Streamline: A Scheduling Heuristic for Streaming Applications on the Grid

Bikash Agarwalla, Nova Ahmed, David Hilley, Umakishore Ramachandran
College of Computing, Georgia Institute of Technology
801 Atlantic Drive NW, Atlanta, GA 30332-0280, USA
Email: {bikash, nova, davidhi, rama}@cc.gatech.edu

Abstract

Streaming applications such as video-based surveillance, habitat monitoring, and emergency response are good candidates for executing on high-performance computing (HPC) resources, due to their high computation and communication needs. Such an application can be represented as a coarse-grain dataflow graph, each node corresponding to a stage of the pipeline of transformations that may be applied to the data as it continuously streams through. Mapping such applications to HPC resources has to be sensitive to the computation and communication needs of each stage of the pipeline to ensure QoS criteria such as latency and throughput. Due to the dynamic nature of such applications, they are ideal candidates for using ambient HPC resources made available via the grid. Since grid has evolved out of traditional high-performance computing, the tools available, especially for scheduling, tend to be more appropriate for batch-oriented applications. We have developed a scheduler, called Streamline*, that takes into account dynamic nature of the grid and runs periodically to adapt scheduling decisions using application requirements (per-stage computation and communication needs), application constraints (such as co-location of stages on the same node), and current resource availability. The scheduler is designed to be integrated with the existing grid framework using Globus Toolkit. The performance of Streamline is compared with an Optimal placement for small number of resources and approximation algorithms using Simulated Annealing for large resources and dataflow graphs. We have also compared Streamline with a baseline grid scheduler, E-Condor, built on top of Condor for streaming applications. For kernels of such streaming applications, we show that our heuristic performs close to Optimal, and can be nearly an order of magnitude better than the E-Condor under non-uniform load conditions. We considered two Simulated Annealing algorithms with different execution times, and show that neighbor-selection and annealing schedule have a relatively larger impact on the performance of Simulated Annealing for communication-intensive kernels than for computation intensive kernels. We have also conducted scalability studies and show that our scheduler is more effective than E-Condor, and performs close to Simulated Annealing algorithms, with smaller scheduling time, in allocating resources for a large streaming application.*

1 Introduction

Advances in sensing capabilities, and computing and communication infrastructures are paving the way for new and demanding applications. Video-based surveillance, emergency response, disaster recovery, habitat monitoring, and telepresence are all examples of such applications. These applications in their full form are capable of stressing the available computing and communication infrastructures to their limits. *Streaming applications*, as we refer to such applications in this paper, have the following characteristics: (1) they are continuous in nature, (2) they require efficient transport of data from/to distributed sources/sinks, and (3) they require the efficient use of high-performance computing resources to carry out compute-intensive tasks in a timely manner.

The focus of this work is in addressing the third component of the above characteristics, namely, the use of high-performance computing (HPC) resources to carry out compute-intensive tasks. Consider for example, a video-based surveillance application. The compute intensive part of such an application may consist of analyzing multiple camera feeds from a region to extract higher level information such as "motion", "presence or absence of a human face", or "presence or absence of any kind of suspicious activity". Such an application can be represented as a coarse-grain dataflow graph, wherein the nodes represent increasing sophistication of computations that may need to be performed on the data stream to facilitate the extraction of high-level information.

At some level such coarse-grain dataflow graphs resemble task-graphs that have been the focus of multiprocessor scheduling work from the 70's [2, 11, 23, 17, 24, 16, 20, 26].

However, there are several crucial differences. In multiprocessor scheduling, a task-graph (a directed acyclic graph) is used as an ordering mechanism to show the dependencies among the individual tasks of a parallel computation. These dependencies are respected and exploited in arriving at a mapping heuristic (since the scheduling problem is known to be NP-Complete) that maximizes the utilization of the computational resources and reduces the completion time of the application. The coarse-grain dataflow graph of a streaming application, on the other hand, is a representation of the processing that is carried out on the data during its passage through the pipeline of stages. In the steady state, all the stages of the pipeline are working on different snapshots of the stream data. For example in a video-based surveillance application, when the n-th stage is working on the (hitherto transformed results of the) i^{th} frame of video, the first stage of the pipeline is working on the $(n+i)^{th}$ frame. So the scheduling of such streaming applications is not an ordering issue; rather, it is a matter of mapping the different stages of the pipeline to the available resources respecting the computation and communication requirements of each stage with a view to optimizing the latency and throughput metrics of the entire pipeline.

An interesting aspect of this emerging class of streaming applications is that they are *ubiquitous* and *dynamic*. Thus, HPC resources are needed in a distributed manner to address the dynamic needs of an application. For example, in a video-based surveillance application the video feeds from the northwest corner of a campus may need to be analyzed due to some suspicious activity detected there. Grid computing offers the ability to harness the ambient HPC resources for a compute intensive problem. Grid computing has its roots in traditional high-performance computing, with initial efforts focused on scientific and engineering applications. While there have been efforts to expand the reach of the grid to support interactive [25] and streaming applications [9], the scheduling infrastructure available in the grid is largely geared to support batch-oriented applications.

In this paper, we study the problem of scheduling streaming applications on the grid. The scheduling heuristic, called *Streamline*, is designed specifically to adapt to the dynamic nature of grid environment and varying demands of a streaming application. It runs periodically and takes into account (a) computation and communication requirements of the various stages of the dataflow graph, (b) any application-specified constraints, and (c) the current resource (processing and bandwidth) availability. The output of the scheduling heuristic is a placement of the stages of the pipeline on the available HPC resources such that the latency and throughput of the application are optimized. Streamline is evaluated as a placement algorithm to measure the quality of the generated solutions so that proper dynamic task migration decisions can be taken in a grid deployment without degrading the overall performance of a streaming application.

We have designed our scheduling heuristic over the existing grid framework, using Globus Toolkit [14]. In our experimental study, we compare Streamline with the Optimal placement for small dataflow graphs and with Simulated Annealing algorithms otherwise. We also analyze how existing batch schedulers in grid can be enhanced to support streaming applications using Condor[4] as an example. Condor, a well studied resource allocator for grid, uses DAGMan[15] for task graph based applications. DAGMan is designed for batch jobs with control-flow dependencies and ensures that jobs are submitted in proper order, whereas different stages of a streaming application work concurrently on a snapshot of data. Thus, we have extended Condor to meet the particular streaming requirements, resulting in a baseline scheduler called *E-Condor*. We compare the performance of Streamline with Optimal, Simulated Annealing, and E-Condor for "kernels" of streaming applications. The results show that our heuristic performs close to Optiman and Simulated Annealing, and is better than E-Condor by nearly an order of magnitude when there is non-uniform CPU resource availability, and by a factor of four when there is non-uniform communication resource availability. We have considered two variants of Simulated Annealing algorithm with different execution times and observe that neighbor-selection and annealing schedule in an Simulated Annealing algorithm have a relatively larger impact on the performance of generated schedule for communication-intensive kernels than for computation intensive kernels. We have also conducted scalability studies and demonstate the scalability of our heuristic for handling large-scale streaming applications. The results show that our scheduler is more effective than E-Condor in handling large dataflow graphs, and performs close to Simulated Annealing algorithms, with smaller scheduling time.

The rest of the paper is organized as follows. In Section 2, we define the scheduling problem. Section 3 describes our Streamline scheduling heuristic. We present the overall system architecture that integrates Streamline into the grid computing framework in Section 4. The experimental setup, performance evaluation and results are presented in Section 5. We put our work in the context of other related works in Section 6 and present our conclusions in Section 7.

