Some basic HPC considerations

Paul Whitford

CTBP Computing Work Group

5/15/21

Objective

- Provide an overview of HPC strategies and considerations
- Not a programming guide
- Guide on how to be an excellent user (i.e. one that uses everything and wastes nothing)

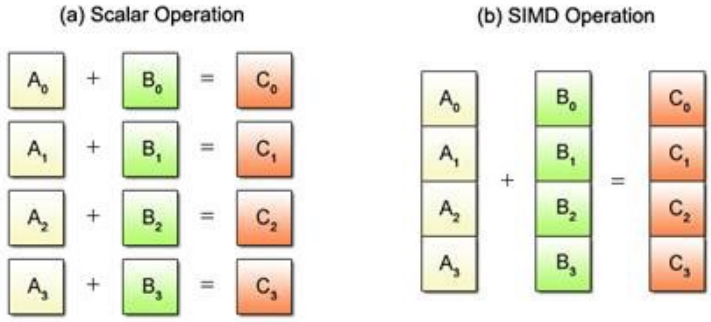
Topics

- Types of parallelization
  - SIMD (vector-based acceleration)
  - CPU-based openMP/threads
  - CPU-based MPI
  - GPU-based “massively” parallel calculations
- Discuss some pros/cons/limits of each level of parallelization
- Performance benchmarking

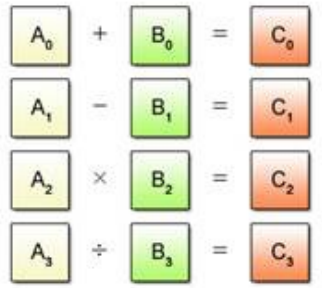
SIMD – single instruction, multiple data

- Vector-based calculations
- Idea:
  - The CPU register may be able to hold more data than a single variable
  - Load multiple pieces of data into a single processible unit
  - CPU will process identical instruction for all pieces at once
- Sometimes (auto)enabled at the compiler level
  - e.g. SSE, or AVX acceleration
- Coding can be very challenging
  - e.g. non-bonded (vdW) routines in GROMACS
- Before each operation, data must be organized for SIMD processing
- Can improve performance on modern cores
- More doesn't always mean faster
  - E.g. AVX256 can be faster than AVX512, even though 512 processes twice the number of vector elements per cycle
  - Slow down can be due to different clock speed with larger registers.

Possible



Not possible



Thread-based parallelization (e.g. openMP)

- All threads must use a single node
- Each thread controls one compute core
- All threads share/access the same memory
- Usually good for parallelizing simple portions of code (e.g. “for” loops, initialization, I/O)
- Often only efficient for a low thread count (~10)
  - Threads can compete for the same data, or wait for one another to finish.
- Simple to implement (available by default with most/all modern compilers)
- Idea: all threads execute the same instructions on different elements of memory
  - e.g. in a for loop over “i”, core 1 will runs i=0-100, cores 2 will handle i=101-200, etc
- Introduction at https://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf

Example: finding the max and min values of a quantity in a trajectory

```
#pragma omp parallel for private(i,ii,j,Rtmp) reduction(min:R_min) reduction(max:R_max)
                for(int ii=0;ii<frames;ii++){
/* Here are the possible combination rules*/
#ifdef CONTACTS
                        #ifdef C_TANH
                        Rtmp = CONTACTS_R_TANH(ii,ncoords,R,W,W1,inputData);
                        #else
                        Rtmp = CONTACTS_R(ii,ncoords,R,W,inputData);
                        #endif
#endif  /* end CONTACTS*/

#ifdef LINEAR
                        Rtmp=LINEAR_R(ii,ncoords,R,W);
#endif /* end LINEAR*/

#ifdef POWPROD
                        Rtmp=POWPROD_R(ii,ncoords,endpointA,endpointB,coordD,R,W);
#endif /* end POWPROD*/
```

