2008

Optimizing Network Performance of Computing Pipelines in Distributed Environments

Qishi Wu
University of Memphis

Yi Gu
University of Memphis

Mengxia Zhu
Southern Illinois University Carbondale, mzhu@cs.siu.edu

Nageswara S.V. Rao
Oak Ridge National Laboratory

Published in Wu, Q., Gu, Y., Zhu, M., & Rao, N.S.V. (2008). Optimizing network performance of computing pipelines in distributed environments. IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008, 1-8. doi: 10.1109/IPDPS.2008.4536465

Recommended Citation

Wu, Qishi, Gu, Yi, Zhu, Mengxia and Rao, Nageswara S. "Optimizing Network Performance of Computing Pipelines in Distributed Environments." (Jan 2008).

Optimizing Network Performance of Computing Pipelines in Distributed Environments

Qishi Wu, Yi Gu
Dept of Computer Science
University of Memphis
Memphis, TN 38152
{qishiwu,yigu}@memphis.edu

Mengxia Zhu
Dept of Computer Science
Southern Illinois University
Carbondale, IL 62901
mzhu@cs.siu.edu

Nageswara S.V. Rao
Computer Science & Math Div
Oak Ridge National Laboratory
Oak Ridge, TN 37831
raons@ornl.gov

Abstract

Supporting high performance computing pipelines over wide-area networks is critical to enabling large-scale distributed scientific applications that require fast responses for interactive operations or smooth flows for data streaming. We construct analytical cost models for computing modules, network nodes, and communication links to estimate the computing times on nodes and the data transport times over connections. Based on these time estimates, we present the Efficient Linear Pipeline Configuration method based on dynamic programming that partitions the pipeline modules into groups and strategically maps them onto a set of selected computing nodes in a network to achieve minimum end-to-end delay or maximum frame rate. We implemented this method and evaluated its effectiveness with experiments on a large set of simulated application pipelines and computing networks. The experimental results show that the proposed method outperforms the Streamline and Greedy algorithms. These results, together with polynomial computational complexity, make our method a potential scalable solution for large practical deployments.

1 Introduction

The demands of large-scale collaborative applications in various scientific, engineering, medical, and business domains are beyond the capabilities of the traditional solutions based on standalone workstations. These applications typically involve distributed compute-intensive tasks of ever-increasing complexity that require pooling globally-dispersed resources to produce unprecedented data collections, simulations, visualizations, and analysis. In recent years, a wide variety of system resources including supercomputers, data repositories, computing facilities, network infrastructures, storage systems, and display devices have been increasingly developed and deployed around the globe. Such resources are typically shared over the Internet or dedicated connections, and must be optimally scheduled to account for their availability, utilization, capacity, and performance. Optimizing the network performance of a complex computing task in distributed environments is crucial to improving both the utilization of expensive system resources and the productivity of application end users.

We consider two types of large-scale computing applications with linear workflows or pipelines comprising of a number of modules or subtasks that are to be executed in a sequential manner in a distributed network environment:

1. We first consider interactive applications where a single dataset is sequentially processed along a computing pipeline. A typical example is an interactive parameter update on a remote visualization system that triggers a sequence of processing subtasks for data filtering, isosurface extraction, geometry rendering, image compositing, and final display [13].
2. We consider streaming applications where a series of datasets continuously flow through a computing pipeline. Typical examples include a video-based real-time monitoring system for detecting criminal suspects at an entrance that performs feature extraction and detection, facial reconstruction, pattern recognition, data mining, and identity matching on images that are continuously captured.

For interactive applications, we aim to minimize the end-to-end delay of a pipeline to provide fast response, while for streaming applications, our goal is to maximize the frame rate of a pipeline to achieve smooth data flow [1].

The application performance in terms of end-to-end delay or frame rate is determined by the computing times of modules running on a network node and the transport

[1] In some contexts, the frame rate is also referred to as throughput, i.e. the number of final data items produced or transferred in a unit of time.

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

times incurred over communication links. Due to the disparate characteristics of data sources, computing modules, network nodes, and communication links, a common design goal is to optimally map the modules of a computing pipeline onto a set of strategically selected network nodes for execution. Such a mapping scheme must account for the temporal constraints in the form of linear execution order of computing modules and spatial constraints in the form of geographical distribution of network nodes and their connectivity. Note that conventional scheduling methods is mainly focused on the temporal aspects of the modules sharing multiple instances of the resource of the same type.

The mapping and scheduling problems have been extensively studied by researchers in various disciplines [7, 10, 12, 8] and continue to be the focus of attention of the distributed computing community due to their theoretical significance and practical importance, especially as the grid computing technology prevails [5, 4, 6]. In [3], Benoit *et al.* discussed the mapping of computing pipelines onto different types of fully connected networks with identical processors and links (fully homogeneous platform), with identical links but different processors (communication homogeneous platform), or with different processors and links (fully heterogeneous platform). A grid scheduling algorithm, called *Streamline* [2], is developed for placing a coarse-grain dataflow graph on available grid resources. This scheduling heuristic is specifically designed to improve the performance of streaming applications with various demands in grid environments. Kwok *et al.* proposed Dynamic Critical-Path (DCP) scheduling algorithm [11] to map task graphs with arbitrary computation and communication costs to a distributed network environment consisting of fully-connected identical nodes. Chen *et al.* proposed and evaluated a runtime algorithm for supporting adaptive execution of distributed data mining on streaming data [9].

