A Fully Distributed Algorithm for Multiple Bag-of-tasks Application Scheduling on Grids

Rémi Bertin, Arnaud Legrand, Corinne Touati
INRIA, CNRS, INRIA

Laboratoire LIG, Grenoble, France

October 26, 2008
Outline

1. Framework
2. Lagrangian Optimization
3. Simulations
Large-scale distributed computing platforms result from the collaboration of many users:

- Sharing resources amongst users should somehow be fair.
Large-scale distributed computing platforms result from the collaboration of many users:

- **Sharing** resources amongst users should somehow be fair.
- The size of these systems prevents the use of centralized approaches \(\leadsto\) need for **distributed** scheduling.
Large-scale distributed computing platforms result from the collaboration of many users:

- **Sharing** resources amongst users should somehow be fair.
- The size of these systems prevents the use of centralized approaches $\rightsquigarrow$ need for **distributed** scheduling.
- Task **regularity** (SETI@home, BOINC, . . .) $\rightsquigarrow$ **steady-state** scheduling.
Large-scale distributed computing platforms result from the collaboration of many users:

- **Sharing** resources amongst users should somehow be fair.
- The size of these systems prevents the use of centralized approaches \( \leadsto \) need for **distributed** scheduling.
- Task **regularity** (SETI@home, BOINC, \ldots) \( \leadsto \) steady-state scheduling.

Designing a **Fair and Distributed** scheduling algorithm for this framework.
Outline

1. Framework
2. Lagrangian Optimization
3. Simulations
Platform Model

▶ General platform graph $G = (N, E, W, B)$. 

C. Touati (INRIA)
- General platform graph $G = (N, E, W, B)$.
- Speed of $P_n \in N$: $W_n$ (in MFlops/s).
- Bandwidth of $(P_i \rightarrow P_j)$: $B_{i,j}$ (in MB/s).
- Linear-cost communication and computation model: $X/B_{i,j}$ time units to send a message of size $X$ from $P_i$ to $P_j$. 
- Communications and computations can be overlapped.
- Multi-port communication model.
General platform graph $G = (N, E, W, B)$.

- Speed of $P_n \in N$: $W_n$ (in MFlops/s).
- Bandwidth of $(P_i \rightarrow P_j)$: $B_{i,j}$ (in MB/s).
General platform graph $G = (N, E, W, B)$.

- Speed of $P_n \in N$: $W_n$ (in MFlops/s).
- Bandwidth of $(P_i \rightarrow P_j)$: $B_{i,j}$ (in MB/s).
- Linear-cost communication and computation model: $X/B_{i,j}$ time units to send a message of size $X$ from $P_i$ to $P_j$. 

Communications and computations can be overlapped.

Multi-port communication model.
▼ General platform graph $G = (N, E, W, B)$.
▼ Speed of $P_n \in N$: $W_n$ (in MFlops/s).
▼ Bandwidth of $(P_i \rightarrow P_j)$: $B_{i,j}$ (in MB/s).
▼ Linear-cost communication and computation model: $X/B_{i,j}$ time units to send a message of size $X$ from $P_i$ to $P_j$.
▼ Communications and computations can be overlapped.
General platform graph $G = (N, E, W, B)$.

Speed of $P_n \in N$: $W_n$ (in MFlops/s).

Bandwidth of $(P_i \rightarrow P_j)$: $B_{i,j}$ (in MB/s).

Linear-cost communication and computation model: $X/B_{i,j}$ time units to send a message of size $X$ from $P_i$ to $P_j$.

Communications and computations can be overlapped.

Multi-port communication model.
Multiple applications:

- A set $A$ of $K$ applications $A_1, \ldots, A_K$. 

![Diagram showing $A_1$, $A_2$, and $A_3$]
Multiple applications:

- A set $A$ of $K$ applications $A_1, \ldots, A_K$.

- Each consisting in a large number of same-size independent tasks $\sim$ each application is defined by a communication cost $w_k$ (in MFlops) and a communication cost $b_k$ (in MB).
Multiple applications:

- A set $A$ of $K$ applications $A_1, \ldots, A_K$.
- Each consisting in a large number of same-size independent tasks $\sim$ each application is defined by a communication cost $w_k$ (in MFlops) and a communication cost $b_k$ (in MB).
- Different communication and computation demands for different applications.
Each application originates from a master node $P_{m(k)}$ that initially holds all the input data necessary for each application $A_k$. 
Hierarchical Deployment

▶ Each application originates from a master node $P_{m(k)}$ that initially holds all the input data necessary for each application $A_k$.

