Computer Architecture

Discussion 13

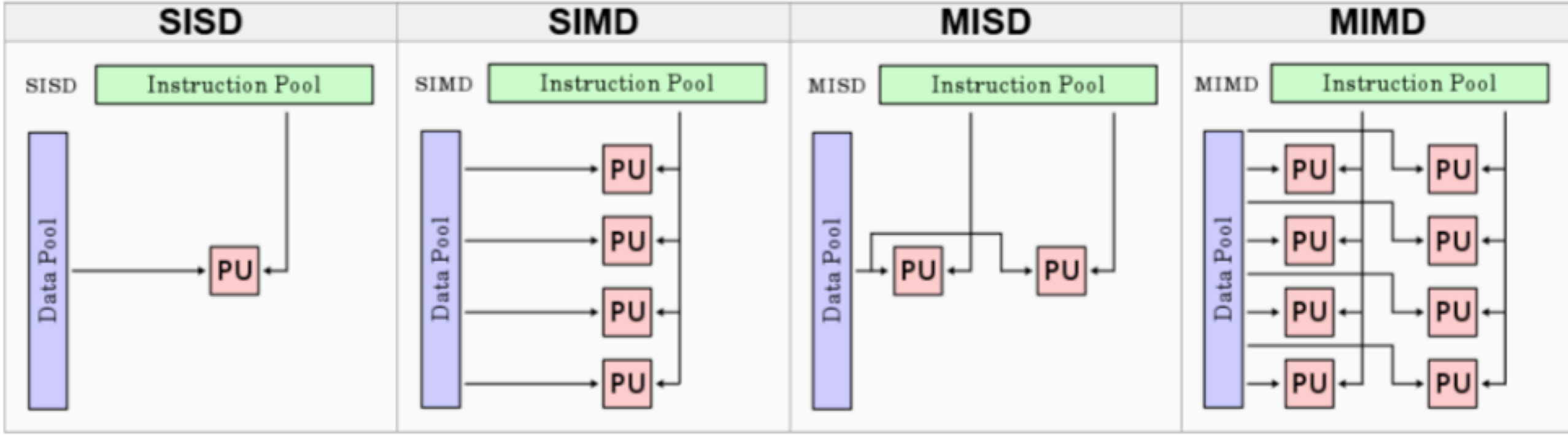
Parallelism

Parallelism

- Thread Level Parallelism (TLP):
- Executing different processes (threads) of the same program on different processors. Threads can communicate with each other.

- Data Level Parallelism (DLP):
- Operating on independent data simultaneously.
- …

Flynn's Taxonomy



SSE

```
#include <stdio.h>
#include <emmintrin.h> /* header file for the SSE intrinsics */

int main( int argc, char **argv ) {
    /* set   A  =   |1 3|,     B  =   |3 0|       C =   |0 0|
                    |2 4|             |0 2|             |0 0|  */
    double A[4] = {1,2,3,4}, B[4] = {3,0,0,2}, C[4] = {0,0,0,0};

    /*   We are computing C = C + A x B, which means:
         C[0] += A[0]*B[0] + A[2]*B[1]
         C[1] += A[1]*B[0] + A[3]*B[1]
         C[2] += A[0]*B[2] + A[2]*B[3]
         C[3] += A[1]*B[2] + A[3]*B[3] */

    /* load entire matrix C into SIMD variables */
    __m128d c1 = _mm_loadu_pd( C+0 ); /* c1 = (C[0],C[1]) */
    __m128d c2 = _mm_loadu_pd( C+2 ); /* c2 = (C[2],C[3]) */

    for( int i = 0; i < 2; i++ ) {
        __m128d a  = _mm_loadu_pd( A+i*2 ); /* load next column of A */
        __m128d b1 = _mm_load1_pd( B+0+i );
        __m128d b2 = _mm_load1_pd( B+2+i ); /* load next row of B */

        c1 = _mm_add_pd( c1, _mm_mul_pd( a, b1 ) ); /* multiply and add */
        c2 = _mm_add_pd( c2, _mm_mul_pd( a, b2 ) );
    }

    /* store the result back to the C array */
    _mm_storeu_pd( C+0, c1 ); /* (C[0],C[1]) = c1 */
    _mm_storeu_pd( C+2, c2 ); /* (C[2],C[3]) = c2 */

    /* output whatever we've got */
    printf( "|%g %g| * |%g %g| = |%g %g|\n", A[0], A[2], B[0], B[2], C[0], C[2] );
    printf( "|%g %g|   |%g %g|   |%g %g|\n", A[1], A[3], B[1], B[3], C[1], C[3] );

    return 0;
}
```

