Kavli IPMU High-Performance Computing Tutorial

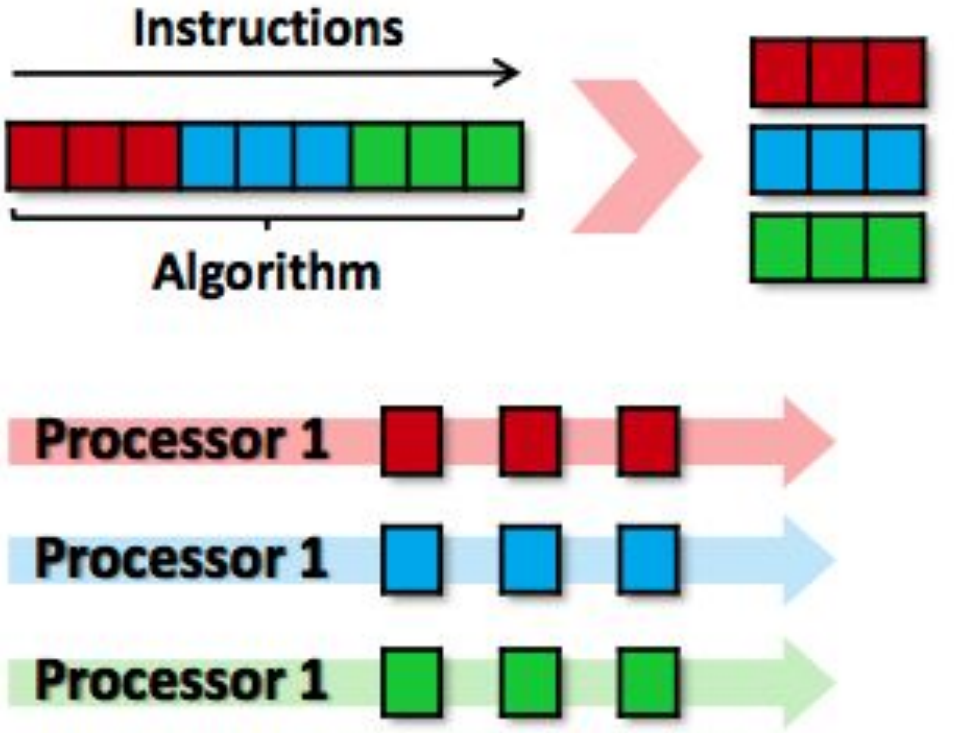
Connor Bottrell and Youngsoo Park
6/24/2021

Serial and parallel computing

Serial and Parallel Computing

Serial tasks

- A list of commands performed sequentially.
- All commands are carried out on a single CPU (core/processor).
- Multiple tasks (job) using the same commands are also carried out sequentially.
- Generally difficult to break up an individual serial task (colour in the right plot).
- Very efficient for small operations.



https://skirt.ugent.be/skirt8/_parallel_computing.html

Example of a set (**seri**es) of tasks (colours) carried out in **seri**al. Each task comprises a set of instructions which are performed sequentially (blocks).

Serial and Parallel Computing

Example of a serial task

Write a file whose whose contents are determined by some (set of) input argument(s).

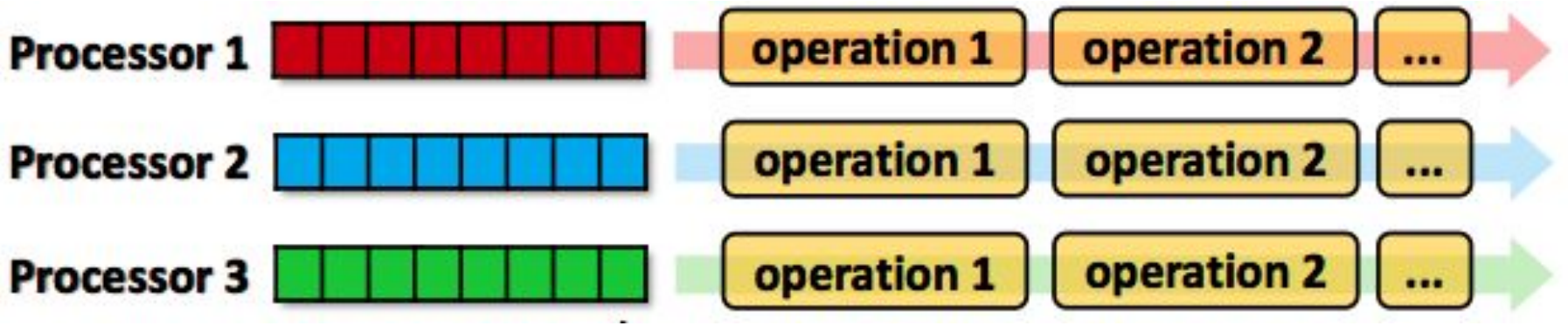
In the example to the right, **the serial task is the code inside the function** `serial_task`.

The function uses `idx` to determine the name of the output file and its contents.

In this example, each serial task takes ~0.5 s. But what if each took 1 min? 1 hour? 1 day?

```
1  # Example Serial Job
2  import time
3
4  # Serial task
5  def serial_task(idx):
6      time.sleep(0.5)
7      filename = f'Serial.{idx}.txt'
8      with open(filename,'w') as f:
9          for i in range(idx+1):
10             f.write(f'{i**2}\n')
11     return
12
13 def main():
14     # Main code
15     start = time.time()
16     for i in range(16):
17         serial_task(i)
18     runtime = time.time()-start
19     print(f'Time: {runtime}s')
20
21 if __name__=='__main__':
22     main()
```

Serial and Parallel Computing



https://skirt.ugent.be/skirt8/_parallel_computing.html

Example of a series of tasks (colours) carried out in parallel. Each task is independent and assigned to a different processor. Alternatively, each block may represent a chunk of a large dataset.

