MP 1

Addition

- Since each iteration is independent of other, a simple omp parallel do with critical region or reduction is sufficient. Critical regions are used to avoid the race condition in writing to arraySum. When no. of threads is doubled the speedup is doubled as well, because each thread performs equal no. of iterations. Critical is slightly faster than reduction.
- Times for each implementation with speedups in brackets are given below. Speedups are measured with respect to time for original serial code.

Threads	Critical	Reduction
1	3.83(serial)	3.83(serial)
2	1.90(2.02)	2.00(1.92)
4	0.968(3.96)	1.02(3.75)
6	0.681(5.62)	0.756(5.06)
12	0.349(11)	0.423(9.05)

Multiplication

Three parallelizations were tested.

- Parallelizing i loop with omp parallel do ,i.e. each thread computes a set of rows of C. In fortran arrays are stored column wise, so there this might result in Cache thrashing in accessing $C(i, j$.
- Parallelizing j loop with omp parallel do. This is slightly better because it improves cache hits in write allocates for $C(i, j)$. But in terms of cost for inner loops for computing multiplication of rows of A and columns of B, it is equivalent to above choice. Speedup is linear.
- First serial code is optimized by interchanging j and i loop (which improves cache reuse for C) and then unrolling and jamming i loop(by factor of 50) (which imporoves cache reuse for A). Then j loop is parallelized using omp parallel do. Speedup when compared to original serial code is superlinear, but when compared to optimized serial code , it is linear. This because all the threads perform even amount of work.
- Speedups given in brackets are with respect to original serial code.

Threads	Parallel i	Parallel j	Parallel j and unroll i
1	1.7204(serial)	1.7152(1.00)	1.31936(1.304)
2	0.8765(1.96)	0.8718(1.97)	0.6615(2.60)
4	0.5070(3.39)	0.4295(4.00)	0.3319(5.18)
6	0.3701(4.65)	0.3002(5.70)	0.2332(7.38)
12	0.2448(7.03)	0.1528(11.30)	0.1223(14.07)

Transpose

Three implementations were tested.

- outermost i loop is parallelized using omp parallel do. Speedup is sublinear as this might result in cache thrashing since fortran stores array in column major.
- loop k is interchanged with i and it is parallelized using omp parallel do. Just interchanging look k resulted in speedup by factor of 2, as it increases Cache reuse. It's parallelization further resulted in speedup, which is slightly sublinear. I think it's because of load imbalance between threads, as no. of iterations for loop j decrease with increasing k.
- serial code was optimized first by interchanging k with i and unrolling and jamming i (with a factor of 10). This resulted in speedup by factor 8 with respect to original code. Then k is parallelized using omp parallel do. With parallelization the speedup is sublinear(with respect to optimized code) and there is no speedup after 4 threads. I think this is again due to load imbalancing.
- Guided , static and dynamic scheduling were tried to offset load imbalancing, but there wasn't significant difference in times. Results below are shown for Static,50. Speedups in brackets are wih respect to original serial code.

Threads	Parallel i	Parallel k	Parallel k with unroll i
1	9.8051(serial)	5.4201(1.81)	1.2406(7.904)
2	7.7545(1.264)	2.9613(3.31)	0.7778(12.606)
4	6.4618(1.517)	1.6764(5.84)	0.4957(19.78)
6	6.3715(1.539)	1.3432(7.30)	0.4515(21.72)
12	6.3819(1.536)	0.9368(10.47)	0.4178(23.468)

4 Credit: Transpose

- omp parallel is used. k is the outermost loop and i is the innermost loop. for each k , loop j is divided into chunks of size $(n-k)/numthreads$ and one chunk is assigned to each thread. Therefore each nearly every column starting from below k^{th} diagnoal is tranposed in parallel using every thread. As k is increased chunks reduce in size and for $k = n-1$ only thread 0 will work. This division is done to avoid cache thrashing, by letting each thread access elements in both nearby columns and rows . Speedup with respect to original code is superlinear. But it saturates after 4 threads. This might due to the size of array tested.

- Nested parallization by using omp do on i (along with above) was also tested. But it didn't result in improvement in performance for small size along i axis. So the results are not shown here.

- Results are shown for array of size $4 \times 5000 \times 5000$. Speedup is measured with respect to original serial code (time taken = 0.32574)

Threads	OMP Parallel
1	0.15193(2.14)
2	0.13743(2.37)
4	7.05955e-2(4.61)
6	5.37491e-2(6.06)
12	4.93603e-2(6.6)