

Matrix Multiplication

Mark Greenstreet

CpSc 418 – Feb. 26, 2016

Outline:

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

The Midterm

- statistics

mean	87.4	$40 \leq G < 45$	*
median	87.5	$70 \leq G < 75$	***
std	10.4	$75 \leq G < 80$	*****
		$80 \leq G < 85$	*****
		$85 \leq G < 90$	*****
		$90 \leq G < 95$	*****
		$95 \leq G < 100$	*****
		$100 \leq G \leq 101$	****

- Other remarks:

Objectives

Apply concepts of algorithm analysis, parallelization, overhead, and performance measurement to a real problem.

- Design sequential and parallel algorithms for matrix multiplication.
- Analyse algorithms and measure performance.
- Identify bottlenecks and refine algorithms.

Matrix representation in Erlang

- I'll represent a matrix as a list of lists.
- For example, the matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \end{bmatrix}$$

is represented by the Erlang nested-list:

```
[ [1, 2, 3, 4]  
  [1, 4, 9, 16]  
  [1, 8, 27, 64] ]
```

- The empty matrix is `[]`.
 - ▶ This means my representation can't distinguish between a 2×0 matrix, a 0×4 matrix, and a 0×0 matrix.
 - ▶ That's OK. This package is to show some simple examples.
 - ▶ I'm not claiming it's for advanced scientific computing.

Sequential Matrix Multiplication

```
mult(A, B) ->
  BT = transpose(B),
  lists:map(
    fun(Row_of_A) ->
      lists:map(
        fun(Col_of_B) ->
          dot_prod(Row_of_A, Col_of_B)
        end, BT)
    end, A).

dot_prod(V1, V2) ->
  lists:foldl(
    fun({X,Y}, Sum) -> Sum + X*Y end,
    0, lists:zip(V1, V2)).
```

- Next, we'll use **list comprehensions** to get a more succinct version.

Matrix Multiplication, with comprehensions

```
mult(A, B) ->
  BT = transpose(B),
  [ [ dot_prod(RowA, ColB) || ColB <- BT ] || RowA <- A ].

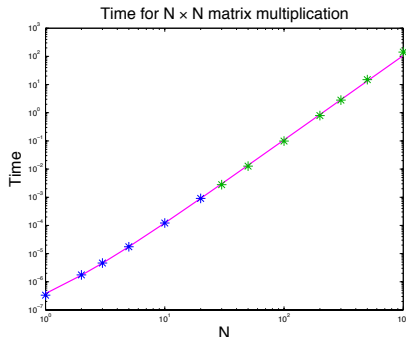
transpose([]) -> []; % special case for empty matrices
transpose([[ ]|-]) -> []; % bottom of recursion, the columns are empty
transpose(M) ->
  [ [H || [H | _T] <- M ] % create a row from the first column of M
    | transpose([ T || [_H | T] <- M ]) % now, transpose what's left
  ].
```

- `[Expr(X) || X <- List]` is equivalent to `lists:map(fun(X) -> Expr(X) end, List)`.
- And you can do much more with comprehensions.
- See slides 23 and 24 for examples.

Performance – Modeled

- 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: $2 * \# \text{Multiplications}$.
 - ▶ 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 1000×1000 .
 - ▶ 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.

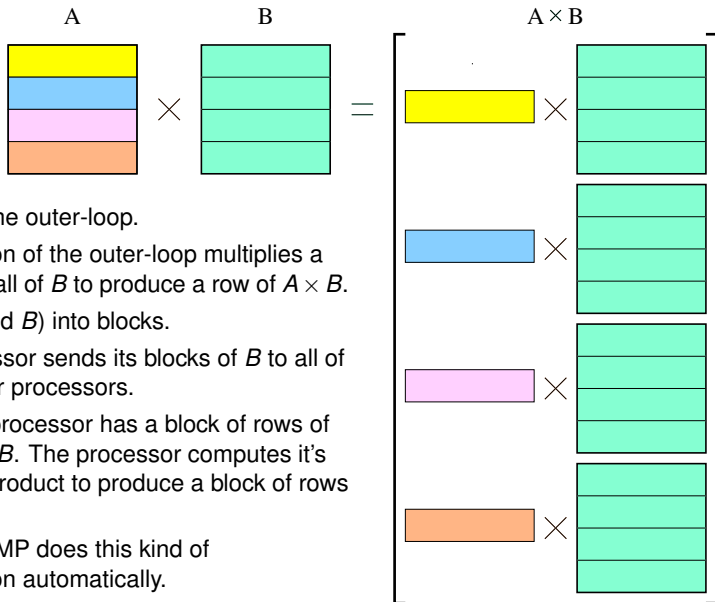
Performance – Measured



- Cubic of best fit: $T = (107N^3 + 134N^2 + 173N - 32)\text{ns}$.
- Fit to first six data points.
- Cache misses effects are visible, for $N=1000$:
 - ▶ model predicts $T = 107\text{seconds}$,
 - ▶ but the measured value is $T = 142\text{seconds}$.

Tiling Matrices

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.

