Assignment 2

Ayush Kumar (170195)

1 October 2020

I pledge on my honor that I have not given or received any unauthorized assistance.

Solutions

1. There appear to be 3 data dependences in the array references present in the only statement of the innermost loop. The pairs of array references and their correponding dependence is analysed below using the **Delta Test**.

   - $A[i, i-j]$ and $A[i, i-j-1]$
     Let the two loops in which the array references match be (I, J) and $(I + \Delta I, J + \Delta J)$. Then we have,

     $$I = I + \Delta I \tag{1}$$
     $$I - J = I + \Delta I - J - \Delta J - 1 \tag{2}$$

     Solving the system of two equations simultaneously, we get $\Delta I = 0$ and $\Delta J = -1$. This implies a **loop-carried anti-dependence in loop j** having **distance vector** of $(0, 1)$ and a **direction vector** of $(0, +)$.
   - $A[i, i-j]$ and $A[i-1, i-j]$
     Proceeding similarly as in the previous case, we get the following two equations,

     $$I = I + \Delta I - 1 \tag{1}$$
     $$I - J = I + \Delta I - J - \Delta J \tag{2}$$

     Solving these equations simultaneously, we get $\Delta I = 1$ and $\Delta J = 1$. This implies a **loop-carried flow-dependence in loop i** having a **distance vector** of $(1, 1)$ and a **direction vector** of $(+, +)$.
   - Again, using the delta method we get the following two equations,

     $$I = I + \Delta I + 1 \tag{1}$$
     $$I - J = I + \Delta I - J - \Delta J + 1 \tag{2}$$

     Solving these equations simultaneously, we get $\Delta I = -1$ and $\Delta J = 0$. This implies a **loop-carried anti-dependence in loop i** with a **distance vector** of $(1, 0)$ and a **direction vector** of $(+, 0)$.

2. Using the data dependences given for the only statement in the innermost loop, we construct the dependence matrix as shown below.

   $$\begin{bmatrix} 1 & 0 & -1 & 1 \\ 1 & -1 & 0 & 1 \\ 0 & 1 & 0 & -2 \end{bmatrix}$$

   (a) Since **unrolling** a loop is always valid irrespective of the dependence vectors, hence **we can unroll the loops t, i, j and k.**
   Unrolling & Jamming a loop is equivalent to bringing that loop to the innermost level. Hence we can see that **only loops j and k can be unrolled and jammed** since unrolling & jamming loop t will result in dependence vectors $(0, -1, 1, 1)$, $(-1, 0, 1, 1)$ and i in dependence vector $(0, 0, -2, 1)$ all of which are invalid due to the first nonzero element being negative.

(b) While permuting the loops, it is necessary that after applying the same permutation to the columns of the dependence matrix, the dependence vectors should remain valid i.e. the first nonzero element of all dependences should remain positive. Hence **all such valid permutations are** $tijk$**,** $tikj$ **and** $tjik$.

(c) The validity condition for tiling a band of loops is that it should be fully permutable. **2-D tiling is possible since bands** ij **and** jk **are fully permutable** (Note that 1-D tiling of each loop is also possible but is trivial and won't bring in any performance improvement).

(d) **The loops** j **and** k **are parallel denoted by** $j||k$. This is because loops j and k do not carry any dependences and hence can be fully vectorized.

(e) The code for $tikj$ permutation is given below.

```
int i, j, t, k;
for (t = 0; t < 2048; t++) {
    for (i = t; i < 1024; i++) {
        for (k = 1; k < i; k++) {
            for (j = max(t, k+1); j < i; j++) {
                S(t, i, j, k);
            }
        }
    }
}
```

3. Looking at the only statement in the loop nest, there appears to be 5 dependences. The analysis for each pair of array references lying on either side of the equality is done below using the **Delta Test**.

(a)
- $A[i,j]$ and $A[i-1,j]$
  Let the array references match in the loops (T,I,J) and $(T+\Delta T, I+\Delta I, J+\Delta J)$. Then we have the equations,

$$I = I + \Delta I - 1 \quad (1)$$
$$J = J + \Delta J \quad (2)$$

  Solving this gives $\Delta I = 1$ and $\Delta J = 0$ where ΔT can take any value (+ve or -ve). Hence we have,
  - $\Delta T > 0$: we have a **loop-carried flow-dependence** having **distance vector** $(\delta t, 1, 0)$ where $1 \leq \delta t \leq 4094$ and **direction vector** $(+,+,0)$.
  - $\Delta T = 0$: we have a **loop-carried flow-dependence** having **distance vector** $(0,1,0)$ and **direction vector** $(0,+,0)$
  - $\Delta T < 0$: we have a **loop-carried anti-dependence** having **distance vector** $(\delta t, -1, 0)$ where $1 \leq \delta t \leq 4094$ and **direction vector** $(+,-,0)$

