Parallelism and OpenMP

Will Qian

January 23, 2015
Today’s Agenda

1. Introduction
2. Three places for parallelism
3. OpenMP
4. Theory
Multitasking for the computer

Computer : parallel computing :: you : multitasking
Using more of your computer

Serial computations are limited by:
Using more of your computer

Serial computations are limited by:

- Transistor density (physical space)
Using more of your computer

Serial computations are limited by:

- Transistor density (physical space)
- Clock speed (Joule heating)
Using more of your computer

Serial computations are limited by:

- Transistor density (physical space)
- Clock speed (Joule heating)

Hardware solution: instead of stuffing more into cores, just stuff more cores!
Why parallel?

Problems with high parallelism:
Why parallel?

Problems with high parallelism:

- Independent computations (e.g. computing $n^{th}$ powers)
Why parallel?

Problems with high parallelism:

- Independent computations (e.g. computing $n^{th}$ powers)
- Divide-and-conquer (e.g. mergesort, alpha-beta searches)
Why parallel?

Problems with high parallelism:

- Independent computations (e.g. computing $n^{th}$ powers)
- Divide-and-conquer (e.g. mergesort, alpha-beta searches)
- Multi-interface models (e.g. server-client)
Why parallel?

Problems with high parallelism:

- Independent computations (e.g. computing $n^{th}$ powers)
- Divide-and-conquer (e.g. mergesort, alpha-beta searches)
- Multi-interface models (e.g. server-client)

And many others!
Many use cases
Many use cases

- Database management systems
Many use cases

- Database management systems
- Web servers
Many use cases

- Database management systems
- Web servers
- GUI programs
- Decoding genomes
- Physics simulations
- Artificial intelligence
- Encryption and decryption
- Operating systems
- ...and so much more!
Your own computer

OS runs programs as processes:
- Each process has a thread
  - Each core can only run 1 thread at a time
Your own computer

OS runs programs as processes:

- Each process has a thread
  - Each core can only run 1 thread at a time
  - Context switching allows for more than 4 threads at a time
The concept of parallelism

Parallelism: how much of the program can be executed in parallel

- Not everything can be parallelized
- Greater parallelism means greater benefit from parallelization
- High-parallelism code is often engineered
- Must take into account overhead of spawning and managing threads
Efficiency of parallelism

\[ n = \text{number of threads}, \alpha = \text{fraction of code that is serial} \]

Amdahl’s Law:
\[ T(n) = \alpha + \frac{1 - \alpha}{n} \]

Potential (max) parallelism:
\[ \frac{T(1)}{T(\infty)} \]
Today’s Agenda

1. Introduction
2. Three places for parallelism
3. OpenMP
4. Theory
A network of computers

A cluster of machines (e.g. most supercomputer clusters)

- Method: MPI
Multiple cores on one computer

A single machine with many computing clusters (commonly CPU’s, but sometimes also GPU’s)

- Methods: many, including OpenMP (today)
One core

Instruction-level and bit-level parallelism, e.g. vectorization

- Method: compiler optimizations
Today’s Agenda

1. Introduction
2. Three places for parallelism
3. OpenMP
4. Theory
POSIX threads (pthreads)

“Ancient” way of directly spawning and managing threads.

Pros:
- Lots of control
- Works on any POSIX system
POSIX threads (pthreads)

“Ancient” way of directly spawning and managing threads.

Pros:

- Lots of control
- Works on any POSIX system

Cons:

- Lots of scaffolding for even the simplest tasks
- Really dangerous operations (involving `void*` casting)
- Incredibly difficult to debug
- Produces really ugly code
## POSIX threads (pthreads)

```c
#define N_THREADS 1000

pthread_t thread_ids[N_THREADS];
pthread_attr_t t_attr;
void *t_data[N_THREADS];

int id_numbers[N_THREADS];
```

...
int main() {
    ...
    for (i = 0; i < N_THREADS; i++) {
        id_numbers[i] = i;
        t_data[i] = rand();
        pthread_create(thread_ids + i, &t_attr, 
                        my_func, (void*)(id_numbers + i));
    }
    ...
    for (i = 0; i < N_THREADS; i++)
        pthread_join(thread_ids[i], NULL);
    ...
}
POSIX threads (pthreads)

tl;dr: pthread kind of sucks
POSIX threads (pthreads)

tl;dr: pthread kind of sucks

Alternative: Open Multi-Processing (OpenMP)
OpenMP isn’t a library: it’s used via compiler directives

#pragma omp parallel
#pragma omp parallel for
#pragma omp parallel private(var1, var2, ...)
#pragma omp parallel for private(var1, var2, ...)
#pragma omp barrier
#pragma omp critical
#pragma omp parallel for

Most basic declaration; parallelizes a for-loop

```c
#include <stdio.h>

char bits[1 << 8][9];

int main() {
    int i;
    ...
```
#pragma omp parallel for

for (i = 0; i < (1 << 8); i++) {
    int j;
    for (j = 0; j < 8; j++)
        bits[i][7 - j] = (i / (1 << j)) % 2 + '0';
    bits[i][8] = '\0';
}

...
#pragma omp parallel for

... 
for (i = 0; i < (1 << 8); i++)
    printf("%s\n", bits[i]);
return 0;
}
Compilation

Use the -fopenmp compiler flag.

```
gcc -fopenmp openmp_example.c -o example
```

or

```
g++ -fopenmp openmp_example.cpp -o example
```

No flag = no parallelism. Code will still compile and run as normal, in serial.
Sometimes, you want each thread to allocate its own memory for some variable, e.g. loop and temporary variables

```c
int i;
MyData data;

#pragma omp parallel for private(i, data)
for (i = 0; i < N; i++) {
    data = (MyData) { .i = i, .val = i * i };
    some_func(&data);
}
```
Nesting loops

