Parallel Spectral Clustering in Distributed Systems

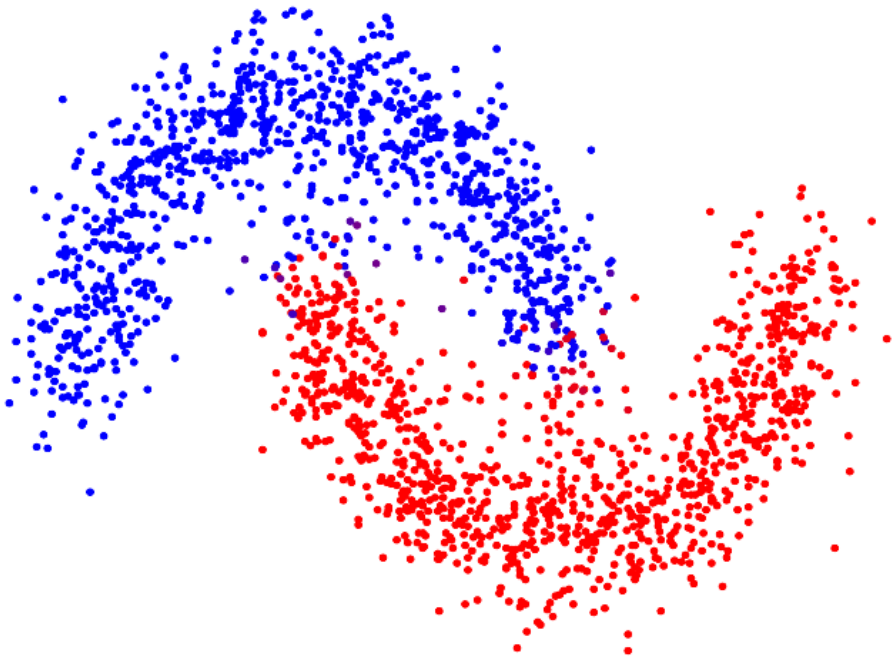
Liang Ge

Department of Computer Science and Engineering

State University of New York at Buffalo

Clustering

- **Clustering**
  - Important subroutines in machine learning and data mining
  - Partition data objects into groups where they are similar within group while dissimilar between group



Spectral Clustering

➢ Spectral Clustering

- The most recent state-of-the-art clustering (Shi. et al. PAMI 2000)

Spectral Clustering

Algorithm 1

Input: Data points $x_1, \cdots, x_n$, k: number of desired clusters

Output: Clustering: $\{C_1, ..., C_k\}$

1) Construct similarity matrix $S \in R^{n \times n}$
2) Modify S to be a sparse matrix
3) Compute the normalized Laplacian matrix L
4) Compute the first k eigenvectors of L
5) Construct $V \in R^{n \times k}$, whose columns are the k eigenvectors
6) Use k-means algorithm to cluster n rows of U into k groups

Spectral Clustering

Similarity Matrix S (Gaussian Kernel):

$$S_{ij} = exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})$$

Normalized Laplacian:

$$L = I - D^{-1/2} S D^{-1/2}$$

Diagonal matrix:

$$D_{ii} = \sum_{j=1}^{n} S_{ij}$$

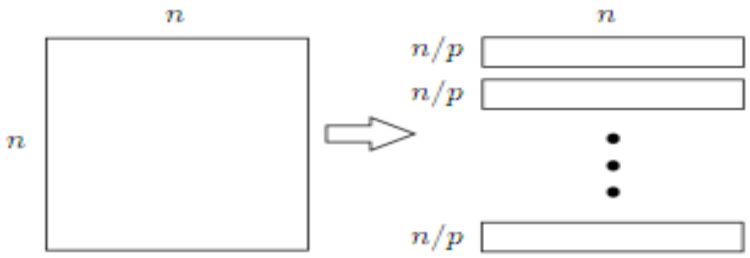
Data Set

- **RCV-1 Data set from MIT**
  - **199328 Documents**
  - **Each document is a vector of <index, value>**

