Matrix Multiplication – Algorithms

Mark Greenstreet

CpSc 418 – Mar. 22, 2017

Outline:

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

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

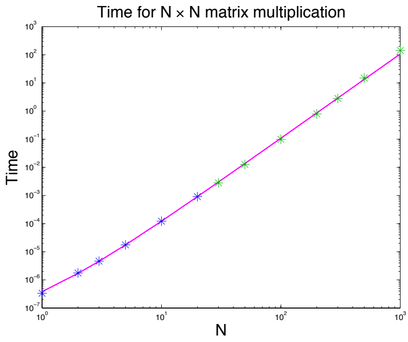
```
mult(A, B) ->
  BT = transpose(B),
  [ [ dot_prod(RowA, ColB) || ColB <- BT ] || RowA <- A].

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
  ].
```

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$*$#Multiplications.
  - Memory-writes: `n_rows_a` $*$ `n_cols_b`.
  - Time is $\mathcal{O}$(`n_rows_a` $*$ `n_cols_b` $*$ `n_cols_a`),
    If both matrices are $N \times N$, then its $\mathcal{O}(N^3)$.
- 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!
  - Cache miss dominates everything else.
- This is why there are carefully tuned numerical libraries.

Performance – Measured



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

Tiling Matrices

An Example

- Let A, B, and $C = AB$ be 16×16 matrices.
- Let $A1 = A[1:4, 1:16]$, i.e. the first four rows of A.
  - In our Erlang represntation, `[A1,_] = lists:split(4, A)`.
- Let $A2 = A[5:8, 1:16]$; $A3 = A[9:12, 1:16]$; $A4 = A[13:16, 1:16]$; and likewise for $C1$, $C2$, $C3$, and $C4$.
- Big important fact:

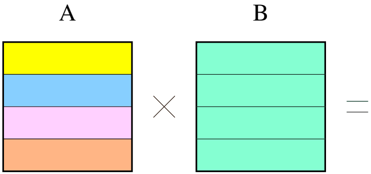
$$\begin{array}{rclcrcl} C1 & = & A1\,B & \quad & C2 & = & A2\,B \\ C3 & = & A3\,B & \quad & C4 & = & A4\,B \end{array}$$

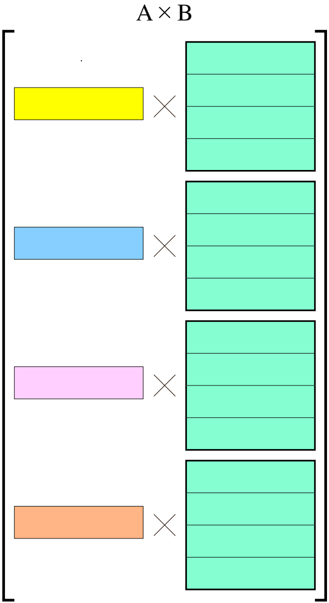
- In **sequential** Erlang:

```
[C1, C2, C3, C4] = [mult(AA, B) || AA <- A]
```

- To make it **parallel**, we compute each of the $C_I = A_I\,B$ with a separate process.