We consider the problems of minimizing the end-to-end delay and maximizing the frame rate of a linear computing pipeline for interactive applications and streaming applications, respectively, in an arbitrary computing network. Our design goal is to find an efficient mapping scheme that allocates the modules of a computing pipeline to network nodes in physical networks to achieve minimum end-to-end delay or maximum frame rate. Based on the analytical cost models for modules, nodes, and links, we formulate an optimization version of the mapping problem and propose a solution based on the Efficient Linear Pipeline Configuration (ELPC) algorithms. In particular, we develop an optimal polynomial-time algorithm based on dynamic programming to solve the mapping problem for minimum end-to-end delay. Furthermore, we prove that a restricted version of the mapping problem for maximum frame rate without node reuse is NP-complete and develop an approximate solution based on dynamic programming.

We implement the ELPC algorithms and conduct extensive mapping experiments in a large number of simulated application and network settings. In practical applications, the bandwidth of a network transport path can be measured using active traffic measurement technique based on a linear regression model described in [14], and the processing time of a computing or visualization module can be measured using similar techniques described in [13]. These methods achieve high accuracy in performance estimation as evidenced by extensive real experimental results. However, the details of these cost models and performance measuring techniques are out of the scope of this paper.

For comparison purposes, the *Streamline* algorithm adapted to linear pipelines and a *Greedy* algorithm are also implemented and tested with the same simulation datasets on the same computing platform. The performance measurements show that the ELPC algorithms yield superior mapping performance in terms of minimum end-to-end delay or maximum frame rate over the existing methods.

The rest of the paper is organized as follows. In Section 2, we construct the analytical cost models for pipeline and network components and formulate an optimization version of each pipeline mapping problem. In Section 3, we propose solutions based on the ELPC algorithms to the pipeline mapping problems and also describe the other two algorithms for comparison. The implementation details, simulation setup, and performance evaluations are presented in Section 4. We conclude our work and discuss some future work in Section 5.

2 Cost Models and Problem Formulation

2.1 A General Computing Pipeline

A number of large-scale computational applications in various scientific, engineering, medical, and business fields require efficient executions of computing tasks that consist of a sequence of linearly arranged modules, also referred to as subtasks or stages. These modules form a so-called computing pipeline between a data source and an end user.

For a small-scale standalone application where an end user accesses a local data source, the entire computing pipeline may be executed on a single computer. However, for large-scale distributed applications where data sources are not located at the same site as end users, we are faced with a challenge to support increasingly complex computing pipelines over wide-area networks that comprise heterogeneous computing nodes and communication links. The remote visualization in next-generation scientific applications such as Terascale Supernova Initiative (TSI) [1] is a typical example, where the simulation datasets generated on remote supercomputers must be retrieved, filtered, transferred, processed, visualized, and analyzed by a collaborative team of geographically distributed scientists. Note that a computing pipeline with only two end modules reduces to a traditional client/server based computing paradigm.

Due to the disparate nature of data sources and the intrinsic heterogeneity of network nodes, communication links, and application computing tasks, deploying component modules on different sets of computing nodes can result in substantial performance differences. The pipeline mapping problems in our work are to find an efficient mapping scheme that maps the computing modules onto a set of strategically selected nodes to (i) minimize end-to-end delay for interactive applications where a single dataset is processed sequentially along a computing pipeline, and (ii) maximize frame rate for streaming applications where multiple datasets are fed into a computing pipeline in a batch processing mode to sustain continuous data flow.

2.2 Cost Models of Pipeline and Network Components

We construct an analytical cost model for each pipeline and network component to facilitate the mathematical formulations of the aforementioned mapping problems. The computational complexity of a computing module M_i is denoted as c_i, which, together with the incoming data size m_{i-1}, determines the number of CPU cycles needed to complete the subtask defined in the module. The output data of size m_i is sent to its immediate successor node in the pipeline for further processing. Note that the actual runtime of a computing module also depends on the capacity of the system resources deployed on the selected network node as well as their availability during runtime.

The processing capability of a network node is a complex notion that combines a variety of host factors such as processor frequency, bus speed, memory size, storage performance, and presence of co-processors. For simplicity, we use a normalized quantity p_i to represent the overall computing power of a network node v_i without specifying its detailed system resources. The communication link between network nodes v_i and v_j is denoted as $L_{i,j}$, which is characterized by two attributes, namely bandwidth (BW) $b_{i,j}$ and minimum link delay (MLD) $d_{i,j}$. The transfer time of a large message is mainly constrained by bandwidth, while minimum link delay could be a significant overhead for the transfer of a message with size comparable to the Maximum Transmission Unit (MTU) of the underlying network. In practical applications, we may employ a linear regression-based method to estimate the bandwidth and minimum link delay of a communication link [14]. We estimate the computing time of module M_i running on network node v_j to be $T_{computing}(M_i, v_j) = \frac{m_{i-1}c_i}{p_j}$ and the transfer time of message size m over a communication link $L_{i,j}$ to be $T_{transport}(m, L_{i,j}) = \frac{m}{b_{i,j}} + d_{i,j}$.

