ECE 565 - HW3

Yi Mi

October 7, 2020

1 Question 1

For i loop:
- Read-only: data_array, N
- R/W non-conflicting: data_gridY, data_gridX
- R/W conflicting: i, j, product, sum, measurement

For j loop:
- Read-only: i, data_array, N, sum
- R/W non-conflicting: data_gridY, data_gridX
- R/W conflicting: j, measurement, product

2 Question 2

Dependence:
- Loop independent:

S2[i,j] -> A S2[i,j]
- Loop carried:

S1[i,j] -> T S1[i,j-2]
S2[i,j] -> T S2[i+1,j]
S2[i,j] -> A S2[i+1,j]

Figure 1: ITG

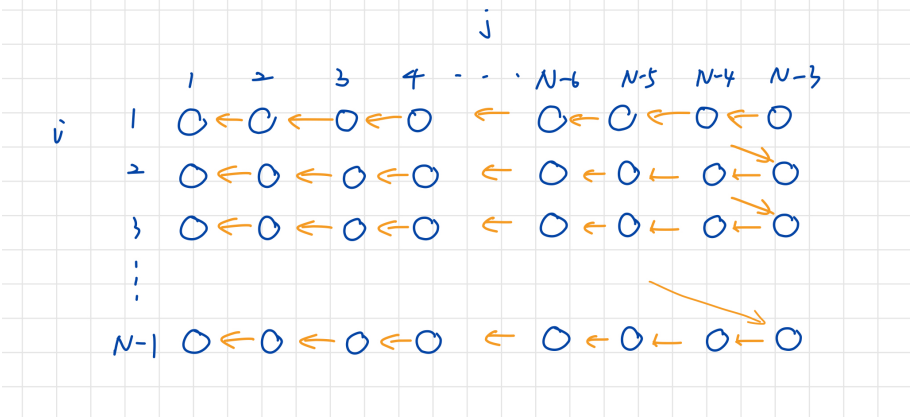
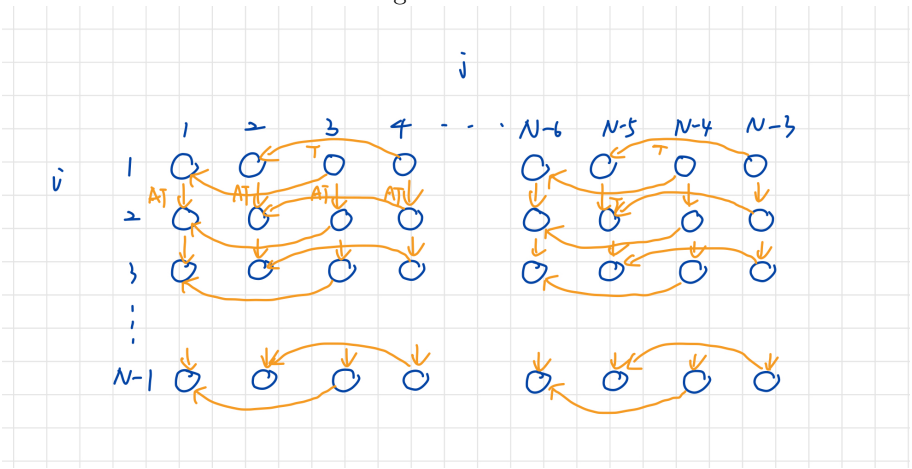


