Network and Load-Aware Resource Manager for MPI Programs

Ashish Kumar  Naman Jain  Preeti Malakar

Indian Institute of Technology, Kanpur

SRMPDS, International Conference on Parallel Processing 2020
Distributed-memory parallel programs and MPI

- More than one processing element using their own local memory.
- Nodes work cooperatively to solve a single big problem.
- Data exchange through communications by sending and receiving messages.
Distributed-memory parallel programs and MPI

- More than one processing element using their own local memory.
- Nodes work cooperatively to solve a single big problem.
- Data exchange through communications by sending and receiving messages.
- Uses Message Passing Interface (MPI) as "de facto" standard for message passing.
Distributed-memory parallel programs and MPI

- More than one processing element using their own local memory.
- Nodes work cooperatively to solve a single big problem.
- Data exchange through communications by sending and receiving messages.
- Uses Message Passing Interface (MPI) as "de facto" standard for message passing.
- Runs on cluster (shared or dedicated) or a supercomputer.
Introduction

- Need for node allocation to run MPI jobs.
Introduction

- Need for node allocation to run MPI jobs.
- In this work, we address the problem of allocating a good set of nodes to run MPI jobs in a shared non-dedicated cluster.
Non Dedicated/Shared Cluster and challenges

- Non exclusive access of nodes
Non Dedicated/Shared Cluster and challenges

- Non exclusive access of nodes
- Shared among many users, same node can be used by different users/processes at same time for different purposes.
Non Dedicated/Shared Cluster and challenges

- Non exclusive access of nodes
- Shared among many users, same node can be used by different users/processes at same time for different purposes.
- Variation in resource uses across time/nodes
Non Dedicated/Shared Cluster and challenges

- Non exclusive access of nodes
- Shared among many users, same node can be used by different users/processes at same time for different purposes.
- Variation in resource uses across time/nodes
- Which nodes to run our job on? What parameters should be considered?
Node Resource Usage Variation

- Variations in node resource usage across time and node in shared cluster

![CPU Load](image)

![CPU Utilization and Memory Usage](image)
Variation in network usage between nodes in shared cluster

- Network Usage Variation

Variation in network usage between nodes in shared cluster
Going towards our approach

- Use knowledge of these variations across nodes, time and network to allocate resources in a better way.
Going towards our approach

- Use knowledge of these variations across nodes, time and network to allocate resources in a better way.
- Take into account both static and dynamic attributes of resources, including network availability.
Overview

1. Node Allocation Algorithm
2. Resource Monitoring
3. Experiments
4. Conclusions and Future Work
Allocation as Sub Graph Selection

\[ G = (V, E) \]

Vertex \( v \in V \): compute node having compute load \( CL_v \) and available processor count \( pc_v \)

Edge \( e \in E \): network load \( NL(u,v) \) between compute nodes.

\( n \): number of processes to be allocated

Find a sub-graph such that the overall cost/load of the sub-graph is minimized and process demand is fulfilled.
Allocation as Sub Graph Selection

\[ G = (\mathcal{V}, \mathcal{E}) \]

- Vertex \( v \in \mathcal{V} \): compute node having compute load \( CL_v \) and available processor count \( pc_v \)

<table>
<thead>
<tr>
<th>Node</th>
<th>Compute load</th>
<th>#Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>50.2</td>
<td>6</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>43.5</td>
<td>8</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>54.7</td>
<td>10</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>38.3</td>
<td>4</td>
</tr>
</tbody>
</table>
Allocation as Sub Graph Selection

\[ G = (V, E) \]

- Vertex \( v \in V \): compute node having compute load \( CL_v \) and available processor count \( pc_v \)
- Edge \( e \in E \): network load \( NL_{(u,v)} \) between compute nodes.

<table>
<thead>
<tr>
<th>Node</th>
<th>Compute load</th>
<th>#Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>50.2</td>
<td>6</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>43.5</td>
<td>8</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>54.7</td>
<td>10</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>38.3</td>
<td>4</td>
</tr>
</tbody>
</table>
Allocation as Sub Graph Selection

\[ G = (V, E) \]

- Vertex \( v \in V \): compute node having compute load \( CL_v \) and available processor count \( pc_v \)
- Edge \( e \in E \): network load \( NL(u,v) \) between compute nodes.
- \( n \): number of processes to be allocated

Find a sub-graph such that the overall cost/load of the sub-graph is minimized and process demand is fulfilled.