Parallel job

- Parallelization maps a set of tasks (colours) to multiple processors.
- Three different types of parallelization: *multithreading* (OpenMP), *message-passing* (MPI) and *hybrid parallelization* -- which combines message-passing and multi-threading.

Serial and Parallel Computing

Example of a parallel job

Here we see the same serial task as in the serial job example. But what is different about the main() function?

In this case, instead of a *for loop*, the tasks are being mapped to different threads (CPUs) using *multi-threading* parallelism (multiprocessing package in Python).

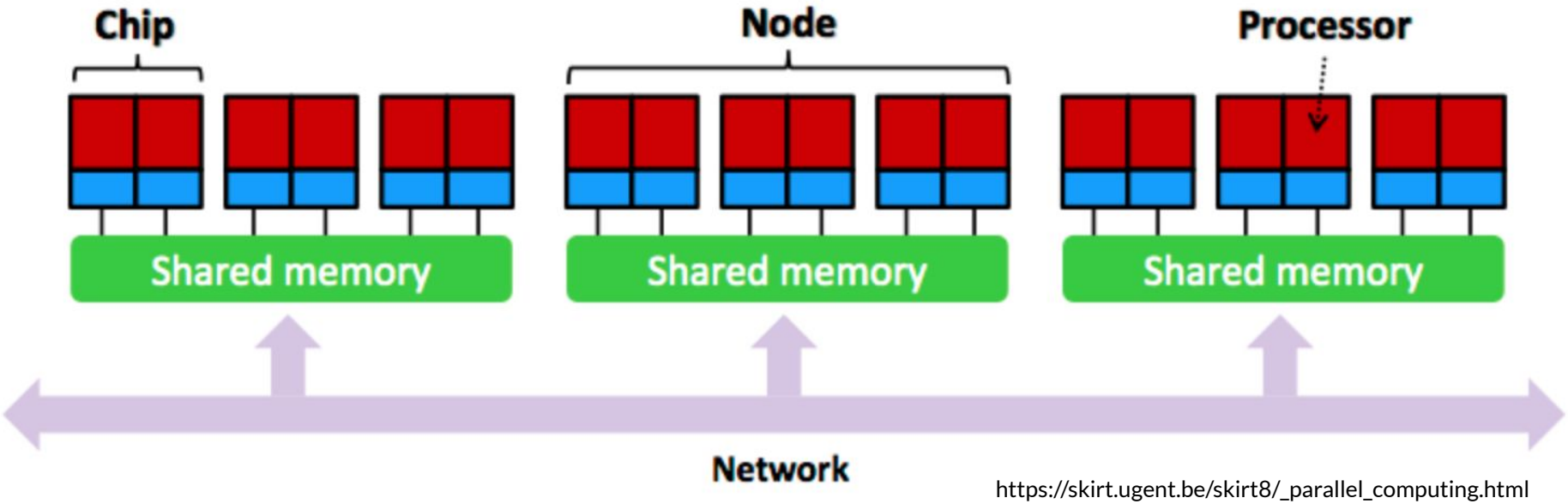
The job executes ~nthreads times faster than the serial job -- since each task is assigned to a separate CPU.

Once a CPU is done with a task, it accepts the arguments for the next unassigned task.

```
1  # Example Parellel OpenMP Job
2  import time
3  from multiprocessing import Pool
4
5  # Serial task
6  def serial_task(idx):
7      time.sleep(0.5)
8      filename = f'OpenMP.{idx}.txt'
9      with open(filename,'w') as f:
10         for i in range(idx+1):
11             f.write(f'{i**2}\n')
12     return
13
14 def main():
15     # Main code
16     start = time.time()
17     args = range(16)
18     nthreads = 8
19     with Pool(nthreads) as pool:
20         pool.map(serial_task,args)
21     runtime = time.time()-start
22     print(f'Time: {runtime}s')
23
24 if __name__=='__main__':
25     main()
```

Computing and the Queue System on IPMU's iDark Cluster

Cluster Computing



https://skirt.ugent.be/skirt8/_parallel_computing.html

A cluster is made up of a large number of CPUs organized onto ***nodes*** -- on which the CPUs share a block of memory. Each CPU on a node share the total memory on that node. Each node is also connected via a ***network*** -- which enables rapid distribution of tasks, passing of information, and even memory sharing across multiple nodes.

iDark Cluster at IPMU

When you login to iDark, you will be connected to a ***head node*** shared by all users. All heavy-lifting should be done via the ***compute nodes*** which are accessed via a queue system:

Node Name	CPU count	Memory
ansys[01-40]	52	376.4 GB
ansys[1-2]	56	1.510 TB

Queue System: PBS Professional (next slides)

User Manual:
https://github.com/cbottrell/HPC_IPMU/blob/main/Docs/idark_users_manual_en.pdf

Check node status with:

```
pbsnodes -a
```

Queue system on idark

Many large computing clusters are shared between many users. This means competition for resources. Some clusters use a ***priority-based queue*** system where a ***fair share*** is enforced by reducing priority when resources are being overused. This ***forces*** users to take care about the jobs they submit.

IPMU clusters use an ***honour-based queue*** system -- which is to say that there is no regulation of resource usage. Therefore, in consideration of other users, it is important to consider the efficiency of your jobs (e.g. CPUs/memory used vs requested), and the duration of jobs.