- $A[i,j]$ and $A[i,j]$
  Proceeding similarly as above, we have the equations,

$$I = I + \Delta I \quad (1)$$
$$J = J + \Delta J \quad (2)$$

  This gives $\Delta I = \Delta J = 0$ where ΔT could take any value (+ve or -ve). Hence we have,
  - $\Delta T > 0$: we have a **loop-carried flow-dependence** having **distance vector** $(\delta t, 0, 0)$ where $1 \leq \delta t \leq 4094$ and **direction vector** $(+,0,0)$.
  - $\Delta T = 0$: we have a **loop-independent anti-dependence** having **distance vector** $(0,0,0)$ and **direction vector** $(0,0,0)$.
  - $\Delta T < 0$: we have a **loop-carried anti-dependence** having **distance vector** $(\delta t, 0, 0)$ where $1 \leq \delta t \leq 4094$ and **direction vector** $(+,0,0)$.

- $A[i, j]$ and $A[i + 1, j]$
  We have the equations,

$$I = I + \Delta I + 1 \quad (1)$$
$$J = J + \Delta J \quad (2)$$

  This gives $\Delta I = -1$, $\Delta J = 0$ and once again ΔT can take any value (+ve or -ve). Hence we have,
  - $\Delta T > 0$: we have a **loop-carried flow-dependence** having **distance vector** $(\delta t, -1, 0)$ where $1 \leq \delta t \leq 4094$ and **direction vector** $(+, -, 0)$.
  - $\Delta T = 0$: we have a **loop-carried anti-dependence** having **distance vector** $(0, -1, 0)$ and **direction vector** $(0, -, 0)$.
  - $\Delta T < 0$: we have a **loop-carried anti-dependence** having **distance vector** $(\delta t, 1, 0)$ where $1 \leq \delta t \leq 4094$ and **direction vector** $(+, +, 0)$.

- $A[i, j]$ and $A[i, j - 1]$
  We have the equations,

$$I = I + \Delta I \quad (1)$$
$$J = J + \Delta J - 1 \quad (2)$$

  This gives $\Delta I = 0$, $\Delta J = 1$ and once again ΔT can take any value (+ve or -ve). Hence we have,
  - $\Delta T > 0$: we have a **loop-carried flow-dependence** having **distance vector** $(\delta t, 0, 1)$ where $1 \leq \delta t \leq 4094$ and **direction vector** $(+, 0, +)$.
  - $\Delta T = 0$: we have a **loop-carried flow-dependence** having **distance vector** $(0, 0, 1)$ and **direction vector** $(0, 0, +)$.
  - $\Delta T < 0$: we have a **loop-carried anti-dependence** having **distance vector** $(\delta t, 0, -1)$ where $1 \leq \delta t \leq 4094$ and **direction vector** $(+, 0, -)$.

- $A[i, j]$ and $A[i, j + 1]$
  We have the equations,

$$I = I + \Delta I \quad (1)$$
$$J = J + \Delta J + 1 \quad (2)$$

  This gives $\Delta I = 0$, $\Delta J = -1$ and once again ΔT can take any value (+ve or -ve). Hence we have,
  - $\Delta T > 0$: we have a **loop-carried flow-dependence** having **distance vector** $(\delta t, 0, -1)$ where $1 \leq \delta t \leq 4094$ and **direction vector** $(+, 0, -)$.
  - $\Delta T = 0$: we have a **loop-carried anti-dependence** having **distance vector** $(0, 0, 1)$ and **direction vector** $(0, 0, +)$.
  - $\Delta T < 0$: we have a **loop-carried anti-dependence** having **distance vector** $(\delta t, 0, 1)$ where $1 \leq \delta t \leq 4094$ and **direction vector** $(+, 0, +)$.

The matrix of all direction vectors is shown below.

$$\begin{bmatrix} + & + & 0 \\ 0 & + & 0 \\ + & - & 0 \\ + & 0 & 0 \\ 0 & 0 & 0 \\ 0 & - & 0 \\ + & 0 & + \\ 0 & 0 & + \\ + & 0 & - \end{bmatrix}$$

(b) Note that loop t cannot be moved anywhere inside without invalidating direction vectors, hence only loops i and j can be interchanged. Hence **only the permutations tij and tji are valid**.

