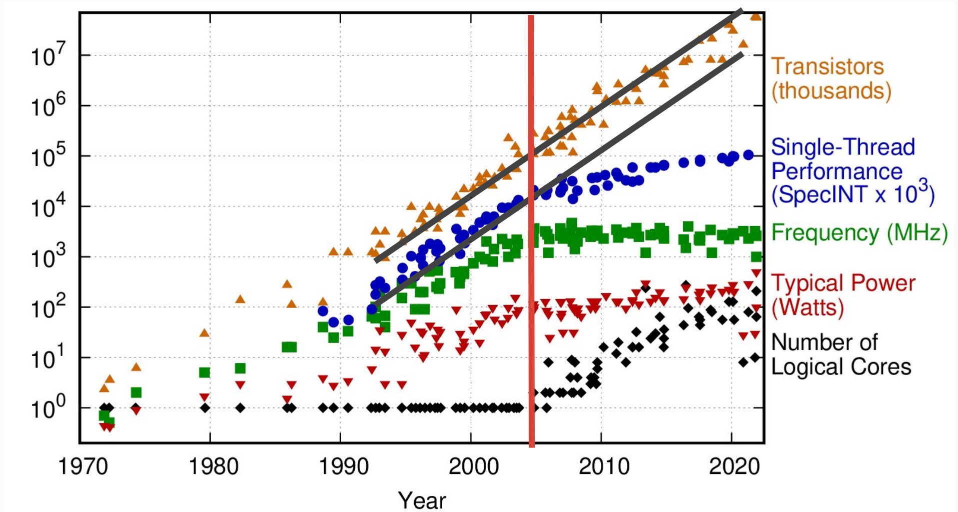
Provocative statement to get attention.

1970 to Mind 2000’s - Increasing Clock Speed

No longer an option due to heat and power.

Mind 2000’s to Current - Increasing Cores

More than one core, i.e., multicore processors.



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Overview

Parallel computing and **OpenMP**

Parallel Computing
What it is parallel computing?

Concurrent Computing
Operations are interleaved/overlapping, instead of sequentially.

Asynchronous Computing
Operations are executed in near future, i.e., nonblocking.

Parallel Computing
Operations are executed simultaneously.

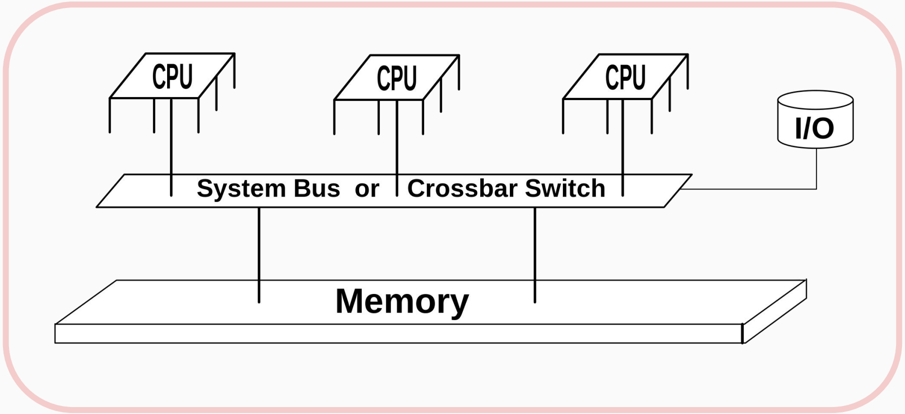
We are concerned with **Parallel Computing**.

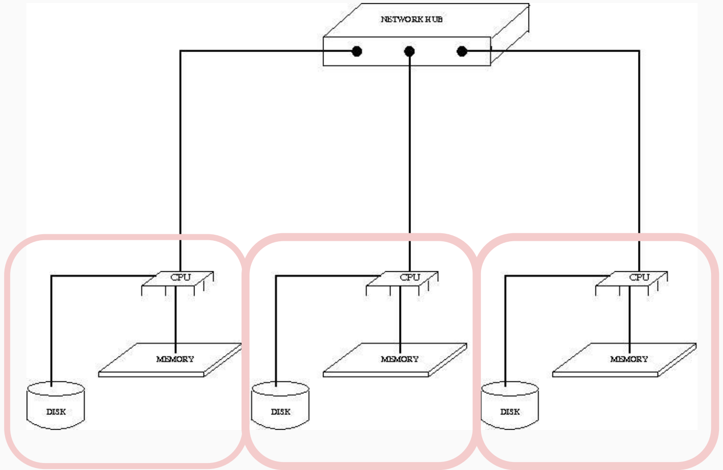
Parallel Computing
Shared & Distributed-Memory Systems

The basic models (example)

Shared Memory
CPUs have common memory.

Distributed Memory
CPUs do NOT have common memory.





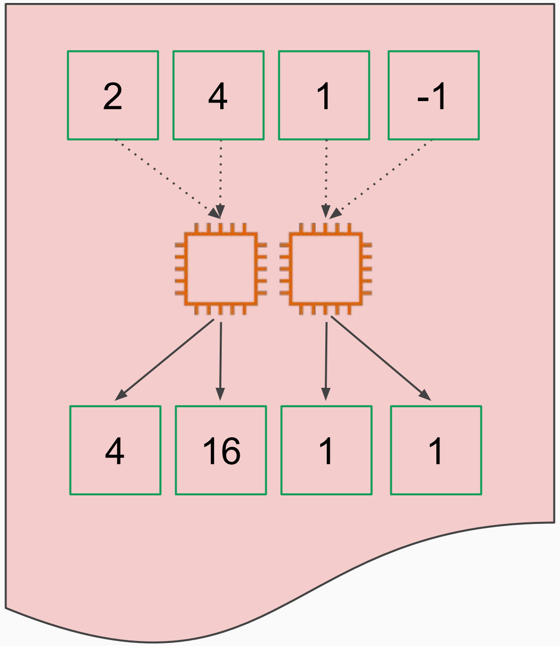
Parallel Computing
Why should I care?

- **Shared-Memory Parallelization**
  - Your computer is multi-core!
  - Efficient utilization of a computer i.e., a node.

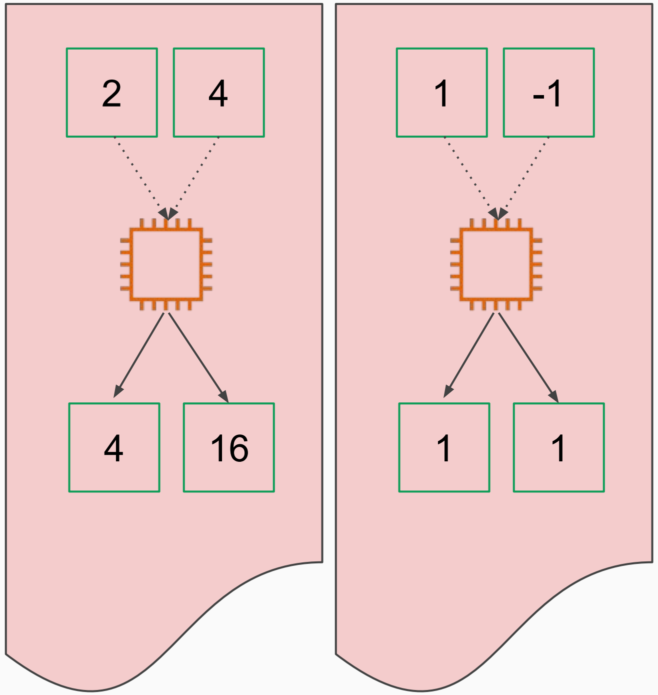
- **Distributed-Memory Parallelization**
  - Fundamental for scientific computing.
  - Base architecture for Supercomputing, i.e., multiple nodes.

Distributed/Shared-Memory Parallelism
Parallelization paradigms

Shared Memory Parallelization
Computation is distributed along ***threads****.*
Synchronization *between threads.*



Distributed Memory Parallelization
Computation is distributed along ***processes****.*
Communication *via message passing.*



Distributed/Shared-Memory Parallelism
Programming OpenMP/MPI

Shared-Memory Parallelization

Interface:
OpenMP: an API for C/C++/Fortran

Get it:
It's part of the GNU compiler pkg. (above gcc 4.2)

Use it:
1) Add `#include <omp.h>`
2) ***Add pragma statements to code**
3) Add `-fopenmp` flag
4) Set environment variable `OMP_NUM_THREADS`
5) Execute

**This can be tricky (race conditions).*

Distributed-Memory Parallelization

Interface:
MPI: Message Passing Interface a standard designed for parallel computing architectures.

Get it:
Different implementations, OpenMPI is common.

