Thread-based parallelism and OpenMP

Paul Whitford

CTBP Tech Talk

11/8/2023

https://github.com/Whitford/ctbp-techtalks/

Our goals for today

- Introduce some basic aspects of OpenMP parallelism
- Compile and run some simple examples using OpenMP

For a far more complete introductions, see

https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf

https://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf

https://github.com/Whitford/ctbp-techtalks/

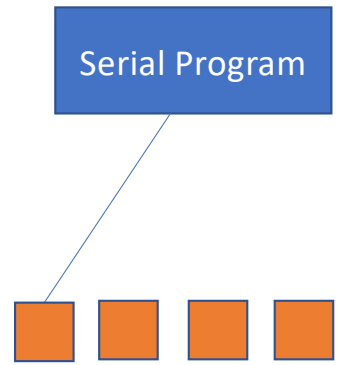
What is multi-threading?

- An approach that allows multiple program instructions to be executed concurrently.
- Each thread may
  - handle different iterations of a single loop
  - execute different intermediate calculations within a loop
  - execute completely independent instructions

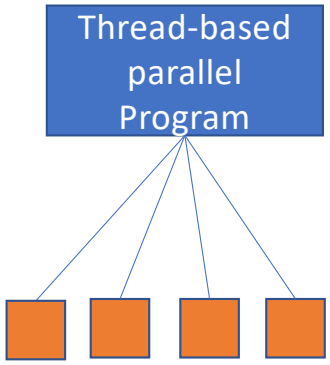
https://github.com/Whitford/ctbp-techtalks/

Why use multiple threads?

- If your code is “serial”, there is a sequence of statements that are executed, in order
  - This limits the calculation to the use of a single compute core
- If you have multiple threads, each thread *may* be handled by a separate core
- Also possible for multiple threads to be handled by the same core
  - This can utilize hyperthreading, which is enabled by the OS – essentially, exploit downtime in the CPU by trying to balance two sets of instructions



Cores on a single node



Cores on a single node

https://github.com/Whitford/ctbp-techtalks/

Threads are everywhere

- Almost all languages have some capacity for thread-based parallelism
  - Python
  - Fortran
  - C++
- Threads come in different flavors
  - POSIX threads (pthreads)
  - OpenMP

- We will use C++ with the OpenMP thread-based strategy

https://github.com/Whitford/ctbp-techtalks/

Thread-based parallelization (e.g. OpenMP)

- All threads must use a single node
- Often, each thread runs on a single compute core
- All threads share/access the same memory
- In practice, good for parallelizing simple portions of code (e.g. “for” loops, initialization, I/O)
- Can be used to parallelize complex portions of code
- Often efficient for lower 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

https://github.com/Whitford/ctbp-techtalks/

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*/
```

https://github.com/Whitford/ctbp-techtalks/