Similarity Matrix Computation

General Idea

- Divide the matrix into p parts and stores them into p machines
- For each data point a in the master node, compute the distance with local point b in each machine
- Use p min-heap in each machine to save the local t-nearest neighbor
- The master node reduces the local min-heap to obtain the global t-nearest neighbor

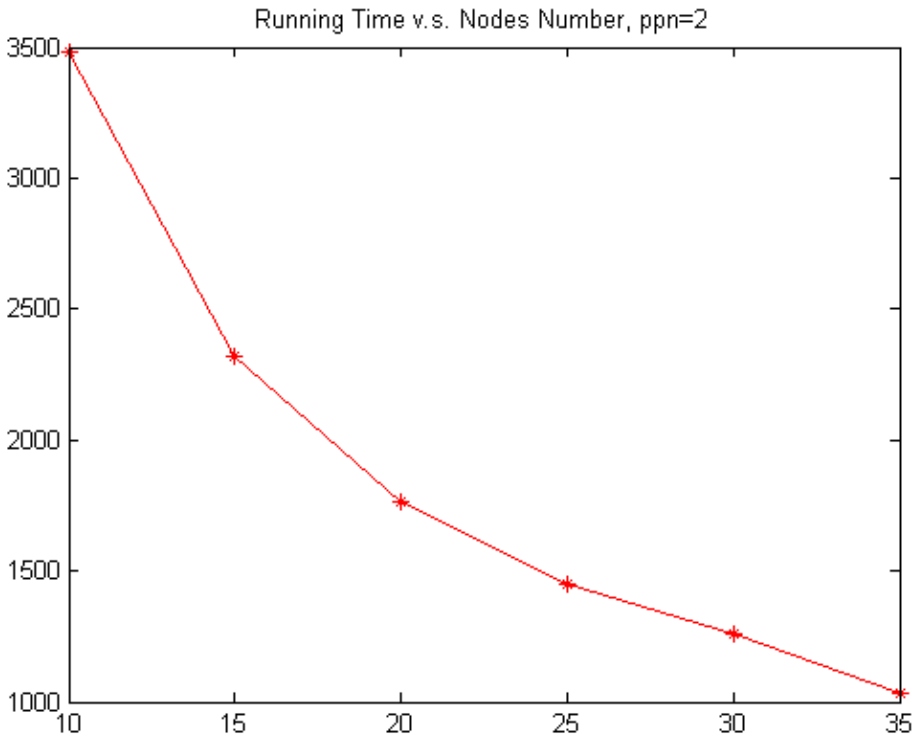


Major Codes

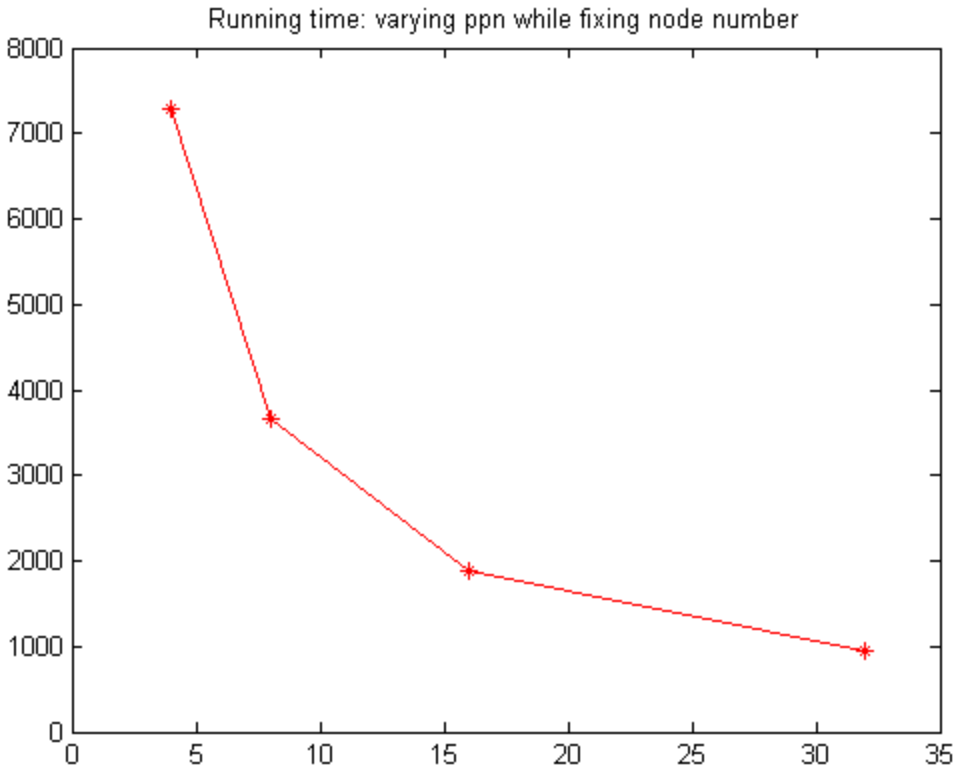
```
// Broadcast this document to all computer
Document doc;
if (which_computer == myid_) {
  doc = docs_[i / pnum_];
}
BroadcastDocument(&doc, which_computer);
for (int j = 0; j < docs_.size(); ++j) {
  if (which_computer == myid_ && i / pnum_ == j) {
    // Do not compute myself to myself
    continue;
  }
  double distance = sqrt(doc.two_norm_sq +
         docs_[j].two_norm_sq -
         2 * InnerProduct(doc, docs_[j]));
```

```
void ComputeDistance::BroadcastDocument(Document* doc, int root) {
  string s;
  if (myid_ == root) {
    doc->Encode(&s);
  }
  int s_size = s.size();
  MPI_Bcast(&s_size, 1, MPI_INT, root, MPI_COMM_WORLD);
  if (myid_ != root) {
    s.resize(s_size);
  }
  MPI_Bcast(&s[0], s_size, MPI_CHAR, root, MPI_COMM_WORLD);
  doc->Decode(s);
}
```

