After 50 years and 1 day of exponential growth...

Moore’s original paper was published in 19 April 1965
Today

• More about OpenMP

• Vector instructions (SIMD)

• A little bit about instruction-level parallelism and memory hierarchies (caches)
About benchmarking and reports
Suggestions

- Calculate nanoseconds per pixel
  - or clock cycles per pixel
  - typically nanosecond ≈ 2–3 clock cycles

- Calculate speed (= 1/time) as a function of the number of threads
  - compare with linear growth
nanoseconds/pixel

MF1

window $21 \times 21$ pixels

1 thread

image size in megapixels
nanoseconds/pixel

MF3

window 21 × 21 pixels

multithreaded

image size in megapixels
nanoseconds/multiplication

CP1

1000 rows
1 thread
nanoseconds/multiplication

CP4

1000 rows
multithreaded
images/second

threads

4 cores

Hyper-Threading
images/second

2000 × 2000 pixels
21 × 21 window

threads

linear

MF3
4000 × 4000 pixels
201 × 201 window
2 × 12 cores

images/second

linear

MF3
Calculating speedups

- Fair baseline: a single-threaded version (compile *without* OpenMP)
  - if you compile with OpenMP and run with `OMP_NUM_THREADS=1`, it can be much slower than a good single-threaded implementation
OpenMP: memory model

quick recap...
OpenMP memory model

- Contract between programmer & system
- Local "temporary view", global "memory"
  - threads read & write temporary view
  - may or may not be consistent with memory
- Consistency guaranteed only after a "flush"
OpenMP memory model

- Implicit “flush” e.g.:
  - when entering/leaving “parallel” regions
  - when entering/leaving “critical” regions

- Mutual exclusion:
  - for “critical” regions
int a = 0;
#pragma omp parallel
{
    #pragma omp critical
    {
        a += 1;
    }
}
Simple rules

- **Permitted (without explicit synchronisation):**
  - *multiple threads reading*, no thread writing
  - *one thread writing*, same thread reading

- **Forbidden (without explicit synchronisation):**
  - *multiple threads writing*
  - *one thread writing, another thread reading*
Simple rules

• Smallest meaningful unit = array element

• Many threads can access the same array

• Just be careful if they access the same array element
  • even if you try to manipulate different bits
OpenMP: variables

private or shared?
Two kinds of variables

• Shared variables
  • shared among all threads
  • be very careful with data races!

• Private variables
  • each thread has its own variable
  • safe and easy
// shared variable
int sum_shared = 0;
#pragma omp parallel
{
    // private variables (one for each thread)
    int sum_local = 0;
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        sum_local += i;
    }
    #pragma omp critical
    {
        sum_shared += sum_local;
    }
}
print(sum_shared);
// OK:
for (int i = 0; i < n; ++i) {
    float tmp = x[i];
    y[i] = tmp * tmp;
}

// OK:
float tmp;
for (int i = 0; i < n; ++i) {
    tmp = x[i];
    y[i] = tmp * tmp;
}
// OK:
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    float tmp = x[i];
    y[i] = tmp * tmp;
}

// Bad (data race):
float tmp;
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    tmp = x[i];
    y[i] = tmp * tmp;
}
// OK (just unnecessarily complicated):
#pragma omp parallel
{
    float tmp;
#pragma omp for
    for (int i = 0; i < n; ++i) {
        tmp = x[i];
        y[i] = tmp * tmp;
    }
}
Two kinds of variables

• Shared variables and private variables

• If necessary, you can customise this:
  • #pragma omp parallel private(x)
  • #pragma omp parallel shared(x)
  • #pragma omp parallel firstprivate(x)

• Seldom needed, defaults usually fine
Best practices

• Use subroutines!
  • much easier to avoid accidents with shared variables this way

• Keep the function with #pragmas as short as possible
  • just e.g. call another function in a for loop
OpenMP: critical sections

... and atomics
// Good, no critical section needed:
#pragma omp parallel for
for (int i = 0; i < 10000000; ++i) {
  ++v[i];
}

// Bad, very slow:
#pragma omp parallel for
for (int i = 0; i < 10000000; ++i) {
  #pragma omp critical
  {
    ++v[i];
  }
}
// Good, no critical section needed:
#pragma omp parallel for
for (int i = 0; i < 10000000; ++i) {
    ++v[i];
}

4 ms

// Bad, very slow:
#pragma omp parallel for
for (int i = 0; i < 10000000; ++i) {
    #pragma omp critical
    {
        ++v[i];
    }
}

