Homework 1 of CS7344

1.10

If a current computer can solve a problem of size 100,000 in 15 hours, how large of a problem can be solved in 15 hours by a computer that is 100 times faster, given different time complexities.

- $O(n)$ time complexity.
$$n = 100 * 100000 = 10000000$$
- $O(n \log_2 n)$ time complexity.
$$n \log_2 n = 100 * 100000 \log_2 100000$$
$$n = 7285987$$
- $O(n^2)$ time complexity.
$$n^2 = 100 * (100000)^2$$
$$n = 1000000$$
- $O(n^3)$ time complexity.
$$n^3 = 100 * (100000)^3$$
$$n = 464159$$

1.13

Consider the data dependence graph in Figure 1. Identify all sources of data parallelism. Identify all sources of functional parallelism.

- **Data Parallelism**
(1) The top three "A" nodes following the "I" node can be processed in parallel.
(2) The three "A" nodes following node "C" can be processed in parallel.
- **Functional Parallelism**
(1) Nodes "B" and "C" following the initial "A" nodes can be processed in parallel.
(2) "D" and the three "A" nodes following "C" can also be processed in parallel.

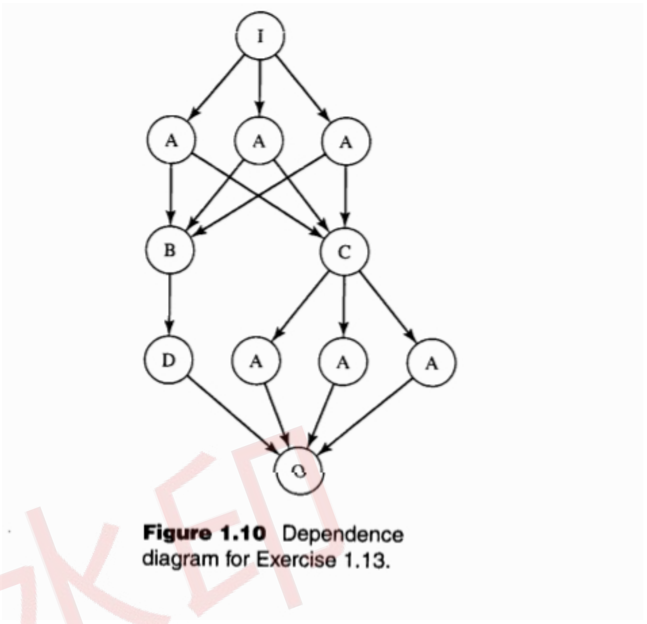

Figure 1: Dependence digram for Exercise 1.13

2.18

A directory-based protocol is popular way to implement cache coherence on a distributed multiprocessor.

- Why should the directory be distributed among the multiprocessor's local memories?
The directory is distributed among the local memories of a multiprocessor system to reduce the bottleneck that can occur if the directory is centralized. By distributing the directory, the system can take advantage of the locality of reference, as each processor will frequently access certain data more often than other data. This distribution helps to reduce the bottleneck (the accessing latency) that can come from multiple processors accessing a single centralized directory. It also helps in scaling the system to larger sizes, as the directory information grows with the number of processors and memory modules.
- Why are the contents of the directory not replicated?
The contents of the directory are not replicated to avoid the overhead and complexity of keeping all the replicas consistent with each other. In a distributed cache coherence protocol, it is crucial that the directory maintains a single view of the truth regarding the state of each block of data in the cache. Replicating this directory information would introduce numerous opportunities for inconsistencies to arise, especially when updates occur frequently. It would require a significant amount of additional coordination and communication to ensure that all replicas of the directory reflect the same state of the data, which could negate the performance benefits of having a directory in the first place. Additionally, not replicating the directory can save on memory usage, which is important in a large distributed system.

17.2

For each of the following code segments, use OpenMP to make the loop parallel, or explain why the code segment is not suitable for parallel execution.

a.

Conference'17, July 2017, Washington, DC, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
for (i = 0; i < (int) sqrt(x); i++) {
    a[i] = 2.3 * i;
    if (i < 10) b[i] = a[i];
}
```

This code can be paralleled as following

```
#pragma omp parallel for
for (i = 0; i < (int) sqrt(x); i++) {
    a[i] = 2.3 * i;
    if (i < 10) b[i] = a[i];
}
```

b.

```
flag = 0;
for (i = 0; (i<n)&(!flag); i++) {
    a[i] = 2.3 * i;
    if(a[i]<b[i]) flag = 1;
}
```

This code can not be paralleled because the flag variable will cause race conditions.

c.

```
for (i = 0; i < n; i++) {
    a[i] = foo(i);
}
```

This code can be paralleled as following

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

d.

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

This code can be paralleled as following

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

e.

```
for (i = 0; i < n; i++) {
    a[i] = foo(i);
    if (a[i]<b[i]) break;
}
```

This code can not be paralleled because break will cause contol problem.

f.

```
double dotp = 0.0;
for (i = 0; i < n; i++) {
    dotp += a[i] * b[i];
}
```

This code can be paralleled as following

```
double dotp = 0.0;
#pragma omp parallel for reduction(+:dotp)
for (i = 0; i < n; i++) {
    dotp += a[i] * b[i];
}
```

g.

```
for (int i = k; i < 2 * k; i++) {
    a[i] = a[i] + a[i - k];
}
```

This code can be paralleled as following

```
#pragma omp parallel for
for (int i = k; i < 2 * k; i++) {
    a[i] = a[i] + a[i - k];
}
```

h.

```
for (int i = k; i < n; i++) {
    a[i] = b*a[i - k];
}
```

This code can not be paralleled because the the a[i] depends on a[i-k], while a[i-k] may wait for being processed by other threads.

17.4

Section 17.7.3 discusses an interleaved scheduling of tasks to balance workloads among threads initializing an upper triangular matrix. Explain why increasing the chunk size from 1 could improve the cache hit rate.

Answer: Modern CPUs are designed to take advantage of spatial locality by loading chunks of contiguous memory into the cache. When a thread accesses a piece of data in memory, the CPU loads not only the requested data but also the surrounding data into the cache. Because the matrix is stored in memory with row-major layout, increasing chunk size will let one thread access more continuous data, which can improve the cache hit rate. What's more, increasing chunk can reduce cache evictions. With smaller chunk sizes, threads might work on smaller sets of data scattered across the matrix. This can lead to more frequent loading of different data blocks into the cache, potentially evicting data that will soon be needed by another thread.

17.6

Give an original example of a parallel for loop that would probably execute in less time if it were dynamically scheduled rather than statically scheduled.

Answer:

```
void process(int i){
    sleep(i);
}
for (int i = 0; i < 100; i++) {
    process(i);
}
```

In this example, each `i` will cause the thread to sleep `i` seconds. If using the statically scheduling, then some threads which processing large `i` will sleep very long which the others processing small `i` will sleep very short. This is a very unbalanced strategy. However, if we use dynamically scheduling, the time of the sleeping will be much smaller.