2 Problem Definition

A scheduling system model in the grid environment consists of an application, available resources, application specific constraints, resource specific constraints, and a performance criteria for scheduling. The streaming application is represented by a coarse-grain directed acyclic dataflow

graph, *G = (V, E)*, where *V* is the set of *v* stages and *E* is the set of *e* edges. Each node s_i of the dataflow graph represents a continuously running application stage with the direction of dataflow denoted by the edges. In our application model, each node of the dataflow graph continuously receives data items from the preceding stage, performs computation, and sends data item to the subsequent stages. Each edge *(i,j)* $\in E$ represents the direction of dataflow such that stage s_j waits for data to arrive from stage s_i before execution. *Ecycle* is a $v \times 1$ matrix of computation data, where $ecycle_i$ is an estimate of the average amount of CPU cycles required by stage s_i for each streaming data item produced. *Ecomm* is a $v \times v$ matrix of communication, where $ecomm_{i,j}$ is the amount of data required to be transmitted from stage s_i to stage s_j. These processing and communication estimates can be provided by the application for each stage of the dataflow graph. Alternatively, these estimates can be derived by application profiling as we have done in [28].

Static information (such as machine architecture, CPU speed, amount of memory, and hardware configuration) about the available resources is obtained by querying the information service [12]. Dynamic information (such as estimate of available processing cycles and end to end network bandwidth) are obtained from Network Weather Service (NWS) [29]. Our target computing environment consists of a set *Q* of *q* resources. *B* is a $q \times q$ communication matrix in which $b_{i,j}$ gives an estimate of available network bandwidth between node n_i and node n_j. Similarly, *Proc* is a $q \times 1$ computation matrix in which $proc_i$ gives an estimate of available CPU cycles on node n_i.

Before scheduling, each stage in the dataflow graph is labeled with an average execution cost ($\overline{w_i}$), and each edge is labeled with an average data transmission cost ($\overline{c_{i,j}}$). The average execution cost ($\overline{w_i}$) of stage s_i is measured as a ratio of the required average CPU cycles $ecycle_i$ and average available CPU cycles across the available resources. The average data transmission cost for edge *(i,j)*, ($\overline{c_{i,j}}$), is defined as the ratio of the estimated data transmission required, $ecomm_{i,j}$, and the average available data transfer across all resource pairs. The scheduler also takes as input a set of application constraints and a set of constraints for available resources. The resource specific constraints are gathered by querying the information service [12]. Each of the constraints specify various site specific policies that may affect the resources allocated to a streaming application.

In a dataflow graph, a stage without any parent is called an *input stage* and a stage without any child is called an *output stage*. After all stages in a dataflow graph are scheduled, the throughput of the application is the actual rate at which data items are produced by the output stage s_{out}. The *objective function* of the scheduling problem is to determine the assignments of a streaming application dataflow graph to available resources such that the resource and application specific constraints are satisfied and throughput is maximized.

3 The Streamline Scheduler

We have developed a grid scheduling algorithm, called *Streamline*, for placement of coarse-grain dataflow graph of a streaming application using available grid resources. Streamline makes the scheduling decision taking into consideration static information of available resources, dynamic information of available processing and communication capabilities in the target environment and different application and resource specific policies. The scheduling heuristic expects to maximize throughput of the application by assigning the best resources to the most needy stage in terms of computation and communication requirements. Streamline works in three phases: *stage prioritization phase* where the stages of the dataflow graphs are prioritized depending on their computation and communication criteria, *resource filtering phase* for filtering available resources based on application and resource specific policies, *resource selection phase* for selecting the "best" resource that maximizes the throughput of the entire graph. Streamline belongs to the general class of *list scheduling* algorithms [2, 11, 17, 23, 18]. We differ from traditional algorithms in (*i*) stage selection where we take computation and communication into account, (*ii*) estimating the required computation and communication cost for a stage in the dataflow graph, (*iii*) taking into account the resource and application specific policies, and (*iv*) taking into consideration the dynamic information about available processing and communication capabilities in the target environment.

3.1 Stage Prioritization Phase

This phase considers the computation and communication cost of the dataflow graph in assigning priorities to different stages. The computation and communication intensive tasks get higher priorities over the other tasks. Also the remaining execution time to process a particular data item by all subsequent stages is taken into account where a stage with the highest cost path to the output stage gets priority.

Two terms *rank* and *blevel* are introduced here. *Rank* calculates the average computation and communication cost of a stage and *blevel* estimates the overall remaining execution time of a data item after being processed by a stage. The *blevel* of a stage s_i is the cost of the longest path from from s_i to an exit node and is recursively defined by

$$blevel(s_i) = \overline{w_i} + \max_{s_j \in succ(s_i)} (\overline{c_{i,j}} + blevel(s_j)) \quad (1)$$

where *succ*(s_i) is the set of immediate successors of stage

s_i, $\overline{w_i}$ is the average computation cost of stage s_i and $\overline{c_{i,j}}$ is the average communication cost of edge (*i,j*).

The *rank* of a stage is the sum of communication and computation cost for a particular stage, and gives a rough estimate of total computation and communication time required by the stage to fetch all input data items, process them, and send the result to successive stages. The *rank* is used in relative ordering of the stages based on their requirements *before* an actual assignment is made. Therefore, this simple model suffices in giving higher *rank* to a more needy stage. We assign a *rank* to each stage s_i, taking into account the average computation and communication cost as

$$rank(s_i) = \overline{w_i} + \sum_{s_j \in pred(s_i)} \overline{c_{j,i}} + \sum_{s_k \in succ(s_i)} \overline{c_{i,k}} \quad (2)$$

where *pred(s_i)* is the set of immediate predecessors of stage s_i.

We assign higher priority to a stage with higher *rank* value and consider *blevel* to break ties. The stages are considered in the order of their priority and are allocated the "best" available resources.

3.2 Resource Filtering Phase

In this phase of the scheduling algorithm, we filter out available resources that may not be permissible by the application or resource policies and obtain a set *R* of *r* candidate resources. We take into account the application specific static resource requirements as well as resource specific policies. Even though there may be a large number of available resources, the candidate resource set may be small after this step, depending on how restrictive the application and resource policies are. Some examples of application specific constraints are (*i*) collocating stages of a streaming application on the same resource to reduce communication latency, (*ii*) any special requirements of a stage such as a graphics co-processor, (*iii*) QoS requirements specifying desired throughput and latency, and (*iv*) application defined priorities among different choices such as image and audio quality degradation for application adaptation.

3.3 Resource Selection Phase

In this phase the appropriate resources are picked from the set of candidate resources obtained in the resources filtering phase. Unlike most of the task graph schedulers that take only the computation capability to select resources, Streamline considers available CPU as well as end-to-end bandwidth. Streamline evaluates a particular resource using a cost function that computes the cost of assigning a stage to a resource node. The cost function estimates the computation and communication time for a particular assignment of a stage and picks a resource which gives the least cummulative time. Since the stages are considered in the order of their resource requirements, represented by *rank*, which may not be consistent with the dataflow order, Streamline uses estimates of average input and output bandwidth availability for each resource in calculating the cost of an assignment. By using information gathered from NWS ([29]), Streamline algorithm estimates the average incoming ($\overline{b_{in}(i)}$) and average outgoing bandwidth ($\overline{b_{out}(i)}$) for each resource node n_i as

$$\overline{b_{in}(i)} = \sum_{n_j \in R} b_{j,i}/r \quad (3)$$

$$\overline{b_{out}(i)} = \sum_{n_k \in R} b_{i,k}/r \quad (4)$$

where *R* is the candidate resource set of *r* resources and $b_{i,j}$ is an estimate of available end to end network bandwidth between node n_i and node n_j. The cost (*A* (n_j, s_i)) of allocating resource node $n_j \in R$ to stage s_i is computed by summing up the estimated computation and communication cost for this particular assignment and is represented as

$$\begin{aligned} A(n_j, s_i) &= ecycles_i/proc_j \\ &+ \sum_{s_k \in pred(s_i)} ecomm_{k,i}/\overline{b_{in}(j)} \\ &+ \sum_{s_k \in succ(s_i)} ecomm_{i,k}/\overline{b_{out}(j)} \quad (5) \end{aligned}$$

where $ecycle_i$ is an estimate of the average amount of CPU cycles required by stage s_i, $proc_j$ is an estimate of available CPU cycles on node n_j and $ecomm_{i,k}$ is the average amount of data transfered from stage s_i to stage s_k. To each stage s_i of the dataflow graph, we assign a resource n_j that has the minimum cost (*A* (n_j, s_i)) as calculated by Equation 6. Since we consider end-to-end network bandwidth available between each pair of resources in the candidate set, the Streamline algorithm has $O(v \times r^2)$ time complexity where *v* is the number of stages in the dataflow graph and *r* is the number of resources in the candidate set *R*. Even in the presence of large number of resources, we expect that the candidate set for each stage will be small and the time complexity is admissible.