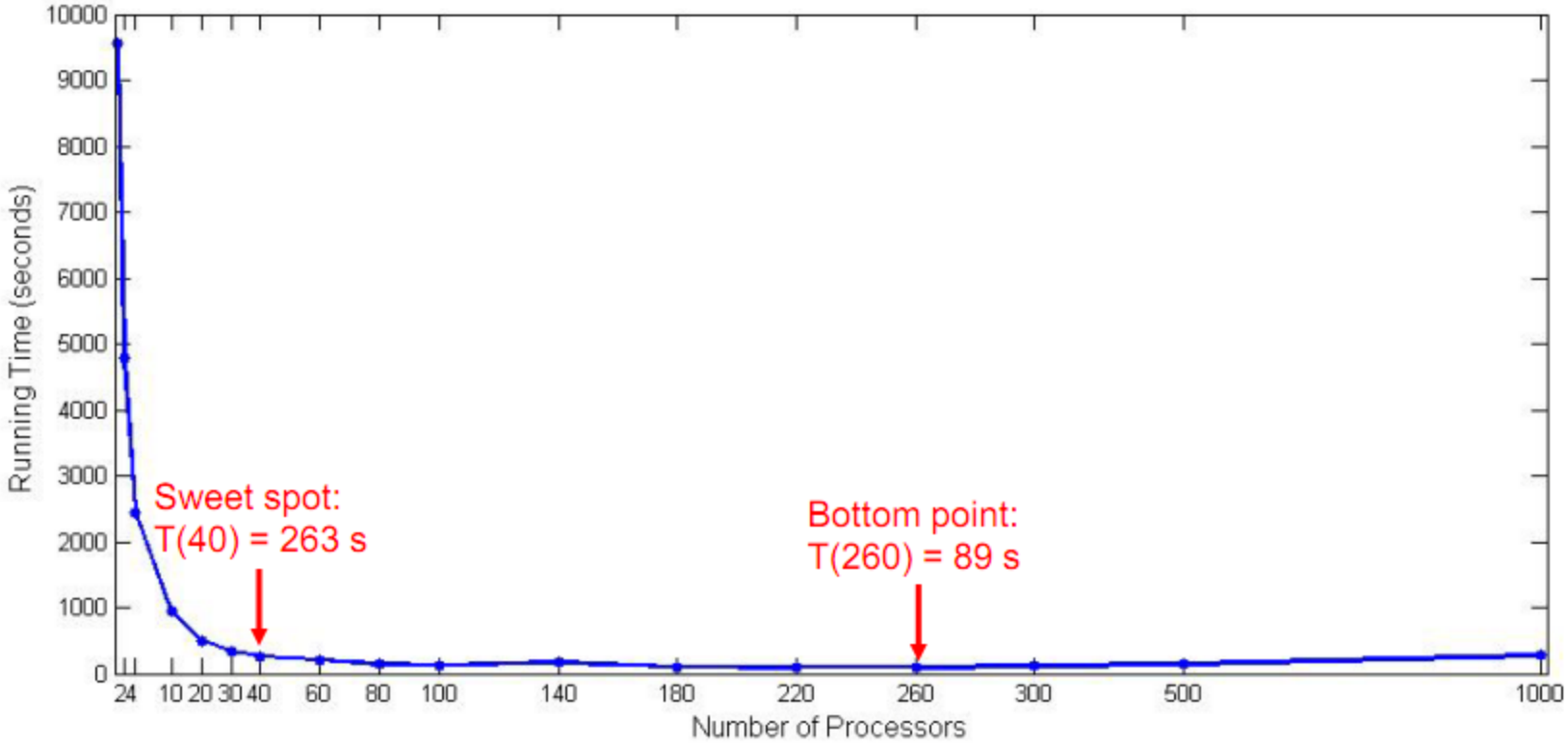
Running Time: varying nodes with fixing ppn



Running Time: varying ppn while fixing nodes



Why we don't see the turning point?

