Parallel Computing

COMP 633  Fall 2019
Written Assignment 2: Sample Solutions

I. Design a parallel algorithm to compute \( C = B^T \) where \( B^T \) is the transpose of \( B \), an \( n \times n \) matrix of (8-byte) doubles, and \( n = 2^h \) for some \( h > 0 \). Assume the input matrix \( B \) is stored in row-major order in memory. The result matrix \( C \) should be stored the same way. Assume a uniform access shared memory with size sufficient for the input, output, and any intermediate results. The algorithm should target a simple parallel memory hierarchy in which each processor has a single fully associative L1 data cache with a cache line size \( L = 64 \) bytes and a capacity of \( S = 256 \) lines. Assume the cache evicts the least recently used cache line when filled to capacity.

The transpose requires \( O(n^2) \) memory references to read the input and write the result. Even though there are no arithmetic operations, we’ll view this as \( O(n^2) \) total work. Thus the problem has \( O(1) \) computational intensity.

(a) What is the minimum number of cache misses necessary to perform the transpose operation in terms of \( n, L \) and \( S \)?

(b) Outline an algorithm meeting your bounds in (a) using pseudocode with OpenMP-like parallelization directives for \( p = 2^j \) (\( j \geq 0 \)) processors. It is not necessary to construct a running program – just explain your approach. To simplify, you can require a minimum size problem \( n \) when using \( p \) processors – just state your restriction.

(c) Now assume the shared memory is distributed across \( p \) processors, and that full memory performance is obtained only when concurrent memory references are equally distributed across the memories at all processors. Assume the matrix \( B \) was allocated so that rows \( (i-1) \cdot (n/p) : i \cdot (n/p) \) are in memory at processor \( 1 \leq i \leq p \). Can you modify your transpose algorithm to achieve full memory performance?

SAMPLE SOLUTION

(a) The transpose requires every element of \( B \) to be read, and requires every element of \( C \) to be written. If we read elements from \( B \) in row-major order the values will be contiguous in memory, and there will be a compulsory cache miss every \( L/8 \) elements, since double values occupy eight bytes each. Hence, to read \( B \) in its entirety requires \( n^2/(L/8) = 8n^2/L \) cache misses. Similarly in writing \( C \), which has the same size as \( B \), a minimum of \( 8n^2/L \) cache lines must be written. Thus the lower bound for the transpose operation is \( 16n^2/L \) misses – we cannot read \( B \) and write \( C \) with fewer cache to memory transfers.

To achieve the lower bound for writing we must avoid write-allocate cache misses, which normally result from a write to a location for which unwritten locations in the same cache line have to be supplied from memory. Processors delay the write-allocation fetch when a cache line is being filled in its entirety with writes that address consecutive locations in the same cache line. This optimization is important in increasing the performance of large copy operations.

(b) First, let us examine the problem for a single-processor memory hierarchy, and then extend it to the parallel memory hierarchy and parallel execution. To achieve the lower bound in part (a), elements have to be read contiguously from \( B \), and have to be written contiguously into \( C \). Our problem is that contiguous values read from \( B \) are non-contiguous in their location in \( C \). The way forward is to read a square block of \( B \) into the cache, and then access this block column-wise within the cache in order
to write the results into \( C \) in row-major order. To simplify presentation of this strategy we need a few definitions.

**Blocked matrices.** If \( n = mk \) we can view an \( n \times n \) matrix as an \( m \times m \) blocked matrix where each block has size \( k \times k \). For a blocked matrix \( M \) we write \((M)_{ij}\) to denote the \( k \times k \) block at position \((i,j)\). The index relationship between a blocked matrix \( M \) and its \( n \times n \) representation \( \bar{M} \) is

\[
\bar{M}_{i\hat{k}+\hat{i},j\hat{k}+\hat{j}} = (\mathbf{(M)}_{ij})_{\hat{i}\hat{j}} \quad \text{where} \quad 0 \leq i,j < m \quad \text{and} \quad 0 \leq \hat{i},\hat{j} < k
\]

**Transposition.** If \( B \) and \( C \) are blocked matrices such that \( C = B^T \) then

\[
(C)_{ij} = (\mathbf{(B)}_{\hat{i}\hat{j}})^T
\]

In words, the block at position \((i,j)\) of \( C \) is the transpose of the block at position \((j,i)\) in \( B \). Using these definitions, we can write the transpose as \( m^2 \) block transpositions:

\[
\text{for } i, j \in \{0, \ldots, m-1\} \times \{0, \ldots, m-1\} \quad \quad (C)_{ij} \leftarrow (\mathbf{(B)}_{\hat{i}\hat{j}})^T
\]

The operation in the box corresponds to the following sequential code in which \( T \) is a \( k \times k \) temporary array intended to be cache resident throughout the entire transpose operation (this will be achieved because \( T \) is repeatedly re-used in the inner most loop, so its elements are never part of the least recently used cache line).

```
for (i = 0; i < k; i++)
  for (j = 0; j < k; j++)
    T_{ij} = B_{j\hat{k}+\hat{i},i\hat{k}+\hat{j}}

for (i = 0; i < k; i++)
  for (j = 0; j < k; j++)
    C_{i\hat{k}+\hat{i},j\hat{k}+\hat{j}} = T_{ji}
```

The first nested loop reads the \((j,i)\) block of \( B \) in row major order and copies it into \( T \). The second nested loop accesses elements in \( T \) in column major order within the cache and writes the \((i,j)\) block of \( C \) in row major order. Writing blocks of \( C \) in this order important because it avoids write-allocate cache misses.