To eliminate any ambiguity, the algorithm also takes into account the following points: (*i*) When multiple stages get assigned to the same resource, distribute the available bandwidth and CPU resources equally between all the stages in

// Input: dataflow graph G(S,E), resource set R
// Output: assign stages $s_i \in$ S to resources $n_j \in$ R
/ Initialize the dataflow graph */*
for (stage $s_i \in S$, edge $e_{i,j} \in E$ in dataflow graph G(S,E))
    $s_i = w_i$ //w_i: avg execution cost
    $e_{i,j} = c_{i,j}$ //$c_{i,j}$: avg transmission cost
/ Set priority to stages */*
for (every stage s_i in the dataflow graph G(S,E))
    blevel(s_i) = $\overline{w_i}$ + $\max_{s_j \in succ(s_i)}(\overline{c_{i,j}}$ + blevel(s_j))
    rank (s_i) = $\overline{w_i} + \sum_{s_j \in pred(s_i)} \overline{c_{j,i}} + \sum_{s_k \in succ(s_i)} \overline{c_{i,k}}$
/ Select the neediest stage */*
Sort stages list S in list SLIST in decreasing order of *rank*, use *blevel* to break ties.
/ Choose the best resource */*
while (there are unscheduled stages in SLIST)
    Select the first stage s_i from SLIST
    Construct candidate set *R* by filtering out permissible available resources by application or resource policies.
    for (each resource $n_k \in R$) // A: Assignment cost
        (a)A(n_k, s_i)=$ecycles_i$ / $proc_j$+
        $\sum_{s_k \in pred(s_i)} ecomm_{k,i} / \overline{b_{in}(j)}$ +
        $\sum_{s_k \in succ(s_i)} ecomm_{i,k} / \overline{b_{out}(j)}$
        (b) $s_i = n_k$ with min($A(n_k, s_i)$)

Table 1. Streamline Scheduling Algorithm

calculating the cost of the assignment[1]. (*ii*) In case there are multiple remaining candidate nodes, pick a node among the remaining candidate nodes at random. By randomly picking a node, the scheduler expects to distribute load among the equally desirable resources for a particular stage.

Since our scheduling heuristic works by picking best resource for each stage of the dataflow graph, additional policies concerning resources, applications, and local schedulers can be easily incorporated in calculating the cost of a particular assignment. In the next section, we present our system architecture that integrates the Streamline scheduling heuristic into grid computing framework.

4 System Architecture

We have designed a system that enables resource allocation for streaming application using grid. Our system architecture is guided by the following design goals: (*i*) For quicker deployment, the system should make use of existing grid functionalities as much as possible so long as doing so does not conflict with application performance requirements. (*ii*) The resource allocation system should function in a dynamic environment where resource availability and node connectivity change frequently. (*iii*) The scheduling algorithm should take into account non-uniform resource characteristics. Figure 1 shows the system which uses the exisiting grid functionalities of Globus Toolkit[14], the Network Weather Service[29] for current information and future prediction of the resources, and some additional services introduced to make the streaming scheduler function properly. The services that are part of the present Globus

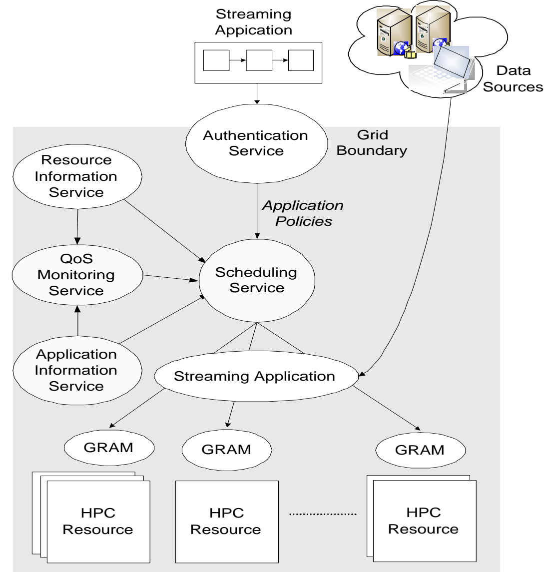

Figure 1. Resource Allocation System Architecture

Toolkit infrastructur are *Authentication Service*, which uses Grid Security Infrastructure (GSI) for authentication to the grid resources, the *Resource Information Service*[12], which provides information about the grid resources and the *Grid Resource Allocation and Management(GRAM)*[13] service, which provides access to individual grid resources. Our system also uses *Network Weather Service* for accessing estimates of dynamic information, such as CPU usage and end-to-end network bandwidth, about available resources. Among the services we have integrated to the grid are (*i*) *Application Information Service* that keeps track of the streaming appication status and (*ii*) the *QoS Monitoring service* that checks if the desired QoS is met. The *Scheduling Service* makes all scheduling decisions by contacting the QoS monitoring service, Network Weather Service and Resource Information Service. It runs the scheduling algorithm periodically to make sure the proper assignment of resources is in place and if the QoS goes below a threshold, a reallocation process takes place. The forecasts from the Network Weather Service are used in the task migration. The details of task migration process is outside the scope of

[1] Assuming that the local scheduler equally allocates the available CPU and network bandwidth, and that all the stages contend for CPU and network usage simultaneously. Given any additional information, the cost calculation can be accordingly adjusted.

this paper since our focus is on the placement heuristic.

The whole system works in the following manner - the user first authenticates to the grid using Grid Security Infrastructure(GSI). Once authenticated, the user submits the streaming application coarse-grain dataflow graph to the scheduling service. The scheduling service takes into account the current resource availability (through resource information service [12, 29]) and existing applications running in the grid (through application information service) in deciding whether to launch a new instance of the application. Once resources have been allocated, the application dataflow graph is instantiated on individual resources using Grid Resource Allocation and Management(GRAM) [13] system. After being instantiated, the input stages of the application access data directly from the sources. The QoS monitoring service is responsible for monitoring the quality of service requirements of the application and dynamically adapting resource assignment by contacting the scheduler. The QoS monitoring service infers the computational requirement of the running application by periodically contacting the application information service and the resource information service and provides this information to the scheduler.

5 Performance Evaluation

In this section, we study the performance of the Streamline scheduler for supporting streaming applications. Streamline provides a rich set of facilities that include: (*i*) APIs for the user to specify the resource requirements of each stage and the dependencies among the stages of the streaming application; (*ii*) APIs for job submission that allow multiple applications to be submitted to the scheduler at the same time; each application receives a distinguished name; and (*iii*) APIs for querying job status using the distinguished name.

