CS420 – Lecture 6

Marc Snir

Fall 2018

Quiz 1

Q1:Prefetching cannot hurt performance.
True
False

Question 2: All dependencies can be removed by register-renaming.
True
False

Question 3: What is the typical latency for accessing memory?
1 cycle
2-4 cycles
100-200 cycles
10-15 cycles

Q1:Prefetching cannot hurt performance.
True
False

Question 2: All dependencies can be removed by register-renaming.
True
False

Question 3: What is the typical latency for accessing memory?
1 cycle
2-4 cycles
100-200 cycles
10-15 cycles

Q1:Prefetching cannot hurt performance.
True
False

Question 2: All dependencies can be removed by register-renaming.
True
False
Only false dependencies can be removed, not RAW

Question 3: What is the typical latency for accessing memory?
1 cycle
2-4 cycles
100-200 cycles
10-15 cycles

Q1:Prefetching cannot hurt performance.
True
False

Question 2: All dependencies can be removed by register-renaming.
True
False
Only false dependencies can be removed, not RAW

Question 3: What is the typical latency for accessing memory?
1 cycle
2-4 cycles
100-200 cycles
10-15 cycles
Can be even higher

Question 4: Please choose the most appropriate option from the following which describes why pipelining improves performance.

Decreases latency

Increases latency

Increases bandwidth

Decreases bandwidth

Question 4: Please choose the most appropriate option from the following which describes why pipelining improves performance.

Decreases latency

Increases latency

Increases bandwidth

Decreases bandwidth

Latency often increases

Question 5: How many cycles will the following code take to execute? Please assume that each stage of the pipeline takes 1 cycle to execute and there are two separate ALUs for addition (3-stage) and multiplication (5-stage), but the other hardware units are shared. One instruction can be issued at every cycle. The format of an instruction is: op dest src1 src2

Code:
add R2 R1 R1
mult R2 R1 R2
add R3 R3 R2
mult R4 R4 R2

issue add 1
issue mult 2
issue add 3
issue mult 4
add 2,3,4
mult 5,6,7,8,9
add 10,11,12
mult 10,11,12,13,14

Question 6: A computer has a 256 KByte, 4-way set associative data cache with block size of 32 Bytes. The processor sends 32 bit addresses to the cache controller.
The number of bits in the tag field of an address is
11
16
17
27

Question 6: A computer has a 256 KByte, 4-way set associative data cache with block size of 32 Bytes. The processor sends 32 bit addresses to the cache controller.
The number of bits in the tag field of an address is
11
16
17
27
The cache contains 256K/32 = 8K lines, hence 2K sets.

- Option A: No hashing – line address determines including set: 5 bits address byte within line, 11 bits address set; the remaining 32-11-5=16 bits form the tag
- Option B: Hashing – a line can be in any set. 32-5=27 bits are used as tag.

false sharing

```
#include <omp.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#define N 10000000
int a[2][16]
  __attribute__((aligned(64)));
int main () {
  int k;
  time_t time;
  omp_set_num_threads(2);
  for (k=0;k<10;k++) {
    time=clock();
    #pragma omp parallel
    {
      int i,j;
      j=omp_get_thread_num();
      for(i=0;i<N;i++)
      a[j][0] = a[j][]+1;
    }
    time=clock()-time;
    printf("%ld ",
      time*1000/CLOCKS_PER_SEC);
    }
  printf("\n");
}
```

false sharing

```
j=omp_get_thread_num();
for(i=0;i<N;i++)
a[j][0] = a[j][1]+1;
```

```
j=omp_get_thread_num();
for(i=0;i<N;i++)
a[0][j] = a[1][j]+1;
```

false sharing

```
j=omp_get_thread_num();
for(i=0;i<N;i++)
a[j][0] = a[j][1]+1;
```

No false sharing

```
j=omp_get_thread_num();
for(i=0;i<N;i++)
a[0][j] = a[1][j]+1;
```

False sharing

false sharing

```
j=omp_get_thread_num();
for(i=0;i<N;i++)
a[j][0] = a[j][1]+1;
```

No false sharing
Running time: 50.7 ms ±2.1

```
j=omp_get_thread_num();
for(i=0;i<N;i++)
a[0][j] = a[1][j]+1;
```

False sharing
Running time: 91.1 ms ±7.4

Even worse false sharing

Each thread executes

```
j=omp_get_thread_num();
for(i=0;i<N;i++)
a[j][j] = a[1-j][j]+1;
```

Running time:204.2±4.3

Can you figure out why this code has worse performance?

- Can broadcast the master value to the local copies of private variable at the entry to the parallel section

```
int first=5;
omp_set_num_threads(2);
#pragma omp parallel \
  firstprivate(first)
{
  int myid = omp_get_thread_num();
  printf("at thread %d, first=%d,
         \n", myid, first);
  first=myid;
  printf("at thread %d, first=%d,
         \n", myid, first);
}
printf("when done first=%d \n", first);
```

output is:

```
at thread 0, first=5,
at thread 1, first=5,
at thread 0, first=0,
at thread 1, first=1,
when done first=5
```

Shared memory Semantics

- Always true: local operations appear *to local thread* to execute in program order

Shared memory Semantics

- Always true: local operations appear *to local thread* to execute in program order
- *Sequential consistency:* Operations appear to all threads to execute in same order (operations from different threads can interleave arbitrarily).

