TR TR-PDS-1996-009 March 1996

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Exploiting Data Locality For Multiprocessor Query Scheduling*

Aman Sinha Craig Chase

Parallel and Distributed Systems Laboratory
Electrical and Computer Engineering
University of Texas at Austin, Austin, TX 78712
{sinha,chase}@ece.utexas.edu

Abstract

We analyze the scheduling aspects of database queries submitted to an abstract model of a very large distributed system. The essential elements of this model are (a) a finite number of identical processing nodes with limited storage capacity, (b) a finite number of queries to be serviced, (c) a very large read-only data set that is shared by all queries and (d) a fixed inter-node communication latency. This framework models an important class of applications that use distributed processing of very large data sets. Examples of these applications exist in the very large database and multimedia problem domains. To meet the objective of minimizing flow time of queries while exploiting inter-query locality, various heuristics are proposed and evaluated through extensive simulation.

1 Introduction

1.1 Motivation for the Problem

In recent years, increasing processor speeds and distributed computing are making feasible query processing over very large databases. The size of the database in some applications can be several hundred gigabytes, or even several terabytes. Such a large data set virtually mandates distributed storage. In a distributed environment, several users could be querying the database in parallel. Further, these users would need to process parts of the data fetched by their queries, thus requiring sufficient cache space in the local disk. To design a cost-effective system, one must accept that most of the database will be on a remote data repository and not resident in the local disk. Yet, to design a high-performance system, one must avoid paying the high cost of network access time if at all possible.

If there is sufficient locality in the accesses to the database, we can have the majority of the data on one or more remote storage servers, but still communicate with the servers infrequently. In this paper we consider how to schedule queries to exploit this locality. To simplify the analysis, we consider only non-mutative, *i.e.,* read-only database accesses. Second, we assume that each query can be split into parallel subtasks. If the database consists of M data items, then we allow for the query to consist up to M separate subtasks. We assume the flexibility to schedule these subtasks in any order. These two assumptions are made because they

*This work was supported by NSF grant CCR-9409736. A condensed version of this paper appears in the *International Conference on Parallel Processing*, 1996, Chicago, Illinois

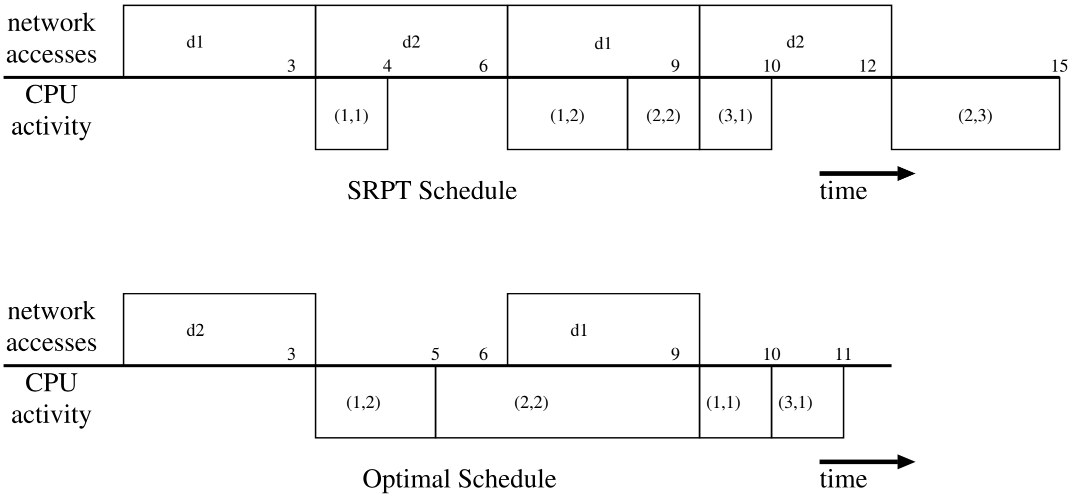
allow us the greatest possible flexibility in scheduling the computations so that data accesses will be minimized. We note that, although certainly not all queries meet these restrictions, many types of realistic data processing do meet them. In particular, we are targeting the Nile application, a NSF National Challenge problem in computational science for High Energy Physics research [18, 20]. This project involves distributed computation using read-only distributed data. The data set exceeds 10 Terabytes of experimental results from collisions of sub-atomic particles. A typical query operation involves first scanning the database for relevant records and then engaging in numerically intensive computation with those records. The output of the query is a set of statistics, for example, the number of particle collisions that resulted in the production of a new particle with a specific mass. Hence, the records can be processed in any order.

In this paper we choose our objective to be minimization of the *flow time*, *i.e.,* the response time, of a set of dynamically arriving queries. We illustrate the nature of the problem by considering the uniprocessor case. When all data is locally available, for deterministic processing times, the shortest processing time rule (SPT) will produce the minimum response time when release time of all queries are zero [19]. If the release times are non-zero, then a preemptive schedule based on the shortest *remaining* processing time (SRPT) will be optimal [19]. However, as we show in the example in Figure 1, this is not necessarily true when data must be accessed from a remote site. In this example, we assume a local disk capable of holding only one data item (*e.g.,* one record), and a total database size of two data items. The time required to fetch a datum from the server is a fixed 3 time units. Three queries are processed by the system. The following table shows the time each query spends processing each data item:

Query	d_1	d_2	release time
Q_1	1	2	0
Q_2	0	4	0
Q_3	1	0	8

Queries Q_1 and Q_2 are available at time 0. However, Q_3 does not arrive at the system until time 8. Figure 1 shows that the SRPT schedule will idle the processor unnecessarily. At time 0, the task with the least processing time is Q_1 to process task d_1. However, when this task is completed, there are no tasks that require d_1, and hence the processor becomes idle for 2 time units while datum d_2 is fetched. In the optimal schedule, data item d_2 is fetched (and processed) first. For this example, the processor is only idle for 3 time units in the optimal schedule.

The problem of deciding which data should be present in the local "cache" is the principal topic of this paper. In particular, we tackle the applications where processing times are stochastic, with exponential distribution. The corresponding known optimal policy, the shortest *expected* remaining processing time (SERPT) rule, is used as our base execution policy. We conducted several experiments to measure the ability of a task scheduler to exploit data locality between multiple queries. In conjunction with our base execution policy, we propose and evaluate five different data fetch policies: (i) We modify the base SERPT algorithm to do efficient prefetching and call it PSERPT (Prefetch-SERPT). That is, a datum will be fetched from the remote server whenever it is needed by the task with the shortest processing time, irregardless of what other (if any) tasks also need to process this data. (ii) If more tasks require a datum compared to any other datum, then we fetch it first - we call this heuristic MaxPop (Maximum Popularity). (iii) If the expected usage time of a certain datum is highest among all data items, then we fetch it first - this is called MaxEx (Maximum Expected time). (iv) FIFO - Fetch the datum whose task has the minimum release time and (v) SLRU - Data is fetched



Query	Response Time	
	SRPT	OPT
Q_1	8	10
Q_2	15	9
Q_3	2	3
Avg. Resp. Time	8.33	7.33

Figure 1: SRPT is not optimal when disk locality must be considered.

according to PSERPT, but LRU (least recently used) is used to determine which currently resident data will be evicted when the new data arrives.

For multiprocessor applications, we analyze 3 global heuristics — a global SERPT (G-SERPT) ordering of queries based on their remaining processing times, a purely local SERPT ordering based on the remaining time of the parallel subparts of different queries, and a global FCFS (first come first served) ordering. Based on the above heuristics, we performed extensive simulations over a large range of grain sizes and locality. Our experimental results show that variations of SERPT, implemented at both the global and local levels, consistently produce excellent response times.

1.2 Related Work

The concept of splitting a job into parallel subcomponents has been proposed and analyzed in the literature, in the contexts of minimizing either the flow time or the makespan. In [10], it is proved that the problem of minimizing the schedule length (makespan) of the parallel subtasks using preemptive scheduling is strongly NP-hard for arbitrary number of processors for a set of independent tasks. In [7], Chang *et. al* consider the problem of minimizing the flow time of a set of jobs on a set of parallel processors, given a job dependent number of tasks. They assume a static list of jobs available at time 0 and do not allow task migration. They analytically prove the conditions under which certain tasks of a job should be processed sequentially instead of in parallel to get optimal results. Empirical results are provided in [15] for the problem of assigning virtual deadlines to parallel subcomponents of a task, in order to meet the global task deadline. For the case of non-preemptive scheduling of a set of parallel independent tasks, Turek *et. al.* [28] provide a shelf packing heuristic and analytically prove that its worst case response time is within a factor of 32 of the optimal. They assume that the number of processors required for each task and corresponding execution times are known. It is not clear how we can adapt the shelf packing algorithm for preemptive dynamic scheduling of tasks. In [1], Abbot and Garcia-Molina consider the problem of scheduling real time transactions, which involve minimizing the makespan. Their scheduling model deals with the problems of concurrency control and I/O scheduling while scheduling the transactions.

In general, considerable work has been done on the problems of preemptive scheduling of jobs, both in the deterministic case and stochastic case, see [16, 3]. One of the earliest works in minimizing response time is papered in [4], where the case of minimizing the weighted flow time for a static list of tasks, on $m \geq 1$ processors is considered. Resource constrained scheduling has long been a subject of active research, see [11, 31]. An excellent survey of complexity results related to various generic scheduling problems appears in [19] and [12].

Most of the research reported above has concentrated on finding good job schedules in cases where data is not shared among the jobs. We consider applications where there can be varying degrees of inter-query locality. The work reported in this paper highlights the importance of an integrated prefetching and caching strategy, especially when the grain size is small and is an extension of the work reported in [22, 23]. Equivalent problems relevant to multimedia systems have been analyzed [8] and [9]. Specifically, the problem of when to fetch a 'hot' video and which video on disk to replace when there is no free space, is conceptually similar to trying to exploit inter-query locality. In [13], a heat-based policy that replaces segments of relatively unpopular videos with segments of currently referenced videos is studied. Theoretical and simulation results of applying integrated prefetching and caching heuristics on several file access traces has been reported in [5]. However, they consider the uniprocessor case and do not use workloads that share data.

1.3 Outline of Paper

This paper is organized as follows: in Section 2, we describe the abstract model of the distributed database system. Section 3 breaks up the problem into one of global and local scheduling and describes various heuristics at both levels. The details of the experimentation methodology are discussed in Section 4. Simulation results are presented in Section 5. Finally, Section 6 provides a summary and conclusion of this work.

2 Model of the System

2.1 Model of the System

We assume n number of queries, $\{ Q_1, Q_2, \ldots, Q_n \}$, are to be processed by a system of m processors, P = $\{p_1, p_2, \ldots, p_m\}$. Queries arrive dynamically at the system, resulting in release times: $\{r_1, r_2, \ldots, r_n\}$. That is, the system has no knowledge of query Q_i until time r_i, and cannot process the query or fetch the data for that query before that time. The database consists of M distinct data items denoted as $D = \{d_1, d_2, \ldots, d_M\}$. We assume $M \gg m$. A data item may be a traditional database record, an HTML document, a frame of video data *etc.* To simplify the analysis, we assume all data items are the same size.

In the following, let i be the index of a query (Q_i), j be the index of an arbitrary processor (p_j) and k be the index of a data item (d_k). We define a *subtask* $\tau_{i,k}$ as the ordered pair (Q_i, d_k). A *task* $T_{i,j}$ is considered a component of the aggregate query Q_i and is defined as a tuple of subtasks, $\{\tau_{i,k} \mid k \leq |D|\}$, all of which have been mapped onto the same processor p_j. Thus, a task $T_{2,5}$ is the component of query Q_2 running on processor p_5 and consists of a set of subtasks $\{\tau_{2,k} \mid k \leq |D|\}$, all of which run on the same processor. In the rest of this paper, the subscripts will be omitted whenever the meaning is obvious from the context.