In our experimental study we focus on evaluating the placement heuristic of Streamline. We compare the performance of Streamline with an Optimal placement algorithm. For large dataflow graphs and large number of resources, the execution time of the Optimal placement algorithm becomes computationally prohibitive. Therefore, we have designed an approximation algorithm using Simulated Annealing and present performance comparisons of Streamline with the Simulated Annealing approach for both small and large dataflow graphs. Since our scheduling algorithm is run periodically, the run-time of the scheduling algorithm is also critical. Therefore, we have considered two variants of Simulated Annealing algorithms, called *SA1* and *SA2*, with different execution times.

While it is possible to develop specialized application specific schedulers, using an existing grid scheduler will reduce the development time of streaming applications significantly in actual deployment. Therefore, we also analyze how existing batch schedulers in grid can he enhanced to support streaming applications. This has resulted in a baseline stream scheduler called *E-Condor*.

Thus, our experimental study has four parts: (*i*) We present an Optimal placement and approximation algorithms using Simulated Annealing for comparison with Streamline; (*ii*) We also investigate how existing grid schedulers can be enhanced to deal with streaming applications; this has resulted in a baseline stream scheduler called *E-Condor*; (*iii*) We compare the performance of Streamline with Optimal placement, Simulated Annealing, and E-Condor for executing "kernels" of streaming applications; (*iv*) We evaluate the scalability of Streamline with respect to Optimal, Simulated Annealing, and E-Condor algorithms

5.1 Optimal Placement Algorithm

The Optimal placement algorithm explores all possible assignments of resources to the individual stages of a dataflow graph and selects an assignment with the minimum cost. The cost of an assignment represents an estimate of the time it takes to produce a single output item as data is processed by the various stages of a dataflow graph. Since we are interested in observing the relative costs of different assignments, we use a simple model where this cost (*F*) is estimated by summing up the estimated computation and communication time of each stage of the dataflow graph for a particular assignment as follows:

$$\begin{aligned} F([n_0...n_v],[s_0...s_v]) &= \sum_{i=0}^{i=v}(ecycles_i/proc_i) \\ &+ \sum_{(i,j)\in E}(ecomm_{i,j}/b_{i,j}) \quad (6) \end{aligned}$$

where resource node n_i is allocated to stage s_i in the assignment under consideration, and $(i,j) \in E$ represents an edge between stages s_i and s_j in the dataflow graph.

The above cost model makes a conservative assessment by summing up the estimate of individual computation and communication times, ignoring any parallelism. The cost model also ignores networking and buffer management overheads in transmission. However, since we are interested only in the relative ordering of different assignments, use of such a simple cost model is justified.

For *v* stages and *r* candidate resources, the computational complexity of the Optimal algorithm is approximately equal to the number of permutations of *v* resources out of *r* ($^{r}P_v$). Thus, Optimal algorithm is computationally infeasible for large dataflow graphs. We have designed Simulated Annealing algorithms as comparison platform for large dataflow graphs and large number of resources.

5.2 Simulated Annealing Algorithms (SA1, SA2)

Simulated Annealing [21, 19] is a generalization of a Monte Carlo method for statistically finding a global optimum for multivariate function. The concept originated from the way in which crystalline structures are brought to more ordered state by an *annealing process* of repeated heating and slowly cooling the structures. Analogy to Simulated Annealing has been used in Operation Research to successfully solve a variety of optimization problems [19].

In simulated annealing, a system is initialized at *temperature T* with some configuration whose cost (analogous to energy in the original process) is evaluated to be F_0. A new configuration is constructed by applying a random perturbation, and change in cost *dF* is computed. If the new configuration lowers the cost of the system, it is unconditionally accepted. If the cost of the system is increased by the change, the new configuration is accepted with a probability given by the Boltzmann factor $\exp(-dF/T)$ [19]. This processes is repeated sufficient times at the current temperature to sample the search space by visiting *neighbors* of the current configuration. Then, the temperature is decreased as specified by a chosen *annealing schedule* and the entire process is repeated at successive lower temperature until a terminating condition (frozen state) is reached. This procedure allows the system to move to a lower cost state, while still getting out of local minima (especially at higher temperatures) due to probabilistic acceptance of some upward moves.

We designed Simulated Annealing algorithms for our scheduling problem. The state space of our simulated annealing algorithm is all possible assignments of candidate resources to the stages of a dataflow graph. We used the same cost function as in the Optimal algorithm (Equation 6) and $\exp(-dF/T)$ as transition probability for our problem. Since running time of the scheduling algorithm is also critical in our problem, We have selected two different strategies for neighbor selection and annealing schedule, leading to two different simulated annealing schedulers (*SA1* and *SA2*). SA1 has run-time complexity comparable to Streamline whereas SA2 requires longer running time, thereby expecting to produce better schedules. At a fixed temperature, SA1 considers *v* randomly selected neighbors whereas SA2 considers v^2 neighbors for a *v* stage dataflow graph. SA1 and SA2 also differ in the length of the annealing schedule whereby SA1 considers at most r^2 temperature reductions. We have also employed simple optimization heuristic that avoid repetitive transitions between the same two states at a fixed temperature by considering the neighbors in a fixed order. The details of the algorithms are presented in Table 2.

// Input: dataflow graph G(S,E), resource set R
// Output: assign stages $s_i \in$ S *to resources* $n_j \in$ R
Select *v* random resources from *r* eligible machines
Compute initial cost of assignment (F_0):
    $F_0([n_0...n_v], [s_0...s_v])$ = $\sum_{i=0}^{i=v}(ecycles_i/proc_i)$
      + $\sum_{(i,j)\in E}(ecomm_{i,j}/b_{i,j})$
Select initial temperature (T_0):
    Compute average increase in cost across neighbors ($\overline{dF_0}$)
    X_0 = 0.8 // average increase acceptance probability
    T_0 = $-\overline{dF_0}/\ln(X_0)$
repeat {at each temperature}
    for *v* steps **do — SA1**
    for v^2 steps **do — SA2**
        swap assignments for two allocated machines
        or remove a machine, add another machine
        compute change in cost (*dF*)
        if *dF* is negative
            accept new schedule unconditionally
        else
            accept if a random number $< \exp(-dF/T)$
        decrease temperature by a factor of α (0.95)
until r^2 steps or temperature is above threshold (0.001)
            or cost does not change — **SA1**
        temperature is above threshold (0.001) or
            cost does not change — **SA2**

Table 2. Simulated Annealing Algorithms (SA1 and SA2)

5.3 E-Condor Architecture

Most of the existing grid schedulers such as Condor [4], Legion [7], and Nimrod-G [5] focus on allocating resources for batch-oriented applications. We have selected Condor [4], due to its maturity and flexibility, as a vehicle for comparison of an existing grid scheduler against Streamline.

Condor uses DAGMan[15] to launch applications that are specified by a task-graph. DAGMan is designed for task-graph based batch jobs with control-flow dependencies and hence launches a stage s_i of a task-graph only after all stages $s_j \in pred(s_i)$ have *finished* execution. However, as we have observed before, in a streaming application, each stage of the dataflow graph is concurrently working on a snapshot of the continuous stream data. Therefore, we have developed a simple stream scheduler on top of Condor called *E-Condor*. E-Condor uses Condor to obtain the resources necessary to launch the individual stages of a streaming application. But prior to launching, E-Condor takes care of setting up all the necessary coupling between the stages commensurate with the dataflow graph of the application.

E-Condor architecture, shown in Figure 2, has four components: (*i*) A *parser* that automatically generates the entire dataflow graph and the per-stage configuration files given a high-level description of the streaming application; (*ii*) A *launcher* that uses Condor to map the stages of the dataflow graph to different computational nodes provided by Condor; (*iii*) A *registration and discovery* service for establishing the predecessor/successor relationships among the stages of the dataflow graph after they are launched. The architecture automatically generates wrapper code for each stage to register itself with this service, determine its predecessors and successors using the per-stage configuration file, and establish the necessary connections to them; and (*iv*) A *synchronization protocol* that ensures that all the stages have established the necessary connections to one another before actually starting the application-supplied code for that stage.