You can even parallelize nested loops!

```c
int i, j;
MyData data;

#pragma omp parallel for
for (i = 0; i < N; i++) {
    #pragma omp parallel for private(i, j, data)
    for (j = 0; j < i; j++) {
        data = (MyData) { .i = i, .val = i * j };  
        some_func(&data);
    }
}
```
Thread numbers and count

You can also get thread id’s and total thread count!

```c
int id, thread_count;

#pragma omp parallel for private(id, thread_count)
for (i = 0; i < N; i++) {
    id = omp_get_thread_num();
    thread_count = omp_get_num_threads();
    printf("I’m thread %d of %d\n", id, thread_count);
}
```
Preventing race conditions

You can even declare blocks to be critical sections (serial)

```c
int i, sum = 0;

#pragma omp parallel for private(i)
for (i = 0; i < N; i++) {
    #pragma omp critical
    { // could cause race conditions
        sum += i;
    }
}
```

Use case: `std::cout` and `std::cin` aren't thread-safe.
Sometimes, you want to make an atomic write. For example, the previous slide can be written as:

```c
int i, sum = 0;

#pragma omp parallel for private(i)
for (i = 0; i < N; i++) {
    #pragma omp atomic
    sum += i;
}
```

Note: `atomic` only affects the next write (not necessarily statement)!
Memory barriers (a.k.a. fences) will block any thread from progressing until all threads get to it (e.g. a “checkpoint”)

```c
int i, sum = 0;

#pragma omp parallel private(i)
for (i = 0; i < N; i++) {
    #pragma omp atomic
    sum += i;

    #pragma omp barrier
    printf("sum = %d\n", sum);
}
```
Memory barriers

Caveat: this is a memory fence on physical threads. So, if your machine only has 2 actual threads, it will only block 2 threads at a time.
Other cool features

OpenMP also supports:

- Reducers: a safer way to aggregate data into a single construct
- Scheduling control: allows more manual control over work scheduling
- Master/slave: allows execution of code only on master thread

Comparison to pthreads

Pros:

- OpenMP is much safer than pthreads
- OpenMP makes code a lot cleaner
- OpenMP makes it a lot easier to do common things (e.g. loops)

Cons:

- OpenMP gives up some control
- It’s easier to manually optimize using pthreads
- OpenMP is slightly more costly in terms of overhead
Comparison to MPI

Pros:

- OpenMP is cleaner
- OpenMP does not require machine cross-compatibility
- OpenMP has a much lower overhead

Cons:

- MPI’s powerfulness is bounded by number of machines
- MPI allows use of resources more powerful than master
- MPI can explicitly exclude master thread
Comparison to CUDA

Pros:

- More computers support POSIX than have NVIDIA’s
- OpenMP is bundled with gcc
- Coding in CUDA is kind of like coding with pthreads

Cons:

- Many more GPU’s than CPU’s on any given machine
- CUDA is incredibly fast
Comparison to C++11

Pros:

- OpenMP works on C++ versions prior to C++11
- OpenMP produces (arguably) cleaner code
- C++11’s parallelism libraries aren’t fully-fledged

Cons:

- C++11 libraries make parallelism more object-oriented
- C++11 has a lot better support for things like mutex’s, atomics, thread variables
Comparison to CILK+

Pros:

- OpenMP has support for C++ (CILK+ currently does not)
- OpenMP is bundled with gcc

Cons:

- CILK+ is even easier to use than OpenMP
- CILK+ has a lot of great parallel debugging tools
- CILK+ has even better support for reducers
- CILK+ is likely to become instruction-level on some machines in the near future
Comparison to transactional memory

Pros:

- OpenMP is available on more systems than TM is
- OpenMP is already popularized
- Software TM is slow, hardware TM is expensive ($$$)

Cons:

- TM has lock elision
- TM makes it much easier to convert serial to parallel
- TM is a lot easier to code in and understand
Today’s Agenda

1. Introduction
2. Three places for parallelism
3. OpenMP
4. Theory
Work

Work is a metric representing the total volume of what needs to be done.

\[ \text{work} = T(1) \]
Span is the minimum of the number of steps of the most time-consuming branch of a parallel program.

\[ \text{span} = T(\infty) \]
Parallelism is the maximum speedup possible, and is represented as

$$\text{parallelism} = \frac{\text{work}}{\text{span}} = \frac{T(1)}{T(\infty)}$$

In particular, parallelism is bounded from above by the number of participating threads.
Sequential consistency

Roughly speaking, sequential consistency is where, whenever a thread reads, it is consistent with *some* serial execution, where other threads’ code are executed “atomically.”
Serializability

At any given time, any value read is correct and consistent with some overall serial execution (i.e. as if there were only 1 thread).
Deadlock-freedom and starvation-freedom

Deadlock: when two threads interdepend on one another, and neither can make progress

Starvation: one thread consistently does not get to execute its code

Deadlock-free: some thread always makes progress

Starvation-free: all threads make progress
Lock-freedom and wait-freedom

Lock-free: deadlock-free and without locks

Wait-free: starvation-free and without locks; this is very desirable, since locks are annoying to manage
More parallel computing at MIT

6.172: Performance Engineering; low-level optimizations using C and parallel
6.886: Performance Engineering for Multicore Applications; 6.172 continued

6.816: (6.836 grad) Multicore Programming; fundamental theories behind parallel systems, taught in Java

6.338: (18.337J) Parallel Computing; parallel computing and big data analysis, taught in Julia

6.846: Parallel Computing; parallel and multicore systems and architectures, taught in C/C++

6.824: Distributed Systems; cluster-level and network communications and security, taught in Go