Assignment 3: MPI

The goal is to distribute the program between the different nodes (in out case different processes on the same machine).

Hardware

For this first assignment, we executed the C code using the Software 2 (SW2) workstations, with the following characteristics.

```
S4825087@sw209:~$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          20
On-line CPU(s) list:             0-19
Thread(s) per core:              1
Core(s) per socket:              12
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           151
Model name:                      12th Gen Intel(R) Core(TM) i7-12700
Stepping:                        2
CPU MHz:                         2100.000
CPU max MHz:                     3876.9570
CPU min MHz:                     800.0000
BogoMIPS:                        4224.00
Virtualization:                  VT-x
L1d cache:                       288 KiB
L1i cache:                       192 KiB
L2 cache:                        7.5 MiB
NUMA node0 CPU(s):               0-19
Vulnerability Itlb multihit:     Not affected
Vulnerability L1tf:              Not affected
Vulnerability Mds:               Not affected
Vulnerability Meltdown:          Not affected
Vulnerability Mmio stale data:   Not affected
Vulnerability Retbleed:          Not affected
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Enhanced IBRS, IBPB conditional, RSB filling, PBRSB-eIBRS S
                                 W sequence
Vulnerability Srbds:             Not affected
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
                                  clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
                                 scp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology n
                                 onstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor
                                  ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_
                                 2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf
                                 _lm abm 3dnowprefetch cpuid_fault epb invpcid_single ssbd ibrs ibpb sti
                                 bp ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase
                                 tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt
                                 clwb intel_pt sha_ni xsaveopt xsavec xgetbv1 xsaves split_lock_detect a
                                 vx_vnni dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp h
                                 wp_pkg_req hfi umip pku ospke waitpkg gfni vaes vpclmulqdq tme rdpid mo
                                 vdiri movdir64b fsrm md_clear serialize pconfig arch_lbr ibt flush_l1d
                                 arch_capabilities
```

Algorithm analysis

Given the function

$$f(x) = \frac{4}{1+x^2}$$

It's proven that the integral in $[0, 1]$ is equal to π:

$$\int_0^1 f(x)\,dx = \pi$$

Through **midpoint Riemann sums**, this integral can be approximated by:

$$\pi \approx \frac{1}{n} \sum_{i=1}^{n} \frac{4}{1 + (\frac{i-0.5}{n})^2}$$

$(i - 0.5)/n$ is indeed the midpoint of the i-th subinterval. Since it discretizes the integral, it becomes an approximation of π.

Parallelization strategy

The sequential case is the following:

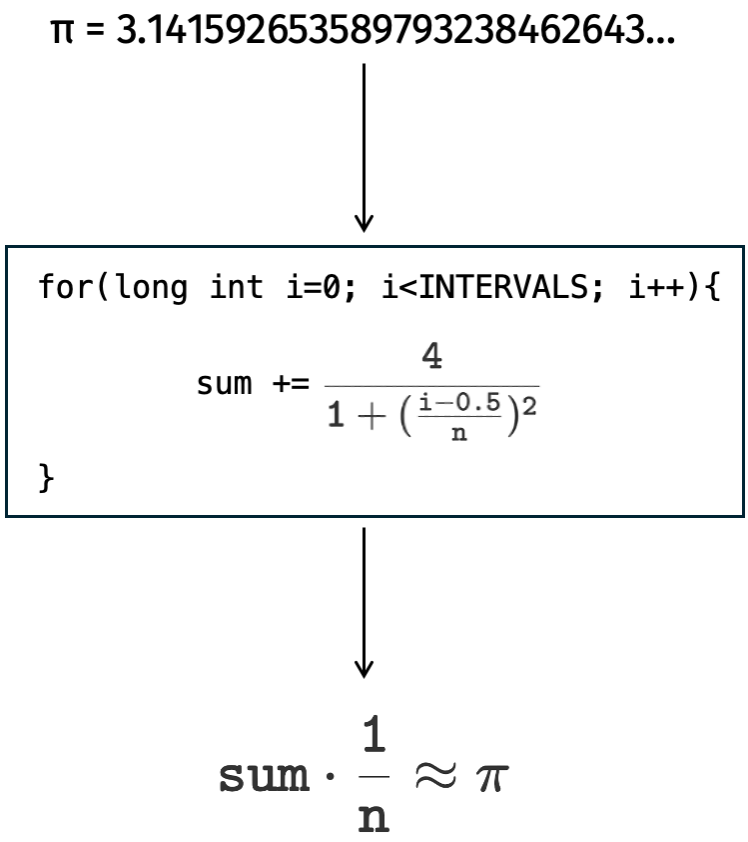


Figure 1: sequential case

As we can see, the sum is performed alltogether on the same node. The simplest yet most powerful way to parallelize this sum is to split the computations on different nodes, making them calculate only a chunk of the total sum.

