Introduction to HPC

Lecture 7

Jakub Gałecki

Agenda

- Miscellaneous parallelism-related tidbits
- Sharing computational resources with `slurm`
- Managing the HPC software stack – `spack`
- Exercise – optimizing a piece of FEM code

Parallelism tidbits

Simultaneous multi-threading

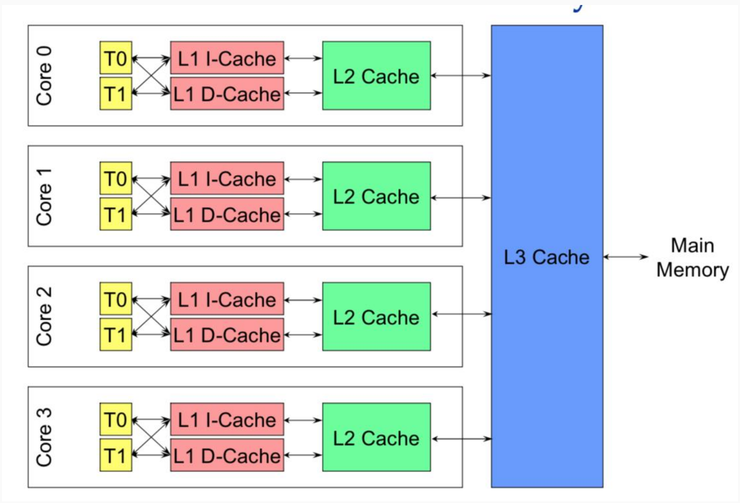
One *physical* core split into multiple (usually 2) *logical* cores

Logical cores share the resources of the underlying physical core: cache and execution units

Pros	Cons
Energy efficiency	Potential resource conflicts
Hides stalls	Reliance on OS scheduler
Better execution unit saturation	

Let's once again take a look at a CPU diagram to better understand how it works...

Private and shared cache



Parallel patterns

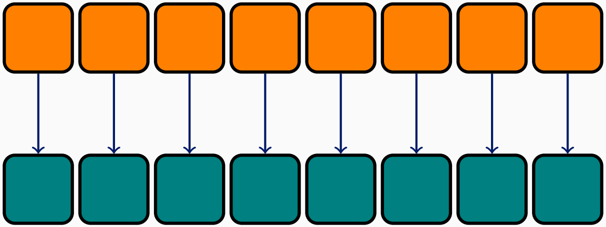
There are 3 classes of data access patterns (algorithms) we should be aware of:

- map
- reduction
- scan

We should know how to identify them, and take advantage of their parallelized implementation in whatever parallel library we're using.

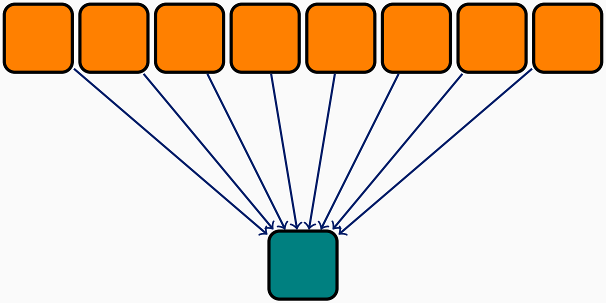
Map

A map is just a `for` loop over the data. This is the simplest pattern to parallelize, as there is no inherent dependency between iterations.



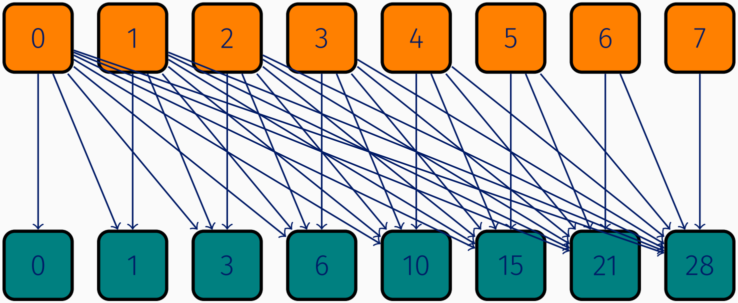
Reduction

A reduction combines multiple elements into a single value.
This requires synchronization between regions.

