ADVANCED PARALLEL COMPUTING

HomeWork - 4

PART 1) **DERIVATION OF ALGORITHMS:**

A) B is split in chunks of columns, A is in every node and computes a column chunk of C.

**Algorithm**:

- Initialize the MPI_communication context.
- Take Matrix A of size n x n and broadcast it to all process / send it individually to all other worker processes ( one process is master, rest are workers).
- The chunk per processor is calculated as column_per_node = n / p (p - no of processes)
- Since the arrays are stored in row major order, the Transpose of matrix B is taken, so its columns becomes rows.
- The rows chunk of the transposed matrix ( B- transpose) is sent to each worker node, so that each gets a unique chunk using MPI_Send.
- Now the other parameters are send to the worker processes using MPI_Send from the root node.
- The worker nodes waits for the Matrix data from the root node using MPI_Recv .
- Each worker node receives a complete matrix A and few rows of B transpose. The transpose is transposed again to get column of B and matrix multiplication is carried out between A & column chunk of B. The resultant chunk 'C' is obtained by this multiplication.
- The chunk 'C' is transposed to make it contiguous in memory and sent to the master/root process. From here the master waits for all workers to finish the computation and collects the data .
- The collected data will be C- Transpose which needs to be Transposed again to obtain C.
- MPI_Finalize ()

---

**B**) A is split in to chunks of rows and B is split in chunks of columns

**Algorithm:**.

- Initialize the MPI context.
- Initialize the Matrices A & B.
- Calculate the no. of rows per process and no. of columns per process.
- If P is the no of processes, and N x N is the size of matrices, then each process will be get N/sqrt (p-1) rows and columns.
- It is essential that the number of processor is of $n^2 + 1$ format, so that the essentially there are $n^2$ worker nodes. Hence if p is the no of processors, p-1 should be a perfect square and sqrt (p-1) must divide the no of rows and columns evenly without a remainder.

- Now, the master process splits the matrix A into chunks, and sends it to each worker process by using MPI_Send .
- The Transpose of B is then taken , and chunks of rows of B -Transpose is made and send to each worker nodes by MPI_Send.
- By this method, if each processor receives k rows and k columns, they can effectively compute k * k sub-matrix.
- The other scalars are sent along with this matrix for additional info.
- The worker process then computes one block of C from one row - block of A and column- block of B.
- Then the chunk is sent in order to root process, which receives it to make a final resultant matrix C.

---

1 C)

- A and B are split in blocks (both the rows and the columns are split in sqrt(p) groups). Each node gets one block of A, one block of B, and computes one block of C. Note that the data blocks need to be moved around as the computation progresses; node Pij needs the blocks Aik and Bkj at stage k of the computation.
- A process grid is then setup with cartesian topology.
- Make the initial alignment by Sending block $Ai,j$ to process ($i, j - i$ +sqrt( $p$) % sqrt($p$) and block $Bi,j$ to process ($i - j$ + sqrt($p$) %sqrt( $p$),$j$) for 0 to sqrt (p) -1;
- Multiply the received matrices to compute Cij.
- Then compute and shift, such that shift matrix A row by left by and matrix B column by upwards and magnitude of rotation depends on the row and column index.
- Compute C[i][j] = C[i][j] + A[i][k] +B[k][j]
- for step :=1 to $p - 1$

Shift $A$[i][j] one step left (with wraparound/ circular shift) and $B$[i][j] one step up (with wraparound/ with circular shift);

Compute C[i][j] in each case.