Amdahl's Law

Big Idea: Amdahl's Law

$$\text{Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}$$

Non-speed-up part

Speed-up part

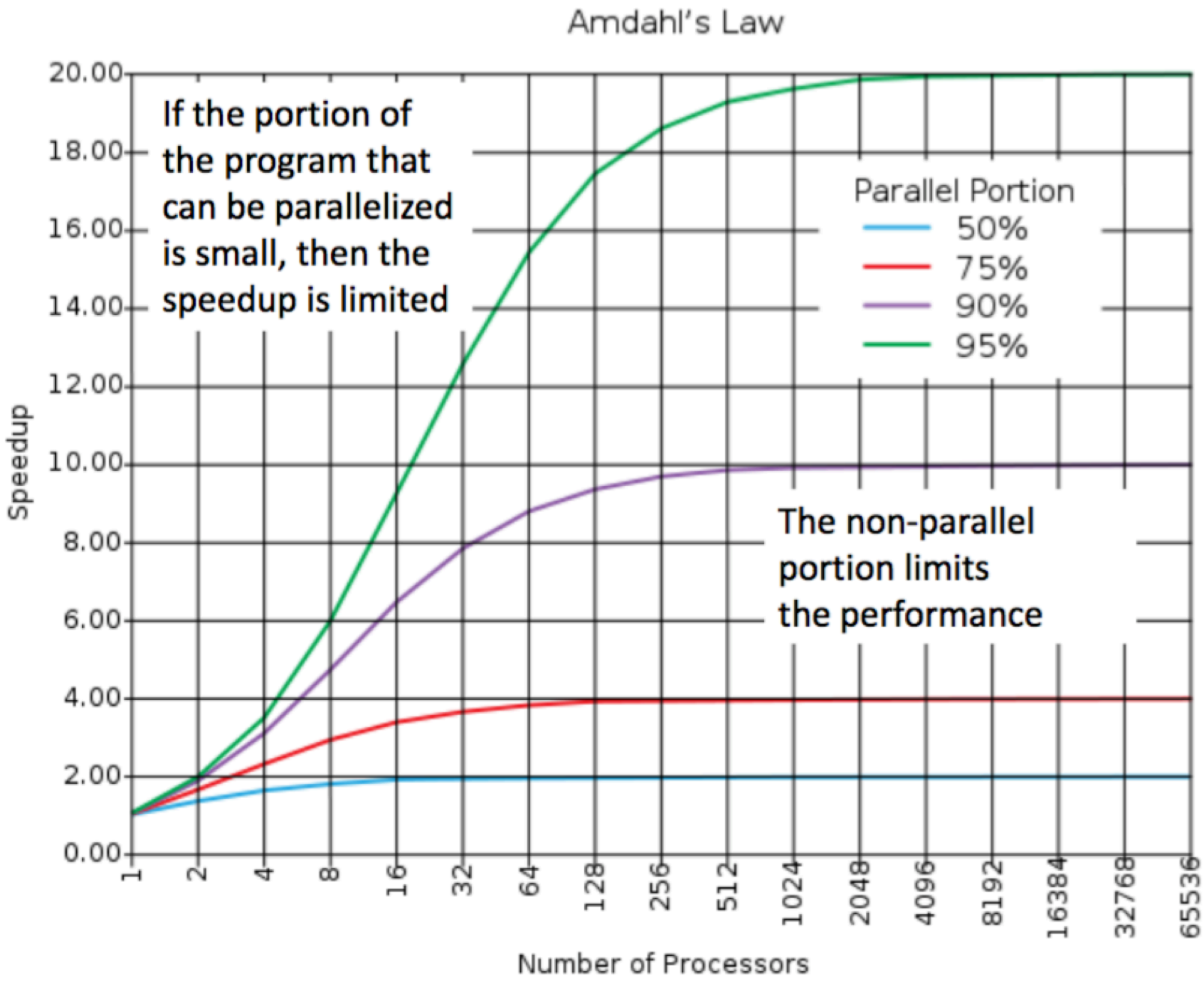
Example: the execution time of half of the program can be accelerated by a factor of 2.
What is the program speed-up overall?

$$\frac{1}{0.5 + \frac{0.5}{2}} = \frac{1}{0.5 + 0.25} = 1.33$$

Exercise

- Speedup
- 1. Consider an enhancement which runs 20 times faster but which is only usable 25% of the time
- 2. What if its usable only 15% of the time?

- In a given program, 95% of the execution time can be parallelized. How many processors are needed to achieve a speed-up of over 10?