E-Condor serves as a baseline scheduler for streaming applications on the grid that uses an existing grid scheduler.

5.4 Distributed Surveillance Application

We use a mock-up of a distributed video-based surveillance application as an example streaming application for the performance study. While the application itself may be distributed, we focus on the compute-intensive part of the application that performs hierarchical processing on video streams produced by cameras. The scheduling of this compute-intensive part is the focus of this experimental study. To arrive at a realistic model of the pipeline that represents this application, we use the following representative image manipulation functions that may form part of this hierarchical processing pipeline: (*i*) **Collage**: A simple concatenation of two images to produce a composite output; (*ii*) **EdgeDetect**: An algorithm to determine the boundaries of objects in an image; (*iii*) **MotionDetect**: An algorithm that computes the magnitude and centroid o f inter-frame differences in images to derive inferences on motion; and (*iv*) **FD/FR**: A compute-intensive algorithm to detect and recognize faces in an image that is based on skin tone analysis.

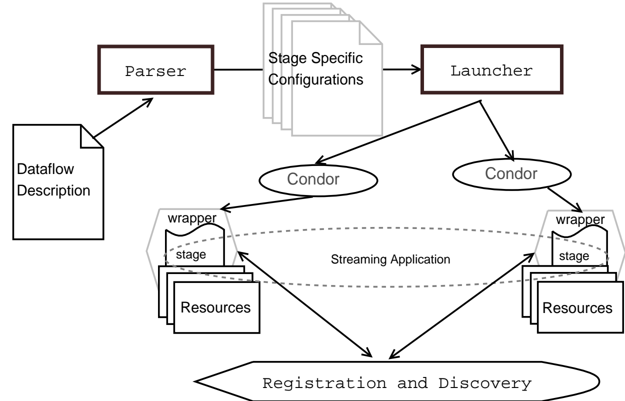

Figure 2. E-Condor Architecture

For each of these functions, we use the computation and communication numbers reported in a companion paper [28], and summarized in Table 3. These numbers are the result of profiling these functions on a StrongARM SA-1110 processor. We construct a pipeline consisting of these functions to serve as the workload for our scheduling experiments[2].

	CPU Cycles	Datasize(Bytes) (I/0)
Collage	803.4K	112K/112K
EdgeD	2616.2K	56K/56K
MotionD	1009K	56K/56K
FD/FR	1959M	30K/30K

Table 3. Computation, Communication Costs of Basic Image Processing Functions

[2]Note: Although our experimental setup (see Section 5.5) uses an x-86 cluster, the use of these numbers is justified since we are only interested in relative performance of the different schedulers on the same dataflow graph.

5.5 Modeling Resource Contention

We define four control variables to model the contention and non-uniformity of resource availability.

- Mean processing availability (μ_p): This is is the average CPU cycles available across all the nodes. We normalize it by the maximum CPU availability so that this is a number between 0 and 1.
- Mean network bandwidth availability (μ_{bw}): This is the average end-to-end network bandwidth available across all pairs of nodes. We normalize it by the maximum network bandwidth availability between any two nodes so that this is a number between 0 and 1.
- Variance in processing availability (σ_p^2): This is the variance in CPU cycle availability across all the nodes.
- Variance in network bandwidth availability(σ_{bw}^2): This is the variance in end-to-end network bandwidth availability across all pairs of nodes.

μ_p and μ_{bw} are indicators of the amount of resource contention in the system. σ_p^2 and σ_{bw}^2 are indicators of the non-uniformity of the load distribution in the system. Higher variance implies resources vary widely from one another in their load characteristics. Clearly, the possible sets of values for these four control variables are quite large. Thus, to keep the scope of the experimental study manageable, we study the performance for a chosen subset of values. Further, to keep the discussion simple as well as to understand the effects of these control variables better, we separately study CPU and network contention.

In the experimental study, we fix three of the control variables and study the effect of the fourth on the performance. We assign the resource settings (i.e. CPU and network bandwidth) to correspond to the desired value of the control variable. We experimented with 3 different values for fixed variables in each study. Since more than one assignment can yield the same value of the control variables, we experimented with 3 resource assignments chosen at random to minimize the effect of any particular choice. For each experiment, we picked 7 data points for the control variable under study. The experiments were performed for each of the two "kernels" introduced below, for the 5 algorithms (Streamline, Optimal, Simulated Annealing (SA1, SA2), and E-Condor) with multiple runs corresponding to each datapoint. Because of our controlled settings, we observed variance across different runs to be negligible ($<$ 0.01%). We present a representative subset of the results to keep the presentation within limit.

Our experimental platform consists of a sixteen node cluster with dual gigabit-Ethernet interconnects. Each node consist of eight Pentium III 550MHz Xeon processors with 4GB RAM. The intent of the scheduling experiments is to determine the quality of node selection by a scheduler commensurate with the application requirements and available resources. To model resource contention in a controlled manner in our experiments we adopted the following strategy. We introduce a synthetic delay in the code for a stage commensurate with the (assumed) load on the node that it is running on (parameterized by the assigned setting for that node). This strategy helps simulate non-uniform processor bandwidth availability in a controlled manner. Similarly, in order to model non-uniform network bandwidth availability, we inflate the data size of the data communication between stages in proportion to the level of (assumed) network contention (once again parameterized by the assigned setting).

5.6 Micro Measurements

For the micro measurements, we use 4 nodes of the cluster (one processor in each node). We use two "kernels" of a distributed surveillance application in these measurements. There are two sets of measurements, one for a compute-bound kernel and the other for a communication-bound kernel.

5.6.1 Compute-Bound Kernel

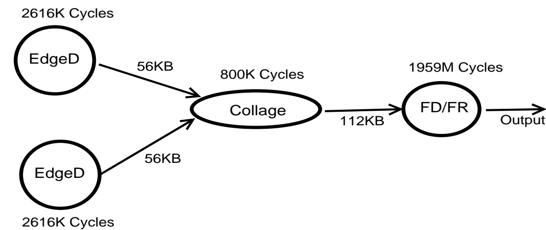

Figure 3. Kernel of Distributed Surveillance Application (Compute Bound)

Figure 3 shows the compute-bound kernel. The scheduler maps the 4 stages of the pipeline to the 4 nodes of the cluster. As we mentioned earlier, we assign the settings (CPU load and network contention) on the processor and the links commensurate with the control variable value for a particular experiment. The metric for comparison of different scheduling algorithms is the average time taken per output data item averaged over 100 data items.