An alternative strategy to avoid write allocation traffic is to read the block of \( B \) in column major order and write it directly to \( C \) in row major order:

```
for (i = 0; i < k; i++)
  for (j = 0; j < k; j++)
    C_{i\hat{k}+\hat{i},j\hat{k}+\hat{j}} = B_{j\hat{k}+\hat{j},i\hat{k}+\hat{i}}
```

While this strategy does not use the temporary array, it generates \( k \) consecutive misses in block \((B)_{\hat{i}\hat{j}}\) at the start of the loop instead of spreading them out evenly. This might decrease the ability of the processor to overlap other work with cache misses, so in practice it may run a bit slower.

What value of \( k \) should we choose? The cache line size is 64 bytes, and the elements of the block each occupy 8 bytes. Thus a \( k \times k \) block will occupy \( k^2/8 \) cache lines when \( k \) is divisible by 8. Depending on our strategy we will need either three or two blocks cache resident. Thus we need \((3k^2)/8 < 256 \text{ or } k < 26 \) for the first strategy, or \( k < 32 \) for the second strategy. Since \( n = mk \) and
A larger \( k \) might amortize block copy overhead better, but uses more cache. Since the columnwise access of the cache likely will only run at full performance in the lowest level L1 of the memory hierarchy, a small cache footprint is a wise idea so \( k = 8 \) is a good choice, enabling use of an L1 cache with as few as \( S = 24 \) lines available for the transpose.

The \( m^2 = (n/k)^2 \) blocks of \( B \) can be transposed into \( C \) in any order – each block is entirely independent in the set of cache lines it references in memory. So for the sequential case the order does not matter and we choose an order that reads blocks of \( B \) in column major order and places them into blocks of \( C \) in row major order, consistent with our exposition thus far. The column major access of blocks of \( B \) does not yield poor cache behavior, because at the level of blocks the cache behaves well regardless of the block’s location.

For the parallel memory hierarchy, under the assumption of uniform memory access, we can simply replace the for loop with a forall construct. The latter can be implemented in OpenMP with a directive to indicate how the loop iterations are to be split up. For example we could give each thread an equal number of iterations in the outermost loop as follows.

```c
#pragma omp parallel for private(i,j,ib,jb) schedule=static
for (i = 0; i < m; i++){
    for (j = 0; j < m; j++){
        for (ib = 0; ib < k; ib++) {
            for (jb = 0; jb < k; jb++){
                // (C)_{ij} \leftarrow ((B)_{ji})^T
                C[k*ib + kb, k*jb] = B[k*jb + k*ib];
            }
        }
    }
}
```

Since each block fits precisely in a fixed number of cache lines, and all blocks are disjoint, there will be no coherence misses between concurrent block-transpose operations. If we had a large number of processors available we could use the `collapse(2)` directive in OpenMP to schedule the entire 2D iteration space \( \{0, \ldots, m - 1\} \times \{0, \ldots, m - 1\} \) across \( p \) processors.

How large must \( n \) be to enable \( p \) processors to be used? Since \( p = m^2 = (n/k)^2 = (n/8)^2 \) this tells us \( n \geq 8\sqrt{p} \) so, in principle, with 1024 processors we could transpose a \( 256 \times 256 \) array concurrently in the time to service 16 sequential cache misses (such a shared memory system must supply three orders of magnitude more memory bandwidth than it provides to a single processor – don’t hold your breath).

(c) In the CC-NUMA shared memory model, \( B \) and \( C \) are distributed across the local memories of the processors. The distribution of \( B \) places successive groups of \( n/p \) rows in successive processors. We assume that \( n/p \geq 8 \) so the rows in the local memory of each processor correspond to an integral number of blocks \( n^2/64p \) blocks). Recall \( B \) and \( C \) are distributed the same way. With this arrangement, we can simply execute the OpenMP program above with the parallelization directive on the outer loop.