<table>
<thead>
<tr>
<th>Node</th>
<th>Compute load</th>
<th>#Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>50.2</td>
<td>6</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>43.5</td>
<td>8</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>54.7</td>
<td>10</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>38.3</td>
<td>4</td>
</tr>
</tbody>
</table>
Some Definitions

- **Compute load**: measure of overall load on the node
  - **Static attributes**: clock speed, core count, total memory.
  - **Dynamic attributes**: CPU load, CPU utilization, available memory.
  - Compute load, $CL_v = \sum_{a \in \text{attributes}} w_a \cdot val_{va}$

- **Network load**: measure of load on the p2p network link
  - Latency
  - Bandwidth
  - Network load, $NL(u, v) = w_{lt} \cdot LT(u, v) + w_{bw} \cdot BW(u, v)$

- **Available processors**: measure of effective number of processors
  - $pc_v = coreCount_v - Load_v \% coreCount_v$

Weights can be tuned according to program need/type.
Some Definitions

- **Compute load**: measure of overall load on the node
  - Static attributes: clock speed, core count, total memory.
  - Dynamic attributes: CPU load, CPU utilization, available memory
  - Compute load, \( CL_v = \sum_{a \in \text{attributes}} w_a \times val_{va} \)

- **Network load**: measure of load on the p2p network link
  - Latency
  - Bandwidth
  - Network load, \( NL_{(u,v)} = w_{lt} \cdot LT_{(u,v)} + w_{bw} \cdot BW_{(u,v)} \)
Some Definitions

- **Compute load**: measure of overall load on the node
  - Static attributes: clock speed, core count, total memory.
  - Dynamic attributes: CPU load, CPU utilization, available memory
  - Compute load, $CL_v = \sum_{a \in \text{attributes}} w_a \ast val_{va}$

- **Network load**: measure of load on the p2p network link
  - Latency
  - Bandwidth
  - Network load, $NL_{(u,v)} = w_{lt} LT_{(u,v)} + w_{bw} BW_{(u,v)}$

- **Available processors**: measure of effective number of processors
  - $pc_v = coreCount_v - Load_v \% coreCount_v$
Some Definitions

- **Compute load**: measure of overall load on the node
  - Static attributes: clock speed, core count, total memory.
  - Dynamic attributes: CPU load, CPU utilization, available memory
  - Compute load, $CL_v = \sum_{a \in \text{attributes}} w_a \times val_{va}$

- **Network load**: measure of load on the p2p network link
  - Latency
  - Bandwidth
  - Network load, $NL_{(u,v)} = w_{lt}LT_{(u,v)} + w_{bw}BW_{(u,v)}$

- **Available processors**: measure of effective number of processors
  - $pc_v = coreCount_v - Load_v \% coreCount_v$

- Weights can be tuned according to program need/type.
Allocation Algorithm

- Find candidate sub-graph corresponding to each node.
Allocation Algorithm

- Find candidate sub-graph corresponding to each node.
- For each sub-graph $G_v = (V_v, E_v)$ define:
  - Compute Load, $C_{G_v} = \sum_{u \in V_v} CL_u$
  - Network Load, $N_{G_v} = \sum_{(x,y) \in E_v} NL(x,y)$

Allocate the best one on the basis of total load.
Allocation Algorithm

- Find candidate sub-graph corresponding to each node.
- For each sub-graph $G_v = (V_v, E_v)$ define:
  - Compute Load, $C_{G_v} = \sum_{u \in V_v} CL_u$
  - Network Load, $N_{G_v} = \sum_{(x,y) \in E_v} NL(x,y)$
  - Total Load, $T_{G_v} = \alpha \times C_{G_v, \text{Normalized}} + \beta \times N_{G_v, \text{Normalized}}$
Allocation Algorithm

