High Performance Computing: Tools and Applications

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Lecture 7

Accounts on Intel Xeon Phi servers

`gotham.cc.gatech.edu` (mic0 to mic7)
`joker.cc.gatech.edu` (mic0 to mic4)

- Each server has separate accounts and home directories (please restrict yourself to 5 GB in your home directory)
- Each server has 8 Intel Xeon Phi coprocessors
- Each coprocessor has 60 cores
- Each core has 4 hardware threads

Change your password on the host using `passwd`

Logins onto the coprocessors use `ssh` (already set up for you)

Using the coprocessors

- Three ways to run on Xeon Phi
  - native mode (coprocessor runs Linux uOS)
  - offload: program executes on host and offloads work to coprocessors
  - symmetric: MPI is used between hosts and coprocessors
- We will primarily use native mode (copy your executable onto the coprocessor and run it on the coprocessor)
- When you log into these machines, treat the host as a head node, where you can cross-compile for the coprocessors

Coprocessors and memory

- 8 GB local memory on the coprocessor card
- Do not have any permanent user file systems
- All created file systems reside in RAM (except network file systems) and thus have limited space
- Coprocessors have separate home directories (assuming native mode)
- Do not store anything of value on the coprocessors (they will disappear if the coprocessors are rebooted)
- In fact, only treat this space as scratch space. Only copy over executables, and immediately copy back output files. (Treat as if you are using the coprocessors as batch processors.)

Intel C/C++ compilers

- `gotham` and `joker` have the latest Intel compilers and tools
- Intel compilers often get better performance than Gnu compilers on Intel processors
- We also need Intel compilers for compiling for Intel Xeon Phi
- We will primarily use Intel Xeon Phi and Intel compilers in the rest of the course

Setting up for Intel compilers

To set up environment variables for Intel compilers on host (for architecture `intel64`):

```
source /opt/intel/compilers_and_libraries/\
  linux/bin/compilervars.sh intel64
```

The C compiler is `icc` and the C++ compiler is `icpc`

```
gotham:~$ which icc
/opt/intel/compilers_and_libraries_2016.2.181/linux/bin/intel64/icc
```

Compiler options

Note some different compiler options

```
icc -O2 -qopenmp ...
```

More information:

https://software.intel.com/en-us/
  intel-cplusplus-compiler-16.0-user-and-reference-guide

Thread affinity

- By default, threads can be assigned to any core, and may hop between cores. This helps balance the load on the cores, but usually you do not want this in HPC.
- Bind threads to specific cores. Advantages:
  - thread local information in the core's cache can be reused
  - thread stays on a socket, and has better access to memory associated with that socket (NUMA issues)
  - some cores share cache with some other cores; place threads that share memory on *nearby* cores

Thread affinity (gnu compilers)

- Controlled by `GOMP_CPU_AFFINITY` environment variable
- Bind thread 0 to core 7, thread 1 to core 6. Any additional threads are assigned starting with the beginning of the list.
  `GOMP_CPU_AFFINITY=7,6`
- Bind all threads to core 0
  `GOMP_CPU_AFFINITY=0`
- Bind first 4 threads to cores 0-3, etc.
  `GOMP_CPU_AFFINITY=0-3`
- Bind first 4 threads to cores 0, 2, 4, 6
  `GOMP_CPU_AFFINITY=0-7:2`
- *core* may mean *logical core* if hyperthreading is used

