HPC Project

Parallel and Distributed Computing

Gabriele Pintus

May 30, 2024

University of Trieste

Introduction

Assignment 1:

Measure the latency of multiple collective operations

Assignment 2:

Implement a parallel code which computes the Mandelbrot set

Assignment 1:
Latency of CoOps

Assignment 1

Measure the latency of the following MPI operations:

- Point to Point communication
- Broadcast
- Scatter

We use the OSU Micro-Benchmarks suite.

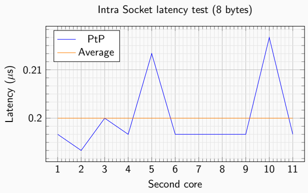
Latency

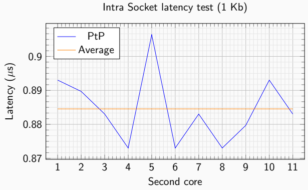
We use the `osu_latency` benchmark to measure the latency of the Point-to-Point communication between two processes.
We perform this test on three different topologies:

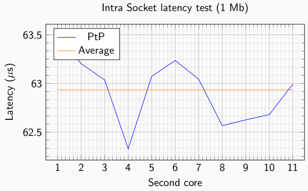
- **Intra-socket**: same socket, different cores
- **Intra-node**: same node, different sockets
- **Intra-cluster**: different nodes

We spawn two processes. The first is always bound to core 0.

Latency - Intra-socket

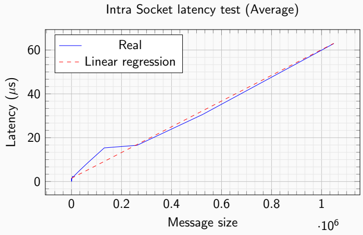






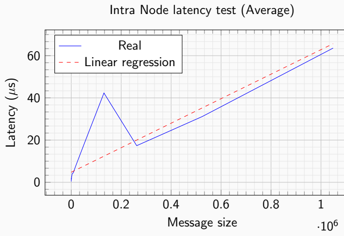
Message size	Average (μs)	Std (μs)	Std/Average
8 bytes	0.19	0.007	0.0362
1 Kb	0.88	0.009	0.0111
1 Mb	62.93	0.33	0.005

The latency become more stable as the message size increases.



β_1: 58.7μs/MB, β_0: 1.49μs.

Latency - Intra-node

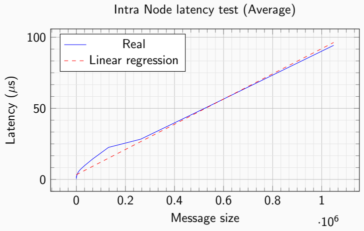


β_1: 58.7μs/MB, β_0: 4.76μs.

Message size	**Average (μs)**	**Std (μs)**	**Std/Average**
8 bytes	0.405	0.005	0.0123
1 Kb	1.9175	0.0217	0.0113
1 Mb	63.6575	0.8918	0.0140

The slope is the same as the intra-socket test, but the intercept is higher.
Higher overhead at low message sizes.

Latency - Intra-cluster



β_1: $88,69\mu$s/MB, β_0: 3.16μs.

Message size	Average (μs)	Std (μs)	Std/Average
8 bytes	1.1250	0.0622	0.0553
1 Kb	1.9550	0.1484	0.0759
1 Mb	94.2250	0.3829	0.0041

Higher slope and intercept.

Less overhead at low message sizes.

Broadcast

We use the `osu_bcast` benchmark to measure the latency of the Broadcast operation. Sending data from the root process to every other process.
Distribiution algorithms:

1. Basic Linear
2. Chain
3. Binary Tree
4. Binomial Tree

Broadcast algorithms

Basic Linear:
The root process sends the data to every other process in a round-robin fashion.
Number of steps: $n - 1$. **Chain**:

The root process sends the data to the next process, which forwards the data to the next process, and so on.
Number of steps: $n - 1$. **Binary Tree**:

The data is divided in chunks. The root process sends the first two to two different processes. Each of these processes sends the data to two other processes, and so on.
Number of steps: $\log_2(n)$.