▶ Communication are only required outwards from the master nodes: the amount of data returned by the worker is negligible.
Each application originates from a master node $P_{m(k)}$ that initially holds all the input data necessary for each application $A_k$.

Communication are only required outwards from the master nodes: the amount of data returned by the worker is negligible.

Each application $A_k$ is deployed on the platform as a tree.

Therefore if an application $k$ wants to use a node $P_n$, all its data will use a single path from $P_{m(k)}$ to $P_n$ denoted by $(P_{m(k)} \sim P_n)$. 
All tasks of a given application are identical and independent;
we do not really need to care about where and when (as opposed to classical scheduling problems).
All tasks of a given application are identical and independent; we do not really need to care about where and when (as opposed to classical scheduling problems).

We only need to focus on average values in steady-state.

Steady-state values:

- Variables: average number of tasks of type $k$ processed by processor $n$ per time unit: $\varrho_{n,k}$.
- Throughput of application $k$: $\varrho_k = \sum_{n \in N} \varrho_{n,k}$.

Theorem 1. From "feasible" $\varrho_{n,k}$, it is possible to build an optimal periodic infinite schedule (i.e., whose steady-state rates are exactly the $\varrho_{n,k}$).

Such a schedule is asymptotically optimal for the makespan.
All tasks of a given application are identical and independent;
we do not really need to care about where and when (as opposed to classical scheduling problems).

We only need to focus on average values in steady-state.

Steady-state values:

Theorem 1. From “feasible” \( \rho_{n,k} \), it is possible to build an optimal periodic infinite schedule (i.r. whose steady-state rates are exactly the \( \rho_{n,k} \)).

Such a schedule is asymptotically optimal for the makespan.
All tasks of a given application are identical and independent; we do not really need to care about where and when (as opposed to classical scheduling problems).

We only need to focus on average values in steady-state.

Steady-state values:
- Variables: average number of tasks of type \( k \) processed by processor \( n \) per time unit: \( \rho_{n,k} \).

Theorem 1.
From “feasible” \( \rho_{n,k} \), it is possible to build an optimal periodic infinite schedule (i.r. whose steady-state rates are exactly the \( \rho_{n,k} \)). Such a schedule is asymptotically optimal for the makespan.
All tasks of a given application are **identical and independent**; we do not really need to care about *where* and *when* (as opposed to classical scheduling problems).

We only need to focus on **average values in steady-state**.

**Steady-state values:**

- **Variables:** average number of tasks of type $k$ processed by processor $n$ per time unit: $\varrho_{n,k}$.
- **Throughput** of application $k$: $\varrho_k = \sum_{n \in N} \varrho_{n,k}$.

**Theorem 1.** From "feasible" $\varrho_{n,k}$, it is possible to build an optimal periodic infinite schedule (i.r. whose steady-state rates are exactly the $\varrho_{n,k}$).

Such a schedule is asymptotically optimal for the makespan.
All tasks of a given application are identical and independent; we do not really need to care about where and when (as opposed to classical scheduling problems).

We only need to focus on average values in steady-state.

Steady-state values:

- Variables: average number of tasks of type \( k \) processed by processor \( n \) per time unit: \( \varrho_{n,k} \).
- Throughput of application \( k \): \( \varrho_k = \sum_{n \in N} \varrho_{n,k} \).

**Theorem 1.**

From “feasible” \( \varrho_{n,k} \), it is possible to build an optimal periodic infinite schedule (i.r. whose steady-state rates are exactly the \( \varrho_{n,k} \)). Such a schedule is asymptotically optimal for the makespan.
Let $U_k(\varrho_k)$ be the utility associated to application $k$. We aim at maximizing $\sum_{k \in K} U_k(\varrho_k)$. It has been shown that different values of $U_k(\varrho_k)$ lead to different kinds of fairness. Typically, $U_k(\varrho_k) = \frac{\varrho_k}{(1 - \alpha)}$ ($\alpha$-fairness).