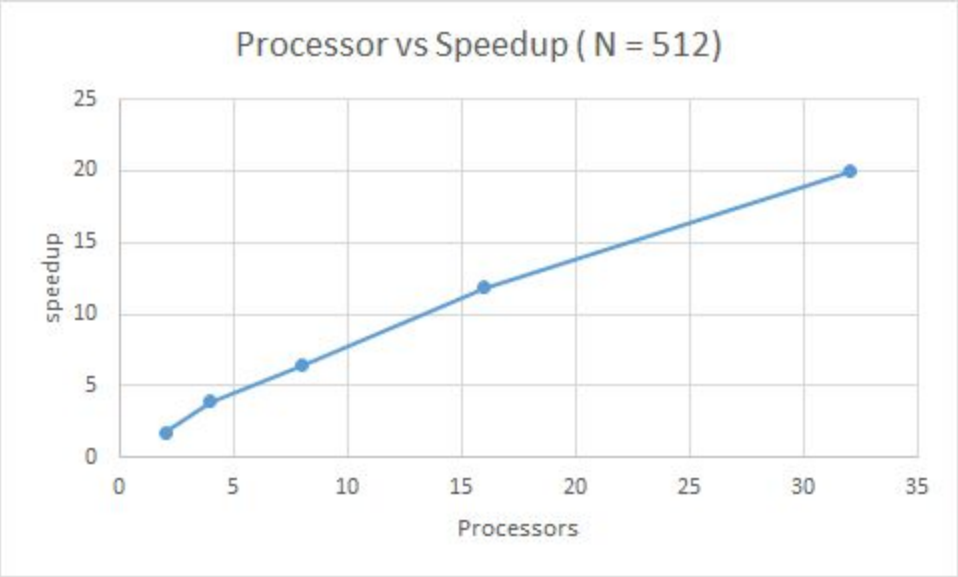
---

PART 2:

A) A is sent to every node and B is split into chunks of columns.

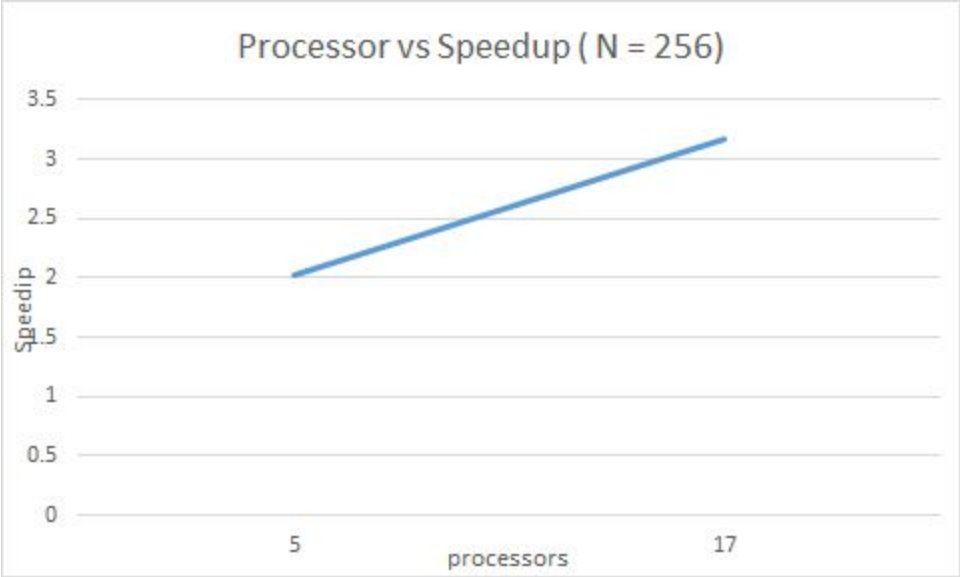
The implementation can be found in problem1.c

| N | Processors | Speedup |
|---|---|---|
| 256 | 4 | 3.21 |
| | 8 | 5.379 |
| | 16 | 9.864 |
| 512 | 4 | 3.42 |
| | 8 | 6.426 |
| | 16 | 11.86 |
| 1024 | 4 | 3.926 |
| | 8 | 7.936 |
| | 16 | 14.242 |