Broadcast algorithms

Binomial Tree:

Each process communicates with a subset of other processes.

$t_0 : \; P_0 \to P_1,$

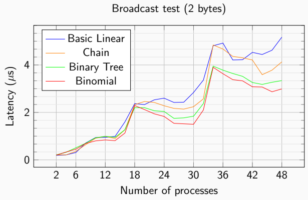
$t_1 : \; P_0 \to P_1, \; P_2 \to P_3,$

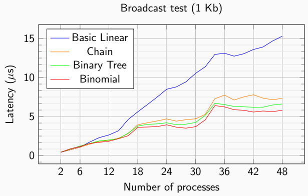
$t_2 : \; P_0 \to P_4, \; P_2 \to P_5, \; P_3 \to P_6,$

$t_3 : \; \dots$

Number of steps: $\log_2(n)$. Data transmitted: $(n-1) \cdot m$.

Broadcast





Especially at low message size, we can notice the two main transitions in the latency:

- At 12 processes the first socket is filled
- at 24 processes the first node is filled

After each spike, the latency slightly decreases.

Scatter

We use the `osu_scatter` benchmark to measure the latency of the Scatter operation. Sending a partition of the data from the root process to every other process.
Distribution algorithms:

1. Basic Linear
2. Binomial Tree

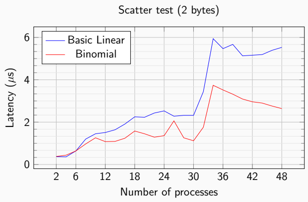
Scatter algorithms

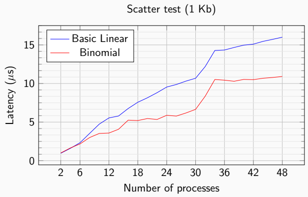
Basic Linear:
The root process sends the specific data chunk to each process in a round-robin fashion.
Number of steps: $n - 1$.

Binomial Tree:
The root process sends a subset of the data to a subset of processes.
Assume P_0 holds an array divided into 8 chunks $[c_0, \ldots, c_7]$ and $P_1, \ldots, p_7$ are the other processes.

$$t_0 : \ P_0 \xrightarrow{c_4, \ldots, c_7} P_1,$$
$$t_1 : \ P_0 \xrightarrow{c_2, c_3} P_2, \ P_1 \xrightarrow{c_6, c_7} P_3,$$
$$t_2 : \ P_0 \xrightarrow{c_1} P_4, \ P_1 \xrightarrow{c_5} P_6, \ P_2 \xrightarrow{c_4} P_5, \ P_3 \xrightarrow{c_7} P_7,$$

Scatter





For both message sizes, the binomial tree algorithm is faster than the basic linear algorithm. This was expected since it has a lower number of steps.

Assignment 2: The Mandelbrot Set

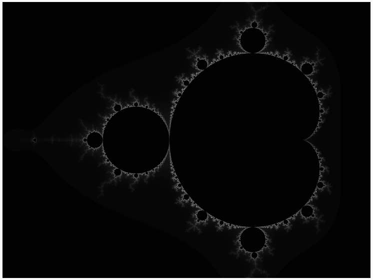
Assignment 2

Implement a hybrid C code which computes the Mandelbrot set using:

- **OpenMP**: Open Multi-Processing
- **MPI**: Message Passing Interface

Mandelbrot Set

Is the set of all complex numbers c for which the sequence defined by $z_{n+1} = z_n^2 + c$ does not diverge.



Arithmetic Intensity

Defined as the ratio of the number of arithmetic operations to the number of memory operations.