- Find candidate sub-graph corresponding to each node.
- For each sub-graph $G_v = (V_v, E_v)$ define:
  - Compute Load, $C_{G_v} = \sum_{u \in V_v} CL_u$
  - Network Load, $N_{G_v} = \sum_{(x,y) \in E_v} NL(x,y)$
  - Total Load, $T_{G_v} = \alpha \times C_{G_v \text{ Normalized}} + \beta \times N_{G_v \text{ Normalized}}$
- Allocate the best one on the basis of total load
**Allocation Algorithm**

- Available core count of $v_4 = 4$
- Required process count = 16
- Compute weight ($\alpha$) = 0.4
- Network weight ($\beta$) = 0.6
- Picked $v_4$
- Allocated process count = 4

<table>
<thead>
<tr>
<th>Node</th>
<th>Compute load</th>
<th>Network load</th>
<th>#Available cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>52</td>
<td>45</td>
<td>6</td>
</tr>
<tr>
<td>$v_2$</td>
<td>47</td>
<td>70</td>
<td>8</td>
</tr>
<tr>
<td>$v_3$</td>
<td>74</td>
<td>65</td>
<td>5</td>
</tr>
</tbody>
</table>
Allocation Algorithm

- Available core count of $v_4 = 4$
- Required process count = 16
- Compute weight ($\alpha$) = 0.4
- Network weight ($\beta$) = 0.6
- Picked $v_4$
- Allocated process count = 4

<table>
<thead>
<tr>
<th>Node</th>
<th>Compute load</th>
<th>Network load</th>
<th>Addition load</th>
<th>#Available cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>52</td>
<td>45</td>
<td>47.8</td>
<td>6</td>
</tr>
<tr>
<td>$v_2$</td>
<td>47</td>
<td>70</td>
<td>60.8</td>
<td>8</td>
</tr>
<tr>
<td>$v_3$</td>
<td>74</td>
<td>65</td>
<td>68.6</td>
<td>5</td>
</tr>
<tr>
<td>$v_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Allocation Algorithm

- Available core count of $v_4 = 4$
- Required process count = 16
- Compute weight ($\alpha$) = 0.4
- Network weight ($\beta$) = 0.6
- Picked $v_1$
- Allocated process count = 10

<table>
<thead>
<tr>
<th>Node</th>
<th>Compute load</th>
<th>Network load</th>
<th>Addition load</th>
<th>#Available cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>52</td>
<td>45</td>
<td>47.8</td>
<td>6</td>
</tr>
<tr>
<td>$v_2$</td>
<td>47</td>
<td>70</td>
<td>60.8</td>
<td>8</td>
</tr>
<tr>
<td>$v_3$</td>
<td>74</td>
<td>65</td>
<td>68.6</td>
<td>5</td>
</tr>
</tbody>
</table>
Allocation Algorithm

- Available core count of $v_4 = 4$
- Required process count = 16
- Compute weight ($\alpha$) = 0.4
- Network weight ($\beta$) = 0.6
- Picked $v_2$
- Allocated process count = 18

<table>
<thead>
<tr>
<th>Node</th>
<th>Compute load</th>
<th>Network load</th>
<th>Addition load</th>
<th>#Available cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>52</td>
<td>45</td>
<td>47.8</td>
<td>6</td>
</tr>
<tr>
<td>$v_2$</td>
<td>47</td>
<td>70</td>
<td>60.8</td>
<td>8</td>
</tr>
<tr>
<td>$v_3$</td>
<td>74</td>
<td>65</td>
<td>68.6</td>
<td>5</td>
</tr>
</tbody>
</table>

Ashish Kumar, Naman Jain, Preeti Malakar
Network and Load-Aware Resource Manager for MPI Programs
Candidate Selection Algorithm

- Compute addition cost for all the nodes w.r.t start node

  Addition Cost: $A_v(u) = \alpha \times CL(u) + \beta \times NL(v, u)$
Candidate Selection Algorithm

- Compute addition cost for all the nodes w.r.t start node
  
  Addition Cost: \( A_v(u) = \alpha \times CL(u) + \beta \times NL(v, u) \)

