A Path Selection Based Algorithm for Maximizing Self-satisfiability of Requests in Real-Time Grid Applications

Mohammed Eltayeb[1], Atakan Doğan[2], and Füsun Özgüner[1]

[1] Department of Electrical Engineering, The Ohio State University,
2015 Neil Avenue, Columbus, Ohio 43210, USA
`{eltayeb,ozguner}@ece.osu.edu`
[2] Anadolu University, Department of Electrical and Electronics Engineering
26470 Eskişehir, Turkey
`atdogan@anadolu.edu.tr`

Abstract. Efficient data scheduling in Grid environments is becoming a seemingly important issue for distributed real-time applications that produce and process huge datasets. Thus, in this paper, we consider the data scheduling problem so as to provide reliable dissemination of large-scale datasets for the distributed real-time applications. We propose a new path selection-based algorithm for optimizing a criterion that reflects the general satisfiability of the system. The algorithm adopts a blocking-time analysis method combined with a simple heuristic (LCSP or SLCP). The simulation results show that our algorithm outperforms the algorithms existing in the literature.

1 Introduction

Research in real-time *Grid* computing is needed to enable Grid services for newly emerging class of *large-scale* real-time distributed applications. The amount of data produced and processed by these new large-scale applications poses a great challenge on the Grid infrastructure. Let us consider the following example. Assume a distributed industrial vision and inspection system that provides complex and sensitive inspection for industrial facility lines [1]. The vision equipments provide images for the product, which needs to be analyzed, matched, verified and stored in *real-time* fashion. A distributed computing system connected by a wide area network, then, provides an efficient computing environment for distributed inspection tasks on the datasets [2]. The *large-scale* datasets may include a combination of real-time still pictures, thermal images, video clips, etc. Such a system requires transferring huge datasets between distributed running tasks. Due to the fact that the datasets are large, an efficient mechanism must be devised to allow dataset transfer between tasks in remote locations. This mechanism must cater for cases by which a particular dataset is requested by more than one task in different locations. They also must account for future requests of a particular dataset and certainly the *deadline* by which the dataset is to be delivered to the final destination(s).

Examples of some other distributed real-time applications that share similar features of the industrial vision system include distributed medical information and im-

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 468–475, 2004.

aging systems [3], computer vision [4], and distributed surveillance applications [5]. In [5], three different heuristics, referred to as PPH, FPH and FPA (Partial Path Heuristic, Full Path Heuristic, and Full Path All destinations heuristic, respectively), were proposed for data transfer scheduling with real-time constraints for a defense information system. In [5], deadlines for requests, each of which represents a data transfer, were assumed. The goal was to minimize the number of requests that miss deadlines. Dissemination of datasets was achieved by adopting a data *staging* technique by which a transferred data-item is cached in intermediate nodes along the path of the transfer from the source to the destination of the transfer.

The three aforementioned heuristics (PPH, FPH and FPA) schedule only one request for transfer along the shortest path from source to destination in each iteration. *Concurrent Scheduling* (CS), on the other hand, as proposed in [6], allows a communication step to include different request transfers simultaneously in an organized fashion. This is possible because some requests may be achievable through separate paths. The CS algorithm was built on top of the EPP (Extended Partial Path) heuristic proposed in [7].

Data replication problem in Grid is related to the dissemination problem and has been studied to minimize the latency of data transfer as well as to reduce bandwidth consumption, improve the system reliability and to load balance the requests [8], [9].

In this paper, we facilitate a discussion for real-time Grid computing and issues of data dissemination and scheduling requirements for large-scale data in distributed real-time applications. We propose an efficient algorithm for the data dissemination of these applications.

2 System Model

Our goal is to provide a solution to the data dissemination problem for applications with large-scale datasets. Thus, we focus on the scheduling the transfer of datasets rather than scheduling the application tasks. In our system model, we assume that applications arrive *aperiodically* online at a specific point (broker and scheduler) by which the tasks of the applications are mapped and scheduled on the distributed resources. The distributed tasks require large dataset transfers from remote locations that are determined upon the arrival of the applications. Each task may have one or more requests for data transfer.

