Methods for Partitioning Data to Improve Parallel Execution Time for Sorting on Heterogeneous Clusters

C. Cérin[1] J.-C. Dubacq[1] J.-L. Roch[2]

[1]LIPN
Université de Paris Nord

[2]ID-IMAG
Université Joseph Fourier, Grenoble

Global and Pervasive Computing 2006 (台中市)

Outline

1. Motivation
   - The partitioning problem
   - Splitting data

2. Contribution
   - General exact analytic approach
   - Dynamic evaluation of complexity function
   - Non uniformly related processors
   - Experiments

Outline

1. Motivation
   - The partitioning problem
   - Splitting data
2. Contribution
   - General exact analytic approach
   - Dynamic evaluation of complexity function
   - Non uniformly related processors
   - Experiments

Partitioning large data sets for sorting

- Large data sets require lot of computation time for sorting;

Partitioning large data sets for sorting

- Large data sets require lot of computation time for sorting;
- Data chunks of equal size used to do the job on parallel machines.

Partitioning large data sets for sorting

- Large data sets require lot of computation time for sorting;
- Data chunks of equal size used to do the job on parallel machines.

Modelisation

- Infinite point-to-point bandwidth;

Partitioning large data sets for sorting

- Large data sets require lot of computation time for sorting;
- Data chunks of equal size used to do the job on parallel machines.

Modelisation

- Infinite point-to-point bandwidth;
- Heterogeneous speed: relative linear speed;

Partitioning large data sets for sorting

- Large data sets require lot of computation time for sorting;
- Data chunks of equal size used to do the job on parallel machines.

Modelisation

- Infinite point-to-point bandwidth;
- Heterogeneous speed: relative linear speed;
- No study of memory effect.

Methodology

1. Data chunks are sent from node 0 to nodes $1, \ldots, p-1$;

Methodology

1. Data chunks are sent from node 0 to nodes $1, \ldots, p-1$;
2. Each processor sorts locally its data chunk;

Methodology

1. Data chunks are sent from node 0 to nodes $1, \ldots, p-1$;
2. Each processor sorts locally its data chunk;
3. Node 0 receives $p-1$ pivots, sorts them and broadcasts them;

Methodology

1. Data chunks are sent from node 0 to nodes $1, \ldots, p-1$;
2. Each processor sorts locally its data chunk;
3. Node 0 receives $p-1$ pivots, sorts them and broadcasts them;
4. Each processor uses the pivots to split its data;

Methodology

1. Data chunks are sent from node 0 to nodes $1, \ldots, p-1$;
2. Each processor sorts locally its data chunk;
3. Node 0 receives $p-1$ pivots, sorts them and broadcasts them;
4. Each processor uses the pivots to split its data;
5. Each processor transmits all its (split) data to the others;

Methodology

1. Data chunks are sent from node 0 to nodes $1, \ldots, p-1$;
2. Each processor sorts locally its data chunk;
3. Node 0 receives $p-1$ pivots, sorts them and broadcasts them;
4. Each processor uses the pivots to split its data;
5. Each processor transmits all its (split) data to the others;
6. Each processor merges all data it received with its own.

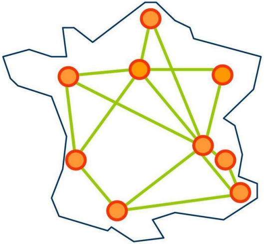
Methodology

1. Data chunks are sent from node 0 to nodes $1, \ldots, p-1$;
2. Each processor sorts locally its data chunk;
3. Node 0 receives $p-1$ pivots, sorts them and broadcasts them;
4. Each processor uses the pivots to split its data;
5. Each processor transmits all its (split) data to the others;
6. Each processor merges all data it received with its own.

Observation

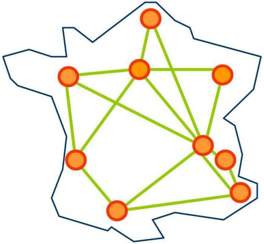
With fixed p, the computation-intensive part is step 2.

