CS 5220

Shared memory

David Bindel

2024-10-01

Logistics

Logistics

- HW1 and P1 due tonight
    - Spend time on your writeup!
- P2 posted today
    - Lesson from P1: start early!

Shared memory

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

Message passing pain

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?

Shared memory vs message passing

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

Shared memory vs message passing

Still need separate global vs local picture?

- “Easy” approach: add multi-threading to serial code
- Better performance: design like a message-passing code

Let’s dig a little deeper on the HW side

Memory model

- Single processor: return last write
    - What about DMA and memory-mapped I/O?
- Simplest generalization: *sequential consistency*

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*

Sequential consistency

Program behaves *as if*:

- Each process runs in program order
- Instructions from different processes are interleaved
- Interleaved instructions ran on one processor

Example: Spin lock

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

Processor 1:

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

Processor 2:

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

Example: Spin lock

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

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)

Snoopy bus protocol

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
- Atomic operations do reads/writes as a single op
- Memory fences tell when to force consistency
- Synchronization primitives (lock/unlock) include fences

Unprotected data races give undefined behavior.

Sharing

True sharing:

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

Sharing

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.

Programming model

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, C11 threads, 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)

Wait, what’s a thread?

- Threads for parallelism
- Threads for concurrency

Mechanisms for thread birth/death

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

Mechanisms for synchronization

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

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

```
# Intel compiler
icc -c -qopenmp foo.c
icc -o -qopenmp mycode.x foo.o

# GCC
gcc -c -fopenmp foo.c
gcc -o -fopenmp mycode.x foo.o

# LLVM / CLang (Linux)
clang -c -fopenmp foo.c
clang -o -fopenmp mycode.x foo.o

# Apple LLVM / CLang (with libomp via Homebrew)
clang -c -Xpreprocessor -fopenmp foo.c
clang -o -Xpreprocessor -fopenmp foo.o -lomp
```

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;
}
```

Data in parallel regions

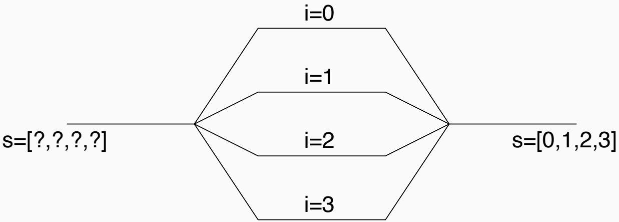
- Basic model: fork-join / cobegin-coend
- Each thread runs same code block
- Annotations distinguish shared/private data
- *Relaxed consistency* for shared data

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 (used in loops and parallel sections)
- `reduction(op:x)`: Does reduction on all thread `x` on exit of parallel region

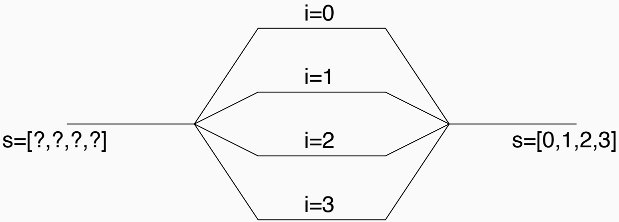
Parallel regions



Parallel region

```
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



Parallel region

```
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 cores?)
- 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)

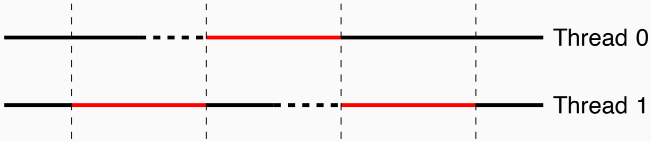
OpenMP synchronization

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 ...
    }
}
```

Critical sections

```c
void list_push(link_t** l, int data)
{
    link_t* link = (link_t*) malloc(sizeof(link_t));
    link->data = data;
    #pragma omp critical(list_cs)
    {
        link->next = *l;
        *l = link;
    }
}
```

Atomic updates

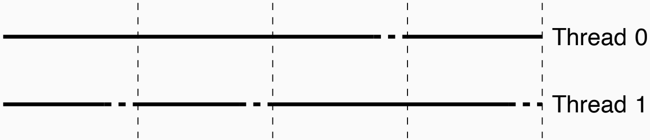
```
#pragma omp parallel
{
    // ...
    double my_piece = foo();
    #pragma omp atomic
    x += my_piece;
}
```

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

Atomic captures

```
void list_push2(link_t** l, int data)
{
    link_t* link = (link_t*) malloc(sizeof(link_t));
    link->data = data;
    #pragma omp atomic capture
    {
        link->next = *l;
        *l = link;
    }
}
```