- The network graph $\boldsymbol{G} = (V, E)$ specifies the connectivity of a set of n vertices $V = \{V_1,\ldots,V_n\}$ and m edges $E = \{E_{i,j}: V_i, V_j \in V$ and there is a communication channel between the vertices$\}$. Each vertex V_i is a node with limited storage capacity C_i. Each edge $E_{i,j}$ represents a time delay for the transfer between the end vertices V_i, V_j. This delay is assumed to be constant on $E_{i,j}$.
- A distributed application $A_j = \{T_{j,1}, \ldots, T_{j,kj}\}$ is composed of k_j tasks running on several predetermined processing nodes. Each task produces a set of *requests* to specific data-items of large, fixed sizes at different times during its execution.
- $R_{Tj,i} = \{r^j_{i,1},\ldots,r^j_{i,li}\}$ is the set of l_i requests produced by the ith task $T_{j,i}$ of application A_j.

- Each request $r^j_{i,u}$ is associated with one of χ data-items I_t (t =1,…,χ) to be transferred to a *destination* node $N^j_{i,u}$ where the corresponding requesting task $T_{j,i}$ resides.
- Each request $r^j_{i,u}$ is assigned a deadline $Dl(r^j_{i,u})$ by which the data-item must be delivered to its destination.
- The request is also assigned a priority value $Py(r^j_{i,u})$ which is inherited from the application, which includes the task that produced the request. A request $r^j_{i,u}$ is hence summarized by the following tuple: $\langle I_t(r^j_{i,u}), N^j_{i,u}, Dl(r^j_{i,u}), Py(r^j_{i,u})\rangle$.

An *achieving* path $P_{r^j_{i,u}}$ of a request $r^j_{i,u}$ is defined here as a path that has a network latency less than or equal to the deadline of the request. An achieving path is also assumed to be *simple*. A simple path between a source V_S and a destination V_N ($V_S \neq V_N$) is given as $P = \{V_S, \ldots, V_i, \ldots, V_j, \ldots, V_N\}$ and $i \neq j$ for all $V_i, V_j \in P$. The set of achieving paths of a request $\mathcal{P}_{r^j_{i,u}}$ is defined as the collection of all achieving paths from all sources of the data-item associated with the request $r^j_{i,u}$. We also assume that the arrival time of a request is its release time.

Our model also assumes a staging mechanism for data transfers which was presented in [5]. The data-item associated with a request will be stored in intermediate nodes for the duration of the deadline of the request associated with the transfer.

3 Problem Statement

Our goal is to satisfy all requests of all tasks present at any specific point of time. These requests form a batch of data-item transfers with specific deadlines. Due to the size of the individual data-items and the storage capacity of the intermediate nodes, it is not possible to accommodate all of the staging at the same time. We are also restricted by the need to allow multiple copies of the data-item exist during a specific time period for a request. An efficient heuristic should aim at maximizing the satisfiability at all times.

Let δ be a specific schedule of data transfers. A request $r^j_{i,u}$ is *satisfied* in δ if and only if the data-item associated with $r^j_{i,\mathrm{u}}$ is delivered at the destination node $N^j_{i,u}$ on or before the deadline $Dl(r^j_{i,u})$. Let the set of satisfiable requests by the schedule δ be defined as $S(\delta) = \{r^j_{i,u}: r^j_{i,u}$ is *satisfied* in $\delta, \forall\, j, i, u\}$. The optimization criterion of the staging heuristic is provided by the *effect* of the schedule δ which is defined as:

$$E(\delta) = \sum_{r^j_{i,u} \in S(\delta)} Py(r^j_{i,u}) \tag{1}$$

4 Blocking Analysis Concurrent Scheduling Algorithm (BACS)

The heuristic proposed in this paper employs a data *concurrent scheduling* (CS) method and a data *blocking analysis* (BA) method (hence the name BACS) for data transfers to solve the data-scheduling problem. By concurrent scheduling, we mean that several data-items will be allowed to stage, allowing the service of multiple requests simultaneously. By blocking analysis, we mean that the delays encountered by the transferred data-items (due to blocking for intermediate storage) will be computed and used for assigning staging paths.

A blocking along the path occurs when a request competes with another request in one or more of the intermediate nodes due to limited capacity. Due to this situation, BACS enforces a special *blocking policy*. This policy compels the lower priory requests to *await* before the specific blocking point (a contention node on the path) until a space adequate for its data-item is available in the contention node. The lower priority request, in such a case, is called an *awaited* request or a *blocked* request. An awaited request will be blocked at a specific contention node for at least the amount of time needed to clear the node from the higher priority request.