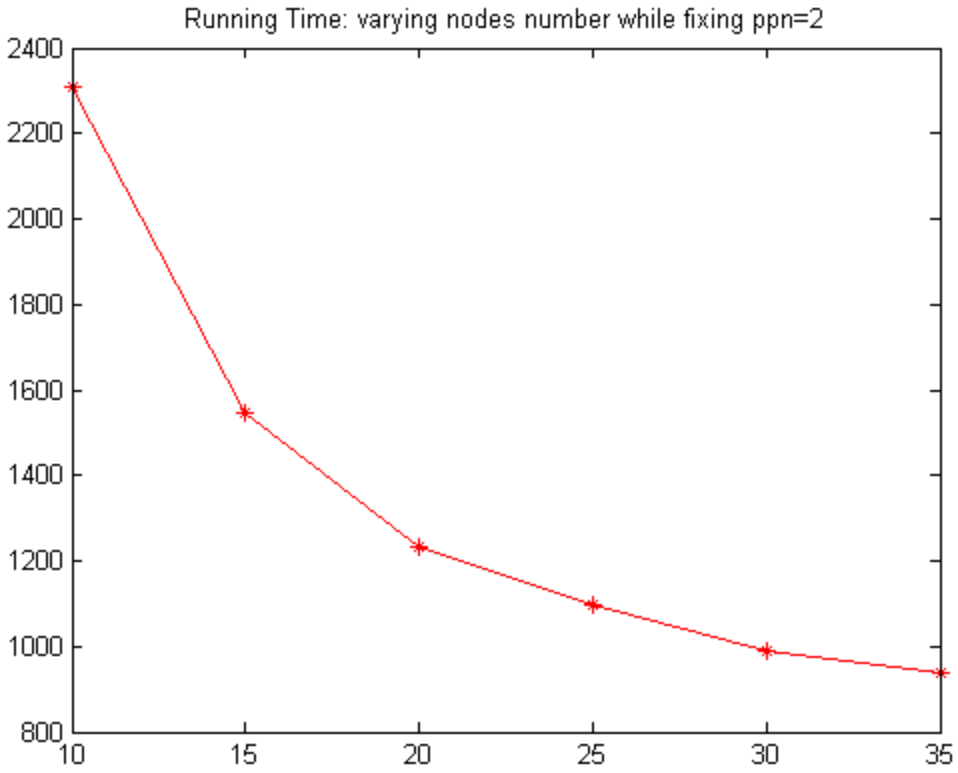


Finding the Eigenvectors

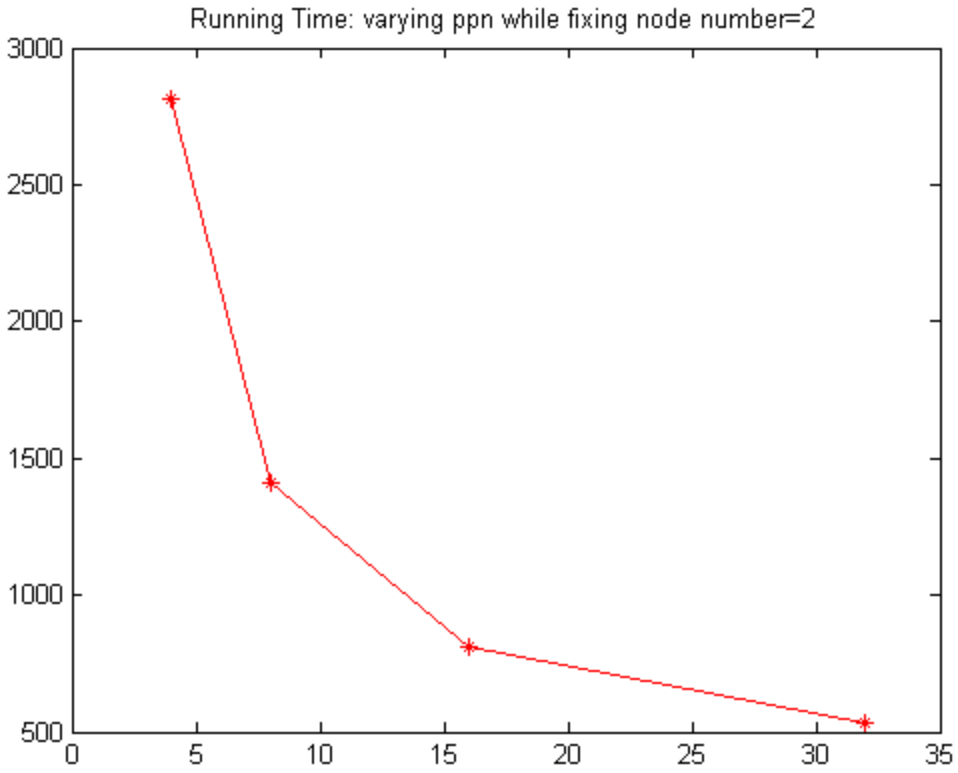
- **Compute the first k eigenvectors**
  - Arnoldi factorization
  - PARPACK: a parallel ARPACK implementation based on MPI

```
void Evd::Compute(int num_eigen, int eigen_space,
                  int max_iterations, double tolerance) {
  CHECK_GT(num_total_rows_, num_eigen);
  CHECK_GE(num_total_rows_, eigen_space);
  solver_ =
      new EigenSolverSymmetric(num_local_rows_,
                               "LA",
                               num_eigen,
                               eigen_space);

  // Do EVD.
  solver_->set_max_iterations(max_iterations);
  solver_->set_tolerance(tolerance);
  solver_->Solve(*this,
                 &num_converged_,
                 &eigen_values_,
                 &eigen_vectors_);
}
```