40 000 ms
/ *Bad — no data race but undefined output:*

```c
int a = 0;
#pragma omp parallel
{
    int b;
    #pragma omp critical
    {
        b = a;
    }
    ++b;
    #pragma omp critical
    {
        a = b;
    }
}
```
// OK:
int a = 0;
#pragma omp parallel
{
    int b;
    #pragma omp critical
    {
        b = a;
        ++b;
        a = b;
    }
    a = b;
}

// Bad: same output file
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    int v = calculate(i);
    std::cout << v << std::endl;
}

// OK (but no guarantees on the order of lines)
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    int v = calculate(i);
    #pragma omp critical
    {
        std::cout << v << std::endl;
    }
}
Naming critical sections

- You can give names to critical sections:
  - `#pragma omp critical (myname)`
- Different threads can enter simultaneously critical sections with different names
- No name = the same name
```c
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    b();
    #pragma omp critical (xxx)
    {
        c();
    }
    #pragma omp critical (yyy)
    {
        d();
    }
}
```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    int v = calculate(i);
    #pragma omp critical (result)
    {
        result += v;
    }
    #pragma omp critical (output)
    {
        std::cout << v << std::endl;
    }
}
Atomic operation

- Like a tiny critical section
- Very restricted: just for e.g. updating a single variable
- Much more efficient
```c
for (int i = 0; i < n; ++i) {
    int l = v[i] % m;
    ++p[l];
}

#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    int l = v[i] % m;
    #pragma omp atomic
    ++p[l];
}

#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    int l = v[i] % m;
    #pragma omp critical
    {
        ++p[l];
    }
}
```
OpenMP: scheduling

#pragma omp for
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    c(i);
}
d();
/ / Good memory locality:
*/
// each thread scans a consecutive part of array
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
  c(x[i]);
}
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    c(i);
}
d();
a();
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < 16; ++i) {
    c(i);
}
d();

```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c(0)</td>
<td>c(4)</td>
<td>c(8)</td>
<td>c(12)</td>
</tr>
<tr>
<td>c(1)</td>
<td>c(5)</td>
<td>c(9)</td>
<td>c(13)</td>
</tr>
<tr>
<td>c(2)</td>
<td>c(6)</td>
<td>c(10)</td>
<td>c(14)</td>
</tr>
<tr>
<td>c(3)</td>
<td>c(7)</td>
<td>c(11)</td>
<td>c(15)</td>
</tr>
</tbody>
</table>
```
a();
#pragma omp parallel for dynamic
for (int i = 0; i < 16; ++i) {
    c(i);
}
d();
OpenMP scheduling

- Performance, \( n = 100\,000\,000 \):
  - sequential: 50 ms
  - parallel: 50 ms
  - schedule(static,1): 200 ms
  - schedule(dynamic): 4000 ms

```c
for (int i = 0; i < n; ++i) ++v[i];
```
OpenMP scheduling

- **Performance, \( n = 100 \, 000 \, 000 \):**
  - sequential: 800 ms
  - parallel: 300 ms
  - `schedule(static, 1)`: 300 ms
  - `schedule(dynamic)`: 4000 ms

```c
for (int i = 0; i < n; ++i) v[i] = sqrt(i);
```
OpenMP: reductions

... just a convenient shorthand
int g = 0;
#pragma omp parallel
{
    int l = 0;
    #pragma omp for
    for (int i = 0; i < n; ++i) {
        l += v[i];
    }
    #pragma omp atomic
    g += l;
}

≈

int g = 0;
#pragma omp parallel for reduction(+:g)
for (int i = 0; i < n; ++i) {
    g += v[i];
}
Vector instructions
Vector instructions

- **1997**: MMX, 64-bit registers MM0 … MM7
- **1999**: SSE, 128-bit registers XMM0 … XMM7
- **2011**: AVX, 256-bit registers YMM0 … YMM15
- **soon**: AVX-512, 512-bit registers ZMM0…ZMM31
Vector instructions

• Hundreds of machine-language instructions
  • interpret AVX registers as vectors
  • “horizontal add with saturation”
  • “conditional dot product”
  • “sum of absolute differences”
  • “fused multiply and add” …
Vector instructions

- Hundreds of machine-language instructions

- Directly available via compiler intrinsics and built-in functions, if needed
  - \( c = \_\text{mm256}_\_\text{hadd}_\text{pd}(a, \ b); \)
  - \( c = \_\_\text{builtin}_\text{ia32}_\_\text{haddpd256}(a, \ b); \)

- see the course web site for pointers
Vector instructions

• Hundreds of machine-language instructions

• Directly available via compiler intrinsics and built-in functions, if needed

• In many cases we do not need to worry about low-level details
Vector types in GCC

typedef double double4_t __attribute__((__vector_size__ (4*sizeof(double))));