MPI – Message Passing Interface

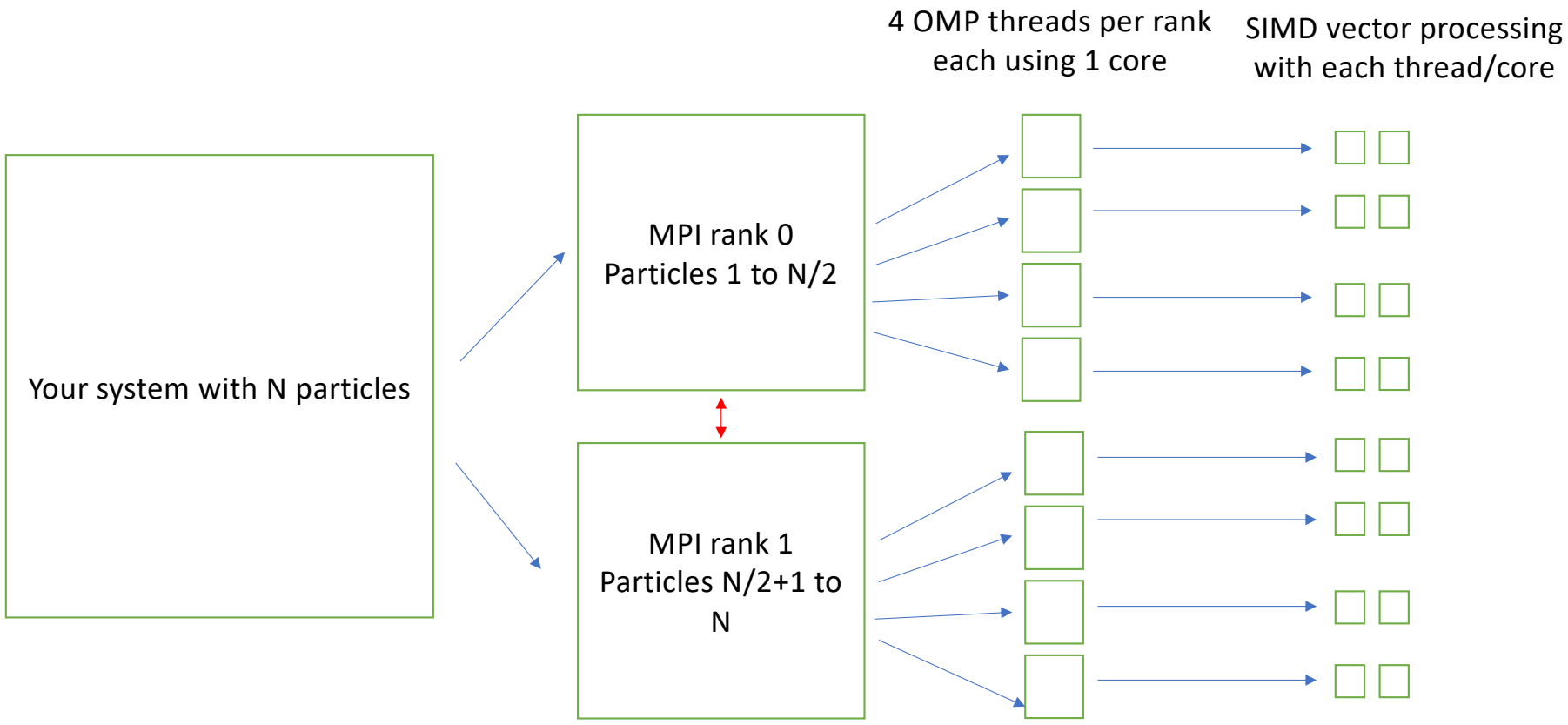
- Each MPI "rank" has separate memory
- Each rank controls its own instructions
- Ranks may be on the same, or different, nodes
- Instructions on each node may be independent of one another
  - e.g. rank 1 could work on 6-12 interactions, rank 2 works on PME, etc
- Usually good for parallelizing complex portions of code, or large-memory calculations (assuming memory doesn't need to be shared)
- Performance strongly depends on the connection between ranks/nodes
  - Latency (time waiting for communication to start) is often more important than bandwidth (speed of data transfers once started)
  - The faster the cores/nodes can communicate, the better calculations will scale
- *NOT* simple to implement
  - Lots of room for inefficient calculations
  - Possible that ranks will often be sitting idle

For a basic introduction, see http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml

Multi-level parallelization

- May integrate MPI and openMP/threads in the same calculation
- Idea:
  - Break a large calculation into chunks with MPI ranks (e.g. molecular system into sets of atoms)
  - Each chunk can have many cores working on it via threads
- Each rank has its own memory
- Within each rank, the threads share memory
- Can allows for very highly-scalable calculations
- Still typically limited to a low thread counts per rank (~10)
- May also integrate SIMD operations
- Nightmare to write the code…