Use it:
1) Add `#include <mpi.h>`
2) ***Add MPI functions to pass messages.**
3) Compile with `mpic++` (not `g++`)
4) Execute `mpirun -np 2` (run with 2 process)

**This can change your program structure.*

Distributed/Shared-Memory Parallelism
Writing OpenMP vs MPI

```
//OpenMP
int main(){

   int n = 1e8;
   std::vector<double> val(n,0);















   #pragma omp parallel for
   for (int i = 0; i < n; i++){
      val[i] = COSTLY_OPERATION(i);
   }





   return 0;
}
```

```
// MPI
int main(){

   int n = 1e8;
   std::vector<double> val(n,0);

   int size, rank;
   MPI_Init(NULL,NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   int block_size = n/size;
   int start = rank*block_size;
   int end   = (rank+1)*block_size;
   if(rank==size-1){
      end=n;
   }

   for ( int i = start; i < end; i++){
      val[i] = COSTLY_OPERATION(i);
   }

   MPI_Allreduce( MPI_IN_PLACE, &val[0] , n , MPI_DOUBLE,
MPI_SUM,MPI_COMM_WORLD);
   MPI_Finalize();

   return 0;
}
```

OpenMP
Writing/Compiling a Basic OpenMP Program

OpenMP can be **easy to use** (e.g., `#Pragma omp…`), a lot of the work is done for you…

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

int main(){
    std::vector<double> val(1e8,0);
    #pragma omp parallel for
    for (int i = 0; i < val.size(); i++)
        val[i] = COSTLY_OPERATION(i);
    return 0;
}
```

Parallel Region

```
// In Terminal/Command line


// Compile via command line (or makefile)
g++ -fopenmp -O3 main.cpp -o main.exe


// Run
export OMP_NUM_THREADS=2; ./main.exe
```

… but debugging can be tricky.

OpenMP
Basics–Usage Blueprint

*“OpenMP” = **Directives** + **Clauses** + **Functions** + **Environment Variables**.*

Provide control over **parallelization** and **memory**.

```
double res = 0.0;
#pragma omp parallel for default(none) shared(a,b,n) reduction(+:res)
for (int i = 0; i < n; i++) {
  res += a[i] * b[i];
}
```

Parallelization: What do you want to do? (*Parallel for*).
Memory: How will the memory be mapped? (*shared/private*).

- *Private*: The memory is private to the thread (private variables are not initialized).
- *Shared*: The memory is shared among the threads.
- *Reduction*: The variable is "kind of shared and kind of not" ... more on this later.

For details see https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-library-reference

OpenMP
Basics–Number of Threads

There are **3** ways

```
// via terminal/command line
export OMP_NUM_THREADS=2; ./main.exe
```

1) **Runtime:** Set the environment variable `OMP_NUM_THREADS`.

```
// global
 omp_set_num_threads(4);
 #pragma omp parallel
 {
   …
 }
```

2) **Compile-Time Globally:** Via function `omp_set_num_threads`.

```
// local
 omp_set_num_threads(4);
 #pragma omp parallel num_threads(3)
 {
   …
 }
```

3) **Compile-Time Locally:** Via directive clause `num_threads`.

***Note**: (2) overrides (1) and (3) overrides (1) and (2). Option (1) is common.*

OpenMP
Directives–Parallel, Sections, For ...

```
// all threads enter this parallel region
#pragma omp parallel
{
  std::cout<<omp_get_thread_num()<< "/" << omp_get_num_threads()<<std::endl;
}


// give each section to any available thread (one thread per section)
#pragma omp parallel sections
{
   #pragma omp section
   {
       std::cout << omp_get_thread_num() << "/" <<omp_get_num_threads()<<std::endl;
   }
   #pragma omp section
   {
    std::cout << << omp_get_thread_num() <<"/"<< omp_get_num_threads()<<std::endl;
   }
}

 // divide work evenly among each thread
 #pragma omp parallel for
 for (int i = 0; i < a.size(); i++){
   a[i]= my_funct(i);
 }
```

There are many more! We will look at other directives in examples `barrier`, `critical`, and `atomic`.

OpenMP
Memory–Race Conditions

Behavior is dependent on the memory access sequence.

```
//This program is not deterministic
int sum=0;
#pragma omp parallel for default(none) shared(sum)
for (int i = 0; i < 100; i++){
        sum+= 1;
}
```

What is `sum` **before** you update it?

Use reduction clause → **`reduction(+:sum)`**

OpenMP
Memory–Data-Sharing Clauses

Challenges of shared-memory ...

```
//This program is not deterministic (but the print-out to screen is)
   sum=0.0;
   b=5.0;
   #pragma omp parallel for default(none) reduction(+:sum) private(b)
   for (int i = 0; i < 100; i++){
       sum+= 5.0/b;
   }
```

What is `b`, *before*, *in* and *after* the for-loop ?

- **shared/private** – All threads have a shared/private copy of the variable.
- **default** – Behavior of unscoped variables in a parallel region.
- **firstprivate** – Private variable initialized with the value of the variable outside `#pragma`.
- **lastprivate** – local copy of variable set with the value of the last iteration (or section) of parallel loop.

OpenMP
Reductions (A very common operation)

Common synchronization mechanism

```
// This program is deterministic
   sum=0.0;
   b=5.0;
   #pragma omp parallel for default(none) reduction(+:sum) firstprivate(b)
   for (int i = 0; i < 100; i++){
       sum+= 1.0/b;
   }
```

- **atomic** – No simultaneous reading and writing threads.
- **critical** – Execution restricted to a single thread at a time (more general than atomic).
- **barrier** – Wait for all threads in parallel region to reach the same point.

Consider these operations:

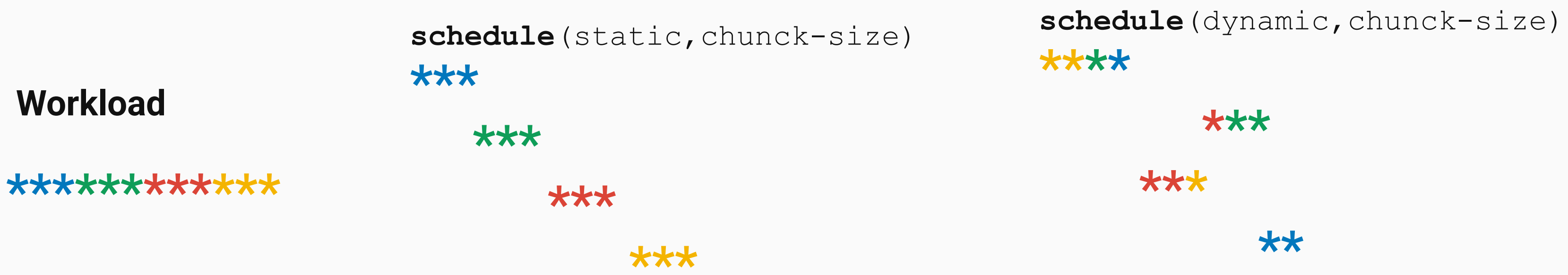
```
#pragma omp atomic
sum+= 5.0/b;
```

```
#pragma omp critical
sum+= 5.0/b;
```

```
//Not allowed (will not compile)…
even if it compiled it won't work!
     #pragma omp barrier
     sum+= 5.0/b;
```

OpenMP
Scheduling

Which thread does what ?



Static: Allocated workload at “compile time” (default chunck-size=work_load/num_threads)
Divide the iterations by chunk-size and distributes chunks to threads in a circular order (round-robin).

- *Thread is fixed to specific iteration.*
- *No performance overhead.*

Dynamic: Allocated workload at “run time” (default chunck-size=1)
Divide the iterations by chunk-size and request chunks until there are no more (first come, first serve).

- *Thread is NOT fixed to specific iteration.*
- *Performance overhead.*

There are other scheduling methods, see e.g, OpenMP 5.0 Technical Report 6

OpenMP
Motivation for using OpenMP

OpenMP **simplifies** shared-memory parallelism by abstracting the low-level details.

Università della Svizzera italiana
Institute of Computing CI

High-Performance Computing, Fall 2024
Lecturer: Dr. A. Eftekhari, Prof. O. Schenk
Assistants: M. Lechekhab, D. Vega
D. Santarsiero, J. Palumbo, I. Ecevit

Project 2 – Parallel Programming using OpenMP

Due date: 23 October 2024 at 23:59 (See iCorsi for updates)

In this project, we will use OpenMP. If OpenMP is new to you, we highly recommend the LLNL tutorial. See also the iCorsi page for further resources. For reference, we also recommend Chapters four to eight of the book [1].

All tests and simulation results must be run on the compute nodes of the Rosa cluster . However, feel free to try and develop on other available systems (e.g., your workstation or laptop) and compilers (the final code must compile and run on Rosa cluster). You will find all the skeleton source codes for the project on the course iCorsi page.
To help you get started with OpenMP, here is a sample OpenMP program that you can run using interactive jobs. The source code for the sample program is available in `skeleton_codes/hello_omp.cpp`