At the end of this tutorial, we will explain how to monitor/check the efficiencies of current jobs.

Queue system on idark

queue	Maximum number of jobs to be executed /user	Maximum number of cores in use /job	Maximum number of nodes in use /job
tiny	256	1	1
mini	6	52	1
small	3	208	4
large	1	1040	20
mini2	1	56	1

There are currently 5 queues on idark -- each with different restrictions on (1) number of active jobs (2) maximum number of CPUs/job and (3) maximum number of active nodes/job.

One thing to consider, however, is the role of memory. Each CPU on a node uses shared memory. If I request a single CPU in the tiny queue with 376 GB of memory, I am effectively blocking off the whole node for myself and other users. So be careful, and considerate, about memory requests.

Running jobs

Running an *interactive* job

An ***interactive job*** is a resource allocation which will connect the user to (a) compute node(s) via ssh. Interactive jobs are useful for:

1) Testing and debugging programs on the compute nodes before deployment in the main queue.
2) Regular work to relieve congestion on the login node (e.g. jupyter lab sessions).

```
qsub -l select=1:ncpus=1:mem=4gb -l walltime=3:0:0 -q tiny -I
```

This interactive job request asks for 1 chunk of resources (discussed later) with 1 CPU/chunk and 4GB of memory per chunk for 3 hours 0 minutes 0 seconds on the `tiny` queue.

Running an *interactive* job

Let's run the serial job script in interactive mode using the resources we have requested. First, ssh onto iDark and request the interactive job:

```
qsub -l select=1:ncpus=1:mem=4gb -l walltime=3:0:0 -q tiny -I
```

Then, from the compute node, clone the tutorial repository to a directory of your choice (current directory) with git:

```
git clone https://github.com/cbottrell/HPC_IPMU.git
cd HPC_IPMU/Code/Serial
```

The scripts are written in Python 3, which needs to be activated from conda or by activating your own virtual Python 3 environment.

```
source /home/anaconda3/bin/activate
python Serial_Example.py
```

Running a job

Interactive jobs are for testing with limited resources (less than 3 hours with only the CPUs needed to test the code).

All long/resource-intensive jobs should be submitted to the queue via a job script:

```
#!/bin/bash
#PBS -N Serial_Example
#PBS -o /home/connor.bottrell/Scratch/pbs
#PBS -e /home/connor.bottrell/Scratch/pbs
#PBS -l select=1:ncpus=1:mem=4gb
#PBS -l walltime=00:30:00
#PBS -u bottrell
#PBS -M connor.bottrell@ipmu.jp
#PBS -m ae
#PBS -V
#PBS -q tiny

# activate Python 3
source /home/anaconda3/bin/activate
# you can set environment variables in the job script
export HPC_DIR=$HOME/Demos/HPC_IPMU
cd $HPC_DIR/Code/Serial
# run program
python Serial_Example.py
```

Options:

-N: Job name in queue
-o: Output file full path (non-dynamic)
-e: Error file full path (non-dynamic)
-l: Resources requested
(1 CPU with 4GB mem for 30 minutes)
-u: Username
-M: Mail address for job updates
-m: Options for when to receive mail
-V: Import environment
-q: The queue in which to run the job

Please personalize! I don't want your job mail.

Running a job and checking job status

```
cd HPC_IPMU/Code/PBS
# edit file for your mail address and path
qsub Serial_Example.pbs
```

qsub is the job submission command. We already used it for the interactive session.

The job should complete very quickly. Too fast to look at it in the queue. Check your output/error file path for the output/error files.

Active/queued jobs can be checked using **qstat** :

```
qstat -ntr -u user_name # running jobs/subjobs
qstat -ntr # running jobs/subjobs by all users
qstat -nt  # all queued/running jobs/subjobs
```

Other job/queue management tools

Suppose you've made an error in your job script or program. Jobs can be cancelled using **qdel** and the **Job ID** from **qstat**:

```
qstat -ntr -u bottrell
```

```
idark:
                                                                  Req'd  Req'd   Elap
Job ID          Username Queue    Jobname    SessID NDS TSK Memory Time  S Time
--------------- -------- -------- ---------- ------ --- --- ------ ----- - -----
8220.idark      sunao.su small    run_mn     216318   4 208   40gb   --  R 291:3
   ansys02/0*52+ansys25/0*52+ansys03/0*52+ansys28/0*52
8221.idark      sunao.su small    run_mn     248886   4 208   40gb   --  R 291:3
   ansys29/0*52+ansys30/0*52+ansys31/0*52+ansys32/0*52
9080.idark      toshiki. small    run_mn     268982   4 208   40gb   --  R 242:3
   ansys17/0*52+ansys21/0*52+ansys22/0*52+ansys34/0*52
13356[1492].ida bottrell mini     Run_SKIRT  177020   1  50  150gb 24:00 R 01:24
   ansys06/0*50
15231.idark     bottrell tiny     TNG_Downlo 179412   1   1  512mb 36:00 R 00:00
   ansys06/1
```

```
qdel 15231
```

Checking status of a job

You may ssh onto any node on which you are running jobs. Indeed, you can ssh onto any node on iDark, but it is bad practice as you may disrupt the workflow of other users.