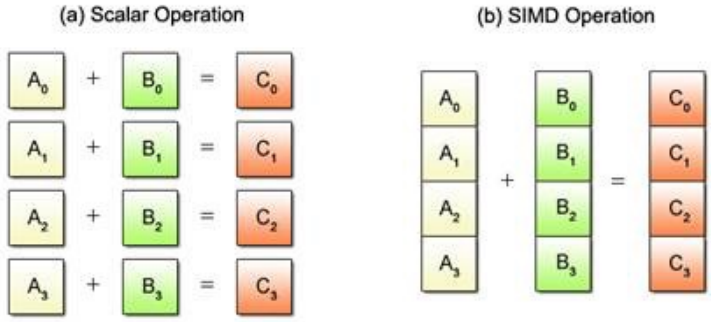
Example of multi-level parallelization (as implemented in Gromacs)



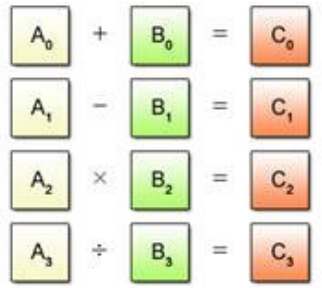
GPUs – “extreme SIMD”

- GPUs are massively parallel
  - 1000s of cores
- Data can only be processed in a SIMD-like form
- Challenge:
  - One needs to organize massive amounts of data for each cycle of the GPU to process
  - Organizing/streaming the data can rapidly become rate limiting
  - Since each cycle of the GPU can only process a single instruction, conditional statements become extremely slow (must execute all possible true and false variations of each calculation)
- If at all possible, leave GPU programming for the pros
- Can provide incredible speedup, or slowdown…