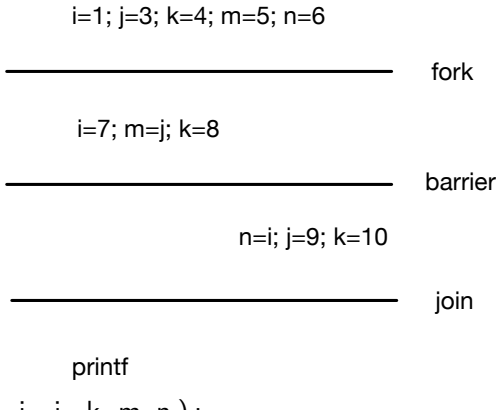
Shared memory Semantics

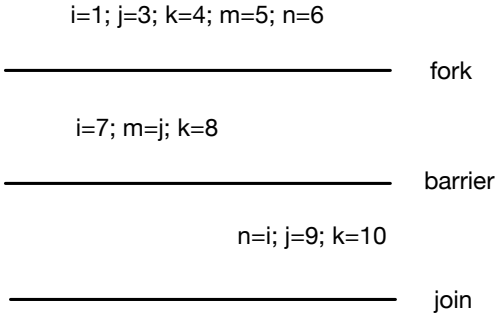
- Always true: local operations appear *to local thread* to execute in program order
- *Sequential consistency:* Operations appear to all threads to execute in same order (operations from different threads can interleave arbitrarily).
- *Most processors ARE NOT sequentially consistent*

Shared memory Semantics

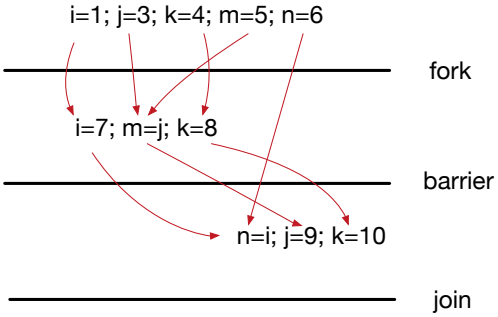
- Always true: local operations appear *to local thread* to execute in program order
- *Sequential consistency:* Operations appear to all threads to execute in same order (operations from different threads can interleave arbitrarily).
- *Most processors ARE NOT sequentially consistent*
- User should write only *race-free* code: If two threads perform conflicting accesses to a shared variable, then the accesses must be explicitly ordered by an OpenMP synchronization operation
- The OpenMP compiler and runtime will make sure that properly synchronized accesses appear to occur in the right order

```
...
int i=1,j=3,k=4,m=5,n=6;
omp_set_num_threads(2);
#pragma omp parallel
{
if(omp_get_thread_num()==0) {
i=7;m=j;k=8;
#pragma omp barrier
}
else {
#pragma omp barrier
n=i;j=9;k=10;
}
}
printf("i=%d,j=%d,k=%d,m=%d,n=%d \n",i,j,k,m,n);
}
```





output is: i=7,j=9,k=10,m=3,n=7



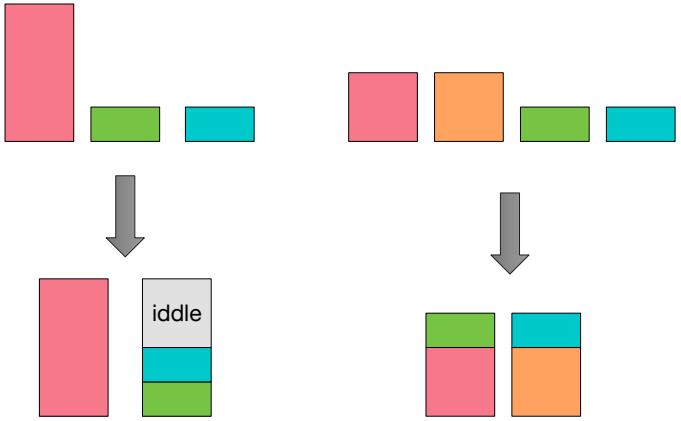
Synchronization

- Ordering constructs: barrier, fork, join
- Mutual Exclusion constructs: atomic, critical, lock

Parallel Loops

Work sharing

- The `parallel` construct assigns an *implicit task* (the execution of a structured block of code) to each thread.
  - Good: Programmer is in full control of what each thread executes
  - Bad: Programmer needs to fully control what each thread executes
- *Load balancing:* Ensure that each thread has equal amount of work (assuming they all run at same speed)



Work sharing

Assume there is a bag of independent tasks to execute in parallel

- Allocate them to threads statically (as for sum of square example)
  - Good if know how long each task will run
- Allocate them dynamically, at run time
  - Let the system do it (use *work sharing* constructs)

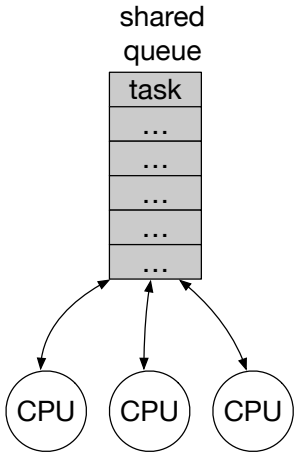
```
...
#pragma omp parallel for \
                    reduction(+:sum)
  for(int i=0; i<N; i++)
    sum+=i*i;
...
```

- The system will allocate iterates of the loop to running threads using some scheduling policy
  - Loops need be of simple form

How is work sharing done?