Amdahl's Law
If the portion of the program that can be parallelized is small, then the speedup is limited
The non-parallel portion limits the performance
Parallel Portion
50%
75%
90%
95%
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

Strong and Weak Scaling

- *Strong scaling*: when speedup can be achieved on a parallel processor without increasing the size of the problem

- *Weak scaling*: when speedup is achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors

Strong scaling: when speedup can be achieved on a parallel processor without increasing the size of the problem

Weak scaling: when speedup is achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors

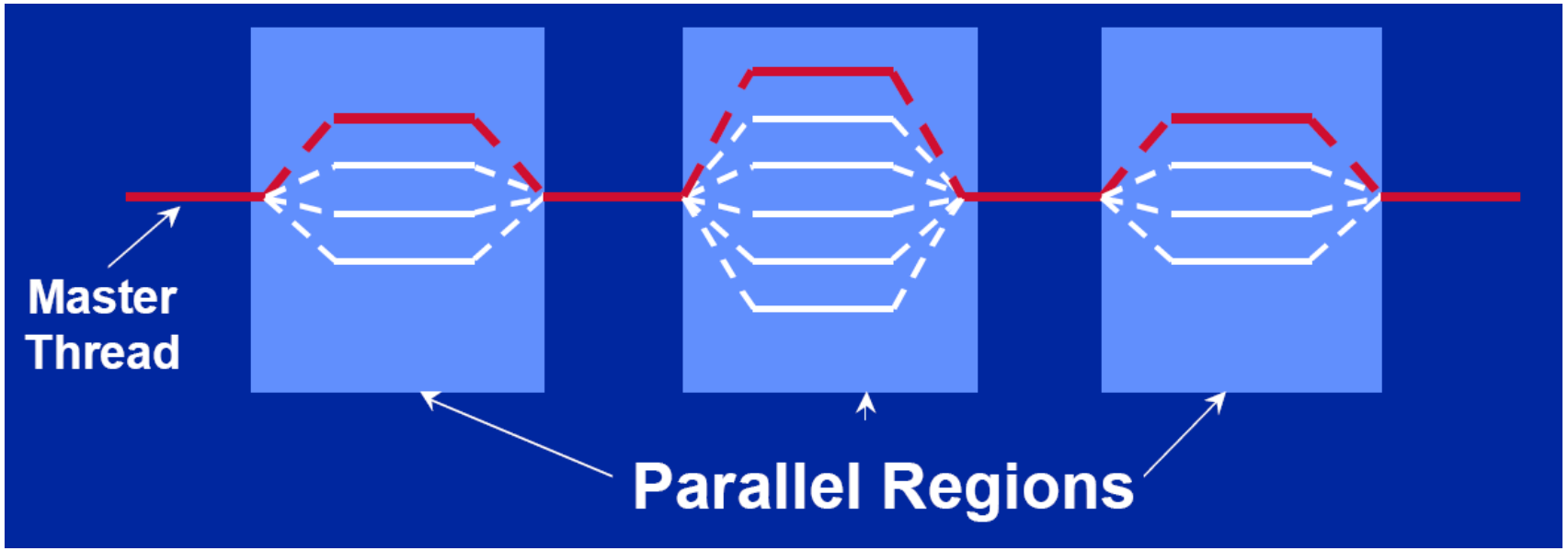
You have to solve a problem using Amazon EC2 servers. You know the server will finish the problem in an hour using 10 machines, but the deadline for your solution is just over 1 minute away. You attempt to solve the problem quickly by running an instance with 600 machines.

However, even though the cluster magically booted up instantaneously, you were late turning in your project and wasted a lot of money in the endeavor. This scenario indicates your solution lacked which kind of scaling? Circle one:

Strong Scaling **Weak Scaling**

Openmp

- **Thread Level Parallelism**



Parallel regions

- The `parallel` directive forks a team of threads, each of which executes the following region, enclosed in {...}.

Basic OpenMP construct for parallelization:

```
#pragma omp parallel
{
   /* code goes here */
}
```

This is annoying, but curly brace MUST go on separate line from #pragma

- Threads do a join at end of parallel region, and execution resumes with the single master thread.
- Number of threads can be set by
  - `num_threads` clause after the parallel directive.
  - `omp_set_num_threads()` library routine previously called.
  - Environment variable `OMP_NUM_THREADS.`
  - Recommendation is one thread per processor / core.
- Threads can do the work in the region in parallel.
  - Can do different things based on thread ID.
  - Can share work using `for`, `sections`, `task`, etc. directives.
- Parallel regions can be nested.

Parallel regions

- Example

```
#pragma omp parallel private(iam, np)
{
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d\n",
            iam, np);
}
```

  - All threads in parallel region run this code.
  - `iam` and `np` are private variables (i.e. instance of variable for each thread).
  - `omp_get_num_threads()` returns the number of threads n in the team used for the parallel region.
  - `omp_get_thread_num()` returns thread number (identity) in range 0 to n-1 with master thread 0.
  - Messages printed in arbitrary order.