2.3 Problem Formulation

We now present the mathematical formulations of the general computing pipeline mapping problems based on the cost models defined above. We consider an underlying transport network that consists of k geographically distributed computing nodes denoted by $v_1, v_2, \ldots, v_{k-1}, v_k$. Node $v_i, i = 1, 2, \ldots, k-1, k$ has a normalized computing power p_i and is connected to its neighbor node $v_j, j = 1, 2, \ldots, k-1, k, j \neq i$ with a network link $L_{i,j}$ of bandwidth $b_{i,j}$ and minimum link delay $d_{i,j}$. The transport network is represented by a graph $G = (V, E), |V| = k$, where V denotes the set of network nodes (vertices) and E denotes the set of communication links (edges). Note that the transport network may or may not be a complete graph, depending on whether the node deployment environment is the Internet or a dedicated network. The general computing pipeline consists of n sequential modules, $M_1, M_2, \ldots, M_{u-1}, M_u, \ldots, M_{v-1}, \ldots\ldots, M_w, \ldots, M_{x-1}, M_x, \ldots, M_n$, where M_1 is a data source and M_n is an end user. Module $M_j, j = 2, \ldots, n$ performs a computing module of complexity c_j on the data of size m_{j-1} received from its predecessor module M_{j-1} and generates and sends data of size m_j to its successor module M_{j+1}.

The objective of a general mapping scheme is to decompose the pipeline into q groups of modules denoted by $g_1, g_2, \ldots, g_{q-1}, g_q$, and map them onto a selected path P of q nodes from a source node v_s to a destination node v_d in the transport network, where $q \in [1, min(k,n)]$ and path P consists of a sequence of unnecessarily distinct nodes $v_{P[1]} = v_s, v_{P[2]}, \ldots, v_{P[q-1]}, v_{P[q]} = v_d$. For each mapping, we consider two cases: (i) with node reuse, two or more modules, either contiguous or non-contiguous (the selected path P contains a loop) in the pipeline, are allowed to run on the same node; (ii) without node reuse, a node on the selected path P executes exactly one module. Note that the path reduces to a single computer when $q = 1$.

- **Minimal total delay for interactive applications**

An important requirement in many collaborative applications is the interactivity of the system. We achieve the fastest system response by minimizing the total computing and transport delay of the pipeline from the source node to the destination node:

$$\begin{aligned}
&T_{total}(Path\ P\ of\ q\ nodes) = T_{computing} + T_{transport} \\
&= \sum_{i=1}^{q} T_{g_i} + \sum_{i=1}^{q-1} T_{L_{P[i],P[i+1]}} \\
&= \sum_{i=1}^{q} \left(\frac{1}{p_{P[i]}} \sum_{j \in g_i, j \geq 2} \left(c_j m_{j-1} \right) \right) + \sum_{i=1}^{q-1} \left(\frac{m(g_i)}{b_{P[i],P[i+1]}} \right),
\end{aligned} \quad (1)$$

where we assume that the inter-module transport time within one group on the same node is negligible.

- **Maximal frame rate for streaming applications**

We maximize the frame rate to produce the smoothest data flow for streaming applications where datasets are continuously generated and fed into the pipeline. This goal is achieved by identifying and minimizing the time incurred on a bottleneck link or node, which is defined as:

$$
\begin{aligned}
& T_{bottleneck}(Path\ P\ of\ q\ nodes) \\
&= \max_{\substack{Path\ P\ of\ q\ nodes \\ i=1,2,\ldots,q-1}} \left\{ \begin{array}{l} T_{computing}(g_i), \\ T_{transport}(L_{P[i],P[i+1]}), \\ T_{computing}(g_q) \end{array} \right\} \\
&= \max_{\substack{Path\ P\ of\ q\ nodes \\ i=1,2,\ldots,q-1}} \left\{ \begin{array}{l} \frac{1}{p_{P[i]}} \sum\limits_{j \in g_i\ and\ j \geq 2} (c_j m_{j-1}), \\ \frac{m(g_i)}{b_{P[i],P[i+1]}}, \\ \frac{1}{p_{P[q]}} \sum\limits_{j \in g_q\ and\ j \geq 2} (c_j m_{j-1}) \end{array} \right\}.
\end{aligned}
\qquad (2)
$$

In Eqs. 1 and 2, we assume that the first module M_1 only transfers data from the source node and the last module M_n only performs certain computation without data transfer.

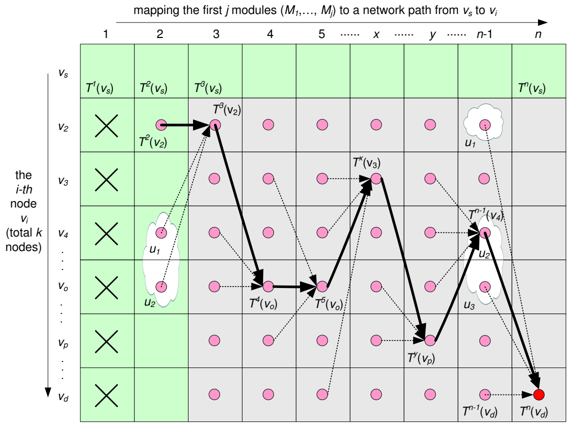


Figure 1. Construction of 2D matrix in ELPC for minimum end-to-end delay.