// Now these are almost equivalent:
double4_t a;
double a[4];
Vector types in GCC

typedef float float8_t __attribute__((__vector_size__(8*sizeof(float))));

// Now these are almost equivalent:
float8_t a;
float a[8];
Vector types in GCC

// Can address individual elements as usual:
float8_t a;
for (int i = 0; i < 8; ++i) {
    a[i] = 123.0 + i;
}
Vector types in GCC

// Can address individual elements as usual:
float8_t a[3];
for (int j = 0; j < 3; ++j) {
    for (int i = 0; i < 8; ++i) {
        a[j][i] = 123.0 + i;
    }
}
Vector types in GCC

// Operations on entire vectors:
float8_t a, b, c;
a += b * c;

// Same as:
for (int i = 0; i < 8; ++i) {
    a[i] += b[i] * c[i];
}
Vector types in GCC

// Operations on entire vectors, also with scalars:
float8_t a, b, c;
a += b * c / 3 + 2;

// Same as:
for (int i = 0; i < 8; ++i) {
    a[i] += b[i] * c[i] / 3 + 2;
}
Vector types in GCC

- Always available
- Compiler uses special vector registers and instructions whenever possible
- Remember to specify the architecture
  - `g++ -march=native`
Memory alignment

- Vector data always properly aligned
  - memory address divisible by sizeof(vector)

- Compiler takes care of this for local variables allocated from stack

- You take care of this for arrays allocated from heap
Memory alignment

- `malloc()` not necessarily good enough
- `posix_memalign()` to allocate memory, `free()` to release

- See `common/vector.*` for helper functions
  - `float8_alloc()`, `double4_alloc()`
double4_t* x = double4_alloc(n);
double4_t* y = double4_alloc(n);

for (int i = 0; i < n; ++i) {
    for (int j = 0; j < 4; ++j) {
        x[i][j] = ...;
    }
}

for (int i = 0; i < n; ++i) {
    double4_t z = x[i];
    y[i] = z * z;
}

free(x);
free(y);
Memory alignment

- CPU vector instructions: require proper alignment
- Vector types: promise of proper alignment
- C compiler can safely generate vector instructions
for (int i = 0; i < n; ++i) {
    double4_t z = x[i];
    y[i] = z * z;
}

L42:

    vmovapd (%rbx,%rax), %ymm0
    vmulpd %ymm0, %ymm0, %ymm0
    vmovapd %ymm0, (%r12,%rax)
    addq $32, %rax
    cmpq %rdx, %rax
    jne L42

“…pd” = packed doubles = vector of doubles
“ymm…” = 256-bit register
How to exploit vector extensions
How to exploit vector instructions?

- Needs some creativity!

- Design your algorithm so that you can do the same operation for many items, in parallel

- Often some preprocessing & postprocessing needed: convert input data to suitable vectors and back
// Goal: sum of squares
s = x[0]*x[0] + ... + x[n-1]*x[n-1];

// Preprocessing: pack to vectors
v[0] = { x[0], x[1], x[2], x[3] };
v[1] = { x[4], x[5], x[6], x[7] };
...
// Pad last vector with zeroes if needed
v[m-1] = { x[n-2], x[n-1], 0, 0 };

// Calculation: each component independently in parallel
y = v[0]*v[0] + ... + v[m-1]*v[m-1];

// Postprocessing: combine components
Other examples

• Interleave input rows:
  in each vector, element $i$ comes from row $i$
  we can then process multiple rows in parallel

• Multidimensional input:
  (red, green, blue)-triples in digital images,
  multiple channels in digital audio
Efficient use of vector instructions

— instruction-level parallelism
— memory hierarchy
Toy example: sum of squares

// Repeatedly do multiply-and-add
// for an array of with “size” vectors
for (int j = 0; j < iter; ++j) {
    for (int i = 0; i < size; ++i) {
        double4_t y = v[i];
        x += y * y;
    }
}

How well does this perform?
nanoseconds/vector multiplication

array size in bytes

disappointing?
Instruction-level parallelism

• Good: parallelism in vector operations

• Bad: very little opportunities for instruction-level parallelism

• Inherently sequential:
  
  \[ x += y[0] \cdot y[0]; \ x += y[1] \cdot y[1]; \]
  
  \[ x += y[2] \cdot y[2]; \ x += y[3] \cdot y[3]; \ldots \]
**Bad**

\[ x \leftarrow y[0] \times y[0]; \]
\[ x \leftarrow y[1] \times y[1]; \]
\[ x \leftarrow y[2] \times y[2]; \]
\[ x \leftarrow y[3] \times y[3]; \]
\[ x \leftarrow y[4] \times y[4]; \]
\[ x \leftarrow y[5] \times y[5]; \]
\[ x \leftarrow y[6] \times y[6]; \]
\[ x \leftarrow y[7] \times y[7]; \]
\[ x \leftarrow y[8] \times y[8]; \]
\[
...\]