Schedule clause

- Used for assigning iterations of parallel `for` to threads.
- `schedule(static[,chunk])`
  - Each thread gets a chunk of iterations of size "chunk" – by default chunks approximately equal.
  - Chunks assigned in round robin order.
- `schedule(dynamic[,chunk])`
  - Each time a thread finishes its iterations, grabs "chunks" more iterations, until all have been executed – default is 1.
  - Dynamic scheduling has some overhead, but can result in better load balancing if iterations not all equal sized.
- `schedule(guided[,chunk])`
  - Each thread dynamically grabs iterations where the size starts large and shrinks down to "chunk".
  - Dynamic load balancing with less overhead.
- `schedule(runtime)`
  - Schedule type and chunk size taken from the OMP_SCHEDULE environment variable.

Different ways to parallelize

```
// sequential

for (i=0; i<N; i++) {
     a[i] = a[i] + b[i];
}
```

```
// create parallel region
// then do worksharing

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

```
// create parallel region and do
//worksharing together

#pragma omp parallel for schedule(static)
     for (i = 0; i < N; i++) {
          a[i] = a[i] + b[i];
     }
```

```
// manual parallelization

#pragma omp parallel {
     int id, i, Nthreads, start, end;
     id = omp_get_thread_num();
     Nthreads = omp_get_num_threads();
     start = id * N / Nthreads;
     end = (id + 1) * N / Nthreads;
     for (i = start; i < end; i++) {
          a[i] = a[i] + b[i];
     }
}
```

```
// threads do redundant work

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

a)

```
// Set element i of arr to i
#pragma omp parallel
(int i = 0; i < n; i++)
        arr[i] = i;
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

b)

```
// Set arr to be an array of Fibonacci numbers.
arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i++)
        arr[i] = arr[i-1] + arr[i - 2];
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

c)

```
// Set all elements in arr to 0;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
        arr[i] = 0;
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

answer

a)

```
// Set element i of arr to i
#pragma omp parallel
(int i = 0; i < n; i++)
        arr[i] = i;
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

Slower than serial – there is no for directive, so every thread executes this loop in its entirety. n threads running n loops at the same time will actually execute in the same time as 1 thread running 1 loop. Despite the possibility of false sharing, the values should all be correct at the end of the loop. Furthermore, the existence of parallel overhead due to the extra number of threads could slow down the execution time.

answer

b)

```
// Set arr to be an array of Fibonacci numbers.
arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i++)
        arr[i] = arr[i-1] + arr[i - 2];
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

Always incorrect (if n>4) – Loop has data dependencies, so the calculation of all threads but the first one will depend on data from the previous thread. Because we said "assume no thread will complete before another thread starts executing," then this code will always be wrong from reading incorrect values.

answer

c)

```
// Set all elements in arr to 0;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
        arr[i] = 0;
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

Faster than serial – the for directive actually automatically makes loop variables (such as the index) private, so this will work properly. The for directive splits up the iterations of the loop into continuous chunks for each thread, so no data dependencies or false sharing.

Cache coherence

```
// Decrements element i of arr. n is a multiple of omp_get_num_threads()
#pragma omp parallel
{
        int threadCount = omp_get_num_threads();
        int myThread = omp_get_thread_num();
        for (int i = 0; i < n; i++) {
                if (i % threadCount == myThread)
                        arr[i] *= arr[i];
        }
}
```

What potential issue can arise from this code?

answer

```
// Decrements element i of arr. n is a multiple of omp_get_num_threads()
#pragma omp parallel
{
        int threadCount = omp_get_num_threads();
        int myThread = omp_get_thread_num();
        for (int i = 0; i < n; i++) {
                if (i % threadCount == myThread)
                        arr[i] *= arr[i];
        }
}
```

What potential issue can arise from this code?

False sharing arises because different threads can modify elements located in the same memory block simultaneously. This is a problem because some threads may have incorrect values in their cache block when they modify the value arr[i], invalidating the cache block. A fix to this will be discussed in lab.

3. Consider the following function:

```
void transferFunds(struct account *from, struct account *to, long cents) {
        from->cents -= cents;
        to->cents += cents;
}
```

a. What are some data races that could occur if this function is called simultaneously from two (or more) threads on the same accounts? (Hint: If the problem isn't obvious, translate the function into MIPS first)

b. How could you fix or avoid these races? Can you do this without hardware support?

answer

a. What are some data races that could occur if this function is called simultaneously from two (or more) threads on the same accounts? (Hint: If the problem isn't obvious, translate the function into MIPS first)

Each thread needs to read the "current" value, perform an add/sub, and store a value for from->cents and to->cents. Two threads could read the same "current" value and the later store essentially overwrites the other transaction at either line.

b. How could you fix or avoid these races? Can you do this without hardware support?

Wrap transferFunds in a critical section, or divide up the accounts array and for loop in a way that you can have separate threads work on different accounts