A query is said to be completed only when all its tasks are completed and a task is complete only when all its subtasks have finished processing the relevant data items. We denote as $x(\tau_{i,k})$ the *expected processing time* of the subtask, as $X(T_{i,j})$ the expected processing time of a task and $X(Q_i)$ denotes the expected execution time of a query. In our analysis, we define it as follows:

$$X(Q_i) = max_{\forall j}(X(T_{i,j}))$$

Note that in our model, a query is not infinitely partitionable, the number of subtask partitions being bounded by the number of data items that a query needs to process.

Researchers in the past have considered a two-level hierarchy of jobs — consisting of a global task and its parallel subtasks, see Section 1.2. For the problem considered in this paper, as shown above, we propose a *three level* hierarchy. The main benefit of having a subtask associated with each data item is that such an arrangement enhances the degree of multiprogramming. Assume this was not the case *i.e.,* a query Q_i had only one subtask τ_i running on a node and processing data items $d_1 \ldots d_k$. Assume that the number of data items that can be resident in the local disk at any time is l, where $k \gg l$. Since there is no way of knowing what data item is currently being processed by τ_i, the scheduler will assume that all the disk resident data items are locked during its execution. This greatly hampers the ability of the scheduler to decide what datum to fetch next, or what datum to evict from the disk when a new one arrives.

The main benefits of grouping related parallel subtasks of a query into a task are *ease of analysis and efficient implementation.* As will be seen in Section 3, the scheduling problem consists of both global and local scheduling. The local scheduler must *try* to execute all subtasks of a single query before another [1], the ordering being defined by some scheduling criteria. If the global scheduler were to monitor the completion status of each and every subtask running on all the nodes, the overhead would be enormous. By communicating only with the tasks of a query, it can delegate the job of subtask monitoring to their respective tasks.

2.2 Parallel Scheduling

The classical multiprocessor scheduling problem, where the objective is to minimize the makespan, has been the subject of intense analysis for several years. This problem can be reduced to variations of the bin-packing problem which has been studied in great detail in the past [14, 12].

2.2.1 Bin Packing

From an engineering point of view, the problem is normally one of making efficient use of time and/or space. A basic mathematical model is defined in the classical one-dimensional bin packing problem: We are given a positive integer bin capacity C and a set of items $L = (p1, p2, \ldots, p_n)$, each item p_i having an integer size $s(p_i)$ satisfying $0 \leq s(p_i) \leq C$. What is the smallest integer m such that there is a partition $L = B_1 \cup B_2 \cup \ldots \cup B_m$ satisfying $\sum_{p_i \in B_j} s(p_i) \leq C, 1 \leq j \leq m$? Usually, B_i is thought of as being the contents of a *bin* of capacity C and the objective is to minimize the number of bins needed for a packing of L. This problem is known to be NP-hard in the strong sense (it contains 3-PARTITION as a special case), so there is little hope of finding even a pseudo-polynomial time algorithm for it. However, a number of simple approximation algorithms have been proven to provide asymptotic worst case bounds that are within a constant factor of the optimal solution. A few of these are listed in Table 1 [14]. In this table, $R_A = A(L)/OPT(L)$ *i.e.,* the ratio of the performance of the algorithm A to the performance of the optimal algorithm, where performance is measured in terms of how many bins (or fraction of bins) were required for the packing. R_A^∞ is the asymptotic value of R_A. As can be seen, the performance of all the heuristics are within a factor of 2 of the optimal.

Algorithm(A)	Timing	R_A^∞
NEXT FIT	$\theta(n)$	2.0
BEST FIT	$\theta(nlogn)$	1.7
GROUP FIT	$\theta(n)$	1.5
BF DECREASING	$\theta(nlogn)$	1.222...

Table 1: Asymptotic worst case bounds for certain bin packing algorithms

In the context of bin packing, the multiprocessor scheduling problem can be stated as follows: given a collection of identical processors on which a set of independent tasks with known execution times are to be executed, determine the least number of processors that must be used in order that all tasks be completed by some given deadline. Here, the processors are bins, the deadline is the common bin capacity and the elements of L are the task execution times.

The above description of the problem assumes that the objective is the minimization of the makespan of the tasks. However, it is also known that the corresponding problem where the objective is the minimization

[1] the actual order of execution will depend on data availability.

of average response time is also NP-hard in the strong sense [28]. In the next subsection, we present a brief analysis of the multiprocessor scheduling problem as studied in this paper.

2.2.2 Multiprocessor Scheduling of Queries

Since queries arrive at different times, it is obvious that a dynamic scheduling policy will be needed. Among dynamic policies, if a preemptive policy is used, the exact nature of preemption must be determined. In this section, for the sake of analysis, we assume that all data is locally available and data access time is zero.

Case (i)

$$\forall i, j, k, l : \ x(\tau_{i,j}) = x(\tau_{k,l})$$

i.e., any given query takes the same time to process all data items and all queries process a data item in equal times. Thus, the total processing time of all queries is the same. It is easy to see that in this case, there is no need for preemption. Even if the release times are different, a task need not preempt an already running task on a node because the new task has the same processing time as the old one.

Case (ii)

$$\forall i, j, k : (X(Q_i) \neq X(Q_j)) \wedge (x(\tau_{i,j}) = x(\tau_{i,k}))$$

i.e., total processing times of two queries are different, but for any given query, all its subtasks require the same processing time. The best distribution of subtasks would be such that equal number of them are placed on each processor. Thus, all tasks of any query always have the same number of subtasks and therefore have the same length. In this case, the best scheduling policy would be that a new query Q_i preempts an already running query Q_j if $X(Q_i) <$ remaining processing time of $X(Q_j)$. Thus, on *all* processors, a task of the new query will preempt an already running task if the above condition is satisfied *i.e.,* the partial ordering of tasks on *each* processor follows the total order of the queries at a global level and all tasks of a query still finish up in a lockstep fashion.

Note: The cases (i) and (ii) are trivially solvable in polynomial time using SRPT scheduling.

Case (iii)

$$\forall i, j, k : (X(Q_i) \neq X(Q_j)) \wedge (x(\tau_{i,j}) \neq x(\tau_{i,k}))$$

i.e., total processing times of two queries are different and for any given query, the subtask processing times are also different. To analyze the complexity aspects of scheduling under this condition, consider the very first query Q_1 that comes into the system. It is represented by a set of its subtasks as $Q_1 = \{\tau_{1,1}, \tau_{1,2}, \ldots, \tau_{1,M}\}$. Consider the base case with only 2 processors. We would like to schedule the subtasks on the 2 processors in such a way that the total completion time on both processors is the same *i.e.,* none of them is idle while the other one is busy. However, this quickly turns out be an intractable problem, as can be seen from the following. Associated with each subtask is the processing time $x(\tau_{i,j})$. Our aim is to find *two* groups of subtasks, denoted as tasks T_1' and $(Q_1 - T_1')$ such that the following condition holds:

$$\sum_{\tau_{i,j} \in T_1'} x(\tau_{i,j}) = \sum_{\tau_{i,j} \in (Q_1 - T_1')} x(\tau_{i,j})$$

This problem is directly reducible to the PARTITION problem and is thus NP-Complete [12].

2.2.3 Queuing Network Models

Researchers in the past have formulated queuing network models of various abstract scheduling policies and analyzed them, see [24]. For many applications, queuing network models achieve a favorable balance between accuracy and efficiency and thus are considered an extremely useful analytical tool [17]. An example system model from [24] is shown in Figure 2. The model consists of two service centers. The first is a *multiple server queuing center*, *i.e.,* the center can service up to a fixed number of tasks simultaneously, and represents a multiprocessor composed of M identical processors. The queue of this center can be referred to as the ready queue because it contains those tasks that are ready to execute but are waiting for an available processor. The second center is an *infinite capacity server*, *i.e.,* there is no queuing because all tasks are serviced simultaneously, and represents the time spent by tasks while nonschedulable. For this model, the authors consider a single customer class, *i.e.,* all tasks are assumed to be identical (in our problem, however, tasks are non-identical). The scheduling policy defines the manner in which tasks are removed from the ready queue. The computing demands of tasks per visit to the multiprocessor are assumed to be independent and identically distributed as exponential random variables with mean D. Tasks become non-schedulable (due to I/O, synchronization, quantum expiration, or preemption) for random periods of time which are assumed to be independent and exponentially distributed with mean Z. Under these assumptions, the authors are able to obtain analytically tractable results for various scheduling policies.

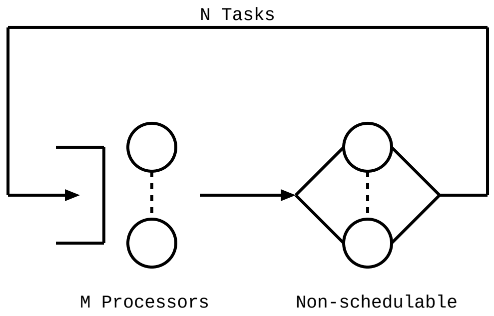


Figure 2: A queuing network model

The model shown in Figure 2 has been presented mainly to show a typical application of queuing network models in analyzing a multiprocessor scheduling application. A detailed analysis of this model can be found in [24].

In the remainder of this subsection, we analyze the usefulness of queuing models to the problem discussed in this paper.

Each node in our system can be modeled as an $M/G/1$ server [2] under certain specific conditions. In particular, we can treat each processor as an $M/G/1$ system with the difference that arriving tasks are divided into n different *priority classes* (corresponding to the ordering of the tasks). Class 1 has the highest priority, class 2 has the second highest and so on. We use the following notations: for each service class k
λ_k = arrival rate of tasks

[2] where M implies *M*emoryless arrival process (a Poisson process), G implies that service times have a *G*eneral distribution and 1 means there is only one server.

$\overline{X_k}$ = $1/\mu_k$ = mean service time
$\overline{X_k^2}$ = second moment of service time
Ω_k = average delay per task
$\rho_k = \lambda_k/\mu_k$ = system utilization for priority k (Intuitively, this gives the proportion of time the system is busy).
R_k = Mean residual service time. This means that if task j is already being served when k arrives, R_k is the remaining time until task j's service time is complete. If no task is in service (*i.e.,* the system is empty when k arrives), then R_k is zero.

We are interested in knowing what is the average delay for a task for each class. Note that it is possible to affect the average delay by choosing the priority classes appropriately. It is true that average delay tends to be reduced when customers with short service times are given higher priority. For a preemptive resume priority scheduling, it is shown in [2] that

$$\Omega_k = \frac{(1/\mu_k)(1 - \rho_1 - \ldots - \rho_k) + R_k}{(1 - \rho_1 - \ldots - \rho_{k-1})(1 - \rho_1 - \ldots - \rho_k)}$$

(the first term in the denominator is discarded for $k = 1$ because ρ_0 is not defined). In the above, the mean residual time is given by

$$R_k = \frac{\sum_{i=1}^{k} \lambda_i \overline{X_i^2}}{2}$$

Unfortunately, it is unclear if we can easily extend these results to the case of m servers unless the service times of all priority classes are identically and exponentially distributed [2], having the *same* mean $1/\mu_k$. In our model, each query can have its own independent mean (and thus each priority class will have a different mean). For this reason, the applicability of this approach to our problem is highly limited.

