Introduction to Parallel Computing
Overview of methods

Fabian Jakub[1], Oriol Tinto-Prims[1], Robert Redl[1]

[1]Meteorologisches Institut München, Ludwig-Maximilians-Universität München

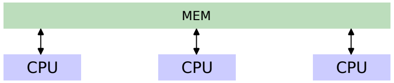
Overview

Topics we will touch

- Message Passing, e.g. MPI
- Shared memory parallelization, e.g. OpenMP
- Cluster environment, here SLURM
- How to tap into parallel resources from within Python

Message Passing vs Shared memory parallelization

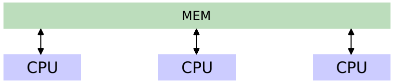
Shared memory parallelization, e.g. OpenMP



- Spawn multiple threads that act on global memory in parallel
- Parallelization bound to single node (computer)

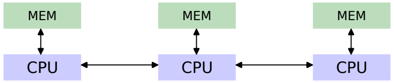
Message Passing vs Shared memory parallelization

Shared memory parallelization, e.g. OpenMP



- Spawn multiple threads that act on global memory in parallel
- Parallelization bound to single node (computer)

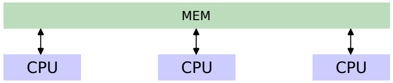
Message Passing, e.g. MPI



- Spawn multiple processes, each with own address space
- If data exchange needs to happen, send it explicitly
- Parallelization can be over any number of nodes as long as they are able to communicate

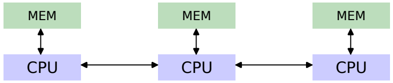
Message Passing vs Shared memory parallelization

Shared memory parallelization, e.g. OpenMP



- Spawn multiple threads that act on global memory in parallel
- Parallelization bound to single node (computer)

Message Passing, e.g. MPI



- Spawn multiple processes, each with own address space
- If data exchange needs to happen, send it explicitly
- Parallelization can be over any number of nodes as long as they are able to communicate

Others:

- OpenACC (like OpenMP but for GPU's and accelerators)
- CUDA / OpenCL (GPU's)
- Pthreads, Co-array Fortran
- ZeroMQ, Celery etc. (message passing queues, higher latency, fault tolerance, ...)

What is MPI?

MPI — the Message Passing Interface is a library

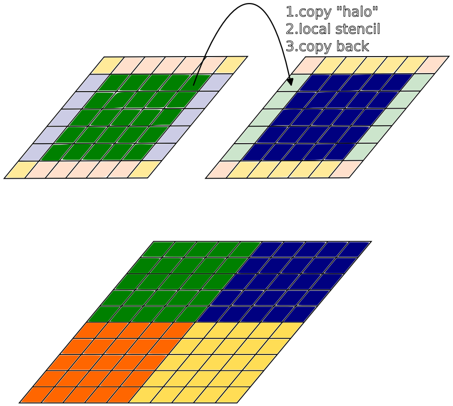
- Industry standard since 1991
- Easy but rather low-level API
- Nothing is shared, if you need something from your neighbor, send it explicitly
- MPI standards 2 and 3 implement shared memory parallelism and one sided communication

MPI Hello World!

```
from mpi4py import MPI
import numpy as np
comm = MPI.COMM_WORLD                                         # default communicator for all processes
rank = comm.Get_rank()                                        # get the integer id of this rank
numranks = comm.Get_size()                                    # get the number of ranks in the communicator

N = 6

if rank == 0:                                                 # if I am the first
    workload = np.arange(N).reshape(numranks, -1)             # generate a list of work to for each rank

localtasks = comm.scatter(workload, root=0)                   # communicate (scatter) from 0 to all

print(f"Hi, I am rank {rank} out of {numranks}",              # see who wants to do what
      f" and will work on problems {localtasks}")
```

```
$> mpirun -np 2 python mpi_example.py
  Hi, I am rank 0 out of 2 and will work on problems [0 1 2]
  Hi, I am rank 1 out of 2 and will work on problems [3 4 5]

$> mpirun -np 3 python mpi_example.py
  Hi, I am rank 1 out of 3 and will work on problems [2 3]
  Hi, I am rank 2 out of 3 and will work on problems [4 5]
  Hi, I am rank 0 out of 3 and will work on problems [0 1]
```

MPI in NWP



MPI parallelization scheme in NWP models:

- split domain (in NWP usually 2D)
- exchange borders to update ”halo” regions
- overlap width depends on stencil size
- run local computations

What is OpenMP?

OpenMP — Open MultiProcessing is a compiler extension

- Basically all compilers support it
- Using comment lines in code (called *pragmas*) to steer thread creation and data movement
- available in Fortran, C, C++
- Newer OpenMP standards target GPU's and accelerators

Process vs. Thread?

- Starting a program with a single process
- Each process has its own address space
- A process can spawn one or multiple threads
- Threads share address space but have unique instruction and stack pointers

OpenMP Hello World!

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

int main() {

  #pragma omp parallel                // generate parallel environment
  {
    const int myid = omp_get_thread_num();
    const int numthreads = omp_get_num_threads();
    fprintf(stderr, "Hi, I am thread %d of %d\n", myid, numthreads);
  }
  return 0;
}
```

```
$> gcc -Wall -fopenmp openmp.c && OMP_NUM_THREADS=3 ./a.out
  Hi, I am thread 0 of 3
  Hi, I am thread 2 of 3
  Hi, I am thread 1 of 3
```

OpenMP CumSum!