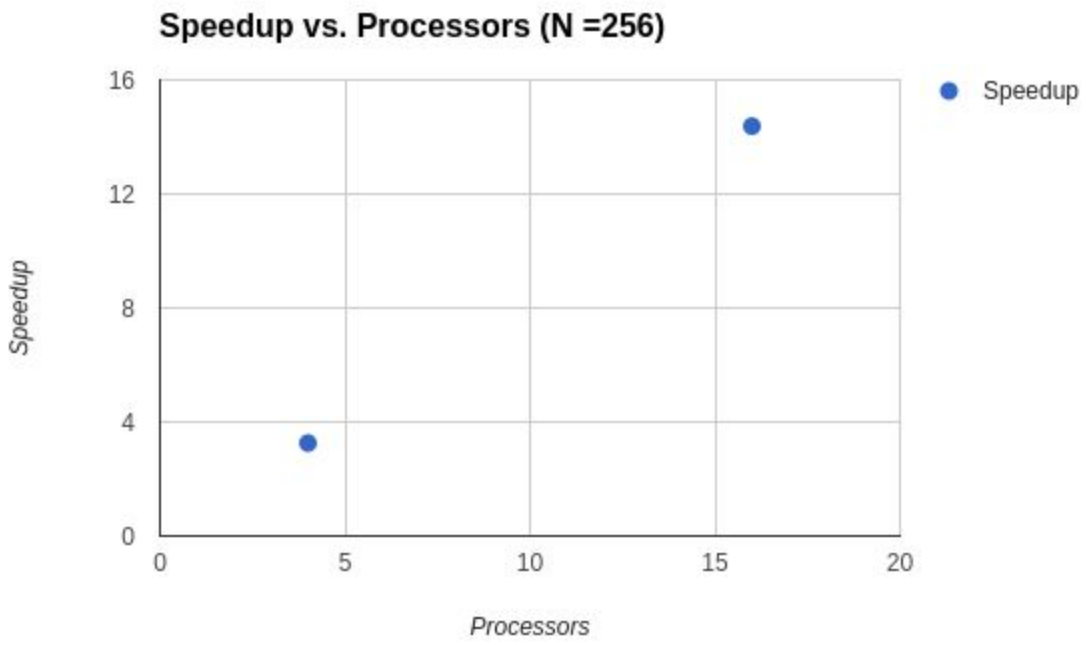

2)b) The implementation is available in prob2.c

| N | Processors | Speedup |
|---|---|---|
| 256 | 5 | 2.02 |
| | 17 | 3.17 |
| 512 | 5 | 3.738 |
| | 17 | 9.727 |
| 1024 | 5 | 3.933 |
| | 17 | 15.987 |




C) Canon's Algorithm:

| N | Processors | Speedup |
|---|---|---|
| 256 | 4 | 3.26 |
| | 16 | 14.38 |
| 512 | 4 | 3.98 |
| | 16 | 15.96 |




i) **Algorithm 1**:

Note : IN the below derivations O(n) is used to refer to theta (n).

I) **Iso-efficiency function**:

For Matrix multiplication:

Since for Matrix multiplication, of N x N, the serial algorithm takes O(n3) time. If each operation (addition/multiplication) is assumed to take tc time, then serial time:

T(serial) = $n^3 t_c$;

Assuming a hypercube network,

For Parallel time, each processor receives a complete matrix A (Broadcasted) and a column of matrix B. Also each processor performs multiplying n rows of A with n/p columns of B. SO total no of operations will be n * n * n/p = $n^3/p$.

The all to all broadcast can be performed in $2t_s \log p + 2t_w n^2$, where ts is the startup time and since n rows ( n* n elements) is transmitted .

T (parallel) = $t_c(n^3/p) + 2t_s \log p + 2t_w n^2$.

Speedup = $n^3 t_c$ / $t_c(n^3/p) + 2t_s \log p + 2t_w n^2$

=pT(serial)/T(serial) + T(overhead)

= p /( 1+ T(overhead)/T(serial))

= p / (1 + ($2t_s \log p + 2t_w n^2/n^3 t_c$ ))

Efficiency = 1/ (1 + ($t_s \log p + 2t_w n^2)/n^3 t_c$ ))

W = K * T(overhead)

W = K * ($2t_s$ plog $p + 2pt_w n^2$);

SO for analysis, taking first term,

W = n3 = $2t_s$ plog $p$ ;

By taking the second terms,

W = $n^3$ = K *$2pt_w n^2$

èn = (K *$2pt_w$ )

Since W = $n^3$ = (K *$2pt_w$ )$^3$

W = $K^3 * 2^3 p^3 t_w^3$ .

Hence Iso-efficiency => W = O($p^3$).

ii) **Scaling function for memory**:

For a nxn matrix multiplication, we need n2 memory spaces.

Each processor will need n2/p memory locations .

For the memory per node to be constant $n^2/p = c$;

Hence $n^2 = pc$.

For Memory requirement to be constant, it should scale as function of $n^2$.

Hence W(p) = O($n^2$).

iii) **Scaling function such that parallel time is constant**:

T(parallel) = $t_c(n^3/p) + 2t_s \log p + 2t_w n^2$

Speedup = $n^3 t_c$ / $t_c(n^3/p) + 2t_s \log p + 2t_w n^2$

Total mem requirement = O($n^2$).

