Matrix Multiplication

Mark Greenstreet

CpSc 448B – Sept. 27, 2011

Outline:

- Sequential Matrix Multiplication
- Parallel Implementations, Performance, and Trade-Offs.

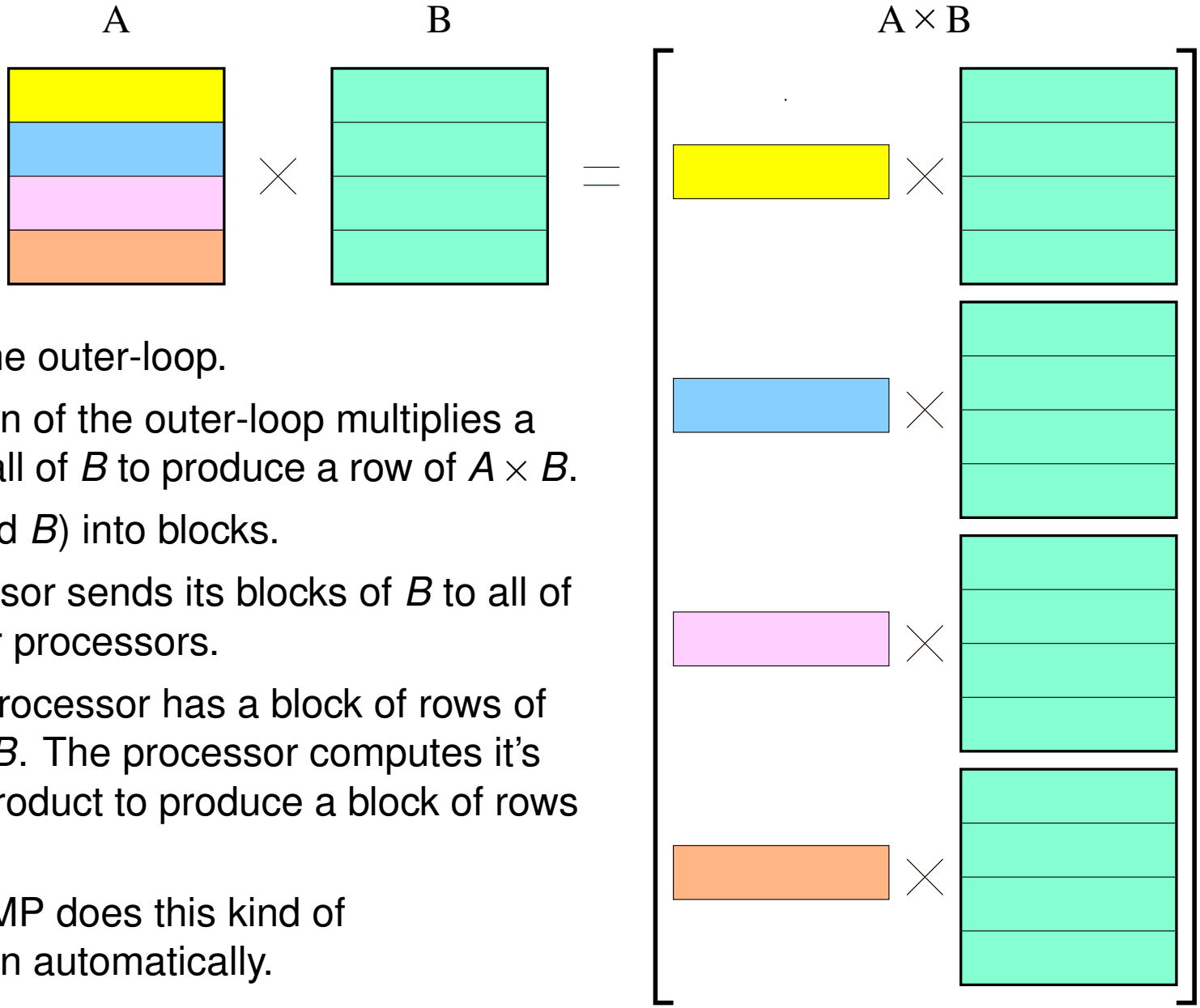
Sequential Matrix Multiplication

```
for(int i = 0; i < n_rows_a; i++) {
    for(int j = 0; j < n_cols_b; j++) {
        sum = 0.0;
        for(int k = 0; k < n_cols_a; k++) {
            % n_cols_a = n_rows_b
            sum += a[i,k] * b[k,j];
        }
        c[i,j] = sum;
    }
}
```