Possible



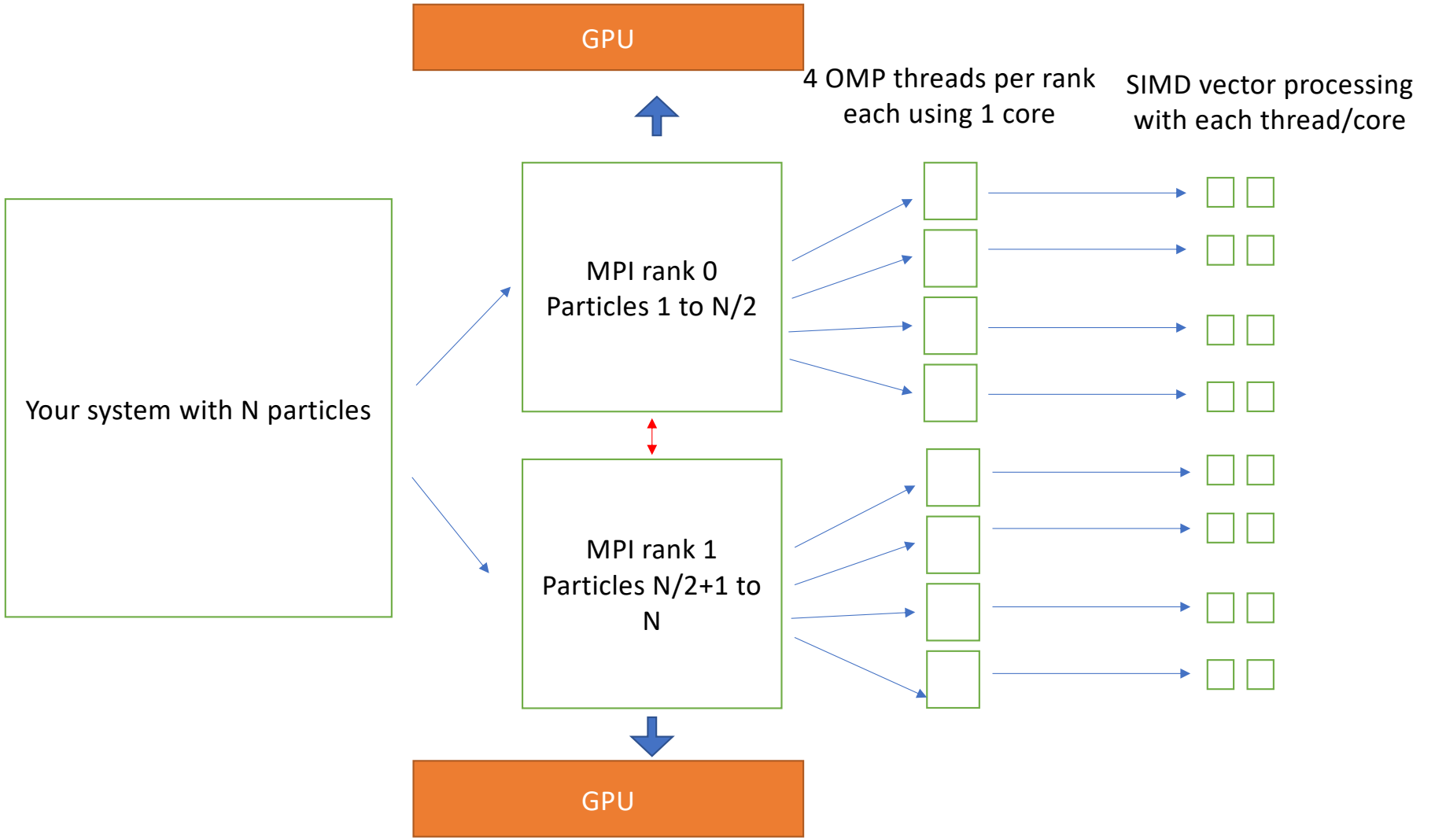
Not possible



GPU Challenges

- Data is not always easily streamlined for transfer to the GPU
- Communication between the GPU and CPU can be rate limiting
- Ideally one loads a small amount of data on the GPU and does a lot with it
  - Best for repetitive matrix operations that have no conditionals

Example of multi-level parallelization with GPUs (as implemented in Gromacs)



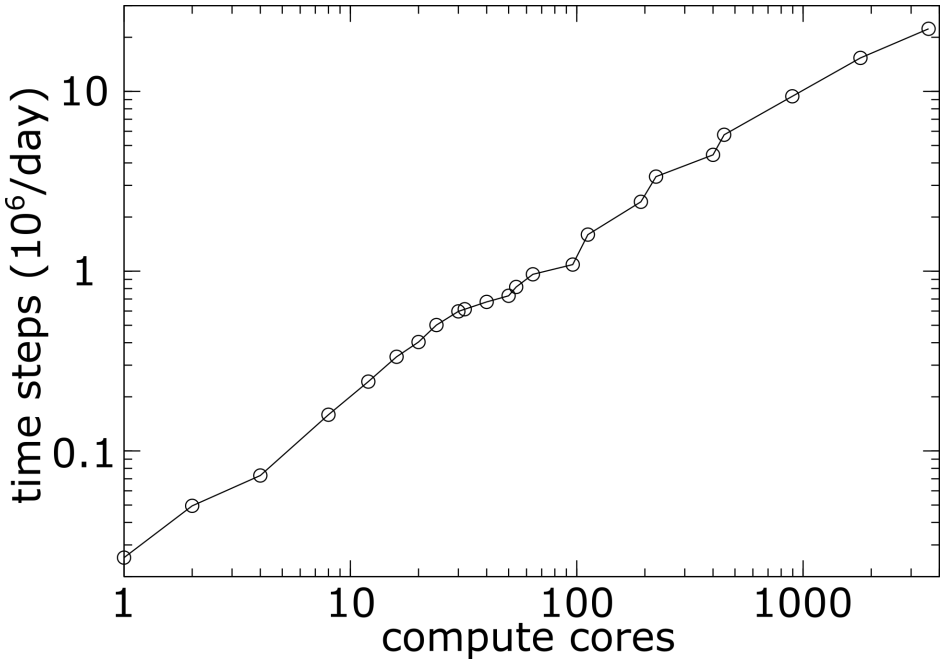
How to figure out what works?

- Test it…
- Test for strong scaling
  - Fixed-size calculation (e.g. N step simulation)
    - Perform on 1 core, 2 cores,…
  - “strong scaling” if time to solution scales with 1/# cores
- Test for weak scaling
  - Variable-size calculations
  - Perform:
    - Small calculation on 1 core, larger on 2, larger on 4, etc.
  - Weak scaling if time to solution remains constant.