3 Algorithm Design

To optimize the network performance of computing pipelines in distributed environments, we propose a polynomial-time mapping scheme, Efficient Linear Pipeline Configuration (ELPC) to strategically map computing modules to network nodes for minimum end-to-end delay or maximum frame rate. We will also briefly present other two mapping algorithms we used for performance comparison.

3.1 ELPC Algorithms

3.1.1 Minimum End-to-end Delay with Node Reuse

For interactive applications, our goal is to minimize the end-to-end delay incurred on the nodes and links from the source to the destination to achieve the fastest response. Since a single dataset is processed and there is only one module being executed at any particular time, nodes can be reused but are not shared simultaneously among different modules.

Let $T^j(v_i)$ denote the minimal total delay with the first j modules mapped to a path from the source node v_s to node v_i under consideration in the network. Then, we have the following recursion leading to the final solution $T^n(v_d)$:

$$
\begin{aligned}
& \underset{j=2\ to\ n, v_i \in V}{T^j(v_i)} = \\
& \min \left\{ \begin{array}{l} T^{j-1}(v_i) + {c_j m_{j-1}}/{p_{v_i}}, \\ \min\limits_{u \in adj(v_i)} \left(T^{j-1}(u) + {c_j m_{j-1}}/{p_{v_i}} + {m_{j-1}}/{b_{u,v_i}} \right) \end{array} \right.
\end{aligned}
\qquad (3)
$$

with the base condition computed as:

$$
\underset{v_i \in V, and\ v_i \neq v_s}{T^2(v_i)} = \left\{ \begin{array}{ll} {c_2 m_1}/{p_{v_i}} + {m_1}/{b_{v_s,v_i}}, & \forall e_{v_s,v_i} \in E \\ \infty, & otherwise \end{array} \right. \qquad (4)
$$

on the second column. Here, we ignore the transport time between modules within one group on the same computing node. Every cell $T^j(v_i)$ in the table shown in Fig. 1 represents an optimal mapping solution that maps the first j modules in the pipeline to a path between the source node v_s and node v_i in the network and is calculated from the intermediate mapping results stored in the left column $T^{j-1}(\cdot)$.

We provide a correctness proof of this ELPC algorithm, where we consider two sub-cases at each recursive step, the minimum of which is chosen as the minimum total delay to fill in a new cell $T^j(v_i)$. In sub-case (i), we run the new module on the same node running the last module in the previous mapping subproblem $T^{j-1}(v_i)$. In other words, the last two or more modules are mapped to the same node v_i. Therefore, we only need to add the computing time of the last module on node v_i to the previous total delay, which is represented by a direct incident link from its left neighbor cell in the two-dimensional table. In sub-case (ii), the new module is mapped to node v_i and the last node u in a previous mapping subproblem $T^{j-1}(u)$ is one of the neighbor nodes of node v_i, which is represented by an incident link from a neighbor cell on the left column to node v_i. In Fig. 1, a set of neighbor nodes of node v_i are enclosed in a cloudy region in the previous column. Some cells may not be the neighbor nodes of node v_i since the network under consideration has an arbitrary topology. We calculate the end-to-end delay for each mapping of an incident link of node v_i and choose the minimal one, which is further compared with the one calculated in sub-case (i) (direct incident link from the left neighbor cell). The minimum of these two sub-cases is selected as the total minimum end-to-end delay for the partial computing pipeline mapping to a path between node v_s and node v_i. The fact that adding a new module and a new link at each step does not affect the optimality of the partial solutions previously calculated for the subproblems guarantees that the final solution is optimal for a given mapping problem. The complexity of this algorithm is $O(n \times |E|)$, where n denotes the number of modules in the linear computing pipeline and $|E|$ is the number of edges in the distributed network.

3.1.2 Maximum Frame Rate without Node Reuse

The maximum frame rate that a computing pipeline can achieve is limited by the bottleneck unit, i.e. the slowest transport link or computing node along the entire pipeline [15]. Node reuse in streaming applications causes resource sharing, and hence affects the optimality of the solutions to previous mapping subproblems. Here we consider a restricted version of the mapping problem for maximum frame rate by limiting the use of each node to a single module. In this restricted mapping problem, we attempt to find the widest network path with exact n nodes to map n modules in the pipeline on a one-to-one basis. By "widest", we mean that the network path has the minimum bottleneck among all feasible paths. Hence, the problem can be simplified to the exact n-hop widest path problem, whose complexity is equivalent to that of the exact n-hop shortest path problem (ENSP), which is shown to be NP-complete below.

Theorem: ENSP is NP-complete.

Proof: Obviously, ENSP is in NP class. We show its NP-hardness by reducing Hamiltonian Path (HP) problem to it.