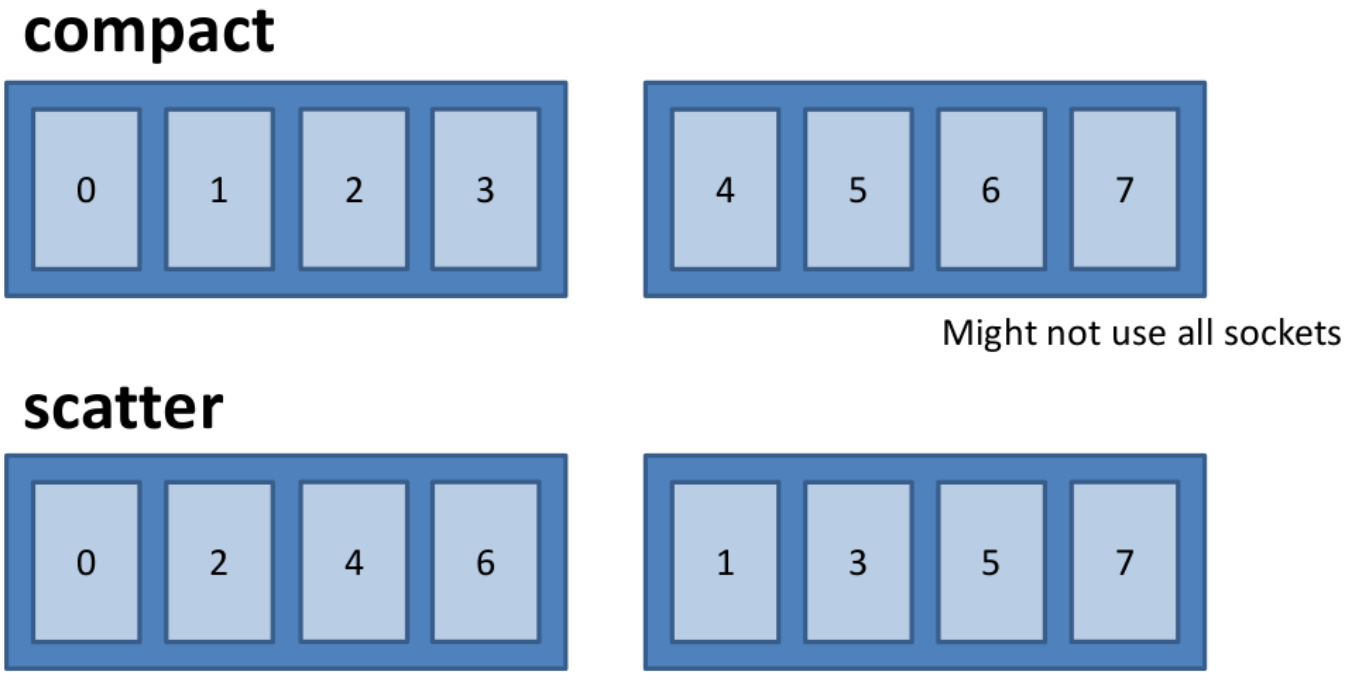
Thread affinity (Intel compilers)

- Controlled by `KMP_AFFINITY` environment variable, and is more flexible than the Gnu version
- Two main modes: `compact` and `scatter`

```
KMP_AFFINITY=compact
KMP_AFFINITY=scatter
```

`compact` and `scatter`

- 2 sockets, 4 cores/socket
- Number is thread id

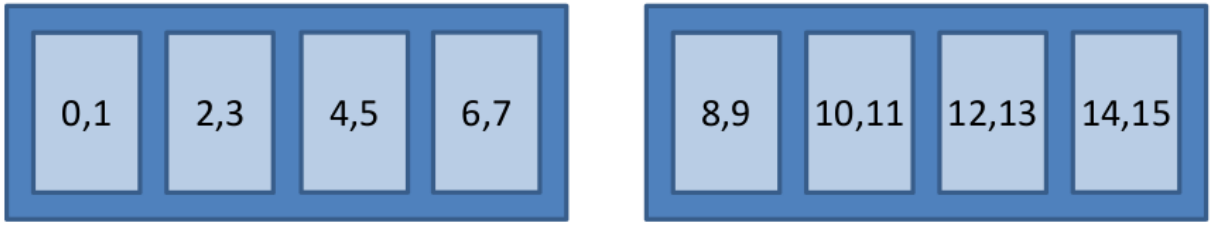


`compact` can be a bad option if you might be using fewer threads than cores on a node.