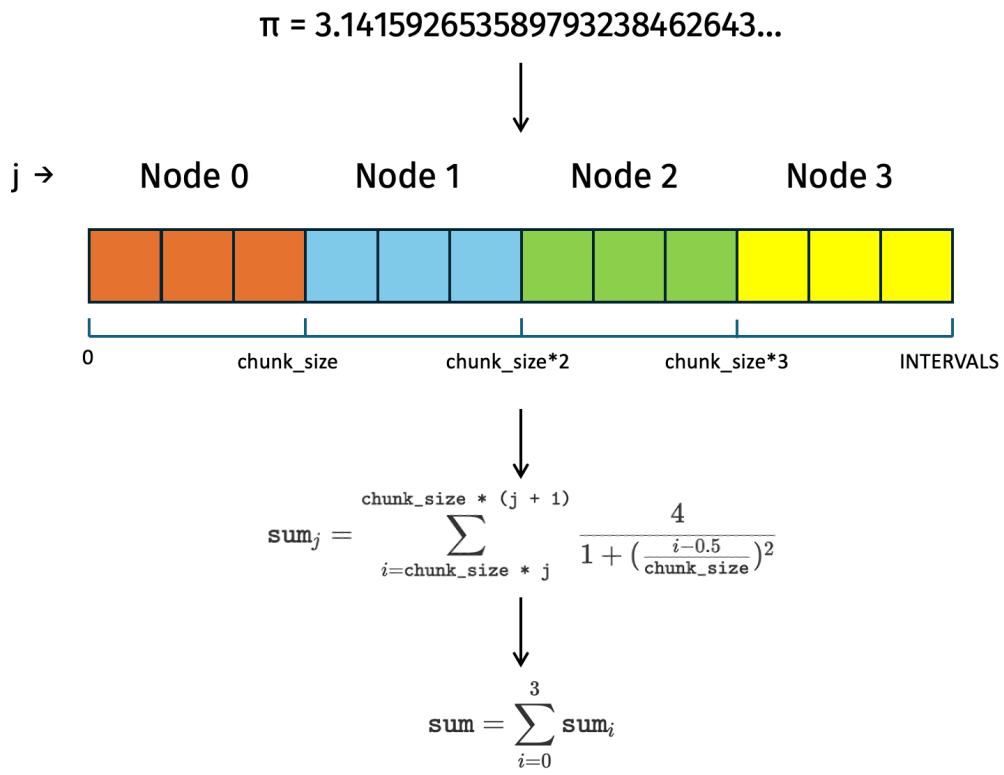


Figure 2: parallel case

The final step would be re-aggregate all the partial sums into the same global result, i.e. **reduce** them.

Workload distribution

Now that we know how the algorithm works, we can decide how to divide the workload in each MPI node. We start by noticing that it is a sum over n elements (in the code $n =$ `INTERVALS`), so, if we have m worker nodes with same resources and performances, we can divide this sum into $m/\texttt{INTERVALS}$ chunks

```
int size, rank;
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );

long int chunk_size = intervals / size;
```

and, for each node, determine the starting index and the ending one:

```
long int start = rank * chunk_size + 1;
long int end = (rank == size - 1) ? intervals : start + chunk_size - 1;
```

at this point, the local sum on the node is computed

```
double local_sum = 0.0;
double x, f;
double dx = 1 / (double)intervals; // 1 / n

for (long int i = start; i <= end; i++) {
  x = dx * ((double)(i - 0.5));
  f = 4.0 / (1.0 + x * x);
  local_sum += f;
}
```

At the end of the loop, the local sum on the node will be computed. After this loop, we reduce all the results by summing the partial sums on the master node (0 in our case), getting the final result.

```
double global_sum;
// (send_bf, recv_bf, n_elems, datatype_elems, mpi_op, receiver, comm)
MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, MASTER_NODE, MPI_COMM_WORLD);

if(rank == MASTER_NODE){
  double pi = dx * global_sum;
  printf("Computed PI %.24f\n", pi);
  printf("The true PI %.24f\n\n", PI25DT);
}
```