With the default static schedule implied by this approach, processor \( 0 \leq \ell < p \) will read all blocks in the \( \ell \)th vertical “strip” of \( B \), i.e. columns \( \ell \cdot (n/p) : (\ell + 1) \cdot (n/p) - 1 \) which has blocks distributed across all processor memories and write all blocks in the \( \ell \)th horizontal “strip” of \( C \) (all of which reside in its local memory). While this may appear reasonable, the particular schedule of accesses to \((B)_{ji}\) blocks given here will start with all \( p \) processors fetching some block \((B)_{0,i_0}\) where \( i_0 = \ell \cdot (n/p) \), each of which are located in the memory of processor 0. In the next iteration of the \( j \) loop all \( p \)
processors will be referencing block \((B)_{1,i}\), which again are all located in the same processor. In fact any given iteration \((i_p,j)\) executed concurrently (with \(p\) different values for \(i_p\)) will result in \(p\) reads of block \((B)_{j,i}\), all of which reside in a single processor’s memory, which leads to contention for available memory bandwidth at that single processor. These “hotspots” dramatically limit the scalability of the algorithm as the number of processors increase. To ensure that the writes on \(C\) are equally distributed across processors, the iteration space over \(j\) can be “skewed” as follows:

```c
#pragma omp parallel for private(i,jp) schedule=static
for (i = 0; i < m; i++){
    for (jp = 0; jp < m; jp++) {
        int j = (procid * (m/p) + jp)% m
        // (C)_{ij} = (B)_{ji}^T
    }
}
```

This ensures that different processors start reading blocks with different \(j\) offsets in order to distribute the reads across the memories of all processors. A barrier synchronization could be used to guarantee strict separation between successive iterations on \(i\) in each processor, but it will add overhead. Since all we’re trying to do is avoid the worst case, rather than ensure the best case with high cost, we may well be better off, on average, not using barriers.

II. Threads T1 and T2 share variables \(a\), \(b\) and \(c\) in memory. Initially \(a = b = c = 0\). Assume that \(r1\), \(r2\) and \(r3\) are processor registers and consider the following sequence of instructions executed by the two threads as shown below. For each of the 8 conceivable values that might be printed for \((r1,r2,r3)\) from \((0,0,0)\) to \((1,1,1)\), give the strongest memory consistency model for which the result is a permitted outcome. The consistency models to choose from are (from strongest to weakest): sequential consistency, total store ordering (TSO), partial store ordering (PSO), and inconsistent.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1:</td>
<td>(r2 := b)</td>
<td>(a := 1)</td>
</tr>
<tr>
<td>A2:</td>
<td>(c := 1)</td>
<td>(r3 := c)</td>
</tr>
<tr>
<td>A3:</td>
<td>(r1 := a)</td>
<td>(b := 1)</td>
</tr>
<tr>
<td>A4:</td>
<td>(\text{print}(r1,r2))</td>
<td>(\text{print}(r3))</td>
</tr>
</tbody>
</table>

**SAMPLE SOLUTION**

First, we must be clear what are the operations performed by each thread. We label the statements in the two threads as shown above. Statements A2, B1, and B3 consist of a single write operation. Statements A1, A3, and B2 consist of a single read operation (registers are processor specific and are not memory). Statements A4 and B4 are observations of values in the thread’s registers, and are treated as reads.

Here are the partial orders that must be preserved among the statements of each thread under the various consistency models:

- **SEQ:** \(A1\to A2\to A3\to A4\) \(\quad B1\to B2\to B3\to B4\)
- **TSO:** \(A1\to A3\to A4\), \(A1\to A2\) \(\quad B2\to B4\), \(B1\to B3\), \(B2\to B3\)
- **PSO:** \(A1\to A3\to A4\), \(A1\to A2\) \(\quad B2\to B4\), \(B2\to B3\)

For an execution to be observable under a given consistency model, it must preserve all the orderings for the consistency model as shown above.
The table below shows the ordering of an execution sequence that establishes the desired result and identifies the strongest consistency model that admits the result (or that the result is impossible).

<table>
<thead>
<tr>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>consistency model</th>
<th>execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TSO</td>
<td>A1 A3 A4 B2 B4 A2 B1 B3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>SEQ</td>
<td>A1 A2 A3 A4 B1 B2 B3 B4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>PSO</td>
<td>B2 B3 B4 A1 A2 A3 A4 B1</td>
</tr>
</tbody>
</table>
| 0  | 1  | 1  | inconsistent      | T1: B3 → A1 → A2  
T2: A2 → B2 → B3 |
| 1  | 0  | 0  | SEQ               | B1 B2 A1 A2 A3 A4 B3 B4 |
| 1  | 0  | 1  | SEQ               | B1 A1 A2 A3 A4 B2 B3 B4 |
| 1  | 1  | 0  | SEQ               | B1 B2 B3 B4 A1 A2 A3 A4 |
| 1  | 1  | 1  | inconsistent      | T1: B3 → A1 → A2  
T2: A2 → B2 → B3 |

For the two inconsistent cases the explanation is the same: the “→” that indicates “must precede” does so either

(a) by causality (T1 reads value 1 for variable b in A1, so this means that statement B3 in T2 must have preceded it), or

(b) because a read operation must be performed before a later read or write operation in the same thread under all non-weak consistency models.

Combining the relations that must hold in T1 and T2 yields a precedence cycle, hence corresponds to an inconsistent memory model.