Maximize $\sum_k \log(\varrho_k)$ under the constraints:

- $\varrho_k = \sum_n \varrho_{n,k}$ for all $n$,
- $\sum_k \varrho_{n,k} w_k \leq W_n$ for all $(P_i \rightarrow P_j)$,
- $\sum_k \sum_n$ such that $(P_i \rightarrow P_j) \in (P_m(k); P_n)$ $\varrho_{n,k} b_k \leq B_{i,j}$

Can be solved in polynomial time with semi-definite programming [Touati.et.al.06]. It is very centralized though. Can we solve it in a distributed way?
Let $U_k(\varrho_k)$ be the utility associated to application $k$. We aim at maximizing $\sum_{k \in K} U_k(\varrho_k)$.

It has been shown that different values of $U_k$ leads to different kind of fairness. Typically, $U_k(\varrho_k) = \log(\varrho_k)$ (proportional fairness) or $U_k(\varrho_k) = \varrho_k^\alpha/(1 - \alpha)$ ($\alpha$-fairness).
Utility and Optimization Problem

- Let $U_k(\varphi_k)$ be the utility associated to application $k$. We aim at maximizing $\sum_{k \in K} U_k(\varphi_k)$.

- It has been shown that different values of $U_k$ leads to different kind of fairness. Typically, $U_k(\varphi_k) = \log(\varphi_k)$ (proportional fairness) or $U_k(\varphi_k) = \varphi_k^\alpha/(1 - \alpha)$ ($\alpha$-fairness).
Utility and Optimization Problem

Let $U_k(\varrho_k)$ be the utility associated to application $k$. We aim at maximizing $\sum_{k \in K} U_k(\varrho_k)$.

It has been shown that different values of $U_k$ leads to different kind of fairness. Typically, $U_k(\varrho_k) = \log(\varrho_k)$ (proportional fairness) or $U_k(\varrho_k) = \varrho_k^\alpha/(1 - \alpha)$ (\(\alpha\)-fairness).

Maximize $\sum_k \log(\varrho_k)$ under the constraints:

$$\varrho_k = \sum_n \varrho_{n,k}$$

$$\forall n, \sum_k \varrho_{n,k} w_k \leq W_n$$

$$\forall (P_i \rightarrow P_j), \sum_k \sum_{n \text{ such that } (P_i \rightarrow P_j) \in (P_m(k) \sim P_n)} \varrho_{n,k} b_k \leq B_{i,j}$$
Utility and Optimization Problem

Let $U_k(q_k)$ be the utility associated to application $k$. We aim at maximizing $\sum_{k \in K} U_k(q_k)$.

It has been shown that different values of $U_k$ leads to different kind of fairness. Typically, $U_k(q_k) = \log(q_k)$ (proportional fairness) or $U_k(q_k) = q_k^\alpha/(1 - \alpha)$ ($\alpha$-fairness).

Maximize $\sum_k \log(q_k)$ under the constraints:

\[
q_k = \sum_n q_{n,k}
\]
\[
\forall n, \quad \sum_k q_{n,k} w_k \leq W_n
\]
\[
\forall (P_i \rightarrow P_j), \quad \sum_k \sum_{n \text{ such that } (P_i \rightarrow P_j) \in (P_m(k) \sim P_n)} q_{n,k} b_k \leq B_{i,j}
\]

Can be solved in polynomial time with semi-definite programming [Touati.et.al.06]. It is very centralized though.

Can we solve it in a distributed way?
Let $U_k(\varrho_k)$ be the utility associated to application $k$. We aim at maximizing $\sum_{k \in K} U_k(\varrho_k)$.

It has been shown that different values of $U_k$ lead to different kinds of fairness. Typically, $U_k(\varrho_k) = \log(\varrho_k)$ (proportional fairness) or $U_k(\varrho_k) = \varrho_k^\alpha / (1 - \alpha)$ ($\alpha$-fairness).

Maximize $\sum_k \log(\varrho_k)$ under the constraints:

\[
\varrho_k = \sum_n \varrho_{n,k}
\]
\[
\forall n, \quad \sum_k \varrho_{n,k} w_k \leq W_n
\]
\[
\forall (P_i \to P_j), \quad \sum_k \sum_{n \text{ such that } (P_i \to P_j) \in (P_{m(k)} \sim P_n)} \varrho_{n,k} b_k \leq B_{i,j}
\]

Can be solved in polynomial time with semi-definite programming [Touati.et.al.06]. It is very centralized though.