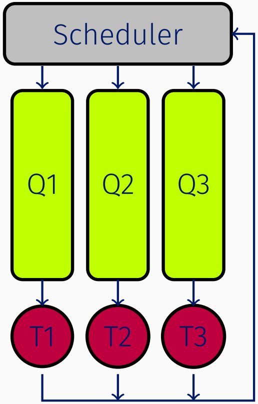


Scan

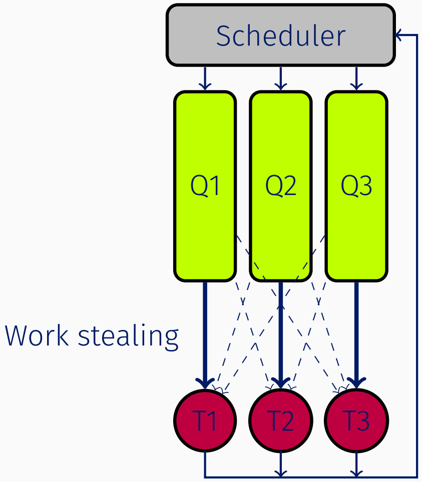
A scan propagates data such that the *i*-th iteration depends on the result of iteration $i - 1$. A good example is computing the vector of partial sums. This can still be done in parallel!



Static scheduler



Dynamic scheduler



Sharing resources with `slurm`

Background

Computational clusters are big.

In fact, they are so big, that many different people will use them simultaneously.

Furthermore, due to sub-linear scaling, it is usually more efficient to run many small jobs (e.g. simulations) than one large job.

This section will touch on how to efficiently share these computational resources and how to be a good HPC citizen.

Manually launching MPI jobs

Launching MPI jobs directly via `mpiexec`/`mpirun` can be tricky, and it requires that we know something about the topology of our machine (and that MPI knows it too)

```
mpiexec -n 4 my_program
```

```
mpiexec -n 2 --map-by ppr:1:socket my_program
```

Ideally, we'd like an abstraction layer which launches and manages MPI jobs for us, instead of doing it manually

slurm

`slurm` is a popular cluster management and job scheduling system (which manages the computing resources you've been given access to). We will focus on the job scheduling part. `slurm` can help us answer the following questions:

- How big is the cluster?
- What is the current workload?
- Who is currently using the machine?
- What are the currently available resources?

Most importantly, it will allow us to run our code on the number of nodes that we require, or queue our job for execution at a later time.

Terminology

Job – unit of work, a set of commands to be executed.

Interactive job – a job where we actively log onto a node and type commands into the terminal in real time.

Batch job – a job where we submit a script for execution without any further user input.

Partition – A subset of the computational resources (nodes). Clusters can be partitioned based on the specs of the particular nodes, e.g. one partition for CPU nodes and one for GPU nodes.

Allocation – a number of nodes (or cores) assigned to us for use (possibly for a limited time)

Useful commands

`sinfo` – view cluster info, e.g., number of nodes, partition info

`squeue` – view executing and scheduled jobs

`salloc` – request allocation

`srun` – run parallel job

`sbatch` – submit script for execution (batch job)

Note: you should call `srun` from a script submitted via `sbatch` to avoid directly invoking MPI

More info: `https://slurm.schedmd.com/man_index.html`

Example `slurm` options

- `--job-name` - name your job
- `--time` - set a deadline
- `--ntasks` - set number of MPI ranks (not recommended)
- `--nodes` - request specific number of nodes
- `--ntasks-per-node`, `--ntasks-per-socket`, `--ntasks-per-core`
- `--exclusive` - request exclusive access to nodes
- `--chdir` - specify working directory
- `--output` - redirect output to file

Model workflow

1. Copy code/data to scratch space
2. Request allocation/submit batch script
3. Execute the job
4. Copy data from scratch to permanent storage
5. Clean up scratch

Some etiquette:

- Name your jobs reasonably
- Clean up scratch space!
- Give reasonable timeouts
- Use only as much as you need

Building libraries & managing dependencies with `spack`

C++ library

A library is a distributed piece of software, intended for use via its API. Libraries can consist of:

- Header files
- Statically/dynamically linked binary files (libraries)
- Build files (e.g. CMake)
- Tests
- Documentation
- License
- ...