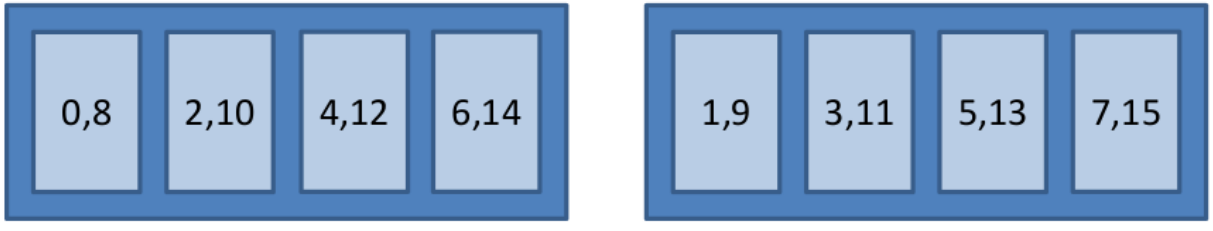
granularity=core

- The affinity mode can be *modified* by adding a `granularity` option which can be useful if hyperthreading is used
- default granularity is `core`
- 2 sockets, 4 cores/socket, 2-way hyperthreading

granularity=core,compact



granularity=core,scatter

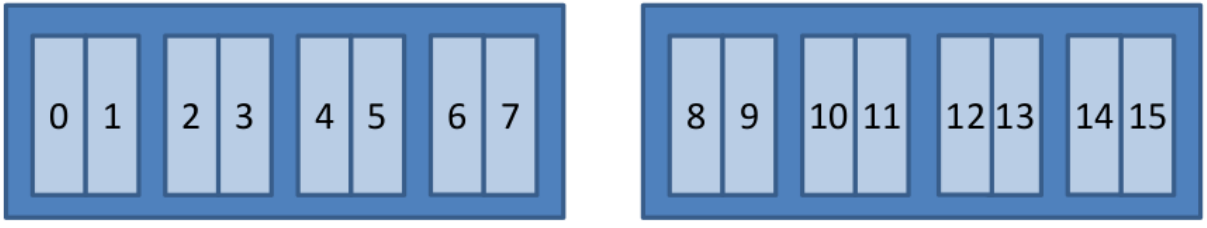


Thread can only float between hardware contexts on a core

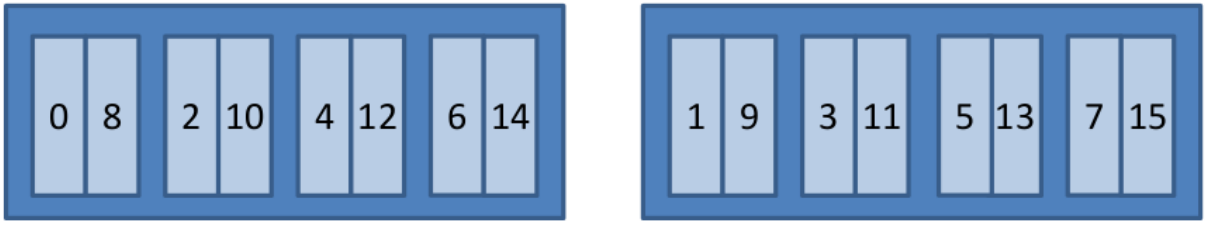
granularity=fine

- 2 sockets, 4 cores/socket, 2-way hyperthreading

granularity=fine,compact



granularity=fine,scatter

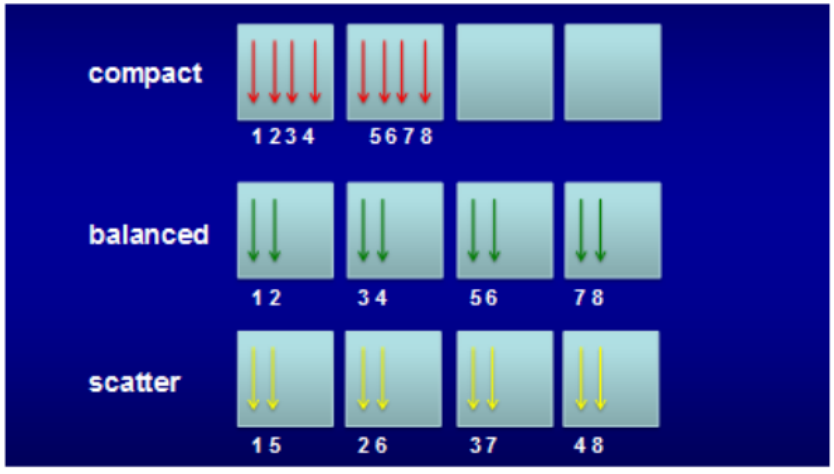


Usually no need to bind threads to hardware thread contexts

`balanced` mode (Intel Xeon Phi only)

On Intel Xeon Phi coprocessors, `balanced` mode is available

Example for 8 threads



4 cores and 4 HW threads per core.
Note: “balanced” is only for Intel Xeon Phi.