```
#include <iostream>
#ifdef _OPENMP
#include <omp.h>
#endif

int main() {
#ifdef _OPENMP
    #pragma omp parallel
    {
        if (omp_get_thread_num() == 0) {
            std::cout << "OpenMP threads=" << omp_get_num_threads() << std::endl;
        }
    }
#else
    std::cout << "OpenMP is not available." << std::endl;
#endif
    return 0;
}
```

You can compile the code using using, for example, `g++ -fopenmp hello_omp.cpp -o hello_omp`. Don't forget the `-fopenmp` flag! We recommend using a `makefile` (either provided or from other in-class examples) and adjusting them as needed. While developing/debugging your code, it can be useful to work in an interactive session (you can use the `--reservation=hpc-tuesday` or `--reservation=hpc-wednesday` for better priority) as follows:

```
[user@icslogin01 Test]$ srun --nodes=1 --exclusive --time=00:01:00 --pty bash -i
        srun: job 9737 queued and waiting for resources
        srun: job 9737 has been allocated resources
[user@icsnodeXX Test]$ export OMP_NUM_THREADS=2
[user@icsnodeXX Test]$ ./hello_omp
OpenMP threads=2
```

This allocates an interactive session on 1 node of the Rosa cluster for 1 minutes. The program sets the number of threads for parallel regions according to the value of the environment variable `OMP_NUM_THREADS` which is assigned before executing the code.
Note: Using `--exclusive` grants you exclusive access to the node, meaning that there is no sharing of resources—its all yours! It is very important to use exclusive allocation sparingly, as it restricts others from accessing the node. It is recommended to use, for example, `srun --nodes=1 --time=00:20:00 --pty bash -i` for code compilation and testing. When you are ready to run larger tests and collect results, such as strong scaling results, you would use `srun --nodes=1 --exclusive --time=00:01:00 --pty bash -i`.

1. Parallel reduction operations with `OpenMP` [20 points]

1.1. Dot Product

The file `dotProduct/dotProduct.cpp` contains the serial implementation of a `C/C++` program that computes the dot product $\alpha = a^T \cdot b$ of two vectors $a \in \mathbb{R}^N$ and $b \in \mathbb{R}^N$. A snippet of the code is shown below:

```
time_start = wall_time();
for (int iterations = 0; iterations < NUM_ITERATIONS; iterations++) {
  alpha = 0.0;
  for (int i = 0; i < N; i++) {
    alpha += a[i] * b[i];
  }
}
time_serial = wall_time() - time_start;
cout << "Serial execution time = " << time_serial << " sec" << endl;

long double alpha_parallel = 0;
double time_red = 0;
```

Solve the following tasks (Dot Product):

1. Implement two parallel versions of the dot product using `OpenMP`: (i) one using the `reduction` clause, and (ii) another using the `critical` directive.

2. Perform a strong scaling analysis on the Rosa cluster for both parallel implementations, using different numbers of threads $t = 1, 2, 4, 8, 16$ and 20 for various vector lengths $N = 10^5, 10^6, 10^7, 10^8$ and 10^9 (provide a strong scaling plot using log-axis where applicable). Discuss the observed differences between the implementation that uses the `critical` directive and the `reduction` clause. Keep all output of the code, these results will be used in the next question.

3. Perform an analysis on the parallel efficiency between the two parallel implementations (using the results from the above, provide a parallel efficiency plot using log-axis where applicable). Discuss `OpenMP` overhead and the relation between thread count, parallel efficiency, and workload (in this case the size of the vectors N). At what size of N does it become beneficial to use a multi-threaded version of the dot product?

1.2. Approximating π

From elementary calculus we know that for

$$f(x) = \frac{4}{1+x^2}, \text{ we have that } \int_0^1 f(x)\,\mathrm{d}x = 4\,\arctan(x)\Big|_0^1 = \pi.$$

We can approximate the integral with the midpoint rule as

$$\int_0^1 f(x)\,\mathrm{d}x \approx \sum_{i=1}^{N} f(x_i)\,\Delta x,$$

where the integration interval $[0,1]$ is uniformly partitioned into N subintervals of size $\Delta x = \frac{1}{N}$ and subinterval centers $x_i = (i + \frac{1}{2})\Delta x$ $(i = 1, \ldots, N)$.

The file `pi/pi.cpp` serves as a template (it does not implement the `C/C++` program that approximates π). Your goal is to implement both a serial version of the approximation and a parallelized one using `OpenMP`.

Solve the following tasks (Approximating π):

1. Implement serial and parallel versions of the π approximation method shown, using any parallelization scheme you prefer. Discuss your choice. Use a fixed value of $N = 10^{10}$.

2. Show the speedup between your parallel implementation and the serial version for $t = 1, 2, 4$,and 8 (include the plot using log-axis where applicable). Discuss the concept of speedup in relation to strong scaling and its connection to parallel efficiency (you don't need to create a strong scaling or parallel efficiency plot here; just discuss it).

2. The Mandelbrot set using `OpenMP` [20 points]

Write a sequential code in `C/C++` to visualize the Mandelbrot set. The set bears the name of the "Father of Fractal Geometry," Benoit Mandelbrot. The Mandelbrot set is the set of complex numbers c for which the sequence $(z, z^2 + c, (z^2+c)^2+c, ((z^2+c)^2+c)^2+c, (((z^2+c)^2+c)^2+c)^2+c, \dots)$ tends toward infinity. Mandelbrot set images are made by sampling complex numbers and determining for each whether the result tends towards infinity when a particular mathematical operation is iterated on it. Treating the real and imaginary parts of each number as image coordinates, pixels are colored according to how rapidly the sequence diverges, if at all. More precisely, the Mandelbrot set is the set of values of c in the complex plane for which the orbit of 0 under iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ remains bounded. That is, a complex number c is part of the Mandelbrot set if, when starting with $z_0 = 0$ and applying the iteration repeatedly, the absolute value of z_n remains bounded however large n gets. For example, letting $c = 1$ gives the sequence $0, 1, 2, 5, 26, \dots$ which tends to infinity. As this sequence is unbounded, 1 is not an element of the Mandelbrot set. On the other hand, $c = -1$ gives the sequence $0, -1, 0, -1, 0, \dots$ which is bounded, and so -1 belongs to the Mandelbrot set.
The set is defined as follows:

$$\mathcal{M} := \{c \in \mathbb{C} : \text{the orbit } z, f_c(z), f_c^2(z), f_c^3(z), \dots \text{ stays bounded}\}$$

where f_c is a complex function, usually $f_c(z) = z^2 + c$ with $z, c \in \mathbb{C}$. One can prove that if for a c once a point of the series $z, f_c(z), f_c^2(z), \dots$ gets farther away from the origin than a distance of 2, the orbit will be unbounded, hence c does not belong to $\mathcal{M}$. Plotting the points whose orbits remain within the disk of radius 2 after `MAX_ITERS` iterations gives an approximation of the Mandelbrot set. Usually a color image is obtained by interpreting the number of iterations until the orbit "escapes" as a color value. This is done in the following pseudo code:

```
for all c in a certain range do
        z = 0
        n = 0
        while |c| < 2 and n < MAX_ITERS do
                z = z^2 + c
                n = n + 1
        end while
        plot n at position c
end for
```

The entire Mandelbrot set in Fig. 2 is contained in the rectangle $-2.1 \le \Re(c) \le 0.7$, $-1.4 \le \Im(c) \le 1.4$. To create an image file, use the routines from `mandel/pngwriter.c` found in the provided sources like so:

```
#include "pngwriter.h"
png_data* pPng = png_create (width, height); // create the graphic
// plot a point at (x, y) in the color (r, g, b) (0 <= r, g, b < 256)
```

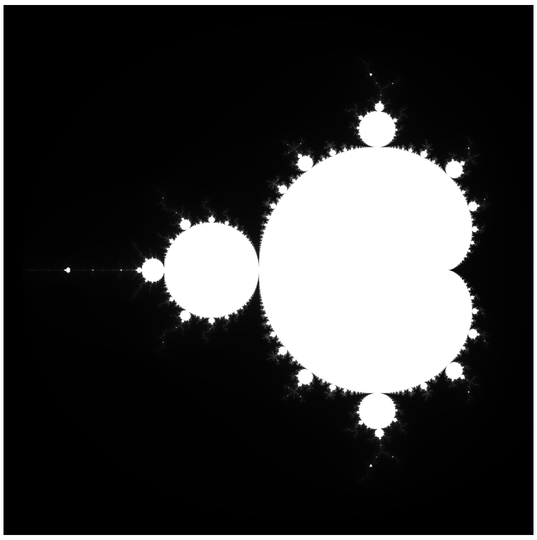


Figure 1: The Mandelbrot set

```c
png_plot (pPng, x, y, r, g, b);
png_write (pPng, filename); // write to file
```

You need to link with `-lpng`. You can set the RBG color to white $(r, g, b) = (255, 255, 255)$ if the point at (x, y) belongs to the Mandelbrot set, otherwise it can be $(r, g, b) = (0, 0, 0)$

```c
      int c = ((long)n * 255) / MAX_ITERS;
      png_plot(pPng, i, j, c, c, c);
```

Record the time used to compute the Mandelbrot set. How many iterations could you perform per second? What is the performance in MFlop/s (assume that 1 iteration requires 8 floating point operations)? Try different image sizes. Please use the following `C` code fragment to report these statistics.