Given an arbitrary instance I_{HP} in HP problem, we can transform it into an instance I_{ENSP} of ENSP problem, i.e. $I_{HP} \in HP \Rightarrow I_{ENSP} = f(I_{HP}) \in ENSP$, where $f(\cdot)$ is a polynomial-time transformation function. Consider instance I_{HP}: given an arbitrary graph $G = (V, E)$ with $n+1$ vertices, $v_0, v_1, v_2, \cdots, v_n$, does there exist a simple path from v_0 to v_n such that it contains each vertex exactly once? We build instance I_{ENSP} from instance I_{HP} as follows. First, we make a copy of the entire graph topology of G and denote it as graph $G' = (V', E')$ with $n+1$ vertices, where $V' = V$ and $E' = E$. Second, in graph G', we set the weight of all edges to be 1 and choose a bound $B = n$. Instance I_{ENSP} asks if there exists a simple path from v'_0 to v'_n with exact n hops such that the total path distance $D \leq B = n$.

Now we prove that there exists a simple path from v_0 to v_n in G that contains each vertex exactly once if and only if there exists a simple path with exact n hops from v'_0 to v'_n in G' whose distance $D \leq B = n$. Given a solution (path) P to I_{HP}, we can find a path P' in G' that only consists of edges corresponding to those of P in G. Obviously, path P' has exact n hops and satisfies the bound condition on its path distance, i.e. $D = n \leq B$. Therefore, path P' is a solution to I_{ENSP}. Similarly, given a solution (path) P' to I_{ENSP}, i.e. P' has exact n hops and its distance $D \leq B = n$, we can find a path P in G that consists of edges corresponding to those of P' in G'. Path P contains each vertex exactly once and therefore is a solution to I_{HP}. This concludes the proof for the NP-completeness of ENSP.

We develop an approximate solution to this problem by adapting the dynamic programming method for minimum end-to-end delay to this problem with some necessary modifications. Let $1/T^j(v_i)$ denote the maximal frame rate with the first j modules mapped to a path from source node v_s to node v_i in an arbitrary computer network. We have following recursion leading to the final solution $T^n(v_d)$:

$$\underset{j=2\ to\ n, v_i \in V}{T^j(v_i)} = \min_{u \in adj(v_i)} \left(\max \left(T^{j-1}(u), {}^{c_j m_{j-1}}/_{p_{v_i}}, {}^{m_j}/_{b_{u,v_i}} \right) \right) \tag{5}$$

with the base condition computed as:

$$\underset{v_i \in V, and\ v_i \neq v_s}{T^2(v_i)} = \begin{cases} \max \left({}^{c_2 m_1}/_{p_{v_i}}, {}^{m_1}/_{b_{v_s,v_i}} \right), & \forall e_{v_s,v_i} \in E \\ \infty & , \quad otherwise \end{cases} \tag{6}$$

on the second column in the table.

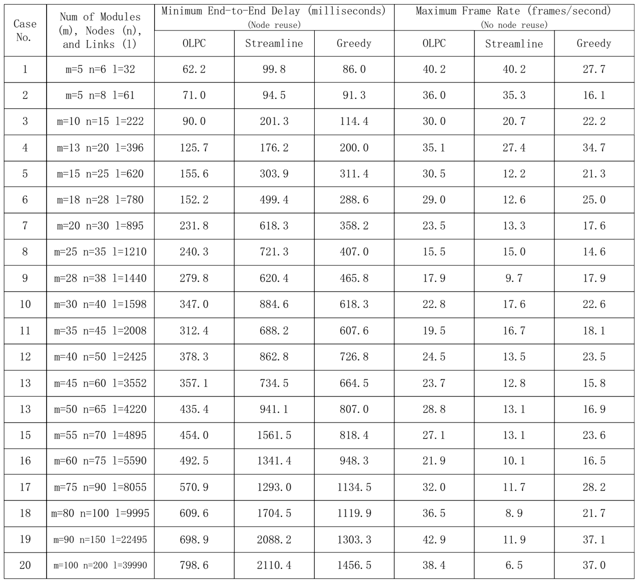
The steps for filling out the 2-D table for the maximum frame rate differ from those for the minimum total delay in the following aspects: at each step, we ensure that the current node has not been used previously in the path and calculate the bottleneck of the path instead of the total delay. This solution is heuristic because when a node has been selected by all its neighbor nodes at previous optimization steps, we may miss an optimal solution if this node is the only one leading to the destination node or obtain a suboptimal solution if there are multiple nodes leading to the destination node. We would also like to point out that this case is extremely rare as shown in our extensive experiments.

3.2 Streamline Algorithm

Agarwalla *et al.* proposed a grid scheduling algorithm, *Streamline*, for graph dataflow scheduling in a network with n resources and $n \times n$ communication links [2]. The Streamline algorithm considers application requirements in terms of per-stage computation and communication needs, application constraints on co-location of stages (node reuse), and availability of computation and communication resources. This scheduling heuristic works as a global greedy algorithm that expects to maximize the throughput of an application by assigning the best resources to the most needy stages in terms of computation and communication requirements at each step. The complexity of this algorithm is $O(m \times n^2)$, where m is the number of stages or modules in the dataflow graph and n is the number of resources or nodes in the network.