Performance

- Really simple, operation counts:
  - Multiplications: `n_rows_a` $*$ `n_cols_b` $*$ `n_cols_a`.
  - Additions: `n_rows_a` $*$ `n_cols_b` $*$ (`n_cols_a` $-$ 1).
  - Memory-reads: 2$*$#Multiplications.
  - Memory-writes: `n_rows_a` $*$ `n_cols_b`.
  - Time is $O($`n_rows_a` $*$ `n_cols_b` $*$ (`n_cols_a` $-$ 1)), If both matrices are $N \times N$, then its $O(N^3)$.
- But, memory access can be terrible.
  - For example, let matrices `a` and `b` be 1000 $\times$ 1000.
  - Assume a processor with a 4M L2-cache (final cache), 32 byte-cache lines, and a 200 cycle stall for main memory accesses.
  - Observe that a row of matrix `a` and a column of `b` fit in the cache. (a total of $\sim$40K bytes).
  - But, all of `b` does not fit in the cache (that's 8 Mbytes).
  - So, on every fourth pass through the inner loop, every read from `b` is a cache miss!
  - The cache miss time would dominate everything else.
- This is why there are carefully tuned numerical libraries.

Parallel Algorithm 1



- Parallelize the outer-loop.
- Each iteration of the outer-loop multiplies a row of A by all of B to produce a row of $A \times B$.
- Divide A (and B) into blocks.
- Each processor sends its blocks of B to all of the the other processors.
- Now, each processor has a block of rows of A and all of B. The processor computes it's part of the product to produce a block of rows of C.
- Note: OpenMP does this kind of parallelization automatically.

Algorithm 1 in Erlang

```
par_matrix_mult1(ProcList, MyIndex, MyBlockA, MyBlockB) ->
    NProcs = length(ProcList),
    % send MyBlock to all other processes
    [ P ! {MyIndex, MyBlock} || P <- ProcList],
    % receive all the blocks
    Bblocks = [    receive  I, Block  -> Block end
                || I <- lists:seq(1,NProcs) ],
    % concatenate these blocks to make the B matrix
    B = lists:append(Bblocks),
    matrix:mult(MyBlockA, MyBlockB). % our block of A*B
```

The math:

- Let $A(i,:)$ denote the i^{th} row of A, and
- Let $B(:,j)$ denote the j^{th} column of B.
- Let $C = A * B$ we have: $C(i,:) = A(i,:) * B$.
- In English:
  - The processor that holds a block of rows of A can compute the corresponding rows of C.
  - The processor has to have all of B. That's what the sends and receives do at the begining of `par_matrix_mult1`.