- bind to any hardware thread on core (default)
  `granularity=core,balanced`
- bind to single hardware thread
  `granularity=fine,balanced`

Other `KMP_AFFINITY` options

- another mode: `none`
- another modifier: `verbose`

Print affinity information, set mode to `none`

```
KMP_AFFINITY="verbose,none"
```

How to choose different modes

- For bandwidth-bound codes, the optimum number of threads is often less than the number of cores. Distribute and choose the number of threads to reduce contention and fully utilize the memory controllers (use scatter).
- For compute-bound codes, use hyperthreading and all available hyperthreads. Threads operating on adjacent data should be placed on the same core (or nearby) in order to share cache (use compact).
- On Intel Xeon Phi, `balanced` often works best. (Should we have this option on CPUs?)

Mini-Project 1

- Write a shared memory parallel code for Brownian dynamics simulation with steric interactions
- Main effort will be to parallelize the `interactions` function and to compute the repulsive forces in parallel
- Run on single Intel Xeon Phi coprocessor
- Speed will be a major factor of your evaluation; you can use any Intel/gcc compiler feature to make your code faster and you can even write assembly if you want!
- Due in `proj1` directory at 10 pm, Sat., Sept. 24

Mini-Project 1: Grading

- 0-6 points for parallel implementation producing correct results
  - correctness will be checked by computing diffusion coefficient of a simulation
  - points will be deducted for bad coding style, e.g., spaghetti code, insufficient comments, irregular indentation
- 0-6 points for speed
  - use the test harness code (checked into `proj1`) and provide a script (called from the makefile) for us to run (including copying files to the coprocessor, setting any environment variables for affinity, number of threads, etc.)
  - 6 points for fastest submission and 0 points for slowest submission
  - 0 points for incorrect code
- 0-3 points for report (`proj1.pdf`)
  - graph of time per timestep for 10000 particles at 20 percent volume fraction (use logarithmic y-axis) vs. number of threads
  - describe your final parallelization and why you consider your parallelization to be faster than other choices

Test harness in `proj1` directory

- `harness.c` is the main program (you can make minimal changes if necessary)
- most of your changes should go in `bd.c` or other files
- Update the `makefile` as needed
- Note: the test harness outputs a *trajectory* in xyz file format. This trajectory can be used for data analysis and making movies.

xyz format

- This is a single plain text file, with each *frame* concatenated one after the other.
- Each frame has the following format:
  - First line is the number of particles
  - Second line is a comment. But we use it to specify the simulated time since the start of the simulation, and the box width L
  - Each subsequent line gives the particle type and the particle position (you can ignore the particle type for now)
- Example:

```
344
227.000000 20.180992
0 24.685087 -12.790052 3.772569
0 -26.050274 10.934415 -32.012157
0 -5.682880 -19.800725 9.765145
...
```

Making a movie of your simulation (useful for debugging)

- Download and install VMD (visual molecular dynamics) `http://www.ks.uiuc.edu/Research/vmd/` on your local computer (Windows, OSX, Linux)
- run using `vmd output.xyz`
- at the VMD prompt, use this command to set the visualization style:

```
mol modstyle 0 0 {vdw 1 20}
```

- to see a simulation using periodic boundary conditions:

```
pbc set {20.181 20.181 20.181} -all
pbc wrap -all
pbc box -center unitcell -color tan
```

  where `20.181` is the box width

- use the *play* button to start the movie

Measuring MSD

Use the matlab script `a_msd3.m`. At the matlab prompt:

```
out = a_msd3('output.xyz');
```