Strong scaling strategy

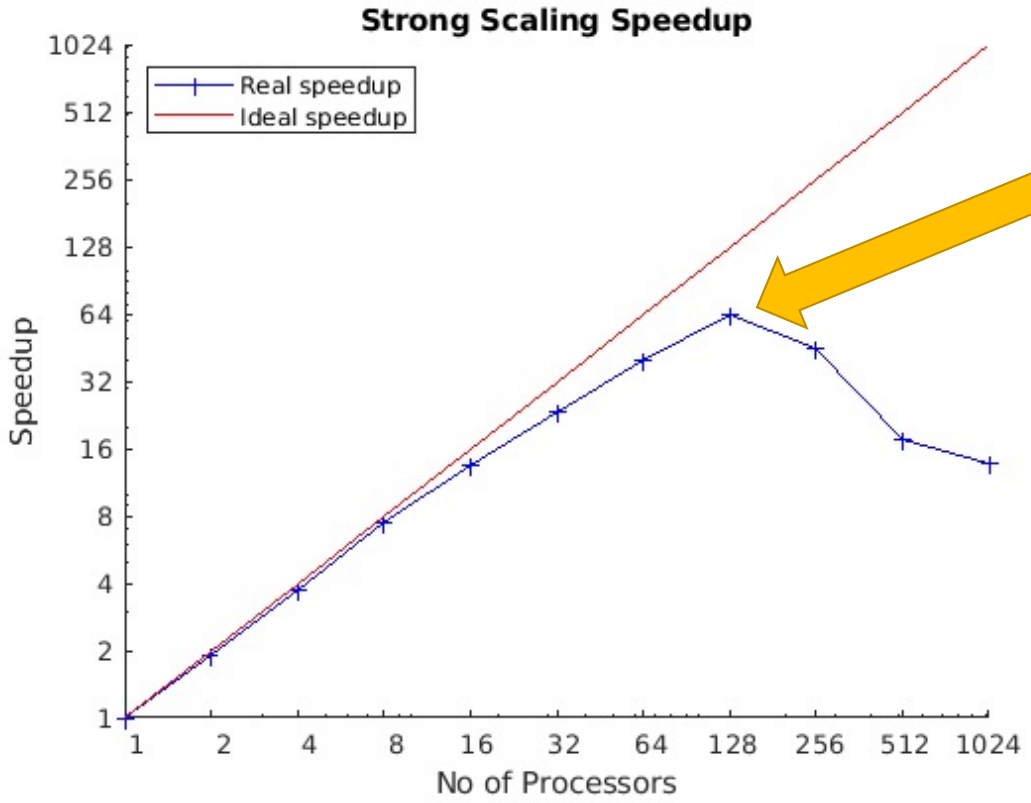
- Design a simulation that is representative of the production calculation
  - Typically the same system for fewer timesteps
- Simulate the system using 1, 2, 4…2^n cores
- Monitor the time to complete all non-initialization steps
  - e.g. any one-time calculations (e.g. loading the input deck) should not be included
- Plot 1/time versus number of cores
- Compare to linear extrapolation from 1 core
- Can quantify scalability as (time on 1 core)/(n*(time on n cores))
  - If perfectly scalable, then each core will have 1/n of the load and take 1/n of the time
  - If not perfectly scalable, each core will take longer than 1/n of the single-core time.
- Test all combinations of parallelization (MPI, openMP, SIMD)
  - The balance between ranks and threads can have tremendous impacts on performance
  - More cores doesn't always mean better performance

