Principles of Optimization

Code Optimization Workshop | 27.6.2022 | Jonathan Coles

Goals of this workshop

- **Understanding...**
  - Modern CPU architecture
  - Data layout and structures
  - Parallelization options
  - What and how to optimize
- **Measuring...**
  - Code behavior under varying assumptions
- **Implementing...**
  - A simple N-body code

Decide on goals for optimization

- **Reduced time to solution**
- **Reduced memory requirements**
- **Other resources**
  - Compute hardware → More cores / socket?
  - Energy → Different CPU or frequency?

Types of optimization

- **Compiler optimization**
- **Algorithmic optimization**
- **Implementation optimization**
- **Data movement**
- **Parallelization**

Some of the many challenges

- **Algorithmic optimization**
  - Theoretical improvements often require fundamental changes to existing software
  - New / different data structures

- **Implementation optimization**
  - There is a growing disconnect between the computer "model" behind programming languages and the underlying hardware.
  - Requires code to be written in a way that translates well on to the machine architecture.

- **Data movement**
  - Cost of data movement is usually not apparent at the language level.
  - Caching effects can be hard to predict.

- **Parallelization**
  - Multiple levels possible: instruction level, thread level, machine level
  - Coordinating multiple threads of execution is difficult.
  - Complex data dependencies and data structures can interfere with parallelization efforts.

Process of optimization

- **Measure**
  - Establish a method of measuring your performance criteria and correct/expected results.
  - Must be simple to run so that it can be used frequently.
  - Measure a base line to compare future improvements again.
  - Not always obvious where performance problems lie and can be machine or simulation dependent!
- **Record**
  - Save your measurements to a file or database.
  - Include as much information about the configuration used as is reasonable.
- **Plot**
  - Visualizing how performance changes with code changes is invaluable.
  - Trends become clearer and can suggest new ideas.
- **Modify**
  - Based on what was learned investigate new ideas and code changes.
- **Test**
  - Always test that the results haven't changed in a significant way.
  - It won't matter how fast the code is if it produces the wrong answer!

Things to keep in mind

- Simple is often better than complex
  - Despite the complexity of modern hardware, often the most straightforward thing is the fastest.
- Might need to "unlearn" some design patterns
- Optimization is a balance of resources
  - Sometimes being wasteful in one area can lead to gains in another

Things to keep in mind

- Two ways to make something run in less time:
  - Do the same amount of work faster (or in parallel).
  - Do less work ← often the easiest solution!
    - Don't spend a year rewriting a routine to run 10x faster on a GPU if you can simply use the routine 10x less frequently.
    - Often true regarding I/O.

Things to keep in mind

- Performance may be dependent on input
- Need to understand if the test case is representative of how the code will be used. Particularly true of scientific codes.
- Danger of optimizing the wrong thing
- Tools won't tell you this directly.

Tools to help

- **Source code control**
  - git, svn, etc.
  - Keep track of your changes and commit often.
  - Useful to go back easily to make comparisons.
- **Profiler**
  - Intel Advisor
  - perf
  - likwid
- **Debugger**
  - gdb, totalview
- **Compiler optimization reports**
  - Listen to the compiler.
  - Although cryptic, can often suggest why optimizations like vectorization aren't working.

Parallelization with OpenMP

- OpenMP (Open Multi-Processing) is standard that defines an API to support multi-processing within a programming language.
- Enables parallel execution of code with cores on a single machine. Does not deal with
- Does not directly change the language but defines annotations that a compiler can recognize to transform code into a parallel version.
- OpenMP is a vast topic, which is the subject of many other courses. Here you will only need to recognize ideas from the following example.

Parallelization with OpenMP

```
/* Open a parallel region. Threads are created that begin to execute the block simultaneously.
 * Variables declared before this region will be shared (no copies made) between all threads. */
#pragma omp parallel default(shared)
{
    printf("Hello!"); /* Each thread says hello */
    /* Partition the loop and assign each thread an (~)equally sized contiguous subset of the full range. */
    #pragma omp for schedule(static)
    for (int i=0; i < N; i++)
    {
        c[i] = a[i] + k*b[i];
    }
}
```

Parallelization with OpenMP

```
/* For self-contained parallel regions, keywords can be combined */
#pragma omp parallel for default(shared) schedule(static)
for (int i=0; i < N; i++)
{
    c[i] = a[i] + k*b[i];
}
```

Leibniz Supercomputing Centre

of the Bavarian Academy of Sciences and Humanities