- Keep adding nodes to sub-graph until allocated processes is more than required number of processes
Candidate Selection Algorithm

Input : Node $v$, Graph $G$, $PC$ List of effective processor count, $n$ Requested number of processes
Output: $G_v$ sub-graph with $v$ included
Candidate Selection Algorithm

Input : Node \( v \), Graph \( G \), \( PC \) List of effective processor count, \( n \) Requested number of processes

Output: \( G_v \) sub-graph with \( v \) included

\[
V_v \leftarrow \emptyset; \\
\text{allocated process} \leftarrow 0; \\
k \leftarrow \text{Total number of nodes in } G; \\
V_v \leftarrow V_v \cup \{u_i\} \\
\text{allocated process} \leftarrow \text{allocated process} + \text{pc}_{u_i} \\
i \leftarrow i + 1;
\]

Algorithm 2: Candidate Selection Algorithm
Candidate Selection Algorithm

**Input**: Node $v$, Graph $G$, $PC$ List of effective processor count, $n$ Requested number of processes

**Output**: $G_v$ sub-graph with $v$ included

\[
V_v \leftarrow \phi; \\
\text{allocated process} \leftarrow 0; \\
k \leftarrow \text{Total number of nodes in } G; \\
A_v(v) \leftarrow 0; \\
\text{Calculate } A_v(u) \text{ for each node other than } v; \\
\text{Let } u_1, u_2, u_3, \ldots, u_k \text{ be the vertices sorted in increasing order of addition load } A_v(u);
Candidate Selection Algorithm

Input: Node \( v \), Graph \( G \), \( \mathcal{PC} \) List of effective processor count, \( n \) Requested number of processes

Output: \( G_v \) sub-graph with \( v \) included

1. \( \mathcal{V}_v \leftarrow \phi \);
2. allocated process \( \leftarrow 0 \);
3. \( k \leftarrow \) Total number of nodes in \( G \);
4. \( A_v(v) \leftarrow 0 \);
5. Calculate \( A_v(u) \) for each node other than \( v \);
6. Let \( u_1, u_2, u_3, \ldots, u_k \) be the vertices sorted in increasing order of addition load \( A_v(u) \);
7. \( i \leftarrow 1 \);
8. While \( \text{allocated processes} < n \) do
   9. \( \mathcal{V}_v \leftarrow \mathcal{V}_v \cup \{u_i\} \);
   10. allocated process \( \leftarrow \) allocated process + \( pc_{u_i} \);
   11. \( i \leftarrow i+1 \);
9. End

Algorithm 4: Candidate Selection Algorithm
Resource Monitoring

- Developed a distributed monitoring system for flexibility

Livehosts: nodes which are up and running

Node statistics: available memory, CPU load and CPU utilization, etc.

Network statistics: available bandwidth and latency
Resource Monitoring

- Developed a distributed monitoring system for flexibility
- Light-weight daemons for periodically updating:
  - Livehosts: nodes which are up and running
  - Node statistics: available memory, CPU load and CPU utilization, etc.
  - Network statistics: available bandwidth and latency
Resource Monitoring

- Developed a distributed monitoring system for flexibility
- Light-weight daemons for periodically updating:
  - Livehosts: nodes which are up and running
Resource Monitoring

- Developed a distributed monitoring system for flexibility
- Light-weight daemons for periodically updating:
  - Livehosts: nodes which are up and running
  - Node statistics: available memory, CPU load and CPU utilization, etc.
Resource Monitoring

- Developed a distributed monitoring system for flexibility
- Light-weight daemons for periodically updating:
  - Livehosts: nodes which are up and running
  - Node statistics: available memory, CPU load and CPU utilization, etc.
  - Network statistics: available bandwidth and latency
Experimental setup:

- 40 12-core Intel Core nodes (4.6 GHz) and 20 8-core Intel Core nodes (2.8 GHz)
- Cluster has a tree-like hierarchical topology with 4 switches

Mantevo benchmark (miniMD): a simple, parallel molecular dynamics mini-application