4.1 BACS with Shortest Least Contending Path (SLCP) First Heuristic

The BACS algorithm attempts to generate an optimum set of paths for the individual requests. BACS iterates through three phases of execution. The algorithm starts execution by accepting a batch of requests, each defined by a tuple ‹$I_t(\mathrm{r}^j_{i,u})$, $N^j_{i,u}$, $Dl(\mathrm{r}^j_{i,u})$, $Py(\mathrm{r}^j_{i,u})$ ›. BACS, then, finds a set of achieving paths for each request from a set of multiple sources. This is accomplished by running a version of Dijkstra's shortest path algorithm for each request which can find a shortest path to a specific node in the network from multiple sources [5], [10]. These paths are later sorted based on their lengths for each request.

BACS generates an initial set of paths composed of the shortest path for each request. It is obvious that this set neither guarantees the satisfiability of all requests nor maximizes this satisfiability. The reason is simply enforced blocking policy by which lower priority requests must be blocked for higher priority requests for an amount of time that is proportional to the individual path lengths.

From the initial set of paths, BACS uses a graphical method to compute the total delay incurred by each request, which is the *first phase* of the BACS:

- Determine the effective priorities for all requests. They are computed as the weight of the task multiplied by the weight of the application:

  $$Py(\mathrm{r}^j_{i,u}) = Py(A_j) \times Py(\mathrm{T}_{j,i}) \tag{2}$$

  where $Py(r^j_{i,u})$ is the effective priority of $r^j_{i,u}$, $Py(T_{j,i})$ is the priority of task $\mathrm{T}_{j,i}$, and $Py(A_j)$ is the priority of application A_j.
- Compute the direct blocking delays between all possible request pairs as explained in [11]. The amount of time a lower priority request will be blocked at a contention point is determined by the time needed for the higher priority request to clear its path (clear the intermediate nodes on the request's path).
- Finally, in this step, we develop the *Blocking Dependency Graph* (BDG) for the requests. Each node in this graph represents a request and each directed edge

represents the awaited time incurred on a request by a higher priority request sharing the same path as if the two are the only requests in the system. The developed BDG represents the dependencies between the requests.

Once we have a BDG, it is possible to compute the total end-to-end delays of all available requests in the batch, which is the *second phase*. This delay for each request is the length of the *critical path* of the request in the resultant DAG and must be equal to or less than the corresponding request's deadline in order for the request to be satisfied. The critical path of a request is the longest path to the request from all available nodes in the graph and its length represents the total blocking time.

Once the blocking delays are found, it is possible to compute the effect of the schedule from defined by (1). BACS then checks for the total satisfiability condition shown by the diamond in the second column. If the condition is not satisfied, the algorithm performs a path set modification phase.

In the *third phase*, BACS finds a subset of requests which is referred to as the set of *candidate requests*. This set is composed of all requests (represented by nodes) in the critical path of only unsatisfied requests (or some of the unsatisfied requests). Then, BACS attempts to modify the path set for the candidate requests. By changing the path of a higher (blocking) request, it is possible that the delay incurred by a lower (blocked) request is reduced. The following is performed in this phase:

- Starting from the highest requests in the chain, the algorithm searches for an alternative path.
- The alternative path is the *least contending* based on the *Contention Index* (CIX) function defined in [11] among all achieving paths of the request.
- If two requests with same contention amount exist the shorter is picked first (hence SLCP).

4.2 Least Contending Shortest Path (LCSP) First Heuristic

In the LCSP heuristic, the algorithm iteratively replaces the paths for each request in the candidate set produced in the third phase of the algorithm. The following steps are performed by this heuristic until reaching a feasible solution or exhausting the set:

1. Starting with the highest priority request in the set of candidates, LCSP replaces its current path with the next shortest path. If two paths are of the same length the heuristic selects the *least contending* of the two based on the CIX value defined in [11]. Note that computing the CIX value is performed only when arbitration is needed in this step.
2. The algorithm evaluates the total delays of the requests and computes the effect value by jumping to the first phase. If no better effect value is found, the next path for the current request is tested and so on. Once a better effect value is found for this request, the path of the particular request is fixed.
3. The heuristic moves to another request in the list of candidates and repeats Step 1 and Step 2 until a feasible solution is found or the candidate requests list is covered. The algorithm exits with the best solution found.