3.3 Greedy Algorithm

A greedy algorithm iteratively obtains the greatest immediate gain based on certain local optimality criteria at each step, which may or may not lead to the global optimum. We design a greedy mapping scheme that calculates the end-to-end delay or maximum frame rate for the mapping of a new



Case No.	Num of Modules (m), Nodes (n), and Links (l)	Minimum End-to-End Delay (milliseconds) (Node reuse)			Maximum Frame Rate (frames/second) (No node reuse)		
		OLPC	Streamline	Greedy	OLPC	Streamline	Greedy
1	m=5 n=6 l=32	62.2	99.8	86.0	40.2	40.2	27.7
2	m=5 n=8 l=61	71.0	94.5	91.3	36.0	35.3	16.1
3	m=10 n=15 l=222	90.0	201.3	114.4	30.0	20.7	22.2
4	m=13 n=20 l=396	125.7	176.2	200.0	35.1	27.4	34.7
5	m=15 n=25 l=620	155.6	303.9	311.4	30.5	12.2	21.3
6	m=18 n=28 l=780	152.2	499.4	288.6	29.0	12.6	25.0
7	m=20 n=30 l=895	231.8	618.3	358.2	23.5	13.3	17.6
8	m=25 n=35 l=1210	240.3	721.3	407.0	15.5	15.0	14.6
9	m=28 n=38 l=1440	279.8	620.4	465.8	17.9	9.7	17.9
10	m=30 n=40 l=1598	347.0	884.6	618.3	22.8	17.6	22.6
11	m=35 n=45 l=2008	312.4	688.2	607.6	19.5	16.7	18.1
12	m=40 n=50 l=2425	378.3	862.8	726.8	24.5	13.5	23.5
13	m=45 n=60 l=3552	357.1	734.5	664.5	23.7	12.8	15.8
13	m=50 n=65 l=4220	435.4	941.1	807.0	28.8	13.1	16.9
15	m=55 n=70 l=4895	454.0	1561.5	818.4	27.1	13.1	23.6
16	m=60 n=75 l=5590	492.5	1341.4	948.3	21.9	10.1	16.5
17	m=75 n=90 l=8055	570.9	1293.0	1134.5	32.0	11.7	28.2
18	m=80 n=100 l=9995	609.6	1704.5	1119.9	36.5	8.9	21.7
19	m=90 n=150 l=22495	698.9	2088.2	1303.3	42.9	11.9	37.1
20	m=100 n=200 l=39990	798.6	2110.4	1456.5	38.4	6.5	37.0

Figure 2. Mapping performance comparison of ELPC, Streamline, and Greedy.

module onto the current node when node reuse is allowed or one of its neighbor nodes and chooses the minimal one. This greedy algorithm makes a mapping decision at each step only based on current information without considering the effect of this local decision on the mapping performance in later steps. The complexity of this algorithm is $O(m \times n)$, where m denotes the number of modules in the linear pipeline and n is the number of nodes in the network.

4 Implementation and Experimental Results

4.1 Implementation

The proposed ELPC is implemented in C++ and runs on a Windows XP desktop equipped with a 3.0 GHz CPU and 2 Gbytes memory. For performance comparison purposes, we implement the other two algorithms, namely, Streamline and Greedy, in C++ on the same computing platform. We conduct an extensive set of mapping experiments for minimum end-to-end delay and maximum frame rate using a wide variety of simulated application pipelines and computing networks. We generate these simulation datasets by randomly varying the following pipeline and network attributes within a suitably selected range of values: (i) the number of modules, module complexities, input data sizes, and output data sizes in a pipeline; (ii) the number of nodes, node processing power, number of links, link bandwidth, and minimum link delay in a network.

We consider four parameters for each module in a pipeline: *ModuleID*, *ModuleComplexity*, *InputDataInBytes* and *OutputDataInBytes*. Note that the parameter *ModuleComplexity* is an abstract quantity that does not only depend on the computational complexity of the algorithm in the module but also the implementation details such as the specific data structures used in the program. The parameter *InputDataInBytes* denotes the size of the data received and processed by the current module, which together with the module complexity and the node computing power, determine the module execution time. The partial result produced by an intermediate module is denoted by the parameter *OutputDataInBytes* and serves as input data to its successor node along the pipeline.

We define three parameters for a computing node: *NodeID*, *NodeIP*, and *ProcessingPower*. Note that the parameter *ProcessingPower* is an abstract quantity that characterizes the general computing capability of a network node, which is primarily determined by the processor frequency, memory size, and bus speed. For each transport link, we define five parameters: *startNodeID*, *endNodeID*, *LinkID*, *LinkBWInMbps* and *LinkDelayInMilliseconds*. The computing networks considered in our experiments are not necessarily completely connected but essentially arbitrary in topology described in the form of an adjacency matrix.

For each mapping problem, we designate a source node and a destination node to run the first module and the last module of the pipeline. This is based on the consideration that the system knows where the raw data is stored and where an end user is located before optimizing the pipeline configuration over an existing network.

4.2 Illustration of ELPC Mapping Scheme

To better illustrate the pipeline mapping process, we plot the path selected by ELPC for minimum end-to-end delay in Fig. 3 and the one for maximum frame rate in Fig. 4 for the small-scale problem consisting of 5 modules, 6 nodes, and 32 links. In Fig. 3, the first two modules run on the source node with $NodeID = 0$, both module 2 and module 3 run on an intermediate node with $NodeID = 4$, and the last module runs on the destination node with $NodeID = 5$. In Fig. 4, a path consisting of nodes with $NodeID = 0, 3, 1, 4$ and 5 is selected for running five consecutive modules and the bottleneck is located on the last node.