Mantevo benchmark (miniFE): proxy application for unstructured implicit finite element codes which sets up a brick-shaped problem domain of hexahedral elements

Comparison with:

- Random allocation
- Sequential allocation
- Load-aware allocation
Experimental Setup and Benchmark

- **Experimental setup:**
  - 40 12-core Intel Core nodes (4.6 GHz) and 20 8-core Intel Core nodes (2.8 GHz)
  - Cluster has a tree-like hierarchical topology with 4 switches
- **Mantevo benchmark (miniMD):** a simple, parallel molecular dynamics mini-application

- **Mantevo benchmark (miniFE):** proxy application for unstructured implicit finite element codes which sets up a brick-shaped problem domain of hexahedral elements

- Comparison with:
  - Random allocation
  - Sequential allocation
  - Load-aware allocation
Experimental Setup and Benchmark

- **Experimental setup:**
  - 40 12-core Intel Core nodes (4.6 GHz) and 20 8-core Intel Core nodes (2.8 GHz)
  - Cluster has a tree-like hierarchical topology with 4 switches

- **Mantevo benchmark (miniMD):** a simple, parallel molecular dynamics mini-application

- **Mantevo benchmark (miniFE):** proxy application for unstructured implicit finite element codes which sets up a brick-shaped problem domain of hexahedral elements
Experimental Setup and Benchmark

- **Experimental setup:**
  - 40 12-core Intel Core nodes (4.6 GHz) and 20 8-core Intel Core nodes (2.8 GHz)
  - Cluster has a tree-like hierarchical topology with 4 switches

- **Mantevo benchmark (miniMD)**: a simple, parallel molecular dynamics mini-application

- **Mantevo benchmark (miniFE)**: proxy application for unstructured implicit finite element codes which sets up a brick-shaped problem domain of hexahedral elements

- **Comparison with:**
  - Random allocation
  - Sequential allocation
  - Load-aware allocation
Weights for miniMD and miniFE

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Load</td>
<td>0.3</td>
</tr>
<tr>
<td>CPU Utilization</td>
<td>0.2</td>
</tr>
<tr>
<td>Node Bandwidth</td>
<td>0.2</td>
</tr>
<tr>
<td>Used memory</td>
<td>0.1</td>
</tr>
<tr>
<td>Logical core count</td>
<td>0.1</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>0.05</td>
</tr>
<tr>
<td>Total Memory</td>
<td>0.05</td>
</tr>
</tbody>
</table>

**Table:** Relative weights for compute load
**Weights for miniMD and miniFE**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Load</td>
<td>0.3</td>
</tr>
<tr>
<td>CPU Utilization</td>
<td>0.2</td>
</tr>
<tr>
<td>Node Bandwidth</td>
<td>0.2</td>
</tr>
<tr>
<td>Used memory</td>
<td>0.1</td>
</tr>
<tr>
<td>Logical core count</td>
<td>0.1</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>0.05</td>
</tr>
<tr>
<td>Total Memory</td>
<td>0.05</td>
</tr>
</tbody>
</table>

*Table: Relative weights for compute load*

- Determined empirically
- Relative weights for latency and bandwidth were set to 0.25 and 0.75 respectively.
- Relative weights for compute and network load were set to 0.3 and 0.7 respectively for miniMD, and 0.4 and 0.6 respectively for miniFE.
Experiments: MiniMD

Time vs Atoms, Processes = 8, PPN = 4, Nodes = 2

Time vs Atoms, Processes = 32, PPN = 4, Nodes = 8

Time vs Atoms, Processes = 16, PPN = 4, Nodes = 4

Time vs Atoms, Processes = 64, PPN = 4, Nodes = 16
## Performance Gain: miniMD

<table>
<thead>
<tr>
<th>Allocation Policy</th>
<th>Average Gain</th>
<th>Median Gain</th>
<th>Maximum Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>49.9%</td>
<td>50.7%</td>
<td>87.8%</td>
</tr>
<tr>
<td>Sequential</td>
<td>43.1%</td>
<td>42.1%</td>
<td>84.5%</td>
</tr>
<tr>
<td>Load-Aware</td>
<td>32.4%</td>
<td>29.8%</td>
<td>87.7%</td>
</tr>
</tbody>
</table>
Average CPU load per logical core:

- Load aware = 0.31
- Network and Load aware = 0.43
- Sequential = 0.68
- Random = 0.78
Experiments: MiniFE

Time vs Problem dimension, Processes = 8, PPN = 4, Nodes = 2

Time (Seconds)

Problem Dimension (nx = ny = nz)

Time vs Problem dimension, Processes = 32, PPN = 4, Nodes = 8

Time (Seconds)

Problem Dimension (nx = ny = nz)

Time vs Problem dimension, Processes = 16, PPN = 4, Nodes = 4

Time (Seconds)

Problem Dimension (nx = ny = nz)

Time vs Problem dimension, Processes = 48, PPN = 4, Nodes = 12

Time (Seconds)

Problem Dimension (nx = ny = nz)
## Performance Gain: miniFE

<table>
<thead>
<tr>
<th>Allocation Policy</th>
<th>Average Gain</th>
<th>Median Gain</th>
<th>Maximum Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>47.9%</td>
<td>50.4%</td>
<td>92.1%</td>
</tr>
<tr>
<td>Sequential</td>
<td>31.1%</td>
<td>28.0%</td>
<td>80.4%</td>
</tr>
<tr>
<td>Load-Aware</td>
<td>34.8%</td>
<td>38.7%</td>
<td>91.0%</td>
</tr>
</tbody>
</table>
## Experiment: Resource Allocation Analysis

Configuration: 32 processes, 4 processes per node

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Avg. CPU load</th>
<th>Avg. bandwidth</th>
<th>Avg. latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>1.242</td>
<td>17.07</td>
<td>546.46</td>
</tr>
<tr>
<td>Sequential</td>
<td>1.262</td>
<td>10.72</td>
<td>304.25</td>
</tr>
<tr>
<td>Load Aware</td>
<td>0.453</td>
<td>18.64</td>
<td>354.51</td>
</tr>
<tr>
<td>Network and load-aware</td>
<td>0.633</td>
<td>5.36</td>
<td>82.90</td>
</tr>
</tbody>
</table>

**Table:** Usage of allocated resource group during allocation
Experiment: Resource Allocation Analysis

Configuration: 32 processes, 4 processes per node

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Avg. CPU load</th>
<th>Avg. bandwidth</th>
<th>Avg. latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>1.242</td>
<td>17.07</td>
<td>546.46</td>
</tr>
<tr>
<td>Sequential</td>
<td>1.262</td>
<td>10.72</td>
<td>304.25</td>
</tr>
<tr>
<td>Load Aware</td>
<td>0.453</td>
<td>18.64</td>
<td>354.51</td>
</tr>
<tr>
<td>Network and load-aware</td>
<td>0.633</td>
<td>5.36</td>
<td>82.90</td>
</tr>
</tbody>
</table>

Table: Usage of allocated resource group during allocation

Total execution time:

- Random: 27.61s
- Sequential: 24.91s
- Load aware: 12.31s
- Network & load aware: 4.43s
Experiments: Resource Allocation Analysis