Arithmetic operations:

- 2 mult and 1 add: `z.x*z.x+z.y*z.y`
- 2 mult, 1 sub and 1 add: `temp=z.x*z.x-z.y*z.y+c.x`
- 2 mult and 1 add: `z.y=2*z.x*z.y+c.y`
- 1 add: `++n`

Memory operations:

- 1 write: `int n = 0`
- 2 writes: `Complex z = {0,0}`
- 2 reads: `z.x*z.x+z.y*z.y`
- 3 reads and 1 write: `temp=z.x*z.x-z.y*z.y+c.x`
- 3 reads and 1 write: `z.y=2*z.x*z.y+c.y`
- 1 write: `z.x=temp`
- 1 write: `++n`

Arithmetic Intensity

The arithmetic intensity is then

$$\frac{11 \cdot N}{10 \cdot N + 3} = \frac{11}{10 + \frac{3}{N}} \xrightarrow{N \to \infty} \frac{11}{10} = 1.1 \text{ flops/byte} > 1$$

Compute-bound when the number of iterations is large, i.e. in the light and internal regions of the set.
Memory-bound when the number of iterations is small, i.e. in the dark external regions of the set.
We are not considering any kind of optimization!

OpenMP

General purpose API for parallel programming in C, C++ and Fortran.

Allow to parallelize loops, sections, tasks and more with minimal effort.

```
void mandelbrot_set(uint8_t *image) {
    ...
    #pragma omp parallel for schedule(dynamic, CHUNK_SIZE)
    for (int i = 0; i < total_size; ++i) {
        int x = i / HEIGHT;
        int y = i % HEIGHT;
        Complex c = {X_MIN + x * dx, Y_MIN + y * dy};

        // Compute mandelbrot set in the point
        int iter = mandelbrot(c);
        iter = iter % MAX_ITER;

        // store the value
        image[i] = iter;
    }
}
```

OpenMP

- `#pragma omp parallel for`:
  parallelizes the loop
- `schedule(dynamic, CHUNK_SIZE)`:
  divides the iterations in chunks of size `CHUNK_SIZE` and assigns them to threads as they become available.

Code Optimizations

The following C function `mandelbrot` computes the Mandelbrot set in a given point c.

```c
int mandelbrot(const Complex c) {
    int n = 0;
    Complex z = {0, 0};
    while ((z.x * z.x + z.y * z.y) < 4 && n < MAX_ITER){
        double temp = z.x * z.x - z.y * z.y + c.x;
        z.y = 2 * z.x * z.y + c.y;
        z.x = temp;
        ++n;
    }
    return n;
}
```

Code Optimizations

Let's cnosider a couple of optimizations:

- `const Complex c`: the `const` keyword makes the variable `c` read-only. However, this also enables the compiler to pass a pointer to the variable instead of copying it.
- Magnitude before than `MAX_ITER`: for most of the points the magnitude of z will be greater than 2 before reaching the maximum number of iterations. Once the first condition is false, we don't need to check the second one.
- `++n`: the pre-increment operator is more efficient than the post-increment one since it doesn't need to create a temporary variable.

Compiler's Optimizations

The **gcc** compiler has an optimization flag `-O{0,1,2,3}` which enables several optimizations. We compare the basic code with the level 3 optimized one by giving a look at the assembly code.

- Less memory accesses
- Better data alignment

Less Memory Accesses

```
movsd   -32(%rbp), %xmm1 # read c.x
movsd   -32(%rbp), %xmm0 # read c.x
mulsd   %xmm1, %xmm0     # c.x * c.x
movsd   -24(%rbp), %xmm2 # read c.y
movsd   -24(%rbp), %xmm1 # read c.y
mulsd   %xmm1, %xmm2     # c.y * c.y
movapd  %xmm0, %xmm1     # c.x * c.x
subsd   %xmm2, %xmm1     # c.x^2 - c.y^2
movsd   -48(%rbp), %xmm0 # read z.x
addsd   %xmm1, %xmm0     # z.x^2 - z.y^2 + c.x
movsd   %xmm0, -16(%rbp) # write z.x
movsd   -32(%rbp), %xmm0 # read c.x
movapd  %xmm0, %xmm1     # c.x
addsd   %xmm0, %xmm1     # c.x + c.x
movsd   -24(%rbp), %xmm0 # read c.y
mulsd   %xmm0, %xmm1     # c.x + c.x * c.y
movsd   -40(%rbp), %xmm0 # read z.y
addsd   %xmm1, %xmm0     # 2 * z.x * z.y + c.y
movsd   %xmm0, -24(%rbp) # write z.y
movsd   -16(%rbp), %xmm0 # read z.x
movsd   %xmm0, -32(%rbp) # write z.x
addl    $1, -4(%rbp)     # write ++n
```

All the memory accesses are unnecessary. The optimized code, uses exclusively the CPU registers. One could manually use `register`.