4.3 Performance Comparison

With a large set of simulated application pipelines and computing networks described above, we performed extensive experiments on pipeline mapping using ELPC, Streamline, and Greedy, respectively. The measured execution time of these algorithms varies from milliseconds for small-

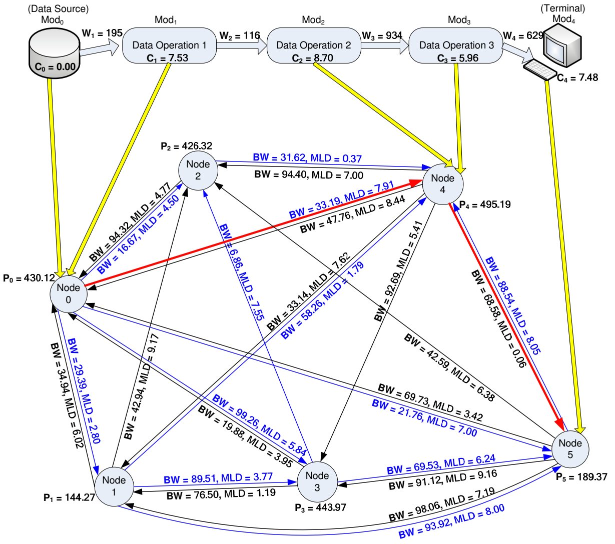


Figure 3. The optimal path with minimum end-to-end delay calculated by ELPC.

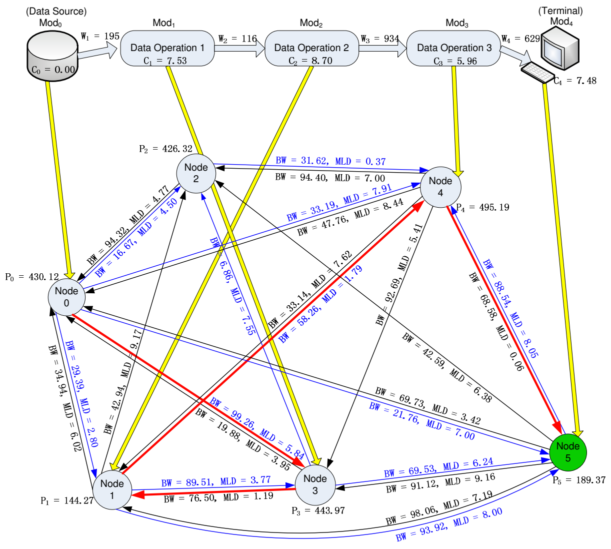


Figure 4. The optimal path with maximum frame rate calculated by ELPC.

scale problems to seconds for large-scale ones. A set of typical performance measurements in terms of minimum end-to-end delay and maximum frame rate collected in 20 different cases are tabulated in Fig. 2 for comparison. The relative performance differences of these three algorithms observed in other cases are qualitatively similar. In interactive applications that minimizes end-to-end delay for the fastest response, since only one single dataset sequentially flows through each module along the path, we allow network nodes to be reused but there is only one module executing on a selected node at any time. In streaming applications that identify and minimize the bottleneck node or link for the smoothest workflow, node reuse is disabled. We would like to point out that there may not exist any feasible mapping solution in some extreme test cases where the shortest end-to-end path is longer than the pipeline or the pipeline is longer than the longest end-to-end path but network nodes are not allowed for reuse. Here, the length of a path or pipeline refers to the number of nodes or modules.

For a visual performance comparison, we plot the performance measurements of minimum end-to-end delay and maximum frame rate produced by these three algorithms in Fig. 5 and Fig. 6, respectively. We observed that ELPC exhibits comparable or superior performances in minimizing end-to-end delay and maximizing frame rate over the other two algorithms in all the cases we studied. Since the end-to-end delay represents the total delay from a source node to a destination node, a larger problem size with more network nodes and computing modules generally (not absolutely, though) incurs a longer mapping path resulting in a longer end-to-end delay, which explains the increasing trend in Fig. 5. The maximum frame rate, the reciprocal of the bottleneck in a selected path, is not particularly related to the path length, and hence the performance curves in Fig. 6 lack an obvious increasing or decreasing trend in response to varying problem sizes.

5 Conclusions and Future Work

We designed an ELPC scheme based on dynamic programming that strategically maps modules of computing pipelines to shared or dedicated network environments to achieve the minimum end-to-end delay and maximum frame rate. We constructed cost models to quantitate the characteristics of modules of application pipelines and computing nodes and communication links in distributed networks. We implemented ELPC as well as the other two scheduling algorithms, Streamline and Greedy, and performed extensive pipeline mapping experiments using simulated application pipelines and computing networks. The experimental results show that the ELPC exhibits superior mapping performance over the other methods.