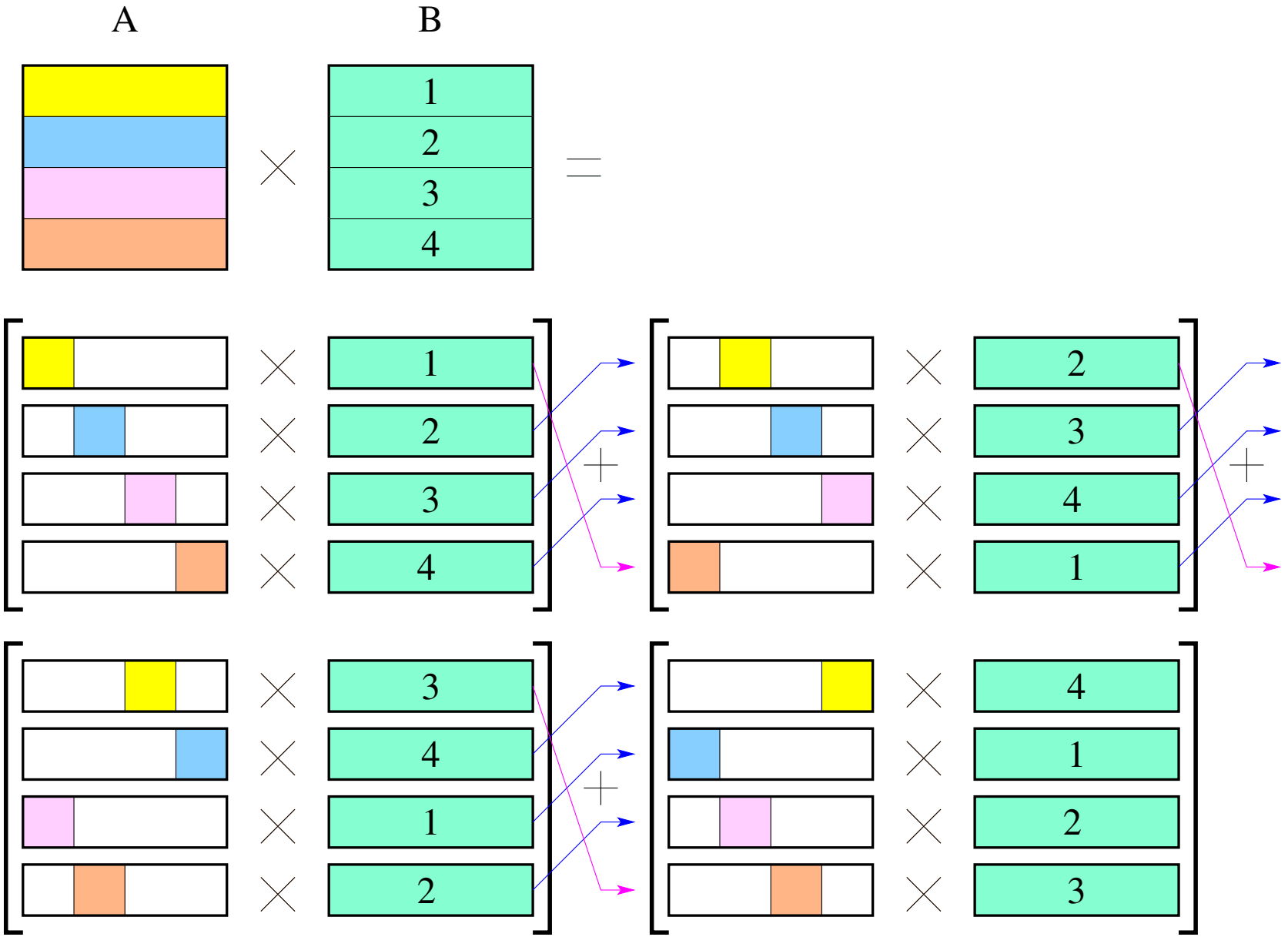
Performance of Parallel Algorithm 1

- CPU operations: same total number of multiplies and adds, but distributed around P processors. Total time: $O(N^3/P)$.
- Communication: Each processors sends (and receives) $P-1$ messages of size N^2/P. If time to send a message is $t_0 + t_1 * M$ where M is the size of the message, then the commmunication time is

$$\begin{aligned}(P-1)\left(t_0 + t_1\frac{N^2}{P}\right) &= O(N^2+P), \quad \text{but, beware of large constants} \\ &= O(N^2), \qquad\qquad N^2 > P\end{aligned}$$

- Memory: Each process needs $O(N^2/P)$ storage for its block of A and the result. It also needs $O(N^2)$ to hold all of B.
  - The simple algorithm divides the computation across all processors, but it doesn't make good use of their combined memory.

Parallel Algorithm 2 (illustrated)



Parallel Algorithm 2 (code sketch)

- Each processor first computes what it can with its rows from A and B.
  - It can only use N/P of its columns of its block from A.
  - It uses its entire block from B.
  - We've now computed one of P matrices, where the sum of all of these matrices is the matrix AB.
- We view the processors as being arranged in a ring,
  - Each processor forwards its block of B to the next processor in the ring.
  - Each processor computes an new partial product of AB and adds it to what it had from the previous step.
  - This process continues until every block of B has been used by every processor.

Algorithm 2, Erlang

```
par_matrix_mult1(ProcList, MyIndex, MyBlockA, MyBlockB) ->
   NProcs = length(ProcList),
   NRowsA = length(A),
   NColsB = length(hd(B)), % assume length(B) > 0
   ABlocks0 = rotate(MyIndex, blockify_cols(A, NProcs)),
   PList = rotate(NProcs - (MyIndex-1),
                  lists:reverse(ProcList)),
   helper(ProcList, ABlocks, MyBlockB,
          matrix:zeros(NRowsA, NColsB)).

helper([P_head | P_tail], [A_head | A_tail], BBlock, Accum) ->
   if A_tail == [] -> ok;
      true         -> P_head !  BBlock
   end,
   Accum2 = matrix:add(Accum, matrix:mult(A_head, BBlock)),
   if A_tail == [] -> Accum2;
      true ->
         helper(P_tail, A_tail, receive BBlock2 -> BBlock2 end,
   end.
```

