MPI + x

Ramses van Zon

PHY1610H 2025 Winter

Why + x?

MPI is scalable and portable.

Why would we want make things more complicated with + x?

1. Communication cost:
   The more processes, the more communication:

   Want less processes, but still use all cores:

   - Reduce the number of processes
   - Increasing the thread count per process

   That requires MPI + Multicore
   Usually: MPI + OpenMP

2. Memory overhead
   The more processes, the more communication.

3. Unused computing hardware:
   You may have compute accelerators that you weren't using.

   Usually, GPUs.

   This needs specialize programming:
   MPI + CUDA
   or
   MPI + OpenMP GPU offloading

MPI + Multicore

Hybrid MPI+OpenMP: Coding

This can be beneficial: pure MPI requires more communications and more memory

As far as coding is involved, that's easy: use MPI calls and OpenMP directives.

Usually, the MPI part is the trickiest: do that first.

But: have to initialize MPI differently, instead of `MPI_Init`, use `MPI_Init_thread`:

```
int required = SOMETHING;
int provided;

MPI_Init_thread(&argc,&argv,required,&provided);

if (provided < required) exit(1);
```

Here, SOMETHING can be:

- `MPI_THREAD_SINGLE`
  Only one thread will execute.
- `MPI_THREAD_FUNNELED`
  Only the thread that called MPI_Init_thread will make MPI calls.
- `MPI_THREAD_SERIALIZED`
  Only one thread will make MPI library calls at one time.
- `MPI_THREAD_MULTIPLE`
  Multiple threads may call MPI at once with no restrictions.

Hybrid MPI+OpenMP: Running

You must be specific about the numbers to avoid overloading cores.

In scheduled jobs

The scheduler can help in this respect. E.g. with SLURM, with 16-core nodes, you can say

```
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=20
#SBATCH --time=1:00:00
module load gcc openmpi
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
mpirun ./hybridcode  # can use srun instead of mpirun too.
```

This gets 6 mpi processes spread over 3 nodes, each running 20 threads.

On login nodes or your own machine

E.g. to get 4 mpi processes on the node each running 3 threads, you'd do

```
$ module load gcc openmpi
$ export OMP_NUM_THREADS=3
$ mpirun -n 4 ./hybridcode
```

Hybrid: Which mix of MPI/OpenMP is best?

There are many, many aspects which factor into this decision.

While it's true that many applications get their best performance from running in hybrid mode, the precise balance of threads and processes is not always the same.

Hybrid: Memory consideration

- Every process has its own memory.
- Even if data is distributed, each process has to have the executable loaded.
- and there will be additional ghost cells
- MPI will use internal buffers.

This would suggest using one process per node, and threads for the rest.

But:

- The memory of the MPI processes tends to be closer to the cores.
- No cache coherency slowdowns, like in OpenMP.
- Less chance of race conditions.

Hybrid: CPU considerations

- Both processes and threads will be assigned to different cores on the CPU by the OS.
- That is, as long as there are enough cores.
- If you underutilize cores, the OS may move your process.
  But it does not move the memory with it!
- If your MPI computation keeps the cores busy, i.e., it's pure MPI, the OS won't see a reason to move them.
- In OpenMP, there are always serial portions, and the chance of "thread migration" is real.
- In OpenMP, there are always serial portions, but in MPI, processes may be waiting for communication or syncronization.

Nodes may have several CPUs in different sockets, e.g, the Teach cluster has 2 sockets with each an 20-core CPU. These have their own caches and different parts of main memory that are closer to each socket.

Binding

When running in hybrid mode, consider **pinning** a.k.a. **binding** processes and threads to specific cores.

OpenMP

There's a few environment variables that control the binding of OpenMP Threads:

- `OMP_PROC_BIND=true` tells openmp to perform binding of threads.
- `OMP_PLACES=X` where X can be `cores`, `threads` or `sockets`, or a list of core numbers.