Figure 2: LDG

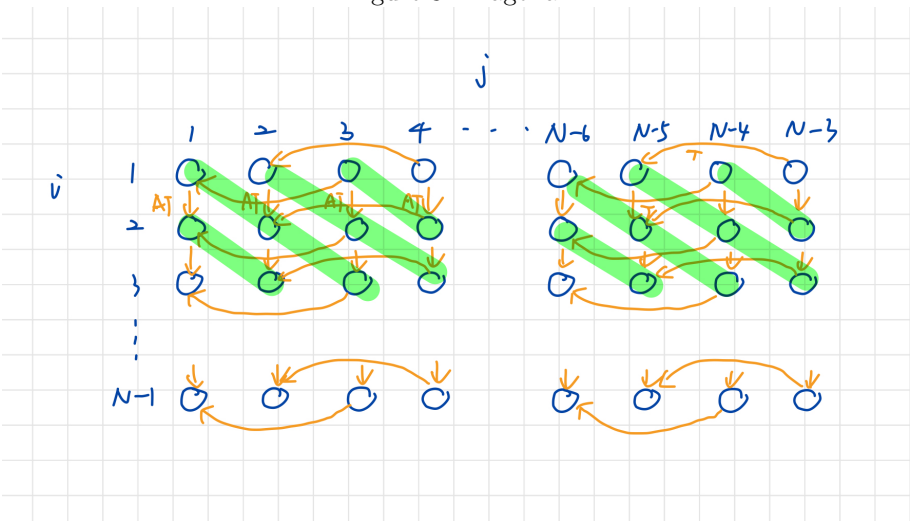


(a) Based on the dependence, ITG and LDG, we can see that each "for i" loop iteration has true and anti dependence so that there is no independent parallel task.

(b) Based on the dependence, ITG and LDG, we can see that the "for j" loop iterations can be divided into two parallel tasks – one with odd iterations and another with even iterations.

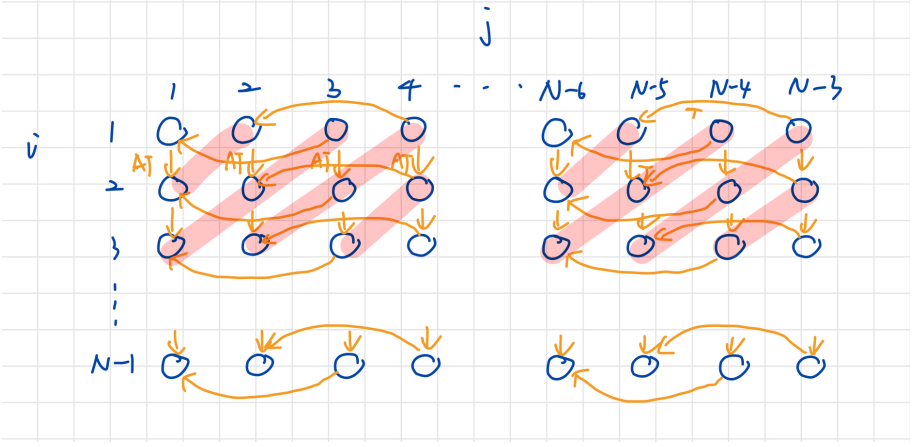
(c) Based on ITG we can see that the traverse order for j is from N-3 to 1. I will just define diagonal as the followed figure though the execution is from the end of each line, which might be a litter counter-intuitive. Looked in to the diagonal figure, we can see that in each diagonal, the nodes are independent of each other so that each node is an independent parallel task.

Figure 3: Diagonal



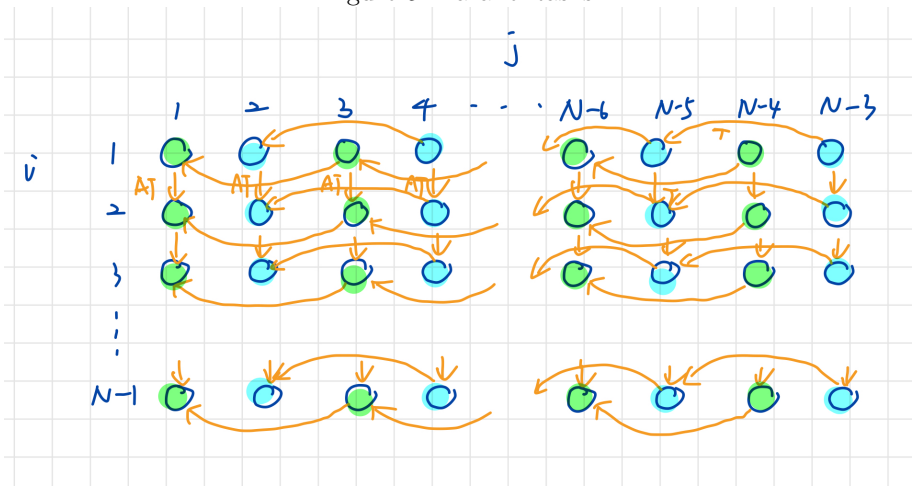
I define the anti-diagonal as shown in the followed figure. Each node along an anti-diagonal is not an independent parallel task.