```c
  // print benchmark data
  printf("Total time:                 %g seconds\n",
         (time_end - time_start));
  printf("Image size:                 %ld x %ld = %ld Pixels\n",
         (long)IMAGE_WIDTH, (long)IMAGE_HEIGHT,
         (long)(IMAGE_WIDTH * IMAGE_HEIGHT));
  printf("Total number of iterations: %ld\n", nTotalIterationsCount);
  printf("Avg. time per pixel:        %g seconds\n",
         (time_end - time_start) / (double)(IMAGE_WIDTH * IMAGE_HEIGHT));
  printf("Avg. time per iteration:    %g seconds\n",
         (time_end - time_start) / (double)nTotalIterationsCount);
  printf("Iterations/second:          %g\n",
        nTotalIterationsCount / (time_end - time_start));
  // assume there are 8 floating point operations per iteration
  printf("MFlop/s:                    %g\n",
        nTotalIterationsCount * 8.0 / (time_end - time_start) * 1.e-6);

  png_write(pPng, "mandel.png");
```

Solve the following problems:

1. Implement the computation kernel of the Mandelbrot set in `mandel/mandel_seq.c`:

```
// do the calculation
cy = MIN_Y;
for (j = 0; j < IMAGE_HEIGHT; j++) {
  cx = MIN_X;
  for (i = 0; i < IMAGE_WIDTH; i++) {
    x = cx;
    y = cy;
    x2 = x * x;
    y2 = y * y;
    // compute the orbit z, f(z), f^2(z), f^3(z), ...
    // count the iterations until the orbit leaves the circle |z|=2.
    // stop if the number of iterations exceeds the bound MAX_ITERS.
    int n = 0;
    // TODO
    // >>>>>>>> CODE IS MISSING

    // <<<<<<<< CODE IS MISSING
    // n indicates if the point belongs to the mandelbrot set
    // plot the number of iterations at point (i, j)
    int c = ((long)n * 255) / MAX_ITERS;
    png_plot(pPng, i, j, c, c, c);
    cx += fDeltaX;
  }
  cy += fDeltaY;
}
```

2. Count the total number of iterations in order to correctly compute the benchmark statistics. Use the variable `nTotalIterationsCount`.

3. Parallelize the Mandelbrot code that you have written using `OpenMP`. Compile the program using the GNU `C` compiler (`gcc`) with the option `-fopenmp`. Perform benchmarking for a strong scaling analysis of your implementation and provide a plot for your results as well as a discussion.

3. Bug hunt [15 points]

You can find in the code directory for this project a number of short OpenMP programs (*bugs/omp_bug1-5.c*), which all contain compile-time or run-time bugs. Identify the bugs, explain what is the problem, and suggest how to fix it (there is no need to submit the correct modified code).

Hints:

1. *bug1.c:* check `tid`
2. *bug2.c:* check shared vs. private
3. *bug3.c:* check barrier
4. *bug4.c:* stacksize http://stackoverflow.com/questions/13264274
5. *bug5.c:* locking order.

4. Parallel histogram calculation using `OpenMP` [15 points]

The following code fragment computes a histogram `dist` containing 16 bins over the `VEC_SIZE` values in a large array of integers `vec` that are all in the range $\{0, \ldots, 15\}$:

```
for (long i = 0; i < VEC_SIZE; ++i) {
  dist[vec[i]]++;
}
```

You find the sequential implementation in `hist/hist_seq.cpp`. Parallelize the histogram computations using `OpenMP` (skeleton code is provided in `hist/hist_omp.cpp`). Report runtimes for the original (serial) code, the 1-thread and the N-thread parallel versions. Document and discuss the strong scaling behaviour in your report (i.e., keeping the size `VEC_SIZE` of the large array fixed at its original value).

Hint: "False sharing" can strongly affect parallel performance (see, e.g., [1, Sec. 7.2.4]).

5. Parallel loop dependencies with `OpenMP` [15 points]

Parallelize the loop in the following code snippet from `loop-dependencies/recur_seq.c` (available in the provided sources) using `OpenMP`:

