Wide operands

CP1: hardware can multiply 64-bit floating-point numbers
Pipelining:
can start the next independent operation before the previous result is available

CP4: better performance for independent operations (instruction-level parallelism)
CP4: clever use of caches helps us to get data from memory to processor faster
Multiple cores

CP2: OpenMP makes it possible to use multiple cores
Many execution ports

CP4: better performance for independent operations (instruction-level parallelism)
Many execution ports

CP2: OpenMP and hyper-threading may also help if some execution ports would be idle
CP3: vector types let us perform many independent operations in parallel
Today: a little bit of theory

- Dependence and independence
- Parallel algorithms
- OpenMP features that help with divide-and-conquer algorithms
Dependency, independence, and depth

independent ≈ parallelisable
Algorithm $\approx$ circuit

d = $\text{op1}(a)$
e = $\text{op2}(a, b)$
f = $\text{op3}(c, e)$
g = $\text{op4}(d, f)$
x = $\text{op5}(g)$
Algorithm $\approx$ circuit $\approx$ dependency graph

d = op1(a)
e = op2(a, b)
f = op3(c, e)
g = op4(d, f)
x = op5(g)
Independent: can be calculated in parallel

d = op1(a)
e = op2(a, b)
f = op3(c, e)
g = op4(d, f)
x = op5(g)
Independent: can be calculated in parallel

d = op1(a)
e = op2(a, b)
f = op3(c, e)
g = op4(d, f)
x = op5(g)
Dependent: inherently sequential

d = op1(a)
e = op2(a, b)
f = op3(c, e)
g = op4(d, f)
x = op5(g)
Dependent: inherently sequential

Depth = length of the *longest path* in dependency graph

No matter how much parallelism there is: running time $\geq$ depth
All operations independent

- depth = 1
- easy to parallelise

```c
for (int i = 0; i < n; ++i) {
    x[i] = op(y[i]);
}
```
Pipelining works automatically
Can use vector operations

· and they are pipelined automatically
Can use multiple threads

- and CPU + GPU
- and multiple GPUs
- and multiple computers …
for (int i = 0; i < n; ++i) {
    x[i+1] = op(x[i]);
}

**No independence**

- depth = \( n \)
- *no opportunities for parallelisation*
Parallel algorithms

• Many commonly-used algorithms are inherently sequential
  • large depth, no opportunities for parallelism

• Key challenge: design efficient algorithms with a smaller depth
for (int i = 0; i < n; ++i) {
    s += x[i] * y[i];
}
for (int i = 0; i < n; ++i) {
    s += x[i] * y[i];
}
for (int i = 0; i < n; ++i) {
    s += x[i] * y[i];
}
for (int i = 0; i < n; ++i) {
    s += x[i] * y[i];
}
for (int i = 0; i < n; ++i) {
    s += x[i] * y[i];
}
for (int i = 0; i < n; i += 2) {
    s0 += x[i] * y[i];
    s1 += x[i+1] * y[i+1];
}

s = s0 + s1;
for (int i = 0; i < n/2; ++i) {
    s0 += x[i] * y[i];
}
for (int j = n/2; j < n; ++j) {
    s1 += x[j] * y[j];
}
s = s0 + s1;}
Algebraic flexibility

- \((a + b) + c = a + (b + c)\)
- \(\max(\max(a, b), c) = \max(a, \max(b, c))\)
Algebraic flexibility

- \(( (a + b) + c ) + d = (a + b) + (c + d) \)
- \( \max(\max(\max(a, b), c), d) = \max(\max(a, b), \max(c, d)) \)
Trade-offs

- Typical: algorithm can be made parallel, but we need to do more work
- Careful! Is it really worth it?
  - benchmark
  - energy consumption may also matter…
y[1] = x[0] + x[1]

y[2] = x[0] + ... + x[2]

y[3] = x[0] + ... + x[3]

...
Prefix sum

\[ y[1] = x[0] + x[1] \]

\[ y[2] = x[0] + \ldots + x[2] \]

\[ y[3] = x[0] + \ldots + x[3] \]

\ldots
Prefix sum

\[ y[1] = x[0] + x[1] \]
\[ y[2] = x[0] + \ldots + x[2] \]
\[ y[3] = x[0] + \ldots + x[3] \]
\[ \ldots \]
Work vs. depth

parallel version: more arithmetic operations to achieve smaller depth
Parallelism and algorithm design

• Divide and conquer often easy to parallelise
  • solve subproblems in parallel

• Exercises:
  • parallel merge sort
  • parallel quicksort
More explicit threading with OpenMP

- Often you can write OpenMP code so that you get a working sequential version by ignoring all #pragmas

- However, this is not necessary

- We can e.g. `#include <omp.h>` and use functions provided there
a();
#pragma omp parallel
{
    int i = omp_get_thread_num();
    int j = omp_get_num_threads();
    b(i, j);
}
c();
a();
#pragma omp parallel num_threads(5) 
{
    int i = omp_get_thread_num();
    int j = omp_get_num_threads();
    b(i, j);
}
c();
int p = omp_get_max_threads();
while (p > 1) {
    #pragma omp parallel num_threads(p)
    {
        int i = omp_get_thread_num();
        b(i, p);
    }
    p = (p + 1) / 2;
}

b(0, 1);
OpenMP tasking

- Many ways to say:
  “run c(0), c(1), ..., c(n-1) in parallel”
  
  - #pragma omp parallel for
  - #pragma omp parallel,
    omp_get_thread_num()

- How to say: “run a() and b() in parallel”
a();
#pragma omp parallel
{
    b();
    #pragma omp single
    {
        c();
    }
    d();
}

e();
a();
#pragma omp parallel
{
    b();
    #pragma omp single nowait
    {
        c();
    }
    d();
}
e();
a();
#pragma omp parallel
#pragma omp single
{
    b();
}
c();
a();
#pragma omp parallel
#pragma omp single
{
    b();
    #pragma omp task
c();
    #pragma omp task
d();
e();
}
f();
static void r(int x) {
    b(x);
    if (x > 0) {
        #pragma omp task
        r(x - 1);
        #pragma omp task
        r(x - 1);
    }
}

#pragma omp parallel
#pragma omp single
{
    r(2);
}
static void r(int x) {
    b(x);
    if (x > 0) {
        #pragma omp task
        r(x - 1);
        #pragma omp task
        r(x - 1);
    }
}
...
#pragma omp parallel
#pragma omp single
{
    r(3);
}
Bit-level parallelism
Instruction-level parallelism
SIMD: vector instructions
Multiple threads
GPU
GPU + CPU in parallel

long words
automatic
vector types
OpenMP
CUDA
CUDA
Bit-level parallelism

Instruction-level parallelism

SIMD: vector instructions

Multiple threads

GPU

GPU + CPU in parallel