Parallel programming using MPI

Analysis and optimization

Bhupender Thakur, Jim Lupo, Le Yan, Alex Pacheco
Outline

- Parallel programming:  
  *Basic definitions*

- Choosing right algorithms:  
  *Optimal serial and parallel*

- Load Balancing  
  *Rank ordering, Domain decomposition*

- Blocking vs Non blocking  
  *Overlap computation and communication*

- MPI-IO and avoiding I/O bottlenecks

- Hybrid Programming model  
  *MPI + OpenMP*  
  *MPI + Accelerators for GPU clusters*
## Choosing right algorithms:
*How does your serial algorithm scale?*

<table>
<thead>
<tr>
<th>Notation</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant</td>
<td>Determining if a number is even or odd; using a hash table</td>
</tr>
<tr>
<td>$O(\log\log n)$</td>
<td>double log</td>
<td>Finding an item using interpolation search in a sorted array</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>log</td>
<td>Finding an item in a sorted array with a binary search</td>
</tr>
<tr>
<td>$O(n^c)$, $c&lt;1$</td>
<td>fractional power</td>
<td>Searching in a kd-tree</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>linear</td>
<td>Finding an item in an unsorted list or in an unsorted array</td>
</tr>
<tr>
<td>$O(n\log n)$</td>
<td>loglinear</td>
<td>Performing a Fast Fourier transform; heapsort, quicksort</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>quadratic</td>
<td>Naïve bubble sort</td>
</tr>
<tr>
<td>$O(n^c)$, $c&gt;1$</td>
<td>polynomial</td>
<td>Matrix multiplication, inversion</td>
</tr>
<tr>
<td>$O(c^n)$, $c&gt;1$</td>
<td>exponential</td>
<td>Finding the (exact) solution to the travelling salesman problem</td>
</tr>
<tr>
<td>$O(n!)$</td>
<td>factorial</td>
<td>generating all unrestricted permutations of a poset</td>
</tr>
</tbody>
</table>
Parallel programming concepts:
Basic definitions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>problem size</td>
</tr>
<tr>
<td>p</td>
<td>number of processors</td>
</tr>
<tr>
<td>m</td>
<td>number of memory cells</td>
</tr>
<tr>
<td>T</td>
<td>number of steps for sequential algorithm</td>
</tr>
<tr>
<td>t</td>
<td>number of steps for parallel algorithm</td>
</tr>
<tr>
<td>w</td>
<td>work done by the parallel algorithm</td>
</tr>
<tr>
<td>S</td>
<td>speedup</td>
</tr>
<tr>
<td>E</td>
<td>efficiency</td>
</tr>
<tr>
<td>$t_s$</td>
<td>execution time for sequential algorithm</td>
</tr>
<tr>
<td>$t_p$</td>
<td>execution time for parallel algorithm</td>
</tr>
</tbody>
</table>
Parallel programming concepts:

**Performance metrics**

**Speedup:**
*Ratio of parallel to serial execution time*

\[ S = \frac{t_s}{t_p} \]

**Efficiency:**
*The ratio of speedup to the number of processors*

\[ E = \frac{S}{p} \]

**Work**
*Product of parallel time and processors*

\[ W = tp \]

**Parallel overhead**
*Idle time wasted in parallel execution*

\[ T_0 = pt_p - t_s \]
Parallel programming concepts
Analyzing serial vs parallel algorithms

Ask yourself

What fraction of the code can you completely parallelize? 
\( f \)?

How does problem size scale? 
Processors scale as \( p \). How does problem-size \( n \) scale with \( p \)? 
\( n(p) \)?

How does parallel overhead grow? 
\( T_o(p) \)?

Does the problem scale? 
\( M(p) \)
Parallel programming concepts
Analyzing serial vs parallel algorithms

Amdahl’s law

What fraction of the code can you completely parallelize?

\[ f \]

Serial time: \( t_s \)

Parallel time: \( ft_s + \frac{(1-f)}{p} t_s \)

\[ S = \frac{t_s}{t_p} = \frac{t_s}{t_s f + \frac{(1-f)}{p} t_s} \]

\[ = \frac{1}{f + \frac{(1-f)}{p}} \]
Parallel programming concepts

Analyzing serial vs parallel algorithms

Quiz

if the serial fraction is 5%, what is the maximum speedup you can achieve?

Serial time = 100 secs
Serial percentage = 5%

Maximum speedup?
Parallel programming concepts
Analyzing serial vs parallel algorithms

Amdahl’s law

What fraction of the code can you completely parallelize? $f$?