Context: Grid'5000, heterogeneous clusters



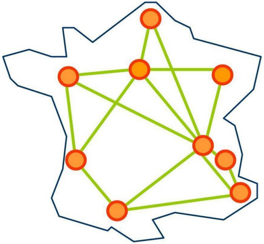
- GRID'5000: French national research project on grids;

Context: Grid'5000, heterogeneous clusters



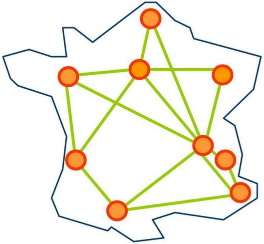
- GRID'5000: French national research project on grids;
- Goal: 5000 nodes dedicated to experimental development;

Context: Grid'5000, heterogeneous clusters



- GRID'5000: French national research project on grids;
- Goal: 5000 nodes dedicated to experimental development;
- Current state: 2300 nodes, 13+ separated clusters, 9 sites, dedicated 10 Gb/s black fibre connexion;

Context: Grid'5000, heterogeneous clusters



- GRID'5000: French national research project on grids;
- Goal: 5000 nodes dedicated to experimental development;
- Current state: 2300 nodes, 13+ separated clusters, 9 sites, dedicated 10 Gb/s black fibre connexion;

Heterogeneity

Clusters have different processors, same family-processors have different clock speeds.

Outline

1. Motivation
   - The partitioning problem
   - Splitting data

2. Contribution
   - General exact analytic approach
   - Dynamic evaluation of complexity function
   - Non uniformly related processors
   - Experiments

From homogeneous to heterogeneous processors

Goal

We have N objects to transmit and transform using p nodes. We want all computation to end at exactly the same time. Final merging is not relevant.

From homogeneous to heterogeneous processors

Goal

We have N objects to transmit and transform using p nodes. We want all computation to end at exactly the same time. Final merging is not relevant.

Theorem (Homogeneous case)

If all nodes work at same speed, the splitting of the data is optimal if one uses chunks of size N/p.

From homogeneous to heterogeneous processors

Goal

We have N objects to transmit and transform using p nodes. We want all computation to end at exactly the same time. Final merging is not relevant.

Theorem (Homogeneous case)

If all nodes work at same speed, the splitting of the data is optimal if one uses chunks of size N/p.

We define the relative speed k_i of a node i as the quantity of operations it can do by unit of time compared to a reference node, and $K = \sum_j k_j$.

Previous works

Naïve algorithm uses chunks of size $\frac{k_i}{K}N$ and yields inadequate computation time.

Previous works

Naïve algorithm uses chunks of size $\frac{k_i}{K}N$ and yields inadequate computation time.

Example (naïve algorithm)

$$\begin{array}{llll} \text{Node 1} & k_1 = 1 & n_1 = \frac{N}{3} & T_1 = n_1 \log n_1 \\ \text{Node 2} & k_2 = 2 & n_2 = \frac{2N}{3} & T_2 = \frac{n_2 \log n_2}{k_2} \end{array}$$

$$T_2 = n_1 \log (2n_1) = T_1 + n_1 \log 2 \neq T_1$$

Previous works

Naïve algorithm uses chunks of size $\frac{k_i}{K}N$ and yields inadequate computation time.

Example (naïve algorithm)

$$\begin{array}{llll} \text{Node 1} & k_1 = 1 & n_1 = \frac{N}{3} & T_1 = n_1 \log n_1 \\ \text{Node 2} & k_2 = 2 & n_2 = \frac{2N}{3} & T_2 = \frac{n_2 \log n_2}{k_2} \end{array}$$

$$T_2 = n_1 \log(2n_1) = T_1 + n_1 \log 2 \neq T_1$$

Theorem (Cérin,Koskas,Jemni,Fkaier)

For large N, optimal chunk size is

$$n_i = \frac{k_i}{K}N + \epsilon_i, \quad (1 \leq i \leq p) \text{ where } \epsilon_i = \frac{N}{\ln N}\left[\frac{k_i}{K^2}\sum_{j=1}^{p} k_j \ln\left(\frac{k_j}{k_i}\right)\right]$$

Outline

1. Motivation
   - The partitioning problem
   - Splitting data
2. Contribution
   - General exact analytic approach
   - Dynamic evaluation of complexity function
   - Non uniformly related processors
   - Experiments