Can we solve it in a distributed way?
Outline

1. Framework
2. Lagrangian Optimization
3. Simulations
Lagrangian Optimization: Basics

- Designed to solve non linear optimization problems:
  - Let $\alpha \rightarrow f(\alpha)$ be a function to maximize.
  - Let $(C_i(\alpha) \geq 0)_{i \in [1..n]}$ be a set of $n$ constraints.
  - We wish to solve:

\[
(P) \begin{cases} 
\text{maximize } f(\alpha) \\
\forall i \in [1..n], C_i(\alpha) \geq 0, \text{ and } \alpha \geq 0
\end{cases}
\]
Lagrangian Optimization: Basics

- Designed to solve non linear optimization problems:
  - Let $\alpha \to f(\alpha)$ be a function to maximize.
  - Let $(C_i(\alpha) \geq 0)_{i \in [1..n]}$ be a set of $n$ constraints.
  - We wish to solve:

$$\begin{align*}
(P) \quad \text{maximize } f(\alpha) \\
\forall i \in [1..n], C_i(\alpha) \geq 0, \text{ and } \alpha \geq 0
\end{align*}$$

- The Lagrangian function: $\mathcal{L}(\alpha, \lambda) = f(\alpha) - \sum_{i \in [1..n]} \lambda_i C_i(\alpha)$. 
Lagrangian Optimization: Basics

- Designed to solve non linear optimization problems:
  - Let \( \alpha \rightarrow f(\alpha) \) be a function to maximize.
  - Let \( (C_i(\alpha) \geq 0)_{i\in[1..n]} \) be a set of \( n \) constraints.
  - We wish to solve:

\[
\begin{aligned}
(P) \quad & \text{maximize } f(\alpha) \\
& \forall i \in [1..n], C_i(\alpha) \geq 0, \text{ and } \alpha \geq 0
\end{aligned}
\]

- The **Lagrangian function**: \( \mathcal{L}(\alpha, \lambda) = f(\alpha) - \sum_{i \in [1..n]} \lambda_i C_i(\alpha) \).

- The **dual functional**: \( d(\lambda) = \max_{\alpha \geq 0} \mathcal{L}(\alpha, \lambda) \).
Lagrangian Optimization: Basics

- Designed to solve non linear optimization problems:
  - Let $\alpha \rightarrow f(\alpha)$ be a function to maximize.
  - Let $(C_i(\alpha) \geq 0)_{i \in [1..n]}$ be a set of $n$ constraints.
  - We wish to solve:
    \[
    (P) \begin{cases}
      \text{maximize } f(\alpha) \\
      \forall i \in [1..n], C_i(\alpha) \geq 0, \text{ and } \alpha \geq 0
    \end{cases}
    \]

- The Lagrangian function: $L(\alpha, \lambda) = f(\alpha) - \sum_{i \in [1..n]} \lambda_i C_i(\alpha)$.

- The dual functional: $d(\lambda) = \max_{\alpha \geq 0} L(\alpha, \lambda)$.

- Under some weak hypothesis, solving $(P)$ is equivalent to solve the dual problem:
  \[
  (D) \begin{cases}
    \text{minimize } d(\lambda) \\
    \lambda \geq 0
  \end{cases}
  \]
Lagrangian Optimization: Basics

- Designed to solve non linear optimization problems:
  - Let $\alpha \rightarrow f(\alpha)$ be a function to maximize.
  - Let $(C_i(\alpha))_{i \in [1..n]}$ be a set of $n$ constraints.

  We wish to solve:
  $$(P) \begin{cases} \maximize f(\alpha) \\ \forall i \in [1..n], C_i(\alpha) \geq 0, \alpha \geq 0 \end{cases}$$

  - The Lagrangian function:
    $$L(\alpha, \lambda) = f(\alpha) - \sum_{i \in [1..n]} \lambda_i C_i(\alpha).$$

  - The dual functional:
    $$d(\lambda) = \max_{\alpha \geq 0} L(\alpha, \lambda).$$

  Under some weak hypothesis, solving $(P)$ is equivalent to solve the dual problem:
  $$(D) \begin{cases} \minimize d(\lambda) \\ \lambda \geq 0 \end{cases}$$

So what?..