Serial time = 100 secs
Parallel percentage = 5 %

$$f = \frac{1}{0.05 + \frac{(1-f)}{p} + 0} = 20$$
Parallel programming concepts
Analyzing serial vs parallel algorithms

Amdahl's Law approximately suggests:
“Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph. No matter how fast you drive the last half, it is impossible to achieve 90 mph average before reaching the second city.”

Gustafson's Law approximately states:
“Suppose a car has already been traveling for some time at less than 90mph. Given enough time and distance to travel, the car's average speed can always eventually reach 90mph, no matter how long or how slowly it has already traveled.”

Source: http://disney.go.com/cars/
http://en.wikipedia.org/wiki/Gustafson's_law
Parallel programming concepts
Analyzing serial vs parallel algorithms

Communication Overhead

Simplest model:

Transfer time
  = Startup time
  + Hop time (Node latency)
  + (Message length)/Bandwidth

= $t_s + t_h + \frac{t_w}{l}$

Send one big message instead of several small messages!
Reduce the total amount of bytes!
Bandwidth depends on protocol
Parallel programming concepts
Analyzing serial vs parallel algorithms

Point to point (MPI_Send)
\( (t_s + t_w m) \)

Collective overhead

All-to-all Broadcast (MPI Allgather):
\( t_s \log_2 p + (p-1) t_w m \)

All-reduce (MPI Allreduce):
\( (t_s + t_w m) \log_2 p \)

Scatter and Gather (MPI Scatter):
\( (t_s + t_w m) \log_2 p \)

All to all (personalized):
\( (p-1)(t_s + t_w m) \)
Isoefficiency:

- Can we maintain efficiency/speedup of the algorithm?
- How should the problem size scale with p to keep efficiency constant?

\[ E = \frac{1}{1 + \left( \frac{T_o}{w} \right)} \]

*Maintain ratio* $T_o(W,p) / W$, *overhead to parallel work constant*
Isoefficiency relation: To keep efficiency constant you must increase problem size such that

\[ T(n,1) \geq T_o(n,p) \]

Procedure:
1. Get the sequential time \( T(n,1) \)
2. Get the parallel time \( pT(n,p) \)
3. Calculate the overhead \( T_o = pT(n,p) - T(n,1) \)

How does the overhead compare to the useful work being done?
Isoefficiency relation: To keep efficiency constant you must increase problem size such that

\[ T(n,1) \geq T_o(n,p) \]

Scalability: Do you have enough resources (memory) to scale to that size

Maintain ratio \( T_o(W,p) / W \), overhead to parallel work constant
Scalability: Do you have enough resources (memory) to scale to that size.

Parallel programming:
Basic definitions

Number of processors
Memory needed per processor

Memory Size

- Cannot maintain efficiency
- Can maintain efficiency

$C_p\log p$
$C_p$

$C$

$C_p$ log $p$
Adding numbers

Each processor has $n/p$ numbers.

Serial time: $n$
Parallel time: $n/p$
Communicate and add: $\log p + \log p$
Adding numbers

Each processor has $n/p$ numbers.
Steps to communicate and add are $2 \log p$

**Speedup:**

$$S = \frac{n}{\left(\frac{n}{p} + 2 \log p\right)}$$

**Isoefficiency**

If you increase problem size $n$ as $O(p \log p)$
then efficiency can remain constant!
Adding numbers

Each processor has \( n/p \) numbers.
Steps to communicate and add are \( 2 \log p \)

**Speedup:**

\[
E = \frac{n}{(n + 2p \log p)}
\]

**Scalability**

\[
M(n) \geq \left(\frac{n}{p}\right) = \frac{p \log p}{p} = \log p
\]
Sorting

Each processor has $\frac{n}{p}$ numbers.

Our plan

1. Split list into parts
2. Sort parts individually
3. Merge lists to get sorted list
Sorting

Each processor has \( n/p \) numbers.

- 4, 3, 9
- 1, 3, 7
- 7, 5, 8
- 5, 2, 9

Background

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Stability</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble sort</td>
<td>( O(n^2) )</td>
<td>Stable</td>
<td>Exchanging</td>
</tr>
<tr>
<td>Selection sort</td>
<td>( O(n^2) )</td>
<td>Unstable</td>
<td>Selection</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>( O(n^2) )</td>
<td>Stable</td>
<td>Insertion</td>
</tr>
<tr>
<td>Merge sort</td>
<td>( O(n \log n) )</td>
<td>Stable</td>
<td>Merging</td>
</tr>
<tr>
<td>Quick sort</td>
<td>( O(n \log n) )</td>
<td>Unstable</td>
<td>Partitioning</td>
</tr>
</tbody>
</table>
Choosing right algorithms:
Optimal serial and parallel