2.3 Scheduling for Data Locality

To characterize the disk cache of a processing node, we define a *slot* as follows: $s_{i,j}$ = a storage unit j for storing exactly *one* data item on processor p_i. Let the set S_i = $s_{i,1}, s_{i,2}, \ldots, s_{i,l}$ denote the disk cache of a processor p_i. We introduce a caching constraint, which can be defined as follows:

Caching Constraint

$$|D| \gg |S_1| + |S_2| + \ldots + |S_m|$$

recall m = $|P|$.

Assume that the space of all data items are statically partitioned among the various processors. Each processor may cache only data in its partition. For simplicity we number the data items as follows: p_1 may cache data items $d_1, d_2, \ldots, d_{M/m}$, processor p_2 may access data items $d_{M/m+1}, \ldots, d_{2M/m}$ and so on (assume that M/m is an integer). This is analogous to saying that a subtask can execute only on one particular node irrespective of whether the data item required by it was present on the node at the time of the subtask-to-processor mapping. Note that we assume homogeneous processing nodes, such that $|S_1| = |S_2| = \ldots = |S_m|$, and processing speeds are the same.

Our goal is to minimize the flow time, the sum of the expected completion times of all the queries. The focus of our research is to achieve this goal while exploiting *data locality* as much as possible. Since all queries are decomposed into parallel subtasks, a *global*, multiprocessor scheduling algorithm would be needed to achieve the above objective. As shown in Section 2.2.2, this problem is NP-complete, even when we assume that all data is locally accessible. Given the *caching constraint* introduced in this section, we have developed a set of heuristics to determine what is the best possible flow time we can obtain. The evaluation of these heuristics is done through simulation.

2.3.1 Cache Affinity Scheduling

It is worthwhile to look at some research on cache-affinity scheduling that has been done in the past, since it has a strong correlation with the locality constrained scheduling problem we are trying to solve. A brief recap of cache affinity scheduling may be in order. In a shared memory multiprocessor, when several processes time share the same cache, one process would displace the state previously built up by a second one. Consequently, when the second process runs again, it generates a stream of misses as it rebuilds its cache state. Also, since an idle processor simply selects the highest priority runnable process, a given process often moves from one CPU to another. This frequent migration results in a stream of cache misses as the process tries to build up its cache state. To avoid the cost of these misses, it is worthwhile to consider a cache affinity based scheduling of the processes. In [29], the scheduling policy used is a *space-sharing* policy that partitions processors among applications. It was found that benefits due to affinity were minimal due to the relative infrequency of process preemption (about once every 400ms). However, this was primarily due to the space sharing policy and the non-blocking (no I/O) nature of their workloads. In [26], a wider mix of workloads is evaluated and the more commonly used time sharing policy is used. It is found that affinity scheduling does provide moderate gains for most of the workloads and does not hurt the performance or response time of the rest. These conclusions are also supported by the findings of [24] and [6]. In [24], queuing network models of different abstract scheduling policies are formulated. These policies span the range from ignoring affinity to fixing tasks on processors. Mean Value Analysis and simulations are used to compare the effectiveness of these policies.

Page Migration Policies have been studied in the context of non-cache-coherent NUMA machines such as the BBN Butterfly and cache-coherent NUMA, such as the Stanford DASH [6]. In the former, these policies are based on the page fault mechanism, and migrate a page to the local memory of the processor that references it. However, pages that are actively shared between several processors may be repeatedly migrated from one processor's memory to another. To avoid this ping-ponging effect, pages are usually frozen in memory after a certain number of migrations, and defrosted after a timeout period so that they are again eligible for migration. Variations of this basic strategy have been studied in [30] and [21]. In contrast, for cache-coherent NUMA, [6] proposes that TLB miss counts be used as an approximation to cache miss counts for making page migration decisions.

2.4 Limitations of Existing Scheduling Techniques

The heuristics for solving bin-packing assume that release times are zero and that all data is instantly available. In our model, queries arrive at different times (release times are non-zero) and remote data accesses have a finite non-zero delay. Thus, results from bin packing are not directly applicable.

It is possible to model our abstract system as an $M/G/m$ server, but only if all the task processing times are identically and exponentially distributed with the same mean. For the applications that we are trying to solve, the task processing times do not necessarily have the above property.

Existing research on cache-affinity scheduling assumes that a task can execute only on one processor at any given time. Such an execution policy greatly reduces the flexibility of the scheduler to interleave tasks that share some data. For reasons given in Section 2.1, we split a query into parallel subtasks, whose execution could be going on simultaneously on different processors. Thus, results from cache-affinity scheduling cannot be directly adapted for our model.

3 The Scheduling Heuristics

3.1 Introduction

In this subsection, we propose heuristics for tackling the scheduling problem for the model of computation described in Section 2.1 and given the caching constraint. We treat scheduling as a two-level problem. A global scheduler determines priorities for each query based on either the overall query processing times or the query release times. Thus, each subtask of a query gets assigned a certain priority. A local scheduler, running at each node, attempts to run the subtasks according to their priorities, while still ensuring that the processor does not become idle because the necessary data for a particular subtask is not available locally.

3.2 Additional Notation

To outline our algorithms, we will use the following notations in addition to those given in Section 2.1 : for any processor p_k :
S_k = set of all disk slots
R_k = set of all fetch requests
M_k = array of *all* data items assigned to p_k
ω_i = set of waiting subtasks corresponding to data item d_i.
$W(x)$ = weight assigned to any arbitrary variable x.

3.3 Global Scheduling

The global scheduler must decide the order in which to process the queries in the system. We evaluated three global policies for deciding the priority ordering of the queries. All of them do the ordering *only* at the time of arrival of a new query. No dynamic load balancing is done. Note that by assuming that all data is statically partitioned among all processors, we have implicitly achieved a fair degree of load balancing. The only situation where such a partitioning would produce poor results is when the task processing times of queries have such a high variance that it may be cheaper to ship the data to another idle node for processing. Given the large number of queries and the large number of subtasks per query, this variance gets evened out.

The global scheduling policies are:

G-SERPT In this case, the queries are ordered according to a SERPT policy. We take the remaining time of the longest task of a query as the remaining time of that query. For a new query, the remaining time amounts to the expected processing time of its longest task.

L-SERPT In this policy, all queries are assigned equal priority. The local nodes are allowed to process the tasks in a greedy Local-SERPT manner.

FCFS The queries are assigned priorities according to their release times *i.e.,* they are ordered on a First Come First Serve basis.

In both G-SERPT and FCFS, the priority of a subtask on a node is exactly the same as the priority of its parent query. However, as explained in the next subsection, the node scheduler is free to choose the next *runnable* task for execution, even if it means skipping one before it in the priority order.

3.4 Local Scheduling

To simplify the discussion of various scheduling policies, we organize the policies around three independent decisions. These decisions are:

1. At each instant in time, what subtask should be allowed to run? — We call this decision the **Execution Policy**. Note that only those subtasks whose data is currently present in a slot may be run.
2. If there is more than one subtask that needs data from the disk, which datum should be fetched next? — We call this decision the **Fetch Policy**
3. When a new datum arrives from the disk, which of the currently resident data items should be removed in order to make room for the new data? — We call this decision the **Eviction Policy**.

3.4.1 Execution Policy

Only those subtasks whose data are resident in the disk are runnable. Other subtasks must wait at least until the data is fetched over the network. The global scheduling policy has assigned a priority to the the tasks at the local node. However, note that it does not mandate that a node process the *subtasks* in a given order. We choose to order all the subtasks in the *runnable queue* of a task, using SERPT. Our estimate of processing time is based on probability distribution functions that we assume are known *a priori*. The distribution function provides a "best-case" estimate which is assumed to be an accurate assessment of the processing required *if the necessary data is already resident in disk.* We assume that the scheduler knows the expected value for the processing time of each subtask, but has no other information about the execution time.

Figure 3, illustrates the simulation methodology. The figure shows the procedure used to simulate each processor in the system. Recall that the system consists of $m = \mid P \mid$ homogeneous processors. Each processor runs the runnable subtask that is closest to the head of the subtask queue. A running subtask can be preempted in two situations: (i) When a new query arrives and the global policy mandates that the new query's task receive higher priority. (ii) When it ceases to be runnable because its data has been evicted.

3.4.2 Data Fetch Policy

Assigning Weights to Data Items

When deciding how to reorder the data request queue or, at the time of a new data arrival, what datum to overwrite (if at all), we assign *weights* to all the data items. For various policies, the assignment is as follows:

- **MaxPop** In this case, the weight of the data item is the *number* of subtasks waiting for it

$$W(d_i) = |\omega_i|$$

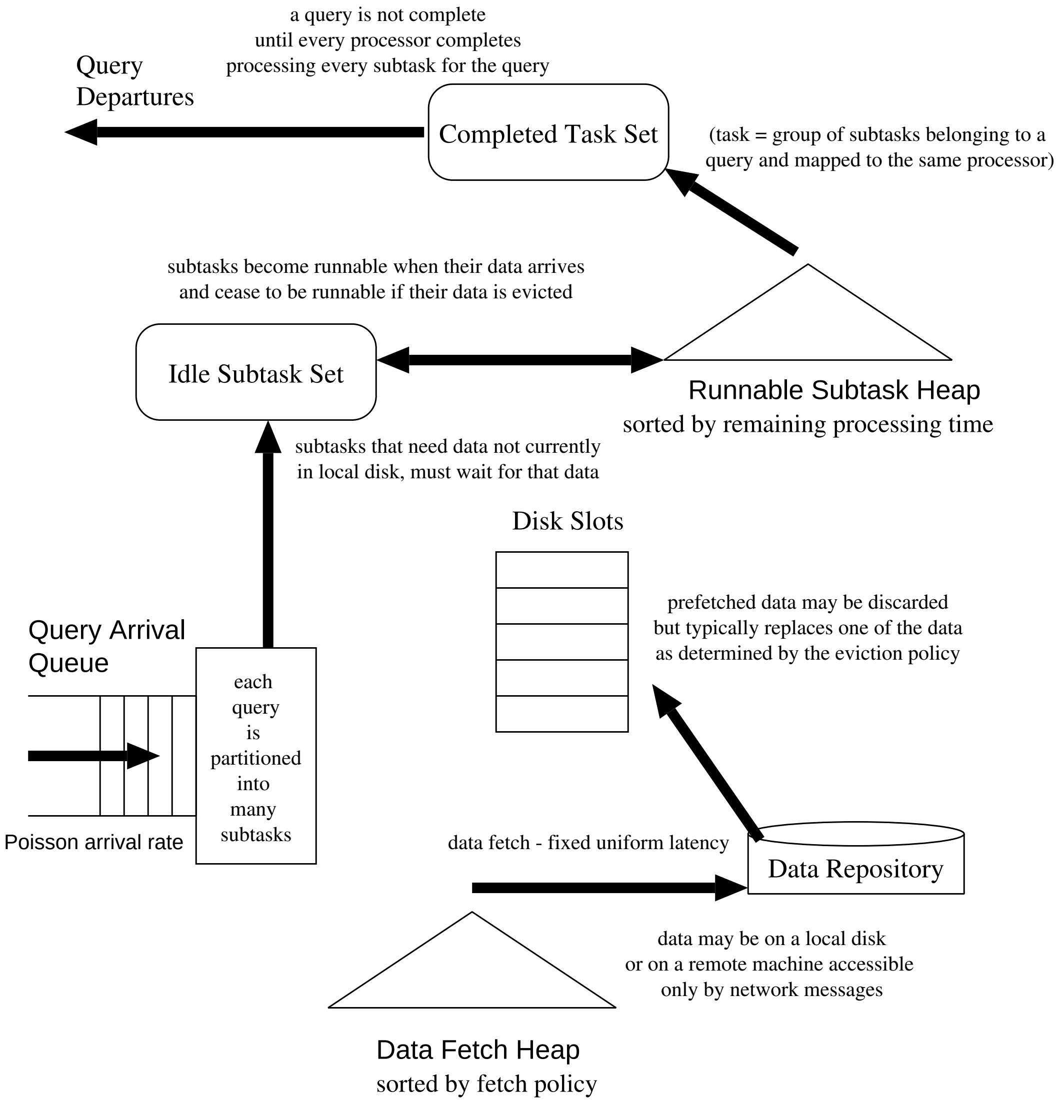


Figure 3: Simulating a single node in the multiprocessor database.