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
  ).
```

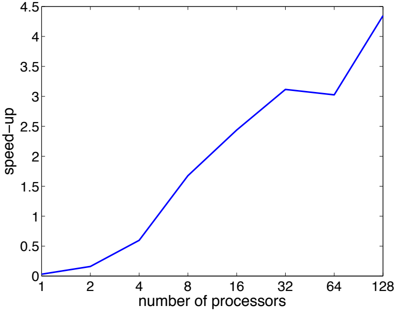
Performance of Parallel Algorithm 1 – Modeled

- CPU operations: same total number of multiplies and adds, but distributed around P processors. Total time: $\mathcal{O}(N^3/P)$.
- 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

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

  Note: I'm assuming t_0 corresponds to λ, and that t_1 is roughly the same as a the time for "typical" sequential operations..
- Memory: Each process needs $\mathcal{O}(N^2/P)$ storage for its block of A and the result. It also needs $\mathcal{O}(N^2)$ 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



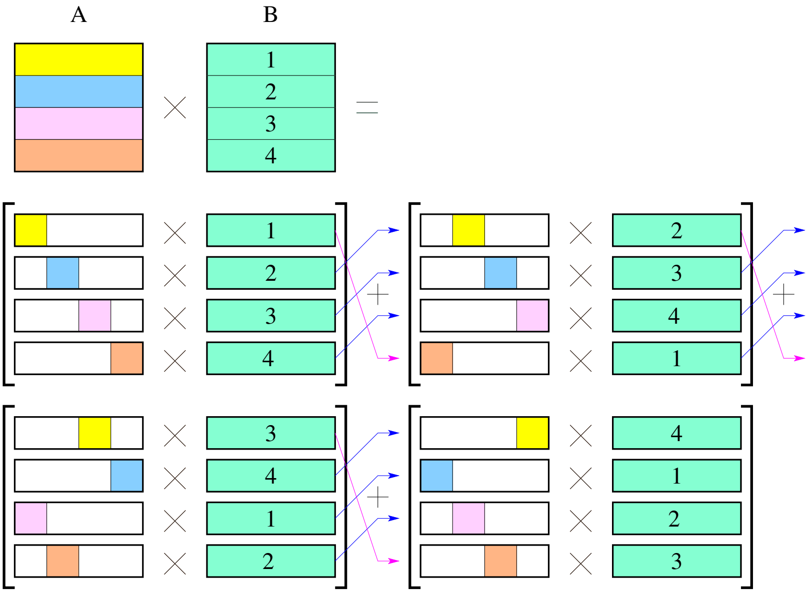
Using Memory more Efficiently

- Main idea: each process works on one "slab" of B at a time.

$$
\begin{array}{rcl}
C[i,j] & = & \sum_{k=1}^{N} A[i,k]\,B[k,j], \quad \text{a dot-product} \\
& = & \left(\sum_{k=1}^{N/4} A[i,k]\,B[k,j]\right) + \left(\sum_{k=(N/4)+1}^{N/2} A[i,k]\,B[k,j]\right) \\
& & + \left(\sum_{k=(N/2)+1}^{3N/4} A[i,k]\,B[k,j]\right) + \left(\sum_{k=(3N/4)+1}^{N} A[i,k]\,B[k,j]\right)
\end{array}
$$

- Each process does each of its four summations when it holds the corresponding slab of B.
  - Each holds one slap of A for the whole computation.
  - Each process only needs to hold one slab of B at at time.
- The algorithm generalizes to having any number of slabs for A and B in the obvious way.
  - Should be "obvious" if I've explained this clearly.
  - If it isn't obvious, that's my bad – please ask a question.

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.

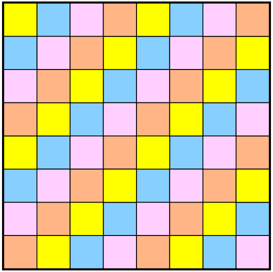
Performance of Parallel Algorithm 2

- CPU operations: Same as for parallel algorithm 1: total time: $\mathcal{O}(N^3/P)$.
- Communication: Same as for parallel algorithm 1: $\mathcal{O}(N^2 + P)$.
  - 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 $\mathcal{O}(N^2/P)$ 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.

Tiling in Real-Life



- Why? If there’s time, I’ll explain in class.

Summary

- Matrix multiplication is well-suited for a parallel implementation.
- Need to consider communication costs.
- 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.
- Connection of theory with actual run time is pretty good:
  - But the matrices have to be big enough to amortize the communication costs.

Preview

March 24: Matrix Multiplication in CUDA	
Homework:	HW4 due at 11:59pm
	HW5 goes out
March 27: Using Parallel Libraries	
March 29: Introduction to Model Checking	
Reading:	TBA
March 31: The PReach Model Checker	
Reading:	Industrial Strength ... Model Checking
April 3: Distributed Termination Detection	
April 5: Party: 50th Anniversary of Amdahl's Law	