Load and Network-aware Node Allocator for MPI Jobs

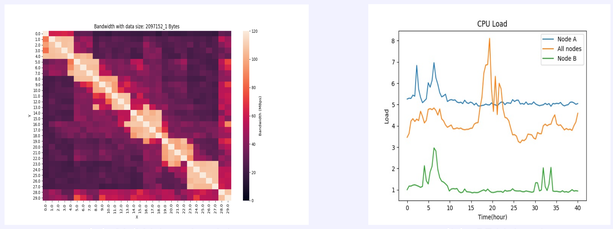
Ashish Kumar, Naman Jain, Preeti Malakar

Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur

Problem

Node allocation in a shared cluster for parallel jobs, with the goal of maximizing performance considering both compute and network load on the cluster.

Challenges



(a) N/W bandwidth (b) CPU load

Figure: Variation across nodes

- Non-exclusive access to nodes in shared cluster
- Variation in load/utilization across time and across nodes
- Topology does not capture the current state of network
- Contention and congestion in the network due to existing jobs
- Varying computation and communication requirements of different programs

Core Components

Eagle

- Distributed monitoring system for cluster
- Uses light-weight daemons for periodically updating livehosts, node statistics and network status

Allocator

- Allocates nodes based on user request
- Considers node attributes and network dynamics
- Uses data collected by Eagle

More details?

- Source Code: https://github.com/aasis21/NodeAllocator
- PPT : tinyurl.com/hipc-ppt
- Detailed workflow can be found here →

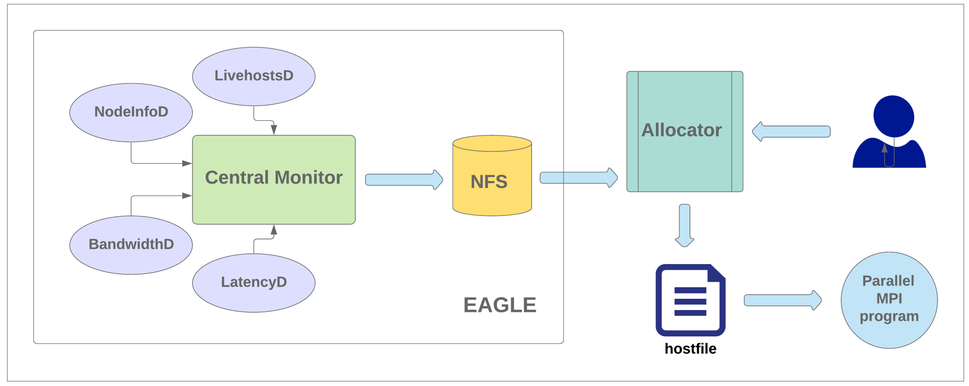


Figure: Allocator workflow

Problem Formulation

Model: Represent cluster as graph with vertices as compute nodes and edges as network links
Objective: Find a sub-graph satisfying user demands, with minimum overall load

Compute Load

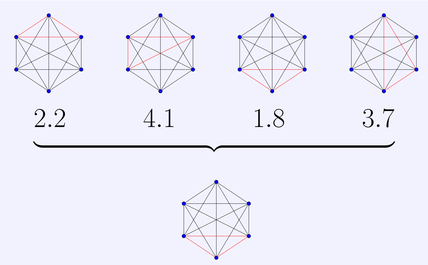
- Measure of overall load on the node
- Static (core count, clock speed) & dynamic (CPU load, available memory) attributes
- $CL_v = \sum_{a \in \text{attributes}} w_a * val_{va}$
- w_a represents attribute relative weight

Network Load

- Measure of load on the P2P network link
- Considers bandwidth and latency
- Topology automatically gets captured
- $NL_{(u,v)} = w_{lt}\text{LT}_{(u,v)} + w_{bw}\text{BW}_{(u,v)}$
- w_{lt} and w_{bw} represent relative weights

Algorithm

- Find candidate sub-graphs
- Calculate total load for each sub-graph
  - Compute Load, $C_{G_v} = \sum_{u \in \mathcal{V}_v} CL_u$
  - Network Load, $N_{G_v} = \sum_{(x,y) \in \mathcal{E}_v} NL_{(x,y)}$
  - Total Load $= \alpha \times C_{G_v} + \beta \times N_{G_v}$
- Pick the best one according to total load



Pictorial representation of allocation algorithm

Candidate Selection Algorithm

- Start with a particular node v
- Calculate addition load for all nodes w.r.t. start node
  $A_v(u) = \alpha \times CL(u) + \beta \times NL(v,u)$
- Keep adding nodes in increasing order of addition load to sub-graph until request is satisfied

Results

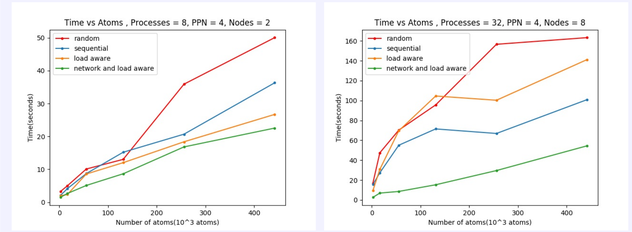


Figure: Comparison of our allocation with random, sequential and load-aware allocation

Weights:

- Weights are determined empirically for miniMD.
- CPU load, utilization and node bandwidth were given higher weight; 0.3, 0.2, 0.2 respectively.
- Logical core count, clock speed and total physical memory given lower weight; 0.1, 0.05, 0.05 respectively.
- w_{lt} and w_{bw} were set to 0.25 and 0.75 respectively.

Observations:

- Our algorithm performs better than random, sequential, and load-aware on an average.
- Load-aware performed better than sequential for less number of nodes whereas worse for a large number of nodes.
- The coefficient of variation for runs of our algorithm was 0.074 as compared to 0.13 for load-aware and 0.27 for sequential.

Algorithm	Avg. gain	Max. gain
Random	49.9%	87.8%
Sequential	43.1%	84.5%
Load Aware	32.4%	87.7%

Table: Performance gain using our allocation method

Conclusions and Future Work

- Our algorithm reduces run-times by more than 40% over random, sequential and load-aware allocations.
- Formalization of weights estimation by profiling the MPI program.
- Extension to large scale systems, spanning over multiple clusters.