Basic approach

We use $\tilde{f}$ as the complexity function ($T_i = \tilde{f}(n_1)/k_i$).

$$T = \frac{\tilde{f}(n_1)}{k_1} = \frac{\tilde{f}(n_2)}{k_2} = \cdots = \frac{\tilde{f}(n_p)}{k_p}$$

Basic approach

We use $\tilde{f}$ as the complexity function ($T_i = \tilde{f}(n_1)/k_i$).

$$T = \frac{\tilde{f}(n_1)}{k_1} = \frac{\tilde{f}(n_2)}{k_2} = \cdots = \frac{\tilde{f}(n_p)}{k_p}$$

$$n_1 + n_2 + ... + n_p = N$$

Basic approach

We use $\tilde{f}$ as the complexity function ($T_i = \tilde{f}(n_1)/k_i$).

$$T = \frac{\tilde{f}(n_1)}{k_1} = \frac{\tilde{f}(n_2)}{k_2} = \cdots = \frac{\tilde{f}(n_p)}{k_p}$$

$$n_1 + n_2 + ... + n_p = N$$

Thus we can derive these compact equations for equality:

Basic approach

We use $\tilde{f}$ as the complexity function ($T_i = \tilde{f}(n_1)/k_i$).

$$T = \frac{\tilde{f}(n_1)}{k_1} = \frac{\tilde{f}(n_2)}{k_2} = \cdots = \frac{\tilde{f}(n_p)}{k_p}$$

$$n_1 + n_2 + ... + n_p = N$$

Thus we can derive these compact equations for equality:

$$n_i = \tilde{f}^{-1}(T.k_i)$$

Basic approach

We use $\tilde{f}$ as the complexity function ($T_i = \tilde{f}(n_1)/k_i$).

$$T = \frac{\tilde{f}(n_1)}{k_1} = \frac{\tilde{f}(n_2)}{k_2} = \cdots = \frac{\tilde{f}(n_p)}{k_p}$$

$$n_1 + n_2 + ... + n_p = N$$

Thus we can derive these compact equations for equality:

$$n_i = \tilde{f}^{-1}(T.k_i) \quad \text{and} \quad \sum_{i=1}^{p} \tilde{f}^{-1}(T.k_i) = N$$

Basic approach

We use $\tilde{f}$ as the complexity function ($T_i = \tilde{f}(n_1)/k_i$).

$$T = \frac{\tilde{f}(n_1)}{k_1} = \frac{\tilde{f}(n_2)}{k_2} = \cdots = \frac{\tilde{f}(n_p)}{k_p}$$

$$n_1 + n_2 + ... + n_p = N$$

Thus we can derive these compact equations for equality:

$$n_i = \tilde{f}^{-1}(T.k_i) \quad \text{and} \quad \sum_{i=1}^{p} \tilde{f}^{-1}(T.k_i) = N$$

Only one unknown variable left!

The polynomial case

Theorem (Polynomial case)

If $\tilde{f} : x \mapsto \alpha x^{\beta}$, then the optimal division is obtained by chunks sizes:

The polynomial case

Theorem (Polynomial case)

If $\tilde{f} : x \mapsto \alpha x^{\beta}$, then the optimal division is obtained by chunks sizes:

$$n_i = \frac{k_i^{1/\beta}}{\sum_{i=1}^{p} k_i^{1/\beta}} N.$$

The polynomial case

Theorem (Polynomial case)

If $\tilde{f} : x \mapsto \alpha x^{\beta}$, then the optimal division is obtained by chunks sizes:

$$n_i = \frac{k_i^{1/\beta}}{\sum_{i=1}^{p} k_i^{1/\beta}} N.$$

Proof: $\tilde{f}$ is multiplicative.

$$\begin{aligned} \sum_{i=1}^{p} \tilde{f}^{-1}(T.k_i) = N \Longrightarrow \qquad & N = \tilde{f}^{-1}(T) \sum_{i=1}^{p} \tilde{f}^{-1}(k_i) \\ \Longrightarrow \qquad & T = \tilde{f}\left(\frac{N}{\sum_{i=1}^{p} \tilde{f}^{-1}(k_i)} \right) \end{aligned}$$