- **MaxEx** The weight of the data item is the *sum* of *expected processing times* of all the subtasks waiting for it
$$W(d_i) = \sum_{\tau \in \omega_i} x(\tau)$$

- **PSERPT** For each subtask waiting for the data item i, compute its *position* from the *head* of the overall subtask queue at the node.
$$W(d_i) = \min_{\forall \tau \in \omega_i} position(\tau)$$

- **SLRU**

  If d_i is a new arrival, $W(d_i)$ = the arrival time
  else $W(d_i)$ = last access time

- For **FIFO**, weight is the inverse of age.

The data fetch policy determines which item will next be fetched from the storage server. We assume that data are fetched one at a time and we assume that the data access time is fixed and uniform for all data. These simplifying assumptions are not completely unrealistic. Parallel or pipelined access to the server will be *less* effective than simply having a single, sequential server that is n-times faster. Hence, by simulating the fetch policy on a faster server we can approximate the effects of parallel I/O. Uniform access times to the data is somewhat unrealistic when data sizes are small. When data sizes are small, message packing and unpacking times dominate the transfer time. These times are quite sensitive to message sizes and thus would affect the access time. However, if we assume that the unit of a data fetch (a "data item") is large, then transfer time will be dominated by the I/O bandwidth. If communication to the server is over relatively reliable channels (*e.g.,* a local area network), then access times will be fairly close to uniform.

Every time a new task arrives at the node, we recompute the weights of the data requests in the request queue. After doing this, we apply the algorithm appropriate to each policy:

- **MaxPop** and **MaxEx** Sort the request queue in *descending* order of data weights.

- **PSERPT** and **SLRU** Sort the request queue in *ascending* order of data weights.

In addition, whenever an in-core data item gets evicted, we add a request for that datum to the request queue, preserving the order of the respective fetch policy. Our fetch policy is aggressive, possibly wasting bandwidth in the aim of ensuring that data is available when it is needed. The actual fetch of a data item is initiated whenever the channel at node k is free and $size(R_k) > 0$. There is no guarantee that the newly fetched data item will be cached into the disk — this depends on the eviction policy, which is the subject of the next subsection. Since the execution time is not known with assurance, our decision to prefetch cannot be perfect. We must either risk wasting bandwidth or risk starving the processor by not prefetching the data. We choose the former evil for this analysis. We note that such aggressive techniques are being contemplated for certain current day applications. In order to meet the response time requirements of the clients connected to the world wide web, [27] proposes a source based anticipatory *presending* of data, as opposed to prefetching. The pre-sent data is marked 'droppable available-bit-rate' traffic (*i.e.,* a low priority packet) and thus avoids consuming valuable bandwidth during peak transmissions of currently useful data.

3.4.3 Eviction Policy

The eviction policy determines which datum will be replaced when a new datum, d_k is fetched from the disk. In order to allow for aggressive speculative prefetching, we permit the scheduler to discard the newly fetched data item if it fails the requirements of the eviction policy. In most instances, such a decision would be a needless waste of disk bandwidth. However, it is possible that during the time that the disk access was in progress new queries arrived. In some pathological cases, the *optimal* response to these new queries may be to discard the just-fetched data and continue to process tasks using the old data. Further, we do *not* permit the scheduler to overwrite a datum which is currently being used. The reason for this is best explained by considering the example of MaxPop. It is quite likely that the currently locked data item has a popularity of 1 (the last subtask is processing it) and assume that the subtask is very close to completion. If a newly arrived data item, by virtue of its higher popularity, evicts the locked datum, the sheer overhead involved in reloading the state of the subtask is really not acceptable, considering that it was close to completion [3].

Intuitively, the eviction policy must complement the fetch policy. For example, if we are fetching the most popular data items, we should be evicting the least popular. We thus define the eviction policies by the name of their respective fetch policies (except for SLRU). In the following description, $W_{ps}(d_i)$ is used to mean that we calculate the weight of the data item based on the PSERPT policy, as given in Section 3.4.2.

MaxPop and **M**axEx

choose $((d_{min}) \in S_k \mid W(d_{min}) = min_{\forall d_i \in S_k}(W(d_i))$
if $(W(d_{new}) > W(d_{min}))$, then
  overwrite d_{min} with d_{new}
else if $(d_{min} == d_{new})$, then
  if $(W_{ps}(d_{new}) < W_{ps}(d_{min}))$, then
    overwrite d_{min} with d_{new}
  endif
endif
if d_{new} is not written, insert request for d_{new} in R_k

PSERPT

choose $((d_{max}) \in S_k \mid W(d_{max}) = max_{\forall d_i \in S_k}(W(d_i))$
if $(W(d_{new}) < W(d_{max}))$, then
  overwrite d_{max} with d_{new}
else insert request for d_{new} in R_k
endif

For **FIFO** fetches, the eviction policy is the same as for MaxPop. For **SLRU** the LRU eviction policy is used in conjunction with the PSERPT fetch policy.

3.4.4 Additional Fetch/Evict Policies

It must be mentioned that in addition to the fetch/evict policies already described in the previous subsections, our simulator can implement the following policies:

- FIFO fetch combined with any of the eviction policies

[3] However, it is possible that it may still get preempted if some new subtask becomes runnable and the execution policy decides to run this other subtask.

- Minimum Expected time (converse of MaxEx) fetch combined with MaxEx evict
- Minimum Popularity (converse of MaxPop) fetch combined with MaxPop evict.

Early simulation results showed that these policies performed particularly badly. Hence, the results for these policies are not reported in this paper.

4 Simulation Methodology

4.1 Introduction

We identify the following three important tunable parameters for the simulation:

Task granularity The ratio of the average time to process a runnable subtask to the time required to fetch a datum from disk.

Locality The percentage of the total data set accessed by a single query.

Load The average number of tasks being processed by a node at any time.

The general experimental methodology is as shown in Figure 3.

4.2 Grain Size

The granularity is an estimate of the ratio of computation to communication performed during query processing. If queries are very coarse grained, then the data fetch time is insignificant. Hence, locality will be of little concern in selecting an optimal schedule. We vary the granularity over three orders of magnitude, ranging from data access latency 100 times that of the processing time to data access latency of 1/20th of the processing time. In our simulation, we choose to hold the processing time constant, and vary the data fetch latency, when measuring the effect of granularity.

4.3 Locality

Locality is the probability that a data item will be used by more than one query. We do not model locality directly. However, by varying the amount of data accessed by each query, we can control the locality indirectly. In each simulation we select a data access volume for each query. This value is the percentage of the entire database that the query will process. The data items that are accessed are uniformly distributed in the database. If the data access volume is very large, then the locality of data references will naturally be high, since most queries will be processing most of the database. By contrast, if the data access volume is very small, then the probability that any two queries will access the same datum is very small.

4.4 System Load

We are interested in the steady state behavior of the system under heavy load. If the load is very light, then virtually any reasonable scheduling algorithm will result in good performance. In our simulations, we control the system load by controlling the interarrival time of the queries. However, an interarrival time that results in "heavy" load with a very large grain size, may result in a relatively light load for fine grained queries. To provide a consistent basis for comparison of the scheduling algorithms, the interarrival time was tuned for each combination of grain size and data locality.

We use the well known Little's result [2] from queuing theory : $N = \lambda * T$, where N is the average number of queries that can be handled by the system, signifying the average load, λ is the rate of arrival of queries and T is the average delay of a query in the system. Our simulations process a fixed number of queries, 250, (since we are interested in the response times, we must run all queries to completion). We fix N to be a fraction of the total number of queries and then find λ, which gives us an approximate estimate of the required interarrival time. Then, we run the experiment iteratively (using the interarrival time from the previous iteration) until the average load per node converges to a 'reasonable' value. In our simulation, we choose this value to be an average load of 1.5 - 3% of the total number of tasks (since the total number of queries considered is 250, this amounts to roughly 5 active tasks on every processor at steady state). Note that interarrival time is determined by grain size and locality. All scheduling algorithms are compared using the same interarrival times and processing times.

Note that a more precise definition of 'load' would have to incorporate the processing times of the subtasks. In that case, we would have load $\kappa = \lambda \frac{\textit{mean number of subtasks}}{\textit{mean subtask processing time}}$. However, to calculate the mean subtask processing time, we must include the time for which a processor has to wait for its data to become available. Unfortunately, it is impossible to determine this time period. This is because even though the data fetch time is fixed, it is not known at what time the fetch of a data item will be initiated. Further, it is not known *a priori* if the newly fetched data item will be written into the disk. Thus, we simplify the definition of load to be the average number of active tasks on a processor at steady state.

Scheduling characteristics are controlled by parameters listed in Table 2.

Parameter	Simulation Range	Comments
number of queries	fixed at 250	
number of CPU's	fixed at 2	
data set size (data items)	fixed at 100	
size of disk cache (slots)	fixed at 5	
mean subtask processing time	0.01 to 5.0	Uniform distribution
data fetch time	0.05 to 100	
locality (%)	10 to 100	
query interarrival time	chosen dynamically	Poisson distribution

Table 2: Simulator Parameters

4.5 Processing Times

We assume that a single query will require similar processing of each data item. Different queries, however, may require substantially different processing time. Accordingly, when a query arrives at the system, the simulator selects a mean subtask processing time using a uniformly distributed (pseudo)random number (over the range (0.1,5.0)). This value is used directly by the global scheduling algorithm to select the task priority for this query. For each subtask, an expected processing time is computed by perturbing the mean processing time by up to 20% using another uniformly distributed random number. It is necessary to perturb the mean because in general, each subtask of a query will have a different mean processing time. The value just computed is used by the local scheduler when computing SERPT priorities. Finally, the *actual* processing time is calculated by generating an exponentially distributed random number about the mean just computed for that subtask.

Thus, each subtask has an unknown (to the scheduler) processing time, with a known expected value. All of the subtasks for a single query have similar expected processing times (within 20% of a common value). The expected value for query processing time is known to the global scheduler. And query processing time varies over about an order of magnitude.

4.6 Data Structures of the Scheduler

In this subsection, we present an outline of the main data structures used by the scheduler. A schematic is shown in Figure 4.