### Table: Network and Load-Aware Resource Manager for MPI Programs

<table>
<thead>
<tr>
<th>NODES</th>
<th>csews1</th>
<th>csews4</th>
<th>csews5</th>
<th>csews6</th>
<th>csews8</th>
<th>csews9</th>
<th>csews10</th>
<th>csews12</th>
<th>csews13</th>
<th>csews15</th>
<th>csews16</th>
<th>csews19</th>
<th>csews20</th>
<th>csews23</th>
<th>csews28</th>
<th>csews32</th>
<th>csews50</th>
<th>csews51</th>
<th>csews54</th>
</tr>
</thead>
<tbody>
<tr>
<td>csews1</td>
<td>0</td>
<td>50</td>
<td>17</td>
<td>15</td>
<td>10</td>
<td>22</td>
<td>10</td>
<td>23</td>
<td>68</td>
<td>11</td>
<td>16</td>
<td>69</td>
<td>56</td>
<td>44</td>
<td>30</td>
<td>70</td>
<td>41</td>
<td>36</td>
<td>38</td>
</tr>
<tr>
<td>csews4</td>
<td>50</td>
<td>0</td>
<td>29</td>
<td>24</td>
<td>14</td>
<td>12</td>
<td>25</td>
<td>21</td>
<td>32</td>
<td>10</td>
<td>9</td>
<td>80</td>
<td>71</td>
<td>45</td>
<td>29</td>
<td>73</td>
<td>61</td>
<td>67</td>
<td>52</td>
</tr>
<tr>
<td>csews5</td>
<td>17</td>
<td>20</td>
<td>0</td>
<td>29</td>
<td>11</td>
<td>18</td>
<td>14</td>
<td>13</td>
<td>51</td>
<td>7</td>
<td>7</td>
<td>70</td>
<td>61</td>
<td>53</td>
<td>35</td>
<td>55</td>
<td>37</td>
<td>52</td>
<td>30</td>
</tr>
<tr>
<td>csews6</td>
<td>15</td>
<td>24</td>
<td>29</td>
<td>0</td>
<td>15</td>
<td>15</td>
<td>9</td>
<td>9</td>
<td>55</td>
<td>21</td>
<td>31</td>
<td>75</td>
<td>65</td>
<td>48</td>
<td>38</td>
<td>59</td>
<td>41</td>
<td>55</td>
<td>51</td>
</tr>
<tr>
<td>csews8</td>
<td>10</td>
<td>14</td>
<td>11</td>
<td>15</td>
<td>0</td>
<td>23</td>
<td>18</td>
<td>6</td>
<td>56</td>
<td>9</td>
<td>12</td>
<td>81</td>
<td>04</td>
<td>57</td>
<td>27</td>
<td>61</td>
<td>33</td>
<td>60</td>
<td>53</td>
</tr>
<tr>
<td>csews9</td>
<td>22</td>
<td>12</td>
<td>18</td>
<td>15</td>
<td>23</td>
<td>0</td>
<td>11</td>
<td>5</td>
<td>50</td>
<td>9</td>
<td>6</td>
<td>75</td>
<td>37</td>
<td>40</td>
<td>55</td>
<td>57</td>
<td>22</td>
<td>55</td>
<td>56</td>
</tr>
<tr>
<td>csews10</td>
<td>10</td>
<td>25</td>
<td>14</td>
<td>9</td>
<td>18</td>
<td>11</td>
<td>0</td>
<td>5</td>
<td>44</td>
<td>22</td>
<td>17</td>
<td>65</td>
<td>68</td>
<td>53</td>
<td>43</td>
<td>61</td>
<td>37</td>
<td>55</td>
<td>52</td>
</tr>
<tr>
<td>csews12</td>
<td>23</td>
<td>21</td>
<td>13</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>32</td>
<td>6</td>
<td>9</td>
<td>71</td>
<td>04</td>
<td>52</td>
<td>43</td>
<td>71</td>
<td>41</td>
<td>60</td>
<td>53</td>
</tr>
<tr>
<td>csews13</td>
<td>68</td>
<td>32</td>
<td>51</td>
<td>55</td>
<td>56</td>
<td>50</td>
<td>44</td>
<td>32</td>
<td>0</td>
<td>43</td>
<td>50</td>
<td>11</td>
<td>10</td>
<td>8</td>
<td>43</td>
<td>10</td>
<td>52</td>
<td>60</td>
<td>54</td>
</tr>
<tr>
<td>csews15</td>
<td>11</td>
<td>10</td>
<td>7</td>
<td>21</td>
<td>9</td>
<td>9</td>
<td>22</td>
<td>6</td>
<td>43</td>
<td>0</td>
<td>10</td>
<td>65</td>
<td>56</td>
<td>54</td>
<td>39</td>
<td>45</td>
<td>50</td>
<td>65</td>
<td>54</td>
</tr>
<tr>
<td>csews16</td>
<td>15</td>
<td>9</td>
<td>7</td>
<td>31</td>
<td>12</td>