Figure 4: Anti-diagonal



(d) The task can be divided into two parallel tasks which are identified using different colors (assume N is odd).

Figure 5: Parallel tasks



3 Question 3

Server machine: kraken.egr.duke.edu

(a)Performance profiling

1. With inline

Add flags:

```
#-----------------------------------------------------------------------

MINIFE_TYPES =  \
        -DMINIFE_SCALAR=double   \
        -DMINIFE_LOCAL_ORDINAL=int      \
        -DMINIFE_GLOBAL_ORDINAL=int
MINIFE_MATRIX_TYPE = -DMINIFE_CSR_MATRIX
#-----------------------------------------------------------------------

CFLAGS = -O3 -pg -fno-inline
CXXFLAGS = -O3 -pg -fno-inline
CPPFLAGS = -I. -I../utils -I../fem \$(MINIFE_TYPES) \$(MINIFE_MATRIX_TYPE)
LDFLAGS =
LIBS=

CXX=g++
CC=gcc

include make_targets
```

Compile:

```
make -f makefile.gnu.serial
```

Run:

```
./miniFE.x -nx 40 -ny 80 -nz 160
gprof miniFE.x gmon.out > analysis.txt
```

Figure 6: Performance profiling - with inline

% time	calls	name
30.82	201	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>&)
16.2	1622833938	frame_dummy
7.55	57598102	std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_S_key(std::_Rb_tree_node<int> const*)
4.77	435928532	std::_Rb_tree_node<int>::_M_valptr()
4.29	435792686	std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>,
3.57	512000	void miniFE::Hex8::diffusionMatrix_symm<double>(double const*, double const*, double*)
3.02	510695430	std::pair<int const, int>* std::__addressof<std::pair<int const, int> >(std::pair<int const, int>&)
3.02	32768000	int* std::lower_bound<int*, unsigned long>(int*, int*, unsigned long const&)

2. Without inline

Edit flags:

```
CFLAGS = -O3 -pg
CXXFLAGS = -O3 -pg
```

Figure 7: Performance profiling - without inline

% time	calls	name
63.95	1	void miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec_std<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int> > > (miniFE::CSRMatrix<double, int, int>&, miniFE::Vector<double, int, int> const&, miniFE::Vector<double, int, int>&, miniFE::matvec_std<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int> >, miniFE::CSRMatrix<double, int, int>::LocalOrdinalType, miniFE::TypeTraits<miniFE::CSRMatrix<double, int, int>::ScalarType>::magnitude_type&, miniFE::CSRMatrix<double, int, int>::LocalOrdinalType&, miniFE::TypeTraits<miniFE::CSRMatrix<double, int, int>::ScalarType>::magnitude_type&, double*)
7.61	2	void miniFE::impose_dirichlet<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int> >(miniFE::CSRMatrix<double, int, int>::ScalarType, miniFE::CSRMatrix<double, int, int>&, miniFE::Vector<double, int, int>&, int, int, int, std::set<miniFE::CSRMatrix<double, int, int>::GlobalOrdinalType, std::less<miniFE::CSRMatrix<double, int, int>::GlobalOrdinalType>, std::allocator<miniFE::CSRMatrix<double, int, int>::GlobalOrdinalType> > const&)
6.62	512000	void miniFE::Hex8::diffusionMatrix_symm<double>(double const*, double const*, double*)
5.49	512000	void miniFE::sum_in_symm_elem_matrix<miniFE::CSRMatrix<double, int, int> > (unsigned long, miniFE::CSRMatrix<double, int, int>::GlobalOrdinalType const*, miniFE::CSRMatrix<double, int, int>::ScalarType const*, miniFE::CSRMatrix<double, int, int>&)
2.68	4096000	void miniFE::Hex8::gradients_and_invJ_and_detJ<double>(double const*, double const*, double*, double&)
2.54	4096000	void miniFE::Hex8::gradients_and_detJ<double>(double const*, double const*, double&)
2.11	1	void miniFE::init_matrix<miniFE::CSRMatrix<double, int, int> >(miniFE::CSRMatrix<double, int, int>&, std::vector<miniFE::CSRMatrix<double, int, int>::GlobalOrdinalType, std::allocator<miniFE::CSRMatrix<double, int, int>::GlobalOrdinalType> > const&, std::vector<miniFE::CSRMatrix<double, int, int>::LocalOrdinalType, std::allocator<miniFE::CSRMatrix<double, int, int>::LocalOrdinalType> > const&, std::vector<int, std::allocator<int> > const&, int, int, int, miniFE::CSRMatrix<double, int, int>::GlobalOrdinalType, miniFE::simple_mesh_description<miniFE::CSRMatrix<double, int, int>::GlobalOrdinalType> const&)
2.11	512000	void miniFE::Hex8::sourceVector<double>(double const*, double const*, double*)