**Better**

\[ t[0] \leftarrow y[0] \times y[0]; \]
\[ t[1] \leftarrow y[1] \times y[1]; \]
\[ t[2] \leftarrow y[2] \times y[2]; \]
\[ t[3] \leftarrow y[3] \times y[3]; \]
\[ t[0] \leftarrow y[4] \times y[4]; \]
\[ t[1] \leftarrow y[5] \times y[5]; \]
\[ t[2] \leftarrow y[6] \times y[6]; \]
\[ t[3] \leftarrow y[7] \times y[7]; \]
\[ t[0] \leftarrow y[8] \times y[8]; \]
\[
...\]
// More opportunities for instruction-level parallelism
// (assuming here that “size” is a multiple of 8)
double4_t t[8];
...
for (int j = 0; j < iter; ++j) {
    for (int i = 0; i < size; i += 8) {
        for (int k = 0; k < 8; ++k) {
            double4_t y = v[i + k];
            t[k] += y * y;
        }
    }
}

for (int k = 0; k < 8; ++k) {
    x += t[k];
}

Any improvements?
Bottlenecks

- Naive version: *latency* of vector operations
- Unrolled version, small data: *throughput* of vector operations
- Unrolled version, large data: getting data from the *memory*
nanoseconds/vector multiplication

array size in bytes

L1 data 32K
L2 256K
L3 8M
memory 16G
Caches
How do caches work?

• CPU ↔ L1 ↔ L2 ↔ L3 ↔ memory

• Whenever you read memory:
  • CPU reads the full cache line (64 bytes) from the nearest cache that contains it
  • stores it in all intermediate caches, makes room by throwing away older data
Some rules of thumb

• Repeatedly work with a small chunk of <= 32KB of data:
  • all data remains in L1
  • small latency (order of 1 ns)
  • large bandwidth
Some rules of thumb

• Random reads in >> 8MB of data:
  • most memory lookups are cache misses
  • large latency (order of 100 ns)
  • small bandwidth
Some rules of thumb

• **Ideal:** linear scanning of L1

• **Good:** random access of L1, linear scanning of L2–L3

• **Tolerable:** linear scanning of main memory

• **Horrible:** random access of main memory
Some rules of thumb

• You can do useful work while you wait for data from memory

• Instruction-level parallelism does it automatically, if there are some other independent operations that you can run
Optimising cache usage

cache blocking in matrix multiplication
Arithmetic intensity

- Throughput of arithmetic operations larger than main memory bandwidth

- Whenever you read data from main memory to caches (or from caches to registers), try to do many arithmetic operations with the same data
Example: matrix multiplication

- Multiplying $n \times n$ matrixes: $O(n^2)$ data, $O(n^3)$ operations

- Naive algorithm: each operation needs to fetch new data from memory

- Better algorithm: most operations use data that is already in cache or registers
Matrix multiplication

\[
\begin{array}{ccc}
A & \times & B \\
\times & & \times \\
= & & = \\
C & & C
\end{array}
\]
Naive solution: poor locality
Cache blocking: better locality

\[
\begin{array}{c}
A \\
\times \\
B \\
= \\
C
\end{array}
\]
Reusing data in registers

- Naive: calculate 1 dot product $x_1 \cdot y_1$
- Better: calculate simultaneously 4 dot products $x_1 \cdot y_1$, $x_1 \cdot y_2$, $x_2 \cdot y_1$, $x_2 \cdot y_2$
  - read 2 times as much data
  - produce 4 times as many results
  - better *arithmetic intensity*
Reusing data in registers

• Naive: calculate 1 dot product $x_1 \cdot y_1$

• Better: calculate simultaneously 9 dot products $x_1 \cdot y_1$, ..., $x_3 \cdot y_3$
  • read 3 times as much data
  • produce 9 times as many results
  • still enough registers to keep everything...?
$A \times B = C$

$C_{11} = X_{11} + Y_{11}$
Summary

• Use *vector instructions* to better exploit parallel processing units in modern CPUs

• Pay attention to *caches*: reuse data

• Do not forget *instruction-level parallelism*

• Do not forget using *multiple threads*