```
const int N=1000;                       // lets compute a cumulative sum

int i;
int serial_sum = 0;
for (i=0; i<N; ++i) serial_sum += i;

int threaded_sum = 0;
#pragma omp parallel for              // parallel for loop, barrier afterwards
for (i=0; i<N; ++i) {
    threaded_sum += i;
}

fprintf(stderr, "cumsum(%d) = %d (expected %d)\n", N, threaded_sum, serial_sum);
```

```
$> gcc -Wall -fopenmp openmp.c && OMP_NUM_THREADS=1 ./a.out
  cumsum(1000) = 499500 (expected 499500)

$> gcc -Wall -fopenmp openmp.c && OMP_NUM_THREADS=10 ./a.out
  cumsum(1000) = 86371 (expected 499500)
```

OpenMP CumSum!

```
const int N=1000;               // lets compute a cumulative sum

int i;
int serial_sum = 0;
for (i=0; i<N; ++i) serial_sum += i;

threaded_sum = 0;
#pragma omp parallel for private(i) reduction(+: threaded_sum)
for (i=0; i<N; ++i) {
    threaded_sum += i;
}

fprintf(stderr, "cumsum(%d) = %d (expected %d)\n", N, threaded_sum, serial_sum);
```

```
$> gcc -Wall -fopenmp openmp.c && OMP_NUM_THREADS=1 ./a.out
  cumsum(1000) = 499500 (expected 499500)

$> gcc -Wall -fopenmp openmp.c && OMP_NUM_THREADS=10 ./a.out
  cumsum(1000) = 499500 (expected 499500)
```

OpenMP vs. MPI

Which one to use is very dependent on your situation!

- OpenMP easier to introduce in established codebase
- MPI scales across nodes but needs some thought beforehand

And not exclusive — Hybrid OpenMP & MPI

- OpenMP for intra node parallelization and or GPU's
- MPI across nodes

OpenMP vs. MPI

Which one to use is very dependent on your situation!

- OpenMP easier to introduce in established codebase
- MPI scales across nodes but needs some thought beforehand

And not exclusive — Hybrid OpenMP & MPI

- OpenMP for intra node parallelization and or GPU's
- MPI across nodes

My personal opinion:

- OpenMP: race conditions hard to debug
- OpenMP: getting good scaling wrt to memory access is wicked difficult
- MPI-2 has shared mem support
- Message Passing as compute model is <u>far</u> easier to understand
- I have never seend speed gains from OpenMP over MPI

Threading in Python

Python threads are not OS level threads!

- Concurrency vs. Parallelism
    - concurrency allows context switches if process is waiting for external resources (e.g. yield co-routine waiting for network response)
    - in contrast parallelism is doing things at the same time
- Python threads only allow concurrency because of the Global Interpreter Lock (GIL)

More often you probably want true (in Python process based) parallelism

- multiprocessing
- concurrent.futures
- joblib
- dask
- ipyparallel

Python ProcessPoolExecutor

```python
from concurrent.futures import ProcessPoolExecutor
import os

def do_the_hard_work(data):
    return sum([ i**0.12345 for i in range(data) ])

work = range(10000)

with ProcessPoolExecutor(int(os.environ['OMP_NUM_THREADS'])) as pool:
    output = pool.map(do_the_hard_work, work)

total = sum(output)
print(f'Sum: {total}')
```

```
$> OMP_NUM_THREADS=1 time python3 pool_example.py
  0:06.93 elapsed

$> OMP_NUM_THREADS=10 time python3 pool_example.py
  0:02.24 elapsed
```

Submitting jobs to the cluster

run a job interactively on the cluster:

```
$> srun
  -N --nodes=1          # number of nodes (computers) the job should be distributed (MPI)
  -n --ntasks=1         # number of MPI ranks (MPI)
  -c --cpus-per-task=4  # number of CPU's per task (sets OMP_NUM_THREADS)
  --mem=4G              # memory needs per node
  --mem-per-cpu=1G      # memory needs per CPU
  <script/binary>       # program to run
  <commandline args>    # options to give to the program
```

Submitting jobs to the cluster

alternatively for long running jobs:

```
$> cat > myjob.slurm << EOF
#!/bin/bash
#SBATCH -o myjob.%j.log
#SBATCH --mail-type=fail
#SBATCH --mail-user=Fabian.Jakub@physik.uni-muenchen.de
#SBATCH --time=24:00:00
#SBATCH --mem=15G
#SBATCH -n 256
#SBATCH -N 1-24
module load spack gcc openmpi
srun mybinary --foo=bar
EOF

$> sbatch myjob.slurm  # submit job script to cluster
```

Bash parallel execution

```
MODELBINARY=sleep
MPIEXEC=srun
JOBS="1 2 3"

pids=()
jobnr=0
for J in $JOBS; do
     ($MPIEXEC $MODELBINARY $J) &
     pids[${jobnr}]=$!
     jobnr=$(($jobnr +1))
done

# allow hitting CTRL-C to cancel the running background jobs
trap 'for pid in ${pids[*]}; do echo "kill $pid"; kill ${pid}; done' EXIT

# wait for all background processes before proceeding
for pid in ${pids[*]}; do echo "waiting for job $pid to finish"; wait $pid; done

echo "Finished all jobs"
```

Recap

We had a very brief overview on:

- Message Passing, e.g. MPI
- Shared memory parallelization, e.g. OpenMP
- How to tap into parallel resources from within Python
- Cluster environment, here SLURM

There ain't no such thing as a free lunch!

- Choosing a feasible approach is important but not always easy.
- Before you try something it is always worthwhile to ask colleagues.