If I have a job running on the ansys06 compute node, I can do:

```
ssh ansys06
```

on the login node and then check the efficiency of my job with the **top** command:

```
top -u user_name
```

```
top - 11:16:48 up 79 days, 22:59,  1 user,  load average: 49.09, 46.24, 46.33
Tasks: 736 total,   7 running, 729 sleeping,   0 stopped,   0 zombie
%Cpu(s): 96.2 us,  0.0 sy,  0.0 ni,  3.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
GiB Mem :    376.4 total,    340.3 free,     14.5 used,     21.6 buff/cache
GiB Swap:      8.0 total,      8.0 free,      0.0 used.    360.7 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2724 bottrell  20   0 1692736 785440   6000 R 999.0  0.2  32:22.67 skirt
 2725 bottrell  20   0 1697260 787996   6012 R 999.0  0.2  32:23.90 skirt
 2726 bottrell  20   0 1697260 789848   5992 R 998.7  0.2  32:23.15 skirt
 2727 bottrell  20   0 1697260 787952   5992 R 998.7  0.2  32:23.34 skirt
 2723 bottrell  20   0 1697268 790040   6020 R 998.3  0.2  32:22.98 skirt
 2905 bottrell  20   0  166924   3072   1692 R   0.3  0.0   0:00.79 top
```

Use **SHIFT+e** to toggle between KiB, MiB, and GiB and **q** to quit top.

Running multi-core and multi-node jobs

Running a parallelized job

A parallelized job script is submitted in the same way as a serial job script. The difference is the resources requested and the program (which is now parellelized).

Below, 1 chunk of 8 CPUs and 4GB total memory is requested.

```
#!/bin/bash
#PBS -N Serial_Example
#PBS -o /home/connor.bottrell/Scratch/pbs
#PBS -e /home/connor.bottrell/Scratch/pbs
#PBS -l select=1:ncpus=8:mem=4gb
#PBS -l walltime=00:30:00
#PBS -u bottrell
#PBS -M connor.bottrell@ipmu.jp
#PBS -m ae
#PBS -V
#PBS -q mini

# activate Python 3
source /home/anaconda3/bin/activate
# you can set environment variables in the job script
export HPC_DIR=$HOME/Demos/HPC_IPMU
cd $HPC_DIR/Code/OpenMP
# run program
python OpenMP_Example.py
```

You can be clever about how the numbers of CPUs/nodes are communicated to a program by setting these as environment variables in the job script that can then be grabbed by the program.

```
cd HPC_IPMU/Code/PBS
# edit file for your info
qsub OpenMP_Example.pbs
```

Running a parallelized job

Resource Specifications:

```
#PBS -l select=4:ncpus=6:mem=8gb
```

select: how many chunks of [ncpus, mem] do you want
ncpus: number of CPUs per chunk
mem: memory requested per chunk

The request above is for 4 x 6 = 24 CPUs and 4 x 8 = 32 GB of total memory. The **select** request ***does not restrict CPUs to the same node*** and ***allocates to empty space wherever it exists*** (starting with nodes that are already busy). ***This maximizes cluster usage efficiency -- leaving as many empty nodes as possible***. If you wish all 24 CPUs to be on the same node:

```
#PBS -l select=1:ncpus=24:mem=32gb
```

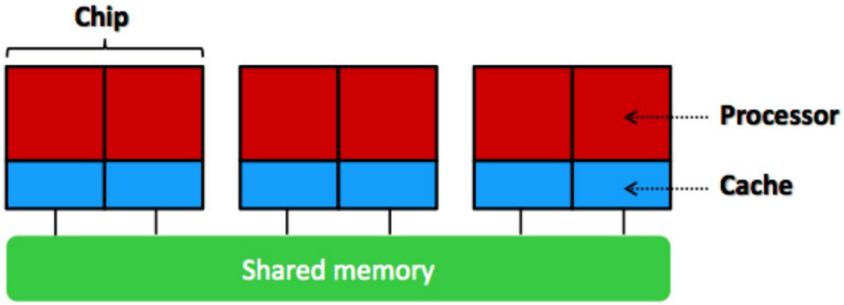
is the resource-equivalent request.

Types of parallelization: Which is right for you?

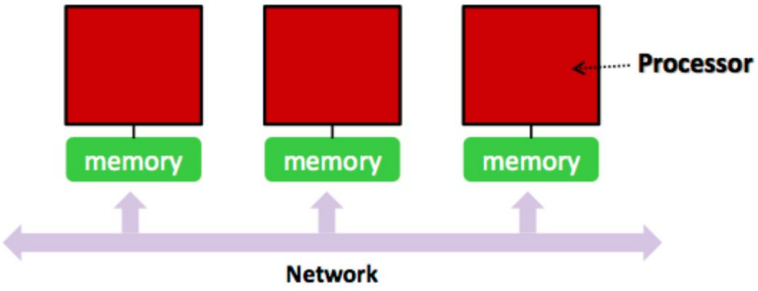
The multi-threading parallel job used in these examples is called an ***embarrassingly parallel job*** because no communication is required between CPUs. These are very useful.

When communication is required, the options are:

Multi-threading, Shared Memory



Message-passing, Distributed memory



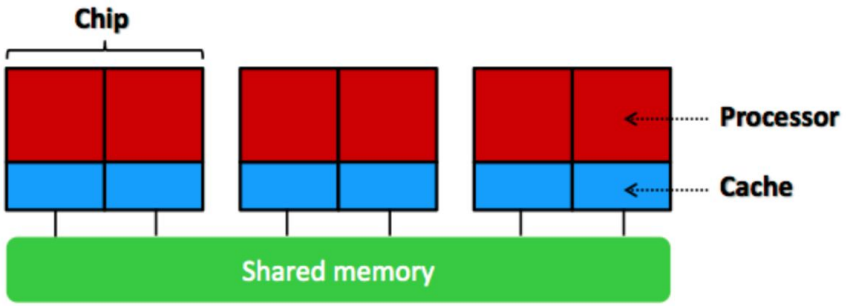
https://skirt.ugent.be/skirt8/_parallel_computing.html

And hybrid parellelization, which combines both by creating multiple instances of the data in memory which can be accessed by all threads connected to each instance.