- Two coupled problems with simple constraints.
- The structure of constraints is transposed to $(D)$ and a gradient descent algorithm is a natural way to solve these two problems.
- This technique has been used successfully for network resource sharing [Kelly.98], TCP analysis [Low.03], flow control in multi-path network [Hang.et.al.03], ...
What does the Lagrangian function look like?

\[
\mathcal{L}(\varrho, \lambda, \mu) = \sum_{k \in K} \log \left( \sum_i \varrho_{i,k} \right) + \sum_i \lambda_i \left( W_i - \sum_k \varrho_{i,k} w_k \right) + \sum_{(P_i \to P_j)} \mu_{i,j} \left( B_{i,j} - \sum_k \sum_{n \text{ such that } (P_i \to P_j) \in (P_{m(k)} \sim P_n)} \varrho_{n,k} b_k \right)
\]
What does the Lagrangian function look like?

\[ \mathcal{L}(\rho, \lambda, \mu) = \sum_{k \in K} \log \left( \sum_i \rho_{i,k} \right) + \sum_i \lambda_i \left( W_i - \sum_k \rho_{i,k} w_k \right) \]

\[ + \sum_{(P_i \rightarrow P_j)} \mu_{i,j} \left( B_{i,j} - \sum_k \sum_{n \text{ such that } (P_i \rightarrow P_j) \in (P_{m(k)} \sim P_n)} \rho_{n,k} b_k \right) \]

We want to compute \( \min_{\lambda, \mu \geq 0} \max_{\rho \geq 0} \mathcal{L}(\alpha, \lambda, \mu) \).

We then just need to do a gradient descent on \( \lambda, \mu \) and \( \rho \)…
First bug: The Lagrangian is strictly convex in each $Q_k$ but not in the $Q_{i,k}$

\[
\mathcal{L}(\varrho, \lambda, \mu) = \sum_{k \in K} \log \left( \sum_i Q_{i,k} \right) + \sum_i \lambda_i \left( W_i - \sum_k Q_{i,k} w_k \right) \\
+ \sum_{(P_i \rightarrow P_j)} \mu_{i,j} \left( B_{i,j} - \sum_k \sum_{n \text{ such that } (P_i \rightarrow P_j) \in (P_m(k) \sim P_n)} Q_{n,k} b_k \right)
\]
First bug: The Lagrangian is strictly convex in each $\varrho_k$ but not in the $\varrho_{i,k}$

$$
\mathcal{L}(\varrho, \lambda, \mu) = \sum_{k \in K} \log \left( \sum_i \varrho_{i,k} \right) + \sum_i \lambda_i \left( W_i - \sum_k \varrho_{i,k} w_k \right) \\
+ \sum_{(P_i \to P_j)} \mu_{i,j} \left( B_{i,j} - \sum_k \sum_{n \text{ such that } (P_i \to P_j) \in (P_{m(k)} \sim P_n)} \varrho_{n,k} b_k \right)
$$

- Solution: proximal optimization technique

$$
- \sum_k \sum_i \frac{c}{2} (\varrho_{i,k} - \tilde{\varrho}_{i,k})^2
$$

→ new problem:

$$
\max_{\tilde{\varrho} \geq 0} \max_{\varrho \geq 0} \min_{\mu \geq 0} L(\tilde{\varrho}, \varrho, \mu)
$$
\[ \max_{\tilde{\varrho} \geq 0} \max_{\varrho \geq 0} \min_{\mu \geq 0} L(\tilde{\varrho}, \varrho, \mu) \]

Solution: Arrow-Hurwicz gradient method with simultaneous update steps
Then, there are some tight bounds on the update steps to guarantee convergence...
\(\lambda_i\) and \(\mu_{i,j}\) are called shadow variables or shadow prices. They can naturally be thought of as the price to pay to use the corresponding resource.

A gradient descent algorithm on the primal-dual can thus be seen as a bargain between applications and resources.

We need to find an efficient way to implement this bargain, i.e., to compute the update. To this end, the following quantities are useful and easy to compute via recursive propagation:

\[
\begin{align*}
\sigma_k^n &= \sum_{p \text{ such that } n \in (P_{m(k)} \sim P_p)} q_{p,k} \\
\eta_k^n &= \sum_{(P_i \rightarrow P_j) \in (P_{m(k)} \sim P_n)} \mu_{i,j}
\end{align*}
\]

\(\sigma_k^n\) \{aggregate throughput of a subtree.\}

\(\eta_k^n\) \{aggregate communication price\}
\( \lambda_i \) and \( \mu_{i,j} \) are called shadow variables or shadow prices. They can naturally be thought of as the price to pay to use the corresponding resource.