Running Time: varying nodes with fixing ppn



Running Time: varying nodes with fixing ppn



Parallel k-means

- **Initial k Cluster Centers**
  - The master node randomly choose one as the first cluster center
  - It broadcasts the center to all the worker nodes
  - Each worker node finds one point that is farthest to this point
  - The master node choose the second cluster center from all worker nodes return
  - Iterate k times to find the k initial cluster centers
  - It is actually a MPI_AllReduce operation for the master node

Initial k Cluster Centers

```
  for (int i = 0; i < num_columns_; ++i) {
    cluster_centers_[0][i] = local_rows_[rand_index][i];
  }
}

MPI_Bcast(&cluster_centers_[0][0],
          num_columns_,
          MPI_DOUBLE,
          0, MPI_COMM_WORLD);


MPI_Allreduce(&cluster_centers_storage_[0],
              &cluster_centers_storage_backup[0],
              num_clusters_ * num_columns_,
              MPI_DOUBLE,
              MPI_SUM, MPI_COMM_WORLD);
memcpy(&cluster_centers_storage_[0],
       &cluster_centers_storage_backup[0], num_clusters_ * num_columns_);
```

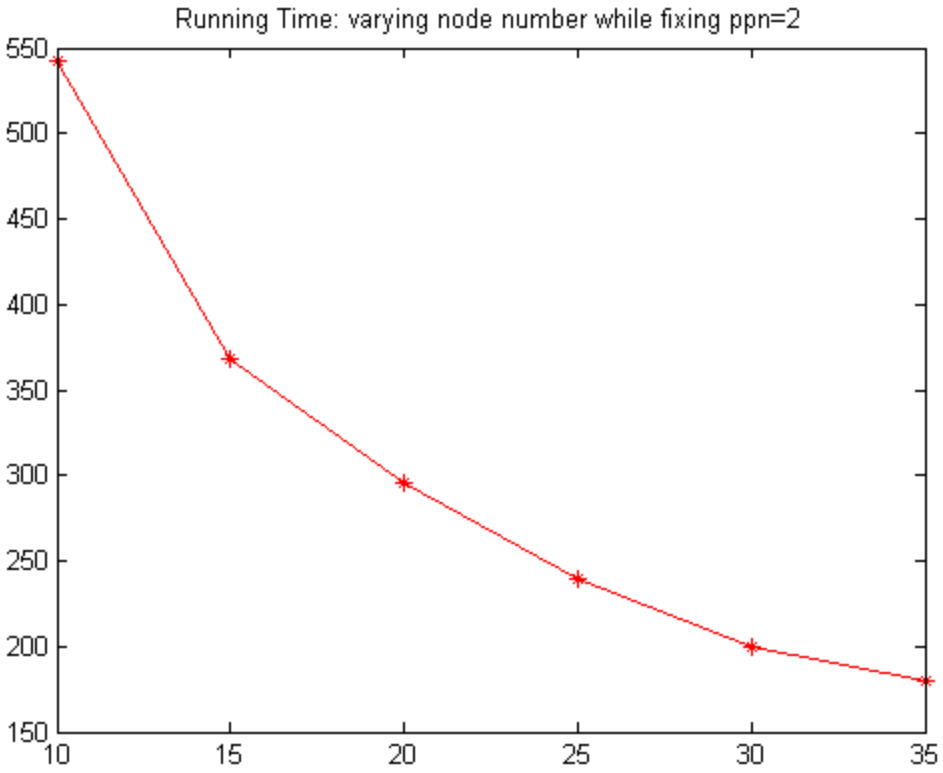
Parallel k-means

- **Parallel K-means**
  - k initial cluster centers will be broadcast to all machines with local data
  - Each machine computes labels of points by assigning to their nearest neighbors