Types of parallelization: Which is right for you?

The advantage of multi-threaded parallelism is that all CPUs can access and update the same information in memory, and these changes are immediately seen by all other CPUs.

The disadvantage is that it is not scalable to large numbers of threads because of the competition for access to memory between threads.



Multi-threading, Shared Memory

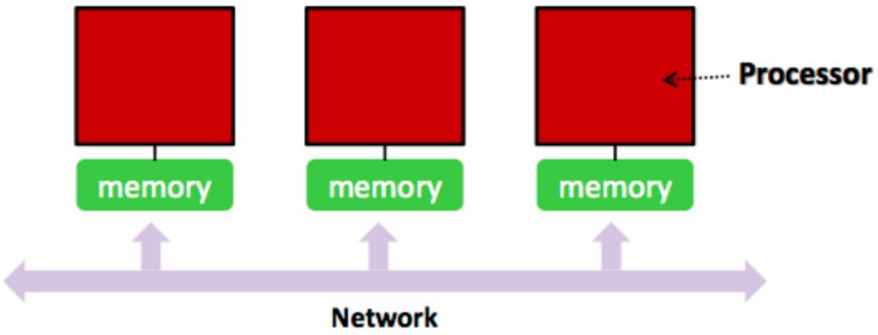
All processors share have access to the same memory chunk. Multi-threading is efficient on memory -- but can become CPU inefficient when too many CPUs are competing for read/write access to a large dataset stored in memory.

Types of parallelization: Which is right for you?

The advantage of distributed memory parallelization is that each chunk of CPUs has access to a unique instance of the data -- so little competition and high efficiency.

The disadvantage is that the memory requirements scale with the number of chunks and communication between chunks is not as fast as communication from processors to their own shared memory.

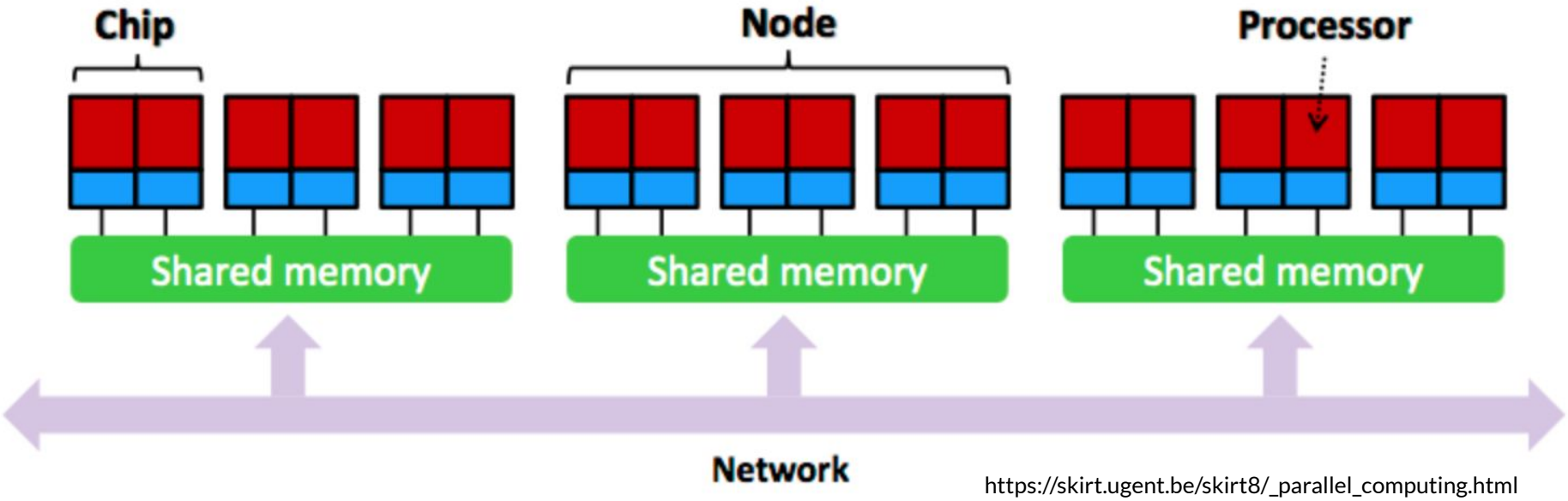
https://skirt.ugent.be/skirt8/_parallel_computing.html



Message Passing, Distributed memory

Each processor has access to its own copy of the data in a separate memory chunk. Changes to the data are communicated via the network.

Types of parallelization: Which is right for you?