<td>6</td>
<td>17</td>
<td>9</td>
<td>50</td>
<td>10</td>
<td>0</td>
<td>44</td>
<td>78</td>
<td>57</td>
<td>61</td>
<td>60</td>
<td>59</td>
<td>59</td>
<td>57</td>
</tr>
<tr>
<td>csews19</td>
<td>69</td>
<td>80</td>
<td>70</td>
<td>75</td>
<td>81</td>
<td>75</td>
<td>66</td>
<td>71</td>
<td>11</td>
<td>65</td>
<td>44</td>
<td>0</td>
<td>13</td>
<td>29</td>
<td>11</td>
<td>10</td>
<td>48</td>
<td>72</td>
<td>38</td>
</tr>
<tr>
<td>csews20</td>
<td>56</td>
<td>71</td>
<td>61</td>
<td>65</td>
<td>64</td>
<td>37</td>
<td>08</td>
<td>04</td>
<td>10</td>
<td>66</td>
<td>78</td>
<td>13</td>
<td>0</td>
<td>14</td>
<td>8</td>
<td>10</td>
<td>49</td>
<td>69</td>
<td>47</td>
</tr>
<tr>
<td>csews23</td>
<td>44</td>
<td>45</td>
<td>53</td>
<td>48</td>
<td>57</td>
<td>40</td>
<td>53</td>
<td>52</td>
<td>8</td>
<td>54</td>
<td>57</td>
<td>29</td>
<td>14</td>
<td>0</td>
<td>10</td>
<td>24</td>
<td>43</td>
<td>77</td>
<td>40</td>
</tr>
<tr>
<td>csews28</td>
<td>30</td>
<td>29</td>
<td>35</td>
<td>38</td>
<td>27</td>
<td>55</td>
<td>43</td>
<td>43</td>
<td>43</td>
<td>39</td>
<td>61</td>
<td>11</td>
<td>8</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>44</td>
<td>58</td>
<td>29</td>
</tr>
<tr>
<td>csews32</td>
<td>70</td>
<td>73</td>
<td>55</td>
<td>59</td>
<td>61</td>
<td>57</td>
<td>61</td>
<td>71</td>
<td>10</td>
<td>45</td>
<td>60</td>
<td>10</td>
<td>10</td>
<td>24</td>
<td>15</td>
<td>0</td>
<td>47</td>
<td>72</td>
<td>38</td>
</tr>
<tr>
<td>csews50</td>
<td>41</td>
<td>61</td>
<td>37</td>
<td>41</td>
<td>33</td>
<td>22</td>
<td>37</td>
<td>41</td>
<td>52</td>
<td>50</td>
<td>59</td>
<td>48</td>
<td>49</td>
<td>43</td>
<td>44</td>
<td>47</td>
<td>0</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>csews51</td>
<td>36</td>
<td>67</td>
<td>52</td>
<td>55</td>
<td>60</td>
<td>55</td>
<td>55</td>
<td>60</td>
<td>69</td>
<td>65</td>
<td>59</td>
<td>72</td>
<td>09</td>
<td>77</td>
<td>58</td>
<td>72</td>
<td>5</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>csews54</td>
<td>38</td>
<td>52</td>
<td>39</td>
<td>51</td>
<td>53</td>
<td>56</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>54</td>
<td>57</td>
<td>38</td>
<td>47</td>
<td>40</td>
<td>29</td>
<td>38</td>
<td>13</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Figure: Peer-to-peer bandwidth and CPU Load

Ashish Kumar, Naman Jain, Preeti Malakar

Network and Load-Aware Resource Manager for MPI Programs

Page 25 / 26
Shortcomings and Future Work

- Network and load aware algorithm reduces runtime by more than 38% over random, sequential, and load-aware allocations due to less interference from external factors.

- Challenging to determine the relative weights for resource attributes and computation-communication characteristics for large applications. We plan to enhance profiling tools for this purpose.

- Time series estimation methods may be used for bandwidth forecast.

- Extension to large scale systems, spanning over multiple clusters.

- Exploring integrating our tool as a plugin for SLURM job scheduler.