5 Simulation Results

The performance of the BACS algorithm is tested by simulation in a network of 30 machines with arbitrary topologies. These machines constitute the nodes which can be sources, destinations and/or intermediate storage locations. Each machine has a limited capacity equal to a data-item size and all data-items are of the same size. We tested the performance of the algorithm for the general situation in which the requests for a particular number of data-items are generated randomly by a subset of the 30 machines with random number of sources and destinations. Random requests are assumed to arrive at the centralized scheduling unit in batches. The parameters used to measure the algorithm performance are the number of requests in the batch and the deadlines of the requests (the urgency of the application). We allowed the load representing the number of requests to vary between 100 and 600 requests while the deadline is set to about 70% of the average path length (500 time units). The performance is measured as the percentage of satisfied requests as well as the effect value. The performance of algorithm was also tested by changing the level of urgency of the applications set be the deadline value. Here, we fixed the load at 500 requests in the batch (high load situation).

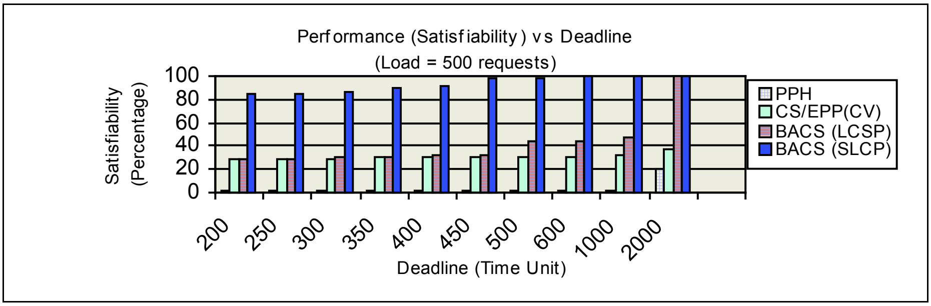

Fig. 1. The performance of the BACS, CS/EPP (CIX=CV) and PPH staging algorithms in terms of the percentage of satisfied requests at 500 request load.

Fig. 1 shows the performance of BACS, CS/EPP [6] and PPH [5] (FPH and FPA have been shown to have comparable performance with PPH even with different cost functions. See [5] for details). BACS shows better performance for the parameters set of the experiment. The PPH adheres to a method by which only one request is transferred at a time. This can result in high deadline miss rate for high load conditions. Although BACS was slightly better than CS/EPP in improving the number of satisfied requests, it showed considerable advantage over CS/EPP when the effect function was evaluated as shown in Fig. 2. This is mainly because BACS algorithm considers the entire batch and not only portions as in CS/EPP. BACS responds very well when deadlines are relaxed since many paths are considered for staging the requests. Fig. 3 and 4 show a comparison between the BACS and the CS/EPP when fixing the deadline and altering the load of the system. BACS shows superiority over CS/EPP especially at light load situations.

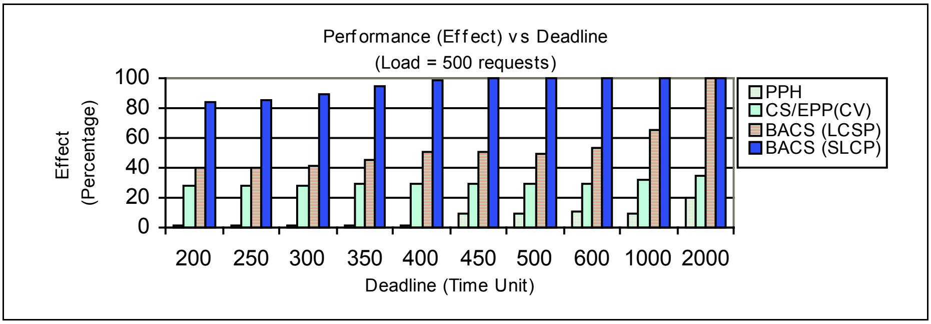

Fig. 2. The performance of the BACS, CS/EPP (CIX=CV) and PPH staging algorithms in terms of the percentage of the priorities of satisfied requests at 500 request load.

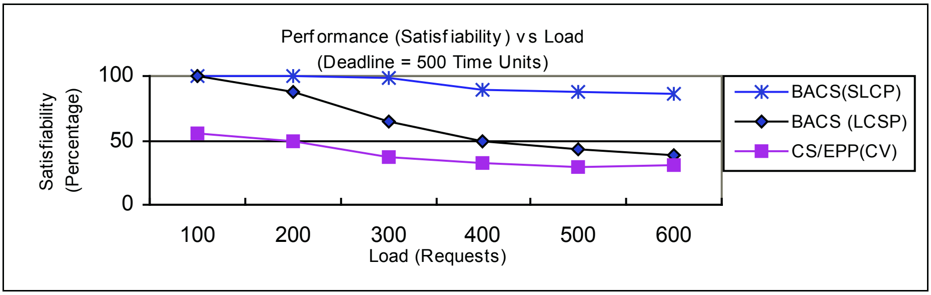

Fig. 3. The satisfiability performance of the BACS and CS/EPP as a function of the load.

