Part 2A: Multicore parallelism · OpenMP
Three forms of parallelism

• Multicore parallelism:
  • CPU has got *multiple streams of instructions* to process ("threads")
  • each core can do useful work

• Instruction-level parallelism:
  • each CPU core *processes its instruction stream as fast as possible*
  • all arithmetic units can do useful work in every clock cycle

• Vector operations:
  • each instruction *does multiple similar operations in parallel*
  • all “lanes” of arithmetic units do useful work
Three forms of parallelism

• Multicore parallelism:
  • CPU has got multiple streams of instructions to process (“threads”)
  • each core can do useful work

• Instruction-level parallelism:
  • each CPU core processes its instruction stream as fast as possible
  • all arithmetic units can do useful work in every clock cycle

• Vector operations:
  • each instruction does multiple similar operations in parallel
  • all “lanes” of arithmetic units do useful work
Three forms of parallelism

• **Multicore parallelism:**
  • CPU has got *multiple streams of instructions* to process (“threads”)
  • each core can do useful work

• **Instruction-level parallelism:**
  • each CPU core *processes its instruction stream as fast as possible*
  • all arithmetic units can do useful work in every clock cycle

• **Vector operations:**
  • each instruction *does multiple similar operations in parallel*
  • all “lanes” of arithmetic units do useful work
How to achieve it?

- **Multicore parallelism:**
  - we must create *multiple threads* — e.g. with OpenMP

- **Instruction-level parallelism:**
  - we must have *independent operations* in the instruction stream
  - CPU parallelizes them automatically whenever possible

- **Vector operations:**
  - we must use *vector instructions* — e.g. with vector types in GCC
Different scales

• **Multicore parallelism:**
  • very *coarse-grained*
  • executing e.g. entire subroutines in parallel
  • amount of work per independent unit:
    e.g. 1 million multiplications

• **Instruction-level parallelism:**
  • very *fine-grained*
  • executing machine language instructions in parallel
  • amount of work per independent unit:
    e.g. 1 multiplication
Multicore & multithreading

• **Assuming:**
  • we have a computer with a *4-core* CPU
  • we have a program that creates *4 threads*
  • no other program is active at the same time

• **Then:**
  • the *operating system* will do the right thing
  • each CPU core will run one thread
  • resources fully utilized
    • at least until some of the threads finish their work...
Multicore & multithreading

• More threads than cores?
  • core 1 runs thread 1 for a short while
  • operating system makes a context switch
  • core 1 runs thread 2 for a short while ...

• Fewer threads than cores?
  • some cores are simply idle
  • there is no way to use 4 cores if you run 1 program with 1 thread
Multicore & multithreading

- How to split long-running computation among multiple threads?
  - **Hard way:** use low-level primitives and do everything manually
    - pthreads
    - std::thread ...
  - **Easy way:** use high-level parallelization tools that do almost everything for you
    - *OpenMP*
    - Intel TBB ...
Using OpenMP
OpenMP parallel for loop

for (int i = 0; i < 10; ++i) {
    c(i);
}

thread 0: c(0) c(1) c(2) c(3) c(4) c(5) c(6) c(7) c(8) c(9)
OpenMP parallel for loop

```c
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
```

thread 0:  
```
c(0)  c(1)  c(2)
```

thread 1:  
```
c(3)  c(4)  c(5)
```

thread 2:  
```
c(6)  c(7)
```

thread 3:  
```
c(8)  c(9)
```
OpenMP parallel for loop

```c
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
```

Threads might do different amounts of work
a();
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
d();

thread 0:    a()  c(0)  c(1)  c(2)  d()
thread 1:    c(3)  c(4)  c(5)
thread 2:    c(6)  c(7)
thread 3:    c(8)  c(9)
d knows that c(0), c(1), ..., c(9) have already finished their work
Loop scheduling

```c
#pragma omp parallel for
  • thread 0: iterations 0, 1, ..., 9
  • thread 1: iterations 10, 11, ..., 19
```

```c
#pragma omp parallel for schedule(static,1)
  • thread 0: iterations 0, 4, 8, ..., 36
  • thread 1: iterations 1, 5, 9, ..., 37
```

```c
#pragma omp parallel for schedule(dynamic,1)
  • iterations 0, 1, 2, ..., 39 are waiting in a queue
  • whenever a thread is available, process the next iteration
```
Sample application: cheapest 2-hop path

d (input):

\[ d[] = \{ 0, 8, 2, 1, 0, 9, 4, 5, 0 \} \]

\[ r[] = \{ 0, 7, 2, 1, 0, 3, 4, 5, 0 \} \]
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
```c
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
It works!

Multithreading with OpenMP helped by a factor of 3.6

Overall 16 times faster than our starting point
Careful with OpenMP!
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
```c
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = \textit{d}[n*i + k];
            float y = \textit{d}[n*k + j];
            float z = x + y;
            v = \text{min}(v, z);
        }
        \textit{r}[n*i + j] = v;
    }
}
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        float v = infinity;
        for (int k = 0; k < n; ++k) {
            float x = d[n*i + k];
            float y = d[n*k + j];
            float z = x + y;
            v = min(v, z);
        }
        r[n*i + j] = v;
    }
}
Rules

• Private data:
  • **OK**: everything

• Shared data:
  • **OK**: multiple threads read, nobody writes
  • **OK**: only one thread touches it
  • **bad**: one thread reads, another writes
  • **bad**: multiple threads write

“Data race”
for (int i = 0; i < 10; ++i) {
    x[i + 1] = f(x[i]);
}

for (int i = 0; i < 10; ++i) {
    y[0] = f(x[i]);
}

#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    y[i] = f(x[i]);
}