Parallel processing with OpenMP

#pragma omp
Bit-level parallelism

Instruction-level parallelism

SIMD: vector instructions

Multiple threads

GPU

GPU + CPU in parallel

long words

automatic

vector types

OpenMP

CUDA

CUDA
Multi-core computers

• Modern CPUs have multiple **cores** (Maari-A: 4)

• Each core has its own execution units, runs its own **thread** of code, independently

• Each core has access to the main memory

• Some shared resources (e.g. some caches)
Multi-core programming

• Simply launch multiple threads
  • easy with OpenMP

• If you have $\leq 4$ threads running in Maari-A, each thread will be on a different core
  • if you have 1000 threads, operating system will have to do some time slicing…
OpenMP

• Extension of C, C++, Fortran

• Standardised, widely supported

• Just compile and link your code with:
  • gcc -fopenmp
  • g++ -fopenmp
OpenMP

• You add `#pragma omp` directives in your code to tell what to parallelise and how

• Compiler & operating system takes care of everything else

• You can often write your code so that it works fine even if you ignore all `#pragmas`
a();
#pragma omp parallel
{
    b();
}
c();
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
}
}
d();
a();
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
e();
a();
#pragma omp parallel
{
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
e();
```c
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
}
```

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
```
a();
#pragma omp parallel
{
    b();
    #pragma omp critical
    {
        c();
    }
}
d();
a();
#pragma omp parallel
{
    b();
#pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
#pragma omp critical
    {
        d();
    }
}
e();
global_initialisation();
#pragma omp parallel
{
    local_initialisation();
#pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        do_some_work(i);
    }
#pragma omp critical
    {
        update_global_data();
    }
}
report_result();
// 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);
OpenMP: memory model

what can be done without synchronisation?
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;
    }
}

thread 1:  0 0 0 1 1 1 ? ? ? 2
thread 2:  0 0 ? ? 1 2 2
memory:    ? 0 0 ? 1 1 ? 2 2

parallel

critical
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
Simple rules

• Safe:
  • thread 1: $p[0] = q[0] + q[1]$
Simple rules

• Safe:
  • thread 1: \( p[0] = p[0] + q[1] \)
Simple rules

• Not permitted without synchronisation:
  • thread 1: \( p[0] = q[0] + p[1] \)

• “Data race”, unspecified behaviour
Simple rules

• Not permitted without synchronisation:
  - thread 1: \( p[0] = q[0] + q[1] \)

• “Data race”, unspecified behaviour
Simple rules

• Not permitted without synchronisation:
  • thread 1: \( p[\emptyset] = 1 \)
  • thread 2: \( p[\emptyset] = 1 \)
  • thread 3: \( p[\emptyset] = 1 \)

• “Data race”, unspecified behaviour
Filtering is very easy

```c
void filter(const int* data, int* result) {
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        result[i] = compute(data[i]);
    }
}
```
Filtering is very easy

```c
static void median(const array_t x, array_t y) {
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                              x[i][ja], x[i][jb]);
        }
    }
}
```
OpenMP: variables

private or shared?
/** 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);
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
// 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: synchronisation

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];
}

// Bad, very slow:
#pragma omp parallel for
for (int i = 0; i < 10000000; ++i) {
    #pragma omp critical
    {
        ++v[i];
    }
}
// Bad — no data race but undefined output:
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;
    }
}
// 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 (mynname)

• 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
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
def (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();
a();
#pragma omp parallel for dynamic
for (int i = 0; i < 16; ++i) {
    c(i);
}
d();

<table>
<thead>
<tr>
<th>c(0)</th>
<th>c(4)</th>
<th>c(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c(1)</td>
<td>c(5)</td>
<td>c(9)</td>
</tr>
<tr>
<td>c(2)</td>
<td>c(6)</td>
<td>c(10)</td>
</tr>
<tr>
<td>c(3)</td>
<td>c(7)</td>
<td>c(11)</td>
</tr>
<tr>
<td></td>
<td>c(12)</td>
<td>c(13)</td>
</tr>
<tr>
<td></td>
<td>c(14)</td>
<td></td>
</tr>
</tbody>
</table>
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
```c
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;
}
```

≈

```c
int g = 0;
#pragma omp parallel for reduction(+:g)
for (int i = 0; i < n; ++i) {
    g += v[i];
}
```
OpenMP: speedups in practice

measure!
images/second

2000 × 2000 pixels
21 × 21 window

linear

MF1
MF2
images/second

4 cores

Hyper-Threading
images/second

2000 × 2000 pixels
21 × 21 window

MF3

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

images/second

linear

MF3
Hyper-threading

• Maari-A:
  • 4 physical cores
  • each core can run 2 hardware threads

• OpenMP defaults:
  • use $4 \times 2 = 8$ threads
Without hyper-threading

• Each core has 1 thread (1 instruction stream)

• CPU looks at the instruction stream (up to certain distance) and executes the next possible instruction
  • must be independent!
  • must have execution units available!
With hyper-threading

• Each core has 2 threads (2 instruction streams)
• CPU looks at both instruction streams
• Possibly more opportunities for finding instructions that can be executed now
Example: multiply and add

- Maari-A computers have “Ivy Bridge” CPUs
- They have independent parallel units for:
  - floating point multiplication (vectorised)
  - floating point addition (vectorised)
- Throughput: one “+” and one “×” per cycle
Example: multiply and add

- Code: ++++++... (all independent)
- 1 instruction / cycle
- Hyper-threading does not help
  - one thread keeps “+” unit busy
  - “×” unit has nothing to do
Example: multiply and add

- Code: \[\times\times\times\times\times\times\ldots\] (all independent)
- 1 instruction / cycle
- Hyper-threading does not help
  - one thread keeps “\(\times\)" unit busy
  - “+” unit has nothing to do
Example: multiply and add

- Code: \(+\times+\times+\times+\times+\times+\times+\times+\times+\ldots\) (all independent)

- 2 instructions / cycle

- Hyper-threading does not help
  - one thread enough to keep both units busy
Example: multiply and add

- Code: +++...××××××... (all independent)

- 1 instructions / cycle
  - CPU does not see far enough
  - first “+” unit busy, “×” unit idle
  - then “+” unit idle, “×” unit busy
Example: multiply and add

- Code:
  - thread 1: ++++++… (*all independent*)
  - thread 2: ×××××××… (*all independent*)

- 1 instruction / cycle without hyper-threading
- 2 instructions / cycle with hyper-threading
Example: multiply and add

• If everything is already perfectly interleaved in your code, hyper-threading does not help
  • getting no speedups may be a good sign!

• May make it easier to program
  • perfect instruction-level parallelism not necessary for maximum performance
Example: multiply and add

• Maybe a good idea to not rely too much on hyper-threading?
  • typically only more expensive CPUs support hyper-threading
  • almost the same performance with cheaper CPUs and careful implementation?
OpenMP: summary
OpenMP: summary

• Splits work across multiple threads, benefits from multiple CPU cores

• Each thread still needs to worry about e.g.:
  • instruction-level parallelism
  • vectorisation
  • getting data from memory to CPU…