**Homework 3**

Yuanrui Zhang (yz545)

**Problem 1. Programming Models**

1. “for i” loop only

* Read-only: N, data\_array
* Read/write non-conflicting: data\_gridX, data\_gridY
* Read/write conflicting: i, j, sum, product, measurement

1. “for j” loop only

* Read-only: N, data\_array, sum, i
* Read/write non-conflicting: None
* Read/write conflicting: j, data\_gridX, data\_gridY, product, measurement.

**Problem 2. Code Analysis for Parallel Task Identification**

Loop-carried dependency:

1. S1[i, j] 🡪 T S1[i, j - 2]
2. S2[i, j] 🡪 T S2[i + 1, j]
3. S2[i, j] 🡪 A S2[i + 1, j]

LDG:

A close up of text on a whiteboard

Description automatically generated

1. No, it isn’t. Because of the loop carried dependency (2) and (3) list above, there can’t be DOALL parallelism.
2. No, the same reason as (a) – there is a loop carried dependency (1) between for j loop iterations.
3. The update of each node along a diagonal is an independent parallel task because there is chain of dependency in this pattern, e.g., [1, 3]🡪[1, 1]🡪[2,1]🡪[3, 1], but along anti-diagonal is not.
4. Yes. We can also parallelize for j loop with odd indices and with even indices. Because according to the LDG, the loop-carried dependency only exists among odd columns or among even columns. That’s said, no such dependency exists across an even and an odd column. Thus, we can effectively divide it into 2 parallel tasks.

**Problem 3. Code Profiling & Performance Counters**

1. Performance profiling

|  |  |  |  |
| --- | --- | --- | --- |
| Function index | Function name | The number of calls | percentage of execution time |
| 1 | miniFE::matvec\_std<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int> >::operator()(miniFE::CSRMatrix<double, int, int>&, miniFE::Vector<double, int, int>&, miniFE::Vector<double, int, int>&) | 201 | 30.09% |
| 2 | frame\_dummy | 1597918831 | 11.08% |
| 3 | std::\_Rb\_tree<int, int, std::\_Identity<int>, std::less<int>, std::allocator<int> >::\_S\_key(std::\_Rb\_tree\_node<int> const\*) | 57598102 | 6.52% |
| 4 | std::\_Rb\_tree<int, int, std::\_Identity<int>, std::less<int>, std::allocator<int> >::\_S\_value(std::\_Rb\_tree\_node<int> const\*) | 435792686 | 4.48% |
| 5 | int\* std::lower\_bound<int\*, unsigned long>(int\*, int\*, unsigned long const&) | 32768000 | 3.69% |
| 6 | \_\_gnu\_cxx::\_\_aligned\_membuf<int>::\_M\_ptr() | 435883250 | 3.22% |
| 7 | std::\_Rb\_tree\_node<int>::\_M\_valptr() | 435883250 | 2.99% |
| 8 | miniFE::decide\_how\_to\_shrink(Box const&, Box const&) | 719 | 2.99% |

1. Amdahl’s Law

Speedup = 1/(1 - 0.3009 + 0.3009 / 5) = 1.32

1. Performance Counters
2. When running *perf stat ./miniFE.x -nx 40 -ny 80 -nz 160* by default:

Text

Description automatically generated

1. Measure Instructions; CPU cycles (and also show IPC, instructions per cycle); Branch instructions; Branches misses (mispredictions)

Text

Description automatically generated

1. Measure Cache references; L1 data cache load misses; L1 instruction cache load misses; LLC (last level cache) loads; LLC (last level cache) load misses; Data TLB load misses

Text

Description automatically generated

**Problem 4. Performance Counters**

1. Re-run my program across different loop nest orderings on the machine where I am using ‘perf’

|  |  |
| --- | --- |
| Loop nest orderings | Time |
| I-J-K | 14.641327 s |
| I-K-J | 0.716340 s |
| J-K-I | 27.432645 s |

1. Use ‘perf’ to see performance counters
2. I-J-K

Text

Description automatically generated

1. I-K-J

Graphical user interface, text

Description automatically generated

1. J-K-I

Graphical user interface, text

Description automatically generated

From the three screenshots above, the first column of performance counter stats represents the raw counter numbers. We are easy to tell from those numbers that I-K-J has the least number of all kinds of cache misses, while J-K-I has the most and I-J-K stays in between. In summary, performance counter results explain why these three patterns have different performance.