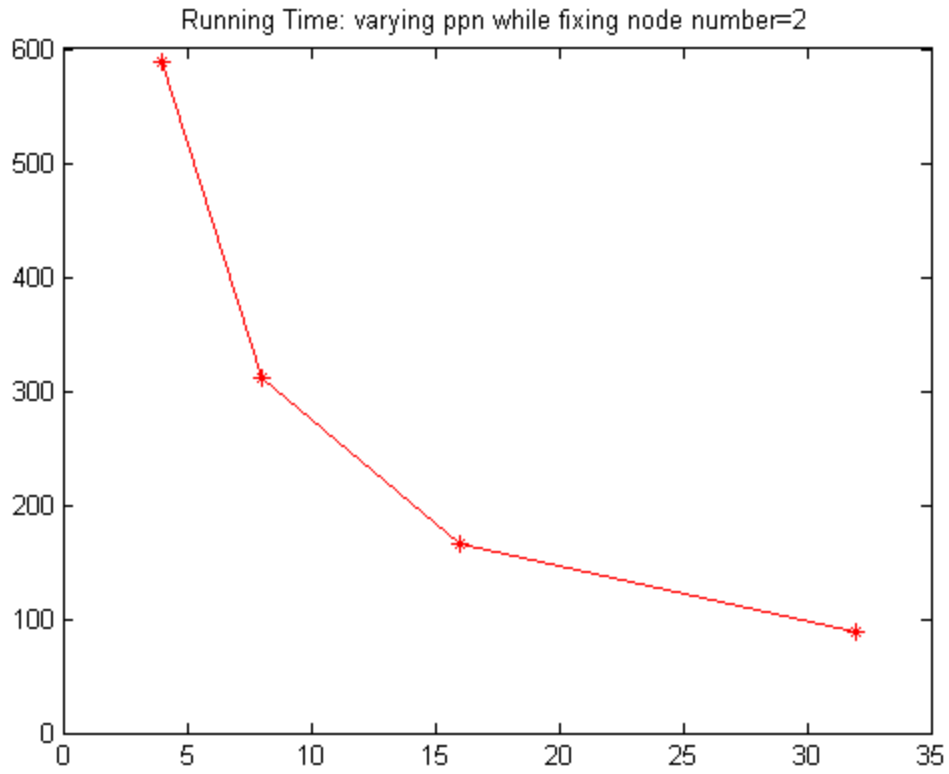
Parallel k-means

```
// Sum all data points.
MPI_Allreduce(&new_cluster_centers_storage[0],
              &cluster_centers_storage_[0],
              num_clusters_ * num_columns_,
              MPI_DOUBLE,
              MPI_SUM, MPI_COMM_WORLD);

MPI_Allreduce(&new_cluster_sizes[0],
              &cluster_sizes_[0],
              num_clusters_,
              MPI_INT,
              MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(&sum_local_loss,
              &sum_total_loss,
              1,
              MPI_DOUBLE,
              MPI_SUM, MPI_COMM_WORLD);

// Compute the loss change.
delta = fabs(sum_total_loss - sum_total_loss_old)
    / (sum_total_loss_old + 1e-10);
sum_total_loss_old = sum_total_loss;

// Average the sum to obtain new centers.
int count_not_zero = 0;
for (int i = 0; i < num_clusters_; ++i) {
  if (cluster_sizes_[i] > 0) {
    for (int j = 0; j < num_columns_; ++j) {
      cluster_centers_[i][j] /= cluster_sizes_[i];
```

Running Time: varying nodes with fixing ppn



Running Time: varying ppn while fixing node=2



Conclusions and Future Work

- **Parallel Computing is a great way of reducing running time with the cost of complicated codes and tricky debugging**

- **Within node communication is faster than between node communication, enabling greater speedup.**

- **The communication and initialization cost, no matter how small, will eventually dominate the running time if we continue to increase number of processors**