HW2 Writeup

Meihong Ge mg476

1. **PB1**

   If we are parallelizing the "for i" loop:

   Read only:

   N, data_array

   R/W non-conflicting:

   Data_gridX, Data_gridY

   R/W conflicting:

   Product, j, i, sum, measurement

   If we are parallelizing the "for j" loop:

   Read only:

   N, i, sum

   R/W non-conflicting:

   Data_gridX, Data_gridY

   R/W conflicting:

   Product, j, measurement

2. **PB2**

   The loop has the following dependencies:

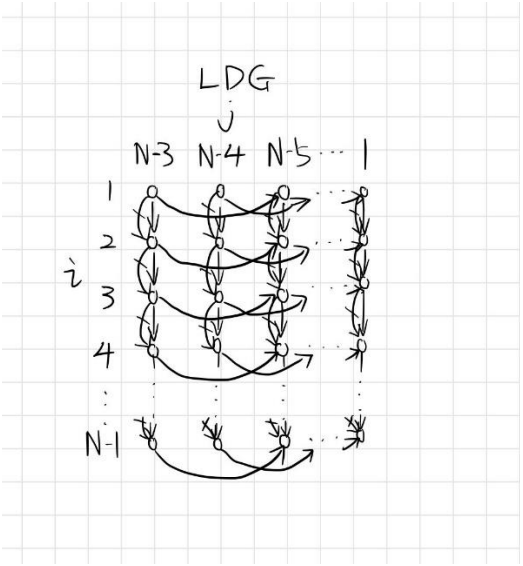
   S2[i,j] => T S2[i+1, j]

   S2[i-1, j] => AS2[i,j]

   S1[i,j] => AS2[i,j]

   S1[i,j] => TS1[i, j-2]

   LDG:



a) No, because there is two loop-carried dependency within each task:
S2[i,j] => T S2[i+1, j]
S2[i-1, j] => AS2[i,j]

b) No, because there is one loop-carried dependency within each task:
S1[i,j] => TS1[i, j-2]

c) If the diagonal refers to my LDG:
Then if update is done along the diagonal, the task is an independent parallel task.
If update is done along the anti-diagonal, the task is also an independent parallel task.

d) When i is fixed, j can be divided into two parallel tasks: all the iterations with odd j and all the iteration with even j.

3. PB3

a) Flat profile result:

Ranking	%time	Calls	Name
1	15.73	1639312485	frame_dummy
2	8.53	51	MatvecOp<miniFE::CSRMatrix<double, int, int, SerialComputeNode> >::operator()(int)
3	6.48	32768000	void std::advance<int*, long>(int*&, long)
4	6.37	364141551	NoOpMemoryModel::~NoOpMemoryModel()
5	5.40	28316792	std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_M_lower_bound(std::_Rb_tree_node<int> const*, std::_Rb_tree_node_base const*, int const&) const
6	5.24	512000	void miniFE::Hex8::diffusionMatrix_symm<double>(double const*, double const*, double*)
7	4.63	435792686	std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_S_key(std::_Rb_tree_node<int> const*)
8	4.52	435837968	__gnu_cxx::__aligned_membuf<int>::_M_addr() const

b) Amdahl's Law:

In my case, the most time-consuming function is "frame_dummy" which takes up a total of 15.73% running time. If this portion of code were to receive a speed-up of 5x, the resulting overall performance could be expressed by the equation:

$$speedup = \frac{\mathrm{T_s}}{\mathrm{T_p}} = \frac{T_s}{T_s \times (\mathrm{S} + \frac{\mathrm{P}}{\mathrm{N}})} = \frac{1}{\mathrm{S} + \frac{P}{N}} = \frac{1}{(1-\mathrm{P}) + \frac{\mathrm{P}}{\mathrm{N}}} = \frac{1}{84.27\% + 15.73\%/5} = 1.144$$

c) Performance Counter event count:

Event	Counts
Instructions	377,076,518,622
CPU Cycles	243,557,643,343
Branch Instructions	70,662,886,332
Branch Misses (mispredictions)	2,164,826,385

Cache References	57,666,269
L1 Data Cache Load Misses	218,681,416
L1 Instruction Cache Load Misses	15,480,543
LLC Loads	89,860,101
LLC Loads Misses	49,702,961
Data TLB Load Misses	4,227,796

4. PB4

Performance of Different Ordering:

Ordering	**Runtime (in second)**
i-j-k	21.803
j-k-i	28.546
i-k-j	16.288

Miss-rate of Different Level of Cache:

Odering	**L1 Data Cache Load Misses**	**L1 Data Cache Load**	**L1 Data Cache Load Miss Rate**	**LLC Cache Load Misses**	**LLC Cache Load**	**LLC Cache Load Miss Rate**
i-j-k	1,892,707,549	34,404,699,348	5.5%	308,404	178,342,086	0.17%
j-k-i	3,510,043,684	34,404,164,391	10.20%	1,205,186	675,627,256	0.18%
i-k-j	140,518,948	34,396,169,710	0.41%	93,498	7,622,501	1.23%

The miss rate as well as the hardware count all show a strong correlation with the runtime. That is, j-k-i being the slowest, the one with most L1 Cache Misses. I-j-k follows it with approximately one-half the misses. I-k-j is significantly better in terms of L1 performance, with much lower cache misses. The LLC cache miss rate is a bit high, but if we consider the miss counts as well as the total load action count, the aggregate penalty is still much lower.