MPI

Binding is done with options to mpirun, but these differ per MPI implementation. For openmpi:

- `--map-by X`, where X is hwthread, core, L1cache, L2cache, L3cache, socket, numa, or, board.
- `--report-bindings` option to allows check the bindings

https://docs.open-mpi.org/en/v5.0.x/man-openmpi/man1/mpirun.1.html

Slurm

Some MPI implementations are integrated enough with the scheduler that it will "do the right thing" by default, if you set `ntasks_per_node`. Still, be explicit if you can.

Hybrid: Communication considerations

Network

Often, nodes have one connection to the network.

If you have multiple MPI processes on a node, they share this connection.

Less processes may mean less communication, and less communication buffers, which can help.

Hybrid: I/O considerations

If the file system is a network file system, it has the same limitations.

If the file system is a single disk in the nodes, having several processes write at once can slow things down.

But network file systems are often parallel, when using multple nodes.

MPI can help here.

You can use MPI to do IO in parallel

- I/O is often the slowest part of a computing system.
- Large HPC installations have parallel file systems to help
- These have many disks on the back-end, enabling parallel reading and writing
- As with many parallel technique, parallelization is not automatic

Solutions:

- Could use a separate file for each process.
  - But now output depends on #processes.
  - Can lead to directory locking.
- MPI-IO: Sub-library that enables binary parallel file I/O to single files from all processes.
- HDF5 and NetCDF also allow parallel I/O if those libraries were built to support it.

MPI-IO is similar to ordinary files

```
MPI_Offset offset = (msgsize*rank);

MPI_File  file;
MPI_Status stat;

MPI_File_open(MPI_COMM_WORLD, "helloworld.txt",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &file);

MPI_File_seek(file, offset, MPI_SEEK_SET);
MPI_File_write(file, msg, msgsize, MPI_CHAR, &stat);
MPI_File_close(&file);
```

You have to control the data layout and what process gets to write where in the file!

One usually creates a so-called 'File view' to help with that.

Another Example

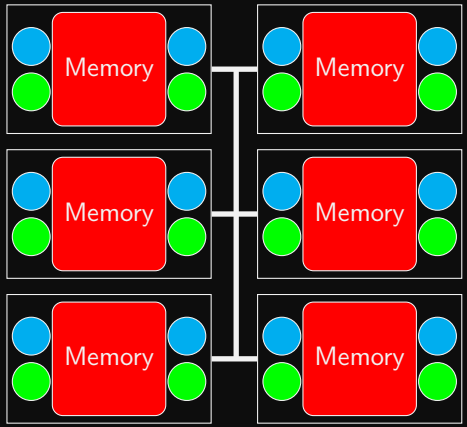
```
MPI_Offset offset = (msgsize*rank);

MPI_File file;
MPI_Status stat;

MPI_File_open(MPI_COMM_WORLD, "helloworld.txt",
                  MPI_MODE_CREATE | MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &file);



// Collective Coordinated Write
MPI_File_write_at_all(file, offset, msg, msgsize, MPI_CHAR, &stat);

MPI_File_close(&file);
```

Here, MPI-IO is similar to MPI collectives.

MPI + Accelerators

Hybrid architectures with accelerators

- Multicore nodes linked together with an (high-speed) interconnect.
- Nodes also contain one or more accelerators, usually GPUs.
- These are specialized, super-threaded (500-2000+) processors.
- GPUs have their own, limited, shared memory.
- Specialized programming languages, CUDA, OpenCL, OpenACC, OpenMP.
- Can be mixed with MPI, OpenMP.



Heterogeneous Computing

What is it?

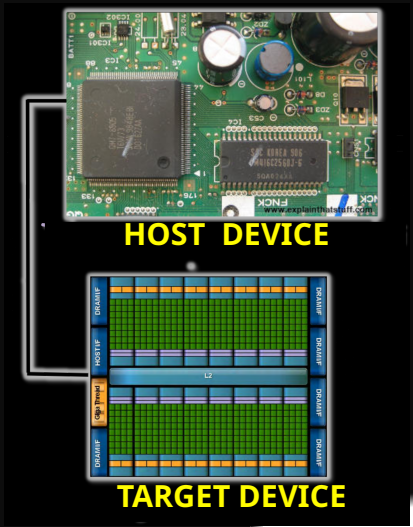
- Use different compute device(s) concurrently in the same computation.
- Example: Leverage CPUs for generatl computing components and GPUs for data parallel and floating point intensive components.
- Pros: GPUs are faster and cheaper ($/FLOP/Watt) for compute
- Cons: More complicated to program, only benefits certain applications.