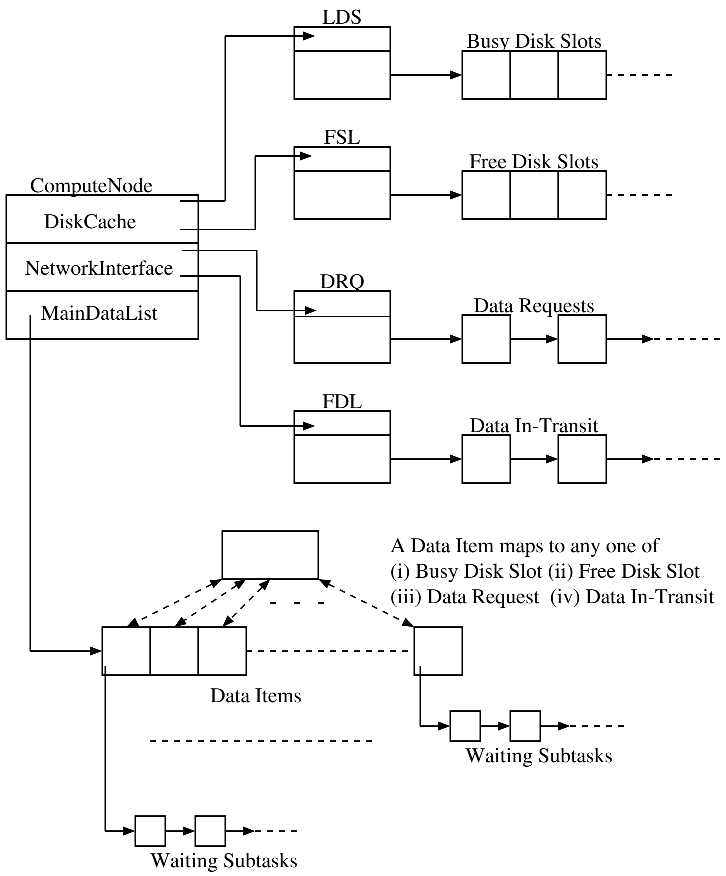


Figure 4: Outline of the main data structures

- *The Main Data List (MDL)*. This list serves as a placeholder for all the data items that are supposed to be serviced by that node. When a new subtask arrives, it first checks the MDL. An array operator of the form $MDL[data\_item\_id]$ allows the subtask to quickly find out of which list is the data item currently a member. Further scheduling decisions may be based on this information. The size of the MDL is fixed by the variable $ITEMS\_PER\_CPU$, which is the ratio of the total number of data items to be processed to the number of cpu's in the system (recall that we assume static partitioning of the entire data set).

- *The Local Data Set (LDS).* This is a map of the data items currently residing on the disk but not a part of the Free Slot List (see below). A subtask can execute *only* if its corresponding data item is a member of LDS. The size of the LDS is bounded by the variable $DISK\_CACHE\_SIZE$. As given by the locality constraint, we are interested in the case where $ITEMS\_PER\_CPU \gg DISK\_CACHE\_SIZE$. The local data set is maintained by the Disk Cache of the compute node.

- *The Free Slot List (FSL).* Initially, all disk slots are 'free' and thus part of the FSL. As data items get loaded, these slots are consumed and removed from the FSL. Conversely, when a subtask finishes execution, it 'releases ' its lock on the slot that was holding the corresponding data item. If the 'waiting subtasks' queue (this is explained later) of this data item is empty, the slot is eligible to be moved to the FSL from the LDS. In other words, it has become a 'free' slot and can be overwritten by a newly arriving data item. If the waiting subtasks queue is not empty, then the decision to move the slot from one list to another will depend on the overall scheduling and prefetch heuristic being used. The justification of keeping this list is the following: when a new data item arrives (possibly as a result of a prefetch), it need only look into the FSL to find a data item that *might* be eligible for overwriting. This way, the cost of search is much lower, being bounded by the smaller size of the FSL. However, if the FSL is empty, a more expensive search operation of the LDS may be unavoidable. The free slot list is also maintained by the Disk Cache.

- *The Data Request Queue (DRQ).* This is a priority queue of data item requests submitted by various subtasks. The requests in the priority queue are ordered according the relevant fetch policy. This queue is maintained by the Network Interface (NI) of the compute node. If a new subtask finds that a request for its data item is already in the request queue, it puts itself on the waiting subtasks queue of the data item.

- *The Fetch Data List (FDL).* This list is also maintained by the NI. When the NI issues a data fetch request to load a data item from the central server to the local disk, it creates an entry in FDL corresponding to this data item. All data items in FDL are stamped with the status of *IN_TRANSIT*.

The Waiting Subtasks Queue (WSQ). An element in this queue is called a 'WaitJob' and consists of two entries: a *task id* and a *pointer to a subtask.* The data item that a subtask is waiting for can be either in transit (in FDL), already requested for (in DRQ), currently cached in the disk and not a free slot (in LDS), or a free slot (in FSL). Thus, for each element in these four lists, we keep a pointer to a WSQ. The justification for keeping a WSQ is that it allows a data item to quickly find out which subtasks will process it.

An Implementation Note: All the above data structures, except the MDL have been implemented using *associative* container classes provided by the Standard Template Library [25], and therefore allow for insert and search operations of logarithmic complexity. The MDL is implemented as a vector.

5 Experimental Results

5.1 Introduction

In this subsection, we analyze the performance of the global and local heuristics, with respect to response time, bandwidth utilization and throughput. In most of the experiments, it was found that as a local scheduling heuristic, PSERPT outperformed other local heuristics. Thus, while comparing the various global heuristics, it is best to analyze PSERPT's comparative performance under different global policies. In the next subsection, we present this comparison. Section 5.3 covers the comparison among local heuristics.

5.2 Comparing Global Heuristics

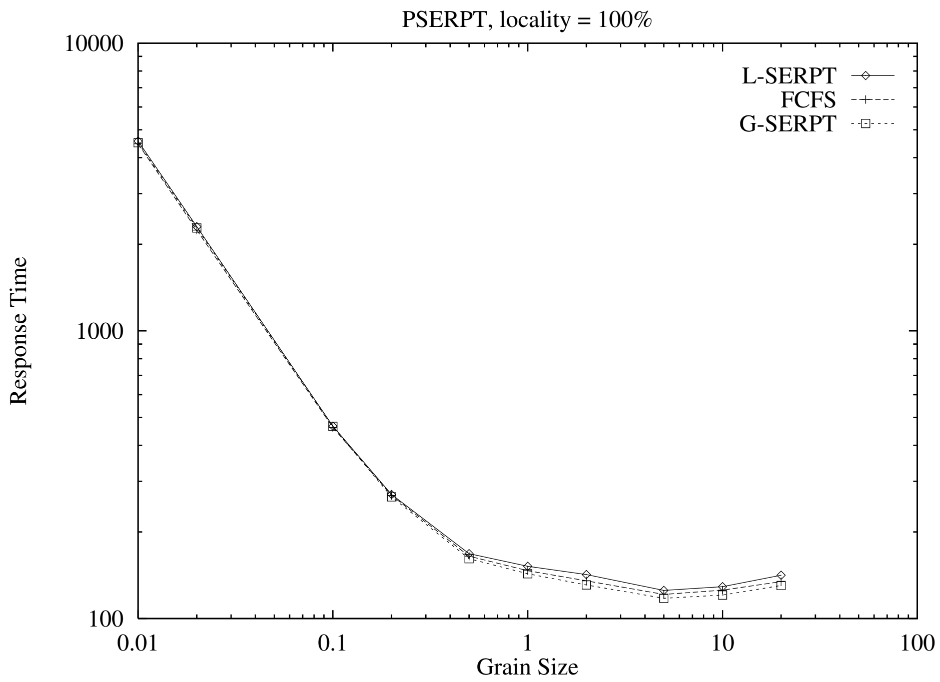


Figure 5: Global Heuristics for locality = 100

Figure 5, and Figure 6 show the performance of PSERPT under the three global heuristics: FCFS, L-SERPT and G-SERPT, for locality values of 100 and 75% respectively. For any heuristic, the response time is going to be higher for very small grain sizes (higher fetch time contributes to a higher response time). With an increase in grain size, a data item can be fetched in much lower time and the response time falls rapidly. At grain sizes closer to 1 and higher, the response time is dominated by the actual subtask processing times since any data item can be fetched in negligible time. This conclusion is validated by the relative flattening of the response time curve for grain sizes closer to 1 and higher.

From these figures, it is seen that at the global level, for small grain sizes, all three policies perform almost equally well. This behavior can be explained by recognizing that for small grain sizes, the processing time is constrained by the data fetch time. Thus the fetch policy, which is a local policy, will essentially de-

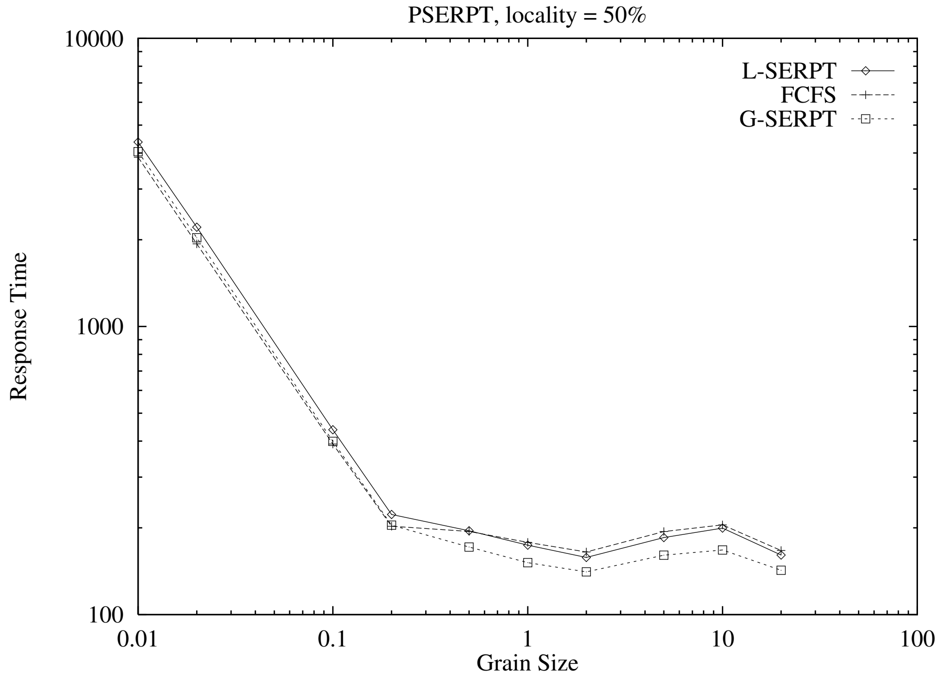


Figure 6: Global Heuristics for locality = 75

termine the response time. Since the local fetch policy for all the heuristics in this experiment is the same (*i.e.,* PSERPT), all global heuristics perform equally well for fine grain queries. As the grain size increases, L-SERPT performs poorly. This result indicates that some global coordination is valuable even in homogeneous environments.

The simple G-SERPT priority policy provides a noticeable improvement to response time compared to FCFS and L-SERPT. For a locality value of 100%, the improvement is in the range of 4 - 17%, while for a locality value of 75%, the improvement is in the range of 8 - 12%. This trend holds over a large range of locality values, as shown in Figure 7 for a grain size of 0.5. For this reason, in the following subsection, unless otherwise mentioned, we will focus on the local scheduling heuristics implemented under G-SERPT.

5.3 Comparing Local Heuristics

In this subsection, we analyze the performance of various local heuristics (PSERPT, MaxPop, SLRU, MaxEx and FIFO) under the global policy G-SERPT. It must be noted that since SLRU has the same fetch policy as PSERPT, one would expect that any difference between them would be caused only by the eviction policy. By definition, SLRU *never* evicts newly fetched data. It is seen that PSERPT is also highly successful in fetching the correct data item and thus rarely has to evict the newly fetched datum. For this reason, it is not a surprise that the performance of SLRU closely tracks that of PSERPT in all of the experiments described in this subsection.

