Keep your multi-core CPU busy: Shared-memory parallelization with OpenMP

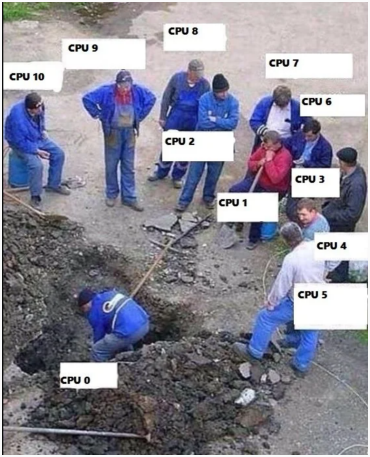
Tommy Chi Ho Lau

27/6/2023

Motivation

Modern CPUs have not advanced much in terms of clock rate but it's easier to have more cores.

But...



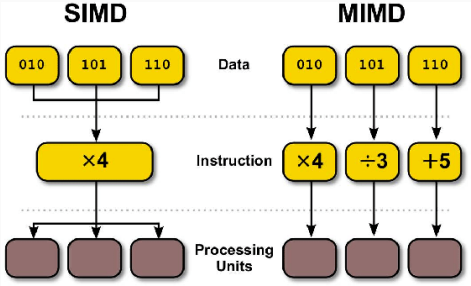
Motivation

- Some work can in principle be done in parallel
  - Can the task be divided into independent parts?
  - Does the sequence of the operations matter?
  - Is the problem large enough to justify the overhead penalty? (later in today's talk)
- e.g. N-body gravity
  - scales with N^2 and involves the notorious $1/\mathrm{sqrt}$

$$\mathbf{a}_i = -\sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{G m_j}{\mathbf{r}_{ij}^{\,3}} \mathbf{r}_{ij}$$

Parallel computing

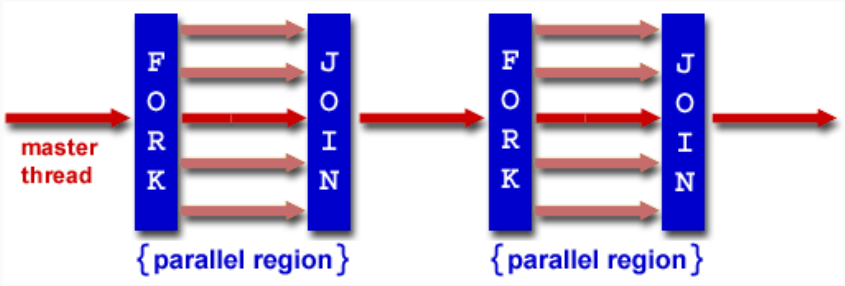
- Single Instruction Multiple Data (SIMD)
  - 'Vectorization'
  - Tiny to no performance penalty
  - Only suitable for specific tasks, e.g. array operations
- Multiple Instruction Multiple Data (MIMD)
  - Suitable for many tasks



OpenMP (Open Multi-Processing)

- A standard parallel programming API for shared memory environments
  - 'All CPU cores share the RAM' (same node/computer)
  - Alternative: MPI (Message Passing Interface) (low-level, distributed memory)
- A cross-platform, cross-compiler solution
  - Implemented by almost all modern compilers (GCC, Intel compilers)
  - Compiler flag, e.g. -fopenmp
- Supports C, C++ and Fortran
- Uses the Fork-Join Model

Fork-Join Model



- Significant overhead at each forking and joining!

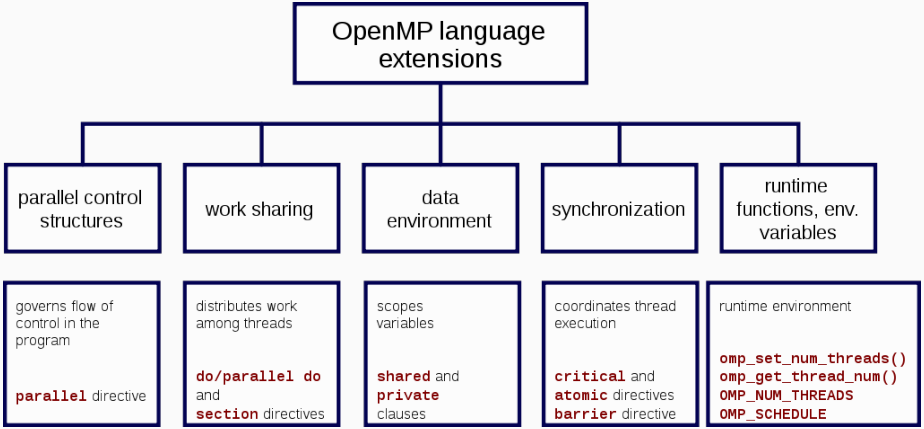
OpenMP Directives

Parallel Region: a block of code executed by all threads simultaneously

```
#pragma omp parallel
#pragma omp directive-name [clause]
{
...
}
```

```
!$OMP PARALLEL
!$OMP DIRECTIVE-NAME [CLAUSE]
...
!$OMP END PARALLEL
```