$\square$

The polylog case

Theorem

Initial values of n_i can be asymptotically computed by

$$\sum_{i=1}^{p} \frac{Tk_i + Tk_i \ln\ln(Tk_i)}{(\ln(Tk_i))^2} = N \textit{ and } n_i = \frac{Tk_i + Tk_i \ln\ln(Tk_i)}{(\ln(Tk_i))^2}$$

The polylog case

Theorem

Initial values of n_i can be asymptotically computed by

$$\sum_{i=1}^{p} \frac{Tk_i + Tk_i \ln\ln(Tk_i)}{(\ln(Tk_i))^2} = N \text{ and } n_i = \frac{Tk_i + Tk_i \ln\ln(Tk_i)}{(\ln(Tk_i))^2}$$

Proof.

We use the Lambert W function which is the inverse function of $x \mapsto x \log x$.

The polylog case

Theorem

Initial values of n_i can be asymptotically computed by

$$\sum_{i=1}^{p} \frac{Tk_i + Tk_i \ln\ln(Tk_i)}{(\ln(Tk_i))^2} = N \text{ and } n_i = \frac{Tk_i + Tk_i \ln\ln(Tk_i)}{(\ln(Tk_i))^2}$$

Proof.

We use the Lambert W function which is the inverse function of $x \mapsto x \log x$.

A well known approximation is $W(x) = \ln x - \ln\ln(x) + o(1)$. $\square$

Outline

Framework for unknown complexity function

Goal

We want to cope with unknown complexity functions. We have several batches of data.

Framework for unknown complexity function

Goal

We want to cope with unknown complexity functions. We have several batches of data.

- If the speed vector is unknown, first submit a batch assuming vector is $[1, \ldots, 1]$. Time-differences will tell what the relative speed is. So we may assume the speed vector is known;

Framework for unknown complexity function

Goal

We want to cope with unknown complexity functions. We have several batches of data.

- If the speed vector is unknown, first submit a batch assuming vector is $[1, \ldots, 1]$. Time-differences will tell what the relative speed is. So we may assume the speed vector is known;
- Deduce n_i chunk sizes to send to node i (in parallel for each node). Node n_i measures the treatment time for the chunk, and reports it at the end.

Framework for unknown complexity function

Goal

We want to cope with unknown complexity functions. We have several batches of data.

- If the speed vector is unknown, first submit a batch assuming vector is $[1, \ldots, 1]$. Time-differences will tell what the relative speed is. So we may assume the speed vector is known;
- Deduce n_i chunk sizes to send to node i (in parallel for each node). Node n_i measures the treatment time for the chunk, and reports it at the end.
- A piecewise representation of the complexity function is built, and missing values are interpolated.

Detailed algorithm

1. For each node i, precompute the mapping $(T, i) \mapsto n_i$ as previously, using interpolated values for f if necessary. Deduce a mapping $T \mapsto n$ by summing the mappings over all i.

Detailed algorithm

1. For each node i, precompute the mapping $(T, i) \mapsto n_i$ as previously, using interpolated values for f if necessary. Deduce a mapping $T \mapsto n$ by summing the mappings over all i.
2. Use a dichotomic search through $T \mapsto n$ mapping to find the ideal value of T (and thus of all the n_i) and assign chunks of data to node i;

Detailed algorithm

1. For each node i, precompute the mapping $(T, i) \mapsto n_i$ as previously, using interpolated values for f if necessary. Deduce a mapping $T \mapsto n$ by summing the mappings over all i.
2. Use a dichotomic search through $T \mapsto n$ mapping to find the ideal value of T (and thus of all the n_i) and assign chunks of data to node i;
3. When chunk i of size n_i is being treated:

Detailed algorithm

1. For each node i, precompute the mapping $(T, i) \mapsto n_i$ as previously, using interpolated values for f if necessary. Deduce a mapping $T \mapsto n$ by summing the mappings over all i.
2. Use a dichotomic search through $T \mapsto n$ mapping to find the ideal value of T (and thus of all the n_i) and assign chunks of data to node i;
3. When chunk i of size n_i is being treated:
    1. Record the cost $C = T_{n_i} k_i$ of the computation for size n_i.