A gradient descent algorithm on the primal-dual can thus be seen as a bargain between applications and resources.

We need to find an efficient way to implement this bargain, i.e., to compute the update. To this end, the following quantities are useful and easy to compute via recursive propagation:

\[
\begin{align*}
\sigma_n^k & = \sum_{p \text{ such that } n \in (P_m^k; P_p)} \rho_{p,k} \\
\eta_{n,k} & = \sum_{(P_i \rightarrow P_j) \in (P_m^k; P_n)} \mu_{i,j}
\end{align*}
\]

Hierarchical deployment
\( \lambda_i \) and \( \mu_{i,j} \) are called shadow variables or shadow prices. They can naturally be thought of as the price to pay to use the corresponding resource.

A gradient descent algorithm on the primal-dual can thus be seen as a bargain between applications and resources.

We need to find an efficient way to implement this bargain, i.e., to compute the update. To this end, the following quantities are useful and easy to compute via recursive propagation:

Hierarchical deployment

\[
\sigma_k^i = \sum_{p \text{ such that } n \in (P_m(k); P_p)} \rho_{p,k}
\]

\[
\eta_n^k = \sum_{(P_i \rightarrow P_j) \in (P_m(k); P_n)} \mu_{i,j}
\]
\( \lambda_i \) and \( \mu_{i,j} \) are called shadow variables or shadow prices. They can naturally be thought of as the price to pay to use the corresponding resource.

A gradient descent algorithm on the primal-dual can thus be seen as a bargain between applications and resources.

We need to find an efficient way to implement this bargain, i.e., to compute the update. To this end, the following quantities are useful and easy to compute via recursive propagation:

\[
\eta_{n,k} = \sum_{p \text{ such that } n \in (P_m(k); P_p)} \rho_{p,k}
\]

aggregate throughput of a subtree.

\[
\eta_{i,k} = \sum_{(P_i \rightarrow P_j) \in (P_i \rightarrow P_j)} \rho_{i,j}
\]

aggregate communication price.

Hierarchical deployment
Prices and rates can thus be propagated and aggregated to perform the following updates ($x(t+1) \leftarrow x(t) - \gamma \nabla F(x(t))$):

- $p_k^i(t+1) \leftarrow b_k \eta_k^i(t) + w_k \lambda_i(t)$
- $q_k(t+1) \leftarrow \sigma_k^{m(k)}(t+1)$
- $q_{i,k}(t+1) \leftarrow \left[ q_{i,k}(t) + \gamma_q (U_k'(q_k(t)) - p_k^i(t)) \right]^+$
- $\lambda_i(t+1) \leftarrow \left[ \lambda_i(t) + \gamma_\lambda \left( \sum_k w_k q_{i,k} - W_i \right) \right]^+$
- $\mu_{i,j}(t+1) \leftarrow \left[ \mu_{i,j}(t) + \gamma_\mu \left( \sum_k b_k \sigma_k^i - B_{i,j} \right) \right]^+$

- This algorithm is fully distributed and converges to the optimal solution provided a good choice of $\gamma_q$, $\gamma_\lambda$ and $\gamma_\mu$ is done.
- This algorithm seamlessly adapts to application/node arrival and to load variations.
Experimental Setting

- The simulator is SimGrid.
Experimental Setting

- The simulator is SimGrid.
- Fully synchronous gradient.

We used three kinds of applications of respective \((b, w)\): \((1000, 5000)\), \((2000, 800)\), and \((1500, 1500)\).
Experimental Setting

- The simulator is SimGrid.
- Fully synchronous gradient.
- Checking the correctness of the results using semi-definite programming.
Experimental Setting

- The simulator is SimGrid.
- Fully synchronous gradient.
- Checking the correctness of the results using semi-definite programming.
- Very simple platform and applications:

We used three kinds of applications of respective \((b, w)\): (1000, 5000), (2000, 800), and (1500, 1500).
Objective function $\sum_k \log \rho_k$: numerical instabilities and global inefficiencies.
Objective function $\sum_k \log \varrho_k$: using a smaller steps $\gamma_\varrho \sim$ no more instability but slow convergence.
Throughput of each of the three applications: between two iterations, a decrease or increase of magnitude five or more can happen!
Basic Version of the Algorithm

