CS 5220: Shared memory programming

David Bindel

2017-09-26

Message passing pain

Common message passing pattern

- Logical *global* structure
- *Local* representation per processor
- Local data may have redundancy
  - Example: Data in ghost cells
  - Example: Replicated book-keeping data (`pidx` in our code)

Big pain point:

- Thinking about many partly-overlapping representations
- Maintaining consistent picture across processes

Wouldn't it be nice to have just one representation?

Shared memory vs message passing

- Implicit communication via memory vs explicit messages
- Still need separate global vs local picture?
    - **No:** One thread-safe data structure may be easier
    - **Yes:** More sharing can hurt performance
        - Synchronization costs cycles even with no contention
        - Contention for locks reduces parallelism
        - Cache coherency can slow even non-contending access
- "Easy" approach: add multi-threading to serial code
- Better performance: design like a message-passing code

Let's dig a little deeper into the hardware side of this...

Memory model

- Single processor: return last write
    - What about DMA and memory-mapped I/O?
- Simplest generalization: *sequential consistency* – as if
    - Each process runs in program order
    - Instructions from different processes are interleaved
    - Interleaved instructions ran on one processor

Sequential consistency

A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

– Lamport, 1979

Example: Spin lock

Initially, `flag = 0` and `sum = 0`

Processor 1:

```
sum += p1;
flag = 1;
```

Processor 2:

```
while (!flag);
sum += p2;
```

Example: Spin lock

Initially, `flag = 0` and `sum = 0`

Processor 1:

```
sum += p1;
flag = 1;
```

Processor 2:

```
while (!flag);
sum += p2;
```

Without sequential consistency support, what if

1. Processor 2 caches `flag`?
2. Compiler optimizes away loop?
3. Compiler reorders assignments on P1?

Starts to look restrictive!

Sequential consistency: the good, the bad, the ugly

Program behavior is “intuitive”:

- Nobody sees garbage values
- Time always moves forward

One issue is *cache coherence*:

- Coherence: different copies, same value
- Requires (nontrivial) hardware support

Also an issue for optimizing compiler!

There are cheaper *relaxed* consistency models.

Snoopy bus protocol

Basic idea:

- Broadcast operations on memory bus
- Cache controllers “snoop” on all bus transactions
  - Memory writes induce serial order
  - Act to enforce coherence (invalidate, update, etc)

Problems:

- Bus bandwidth limits scaling
- Contending writes are slow

There are other protocol options (e.g. directory-based).
But usually give up on *full* sequential consistency.

Weakening sequential consistency

Try to reduce to the *true* cost of sharing

- `volatile` tells compiler when to worry about sharing
- Memory fences tell when to force consistency
- Synchronization primitives (lock/unlock) include fences

Sharing

True sharing:

- Frequent writes cause a bottleneck.
- Idea: make independent copies (if possible).
- Example problem: malloc/free data structure.

False sharing:

- Distinct variables on same cache block
- Idea: make processor memory contiguous (if possible)
- Example problem: array of ints, one per processor

Take-home message

- Sequentially consistent shared memory is a useful idea…
    - “Natural” analogue to serial case
    - Architects work hard to support it
- … but implementation is costly!
    - Makes life hard for optimizing compilers
    - Coherence traffic slows things down
    - Helps to limit sharing

Have to think about these things to get good performance.

Back to software...

Reminder: Shared memory programming model

Program consists of *threads* of control.

- Can be created dynamically
- Each has private variables (e.g. local)
- Each has shared variables (e.g. heap)
- Communication through shared variables
- Coordinate by synchronizing on variables
- Examples: pthreads, OpenMP, Cilk, Java threads

Wait, what's a thread?