They can be open-source (the user can build them from scratch) or closed-source (pre-compiled binaries, source code is “secret”). We will focus on open-source projects.

Building a library

Usually, we need to perform the following steps:

1. Download the source (`git clone` or download a tarball)
2. Configure
3. Build
4. Test
5. Install
6. Make discoverable*
7. Use as a dependency for our project

For complex projects, this may not be straightforward...

Common installation script

To install the library

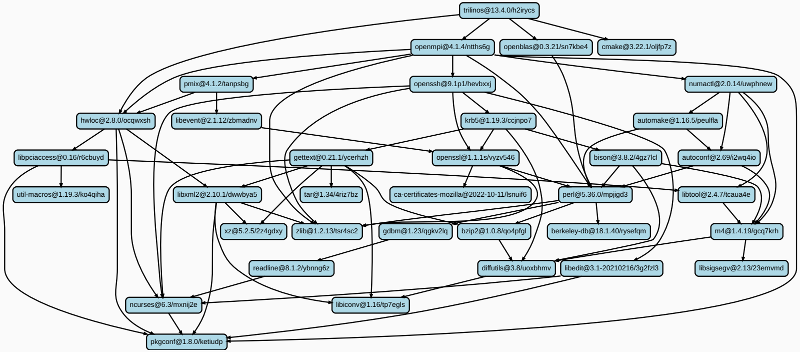
```
git clone [project repo url] && cd repo-name
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
make tests
make install
```

To use the library, add the following to your `CMakeLists.txt` file

```
find_package(lib)
target_link_libraries(my_target lib::lib)
```

If you're lucky...

The problem with managing software



Example dependency DAG

The problem with managing software

We need to install every dependency, and also their dependencies, and their dependencies...

We would like to be able to configure the installation

We would like to be able to install multiple versions of the same library

We would like to be able to use different compilers

How can we automate this?

spack

`spack` is a package manager, developed with HPC in mind (though now used more broadly)

No installation required

It **builds** the world from scratch, meaning it **builds** the dependencies recursively, starting only from a few elementary tools and Python

Supports multiple versions (variants) of a single library

Lots of awesome features, but also very simple:

```
spack install trilinos
```

spack

Some useful commands:

- `install` – installs package
- `info` – lists package info, including available options
- `load` – loads installed package (avoids conflicts)
- `list` – query available packages
- `find` – query installed packages
- `env` – manage environments

More: `https://spack.readthedocs.io/en/latest/`

Spack: variants

In Spack parlance, “variant” refers to a concrete configuration of a library, which includes:

- The version
- The compiler which was used to build it
- The options passed to the installation

Multiple variants can coexist simultaneously

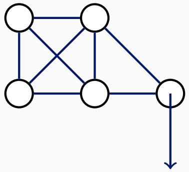
Example:

```
trilinos@13.4.0^gcc@12.1.0 +rol +openmp stdcxx=17
```

Q&A

Exercise: matrix-free FEM operator evaluation

FEM: truss

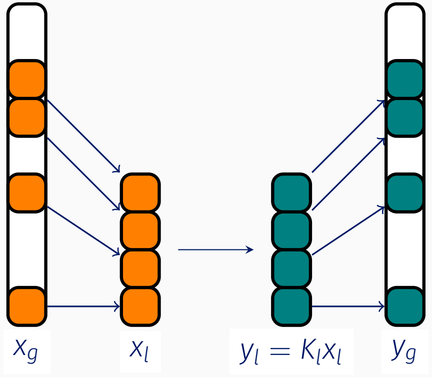


Element

$$K_{local} = \frac{AE}{L}\begin{bmatrix} c^2 & cs & -c^2 & -cs \\ cs & s^2 & -cs & -s^2 \\ -c^2 & -cs & c^2 & cs \\ -cs & -s^2 & cs & s^2 \end{bmatrix}, \; s = \frac{y_2 - y_1}{L}, \; c = \frac{x_2 - x_1}{L}$$

Matrix-free FEM operator

The evaluation of a matrix-free $y = Ax$ operator follows the gather-scatter pattern:


x_g
x_l
$y_l = K_l x_l$
y_g