Hence n2 = c*p.

Speedup' = $(cp)^{1.5}$ / ($t_c(cp^{1.5}/p) + 2t_s \log p + 2t_w cp$) = *o(p)*

For time –constrained scaling the speedup Tp = O(n3/p);
Ie $n^3 = cp$.
Hence based on the above results to maintain iso-efficiency, p has to increase linearly with n , but the memory requirement increases quadratically with increase in N and to maintain the constant parallel time the processor count has to increase as a function of $n^3$ . Hence its is evident that all three factors will not scale uniformly with increase in n.

2) **Algorithm 2:**
Analysis:
If the matrix is distributed by rows and columns across $p^2$ processors.
Then, each processor gets n/sqrt(p) rows and columns.
$T(serial) = n^3 t_c$ .
Assuming a hypercube network,
Since n/sqrt(p) rows and columns gets to each processor,
$T(parallel) = t_c(n3/(p^2)) + t_s logp^2 + 2t_w n^2/sqrt(p)$
$T(overhead) = p^2 T(parallel) – T(serial) = pt_s logp^2 + 2p^{3/2} t_w n^2$ .
**For ISO-efficiency**:
W = K T(overhead)
Since $W = n^3$
$=>n^3 = K * (pt_s logp + 2p^{3/2} t_w n^2)$
Taking the $n^2$ term,
$=>n^3 = K * (2p^{3/2} t_w(n^2))$;
$=>n = K * (2p^{3/2} t_w)$
$W = n3 = K * (2p^{9/2} t_w)$.
$W = O(p^{9/2})$.

ii) **Scaling function for memory**:
The total memory requirement of the algorithm is O(n2).
$T(parallel) = t_c(n3/(p)) + t_s logp + 2t_w n^2 * sqrt(p)$.
Each processor needs O(n2/p) memory.
For memory constrained, scaling the memory of the system grows linearly with no of processors.
Memory = O(p);
Since Memory, $m = O(n^2)$;
=>we will have $n^2 = cp$ for some constant c.
iii) **Scaling function for parallel time**:
$T(parallel) = t_c(n3/(p^2)) + t_s logp^2 + 2t_w n^2/sqrt(p)$.
Since there are p2 processors:
Let $p^2 = k$;
To have a constant parallel time:

$=>n^3 = cp^2$

$=> n = cp^{2/3}$

$=>n = ck^{1/3}$

Hence based on the above results each requirement scales differently with increase in n.

3) **Algorithm 3**:

i) Analysis:

For a n x n matrix,

$T(serial) = n^3 t_c$ .

As it is a matrix multiplication, it will take $n^3$ time.

If there are 'p ' processors, Each processor will get n/sqrt(p) rows and n/sqrt(p) columns.

Then will be divided and circular shifted for matrix computation.

Assuming a hyper cube:

$T(parallel) = n^3 t_c/p + t_s + t_w n^2$ .

$T(overhead) = p\ T(parallel) – T(serial)$

$= p *(t_s + t_w n^2 )$

$W = K * T(overhead)$

$= K * t_s p + t_w n^2 p.$

$W = n^3$

By taking the second term for iso efficiency:

$W = n^3 = K *t_w n^2 p$

$=>n = K *t_w p;$

As $W = n^3 = (K *t_w p)^3$

The **iso-efficiency** would be $O(p^3)$

ii) **Memory scaling**:

Since the each processor ends up computing resultant data of size n2/p.

Hence each node needs $n^2/p$ memory size.

Hence with memory constraint,

As m = O(p) ( memory increasing linearly with processor count).

Hence the memory should increase as $m =>p = cn^2$ , where C is some constant.

Iii) **Parallel time constraint**:

$T(parallel) = n^3 t_c/p + t_s + t_w n^2$ .

By taking the highest order term,

$T(parallel) = n^3 t_c/p;$

Hence the for a constant T(parallel) , p should increase proportional with $n^3$ .

i.e $p = cn^3$.

Based on the above results, the iso efficiency is achieved if processor count increases by $P^3$ with increase in n and the memory requirement increases by $n^2$ with increase in p to match the constraint and to keep constant parallel time the processor count has to match $cn^3 = p$ relation.

---