Case Study: Bubble sort

Main loop
   For i : 1 to length_of(A) -1

Secondary loop
   For j : i+1 to length_of(A)

Compare and swap
so smaller element is to left

[ 5 1 4 2 ]
Choosing right algorithms:
Optimal serial and parallel

Case Study: Bubble sort

Main loop
For i : 1 to length_of(A) -1

Secondary loop
For j : i+1 to length_of(A)

Compare and swap
so smaller element is to left
Choosing right algorithms:
Optimal serial and parallel

Case Study: Bubble sort

Main loop
For i : 1 to length_of(A) -1

Secondary loop
For j : i+1 to length_of(A)

Compare and swap
so smaller element is to left

N(N-1)/2 = O(N^2)
Comparisons
Choosing right algorithms:
Optimal serial and parallel

Case Study: Merge sort

Recursively merge lists having one element each

\[ [5 \ 1 \ 4 \ 2] \]
\[ [1 \ 5] \ [4 \ 2] \]
\[ [1 \ 5][2 \ 4] \]
\[ [1 \ 2 \ 4 \ 5] \]
Choosing right algorithms:
Optimal serial and parallel

Case Study: Merge sort

Recursively merge lists having one element each

Best sorting algorithms need $O(N \log N)$
Choosing right algorithms:
Parallel sorting technique

Merging: Merge $p$ lists having $n/p$ elements each

Sub-optimally:
Pop-push merge on 1 processor

$$O(np)$$

\[
\begin{align*}
[1 & 3 & 5 & 6] & \Rightarrow [1] \\
[2 & 4 & 6 & 8] & \\
[3 & 5 & 6] & \Rightarrow [1 & 2] \\
[2 & 4 & 6 & 8] & \\
[4 & 6 & 8] & 
\end{align*}
\]
Choosing right algorithms:
*Parallel sorting technique*

Merging: Merge p lists having n/p elements each

Optimal: Recursively or tree based

Best merge algorithms need

\[ O(n \log p) \]
Sorting

Each processor has $n/p$ numbers.

<table>
<thead>
<tr>
<th>Serial time</th>
<th>Parallel time</th>
<th>Merge time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n \log n$</td>
<td>$\frac{n}{p} \log \left( \frac{n}{p} \right)$</td>
<td>$(n-1) \cdot \frac{n}{p}$ $\frac{n}{p} + n \log p$</td>
</tr>
</tbody>
</table>

4, 3, 9  
1, 3, 7  
7, 5, 8  
5, 2, 9
Sorting

Each processor has \( n/p \) numbers.

Overhead
\[
= n \log n - p \left( \frac{n}{p} \log \frac{n}{p} + n \log p \right) \\
\approx p \log p + np \log p \\
\approx n \log p
\]

Isoefficiency
\[
n \log n \geq c(n \log p) \\
\Rightarrow n \geq p^c
\]

Scalability
\[
= n / p = p^{c-1} \\
\text{Low for } c > 2
\]
### Data Decomposition

1D: Row-wise

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Data decomposition

1D: Column-wise
Data decomposition

2D: Block-wise
Data decomposition

Laplace solver: \((n \times n)\) mesh with \(p\) processors

Time to communicate 1 cell:

\[
t_{\text{cell}}^{\text{comm}} = \tau_s + t_w
\]

Time to evaluate stencil once:

\[
t_{\text{cell}}^{\text{comp}} = 5 \times (t_{\text{float}})
\]
Data decomposition

Laplace solver: 1D Row-wise (n x n) with p processors

\[ a(i+1, j) \]
\[ a(i, j+1) \]
\[ a(i, j+1) \]
\[ a(i-1, j) \]
Data decomposition

Laplace solver: 1D Row-wise \((n \times n)\) with \(p\) processors

Serial time:
\[
t_{\text{seq}}^{\text{comp}} = n^2 t_{\text{cell}}^{\text{comm}}
\]

Parallel computation time
\[
t_{\text{process}}^{\text{comp}} = \frac{n^2}{p} t_{\text{cell}}^{\text{comp}}
\]

Ghost communication:
\[
t^{\text{comm}} = 2nt_{\text{cell}}^{\text{comm}}
\]

\(a(i+1, j)\)
\(a(i, j+1)\)
\(a(i-1, j)\)
Data decomposition

Laplace solver: 1D Row-wise (n x n) with p processors

Overhead:
\[ \text{Overhead} = t_{seq} - pt_p \]
\[ = pn \]

Isoefficiency:
\[ n^2 \geq cnp \Rightarrow n \geq cp \]

Scalability:
\[ = c^2 p^2 / p = Cp \]