OpenMP Directives



Overheads

- Minimize number of joining and forking
- Not all parallelizable loops worth parallelizing
- Usually scales with number of threads (depends on environments)

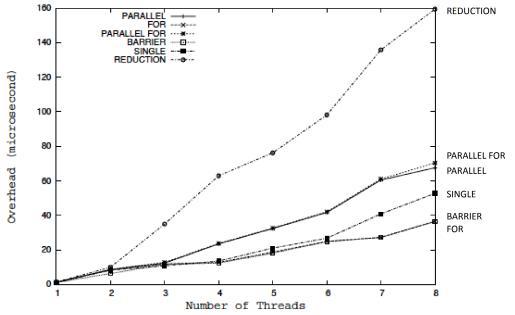


Figure 5.18: **Overheads of some OpenMP directives** – The overhead of several common directives and constructs is given in microseconds.

17

Imbalance

- Happens when the iterations in a loop require different amount of computation
- Scheduling **may** help
  - Default: 'static' each thread gets the same number of iterations

Overhead of OpenMP scheduling

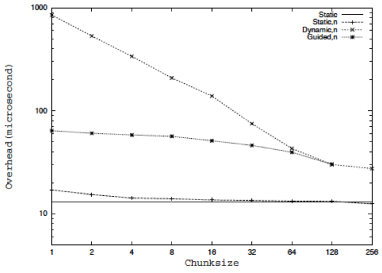
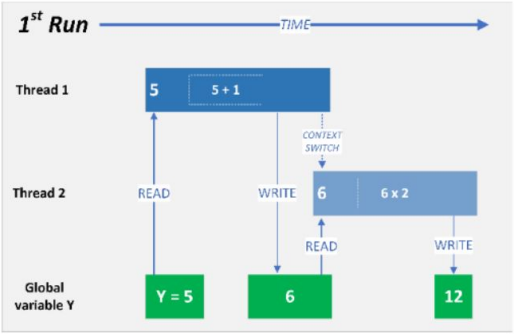
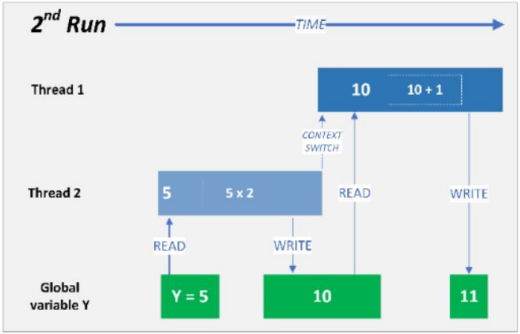


Figure 5.19: **Overhead of OpenMP scheduling** – The overheads for the different kinds of loop schedules are shown. Note that the scale to the left is logarithmic.

Racing conditions

- e.g. thread 1 does y=y+1 and thread 2 does y=y*2
- Careful when defining data scopes (no warning or error will be shown!)





Summary

- OpenMP is a cross-platform, cross-compiler API and easy to implement to existing code (C,C++,Fortran)
- Requires careful (and likely creative) division of works
- Significant overheads: Try other optimizations first!
  - Minimize division, float power
  - Stick to intrinsic functions when possible

Not all CPU operations are created equal

ithare.com

Operation	Cost in CPU Cycles
“Simple” register-register op (ADD,OR,etc.)	<1
Memory write	~1
Bypass delay: switch between integer and floating-point units	0-3
“Right” branch of “if”	1-2
Floating-point/vector addition	1-3
Multiplication (integer/float/vector)	1-7
Return error and check	1-7
L1 read	3-4
TLB miss	7-21
L2 read	10-12
“Wrong” branch of “if” (branch misprediction)	10-20
Floating-point division	10-40
128-bit vector division	10-70
Atomics/CAS	15-30
C function direct call	15-30
Integer division	15-40
C function indirect call	20-50
C++ virtual function call	30-60
L3 read	30-70
Main RAM read	100-150
NUMA: different-socket atomics/CAS (guesstimate)	100-300
NUMA: different-socket L3 read	100-300
Allocation+deallocation pair (small objects)	200-500
NUMA: different-socket main RAM read	300-500
Kernel call	1000-1500
Thread context switch (direct costs)	2000
C++ Exception thrown+caught	5000-10000
Thread context switch (total costs, including cache invalidation)	10000 - 1 million

Useful links

- OpenMP Specifications
- GCC 13 OpenMP Implementation Status

General work flow

If the performance of your code bothers you, then you may

- Profile your code
- Starting from the most time consuming part:
  - Optimize the code before considering parallelization
  - If the performance is still not satisfactory,

    check if anything is parallelizable
  - If so, parallelize it without breaking anything

    (involve anything from 'embarrassing parallelism' to rewriting the entire algorithm)
    - Check the performance and scalability (performance vs thread number)
  - Keep improving until you are satisfied (or give up)
- Move on to the next most time consuming part (until it doesn't worth it)