Barriers



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

Work sharing

Work sharing constructs split work across a team

- Parallel `for`: split by loop iterations
- `sections`: non-iterative tasks
- `single`: only one thread executes (synchronized)
- `master`: master executes, others skip (no sync)

Parallel iteration

Idea: Map **independent** iterations onto different threads

```
#pragma omp parallel for
for (int i = 0; i < N; ++i)
    a[i] += b[i];

#pragma omp parallel
{
    // Stuff can go here...
    #pragma omp for
    for (int i = 0; i < N; ++i)
        a[i] += b[i];
}
```

Implicit barrier at end of loop (unless nowait clause)

Parallel iteration

The iteration can also go across a higher-dim index set

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        a[i*M+j] = foo(i,j);
```

Restrictions

`for` loop must be in "canonical form"

- Loop var is an integer, pointer, random access iterator (C++)
- Test compares loop var to loop-invariant expression
- Increment or decrement by a loop-invariant expression
- No code between loop starts in collapse set
- Needed to split iteration space (maybe in advance)

Restrictions

- Iterations should be independent
    - Compiler may not stop you if you screw this up!
- Iterations may be assigned out-of-order on one thread!
    - Unless the loop is declared monotonic

Reduction loops

How might we parallelize something like this?

```
double sum = 0;
for (int i = 0; i < N; ++i)
    sum += big_hairy_computation(i);
```

Reduction loops

How might we parallelize something like this?

```c
double sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; ++i)
    sum = big_hairy_computation(i);
```

Ordered

OK, what about something like this?

```
for (int i = 0; i < N; ++i) {
    int result = big_hairy_computation(i);
    add_to_queue(q, result);
}
```

Work is *mostly* independent, but not wholly.

Ordered

Solution: ordered directive in loop with ordered clause

```
#pragma omp parallel for ordered
for (int i = 0; i < N; ++i) {
    int result = big_hairy_computation(i);
    #pragma ordered
    add_to_queue(q, result);
}
```

Ensures the ordered code executes in loop order.

Parallel loop scheduling

Partition index space different ways:

- `static[(chunk)]`: decide at start; default chunk is n/nthreads. Lowest overhead, most potential imbalance.
- `dynamic[(chunk)]`: each takes chunk (default 1) iterations when it has time. Higher overhead, auto balances.
- `guided`: take chunks of size unassigned iterations/threads; get smaller toward end of loop. Between static and dynamic.
- `auto`: up to the system!

Default behavior is implementation-dependent.

SIMD loops

As of OpenMP 4.0:

```
#pragma omp simd reduction(+:sum) aligned(a:64)
for (int i = 0; i < N; ++i) {
    a[i] = b[i] * c[i];
    sum = sum + a[i];
}
```

SIMD loops

Can also declare vectorized functions:

```c
#pragma omp declare simd
float myfunc(float a, float b, float c)
{
    return a*b + c;
}
```

Other parallel work divisions

- `sections`: like cobegin/coend
- `single`: do only in one thread (e.g. I/O)
- `master`: do only in master thread; others skip

Sections

```
#pragma omp parallel
{
    #pragma omp sections nowait
    {
        #pragma omp section
        do_something();

        #pragma omp section
        and_something_else();

        #pragma omp section
        and_this_too();
        // No implicit barrier here
    }
    // Implicit barrier here
}
```

Task-based parallelism

- Work-sharing so far is rather limited
    - Work cannot be produced/consumed dynamically
    - Fine for data parallel array processing...
    - ... but what about tree walks and such?
- Alternate approach (OpenMP 3.0+): Tasks

Tasks

Task involves:

- Task construct: task directive plus structured block
- Task: Task construct + data

Tasks are handled by run time, complete at barriers or `taskwait`.

Example: List traversal

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        for (link_t* link = head; link; link = link->next)
            #pragma omp task firstprivate(link)
            process(link);
    }
    // Implicit barrier
}
```

One thread generates tasks, others execute them.

Example: Tree traversal

```
int tree_max(node_t* n)
{
    int lmax, rmax;
    if (n->is_leaf)
        return n->value;

    #pragma omp task shared(lmax)
        lmax = tree_max(n->l);
    #pragma omp task shared(rmax)
        rmax = tree_max(n->l);
    #pragma omp taskwait

    return max(lmax, rmax);
}
```

The `taskwait` waits for all child tasks

Task dependencies

What happens if one task produces what another needs?

```
#pragma omp task depend(out:x)
x = foo();
#pragma omp task depend(in:x)
y = bar(x);
```

Topics not addressed

- Low-level synchronization (locks, flush)
- OpenMP 4.x constructs for accelerator interaction
- A variety of more specialized clauses

See http://www.openmp.org/

A final reminder

Parallelism is not performance!