```
        double up = 1.00001;
        double Sn = 1.0;
        double opt[N+1];
        int n;
        for (n=0; n<=N; ++n) {
                opt[n] = Sn;
                Sn *= up;
        }
```

The parallelized code should work independently of the `OpenMP` schedule pragma that you will use. Please also try to avoid – as far as possible – expensive operations that might harm serial performance. To solve this problem you might want to use the `firstprivate` and `lastprivate` `OpenMP` clauses. The former acts like private with the important difference that the value of the global variable is copied to the privatized instances. The latter has the effect that the listed variables values are copied from the lexically last loop iteration to the global variable when the parallel loop exits. Comment on your parallelisation briefly in the report.

6. Quality of the Report [15 Points]

Each project will have 100 points (out of 15 point will be given to the general written quality of the report).

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.) and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi .

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:

  - All the source codes of your solutions.
  - Build files and scripts. If you have modified the provided build files or scripts, make sure they still build the sources an run correctly. We will use them to grade your submission.
  - project_number_lastname_firstname.pdf, your write-up with your name.
  - Follow the provided guidelines for the report.

- Submit your .tgz through iCorsi .

Code of Conduct and Policy

- Do not use or otherwise access any on-line source or service other than the iCorsi system for your submission. In particular, you may not consult sites such as GitHub Co-Pilot or ChatGPT.
- You must acknowledge any code you obtain from any source, including examples in the documentation or course material. Use code comments to acknowledge sources.
- Your code must compile with a standard-configuration C/C++ compiler.

References

[1] Georg Hager and Gerhard Wellein. Introduction to high performance computing for scientists and engineers. *Chapman & Hall/CRC Computational Science*, July 2010.

Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing Lab **Institute of Computing**

Student: Dennys Mike Huber Discussed with: Alberto Finardi & Leon Ackermann

Solution for Project 2

This project will introduce you to parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP

1.1. Dot Product

The implementation of the parallel versions of the Dot Product are given in Listing 1 and 2. Important to note here is that the critical construct is not used to compute `alpha_local` directly, which would lead to a serial code with an additional overhead. Rather compute the `alpha` locally on each thread and then use the critical construct to compute the final `alpha`. Both methods essentially perform a reduction operation, but is there a difference in performance?

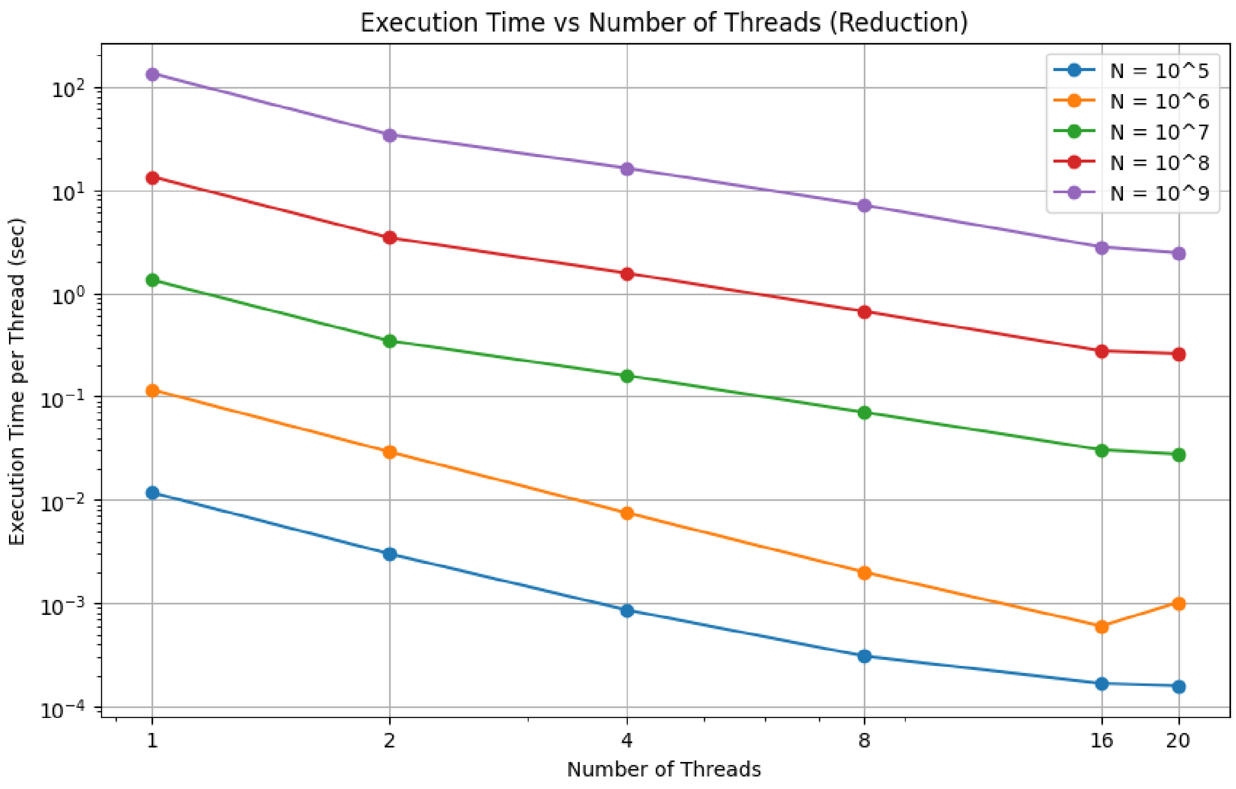
```c
 for (int iterations = 0; iterations < NUM_ITERATIONS; iterations++) {
   alpha_parallel_red = 0.0;
#pragma omp parallel for reduction(+ : alpha_parallel_red)
    for (int i = 0; i < N; i++) {
      alpha_parallel_red += a[i] * b[i];
    }
 }
```

Listing 1: Reduction Method

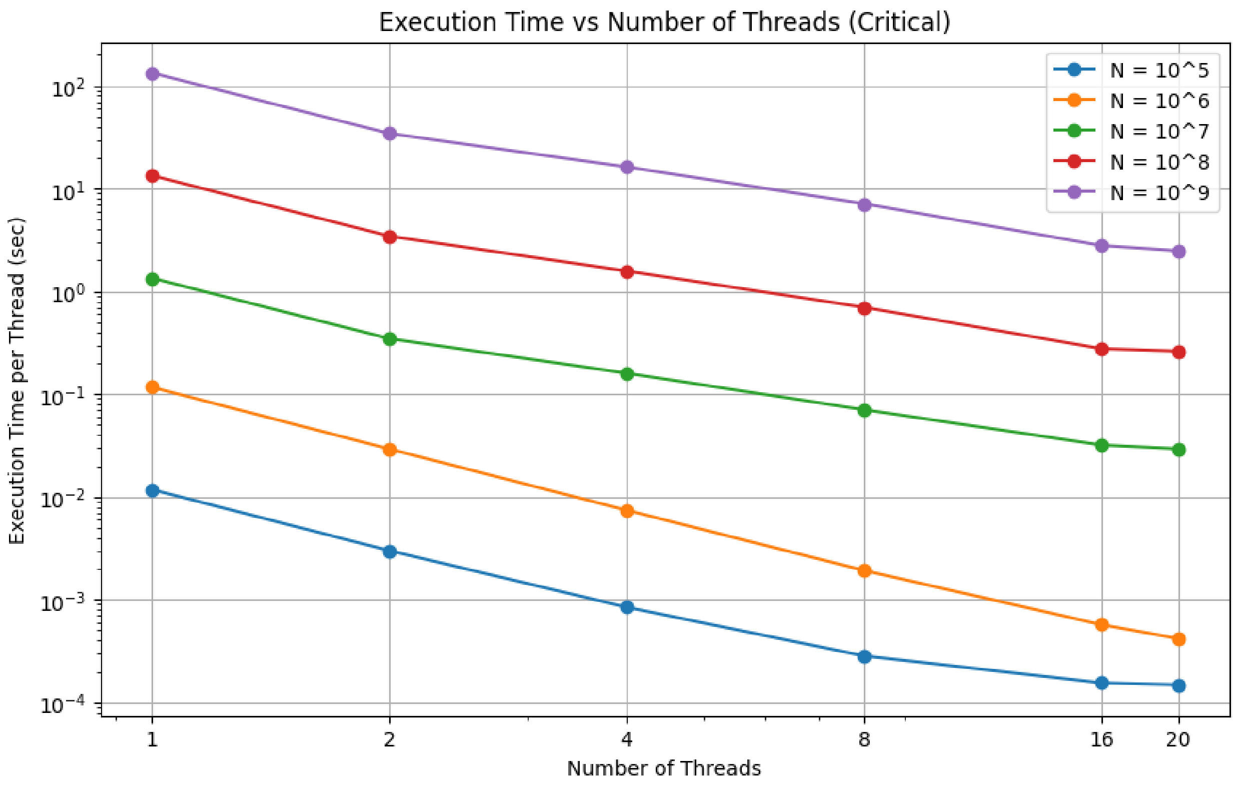
```c
for (int iterations = 0; iterations < NUM_ITERATIONS; iterations++) {
  alpha_parallel_crit = 0.0;
  alpha_local = 0.0;
#pragma omp parallel firstprivate(alpha_local)
{
#pragma omp for
  for (int i = 0; i < N; i++) {
    alpha_local += a[i] * b[i];
  }
#pragma omp critical
  {
    alpha_parallel_crit += alpha_local;
  }
}
}
```

Listing 2: Critical Section Method