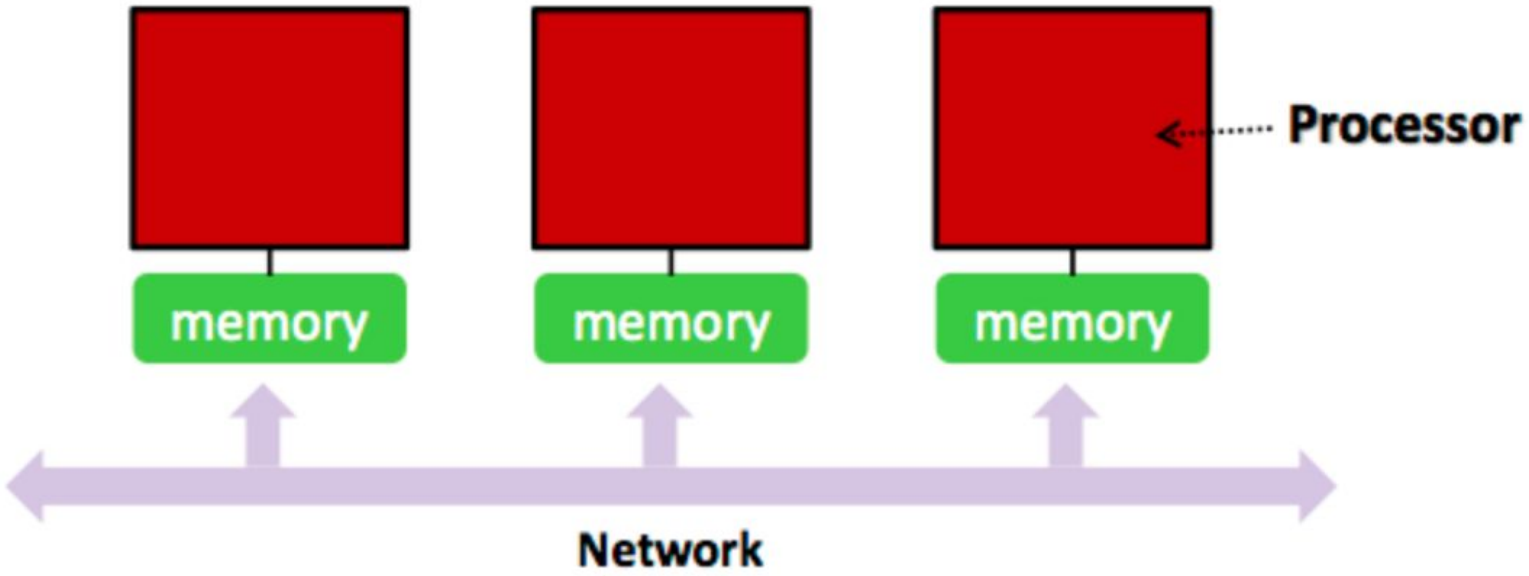
https://skirt.ugent.be/skirt8/_parallel_computing.html

In *hybrid* parallelization, shared memory parallelism (threading) is used together with distributed memory parallelism (multiple chunks/tasks).

E.g. NxN gravity solver with a huge data set. ***Each chunk*** (process/task) needs the whole data set -- but only updates the output (force table) for only Nparticles/Ntasks particles. In each task, the force calculations on each particle use thread-based parallelism.

Message Passing Interface (MPI)

Why use MPI?



https://skirt.ugent.be/skirt8/_parallel_computing.html

Simple answer: ***it's the only way parallelize across multiple nodes!***

Numerous computational tasks in astro can require multi-node resources; processing power, memory allocation, etc. N-body simulations are good examples of such tasks, as well as likelihood analyses using Markov Chain Monte Carlo techniques.

How to run an MPI code?

To run an MPI code, you need to check some compatibilities.

If you are using a compiled language, e.g. C/C++/Fortran, you should *compile* your code and afterwards *run* your code using the same MPI library.

If you are using Python, you may not notice the compilation step but it does happen during the installation of `mpi4py`. So you should also check that the MPI library used for `mpi4py` compilation is consistent with the MPI library used at runtime. We'll get deeper into this issue in just a bit.

Lastly, you should make sure that whatever MPI library you choose to use at compilation/runtime is compatible with the cluster, in particular the job scheduler (PBS Pro for idark).

The setup at idark

At idark, there is a system-installed default MPI library: Intel MPI. This installation has been tested for compatibility with the cluster setup, and is activated by default at login.

Using this default library is the easiest way to compile/run your MPI code. Unfortunately, **this is not always feasible.**

Let's get into some realistic examples.

```
SCRIPTDIR=/home/.common

#####################################
#
# for Compiler
#

COMPILER=INTEL19.0
#COMPILER=INTEL18.0
#COMPILER=INTEL17.0
#COMPILER=INTEL15.0
#COMPILER=PGI17
#COMPILER=PGI16
#COMPILER=PGI15

#####################################
#
# for MPI
#

MPI=IntelMPI
#MPI=OpenMPI
#MPI=MPICH
#MPI=MPICH2
```

Your default `.bashrc` file at idark would look something like this.

Running an MPI code on idark: C/C++

With C/C++, we can just use the default Intel MPI library.

```c
#include <mpi.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv) {

  MPI_Init(NULL, NULL);

  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int name_len;
  MPI_Get_processor_name(processor_name, &name_len);

  printf("Hello world from processor %s, rank %d out of %d processors\n",
     processor_name, world_rank, world_size);

  printf("Now I will take up memory and waste computing power for demonstration purposes\n");

  static double nums[500][500][500] ={{{0.0}}};
  nums[0][0][0] = 1.0;

  while(1){

    nums[0][0][0] *= 3.14;
    nums[0][0][0] /= 3.14;

  }

  MPI_Finalize();
}
```

```
EXECS=test
MPICC?=mpicc

all: ${EXECS}

test: test.c
    ${MPICC} -o test test.c

clean:
    rm -f ${EXECS}
```

You can find this code in the repo, `/Code/MPI/mpi-test-c/`

Running an MPI code on idark: C/C++

Let's try compiling and running.

