**Student:** Hong Quan Doan

**Student ID:** 986956

**Day 05, 06 – Feb 28, Mar 01**

**Matrix Multiplication**

===Matrix Multiplication test===

Generated: matrix1[1024][1024] and matrix2[1024][1024]

Sequential time for typical: 14.5395

Parallel time for typical: 4.5743

Speedup for typical: 3.17853

Processor Utilization for typical: 0.794632

Sequential time for interchange: 3.00531

Parallel time for interchange: 1.12003

Speedup for interchange: 2.68323

Processor Utilization for interchange: 0.670808

Speedup overall: 12.9813

Processor Utilization overall: 3.24533

The result tells us two important things:

* Cache is critical when we implement a solution
* Parallelizing with good solution will significantly improve the performance.

Source code of parallelizing with interchange loop solution

**void** parmatrixmultiplicationinterchangeloop()

{

**double** st = omp\_get\_wtime();

omp\_set\_num\_threads(omp\_get\_num\_procs());

#pragma omp parallel for

**for** (**int** i = 0; i < row1; i++)

{

**for** (**int** k = 0; k < col1; k++)

{

**for** (**int** j = 0; j < col2; j++)

{

matrix3[i][j] += matrix1[i][k] \* matrix2[k][j];

}

}

}

cout << endl;

**double** partime = omp\_get\_wtime() - st;

cout << "Parallel time for interchange: " << partime << endl;

cout << "Speedup for interchange: " << matrixseqtimeinterchange / partime << endl;

cout << "Processor Utilization for interchange: " << (matrixseqtimeinterchange / partime) / omp\_get\_num\_procs() << endl;

cout << endl;

cout << "Speedup overall: " << matrixseqtimetypical / partime << endl;

cout << "Processor Utilization overall: " << (matrixseqtimetypical / partime) / omp\_get\_num\_procs() << endl;

}

**Numerical Integration**

===Numerical Integration test===

Sequential Numerical Integration

Result: 3.14159

Sequential time: 1.54353

Parallel Numerical Integration Using Section with 4 blocks

Number of processors: 4

Result: 3.14159

Parallel using 4 blocks time: 3.04341

Speedup: 507170

Processor Utilization: 126793

Parallel Numerical Integration Using Section with Jumping pointer

Number of processors: 4

Result: 3.14159

Parallel using Jumping pointer time: 3.035

Speedup: 508576

Processor Utilization: 127144

Parallel Numerical Integration Using Parallel with Critical

Number of processors: 4

Result: 3.14159

Parallel using Critical time: 20.1466

Speedup: 76614.8

Processor Utilization: 19153.7

Parallel Numerical Integration Using Parallel with Summary Array

Number of processors: 4

Result: 3.14159

Parallel using Summary Array time: 0.410747

Speedup: 3.75786e+06

Processor Utilization: 939464

The result tells us few things:

* Not always parallel is better.
* A good parallelizing depends very much on how we deal with data, cache, algorithm.

Source code of the fastest Parallel solution.

**void** parnumericalintegrationparallelsummaryarray()

{

**double** sum = 0.0;

**double** st = omp\_get\_wtime();

**int** tnum = omp\_get\_num\_procs();

omp\_set\_num\_threads(tnum);

**int** range = samplepoints / tnum;

**int** i;

**int** start, end;

**int** tid;

**double** sumarr[tnum];

cout << "Number of processors: " << tnum << endl;

#pragma omp parallel private (i, tid, start, end)

{

tid = omp\_get\_thread\_num();

start = tid \* range + 1;

end = tid \* range + range;

sumarr[tid] = 0.0;

**for** (i = start; i <= end; i++)

{

sumarr[tid] = sumarr[tid] + f(i);

}

}

*// Add up the individual sum to the last sum*

**for** (**int** j = 0; j < tnum; j++)

{

sum = sum + sumarr[j];

}

cout << "Result: " << w \* (sum + (sqrt(a) + sqrt(b)) / 2) << endl;

**double** partime = omp\_get\_wtime() - st;

cout << "Parallel using Summary Array time: " << partime << endl;

cout << "Speedup: " << seqtime / partime << endl;

cout << "Processor Utilization: " << (seqtime / partime) / omp\_get\_num\_procs() << endl;

}