Detailed algorithm

1. For each node i, precompute the mapping $(T, i) \mapsto n_i$ as previously, using interpolated values for f if necessary. Deduce a mapping $T \mapsto n$ by summing the mappings over all i.
2. Use a dichotomic search through $T \mapsto n$ mapping to find the ideal value of T (and thus of all the n_i) and assign chunks of data to node i;
3. When chunk i of size n_i is being treated:
   1. Record the cost $C = T_{n_i} k_i$ of the computation for size n_i.
   2. If n_i already had a non-interpolated value, choose a new value C' according to some strategy.

Detailed algorithm

1. For each node i, precompute the mapping $(T, i) \mapsto n_i$ as previously, using interpolated values for f if necessary. Deduce a mapping $T \mapsto n$ by summing the mappings over all i.
2. Use a dichotomic search through $T \mapsto n$ mapping to find the ideal value of T (and thus of all the n_i) and assign chunks of data to node i;
3. When chunk i of size n_i is being treated:
   1. Record the cost $C = T_{n_i} k_i$ of the computation for size n_i.
   2. If n_i already had a non-interpolated value, choose a new value C' according to some strategy.
   3. If n_i was not a known point, set $C' = C$.

Detailed algorithm

1. For each node i, precompute the mapping $(T, i) \mapsto n_i$ as previously, using interpolated values for f if necessary. Deduce a mapping $T \mapsto n$ by summing the mappings over all i.
2. Use a dichotomic search through $T \mapsto n$ mapping to find the ideal value of T (and thus of all the n_i) and assign chunks of data to node i;
3. When chunk i of size n_i is being treated:
   1. Record the cost $C = T_{n_i} k_i$ of the computation for size n_i.
   2. If n_i already had a non-interpolated value, choose a new value C' according to some strategy.
   3. If n_i was not a known point, set $C' = C$.
   4. Ensure that the mapping as defined by $n \neq n_i \mapsto C(n)$ and the new value $n_i \mapsto C'$ is still monotonous increasing.

Detailed algorithm

1. For each node i, precompute the mapping $(T, i) \mapsto n_i$ as previously, using interpolated values for f if necessary. Deduce a mapping $T \mapsto n$ by summing the mappings over all i.
2. Use a dichotomic search through $T \mapsto n$ mapping to find the ideal value of T (and thus of all the n_i) and assign chunks of data to node i;
3. When chunk i of size n_i is being treated:
   1. Record the cost $C = T_{n_i} k_i$ of the computation for size n_i.
   2. If n_i already had a non-interpolated value, choose a new value C' according to some strategy.
   3. If n_i was not a known point, set $C' = C$.
   4. Ensure that the mapping as defined by $n \neq n_i \mapsto C(n)$ and the new value $n_i \mapsto C'$ is still monotonous increasing.
4. A new batch can begin.

Outline

Non-uniformly related processors

Goal

We want to cope with complexity functions that depend on the node characteristics.

Non-uniformly related processors

Goal

We want to cope with complexity functions that depend on the node characteristics.

We can minimise the following formula by dynamic programming:

$$T(N,p) = \max_{i=1,\dots,p} \{f_i(n_i)\} = \min_{\substack{(x_1,\dots,x_p) \in \mathbb{N}^p \\ \sum_{i=1}^{p} x_i = N}} \left\{ \max_{i=1,\dots,p} \{f_i(x_i)\} \right\}$$

Non-uniformly related processors

Goal

We want to cope with complexity functions that depend on the node characteristics.

We can minimise the following formula by dynamic programming:

$$T(N,p) = \max_{i=1,\ldots,p} \{f_i(n_i)\} = \min_{\substack{(x_1,\ldots,x_p) \in \mathbb{N}^p \\ \sum_{i=1}^{p} x_i = N}} \left\{ \max_{i=1,\ldots,p} \{f_i(x_i)\} \right\}$$

$$T(m,i) = \min_{n_i=0..m} \max(f_i(n_i), C(m-n_i, i-1))$$

Non-uniformly related processors

Goal

We want to cope with complexity functions that depend on the node characteristics.