Simplemost: Shared work queue

- Parallel section start: iterates are (virtually) queued
- Threads pick work to execute from queue
- Parallel section ends when queue is empty and all threads are done (each tried to get work from empty queue)



Work sharing tradeoffs

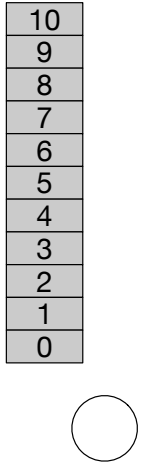
- Need tasks large enough in order to amortize the overhead of scheduling a task
  - Rule of thumb: 1000's of instructions; one iteration in our example is much too small
- Need tasks small enough so that load balancing works well
  - Rule of thumb: If execution time of tasks is not fixed, then number of tasks should be a small multiple of number of threads: *over-decomposition*

Loop schedule

Assume $N = 11$, numthreads=2. Tasks (iterates) are ordered in queue

- Allocation of iterates to threads is controlled by a `schedule` clause

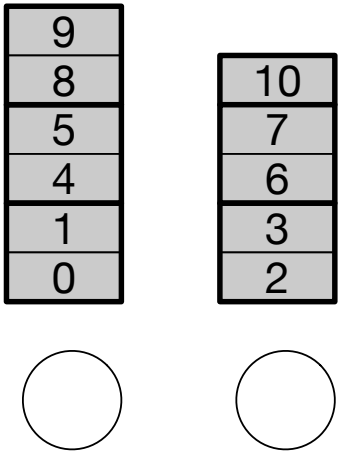
```
...
  #pragma omp parallel for \
       schedule(static, chunk_size) \
     reduction(+:sum)
    for(int i=0; i<N; i++)
      sum+=i*i;
 ...
```



Static

```
...
  #pragma omp parallel for \
      schedule(static,2)
    for(i=0;i<N;i++)
        ...
```

- Iterates are allocated round robin, in chunks of 2, to the threads
- Best when iterates (and threads) are all the same – low scheduling overhead
- Same allocation at each execution (with same number of threads)

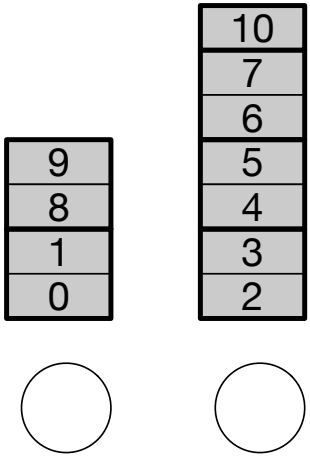


Dynamic

```
...
#pragma omp parallel for \
    schedule(dynamic,2)
  for(i=0;i<N;i++)
      ...
```

- threads are dynamically picking iterates, in chunks of 2
- Best for load balancing, but higher scheduling overhead

One possible execution

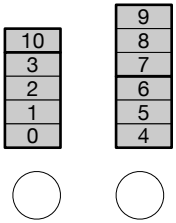


Guided

One possible execution (assuming constant of proportionality 0.7)

```
...
#pragma omp parallel for \
    schedule(guided,2)
  for(i=0;i<N;i++)
      ...
```

- threads are dynamically picking iterates, in chunks of at least two. Number of iterates picked is proportional to number of iterates left divided by numthreads
- Compromise between static and dynamic



Auto, runtime

```
...
#pragma omp parallel for \
    schedule(auto)
for(i=0;i<N;i++)
    ...
```

- I have no idea what's the right schedule; compiler and runtime will do what they think best

```
...
#pragma omp parallel for \
    schedule(runtime)
for(i=0;i<N;i++)
    ...
```

- I delay to runtime the choice of the schedule

What's the difference between auto and runtime?

- Various OpenMP *Internal Control Variables* (ICV) can be set at runtime. Example: number of active threads
  - Environment variable `OMP_NUM_THREADS` can be set before an OpenMP program starts executing; this will be the default number of threads for the program execution
  - `omp_set_num_threads()` can be called inside a program; it will set the number of threads for current team
  - `omp_get_num_threads()` can be called to query the current value of the ICV.
- Example: default schedule for parallel loops
  `OMP_SCHEDULE`, `omp_set_schedule(sched, chunk_size)` and `omp_get_schedule(sched, chunk_size)`
- `schedule(runtime)` will use the schedule defined by the current ICV value (same as a loop with no schedule clause)