Compiling is dead easy: `make clean; make`

To run, first enter into an interactive session (we force 2 nodes here):
`qsub -I -l nodes=2:ppn=3 -q small`

To run with MPI, the usual command is:
`mpirun -n num_processes program_name`

```
youngsoo@idark$ mpirun -n 6 ./test
Hello world from processor ansys12, rank 0 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
Hello world from processor ansys15, rank 3 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
Hello world from processor ansys12, rank 1 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
Hello world from processor ansys15, rank 4 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
Hello world from processor ansys12, rank 2 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
Hello world from processor ansys15, rank 5 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
```

Running an MPI code on idark: C/C++

Once the code starts running, you can see for yourself that the code is eating up resources as designed.

Checking with `ssh ansys12; top -u youngsoo`

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
21834	**youngsoo**	**20**	**0**	**1153316**	**26128**	**3712**	**R**	**100.3**	**0.0**	**7:05.27**	**test**
21835	**youngsoo**	**20**	**0**	**1154880**	**29728**	**3736**	**R**	**100.0**	**0.0**	**7:05.26**	**test**
21836	**youngsoo**	**20**	**0**	**1154880**	**29560**	**3728**	**R**	**100.0**	**0.0**	**7:05.26**	**test**
22249	**youngsoo**	**20**	**0**	**167024**	**3120**	**1688**	**R**	**0.7**	**0.0**	**0:00.04**	**top**
21507	youngsoo	20	0	121544	4112	1904	S	0.0	0.0	0:00.11	bash
21508	youngsoo	20	0	12912	436	328	S	0.0	0.0	0:00.00	pbs_demux
21822	youngsoo	20	0	113288	1468	1220	S	0.0	0.0	0:00.00	mpirun
21827	youngsoo	20	0	17856	1608	1292	S	0.0	0.0	0:00.00	mpiexec.hydra
21828	youngsoo	20	0	17280	1880	1188	S	0.0	0.0	0:00.00	pmi_proxy
21829	youngsoo	20	0	19180	984	800	S	0.0	0.0	0:00.00	rsh
21833	youngsoo	20	0	19180	184	0	S	0.0	0.0	0:00.00	rsh
21845	youngsoo	20	0	158892	2516	1176	S	0.0	0.0	0:00.04	sshd
21846	youngsoo	20	0	121412	3792	1788	S	0.0	0.0	0:00.10	bash

Another check with `ssh ansys15; top -u youngsoo`

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
82108	**youngsoo**	**20**	**0**	**1138980**	**28100**	**3672**	**R**	**100.3**	**0.0**	**6:47.39**	**test**
82109	**youngsoo**	**20**	**0**	**1140544**	**29664**	**3668**	**R**	**100.3**	**0.0**	**6:47.38**	**test**
82110	**youngsoo**	**20**	**0**	**1138980**	**28160**	**3716**	**R**	**100.3**	**0.0**	**6:47.39**	**test**
83042	**youngsoo**	**20**	**0**	**167184**	**3240**	**1688**	**R**	**0.3**	**0.0**	**0:00.06**	**top**
82052	youngsoo	20	0	19400	1984	1272	S	0.0	0.0	0:00.00	pmi_proxy
82928	youngsoo	20	0	158892	2516	1176	S	0.0	0.0	0:00.00	sshd
82929	youngsoo	20	0	121412	3796	1788	S	0.0	0.0	0:00.04	bash

Running an MPI code on idark: Python

With Python, things get a little bit more complicated. This is because conda tries to be self-contained, i.e. conda installs both executables as well as the necessary libraries instead of using system installations.

To use MPI with python, you need to install the `mpi4py` package. The base conda environment at idark doesn't have `mpi4py`, so you need to do it in your own environment:

```
conda create -n mpi-test
conda activate mpi-test
conda install mpi4py
```

The resulting environment can be checked with

```
conda list
```

And this is where it gets interesting...

Running an MPI code on idark: Python

Let's take a look at the output from `conda list`.

# Name	Version	Build	Channel
_libgcc_mutex	0.1	main	
_openmp_mutex	4.5	1_gnu	
ca-certificates	2021.5.25	h06a4308_1	
certifi	2021.5.30	py37h06a4308_0	
ld_impl_linux-64	2.35.1	h7274673_9	
libffi	3.3	he6710b0_2	
libgcc-ng	9.3.0	h5101ec6_17	
libgfortran-ng	7.5.0	ha8ba4b0_17	
libgfortran4	7.5.0	ha8ba4b0_17	
libgomp	9.3.0	h5101ec6_17	
libstdcxx-ng	9.3.0	hd4cf53a_17	
mpi	1.0	mpich	
mpi4py	3.0.3	py37hf046da1_1	
mpich	3.3.2	hc856adb_0	
ncurses	6.2	he6710b0_1	
openssl	1.1.1k	h27cfd23_0	
pip	21.1.2	py37h06a4308_0	
python	3.7.10	h12debd9_4	
readline	8.1	h27cfd23_0	
setuptools	52.0.0	py37h06a4308_0	
sqlite	3.35.4	hdfb4753_0	
tk	8.6.10	hbc83047_0	
wheel	0.36.2	pyhd3eb1b0_0	
xz	5.2.5	h7b6447c_0	
zlib	1.2.11	h7b6447c_3	