We can minimise the following formula by dynamic programming:

$$T(N,p) = \max_{i=1,\dots,p} \{f_i(n_i)\} = \min_{\substack{(x_1,\dots,x_p) \in \mathbb{N}^p \\ \sum_{i=1}^{p} x_i = N}} \left\{ \max_{i=1,\dots,p} \{f_i(x_i)\} \right\}$$

$$T(m,i) = \min_{n_i=0..m} \max(f_i(n_i), C(m-n_i, i-1))$$

Theorem

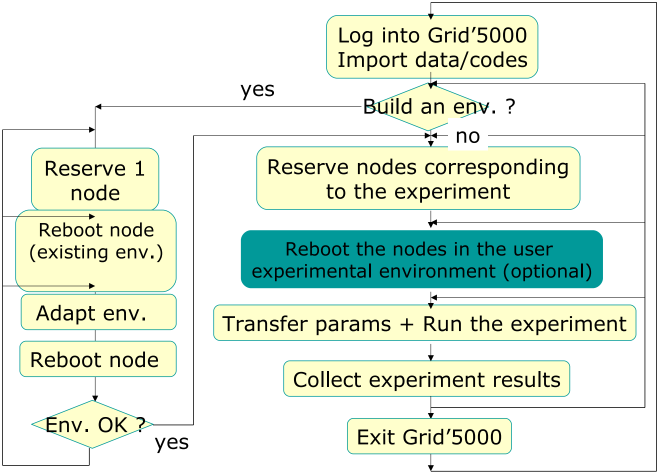
Computation of optimal partition is done in $\mathcal{O}(N^2 p)$ time.

Outline

1. Motivation
   - The partitioning problem
   - Splitting data
2. Contribution
   - General exact analytic approach
   - Dynamic evaluation of complexity function
   - Non uniformly related processors
   - Experiments

Experiments

Experiment workflow



Experiments

- Records of 100 bytes long, two classes of computers ($k = 1$ and $k = 1.5$);

Experiments

- Records of 100 bytes long, two classes of computers ($k = 1$ and $k = 1.5$);
- 54 GB of data, 50 runs for each experiment, bi-opteron processor, cpu-burning;

Experiments

- Records of 100 bytes long, two classes of computers ($k = 1$ and $k = 1.5$);
- 54 GB of data, 50 runs for each experiment, bi-opteron processor, cpu-burning;
- 96 nodes used;

Experiments

- Records of 100 bytes long, two classes of computers ($k = 1$ and $k = 1.5$);
- 54 GB of data, 50 runs for each experiment, bi-opteron processor, cpu-burning;
- 96 nodes used;
- Minute Sort benchmark compliant;

Experiments

- Records of 100 bytes long, two classes of computers ($k = 1$ and $k = 1.5$);
- 54 GB of data, 50 runs for each experiment, bi-opteron processor, cpu-burning;
- 96 nodes used;
- Minute Sort benchmark compliant;

naive algo	partitioning	partitioning (2 threads)
125.4s	112.7s	69.4s

Summary

- Polynomial complexity functions yield a simple formula

$$n_i = \frac{\tilde{f}^{-1}(k_i)}{\sum_{i=1}^{p} \tilde{f}^{-1}(k_i)} N.$$

Summary

- Polynomial complexity functions yield a simple formula

$$n_i = \frac{\tilde{f}^{-1}(k_i)}{\sum_{i=1}^{p} \tilde{f}^{-1}(k_i)} N.$$

- Unknown complexity functions can still be managed, but require incremental construction;

Summary

- Polynomial complexity functions yield a simple formula

$$n_i = \frac{\tilde{f}^{-1}(k_i)}{\sum_{i=1}^{p} \tilde{f}^{-1}(k_i)} N.$$

- Unknown complexity functions can still be managed, but require incremental construction;
- Dynamic programming can also be used in more general cases.

Summary

- Polynomial complexity functions yield a simple formula

$$n_i = \frac{\tilde{f}^{-1}(k_i)}{\sum_{i=1}^{p} \tilde{f}^{-1}(k_i)} N.$$

- Unknown complexity functions can still be managed, but require incremental construction;
- Dynamic programming can also be used in more general cases.

- Future work
  - Limited bandwidth models and heterogeneous network links.
  - Non-linear computation time models.
  - Global optimisation.