In the mathematical model of nodes, we used a normalized quantity to represent the processing power for simplicity. However, a single constant is not always sufficient to describe the node computing capability, which highly depends on the type and availability of system resources and could be time varying in a dynamic environment. The time-

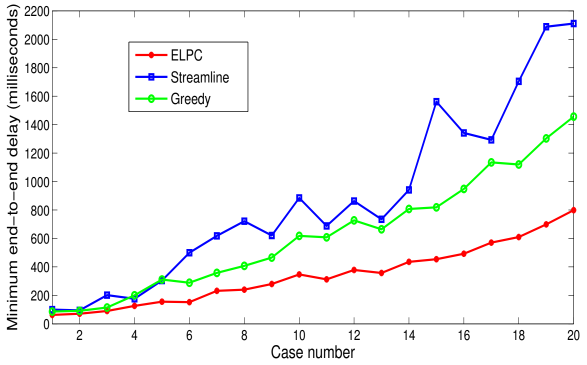


Figure 5. Performance comparison of minimum end-to-end delay for three algorithms.

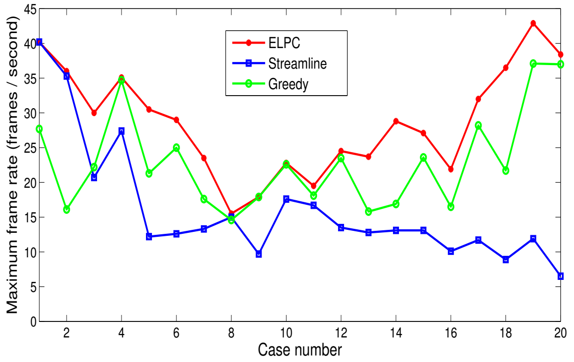


Figure 6. Performance comparison of maximum frame rate for three algorithms.

varying nature of system resources' availability makes it challenging to perform an accurate prediction or estimation of the execution time of a computing module in a real network environment. We will investigate sophisticated performance models to characterize real-time computing node behaviors and estimate more accurate module execution time.

In the perspective of algorithm design, it would be of our future interest to study the pipeline mapping problem for maximum frame rate in the case of node reuse. We will also extend linear pipelines to graph workflows and study the complexity of and develop efficient solutions to graph workflow mapping problems in distributed environments.

Acknowledgments

This research is sponsored by National Science Foundation under Grant No. CNS-0721980 and Oak Ridge National Laboratory, U.S. Department of Energy, under Contract No. PO 4000056349 with University of Memphis.

References

[1] Terascale Supernova Initiative (TSI). http://www.phy.ornl.gov/tsi.

[2] B. Agarwalla, N. Ahmed, D. Hilley, and U. Ramachandran. Streamline: a scheduling heuristic for streaming application on the grid. In *Thirteenth Multimedia Computing and Networking Conference*, San Jose, CA, 2006.

[3] A. Benoit and Y. Robert. Mapping pipeline skeletons onto heterogeneous platforms. In Y. Shi, D. van Albada, J. Dongarra, and P. Sloot, editors, *The International Conference on Computational Science, Lecture Notes in Computer Science*, volume 4487, pages 591–598. Springer, 2007.

[4] R. Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Computing*. PhD thesis, Monash University, Melbourne, Australia, April 2002.

[5] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. In *High Performance Computing*, ASIA, 2000.

[6] J. Cao, S. Jarvis, S. Saini, and G. Nudd. Gridflow: workflow management for grid computing. In *The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 198–205, May 2003.

[7] V. Chaudhary and J. Aggarwal. A generalized scheme for mapping parallel algorithms. *IEEE Transactions on Parallele and Distributed Systems*, 4(3):328–346, May 1993.

[8] L. Chen and G. Agrawal. Resource allocation in a middleware for streaming data. In *The 2nd Workshop on Middleware for Grid Computing*, Toronto, Canada, October 2004.

[9] L. Chen and G. Agrawal. Supporting self-adaptation in streaming data mining applications. In *IEEE International Parallel and Distributed Processing Symposium*, 2006.

[10] S. Kim and J. Browne. A general approach to mapping of parallel computation upon multiprocessors architectures. In *Proceedings of International Conference on Parallel Processing*, pages 1–8, 1988.

[11] Y.-K. Kwok and I. Ahmad. Dynaimic critical-path scheduling:an effective technique for allocating task graph to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5), May 1996.

[12] B. Shirazi, M. Wang, and G. Pathak. Analysis and evaluation of heuristic methods for static scheduling. *Journal of Parallel and Distributed Computing*, (10):222–232, 1990.

[13] Q. Wu, J. Gao, M. Zhu, N. Rao, J. Huang, and S. Iyengar. Self-adaptive configuration of visualization pipeline over wide-area networks. *IEEE Transactions on Computers*, 57(1):55–68, January 2008.

[14] Q. Wu and N. S. V. Rao. On transport daemons for small collaborative applications over wide-area networks. In *Proceedings of the 24th IEEE International Performance Computing and Communications Conference*, pages 159–166, Phoenix, AZ, April 7-9 2005.

[15] M. Zhu, Q. Wu, N. Rao, and S. Iyengar. Optimal pipeline decomposition and adaptive network mapping to support remote visualization. *Journal of Parallel and Distributed Computing*, 67(8):947–956, August 2007.