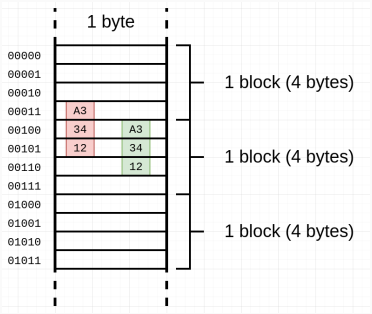
Basic	**Optimized**
70	20

Table 1: Memory accesses

ILP

$-$ pipeline stalls $\rightarrow +$ throughput.

Better Data Alignment



Example of data alignment with a 32-bit cpu and a cache line size of 32 bytes. When the data is aligned it can be mapped with just one cache line.

A memory address is said to be *n*-byte aligned if it is a multiple of *n*, where *n* is a power of 2.
Modern CPUs can access memory by a single memory word at a time, which is usually 64 bits, 128 bits, 256 bits.
The amd64 architecture provides sixteen 128-bit `xmm0-15` registers used for SIMD operations. Two doubles fit in a single register.
By aligning data (`p2align`), we ensure that `c.x` and `c.y` are stored in the same memory word.

Better Data Alignment consequences

Then, the `movapd` instruction, which can move two packed double-precision floating-point values at once, can be used.
Also, when can use the `mulpd` instruction to perform the two multiplications $c.x \cdot c.x$ and $c.y \cdot c.y$ at once.
Nowadays, all CPUs support SIMD instructions, enabling for CPU level parallelism.
Additionally, by aligning data, we optimize the cache usage mapping more data in the same cache line.

Results

With the `perf stat` command when can measure several metrics.

Metric	Basic	Optimized
Wall-clock time	40.53 s	16.09 s
User time	316.67 s	126.71 s
System time	199.23 ms	70.34 ms
Instructions	881 billions	466 billions
Instructions per cycle	1.11	1.29
Cache references	44.743.584	43.485.500
Cache misses	0.98%	0.34%

Intel core i7-8550U, 1.8 GHz, 4 cores, 8 threads, 16 GB RAM.
OMP threads: 8, image size: 2048×2048.

MPI

Message Passing Interface is a standard for parallel and distributed computing. Communication between processes must be explicitly defined.
Idea: we want to distribute the load in a dynamic way as the OpenMP code does.
Solution: we define a `MPI_CHUNK_SIZE` and we build a *work queue*. Each process will get a chunk from the queue and will compute it. A new chunk will be assigned to a process as it becomes available.
This is a simple, yet effective, way to implement the dynamic scheduler, which allows to better distribute the load among the processes.

MPI - Work Queue

The queue data structure, implemented in the `sys/queue.h` library, is built by the root process.

```c
// Create the work queue
work_queue = (WorkQueue *)malloc(sizeof(WorkQueue));
TAILQ_INIT(work_queue);

// Add work items to the queue
for (uint32_t i = 0; i < n_chunks; ++i){
    WorkItem *item = (WorkItem *)malloc(sizeof(WorkItem));
    item->start_idx = i * MPI_CHUNK_SIZE;
    item->end_idx = (i + 1) * MPI_CHUNK_SIZE;
    if (item->end_idx > total_size)
        item->end_idx = total_size;
    TAILQ_INSERT_TAIL(work_queue, item, entries);
}
```

MPI - Communication

We use blocking communication to send the chunks to the processes.