Detailing the rates for application 1.
Correlation between the rate of an application on a given node and the price it experiences.
The original update equation for $\varrho$ is:

$$
\varrho_{i,k}(t+1) \leftarrow \left[ \varrho_{i,k}(t) + \gamma_{\varrho} \left( \frac{1}{\varrho_k(t)} - p_{i,k}(t) \right) \right]^+
$$

A small value of $\varrho$ leads to huge updates and thus to severe oscillations.
The original update equation for $\varrho$ is:

$$
\varrho_{i,k}(t+1) \leftarrow \left[ \varrho_{i,k}(t) + \gamma_\varrho \left( \frac{1}{\varrho_k(t)} - p^i_k(t) \right) \right]^+
$$

A small value of $\varrho$ leads to huge updates and thus to severe oscillations. This is a known issue and, one can normalize as follows [Hang.et.al.03]:

$$
\varrho_{i,k}(t+1) \leftarrow \left[ \varrho_{i,k}(t) + \gamma_\varrho \left( 1 - \varrho_k(t).p^i_k(t) \right) \right]^+.
$$

Unfortunately, this merely avoids division by 0 but is not sufficient to damp oscillations.
The original update equation for $\varrho$ is:

$$
\varrho_{i,k}(t+1) \leftarrow \left[ \varrho_{i,k}(t) + \gamma_{\varrho} \left( \frac{1}{\varrho_{k}(t)} - p_{k}^{i}(t) \right) \right]^{+}
$$

A small value of $\varrho$ leads to huge updates and thus to severe oscillations. This is a known issue and, one can normalize as follows [Hang.et.al.03]:

$$
\varrho_{i,k}(t+1) \leftarrow \left[ \varrho_{i,k}(t) + \gamma_{\varrho} \left( 1 - \varrho_{k}(t) \cdot p_{k}^{i}(t) \right) \right]^{+}.
$$

Unfortunately, this merely avoids division by 0 but is not sufficient to damp oscillations.

Updating $\varrho$ has an impact on the prices $\lambda$ and $\mu$, which in turn impact on the $\varrho$'s update.
The original update equation for $\rho$ is:

$$
\rho_{i,k}(t+1) \leftarrow \left[ \rho_{i,k}(t) + \gamma \rho \left( \frac{1}{\rho_k(t)} - p^i_k(t) \right) \right]^+
$$

A small value of $\rho$ leads to huge updates and thus to severe oscillations. This is a known issue and, one can normalize as follows [Hang.et.al.03]:

$$
\rho_{i,k}(t+1) \leftarrow \left[ \rho_{i,k}(t) + \gamma \rho \left( 1 - \rho_k(t) \cdot p^i_k(t) \right) \right]^+.
$$

Unfortunately, this merely avoids division by 0 but is not sufficient to damp oscillations.

Updating $\rho$ has an impact on the prices $\lambda$ and $\mu$, which in turn impact on the $\rho$’s update. The second update of $\rho$ should have the same order of magnitude (or be smaller) as the first one to avoid numerical instabilities that prevent convergence of the algorithm.
Assume that we have reached the equilibrium. Then increase $\lambda_i$ by $\Delta \lambda_i$. Then:

$$\Delta Q_{i,k} = -\gamma^{(2)} Q_w \Delta \lambda_i Q_k.$$
Scaling Again!

Assume that we have reached the equilibrium. Then increase $\lambda_i$ by $\Delta \lambda_i$. Then:

$$\Delta Q_{i,k} = -\gamma_{Q}^{(2)} w_k \Delta \lambda_i Q_k.$$ 

In turn, such a variation incurs a variation of $\lambda_i$:

$$\sum_k \gamma \lambda \cdot w_k \cdot \Delta Q_{i,k} = \Delta \lambda_i \cdot \left( \sum_k \gamma \lambda \cdot \gamma_{Q}^{(2)} w_k^2 \cdot Q_k \right).$$
Assume that we have reached the equilibrium. Then increase $\lambda_i$ by $\Delta \lambda_i$. Then:

$$\Delta \varrho_{i,k} = -\gamma \varrho_w^2 \Delta \varrho_k.$$ 

In turn, such a variation incurs a variation of $\lambda_i$:

$$\sum_k \gamma \lambda \cdot w_k \cdot \Delta \varrho_{i,k} = \Delta \lambda_i \cdot \left( \sum_k \gamma \lambda \cdot \gamma \varrho_w^2 w_k \varrho_k \right).$$