(c) Since unrolling a loop is always valid irrespective of the dependence vectors, hence **we can unroll the loops t, i and j.**
Also, since loop t cannot be brought to the innermost level while loops i and j can, hence **only loops i and j can be unrolled and jammed**.

(d) Since the 2 innermost loops i and j are fully permutable while the loop t cannot be shifted anywhere inside, hence **only 2-D tilings of the ij band is valid** (Note that 1-D tiling of each loop is also possible but is trivial and won't bring in any performance improvement).

4. The solution file **"problem4.cpp"** for this problem has been provided with this assignment solution. Run it using the following bash commands.

```
$ g++ problem4.cpp -o problem4 -lpthread
$ echo $N | ./problem4
```

In the above bash code snippet, replace **$N** with a value in the range $[0, 24)$ to simulate the bank system on the desired number of customers. Alternatively, one can also run the binary normally, and enter the value of **$N** at the prompt.

5. Please note that all benchmarking for this problem was performed on a **Macbook Pro with Dual core Intel i5 processor (5th Gen)**. Even though each core can support a maximum of 2 threads at a time (Hyperthreading) best performance is often obtained with 1 thread per core. The L2 cache capacity per core is 256 KB while the L3 cache capacity is 3MB.

(a) The sequential implementation of the 4096 x 4096 matrix multiplication runs in **1175.5328 seconds ($\approx$ 19.6 minutes)**. After parallelizing the outermost loop i by chunking and distributing equally across each thread, the following performance was obtained,
The major reason why the speedup factors for 2 and 4 does not scale linearly is because of the extra

Number of Threads	Runtimes (in seconds)	Speedup Factor
2	826.7904	1.4218
4	712.9024	1.6489
8	736.9024	1.5952

Table 1: Runtimes and speedups of parallelized matrix multiplication

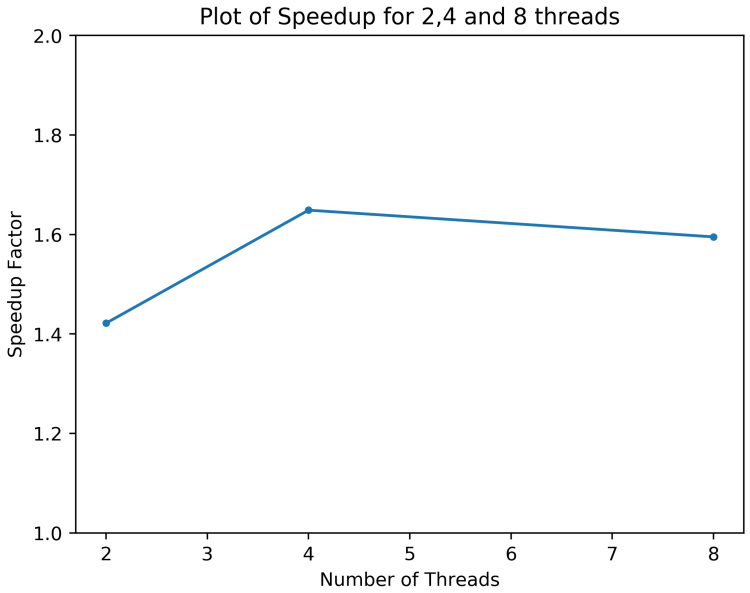

Figure 1: Plot of Speedup vs Number of Threads

cost of thread creation. Another reason is that the host system supports 1 thread per core for best

performance. Also, it is observable that for 8 threads the cost of thread creation negatively offsets the parallelization advantage and hence there is a decrease in performance compared to 4 threads.

(b)

- **Optimized Sequential Matrix Multiplication:**
  For optimizing the sequential matrix multiplication code, several optimization techniques such as **Blocking**, **Loop Unrolling and Jamming** and several combinations of both were applied. For blocking, 3-D blocks of sizes N x N x N were tried where $N \in \{2, 4, 8, 16, 32, 64, 128, 256\}$. Loop unrolling/jamming was performed for loops i and j both individually as well as together with step sizes of $4, 8$ and 16. The **best performance for blocking was obtained with a block size of 16 x 16 x 16 (Speedup 3.2956)** while the **best performance for loop unrolling/jamming was obtained by unrolling/jamming loops i and j together with step sizes of 8 and 8 (Speedup 2.8230)**. The **combined best performance using both blocking and unrolling/jamming was obtained with a block size of 32 x 32 x 32 and an unroll/jam step size of 16 for loop j (Best speedup of 5.3874 was observed)**.