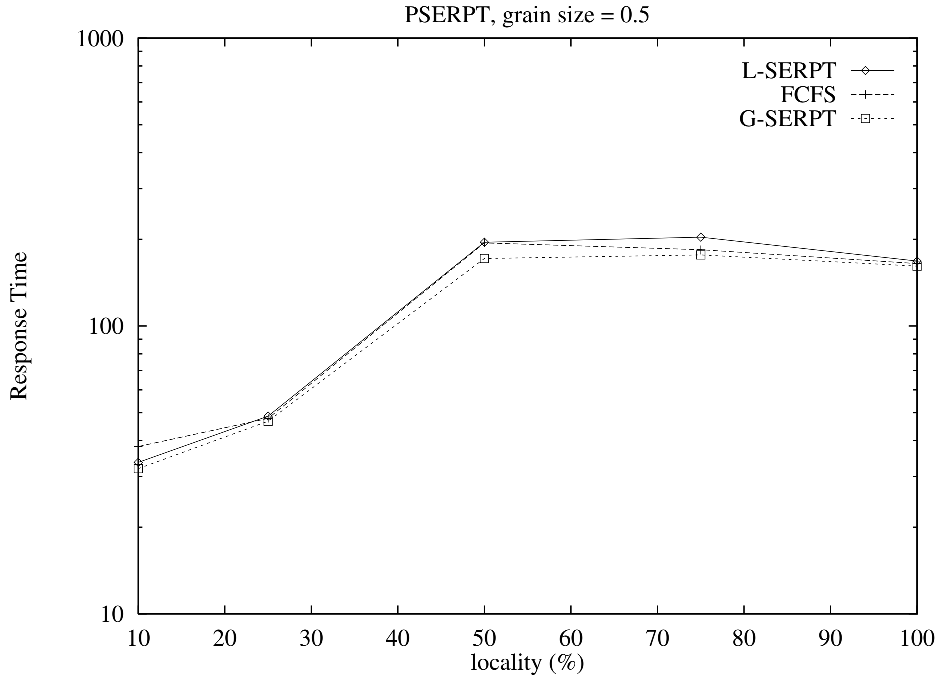


Figure 7: Global Heuristics for Grain Size = 0.5

5.3.1 Response Time versus Locality

It is expected that the average response time of queries would increase for lower values of locality (*e.g.,* locality $< 50\%$), irrespective of the heuristic used. This is because when locality is low, the chances are higher (compared to the probability at higher locality) that a task's in-core dataset does not intersect (or minimally intersects) with another task's dataset. This would force some delay, since a shorter task may not be able to run unless it loads its own fresh set of data. At higher values of locality, a task can be more compute bound, amortizing some of the communication delay, thus reducing the gradient of the response time line.

We conducted several experiments to determine how locality affects the performance of the data fetching and eviction heuristics. We measured the response times, keeping grain size fixed at 0.25. The results are depicted in Figure 8. PSERPT is found to perform consistently better than most other heuristics, with Max-Pop catching up only at very high locality. The gradient of the response time curve is found to follow the expected behavior – an initially high gradient, followed by a range where response time is mostly constant.

5.3.2 Response Time versus Grain Size

By reasoning along the same lines as in the previous subsection, we can conclude that response time is likely to decrease substantially for lower grain sizes and remain relatively constant thereafter. This is because, in effect, there are only two ways to prevent a task from starving : (i) increase the locality (an extreme case: use any data that is available) — this was studied in the previous subsection, or (ii) increase the grain size (negligible fetch time would provide a glut of data and prevent starvation). It is expected that for a given locality, for higher grain sizes, all tasks become more and more compute bound and the query response time

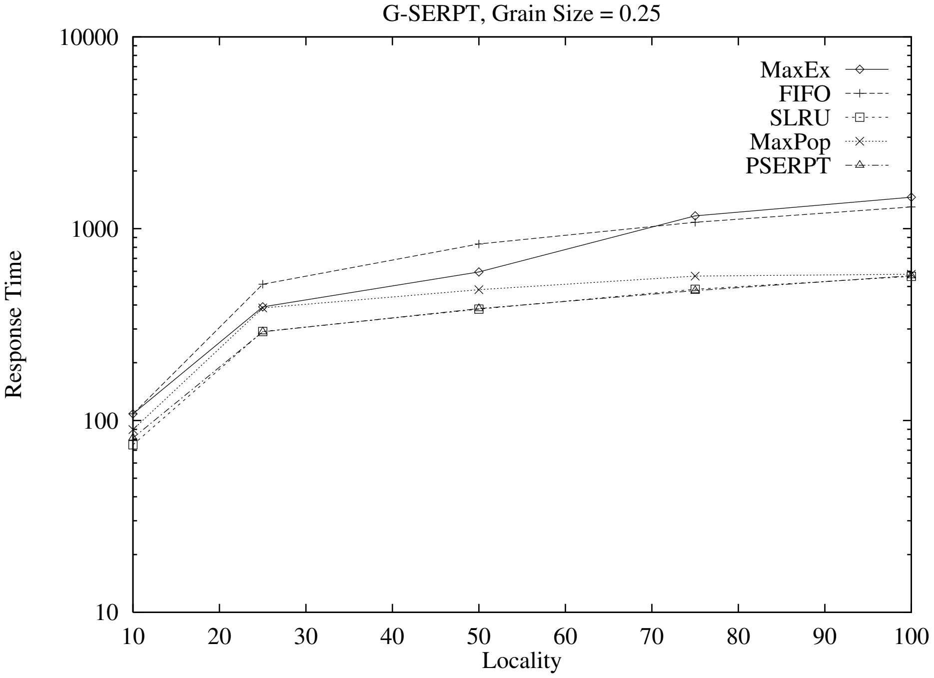


Figure 8: Local Heuristics at Grain Size = 0.25

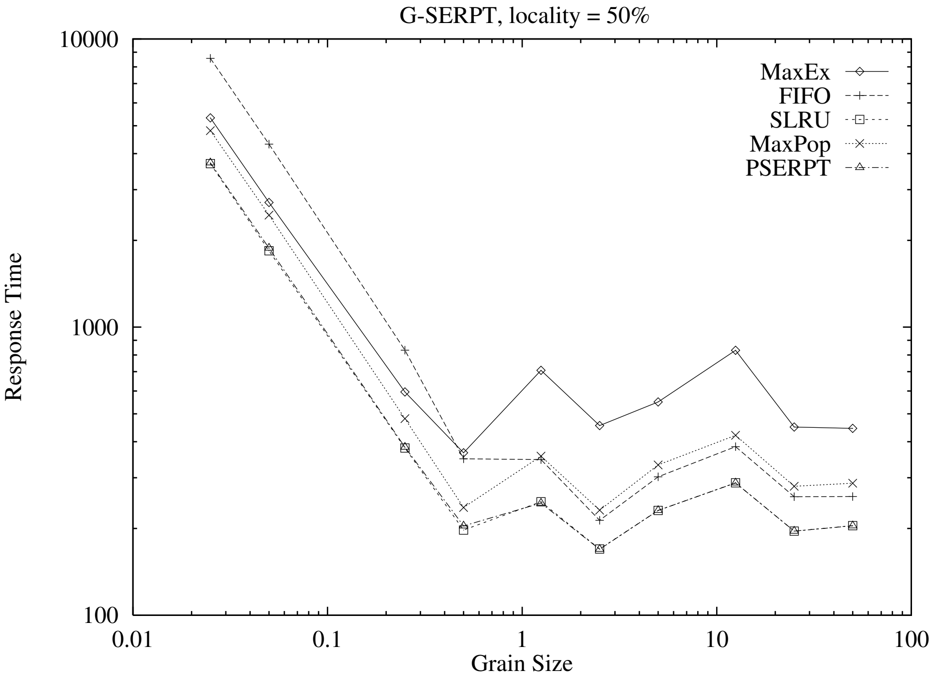


Figure 9: Local Heuristics at Locality = 50

would hardly be affected by communication delays.

We designed an experiment to study case (ii) mentioned above. In this experiment, we consider the effect of grain size on the heuristics, keeping the locality fixed. We measured the response times for processing times varying from 2 orders of magnitude above to about an order of magnitude below the fetch time of a data item. This was done keeping the locality fixed at 50% and 100%. The results of this experiment are shown in Figure 9 and Figure 10. As was expected, the response time decreases rapidly for smaller grain sizes and remains relatively flat for grain size ≥ 1.

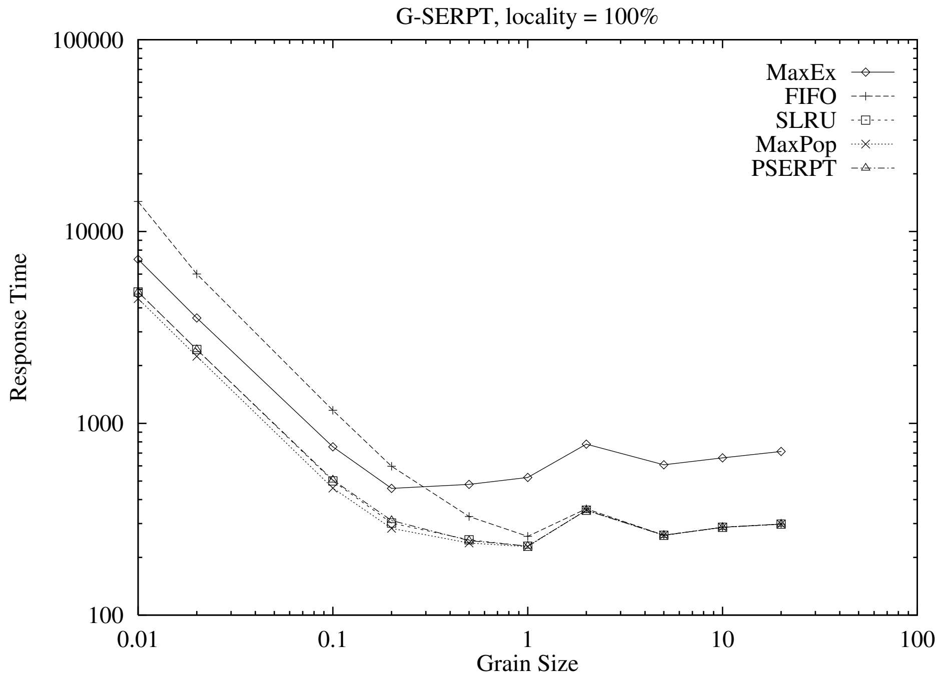


Figure 10: Local Heuristics at Locality = 100

A FIFO fetch policy is inherently weak, in that it does not exploit the difference in the subtask processing times while deciding the fetch order. Thus, its response time is not likely to be very good. MaxEx was designed mainly to reduce the consumption of network bandwidth. By forcing the data items having high processing times to stay resident in the disk much longer, MaxEx is likely to hinder the execution of shorter tasks. Thus, one can expect that it would adversely affect the average response time of the queries. As has already been explained at the beginning of Section 5.3, the performance of SLRU and PSERPT are going to be very similar in terms of response time over the entire range of grain size. These conclusions are validated by Figures 9 and 10.

Now, consider the MaxPop heuristic. At smaller locality, it is more likely that the 'most popular' data item may not have the shortest task in its list of waiting tasks. Further, the chances are higher that the 'least popular' data item does include the shortest task in its list. A pathological case could be when the shortest task has only one remaining subtask but its data item is repeatedly evicted because none of the other tasks need it, preferring instead to fetch some other popular datum. From Figures 9 and 10, it can be seen that MaxPop does indeed perform poorly compared to PSERPT for higher grain sizes at locality = 50%, whereas it is as good as, if not better than PSERPT at locality = 100%.