```
#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*/
```

- “#pragma omp parallel for…” indicates that this loop should be parallelized, if OpenMP is turned on at compile time
- “private” – make copies of the variables on each thread. Since OpenMP shares memory and addresses, this ensures that threads don’t write over each other
- “reduction” – consolidate this variable from across the threads. In this case, each thread had a variable “R_min”, so collect them all and find the min of all

https://github.com/Whitford/ctbp-techtalks/

Let's try it

- You will need a C++ compiler that has OMP support
  - If you are on a cluster, then a gnu compiler module will probably do the trick
  - I've tested 9.2.0
  - If you are on Apple, then installing g++ can be a hassle. So, you can use a container built for this:
    - `docker run –it ––rm –v $(pwd):/workdir smogserver/techtalk:latest`
    - This will download an image with Ubuntu 22.04, some text editors and g++
  - If you need to build your own container, then see the Docker.build.bash file that is in this session's git repo. It will build and launch the container
- If you want to visualize things, then you need VMD

https://github.com/Whitford/ctbp-techtalks/

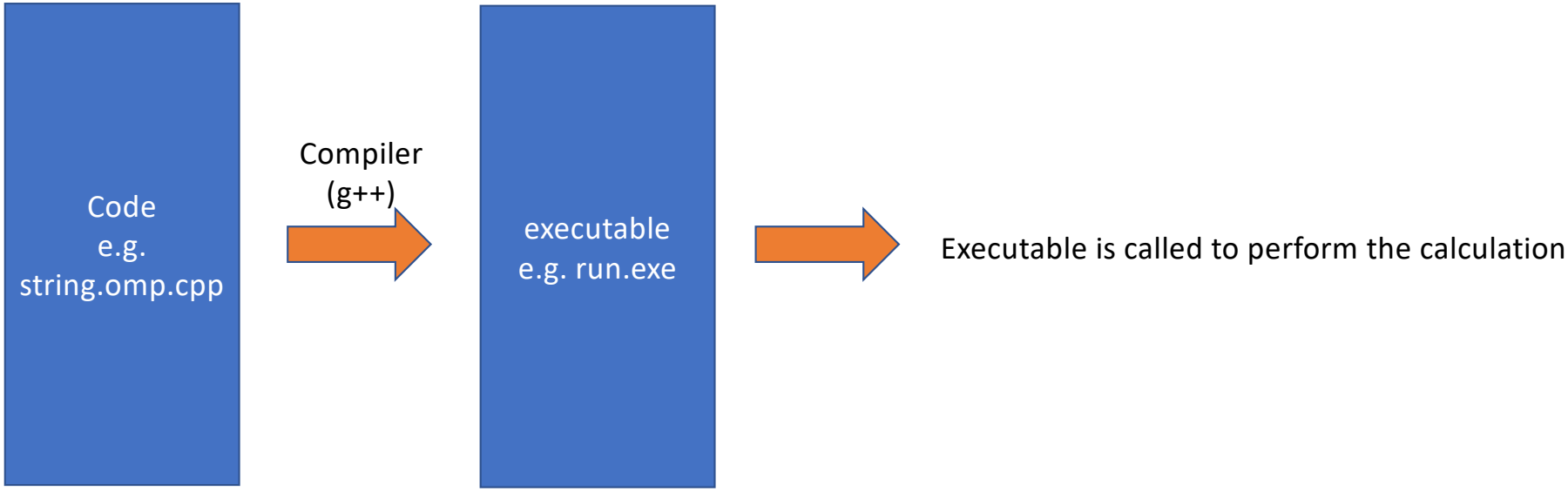
A simple physical system

- N particles connected by harmonic springs with fixed boundaries
- Initialize the system with some initial y displacement for all particles
- Release the system and watch the dynamics using VMD (optional).

- We are going to use C++ and OpenMP parallelization

https://github.com/Whitford/ctbp-techtalks/

C++ is a compiled language



https://github.com/Whitford/ctbp-techtalks/

Simulating our string system: string.omp.cpp

- The code is set to run 100 particles for 10M time steps
- Task: compile and run the program
- Visualize the trajectory in VMD
  - This will require you to download the trajectory

https://github.com/Whitford/ctbp-techtalks/

Open string.omp.cpp with a text editor

https://github.com/Whitford/ctbp-techtalks/

Running and compiling, today

- Set up your machine for use of g++
  - Installed locally
  - Through an environment
  - Through a module
  - Use the container

- Call the compiler with:
  - g++ <filename> -o run.exe
  - run.exe is the executable
- Run the program with:
  - ./run.exe

- Every time we change the cpp file, you MUST recompile the code

https://github.com/Whitford/ctbp-techtalks/

Simulating our string system: string.omp.cpp

- Task: turn on compiler optimization with the –O3 flag during the compile step
- Task: Re-run our code for a small system 100 particles – 10M steps
  - Was it faster?
- What is going on?
  - Compiler tries to analyze the code and figure out how to streamline the calculation
  - E.g. ”unrolls” loops
- Even if it is faster, the code is still *serial*

https://github.com/Whitford/ctbp-techtalks/

If your code was too fast without optimization

- If the last call completed in less than 10 seconds, then increase nstep by a suitable factor, so that the calculation should take 10-30 seconds
- We need to do this to see whether things are faster, or not

- We will not use optimization – this is just to see the effects of parallelism. Normally, you want to optimize and run in parallel.

https://github.com/Whitford/ctbp-techtalks/

Simulating our string system: string.omp.cpp

- Task: Run the same system with 10k particles for 10^5 steps
  - Compile and run without optimization
  - Set savefreq to a very large number (to avoid I/O-limited results)

- This is basically no time at all for a system of this size. If your computer can handle it, visualize the trajectory in VMD
- We clearly want more steps, so let's make it parallel

https://github.com/Whitford/ctbp-techtalks/

Add OpenMP parallelization

- Before any loop that you want to parallelize, add:
  - #pragma omp parallel for
- Add parallelization, compile and re-run your system
- Do you see a performance increase?
- Change the number of cores by issuing the following command on the command line:
  - export OMP_NUM_THREADS=4
  - You can set to 4, or any other number
  - Variable is read at runtime, so you don't need to recompile
- Re-check your performance

https://github.com/Whitford/ctbp-techtalks/

- You should have seen no change in performance. This is because we also need to tell the compiler to enable OMP
- Change our compile command to:
  - g++ -fopenmp string.omp.cpp –o run.exe
- Now, re-run the code
- Any difference in performance?
- How long does the simulation take using 1, 2, 4, 8 or 16 threads?
- Which is fastest?

https://github.com/Whitford/ctbp-techtalks/