In order to figure this out we use strong scaling analysis, where the amount of work stays constant (fixed size of the vector length) [1]. For this analysis we introduce a plot that compares the execution time per thread to the number of threads 1. Looking at the values of the plot, they appear to be identical,yet having a closer look at the actual values, we can see that the reduction method is a tiny bit faster, which is due to having a lower thread overhead compared to the critical implementation. For both methods we can clearly deduct that when increasing the number of threads from 1 to 16, we can see an significant decrease in execution time, while increasing the number of threads from 16 to 20 we either get a very small decrease in execution time or the execution even increases. This is mostly likely caused by the overhead caused the synchronization and the increased competition of resources, such as cache coherency and memory bandwidth. Comparing the slope of the curves, in both cases we can clearly see that for vector sizes 10^5 and 10^6 the execution time decreases more rapidly, meaning when we add more threads to this vector size for small number of threads the speedup is a lot better and faster.

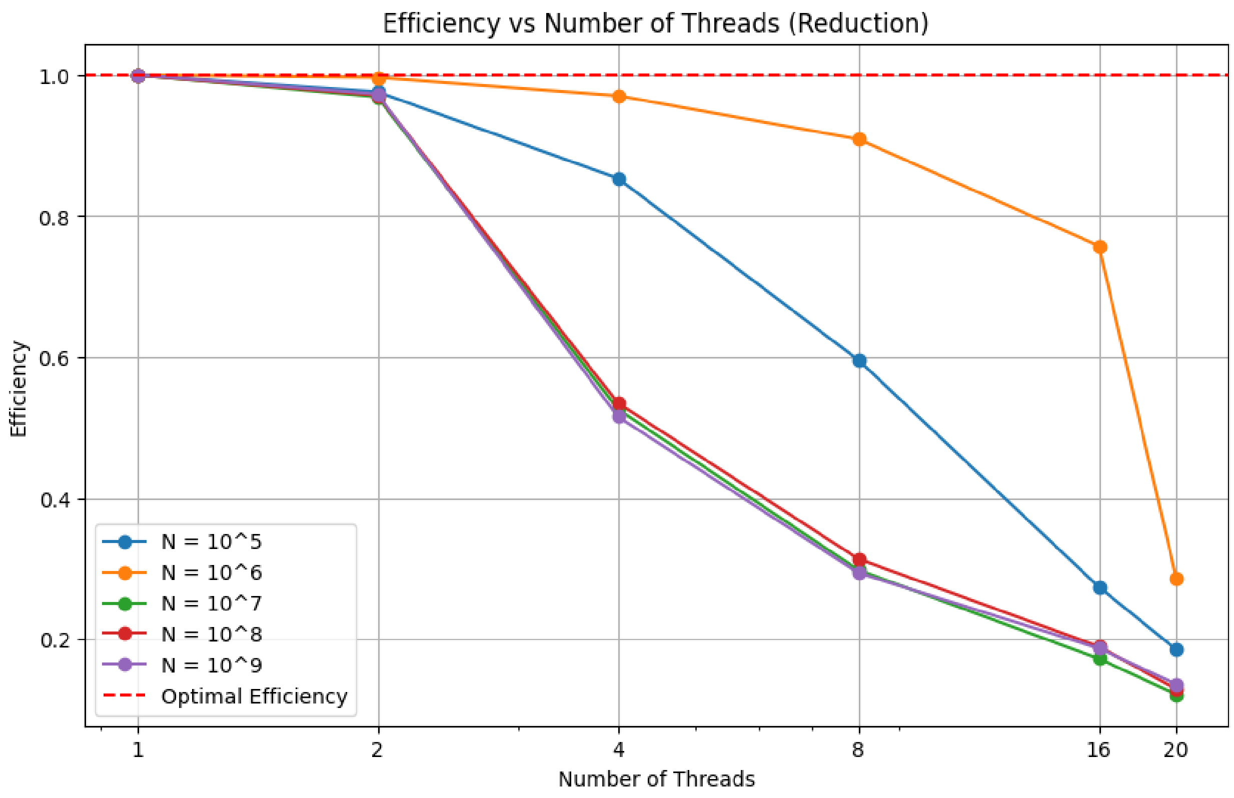


(a) DotProduct using the Reduction clause

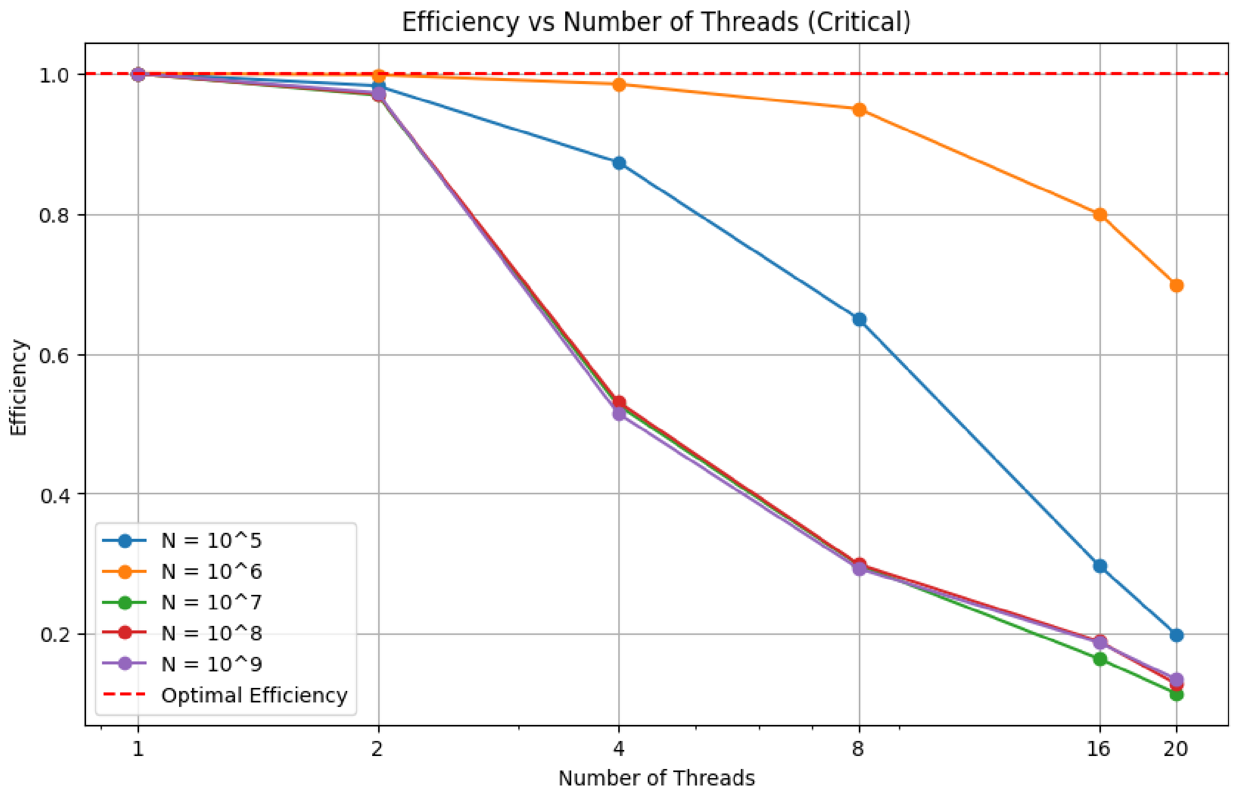


(b) DotProduct using the Critical section

Figure 1: Strong scaling comparison of the Reduction method and the Critical Method using the execution time per thread



(a) Efficiency of DotProduct using the Reduction clause



(b) Efficiency using the Critical section

Figure 2: Efficiency comparison of the Reduction method and the Critical Method

Another question posed in scalability is how effectively can a given resource be used in a parallel program [1]. We can define parallel efficiency as

$$E = \frac{T_1}{nT_n} \tag{1}$$

where T_n is the execution time of thread n-th thread. Looking at the efficiency plots in Figure 2, we can observe once more that both methods critical and reduction behave almost identical. The plot also clearly shows that for vector sizes 10^5 and 10^6 can use the small number of threads more efficiently, while for a higher number of threads they experience a sharp drop in efficiency. This suggests that for higher number of threads the managing of threads is high in comparison to the computational work. For a larger vector size initially there's a large decline of efficiency, but with increasing number of threads the decline is rather gradual, which makes sense, since large arrays can distribute their elements more effectively over a large number of threads. In addition for a larger array, the time OpenMP needs to create and manage threads is lower in relation to the time the threads needs to compute the dot product. If we compare the different vector sizes with the optimal efficiency then the for the two smallest vector sizes are the most efficient or in other words closest to the optimal efficiency.

At what point the multi-threaded version of the dot product becomes beneficial depends on the balancing of the overhead versus the amount of work per thread. This is usually indicated by a sharp drop in the efficiency plot. In our case beneficial for all N because the multi-threaded version is faster than the serial, but it is important not to use too many threads, depending on the vector size.

1.2. Approximating π

For my parallel version of pi simply use `omp parallel for` directive to distribute the work over the available threads. In addition I use the reduction clause to ensure the safe combination of the individual sums coming from each thread, avoiding race conditions when multiple threads write to a shared variable. The load balancing is done automatically by OpenMP, which should be well optimized for such a simple operation.