The important conclusions from this experiment are the following: (i) MaxEx and FIFO are inadequate over a large range of grain size. (ii) MaxPop's window of 'best performance' is limited to very high locality and low grain sizes. (iii) In terms of consistently good performance, if not the best, over a wide range of both grain sizes and locality, PSERPT is the clear winner.

5.3.3 Effective Utilization of Bandwidth

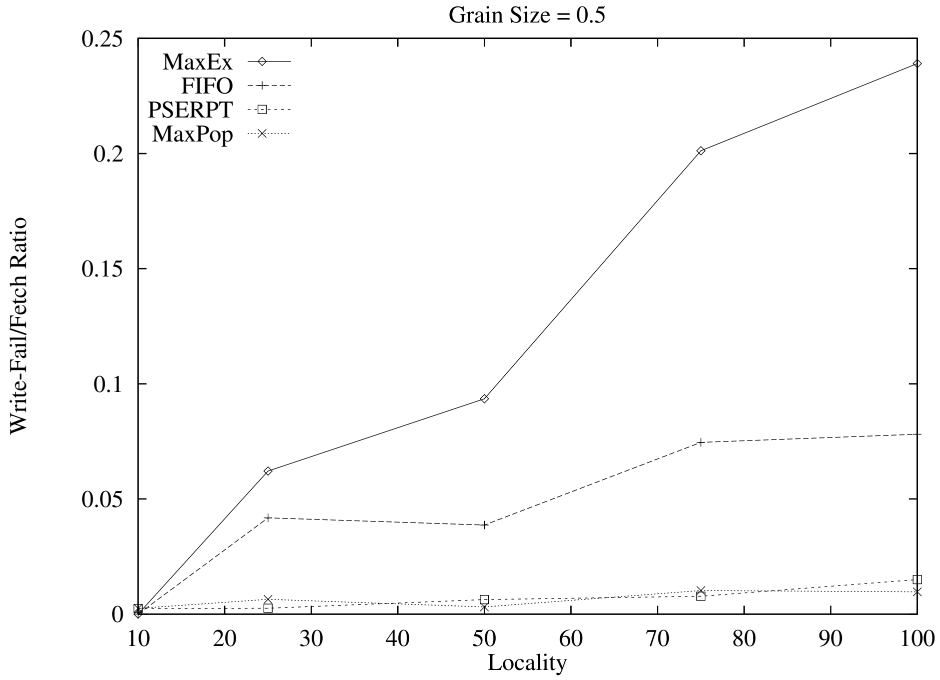


Figure 11: Effectiveness of Fetch Decisions

Our experiments were designed to determine how response time can be minimized at the possible expense of wasted bandwidth. We now analyze how much bandwidth is actually wasted by each heuristic. In this experiment, we recorded the average number of data fetches per node, and the number of times a newly fetched data item was evicted (number of write-fails). The ratio of write-fails to the fetches gives an indication of the wasted bandwidth. We studied the variation of this ratio over a large range of locality values. The results of this analysis are plotted in Figure 11. To understand why MaxPop's wasted bandwidth is very low, consider a case of 100% locality. After the scheduler decides to fetch the most popular data item, and before its arrival, it is possible that a new task arrives at the node. However, this new task would only *add* to the popularity of the fetched data item. Thus, it is likely that the new data item will be written into the disk rather than getting evicted (recall that we allow the eviction of a newly fetched data item). For lower values of locality, such an increase in popularity of the new data item is less likely, but by the same reasoning the probability of increase in popularity of an in-core data item is equally low (the in-core data set being only a small fraction of the total data set). Based on this reasoning, one can conclude that MaxPop would not waste a lot of bandwidth. It must be noted here that one would expect there to be several situations where the popularity of a to-be-fetched datum or a newly arriving datum is exactly the same as for some other data items. This was confirmed by the experimental logs. Recall that in MaxPop, ties are broken by considering the SERPT order of the subtasks associated with the corresponding data items.

The MaxEx heuristic had been expressly designed to minimize the consumption of network bandwidth (fetch data that will be useful for the longest possible period of time). Yet MaxEx appears to perform poorly in Figure 11. The reason why MaxEx performs so poorly can be understood from the following: Assume there are n disk slots and N data items ($N \gg n$). Once the n slots get filled up, a newly arriving data item is *likely* to find all the slots occupied by data items with *higher* total expected processing times than itself. This is because our execution policy is SERPT, which means that even if a subtask finishes execution, there is *only a small decrement* in the total expected resident time of the in-core data item. If however, our objective was to minimize the makespan, we would choose the longest expected remaining processing time (LERPT) execution policy [19], and one would expect MaxEx to give satisfactory results.

5.3.4 Throughput and Idle Time

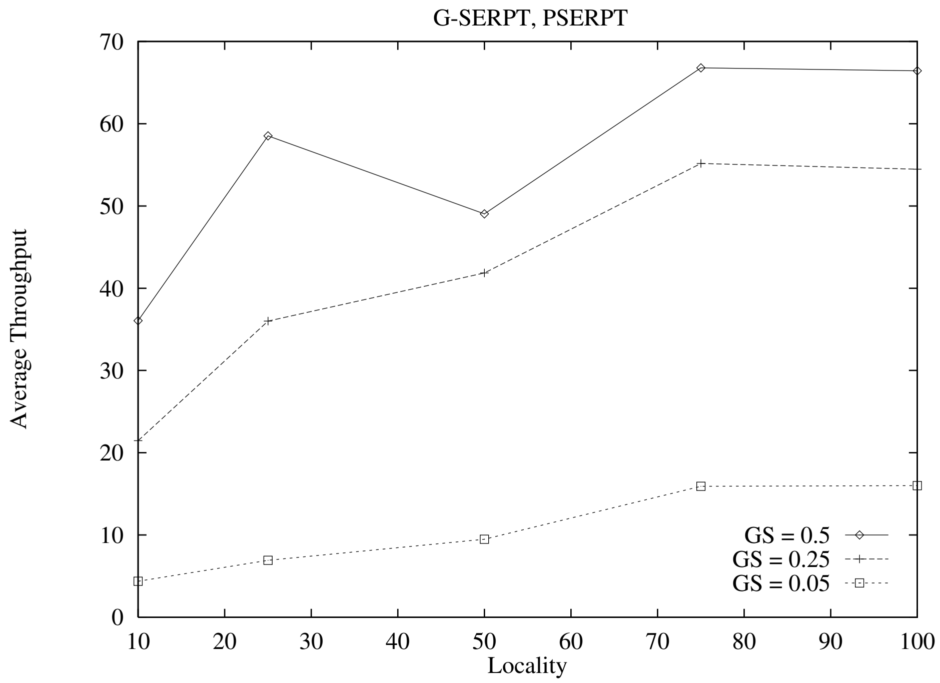


Figure 12: Throughput at Grain Size < 1

In this experiment, we study the variations of both throughput and idle time, since they are closely related to one another. In fact, they have approximately an inverse relationship to one another.

In order to know how many operations are taking place per unit time as we increase the locality or the grain size, we define throughput Γ of the system as follows:

$$\Gamma = (MLn)/\sigma \tag{1}$$

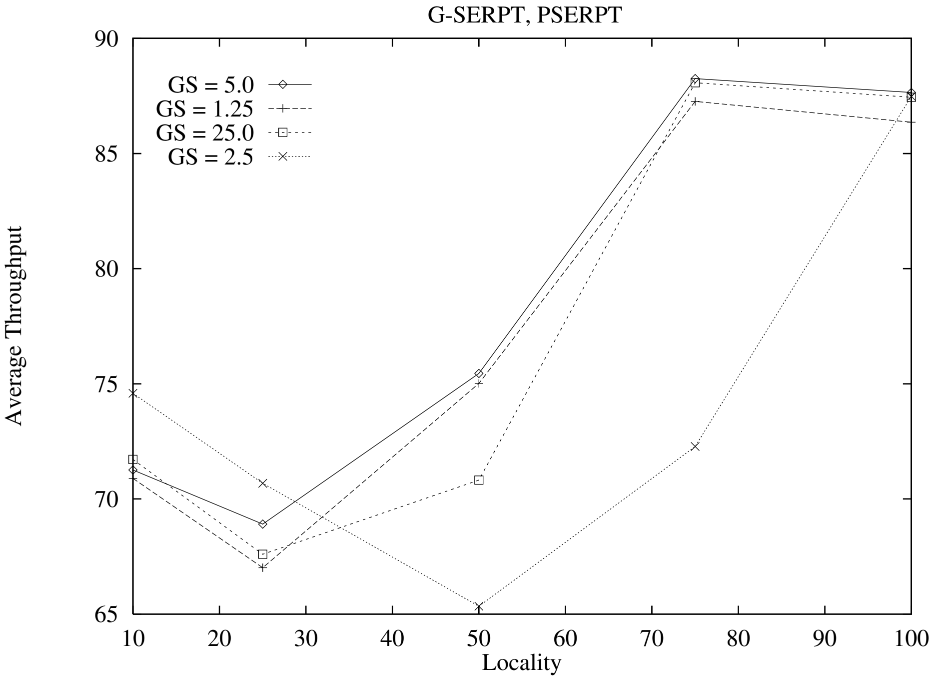


Figure 13: Throughput at Grain Size > 1

where
M = total number of data items
L = locality factor
n = total number of queries to be processed
σ = total running time = (completion time of the last query - arrival time of the first query).

The percent average idle time ρ of the system is defined as follows:

$$\rho = (t_{idle}/\sigma) * 100 \tag{2}$$

where
t_{idle} = average idle time per node

Note that an alternative definition of throughput could simply be $\Gamma = n/\sigma$ *i.e.,* the number of queries per unit time. However, with an increase in locality, σ is certainly going to increase, and it is predictable that the throughput will go down according to this relation. For this reason, we choose the definition given in Equation 1.