Thus, the solution of our gradient is stable only if

$$\sum_k \gamma \lambda \cdot \gamma \varrho_w^2 w_k \varrho_k < 1.$$ 

Therefore, $\lambda$’s update should be replaced by

$$\lambda_i(t + 1) \leftarrow \left[ \lambda_i(t) + \gamma \lambda \sum_k w_k \varrho_{i,k} - W_i \right]^+$$

It doesn’t hurt and similar scaling can be done for the $\mu$’s.
The oscillations, due to a really badly chosen initialization value quickly vanish (left graph).
The algorithm almost instantly reaches a decent value (5% of the optimal value after 17 iterations), and relatively quickly to a good value (1% of the optimal value after 83 iterations) (right plot).
High number of iterations: after 498 iterations, the performance remains higher than 99.5% of the optimal and still further increase with the number of iterations.
Convergence of $\varrho_i$ with $i = 1..3$: no more oscillations occur. The throughput of each application slowly converges to their “optimal” values.
Prices evolve smoothly. As the number of iterations increases, they converge to their optimal value while remaining positive, meaning that the resources they refer to is neither under utilized nor over-loaded.
Applications with parameters of different order of magnitude

→ different values of the step sizes should be used, making the algorithm difficult to tune for real large size platforms...

Newton Descent Algorithm:

\[ x(t + 1) \leftarrow x(t) - \gamma \left( \nabla^2 F(x(t)) \right)^{-1} \cdot \nabla F(x(t)) \]

\[ \varrho_{i,k}(t+1) \leftarrow \left[ (1 - \gamma^{(1)}_\varrho) \varrho_{i,k}(t) + \gamma^{(1)}_\varrho \tilde{\varrho}_{is,k}(t) + \gamma^{(2)}_\varrho \left( 1 - \varrho_k(t) \cdot p^i_k(t) \right) \varrho_k(t) \right]^+ \]

\[ \lambda_i(t + 1) \leftarrow \left[ \lambda_i(t) + \gamma \lambda \frac{\sum_k w_k \varrho_{i,k}(t) - W_i}{\sum_k \text{s.t. } \varrho_{i,k} > 0 w_k^2 \varrho_k^2(t)} \right]^+ \]

\[ \mu_{i,j}(t + 1) \leftarrow \left[ \mu_{i,j}(t) + \gamma \mu \frac{\sum_k b_k \sigma^i_k(t) - B_{i,j}}{\sum_k \sum_n \text{s.t. } \varrho_{n,k} > 0 \text{ and } (P_i \rightarrow P_j) \in (P_{m(k)} \sim P_n) b_k^2 \varrho_k^2(t)} \right]^+ \]
The real thing...

<table>
<thead>
<tr>
<th>variable</th>
<th>$\gamma_{o}$</th>
<th>$\gamma_{e}^{(1)}$</th>
<th>$\gamma_{e}^{(2)}$</th>
<th>$\gamma_{\lambda}$</th>
<th>$\gamma_{\mu}$</th>
<th>$\phi_{\text{init}}$</th>
<th>$\lambda_{\text{init}}$</th>
<th>$\mu_{\text{init}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.05</td>
<td>0.01</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
</tr>
</tbody>
</table>

3 applications:

- Intensive computation application: each task is the multiplication of two squared matrices of size $t = 3500$. Hence $b_1 = 8 \times 2 \times t^2 = 196.10^6$ bits and $w_1 = t^3 = 42875.10^6$ flops.

- A “large” application, the sum of two squared matrices, also of size $t$: $b_2 = b_1 = 196.10^6$ bits and $w_2 = t^2 = 12.25.10^6$ flops.

- An intermediate application: sorting a vector of size $t' = 1,000,000$: $b_3 = 8 \times 1,000,000 = 8.10^6$ bits and $w_3 = t'.\log(t') = 13, 81.10^6$ flops.
A Word on Stability

Objective value vs. Iteration for a Distributed algorithm and Optimal values before and after removal.
A Word on Stability
A Word on Stability
This approach is very inspired by Low’s work [Hang.et.al.03] on flow control in multi-path network.

The setting (BoT applications, grids) is different though and new problems arise.

The resulting algorithms are different (few sources and many sinks here).

Appropriate scaling mechanisms allow the study of realistic, large scale platforms and applications.

Future work deals with the implementation of asynchronous descent gradients and convergence speed analysis.