```
  /*  Serial Version of Computing Pi*/
  for (long long int i = 0; i < N; i++) {
    x = dx * (i + 0.5);
    sum += 4 / (1 + x * x);
  }
  pi = sum * dx;

  /*  Parallel Version of Computing Pi*/
  sum = 0.;
#pragma omp parallel for reduction(+ : sum)
  for (long long int i = 0; i < N; i++) {
    x = dx * (i + 0.5);
    sum += 4 / (1 + x * x);
  }
  pi = sum * dx;
```

Listing 3: Serial and Parallel implementation of Pi

The speedup is defined by the ratio

$$S_n = \frac{T_1}{T_n} \tag{2}$$

where T_1 and T_n are the execution time of the serial and parallel version with n threads. It shows us how much faster a parallelized program runs in comparison to its serial counterpart. The optimal (theoretical) speedup is only achievable with perfect parallelization, where no overhead or bottlenecks exist and is proportion to the number of threads $S_n = n$. In realty this not achievable, because in order to run a program in parallel we always have an overhead for communication,

synchronization, load balancing among others. There are different types of speedup plots we can look at in this case we look at strong scaling problem, which means the input size stays fixed at vector length 10^{10}.
The efficiency is closely related to the speedup and as seen in equation 1, it is the speedup divided by the number of threads.

Looking at the results of the speedup plot in Figure 3, this implementation of computing Pi comes very close to the optimal speedup, which means that the overhead is being held to a minimum and adding an additional thread halves the execution time. In terms of efficiency for this program it is very close to 1 and creates approximately a constant line.

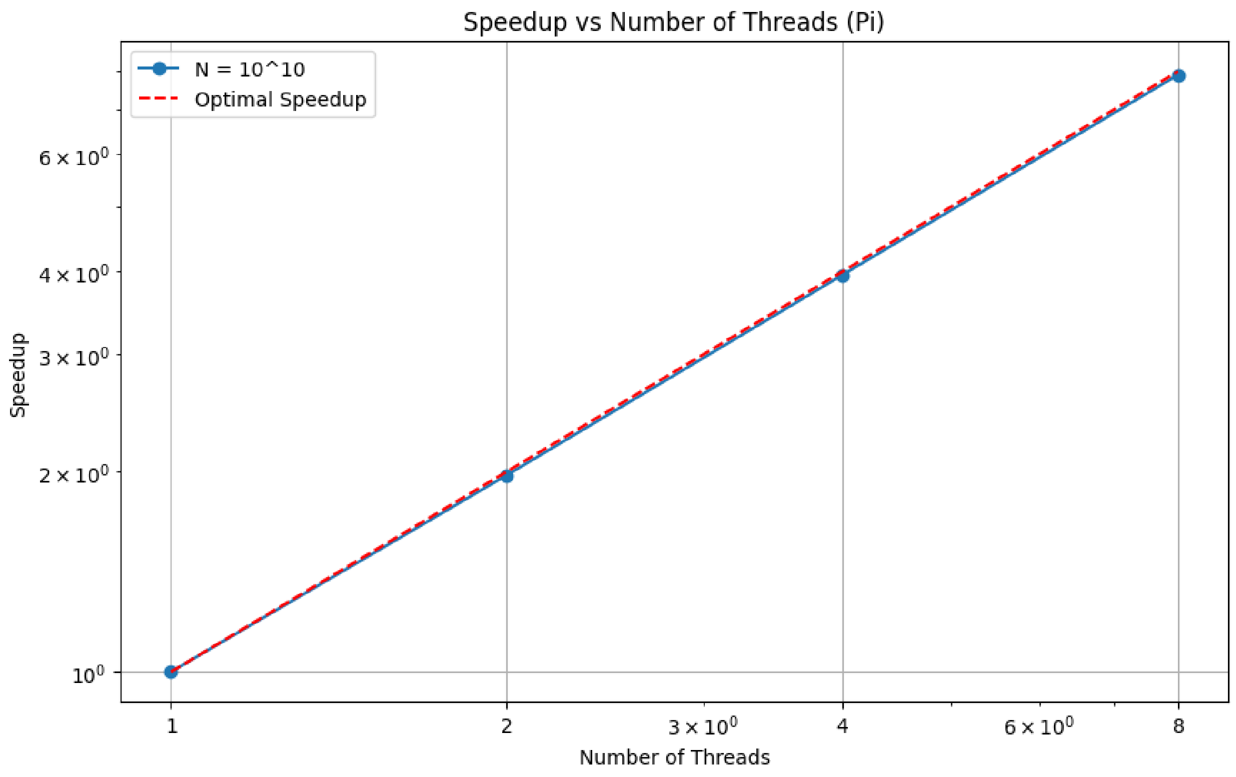


Figure 3: Speedup plot for computing pi

2. The Mandelbrot set using OpenMP

For the Mandelbrot using OpenMp we base our strong scale analysis on the closely related speedup and efficiency plot in Figure 5. For reference the dotted curves represent in both plots theoretical optimum. The first point that stands out is that the speedup and the efficiency does not depend on the resolution. This suggests that when increase the resolution the proportion of workload remains the same. Looking at the observed speedup, from single to 2 threads the speedup almost ideal, but from 2 to 4 threads the speedup stays almost identical, following that, the speedup increases steadily. A key observation is that for higher number of threads the speedup moves farther away from the ideal curve due to increased costs, such as synchronization time, memory bandwidth limitations and cache contention. This behavior can directly translated to the efficiency plot as well, where the efficiency decreases quickly and then deceases at a lower rate until from 16 to 20 threads the efficiency is almost identical. This is a common behavior in strong scaling experiments, because at a certain point the same factor as mentioned in the speedup plot start to weigh in and slow the implementation down, particularly with a higher number of threads.

Threads	**Iterations/second**	**MFlop/s:**	**Total time:**
1	2.075e+08	1660.67	155.782
2	4.144e+08	3315.96	78.036
4	4.291e+08	3433.23	75.370
8	6.393e+08	5115.16	50.587
16	1.130e+09	9045.01	28.608
20	1.397e+09	11177.4	23.150

Table 1: Subset of benchmark key values for the resolution 4096 $\times$ 4096 with roughly 32.34 billion iterations

The benchmarking used in this exercise also evaluates other fascinating key values, in Table 1 we can see an example for a specific resolution, where we can clearly see that when increasing the number of threads more iterations per second can be executed, which is also reflected in the MFlop/s and the decreasing total execution time. This shows the effective distribution of the workloads across multiple threads, but the gains made by adding more threads, as seen in the efficiency and speedup plot, will at some point have diminishing returns. Overall the performance gains are significant, especially up to 16 threads.

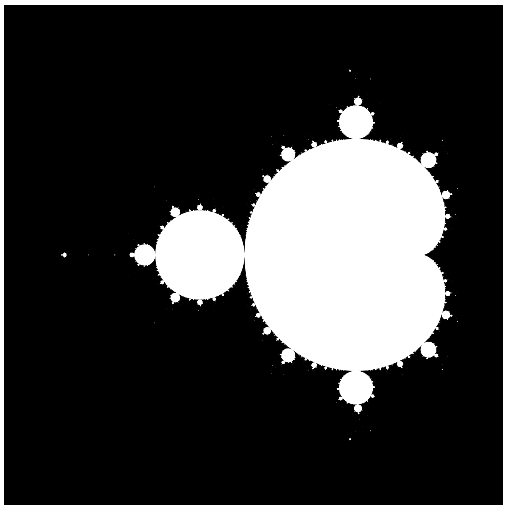
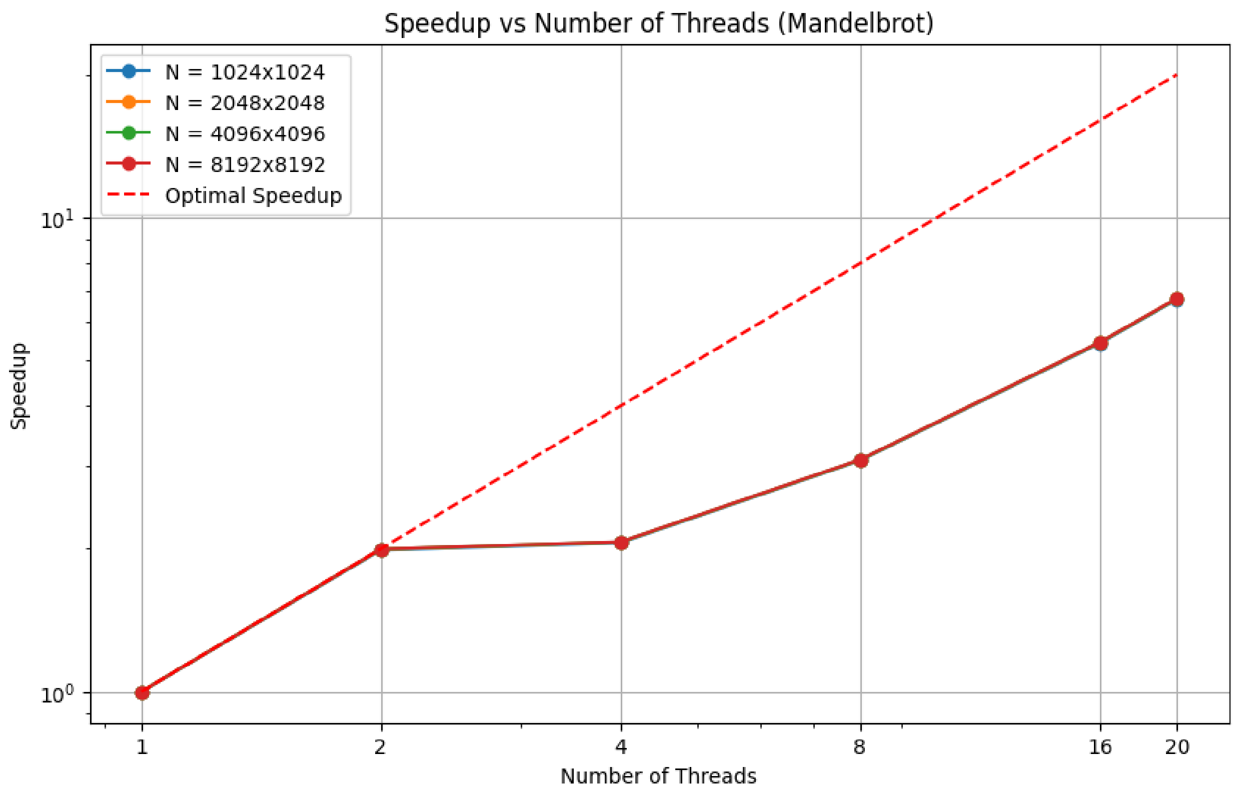
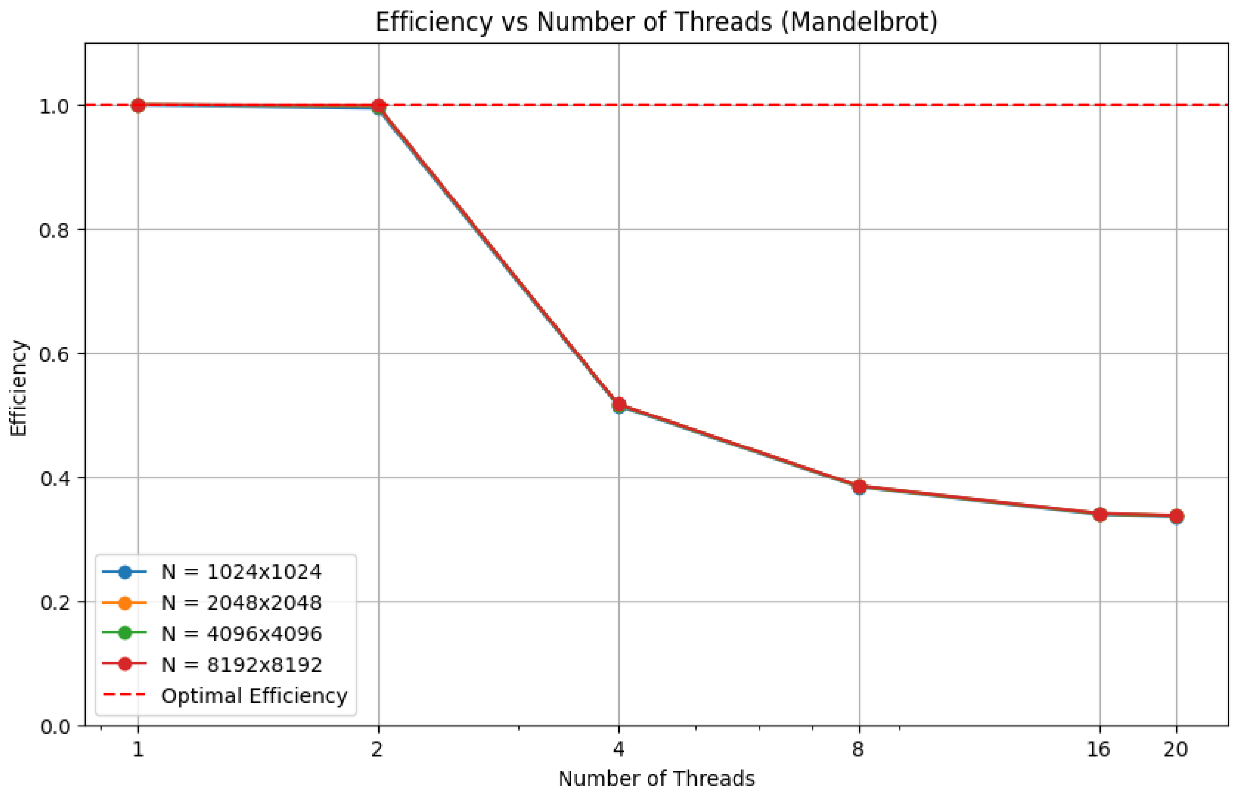


Figure 4: The Mandelbrot set with resolution 4096 $\times$ 4096 using parallel OpenMP implementation

Also important to mention is that in my implementation, I use `omp for collapse(2)` statement, in order to parallelize over both outer loops, which iterate over the x and y coordinates. I used this with the premise to allow multiple loops to be collapsed into a single loop and parallelize them a single entity, hence trying to have the iteration space evenly spread out over all processes. Comparing my results with two of my fellow students, which use a standard `omp for` loop, showed no significant improved or deterioration in execution time. This leads me to believe that OpenMP performs very similar optimization based on both statements.



(a) Speedup



(b) Efficiency

Figure 5: Speedup and Efficiency for the Mandelbrot implementation

3. Bug Hunt

3.1. Bug 1: Check tid

For the first bug, we can find the issue in the #pragma statement, see Listing 4, which is caused by already including the for and the schedule(static, chunk) statement, both of which can only be used right before a for-loop.