(b)Amdahl's Law

1. With inline

$1/(30.83\%/5 + 69.18\%) = 1.327$

2. Without inline

$1/(63.95\%/5 + 36.05\%) = 2.048$

(c)Performance Counters

Use perf list to find the event names, then run followed command line to collect events.

```
perf stat -e instructions -e cpu-cycles -e branch-instructions
-e branch-misses -e cache-references -e L1-dcache-load-misses
-e L1-icache-load-misses -e LLC-loads -e LLC-load-misses
-e dTLB-load-misses ./miniFE.x -nx 40 -ny 80 -nz 160
```

1. With inline

Figure 8: Performance counters - with inline

Event	Count
instructions	379,502,750,345
cpu-cycles	244,275,149,016
branch-instructions	76,918,919,523
branch-misses	627,098,392
cache-references	351,847,274
L1-dcache-load-misses	758,843,172
L1-icache-load-misses	8,882,277
LLC-loads	659,996,736
LLC-load-misses	356,751,424
dTLB-load-misses	11,534,101

IPC (instructions per cycle): 1.55

2. Without inline

Figure 9: Performance counters - without inline

Event	Count
instructions	39,676,380,215
cpu-cycles	18,001,586,092
branch-instructions	6,103,891,719
branch-misses	11,979,359
cache-references	338,355,259
L1-dcache-load-misses	739,740,765
L1-icache-load-misses	891,329
LLC-loads	643,229,367
LLC-load-misses	347,122,102
dTLB-load-misses	12,197,059

IPC (instructions per cycle): 2.2

4 Question 4

I first re-run my program three time separately across different loop nest orderings on my machine.
Compile:

```
gcc -O3 -o matrix_multiplication matrix_multiplication.c -lrt
```

Run:

```
./matrix_multiplication 1

./matrix_multiplication 2

./matrix_multiplication 3
```

Figure 10: Re-run matrix multiplication

Order	1st run (s)	2rd run (s)	3rd run (s)	Average time (s)
i-j-k	5.540775	5.531758	5.640424	5.570985667
j-k-i	23.843624	23.786044	23.812337	23.81400167
i-k-j	0.862951	0.887483	0.987883	0.912772333

I then run my program across different loop nest orderings and use 'perf' to see the miss rates.
Run:

```
perf stat -e L1-dcache-load-misses -e L1-dcache-loads
-e LLC-load-misses -e LLC-loads ./matrix_multiplication 1

perf stat -e L1-dcache-load-misses -e L1-dcache-loads
-e LLC-load-misses -e LLC-loads ./matrix_multiplication 2

perf stat -e L1-dcache-load-misses -e L1-dcache-loads
-e LLC-load-misses -e LLC-loads ./matrix_multiplication 3
```

Figure 11: Run matrix multiplication with perf

Order	L1 data cache load misses	L1 data cache load	L1 data cache miss rate	Last level cache load misses	Last level cache load	Last level cache miss rate
i-j-k	543,710,483	1,114,855,056	48.77%	123,510	541,744,010	0.02%
j-k-i	2,187,206,674	2,197,077,375	99.55%	735,075	2,166,234,861	0.03%
i-k-j	135,976,329	1,113,085,741	12.22%	74,151	7,069,101	1.05%

The L1 data cache miss rate of i-j-k is the highest, which explains why it takes much more time than the other two order nesting, since it has to go to the lower level cache to load the data. For i-k-j, it has the lowest L1 data cache miss rate and takes the least run time. Because it has good spatial locality and

good temporal locality. Looked at LLC load, we can also find that i-k-j has the smallest number of hit and the reason is the same as above - it doesn't have to go to the lower level caches to fetch the data frequently. Above all, it explain the performance results.