- `MPI_Send`: sends a message to a process
- `MPI_Recv`: receives a message from a process

A further optimization could be to use non-blocking communication to overlap computation and communication and reduce the waiting time. The root process sends only 2×4 bytes, but receive

$$2 \times 4 + \texttt{MPI\_CHUNK\_SIZE} \times \texttt{image\_t}$$

bytes, where `image_t` is either 1 or 2 bytes.

MPI - Communication

One could decide to use two “weak” processes, one for sending the chunks and one for receiving them. Moreover, using non-blocking communication reduce the waiting time.
The worker processes can receive a new chunk immediately after finishing the previous one. They don’t need to wait for:

- The root process to end its own computation
- The root process to copy the previous chunk
- The communication to be completed

Scaling

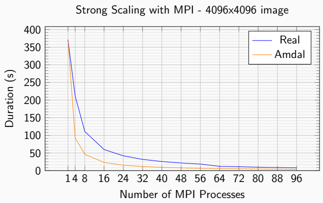
We want to measure the scalability of the code in two ways:

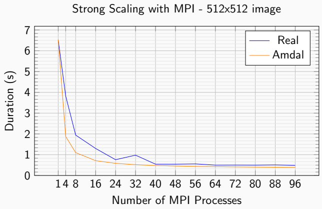
- **Strong scaling**: we fix the problem size and we increase the number of processes. We expect the execution time to decrease.
- **Weak scaling**: we increase the problem size proportionally to the number of processes. We expect the execution time to remain constant.

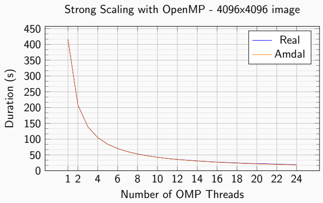
In both cases we perform the same test, once by fixing the number of MPI processes to 1 and varying the number of OpenMP threads, and once by fixing the number of OpenMP threads to 1 and varying the number of MPI processes.
Every test is repeated for two problem sizes: 512×512 and 4096×4096.

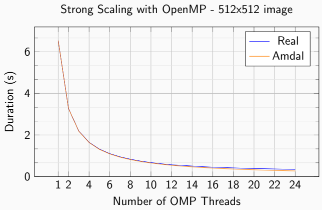
Strong Scaling

We compare the experimental data with the Amdal's law given $p = 1$.





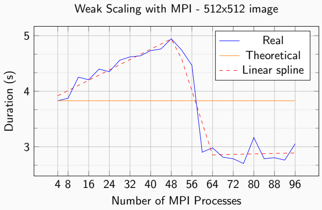


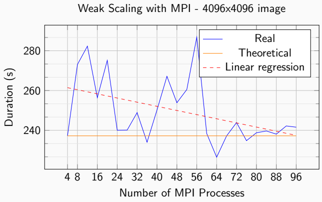


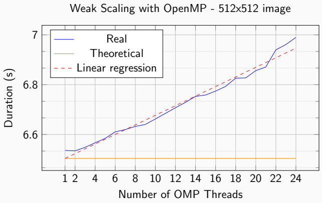
The OpenMP code scales better, touching the lower bound.

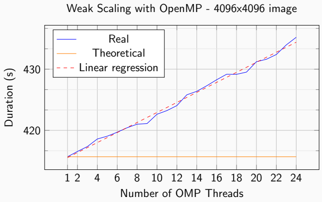
Weak Scaling

Theoretically, the execution time should remain constant.


Weak Scaling with MPI - 512x512 image
Duration (s)
Number of MPI Processes
Real
Theoretical
Linear spline


Weak Scaling with MPI - 4096x4096 image
Duration (s)
Number of MPI Processes
Real
Theoretical
Linear regression


Weak Scaling with OpenMP - 512x512 image
Duration (s)
Number of OMP Threads
Real
Theoretical
Linear regression


Weak Scaling with OpenMP - 4096x4096 image
Duration (s)
Number of OMP Threads
Real
Theoretical
Linear regression