Algorithm 2 – notes on the Erlang code

- `blockify_cols(A, NProcs)` produces a list of `NProcs` matrices.
  - Each matrix has `NRowsA` rows and `NColsA` columns,
  - where `NColsA` is the number of columns of `MyBlockA`.
  - Let $A(\texttt{MyIndex}, j)$ denote the j^{th} such block.
- ```
  rotate(N, List) ->
      {L1, L2} = lists:split(N, List),
      L2 ++ L1.
  ```
- The algorithm is based on the formula:

$$C(\texttt{MyIndex}, :) \quad = \quad \sum_{j=1}^{\texttt{NProcs}} A(\texttt{MyIndex}, j) * B(j, :)$$
  ```

Performance of Parallel Algorithm 2

- CPU operations: Same as for parallel algorithm 1: total time: $O(N^3/P)$.
- Communication: Same as for parallel algorithm 1: $O(N + P)$.
  - With algorithm 1, each processor sent the same message to $P - 1$ different processors.
  - With algorithm 2, for each processor, there is one destination to which it sends $P - 1$ different messages.
  - Thus, algorithm 2 can work efficiently with simpler interconnect networks.
- Memory: Each process needs $O(N^2/P)$ storage for its block of A, its current block of B, and its block of the result.
  - Note: each processor might hold onto its original block of B so we still have the blocks of B available at the expected processors for future operations.
- Do the memory savings matter?

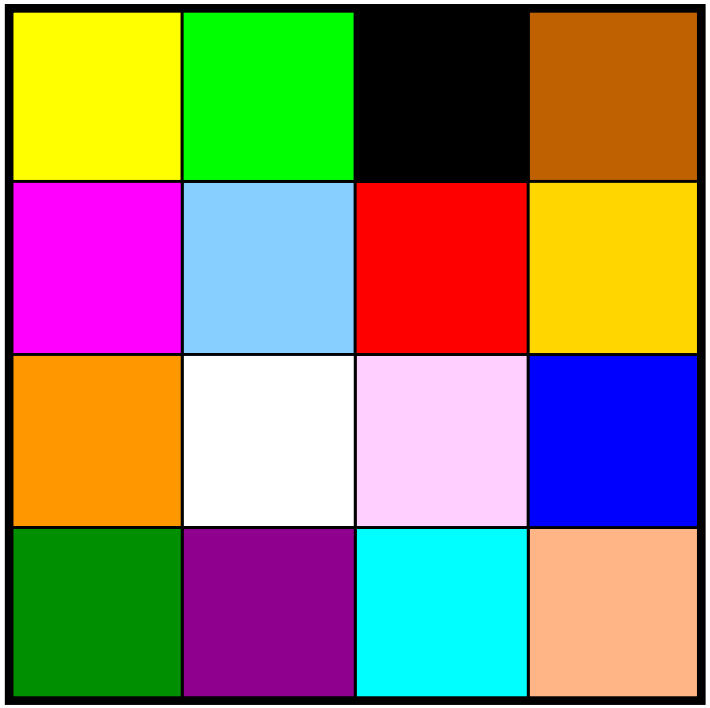
Bad performance, pass it on

- Consider what happens with algorithm 2 if one processor, P_{slow} takes a bit longer than the others one of the times its doing a block multiply.
  - P_{slow} will send it's block from B to its neighbour a bit later than it would have otherwise.
  - Even if the neighbour had finished its previous computation on time, it won't be able to start the next one until it gets the block of B from P_{slow}.
  - Thus, for the next block computation, both P_{slow} and its neighbour will be late, even if both of them do their next block computation in the usual time.
  - In other words, tardiness propagates.
- Solution: forward your block to you neighbour before you use it to perform a block computation.
  - This overlaps computation with communication, generally a good idea.
  - We could send two or more blocks ahead if needed to compensate for communication delays and variation in compute times.
  - This is a way to save time by using more memory.

Even less communication

- In the previous algorithms, computate time grows as N^3/P, while communication time goes as $(N + P)$.
- Thus, if N is big enough, computation time will dominate communication time.
- There's not much we can do to reduce the number of computations required (I'll ignore Strassen's algorithm, etc. for simplicity).
- If we can use less communication, then we'll we won't need our matrices to be as huge to benefit from parallel computation.

Other ways to distribute a matrix



Lower bound for communication