Processes have *separate state*. Threads share *some*:

- Instruction pointer (per thread)
- Register file (per thread)
- Call stack (per thread)
- Heap memory (shared)

Mechanisms for thread birth/death

- Statically allocate threads at start
- Fork/join (pthreads)
- Fork detached threads (pthreads)
- Cobegin/coend (OpenMP?)
    - Like fork/join, but lexically scoped
- Futures
    - `v = future(somefun(x))`
    - Attempts to use `v` wait on evaluation

Mechanisms for synchronization

- Locks/mutexes (enforce mutual exclusion)
- Condition variables (notification)
- Monitors (like locks with lexical scoping)
- Barriers
- Atomic operations

Getting more concrete...

OpenMP: Open spec for MultiProcessing

- Standard API for multi-threaded code
    - Only a spec — multiple implementations
    - Lightweight syntax
    - C or Fortran (with appropriate compiler support)
- High level:
    - Preprocessor/compiler directives (80%)
    - Library calls (19%)
    - Environment variables (1%)
- Basic syntax: `#omp` *construct* `[`*clause* `...]`
    - Usually affects structured block (one way in/out)
    - OK to have `exit()` in such a block

A logistical note

A practical aside…

- Intel has OpenMP support by default

```
icc -c -qopenmp foo.c
icc -o -qopenmp mycode.x foo.o
```

- GCC has OpenMP support by default (several years)

```
gcc -c -fopenmp foo.c
gcc -o -fopenmp mycode.x foo.o
```

- LLVM has OpenMP support by default (more recent)

```
clang -c -fopenmp foo.c
clang -o -fopenmp mycode.x foo.o
```

- Apple LLVM *does not* support OpenMP
  - And `gcc` on OS X now aliases `clang`
  - Can use Hombrew for alternate compiler

Parallel “hello world”

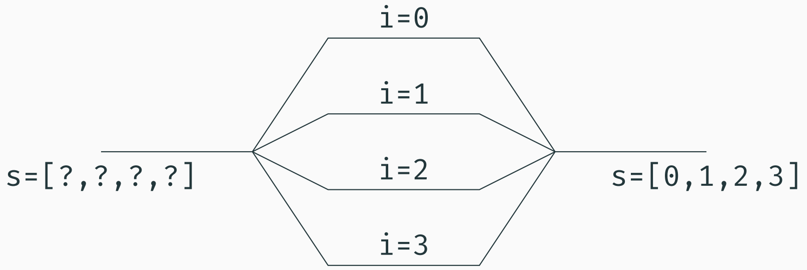
```c
#include <stdio.h>
#include <omp.h>

int main()
{
    #pragma omp parallel
    printf("Hello world from %d\n",
           omp_get_thread_num());

    return 0;
}
```

Shared and private

Annotations distinguish between different types of sharing:

- `shared(x)` (default): One `x` shared everywhere
- `private(x)`: Thread gets own `x` (indep. of master)
- `firstprivate(x)`: Each thread gets its own `x`, initialized by `x` from before parallel region
- `lastprivate(x)`: After the parallel region, private `x` set to the value last left by one of the threads
- `reduction(op:x)`: Does reduction on all thread `x` on exit of parallel region