Example of strong scaling



Gromacs 5.1.4 with modified non-bonded kernels

Example of performance rollover



128 cores is ideal

Image from https://hpc-wiki.info/hpc/Scaling_tutorial

Performance testing GPUs

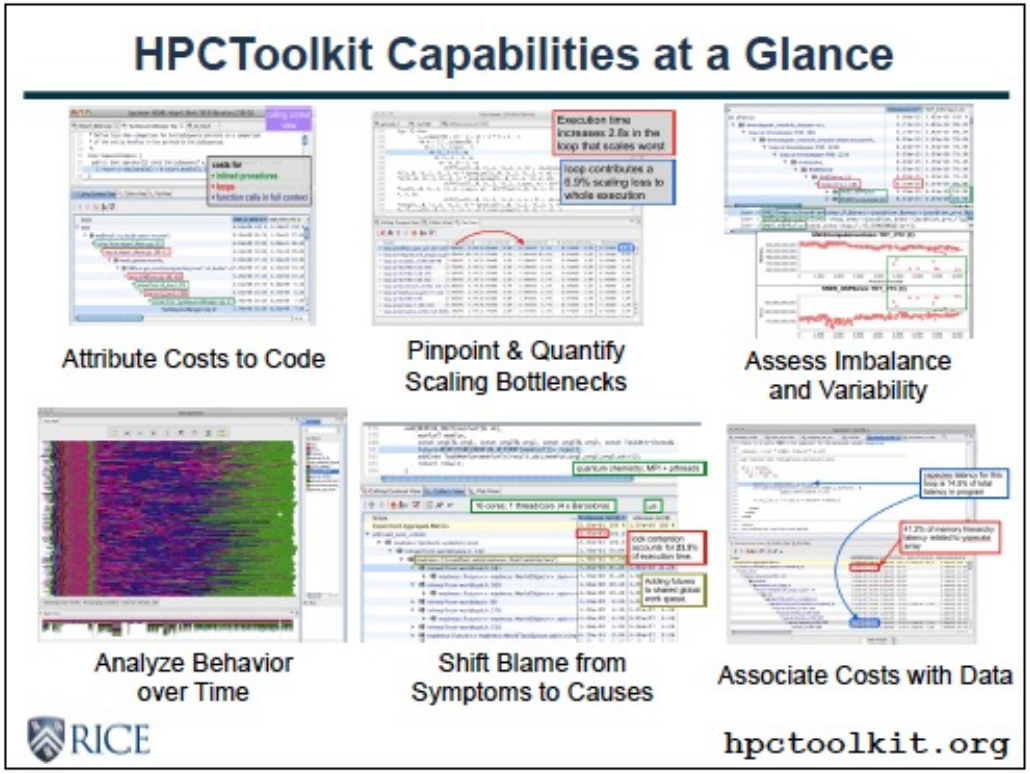
- Very important to compare single-core performance with single-core+GPU performance
  - If you don't see a major speedup (100x, or more), perhaps the code is not very GPU-friendly
- Can test for multiple-GPU performance
  - No guarantee performance will be better
  - Often performance degrades
- GPU performance can depend heavily on the type of calculation, even with the same software (e.g. if PME or 6-12 interactions are offloaded to GPU)
- Different GPUs require different languages (e.g. CUDA, OpenCL)
  - Same package may use completely different routines for each type of GPU
  - Differences in performance may not be due to hardware

For when the code needs improvement…
High-performance tuning with HPC toolkit

HPC Toolkit developed at Rice (Mellor-Crummey Group)

HPC Toolkit functionality

- Accurate performance measurement
- Effective performance analysis
  - Pinpointing scalability bottlenecks
  - Scalability bottlenecks on large-scale parallel systems
- Scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior



Experts are available to help you profile your code, even if you think it works well!

openMM specifics

- Limited MPI support
- Thread-based parallelization supported
- GPU support (obviously)
- Can run in GPU-only mode, GPU-CPU mode, or CPU mode

- Can work well for small systems
- Not known for being scalable, so very large systems may benefit from other software

For description of algorithms, see https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005659

AMD Resources

- GPU nodes:
  1 x AMD EPYC 7642 CPU (48 cores at 2.3GHz)
  8 x AMD Radeon Instinct MI50 32GB
  512 GB RAM
  - 80 in the POD cluster (Hosted by AMD)
  - 19 on Rice Campus
- CPU nodes:
  2 x AMD EPYC 7302 CPU (16 cores at 3.0GHz)
  4 TB RAM
  - 2 available at Rice

AMD POD Cluster Resources

- Hosted by AMD
- GPU nodes:
  1 x AMD EPYC 7642 CPU (48 cores at 2.3GHz)
  8 x AMD Radeon Instinct MI50 32GB
  512 GB RAM
  - 80 in the POD cluster
  - 19 on Rice Campus
- Containers for *Singularity* are already available for openMM, Gromacs 2020, NAMD, LAMMPS
- Obtaining access takes a few days