Control variable: CPU Variance. Figure 4 shows the performance of different schedulers when CPU load distribution is non-uniform. It can be concluded from the graph that: (*i*) Performance of Optimal and Simulated Annealing algorithms (SA1, SA2) are comparable, except for one instance where SA1 performance is close to E-Condor, (*ii*)

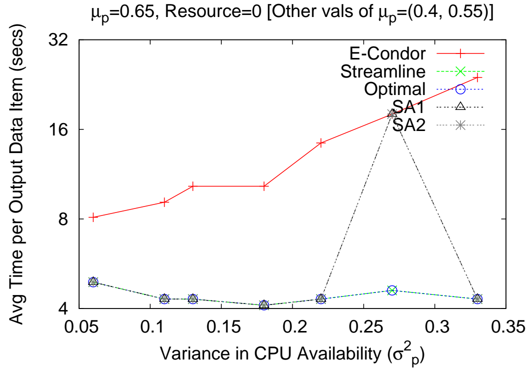

Figure 4. Effect of CPU Availability Variance on Compute Bound Kernel (μ_p = 0.65, σ^2_{bw} = 0, μ_{bw} =1)

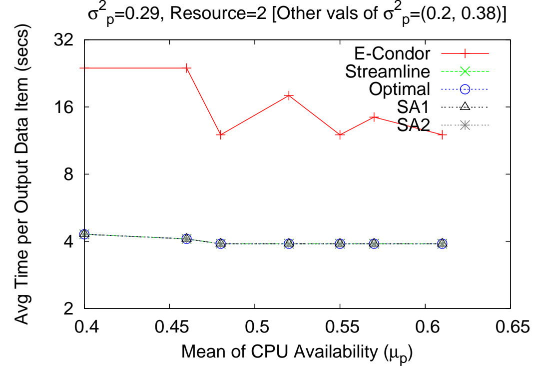

Figure 5. Effect of Mean CPU Availability on Compute Bound Kernel (σ^2_p=0.29,σ^2_{bw} = 0, μ_{bw} =1)

Streamline performs close to Optimal and nearly an order of magnitude better than E-Condor under highly non-even load distribution, and (*iii*) for a given system load (μ_p) the performance of Streamline (like Optimal and Simulated Annealing) *improves* with increasing variance whereas that of E-Condor degrades. E-Condor does not take into account the resource requirements of each stage of a dataflow graph but simply allocates resources to the stages in the order the requests are submitted. On the other hand, Streamline is able to match the variance in the computational requirements of the stages in the application dataflow graph with the variance in the system load to get a better mapping of the stages to the resources. The results were similar for other values of μ_p (0.4, 0.55) and 3 resource configurations for each particular combination of control variables. The degradation in performance of SA1 in one instance is attributed to its neighbor selection policy, though we observed that SA1 performed close to SA2 in other configurations for this experiment.

Control variable: CPU Availability. Figure 5 shows the effect of mean CPU availability for a fixed variance. The results show that performance of Streamline, Optimal, and Simulated Annealing algorithms are indistinguishable. We also observe that with increase in mean CPU availability, the performance of both E-Condor and Streamline improves. However, we see that like Optimal, and Simulated Annealing, Streamline does not benefit as much with increase in mean CPU availability as E-Condor does in some cases. The reason is quite intuitive. Streamline takes advantage of the variance in resource availability in its heuristic when it allocates "best" resource to the most needy stage as determined after stage prioritization; therefore, small increases in the mean CPU availability has little impact in its placement decision and hence in the overall performance. E-Condor, on the other hand, due to its first-fit approach, has severe performance penalty when the mean CPU availability is low. Similar results were observed for other values of σ^2_p (0.2, 0.38) and 3 resource configurations for each combination of control variables.

Control variable: Network Bandwidth Variance. Just for completeness, we also measure the effect of variance in available bandwidth on the compute bound kernel. As the results in Figure 6 show, none of the algorithms is affected by the variation in available bandwidth, thereby confirming the computational nature of this kernel. We observed similar results for other values of μ_{bw} (0.34, 0.74) and all 3 resource configurations for each combination of control variables.

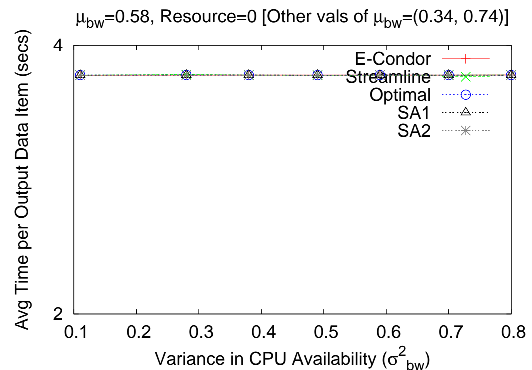

Figure 6. Effect of Variance in Bandwidth Availability on Compute Bound Kernel (μ_{bw} =0.58,σ^2_p=0, μ_p = 1)

We also conducted experiments studying the effect of

network bandwidth availability for 3 different fixed values of σ^2_{bw} (0.32, 0.58, 0.76). These results show (not presented here) the performance of all algorithms to be equivalent for the compute bound kernel, as we expected.

In summary, we observe that for the compute bound kernel, performance of Streamline is close to Optimal and SA2, and an order of magnitude better than E-Condor. Moreover, the performance of SA1 is close to Optimal in most cases. This implies that for the compute bound kernel, even a Simulated Annealing algorithm that examines fewer states (SA1) performs close to Optimal. The reason for this is that for the compute bound kernel, the performance of a particular stage is not dependent on the placement of other stages. Therefore, even a simple random neighbor selection policy performs well.

5.6.2 Communication-Bound Kernel

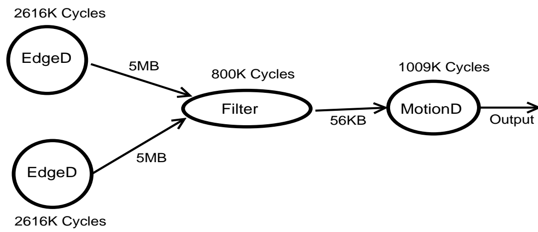

Figure 7. Kernel of Distributed Surveillance Application (Communication Bound)

Figure 7 shows the communication-bound kernel used for this set of micro measurements. In this kernel, a synthetic *Filter* function is used. The filter function receives large amounts of data (video images plus their boundaries) from two *edge detectors*. The filter function selects the subset of the input to send on to a *motion detector* for higher level inference.

Control variable: Network Bandwidth Variance. Figure 8 shows the effect of non-uniform bandwidth for the five schedulers. We observe that performance of SA1 is in between E-Condor and Optimal. This demonstrates that for commmunication bound kernel, a Simulated Annealing algorithm needs to explore a relatively larger part of the search space in order for its performance comparable to Optimal as in SA2. In contrast, Streamline heuristic performs close to Optimal and better than SA1. In addition, Streamline out-performs E-Condor by a factor of four under high variance. We also note that Streamline's performance improves under non-uniform load condition due to efficient placement of stages. We observed similar results for other values of μ_{bw} (0.34, 0.74) and all 3 resource configurations for each combination of control variables.

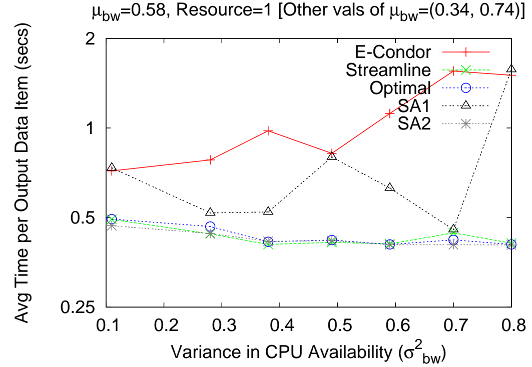

Figure 8. Effect of Bandwidth Availability Variance on Communication Bound Kernel (μ_p = 1, σ^2_p = 0, μ_{bw} =0.58)

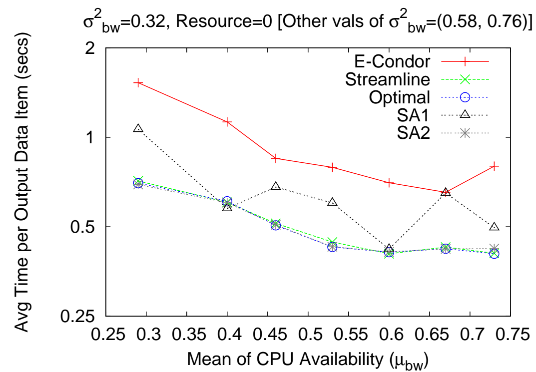

Figure 9. Effect of Mean Bandwidth Availability on Communication Bound Kernel (σ^2_p=0,μ_p = 1, σ^2_{bw} = 0.32)

Control variable: Network Bandwidth Availability. Figure 9 shows the effect of network bandwidth availability on the performance of the five schedulers. The performance of all the algorithms improve in general with increase in mean bandwidth. However, we observe that SA1 performs worse than SA2 in many cases signifying the importance of neighbor selection and length of annealing schedule of a Simulated Algorithm, particularly for communication intensive dataflow graphs. Streamline, in contrast, performs close to Optimal and SA2 in all instances. The performance advantage of Streamline is attributed to the heuristic algorithm in which we take into account computation and communication requirements of different stages as well as dynamic resource availability. We observed similar results for other values of σ^2_{bw} (0.58, 0.76) and 3 resource configurations for each combination of control variables.