Poor Scaling
Data decomposition

Laplace solver: 2D Block-wise
Data decomposition

Laplace solver: 2D Row-wise (n x n) with p processors

Serial time:

\[ t_{seq}^{comp} = n^2 t_{cell}^{comm} \]

Parallel computation time:

\[ t_{process}^{comp} = \frac{n^2}{p} t_{cell}^{comp} \]

Ghost communication:

\[ t_{comm}^{comp} = \frac{4n}{\sqrt{p}} t_{cell}^{comm} \]
Data decomposition

Laplace solver: 2D Row-wise (n x n) with p processors

Overhead:
\[ = p \frac{n}{\sqrt{p}} \]
\[ = n\sqrt{p} \]

Isoefficiency:
\[ n \sim \sqrt{p} \]

Scalability:
\[ = \frac{(c\sqrt{p})^2}{p} \]
\[ = C \quad \text{Perfect Scaling} \]
Data decomposition

Matrix vector multiplication: 1D row-wise decomposition

**Computation:**
Each processor computes $n/p$ elements, $n$ multiplies + $(n-1)$ adds for each

$$O\left(\frac{n^2}{p}\right)$$

**Communication:**
All gather in the end so each processor has full copy of output vector

$$\log p + \sum_{i=1}^{\log p} \frac{n}{2^i} \cdot \frac{n}{p} = \log p + \frac{n(p-1)}{p}$$
Data decomposition

Matrix vector multiplication: 1D row-wise decomposition

Algorithm:
1. Collect vector using MPI_Allgather
2. Local matrix multiplication to get output vector

Wastes much memory
Data decomposition

Matrix vector multiplication: 1D row-wise decomposition

**Computation:**
Each processor computes \(n/p\) elements, \(n\) multiplies + (n-1) adds for each

\[
O\left(\frac{n^2}{p}\right)
\]

**Communication:**
All gather in the end so each processor has full copy of output vector

\[\tau_w n + \tau_s \log p\]

**Overhead:**

\[\tau_s p \log p + \tau_w np\]
### Data decomposition

Matrix vector multiplication: 1D row-wise decomposition

**Speedup:**

\[
S = \frac{p}{1 + \left( \frac{p(\tau_s \log p + t_w n)}{t_c n^2} \right)}
\]

**Isoefficiency:**

\[n^2 \sim p \log p + np\]

\[\Rightarrow n \geq cp\]

**Scalability:**

\[M(p) \geq \frac{n^2}{p} = c^2 p\]

Not scalable!
Data decomposition

Matrix vector multiplication: 1D column-wise decomposition

Serial Computation?

Parallel Computation?

Overhead?

Isoefficiency?

Scalability?
Data decomposition

Matrix vector multiplication: 2D decomposition

Algorithm:
Uses p Processors on a grid

\[ \begin{align*}
A_{00} \text{ proc 0} & & A_{01} \text{ proc 1} & & A_{02} \text{ proc 2} \\
A_{10} \text{ proc 3} & & A_{11} \text{ proc 4} & & A_{12} \text{ proc 5} \\
A_{20} \text{ proc 6} & & A_{21} \text{ proc 7} & & A_{22} \text{ proc 8} \\
\end{align*} \]
Data decomposition

Matrix vector multiplication: 2D decomposition

Algorithm Step 0: Copy part-vector to diagonal
Data decomposition

Matrix vector multiplication: 2D decomposition

Algorithm Step 1: Broadcast vector along columns
Data decomposition

Matrix vector multiplication: 2D decomposition

Algorithm Step 2: Local computation on each processor
Data decomposition

Matrix vector multiplication: 2D decomposition

Algorithm Step 3: Reduce across rows
Data decomposition

Matrix vector multiplication: 2D decomposition

Computation:
Each processor computes $\frac{n^2}{p}$ elements,

$$\tau_c \left( \frac{n^2}{p} \right)$$

Communication:
$$\frac{2n}{\sqrt{p}} \log(\sqrt{p}) + \frac{n}{\sqrt{p}} \sim \frac{n \log p}{\sqrt{p}}$$

Overhead:
$$n \sqrt{p} \log(p)$$
Data decomposition

Matrix vector multiplication: 2D decomposition

Isoefficiency:
\[ n^2 \sim n \sqrt{p \log p} \]
\[ \Rightarrow n \geq c \sqrt{p \log p} \]

Scalability:
\[ M(p) \geq \frac{n^2}{p} = (\log p)^2 \]

Scales better than 1D!
Let's look at the code
Summary

◆ Know your algorithm!

◆ Don’t expect the unexpected!

◆ Pay attention to parallel design and implementation right from the outset. It will save you lot of labor.