```
#pragma omp parallel for shared(a, b, c, chunk) private(i, tid) \
              schedule(static, chunk)
  {
    tid = omp_get_thread_num();
    /* CODE */
  }
```

Listing 4: Bug in bug1.c

In order to fix this bug, we need to split the #pragma statement in two distinct ones. The first one indicating that we are in a parallel region, shown on Line 1 of Listing 5 and the second #pragma omp for statement on Line 4 right before the loop we want to parallelize.

```
1 #pragma omp parallel shared(a,b,c,chunk) private(i, tid)
2 {
3     tid = omp_get_thread_num();
4 #pragma omp for schedule(static, chunk)
5     for (i = 0; i < N; i++) {
6       c[i] = a[i] + b[i];
7       printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
8     }
9 }
```

Listing 5: Fix for bug1.c

3.2. Bug 2: Check shared vs private

The second bug can be fixed by changing the variables tid and i to private variables.

```
#pragma omp parallel private(tid, i)
```

Listing 6: Fix bug2.c

By doing this every thread is now assigned its proper thread Id (tid) and the number of threads is now only printed by thread with index 0. Furthermore by setting i to private, each thread can now process its chunk of the total, otherwise i gets incremented by every thread, which could lead to certain i being skipped in the total computation.

3.3. Bug 3: Check barrier

For this bug it is critical to understand what the sections constructs does. Consulting the documentation [2], sections distribute work among the threads in a parallel region. This means in our case one of the threads executes section 1, another section 2 and the rest are preceding to the omp barrier after the omp sections nowait region. Important to note here is that the nowait option is used which removes the default barrier when exciting the sections construct.
In addition both omp section call the print_results function, which includes another omp barrier statement at the end of the function.
In OpenMP, threads cannot differentiate between different barriers in parallel region, which means that when the two threads executing the code in the section 1 and 2 reach the omp barrier statement in the print_results function, all the other threads are continuing executing the code, because every thread reached a barrier. When the threads that executed the sections reach the barrier after the section they halt there and wait until all the other threads reach barrier, which never happens, hence the program does not halt.

The solution to fix this bug is simply to remove the `omp barrier` statement at the end of the `print_results` function. Then the threads not entering a section, are waiting until the other two threads are finished with their section and all the threads are printing the exciting together.

3.4. Bug 4: Stacksize

In OpenMP there exists a stack size limit for each thread in a parallel region. On Unix like systems you can run the command

```
ulimit -a
# Example output in KB
8176
```

Listing 7: Find current stacksize on Unix-like system

In our case we a 1048×1048 matrix of type double, for which every thread gets its own copy. This Matrix therefore requires $8 \cdot 1048 \cdot 10480.001 = 8786.432KB$ of storage. In our example the matrix requires more storage than the stack size given in 7. In order to store further variables, you have to increase the storage size even further. This is possible by using the following command

```
ulimit -s 9000
```

Listing 8: Setting new stacksize on Unix-like system

3.5. Bug 5: Locking order

In our final bug we found ourselves in a deadlock situation in each of the `omp section` which are execute in parallel, where a thread locks `locka` and simultaneously another thread locks `lockb`, as a next step both threads want to lock each others `omp_lock_t`, but because its already locked they cannot do that, resulting in a deadlock. The solution can be seen in the simplified code in Listing 9, where each thread only locks one of the variable and then as soon as the lock is not needed anymore it is unlocked.

```
#pragma omp sections nowait
    {
#pragma omp section
      {
        omp_set_lock(&locka);
          /* MORE CODE */
        omp_unset_lock(&locka);
        omp_set_lock(&lockb);
          /* MORE CODE */
        omp_unset_lock(&lockb);
      }

#pragma omp section
      {
        omp_set_lock(&lockb);
          /* MORE CODE */
        omp_unset_lock(&lockb);
        omp_set_lock(&locka);
          /* MORE CODE */
        omp_unset_lock(&locka);
      }
    } /* end of sections */
```

Listing 9: Non-Deadlock fix for omp_bug-5.c

4. Parallel histogram calculation using OpenMP

In this exercise we will be analyzing the strong scaling of the parallel implementation of the histogram computation. The measured timings can be found in Table 2. Because this is strong scaling

analysis we kept the array size of the vector constant at $2 \cdot 10^9$ throughout the tests.
The serial execution provides us with a baseline for comparison with the parallelized versions. The single threaded execution shows a minor increase in performance, which is unexpected, due to the fact that there should be an overhead due to the thread management. This indicates that the overhead was somehow less significant and it could be based on the workload of the cluster. As we increase the number of threads, between 2 and 8 threads, we can observe a significant reduction in execution time, which indicates efficient scaling, when the workload is distributed over an increasing number of threads.
When comparing the timing from 8 to 16 threads, the execution time still slightly decreases, but at much lower rate than before. This leads to the conclusion that the efficiency decreases and the thread management overhead and potential contention between threads for shared resources.
From 32 onwards the timing increases, which means that the overhead outweighs the benefits gained from parallelizing the code. Another important factor that plays into the slower execution time is that the node only has 20 cores, resulting in more threads than cores, which means that certain threads potentially have to wait their turn until they can execute their chunk of code.

Threads	**Timing**
Serial	0.839258s
1	0.830452s
2	0.418288s
4	0.213519s
8	0.110719s
16	0.112852s
32	0.120083s
64	0.127492s
128	0.129270s

Table 2: Execution time for strong scaling parallel histogram calculation

5. Parallel loop dependencies with OpenMP

Two major changes were made in order to parallelize the code and keep it working independently of the schedule pragma. Because of the latter point, we are not allowed to use the scheduling to our advantage, therefore we have to compute the `Sn` using the power function, this guarantees that `Sn` is properly computed no matter if it is serial, dynamic or static scheduled.
Furthermore we introduce the `firstprivate` and `lastprivate` for `Sn`, therefore the predefined value before the loop is copy into a private variable for each thread and that after the parallel for loop `Sn` is assigned to the value computed of the last iteration of the loop when $n = N$. `Sn` can then be used for the print statement, which follows afterwards.

```c
#pragma omp parallel for firstprivate(Sn) lastprivate(Sn)
  for (n = 0; n <= N; ++n) {
    Sn = pow(up, n);
    opt[n] = Sn;
  }
```

Listing 10: Parallelized section of recur_omp.c

References

[1] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. en. Google-Books-ID: rkWPojgfeM8C. CRC Press, July 2010. ISBN: 978-1-4398-1193-1.

[2] *IBM OMP Sections*. en-US. June 2024. URL: `https://www.ibm.com/docs/en/zos/3.1.0?topic=processing-pragma-omp-section-pragma-omp-sections` (visited on 10/20/2024).