Terminology

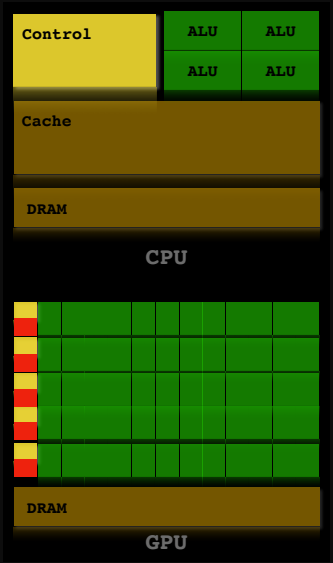
- GPGPU Programming: General Purpose Graphics Processing Unit Programming
- HOST: CPU and its memory
- DEVICE: Accelerator (GPU) and its memory

Accelerators

- Systems with accelerators are machines which contain an "off-host" accelerator, such as a GPU.
- These accelerator devices are very fast and good at massively parallel processing (having 500-2000+ cores).
- Complicated to program.
- Programming model: CUDA, OpenACC, OpenMP offloading, and OpenCL.
- Needs to be combine with at least some 'host' code: **heterogeous computing**.



Accelerators: CPUs vs GPUs



CPU

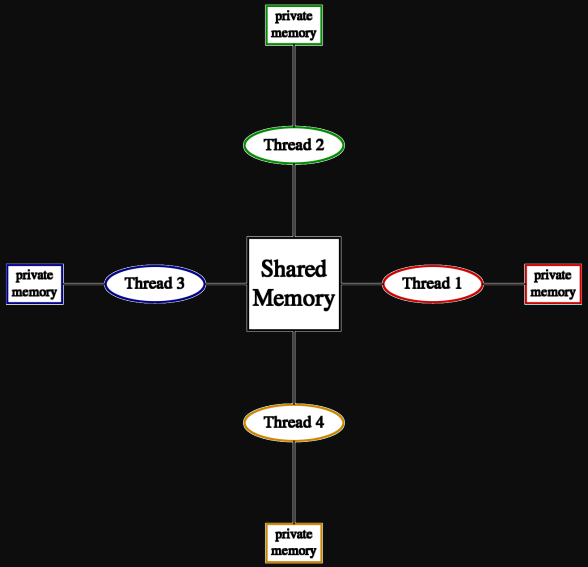
- general purpose
- task parallelism (diverse tasks)
- maximize serial performance
- large cache
- multi-threaded (4-16)
- some Single-Instruction-Multiple-Data (SIMD)

GPU

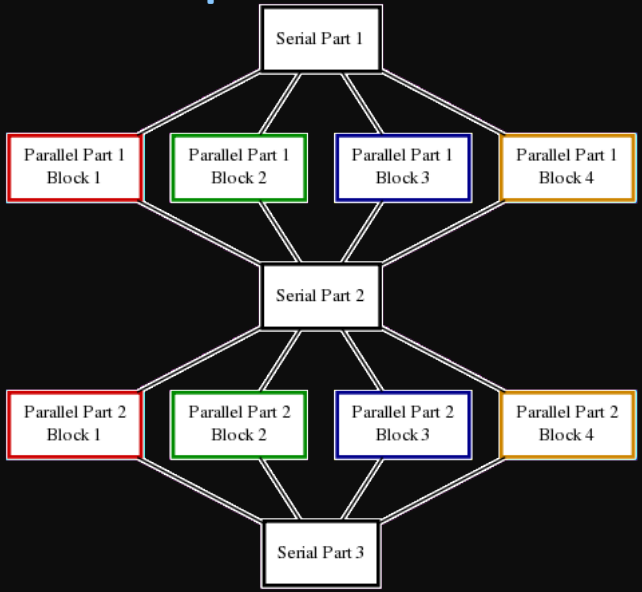
- data parallelism (single task)
- maximize throughput
- small cache
- super-threaded (500-2000+)
- "streaming multiprocessors" (SMs)
- almost all SIMD