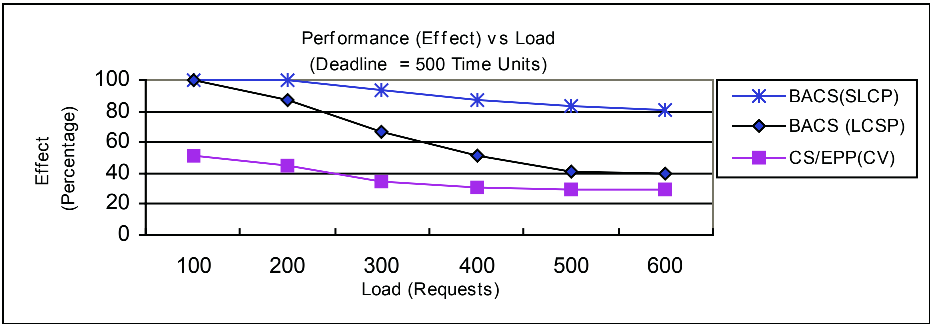

Fig. 4. The effect performance of the BACS and CS/EPP as a function of the load. The deadlines of all requests are fixed at 500 time units.

6 Conclusions

We have addressed the problem of data scheduling in distributed real-time systems with large-scale data communications and the need to consider real-time measures for Grid real-time applications in general. Our goal was to maximize the number of requests that meet their deadlines in a limited capacity environment by adopting the data staging scheme for the purpose of data dissemination. We proposed a path selection-based algorithm which maximized the objectives based on two new heuristics LCSP and SLCP. The performance of the BACS algorithm is shown by simulation to be superior to other static staging algorithms. BACS takes a batch of requests and generates a static schedule that is hopefully close to optimal. It is, however, clear that the complexity of BACS is higher than these other algorithms since multiple path search is performed for solving the problem.

References

1. Meliones, A., Baltas, D., Kammenos, P., Spinnler, K., Kuleschow, A., Vardangalos, G., Lambadaris, P.: A Distributed Vision Network for Industrial Packaging Inspection. High Performance Computing and Networking (1999) 1303-1307
2. Thomas, A., Rodd, M., Holt, J., Neill, C.: Real-Time Industrial Visual Inspection: A Review. Journal of Real-Time Imaging (1995) 139-158
3. Lee, J., Tierney, B., Johnston, W.: Data Intensive Distributed Computing: A Medical Application Example. High Performance Computing and Networking Conference (1999)
4. Shukla, S. B., Agrawal, D. P.: Scheduling Pipelined Communication in Distributed Memory Multiprocessors for Real-time Applications. In Annual International Symposium on Computer Architecture (1991) 222-231
5. Theys, M. D., Tan, M., Beck, N., Siegel, H. J., Jurczyk, M.: A Mathematical Model and Scheduling Heuristic for Satisfying Prioritized Data Requests in an Oversubscribed Communication Network. IEEE Transaction on Parallel and Distributed Systems, Vol. 11, No. 9. (2000) 969-988
6. Eltayeb, M., Do an, A., Özgüner, F.: Concurrent Scheduling for Real-time Staging in Oversubscribed Networks. The 16th International Conference on Parallel and Distributed Computing Systems (2003)
7. Eltayeb, M., Do an, A., Özgüner, F.: Extended Partial Path Heuristic for Real-time Staging in Oversubscribed Networks. The 18th International Symposium on Computer and Information Sciences (2003)
8. Lamehamedi, H., Shentu, Z., Szymanski, B. K., Deelman, E.: Simulation of Dynamic Data Replication Strategies in Data Grids. The Int’l Parallel and Distributed Processing Symposium (2003)
9. Ranganathan, K., Foster, I.: Decoupling Computation and Data Scheduling in Distributed Data Intensive Applications. The 11th Int’l Symposium for High Performance Distributed Computing (2002)
10. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. MIT Press, Cambridge, Massachusetts (1990)
11. Eltayeb, M., Do an, A., Özgüner, F.: A Data Scheduling Algorithm for Autonomous Distributed Real-Time Applications in Grid Computing. Int’l Conf. on Parallel Processing (2004)