Some features of modern CPUs and how they help us
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
CP2: OpenMP makes it possible to use multiple cores
Many execution ports

CP4: better performance for independent operations (instruction-level parallelism)
Vector instructions

CP3: vector types let us perform many independent operations in parallel
Dependency, independence, and depth

independent ≈ parallelisable
Algorithm \approx \text{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 ≈ circuit ≈ 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

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

\[
\begin{align*}
d &= \text{op1}(a) \\
e &= \text{op2}(a, b) \\
f &= \text{op3}(c, e) \\
g &= \text{op4}(d, f) \\
x &= \text{op5}(g)
\end{align*}
\]
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
for (int i = 0; i < n; ++i) {
    x[i] = op(y[i]);
}

All operations independent

- depth = 1
- easy to parallelise
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) {
    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;

odd elements

even elements
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

\[
\begin{align*}
\text{(a + b) + c} & = a + (b + c) \\
\text{max(max(a, b), c)} & = \text{max(a, max(b, c))}
\end{align*}
\]
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
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 on OpenMP

controlling parallelism
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();

b(0, 5)
b(1, 5)
b(2, 5)
b(3, 5)
b(4, 5)
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);

b(0, 4)
b(1, 4)
b(2, 4)
b(3, 4)
b(0, 2)
b(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() b() b() c() d() d() d() 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);
}

<table>
<thead>
<tr>
<th>Bit-level parallelism</th>
<th>long words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction-level parallelism</td>
<td>automatic</td>
</tr>
<tr>
<td>SIMD: vector instructions</td>
<td>vector types</td>
</tr>
<tr>
<td>Multiple threads</td>
<td>OpenMP</td>
</tr>
<tr>
<td>GPU</td>
<td>CUDA</td>
</tr>
<tr>
<td>GPU + CPU in parallel</td>
<td>CUDA</td>
</tr>
<tr>
<td>Parallelism Type</td>
<td>Code</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Bit-level parallelism</td>
<td>MF3</td>
</tr>
<tr>
<td>Instruction-level parallelism</td>
<td>CP4</td>
</tr>
<tr>
<td>SIMD: vector instructions</td>
<td>CP3</td>
</tr>
<tr>
<td>Multiple threads</td>
<td>CP2</td>
</tr>
<tr>
<td>GPU</td>
<td></td>
</tr>
<tr>
<td>GPU + CPU in parallel</td>
<td></td>
</tr>
</tbody>
</table>
Advanced bonus material details that might matter...
False sharing

- Thread A reading and writing x[0], x[2], x[4] …
- Thread B reading and writing x[1], x[3], x[5] …
- Everything in the cache happens in full cache lines (64 bytes)
- x[2k] and x[2k+1] in the same cache line
False sharing

- Thread A reading and writing \(x[0], x[2], x[4] \ldots\)
- Thread B reading and writing \(x[1], x[3], x[5] \ldots\)
- **Permitted:** your code works fine, no data races
- **Difficult for hardware:** might be good to avoid if you do this a lot in performance-critical parts
Branch prediction

• CPUs have very long pipelines
  • fetch instructions, decode, execute …

• Must know what instructions come next

• What if there are conditional branches?
  • make educated guesses based on history
Branch prediction

• **Cheap**: branches that are easy to predict
  • e.g. long for-loops

• **Expensive**: conditional branches that are taken randomly 50% of time
Prefetching

• CPU sees the instruction stream some (small) distance to the future
• Can launch future memory reads whose results will be needed soon
• What if this is not enough?
Prefetching

- **Hardware prefetching:** CPU uses heuristics to guess what might be needed later
  - e.g. detecting *linear reading*

- **Software prefetching:** programmer explicitly tells what might be needed later
  - GCC: `__builtin_prefetch`
Prefetching

- **Hardware prefetching:** CPU uses heuristics to guess what might be needed later
  - good reason to prefer linear reading
- **Software prefetching:** programmer explicitly tells what might be needed later
  - requires additional instructions, not free
Demystifying hardware

- Hardware does many things automatically
  - caches, out-of-order execution, pipelining...

- Do not just blindly assume that all this helps with your code

- Design your code so that all this helps!
Demystifying hardware

• Measure the performance

• Check the specifications of the hardware

• Do the math — is your code efficient?

• Understand why this happens!