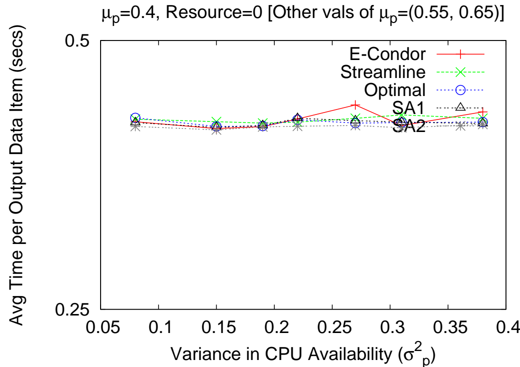

Figure 10. Effect of CPU Availability Variance on Communication Bound Kernel (μ_p = 0.4, σ^2_{bw} = 0, μ_{bw} =1)

Control variable: CPU Variance. Figure 10 shows that the communication bound kernel gives similar performance under varying computational load conditions for all the algorithms, thus confirming its communication intensive nature. We observed similar results for other values of μ_p (0.55, 0.65) and 3 resource configurations for each combination of control variables. We also observed that CPU availability has little effect on the relative performance of all the algorithms for the communication bound kernel (graphs not presented here).

Through these micro measurements we have established that Streamline performs significantly better than E-Condor in general, and especially under non-uniform load conditions. Moreover, performance of Streamline is comparable to the Optimal and SA2 algorithms. We also establish that for a communication bound kernel, SA2 performance is better than SA1, thereby illustrating the relative importance of *neighbor selection* and *annealing schedule* strategies for communication intensive dataflow graphs.

In the next subsection, we evaluate the scalability of Streamline scheduler in handling a large dataflow graph.

5.7 Scalability

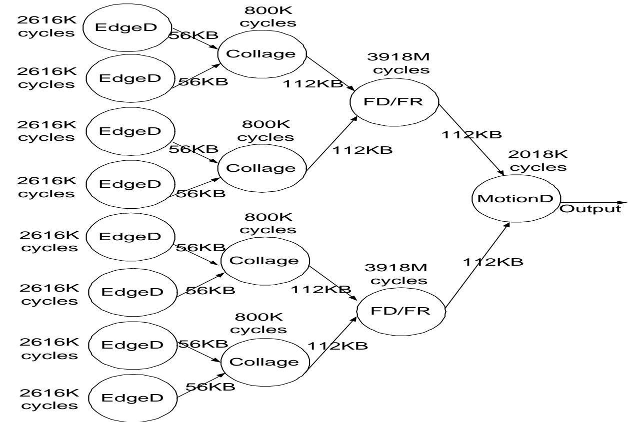

Figure 11. Video-based Tracking Dataflow Graph

For the scalability study, we consider a video-based tracking application where multiple camera feeds are analyzed to identify any suspicious activity. We build a representative dataflow graph for a video-based tracking application by combining our basic building blocks, *Collage*, *EdgeDetect*, *MotionDetect*, and *FD/FR* as shown in Figure 11. The application represents a scenario where streaming data from sensors are fed into an edge detector, merged near the source and are processed through successive stages of face detection, recognition and motion detection in order to derive some higher level hypothesis.

We measure the average time taken per output data item for a 3 stage (2 *EdgeD*, 1 *Collage*), 4 stage (2 *EdgeD*, 1 *Collage*, 1 *FD/FR*), 7 stage (4 *EdgeD*, 2 *Collage*, 1 *FD/FR*) and 15 stage (8 *EdgeD*, 4 *Collage*, 2 *FD/FR*, 1 *MotionD*) dataflow graph with different algorithms. We performed the experiments on 15 nodes (1 processor in each node) with a particular choice of control parameters (μ_{bw} = 0.55, σ^2_{bw} = 0.49, μ_p = 0.55, σ^2_p = 0.27) that falls within the range used in the micro measurements, so as not to bias the experiment in favor of any one particular algorithm. Because of the computational complexity of the Optimal algorithm, we did not evaluate it for the 15 stage dataflow graph.

The results (see Figure 12) show that (*i*) because of the introduction of a computation intensive stage *FD/FR*, all algorithms show an increase in the average time per output data item, when the number of stages increases from 3

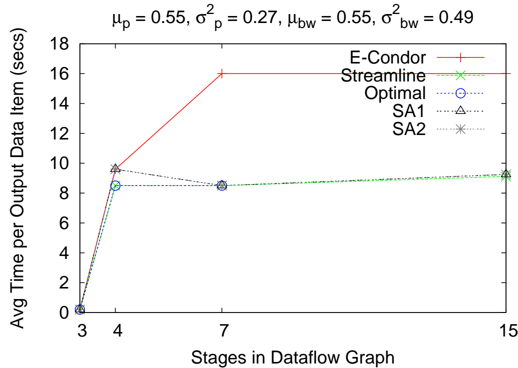

Figure 12. Effect of Increase in Number of Stages on Average Time per Output Data Item

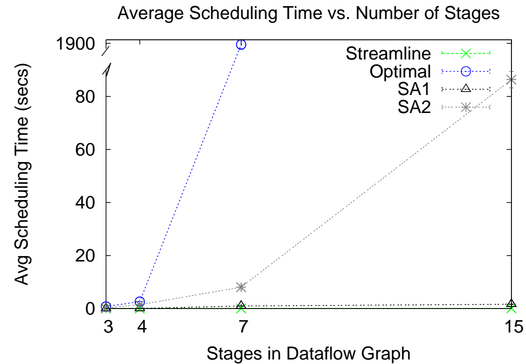

Figure 13. Effect of Increase in Number of Stages on Scheduling Time

to 4; (*ii*) Streamline performs close to Optimal and about 12% better than SA1 and SA2 for 4 stage dataflow graph. This degradation in Simulated Annealing performance is attributed to our neighbor selection policy when the number of resources is more than the number of stages; (*iii*) For the 7 stage dataflow graph, E-Condor takes close to 50% more time than Streamline; (*iv*) Streamline performs close to Simulated Annealing algorithms even when number of stages more than doubles from 7 to 15.

This demonstrates that as the number of stages increase, Streamline is able to allocate resources better by taking into account non-uniform resource availability and the application needs. We also observe that SA1 performs as good as SA2 due to the computation intensive nature of the scalability graph, in consistent with our findings from the micro measurements.

Finally, we compare the relative time taken by the different scheduling algorithms for the dataflow graph presented above. We measured the scheduling time of Streamline, SA1 and SA2 for the 3, 4, 7, and 15 stage dataflow graph. We also report the scheduling time for the Optimal algorithm for 3, 4, and 7 stage dataflow graphs. The numbers presented, in Figure 13, are an average over 15 consecutive scheduling runs.