All the heuristics that we have proposed try to maximize the throughput by running an executable subtask the moment a previous one finishes. Thus, it can be expected that there will not be substantial difference among the heuristics in both the percent idle time and the throughput. The experimental data confirmed this reasoning. Thus, we do not provide the comparison among various heuristics with respect to these two metrics. However, for the best performing local policy, PSERPT, it is of interest to know how the throughput and percent idle time vary with respect to locality. This is because in Equation 1, both the numerator and denom-

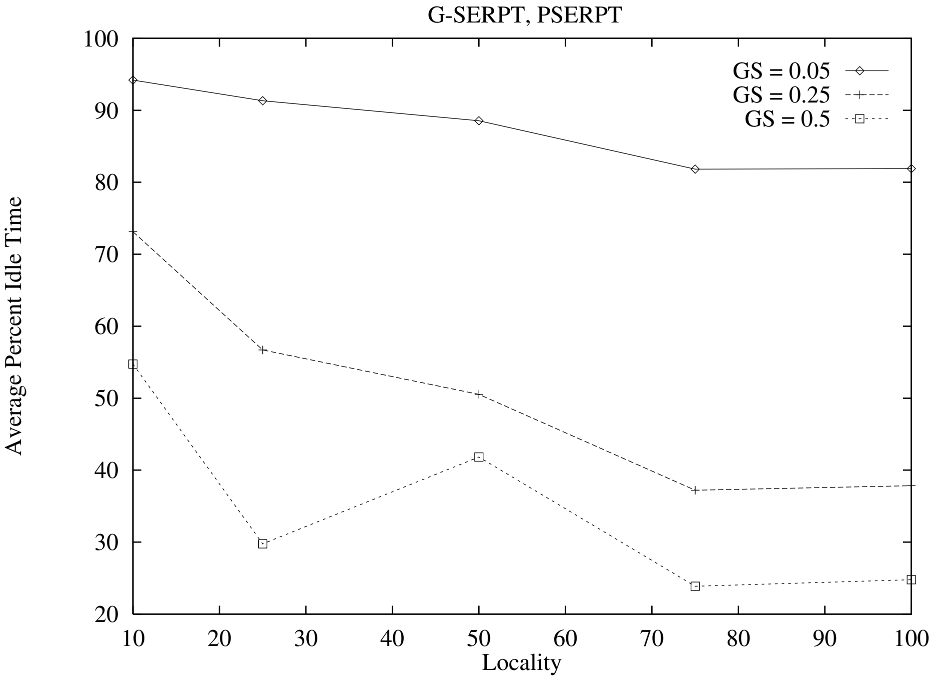


Figure 14: Average Percent Idle Time at Grain Size < 1

inator will increase with increase in locality and the value of Γ will depend on which one increases faster. Also, it is unclear at first whether t_{idle} will increase or decrease with increase in locality. If a task needs to process more data items, it may have to fetch those data items, which entails additional waiting (increased t_{idle} in Equation 2). At the same time, an increase in locality implies that the probability of a task finding its desired data item in the disk cache is increased. This means the processor should be idle for lower time (reduced t_{idle}). Thus, to determine the exact behavior of the system, it is useful to investigate the idleness of the system as locality is increased (in Equation 2 σ is certainly going to increase with increase in locality).

The variations in the average throughput for grain sizes less than and greater than 1 are shown in Figure 12 and Figure 13 respectively. From Figure 12, it is clear that for lower grain sizes, as far as throughput is concerned, the benefit of higher locality far outweighs the effect of higher running time caused by the extra processing required. This can be seen from the fact that in general, there is an increase in the average throughput over the locality range. For grain sizes greater than 1, we note that the throughput drops initially and then increases for the middle range of locality. Consider the example where grain size = 2.5. For locality values close to 50 and below, the average throughput is found to be at its lowest. To explain this, we must answer the following two questions: (1) Why does throughput decrease for values below 50 ? and (2) Why does it increase substantially for values above 50 ? Both these questions can be answered by first realizing that for $L < 50$, there is less chance that the data set of two queries intersect. In the worst case, each new query may have to build up its local cache state from scratch (the 'cold cache' effect). This implies more fetches and therefore a higher value of σ in Equation 1. For $L > 50$, many subtasks experience cache hits. Thus, the fetch time can be overlapped with processing time, decreasing the average value of σ.

For other values of grain sizes that are higher than 1, this 'lowest throughput point' is reached at lower

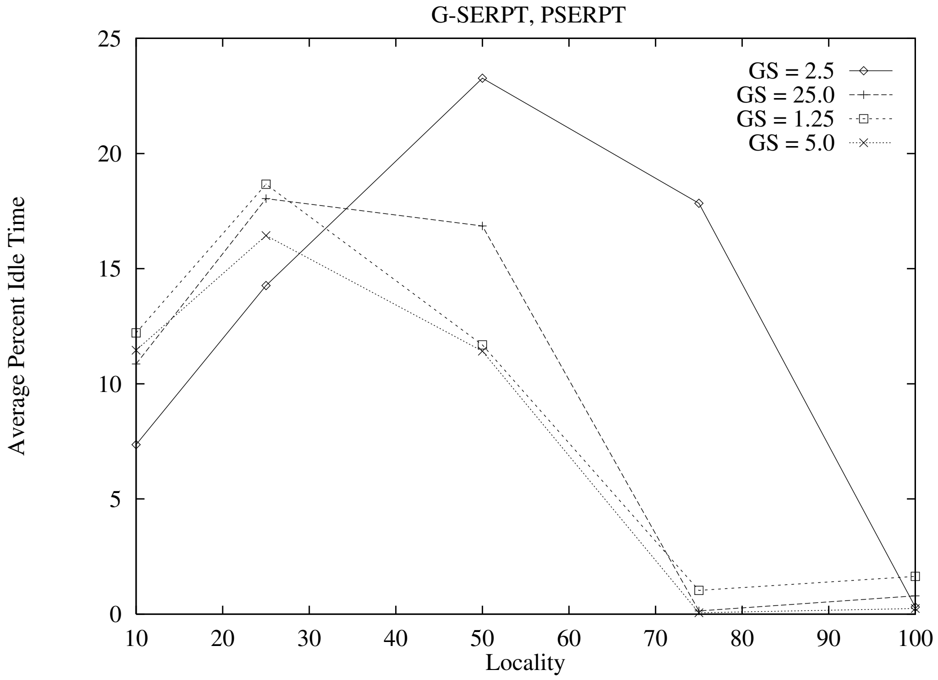


Figure 15: Average Percent Idle Time at Grain Size > 1

values of locality. We can explain this behavior for grain sizes 5.0 and 25.0 by reasoning that the fast fetches compensate for the cold cache at a lower value of locality as compared to, say, when grain size is 2.5.

Higher locality is found to have a beneficial impact on the percent idle time, as seen in Figure 14. As was expected, the average fraction of time the system is idle is found to be inversely proportional to the throughput – this conclusion is confirmed by comparing Figure 14 and Figure 15 with their throughput counterparts, Figures 12 and 13 respectively.

6 Conclusion and Future Work

In this paper, we have studied alternative heuristics for preemptive scheduling of queries accessing a read only data base in a distributed environment. Queries are assumed to have no precedence constraints and their data sets are not mutually exclusive. The model proposed in Section 2 represents an important class of applications that require exhaustive processing of a large distributed data set by several users. Since queries have non zero release times, a dynamic preemptive policy is chosen. Our objective is minimization of the flow time of the queries. In the past, analytical and simulation studies have been done for problems of similar nature. However, the techniques proposed in those studies are not directly applicable to our problem. Analytical techniques related to bin-packing assume that all data is instantly available and tasks have the same release times (*e.g.,* at time zero). In our scheduling problem, queries have different release times and data fetches involve non-zero delay. It is possible to model the system as an $M/G/m$ server, but only if the task processing times are identically and exponentially distributed with the same mean. This limits the applicability of this approach to our problem. Existing simulation studies on cache-affinity based scheduling assume that a task can execute only on one processor at a given time.

We decompose a query into parallel subtasks in order to allow us the greatest flexibility in scheduling the computations such that inter query data locality can be exploited. As described in Section 3, we divide the scheduling problem into global and local scheduling problems and propose and evaluate several heuristics at both levels. Our simulation results establish that variations of the shortest expected remaining time (SERPT) algorithm, implemented at both local and global levels, are very successful in meeting the objective, for a large range of grain size and locality. Both PSERPT (SERPT augmented with prefetching) and MaxPop (a heuristic based on fetching the most 'popular' data item) perform well in terms of effective utilization of network bandwidth. For the best performing local heuristic, PSERPT, we studied the throughput and idle time behavior for grain sizes above and below 1 and for several values of locality.

The research reported in this paper has focussed on read-only databases. In the future, we plan to extend our techniques to distributed transaction processing systems and distributed multimedia systems, both of which require an integrated prefetching and caching scheme to ensure good response times.

References

[1] Robert Abbot and Hector Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceeding of the Fifteenth International Conference on Very Large Data Bases*, pages 385–395, 1989.

[2] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 1987.

[3] Jacek Blazewicz. Selected topics in scheduling theory. *Annals of Discrete Mathematics*, 31:1–59, 1987.

[4] J. Bruno, E. G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17(7), July 1974.

[5] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *SIGMETRICS'95*, 1995.

[6] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *ASPLOS-VI*, pages 12–24, 1994.

[7] Cheng-Shang Chang, Randolph Nelson, and David D. Yao. Optimal task scheduling on distributed parallel processors. *Performance Evaluation*, 20:207–221, 1994.

[8] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *ACM, Multimedia'94*, 1994.

[9] Asit Dan and Dinkar Sitaram. An online video placement policy based on bandwidth to space ratio. In *ACM, SIGMOD'95*, 1995.

[10] Jianzhong Du and Joseph Y-T Leung. Complexity of scheduling parallel task systems. *SIAM Journal of Discrete Math*, 2:473–487, November 1989.

[11] Michael R. Garey and David S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal of Computing*, 4(4):397–411, December 1975.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[13] S. Ghandeharizadeh and C. Shahabi. On multimedia repositories, personal computers, and hierarchical storage systems. In *ACM, Multimedia'94*, 1994.

[14] E.G.Coffman Jr., Michael R. Garey, and David S. Johnson. Approximation algorithms for bin-packing - an updated survey. In G.Ausiello, M.Lucertini, and P.Serafini, editors, *Algorithm Design for Computer System Design*, pages 49–106. Springer-Verlag, 1984.

[15] Ben Kao and Hector Garcia Molina. Subtask deadline assignment for complex distributed soft real-time tasks. In *14th International Conference on Distributed Computing Systems*, pages 172–181, 1994.

[16] E.L. Lawler, J.K. Lenstra, and A. H. G. Rinnooy Kan. Recent developments in deterministic and stochastic scheduling: A survey. In M. A. H. Dempster, J. K. Lenstra, and A. G. H. Rinnooy Kan, editors, *Deterministic and Stochastic Scheduling : Proceedings of the NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems*, volume 84, pages 35–73. D. Reidel Publishing Company, July 6-17 1982.

[17] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queuing Network Models*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[18] M. Ogg and A. Ricciardi. Architecture decisions for wide-area applications. In *14th PODC*, page 259. ACM, 1995.

[19] Michael Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[20] A. Ricciardi, M. Ogg, and E. Rothfus. The nile system architecture: Fault-tolerant wide-area access to computing and data resources. In *6th Computing in High Energy Physics*, 1995.

[21] R.P.Larowe and C.S.Ellis. Experimental comparison of memory management policies for numa multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, 1991.

[22] Aman Sinha and Craig Chase. Scheduling queries with large out-of-core data sets. Technical Report TR ECE-PDS-1995-018, Parallel and Distributed Systems Laboratory, University of Texas at Austin, October 1995.

[23] Aman Sinha and Craig Chase. Exploiting data locality for multiprocessor query scheduling. To appear in *International Conference on Parallel Processing*, August 1996.

[24] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[25] Alexander Stepanov and Meng Lee. *The Standard Template Library*. Hewlett-Packard Laboratories, Palo Alto, California, February 1995.

[26] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Benefits of cache affinity scheduling in shared memory multiprocessors : A summary. In *SIGMETRICS*, 1993.

[27] Joseph D. Touch. Defining high-speed protocols: Five challenges and an example that survives the challenges. *IEEE Journal on Selected Areas in Communications*, pages 828–835, June 1995.

[28] John Turek, Uwe Schwiegelshohn, Joel L. Wolf, and Philip S. Yu. Scheduling parallel tasks to minimize average response time. In *Proceedings of 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 112–121, 1994.

[29] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. *Operating Systems Review*, 24(5):26–40, 1991.

[30] W.J.Bolosky, M.L.Scott, R.P.Fitzgerald, R.J.Fowler, and A.L.Cox. Numa policies and their relation to memory architecture. In *ASPLOS-IV*, pages 212–221, 1991.

[31] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, pages 949–960, August 1987.