# Name	Version	Build	Channel
bzip2	1.0.8	hb9a14ef_6	intel
ca-certificates	2020.12.5	0	intel
certifi	2020.12.5	py37_0	intel
impi_rt	2021.2.0	intel_215	intel
intelpython	2021.1.1	1	intel
libffi	3.3	13	intel
libgcc-ng	9.3.0	hdf63c60_101	intel
libstdcxx-ng	9.3.0	hdf63c60_101	intel
mpi4py	3.0.3	py37hf484d3e_9	intel
openssl	1.1.1j	h14c3975_1	intel
pip	21.0.1	py37_0	intel
python	3.7.9	h86c8c07_7	intel
setuptools	51.1.2	py37_0	intel
sqlite	3.34.0	h2132df7_0	intel
tcl	8.6.10	0	intel
tk	8.6.10	h8e2d9d6_2	intel
wheel	0.36.2	py37_0	intel
xz	5.2.5	h85f8e93_2	intel
zlib	1.2.11.1	h047b5d8_3	intel

This is when you have Intel included in your conda channels list

Conda is now using its own MPI library, and we can only pray that it's compatible with the idark setup. Both of the above environments actually work; things will start breaking if conda gets `openmpi`.
It's also a good idea to now disable the default Intel MPI libraries.

Running an MPI code on idark: Python

But once the environment is properly set up, life becomes easy again.

Let's try running the following code.

```python
import sys
import numpy as np

from mpi4py import MPI

comm = MPI.COMM_WORLD
name = MPI.Get_processor_name()

print("Hello world fromp processor {}, rank {} out of {} processors"\
    .format(name, comm.rank, comm.size))

print("Now I will take up memory and waste computing power for demonstration purposes")

sys.stdout.flush()

nums = np.zeros((500,500,500))

nums[0,0,0] = 1.

while(1):

    nums[0,0,0] *= 3.14
    nums[0,0,0] /= 3.14
```

Running an MPI code on idark: Python

Going into an interactive session resets environment variables, so you should activate the conda environment one more time. Also, if you're really starting from scratch, you should get numpy:

```
conda activate mpi-test
conda install numpy
```

Then you can simply use the `mpirun` command again:

```
youngsoo@idark$ mpirun -n 6 python mpi.py
Hello world fromp processor ansys12, rank 2 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
Hello world fromp processor ansys12, rank 1 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
Hello world fromp processor ansys12, rank 0 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
Hello world fromp processor ansys15, rank 5 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
Hello world fromp processor ansys15, rank 4 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
Hello world fromp processor ansys15, rank 3 out of 6 processors
Now I will take up memory and waste computing power for demonstration purposes
```

Running an MPI code on idark: Batch jobs

Once you've checked interactively that your MPI code runs fine, use batch job submission to make serious calculations. Below are minimal examples for batch submission scripts; for best practices follow Connor's previous example.

```
#!/bin/bash
#PBS -q small
#PBS -l nodes=2:ppn=3
#PBS -l walltime=01:00:00
#PBS -N test
#PBS -o OUT
#PBS -e ERR
#PBS -k eo

source ~/.bashrc
conda activate mpi-test-intel

cd ${PBS_O_WORKDIR}

mpirun -n 6 python mpi.py
```

/Code/MPI/mpi-test-py/submit.sh

```
#!/bin/bash
#PBS -q small
#PBS -l nodes=2:ppn=3
#PBS -l walltime=01:00:00
#PBS -N test
#PBS -o OUT
#PBS -e ERR
#PBS -k eo

cd ${PBS_O_WORKDIR}

mpirun -n 6 ./test
```

/Code/MPI/mpi-test-c/submit.sh

Array Jobs:
Gold Standard of HPC Computing

The array job

Job Script

```
#!/bin/bash
#PBS -N Serial_Example
#PBS -o /home/connor.bottrell/Scratch/pbs
#PBS -e /home/connor.bottrell/Scratch/pbs
#PBS -l select=1:ncpus=1:mem=32mb
#PBS -l walltime=00:30:00
#PBS -J 0-256:1
#PBS -u bottrell
#PBS -M connor.bottrell@ipmu.jp
#PBS -m ae
#PBS -V
#PBS -q tiny

# activate Python 3
source /home/anaconda3/bin/activate
# you can set environment variables in the job script
export HPC_DIR=$HOME/Demos/HPC_IPMU
cd $HPC_DIR/Code/Array
# run program
python Array_Example.py $PBS_ARRAY_INDEX
```

Use job arrays whenever you have several independent tasks.

Program

```
# Example Array Job Program
import time
# To access $PBS_ARRAY_INDEX
import sys

# subjob task
def serial_task(idx):
    time.sleep(0.5)
    filename = f'Array.{idx}.txt'
    with open(filename,'w') as f:
        for i in range(idx+1):
            f.write(f'{i**2}\n')
    return

def main():
    # Get subjob idx from python arg vars
    program_name, pbs_array_idx = sys.argv
    pbs_array_idx = int(pbs_array_idx)
    # Code performed for each subjob
    start = time.time()
    serial_task(pbs_array_idx)
    runtime = time.time()-start
    print(f'Time: {runtime}s')

if __name__=='__main__':
    main()
```

Rationale for the array job

Array jobs generate an array of indices which become unique environment variables ($PBS_ARRAY_INDEX) in each ***subjob*** of the array. Advantages:

- Array jobs can parallelize programs across the *entire cluster* wherever there are empty resources (within queue limits). ***Accelerates your work***!
- Array subjobs can be made short (~minutes) which ***creates availability for other users*** in the queue.
- Two scripts: job script and program. ***No need for generator scripts*** for submitting jobs to the queue.
- Each subjob gets unique output/error files. Good for debugging, ***other subjobs unaffected by errors***.

Thank you for attending the Kavli IPMU HPC Tutorial

We hope you use the information, tips, and tricks in this tutorial to accelerate your computing!

Connor & Youngsoo