Parallel Algorithm 1 in Erlang

```
% mult(W, Key, Key1, Key2) – create a matrix associated with Key
%   that is the product of the matrices associated with Key1 and Key2.
mult1(W, Key, Key1, Key2) ->
  Nproc = workers:nworkers(W),
  workers:update(W, Key,
    fun(PS, I) ->
      A = workers:get(PS, Key1), % my rows of A
      B = workers:get(PS, Key2), % my rows of B
      [WW ! {B, I} || WW <- W], % send my rows of B to everyone
      B_full = lists:append( % receive B from everyone
        [
          receive {BB, J} -> BB end
          || J <- lists:seq(1, Nproc)]),
      matrix:mult(A, B_full) % compute my part of the product
    end
  ).
```

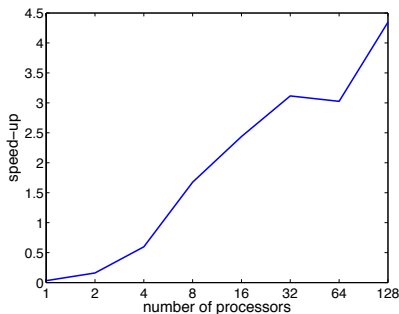
Performance of Parallel Algorithm 1 – Modeled

- **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 communication time is

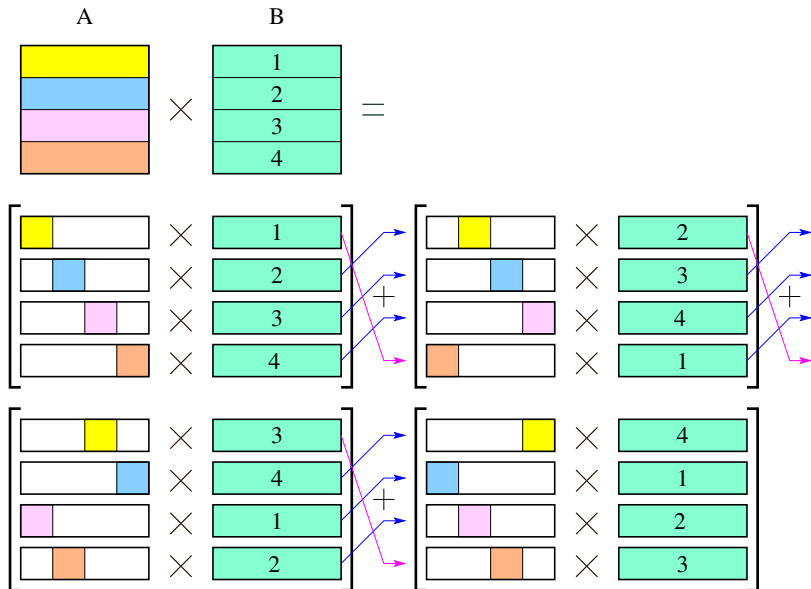
$$\begin{aligned}(P - 1) \left(t_0 + t_1 \frac{N^2}{P} \right) &= O(N^2 + P), \quad \text{but, beware of large constants} \\ &= O(N^2), \quad N^2 > P\end{aligned}$$

- **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.

Performance of Parallel Algorithm 1 – Measured



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_mult2(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, Accum2)
    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 $A(\text{MyIndex}, j)$ denote the j^{th} such block.
- `rotate(N, List) ->`
 `{L1, L2} = lists:split(N, List),`
 `L2 ++ L1.`
- The algorithm is based on the formula:

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

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: $O(N^2 + P)$.
 - ▶ 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, compute time grows as N^3/P , while communication time goes as $(N^2 + P)$.
- 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 won't need our matrices to be as huge to benefit from parallel computation.

Summary

- Matrix multiplication is well-suited for a parallel implementation.
- Need to consider communication costs.
- Connection of theory with actual run time is pretty good:
 - ▶ But the matrices have to be big enough to amortize the communication costs.
- In future lectures, may look at how to further reduce communication.

Preview

February 29: GPU Introduction

Reading: “The GPU Computing Era”, <http://dx.doi.org/10.1109/MM.2010.41>

March 2: GPU Programming

Reading: Programming Massively Parallel Processors, Chap. 3

List Comprehensions, practice examples

Use list comprehensions to implement Erlang functions for the following problems:

- `double(List) -> lists:map(fun(X) 2*X end, List).`

Replace `lists:map` with an Erlang expression that uses a comprehension.

- `divisible(K, List) -> lists:filter(fun(N) (N rem K) == 0 end, List).`

Returns the list of all elements of `List` that are divisible by `K`.

- `qsort([]) -> []; qsort(List1) -> List2.`

Use two list comprehensions, one to find the elements of `List1` that are less than or equal to `hd(List1)`, and another to find the elements that are greater than `hd(List1)`. Sort these two with recursive calls to `qsort` and concatenate the results using `++`.

List Comprehensions, one more practice problem

`pythag(ListX, ListY) -> ListP.`

`ListX` and `ListY` are lists of integers. `ListP` consists of all tuples $\{X, Y\}$ where Y is an element of `ListY`, and $\sqrt{X^2 + Y^2}$ is an integer. where X is an element of `ListX`, Y is an element of `ListY`, and $X \leq Y$ is an integer. Here's a function that tests whether or not an integer is a perfect square:

```
is_square(N, [Lo, Hi]) ->
  Mid = (Lo + Hi) div 2,
  MidSq = Mid*Mid,
  if
    (MidSq == N) -> true;
    (Lo >= Hi) -> false;
    (MidSq > N) -> is_square(N, [Lo, Mid]);
    (MidSq < N) -> is_square(N, [Mid+1, Hi])
  end.
```