Frpm Figure 13, we observe that Streamline has very small execution time (91 milliseconds for 15 stage dataflow graph). The execution time scalability of SA1 is comparable to Streamline, whereas SA2 takes much longer (88 seconds) for 15 stage dataflow graph. The variance in execution time across different runs was observed to be very small ($<$ 3.5% for SA2, $<$ 2.5 % for SA1, and $<$ 0.01% in case of Streamline and Optimal). We conclude that Streamline has performance comparable to SA2 with much smaller execution time, making it suitable for dynamic environment of the grid.

6 Related Work

Scheduling in grid has primarily focused on providing support for batch-oriented jobs (See Nabrzyski, et al. [22] for a survey of current grid schedulers). A wide variety of meta-schedulers and resource brokers using Globus Toolkit [14] have been developed by other research projects such as Condor[4], Legion[7], and Nimrod-G [5]. Most of these schedulers developed out of needs to support scientific batch-oriented applications. As we mentioned earlier, such schedulers do not address the needs of streaming applications. However, through the E-Condor architecture we have shown how these batch schedulers can be used to allocate resources for streaming applications. We have used E-Condor, in addition to Optimal and Simulated Annealing algorithms, as a baseline scheduler in experimental evaluation of Streamline.

Middleware efforts for streaming applications have started gaining attention in the grid community only recently. GATES [9] provides middleware-support for dynamically adapting a streaming application based on the observed processing rates in individual stages. A companion paper [8] describes a middleware for deploying the stages of a generic application, processing stream data, to grid resources. The middleware uses the available communication bandwidth among the nodes to determine an assignment that will result in the best use of the resources under the assumption that earlier stages of the pipeline would need more communication bandwidth. Streamline is a more comprehensive framework for scheduling a streaming application to grid resources taking into account the application characteristics as well as the computational resources available from the grid.

At some level, coarse-grain dataflow graphs of streaming applications resemble task-graphs that have been the focus of multiprocessor scheduling work from the 70’s. The ob-

jective in task-graph scheduling is to minimize the total execution time of the application (represented as a task-graph) on a multiprocessor. The classical approach, called *list scheduling* [2, 11], (and its variants [17, 23, 18]), creates an ordered list of task-graph nodes by assigning them priorities based on certain properties. These priorities are then used to assign the tasks to processors such that each task is started at the earliest possible time commensurate with its priority. The specifics of the algorithms vary in the way priorities are assigned to task-graph nodes including HLF (Highest Level First), LP (Longest Path), LPT (Longest Processing Time), and CP (Critical Path) [16, 24, 20]. The task-graph scheduling problem has also been studied by some research groups for systems with non-uniform resource availability [26, 27]. The scheduling of streaming applications on the grid differs from task-graph schedulers in many ways. First, streaming applications are continuous in nature; therefore, all the stages of the application have to be scheduled to run concurrently. Second, the grid framework does not allow the level of control over the individual resources (for e.g. the operating system scheduler on a node) as assumed by such multiprocessor scheduling work. Third, there could be significant non-uniformity of computational resources (processing and communication bandwidths) leading to additional complexity in resource allocation on the grid.

Stream processing has also been the focus of recent database research [10, 1, 3, 6]. Tools and techniques for the efficient handling of "continuous queries" on stream data are the objectives in such work, while our work focuses on scheduling streaming applications on grid resources.

7 Conclusion

In this paper, we have presented a scheduling heuristic, *Streamline*, that takes as input (a) computation and communication requirements of the various stages of a streaming application represented as a coarse-grain dataflow graph, (b) any application-specified constraints, and (c) the current resource (processing and bandwidth) availability. We have designed Streamline over an existing grid framework using Globus Toolkit [14].

We have compared Streamline with an Optimal placement and Simulated Annealing algorithms. In addition, to serve as a baseline for a comparative study, we have also developed a streaming application scheduler, *E-Condor*, built using existing grid scheduler Condor[4]. We have performed experimental studies and shown that Streamline performs close to Optimal and outperforms E-Condor by nearly an order of magnitude on compute-bound kernels under non-uniform CPU availability, and by a factor four on communication-bound kernels under non-uniform network bandwidth availability. Through two different choices of parameters for Simulated Annealing, we have also shown that the choice of neighbor selection and annealing schedule of a Simulated Annealing algorithm has a relatively larger impact for a communication intensive dataflow graph than for a computation intensive dataflow graph. The study has also shed light on the scalability of Streamline for large-scale streaming applications and shows its performance to be close to Simulated Annealing algorithm, with smaller scheduling time.

While Streamline does the placement for such streaming applications, it is clear that the application dynamics may result in the computation and communication characteristics of the application changing over time. Perhaps even the dataflow graph of the application could change over time with the addition and deletion of new stages to the pipeline. If the application characteristics that are profiled and used in the placement are considered the "typical" (for e.g. mean) values for the respective stages, then the mapping given by Streamline would result in an acceptable level of performance despite this dynamism. Nevertheless, it is important to consider the impact of the application dynamism and the consequent adaptation of the scheduling heuristic. Such a adaptive scheduling heuristic is part of our future work.

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, August 2003.

[2] T. Adam, K. Chandy, and J. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, Dec 1974.

[3] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[4] R. Boer. Resource management in the Condor system. Master's thesis, Delft University of Technology, 1996.

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

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD'03)*, 2003.

[7] S. J. Chapin, D. Katramatos, J. Karpovich, and A. drew S. Grimshaw. The Legion resource management system. In *Job Scheduling Strategies for Parallel Processing*, pages 162–178. Springer Verlag, 1999.

[8] L. Chen and G. Agrawal. Resource allocation in a middleware for streaming data. In *2nd Workshop on Middleware*

for Grid Computing (MGC'04), Toronto, Canada, Oct 18 2004.

[9] L. Chen, K. Reddy, and G. Agrawal. GATES: A grid-based middleware for processing distributed data streams. In *Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-13)*, Honolulu, Hawaii USA, June 4-6 2004.

[10] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Asilomar, CA, January 2003.

[11] E. Coffman. *Computer and Job-Shop Scheduling Theory*. Wiley, New York, 1976.

[12] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, August 2001.

[13] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[14] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[15] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. "condor-g: A computation management agent for multi-institutional grids". In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, pages 55–63, San Francisco, CA, August 2001.

[16] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling dag s on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, Dec 1992.

[17] R. Graham, E. Lawler, J. Lenstra, and A. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[18] T. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, Nov 1961.

[19] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220, 4598:671–680, May 1983.

[20] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors:. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, 1996.

[21] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. . Teller, and E. Teller. Equations of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087–1091, 1953.

[22] J. Nabrzyski, J. M. Schopf, and J. Weglarz. *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, Sep 2003.

[23] C. Ramamoorthy, K. Chandy, and M. Gonzalez. Optimal scheduling strategies in a multiprocessor system. *IEEE Trans. on Computers*, C-21(2):137–146, Feb 1972.

[24] G. Sih and E. Lee. "a compile-time scheduling heuristic for interconnection- constrained heterogeneous processor architectures". *IEEE Trans. on Parallel and Distributed Systems*, 4(2):75–187, Feb 1993.

[25] V. Talwar, S. Basu, and R. Kumar. An environment for enabling interactive grids. In *12th International Symposium on High-Performance Distributed Computing (HPDC'03)*, pages 184–193, Seattle, WA, June 2003.

[26] H. Topcuoglu, S. Hariri, and Min-YouWu. Task scheduling algorithms for heterogeneous processors. In *Proceedings of the 8th Heterogeneous Computing Workshop*, pages 3–14, 1999.

[27] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[28] M. Wolenetz, R. Kumar, J. Shin, and U. Ramachandran. Middleware guidelines for future sensor networks. In *First Workshop on Broadband Advanced Sensor Networks (BASENETS'04)*, San Jose, CA, Oct 2004.

[29] R. Wolski. Dynamically forecasting network performance using the network weather service. *Journal of Cluster Computing*, 1:119–132, January 1998.