Block Size	Runtimes (in seconds)	Speedup Factor
2x2x2	1212.0512	0.9699
4x4x4	479.7158	2.4505
8x8x8	364.0512	3.2290
16x16x16	356.6995	3.2956
32x32x32	372.5984	3.1549
64x64x64	436.4960	2.6931
128x128x128	458.7821	2.5623
256x256x256	469.1917	2.5054

Table 2: Runtimes and speedups in Sequential Matrix Multiplication using Blocking

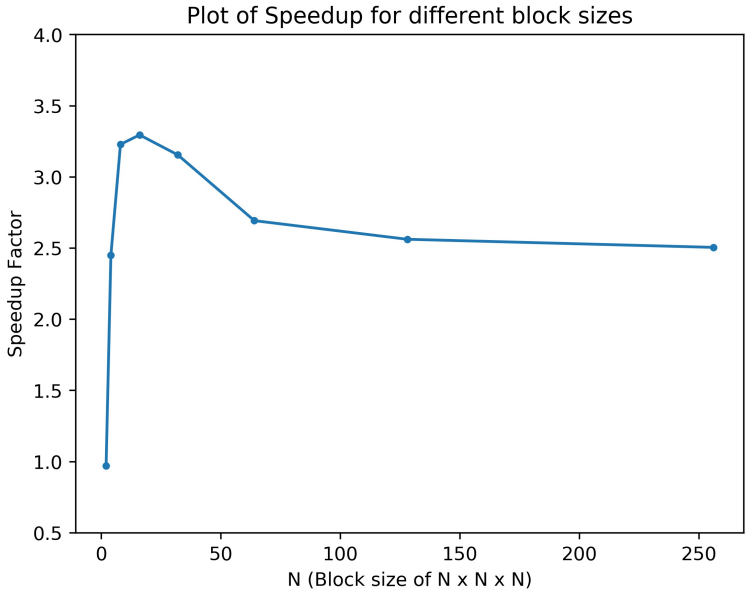

Figure 2: Plot of Speedup vs Blocking Size for Optimized Sequential Matrix Multiplication

- **Optimized Parallelized Matrix Multiplication:**
  Similar optimization techniques were applied for the parallelized version of the code. The best performing code after experimenting with several combinations of blocking, unrolling/jamming and the number of threads was obtained using **4 threads on 32x32x32 block size with loop j unrolled/jammed 16 times (Best speedup of 11.9008 was observed)**. The optimum number of threads can easily be explained by the fact that the host system supported a maximum of 4 threads and creating new threads would only deteriorate performance due to extra thread creation

Loop Unrolled/Jammed	Step Size	**Runtimes (in seconds)**	**Speedup Factor**
i	2	1216.32	0.9665
i	4	915.3728	1.2842
i	8	896.864	1.3107
i	16	653.6064	1.7985
j	2	1184.864	0.9921
j	4	980.9408	1.1984
j	8	890.4064	1.3202
j	16	613.3497	1.9166
i & j	8 & 8	416.4032	2.8231

Table 3: Runtimes and speedups in Optimized Sequential Matrix Multiplication using unrolling/jamming

Combination	**Runtimes (in seconds)**	**Speedup Factor**
4x4x4, jam j with 4	466.7884	2.5183
8x8x8, jam j with 4	279.6684	4.2033
8x8x8, jam j with 8	301.1353	3.9037
16x16x16, jam j with 8	234.7136	5.0084
16x16x16, jam j with 16	221.6691	5.3031
32x32x32, jam j with 16	218.2003	5.3874
16x16x16, jam (i,j) with (8,8)	322.9612	3.6398
32x32x32, jam (i,j) with (8,8)	302.0864	3.8914

Table 4: Runtimes and speedups in Optimized Sequential Matrix Multiplication using combination of blocking and unrolling/jamming

Combination	**Runtimes (in seconds)**	**Speedup Factor**
16x16x16, jam j with 16, threads = 2	120.4499	9.7595
32x32x32, jam j with 16, threads = 2	101.9232	11.5335
32x32x32, jam j with 16, threads = 4	98.7776	11.9008
16x16x16, jam (i,j) with (8,8), threads = 4	138.8544	8.4659
32x32x32, jam (i,j) with (8,8), threads = 4	134.4422	8.7438
16x16x16, jam j with 16, threads = 8	115.3721	10.1890
32x32x32, jam j with 16, threads = 8	100.2649	11.7243

Table 5: Runtimes and speedups in Optimized Parallelized Matrix Multiplication using combination of blocking and unrolling/jamming

overheads. The block size and unroll step size is the same as for the best performing optimized sequential code.