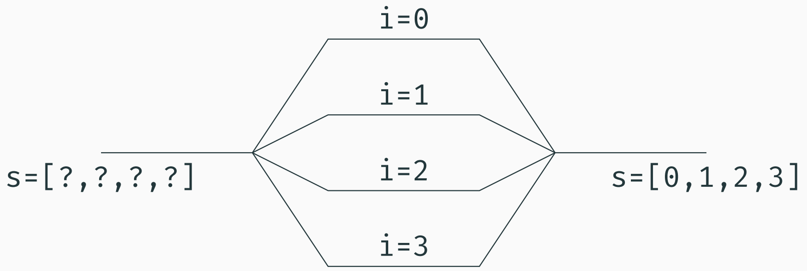
Parallel regions



Parallel region

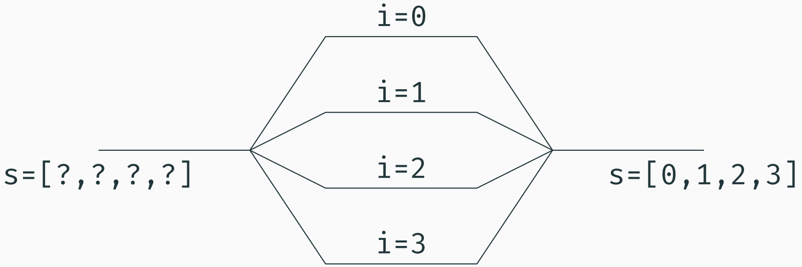
- Basic model: fork-join
- Each thread runs same code block
- Annotations distinguish shared (*s*) and private (*i*) data
- *Relaxed consistency* for shared data

Parallel regions



```
double s[MAX_THREADS];
int i;
#pragma omp parallel shared(s) private(i)
{
  i = omp_get_thread_num();
  s[i] = i;
}
// Implicit barrier here
```

Parallel regions



```
double s[MAX_THREADS];  // default shared
#pragma omp parallel
{
  int i = omp_get_thread_num();  // local, so private
  s[i] = i;
}
// Implicit barrier here
```

Parallel regions

Several ways to control num threads

- Default: System chooses (= number processors?)
- Environment: `export OMP_NUM_THREADS=4`
- Function call: `omp_set_num_threads(4)`
- Clause: `#pragma omp parallel num_threads(4)`

Can also nest parallel regions.

Parallel regions

What to do with parallel regions alone? Maybe Monte Carlo:

```
double result = 0;
#pragma omp parallel reduction(+:result)
  result = run_mc(trials) / omp_get_num_threads();
printf("Final result: %f\n", result);
```

Anything more interesting needs synchronization.

OpenMP synchronization

High-level synchronization:

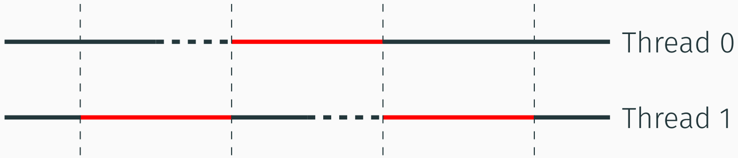
- `critical`: Critical sections
- `atomic`: Atomic update
- `barrier`: Barrier
- `ordered`: Ordered access (later)

Low-level synchronization:

- `flush`
- Locks (simple and nested

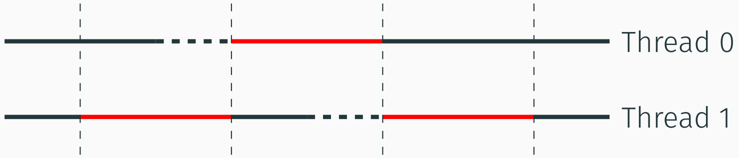
We will stay high-level.

Critical sections



- Automatically lock/unlock at ends of *critical section*
- Automatically memory flushes for consistency
- Locks are still there if you really need them...

Critical sections



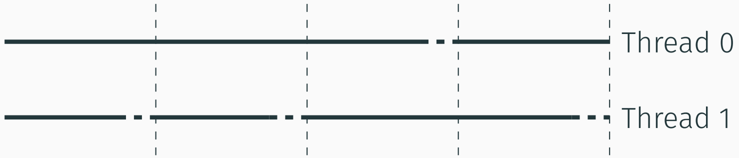
```
#pragma omp parallel
{
  ...
  #pragma omp critical my_data_cs
  {
    ... modify data structure here ...
  }
}
```

Atomic updates

```
#pragma omp parallel
{
  ...
  double my_piece = foo();
  #pragma omp atomic
  x += my_piece;
}
```

Only simple ops: increment/decrement or `x += expr` and co

Barriers



```
#pragma omp parallel
for (i = 0; i < nsteps; ++i) {
  do_stuff
  #pragma omp barrier
}
```

Is this all?

Next time:

- Parallel loop constructs
- Task-based parallelism (OpenMP 3.0+)
- Other parallel work divisions

But for now, let’s do some examples!