Program optimizations

Vectorization

Now that the program is distributed, we can start thinking at the other optimization aspects. The first thing to tune is the compiler, in our case we used `mpiicx` with all the optimization flag needed. The command ran to compile the program is

```
mpiicx -g -O3 -xHost -qopenmp -qopt-report=3 -ffast-math pi_homework.c
```

while the one to execute it is

```
mpirun -np 10 ./pi_homework
```

Parallelization

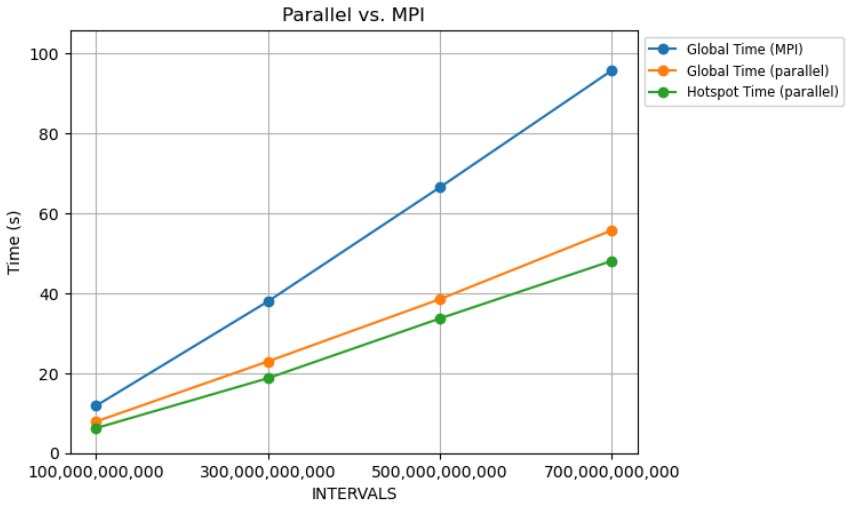
Another way to improve the performances is to use multithreading. We used **OpenMP** to parallelize the MPI code in this way:

```
#pragma omp parallel for num_threads(NTHREADS) private(x, f) reduction(+ : local_sum)
for (i = start; i <= end; i++) {
  x = dx * ((double)(i - 0.5));
  f = 4.0 / (1.0 + x * x);
  local_sum += f;
}
```

That is, simply subdivide the for loop on different threads and apply the reduction on the sum.

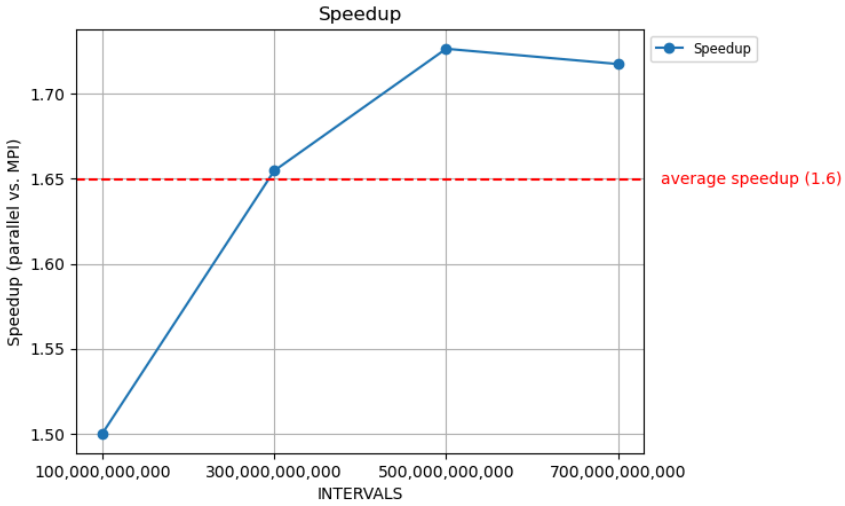
Performance evaluation

In this program, the only heavy hotspot is `loop in main at pi_homework.c:26` , that is the one that computes the local sums. The following measurements are taken considering the global execution time and the hotspot execution time (since it's only one).



execution time comparison

In general, with multithreading we avoid introducing useless overheads due to process creation



Speedup (parallel vs. MPI)

Because of this, we see that the parallel program is 1.6 times faster than the MPI program.

Conclusions

The main concept to reason on in this case is the scalability. Multithreading is useful and fast, but, in the best case, we can have no more than ≈ 64 threads in a single machine, while we can add potentially infinite nodes to an MPI cluster.