Programming Accelerators with OpenMP

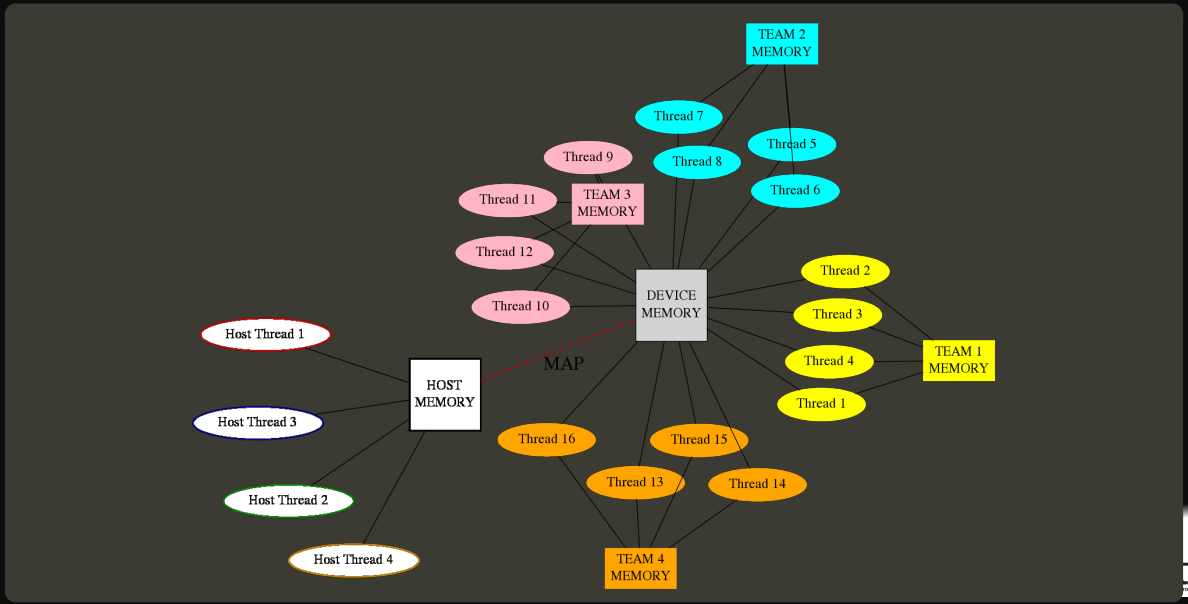
Memory Model in OpenMP (3.1)



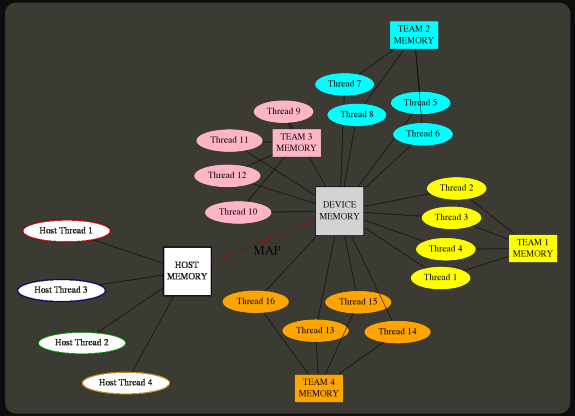
Execution Model in OpenMP



Memory Model in OpenMP 4+



Memory Model in OpenMP 4+



- Device has its own data environment
- And its own shared memory
- Threads can be bundled in a teams of threads
- These threads can have memory shared among threads of the same team
- Whether this is beneficial depends on the memory architecture of the device.

Next time

How to program GPUs with OpenMP

If you haven't yet, take some minutes to complete the **course evaluation**!