Approximate Parallel Simulation of Web Search Engines

Mauricio Marin[1,2] Veronica Gil-Costa[1,3] Carolina Bonacic[2] Roberto Solar[1]

[1]Yahoo! Labs Santiago, Chile
[2]DIINF, University of Santiago, Chile
[3]CONICET, University of San Luis, Argentina

ABSTRACT

Large scale Web search engines are complex and highly optimized systems devised to operate on dedicated clusters of processors. Any, even a small, gain in performance is beneficial to economical operation given the large amount of hardware resources deployed in the respective data centers. Performance is fully dependent on users behavior which is featured by unpredictable and drastic variations in trending topics and arrival rate intensity. In this context, discrete-event simulation is a powerful tool either to predict performance of new optimizations introduced in search engine components or to evaluate different scenarios under which alternative component configurations are able to process demanding workloads. These simulators must be fast, memory efficient and parallel to cope with the execution of millions of events in small running time on few processors. In this paper we propose achieving this objective at the expense of performing approximate parallel simulation.

Categories and Subject Descriptors

I.6 [**SIMULATION AND MODELING**]: General

General Terms

Algorithms, Design, Performance

Keywords

Discrete Event Simulation, Parallel Computation.

1. INTRODUCTION

Web search engines are usually built as a collection of services hosted in large clusters of processors wherein each service is deployed on a set of processors. Services are software components executing operations such as (a) calculation of the top-k documents that best match an user query; (b) routing queries to the appropriate services and blending of results coming from them; (c) construction of the result Web page for queries; (d) advertising related to query terms; (e) query suggestions, among many other operations. Services are assembled to meet specific application requirements. For instance, the design of a vertical search engine dedicated to retrieve advertisement pertinent to user queries is quite different from the design of a search engine dedicated to retrieve Web documents relevant to user queries. Consequently, the realization of each service may involve the use of different algorithms and data structures devised to support efficient query processing. Discrete-event simulation is a suitable tool for assisting the development cycles associated with the algorithm engineering required to achieve efficient and scalable search engines [10].

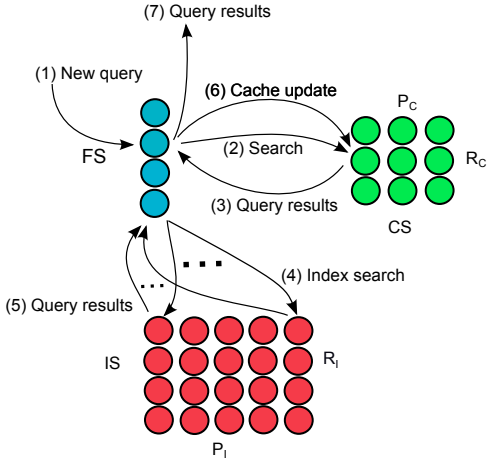


Figure 1: A typical query processing scheme

Figure 1 shows an example of an organization of services for a vertical search engine [1, 12]. The Front-end Service (FS) is composed of several nodes where each node is hosted by a dedicated cluster processor. Each FS node receives user queries and sends back the top-k results for each query to the users. After a query arrives to a FS node, it selects a caching service (CS) node. These nodes keep the top-k results for frequent queries previously issued by users so that response time for these queries is made very small. The query-results set for this service is kept partitioned into P_C disjoint subsets and each partition is replicated R_C times to increase query throughput. A simple LRU strategy can be used [12] to implement the CS nodes. A hash function on the query terms is used to select a CS partition whereas a respective replica (i.e., a CS node allocated in a cluster processor) can be selected in a round-robin manner.

PADS 2013 Montreal, Canada

If the query is not cached, it is sent to the index service (IS) cluster. The IS contains a distributed index built from the document collection held by the search engine (e.g. HTML documents from a big sample of the Web). This index is used to speed up the determination of what documents contain the query terms and calculate scores for them. The k documents with the highest scores are selected as the query answer. The index and respective document collection is partitioned into P_I partitions and each partition is kept replicated R_I times to enable high query throughput. Partitions help to reduce response time for individual queries since time is proportional to the number of documents that contains the query terms. Queries are sent to all partitions and the results are merged to obtain the global top-k results.

Figure 1 also shows an overall description of the message traffic among processors in a cluster of computers. These messages must pass through a number of communication switches to reach their destinations. Switches are usually organized in a fat-tree topology. On the other hand, each processor is expected to be a multi-core cluster node capable of concurrently executing a number of threads. Thus search engine service nodes are multithreaded components where it is necessary to take into consideration the effects in performance of concurrency control strategies.

The above description sets the level of detail at which simulators can be constructed to predict performance. We refer to simulators representing a good compromise between precision of performance metrics as compared against the ones obtained with the actual implementation of search engine components, and efficient running time of simulations.

Practice and experience tell us that such simulators can be achieved by considering the following guidelines:

- Include user behavior by feeding simulators with queries submitted by actual users from search engines. This implies executing large query logs with millions of queries containing different query arrival rates and topic trends expressed in the query terms.
- Hide complexities of simulating data center hardware by using models of parallel computation devised to represent key features of hardware cost. The model drives the construction of benchmark programs used to measure costs on the actual hardware.
- Determine the cost of relevant query processing operations by executing benchmark programs on actual samples of the Web. Typically there are available unithread implementations of query processing methods, or it is fairly simple to prototype them, and the aim is to study performance of very large systems.
- Organize the simulation of individual queries as directed graphs where arcs indicate the order in which query operations take place and vertices represent the cost of executing these operations on the hardware resources (threads, processors, shared memory and communication layer). Each operation competes for using the resources with co-resident queries.

In this paper we present two use cases of the above guidelines and propose their respective approximate parallelizations. The first one applies process oriented simulation to model query processing at multi-core architecture level where it is relevant to study how to efficiently organize threads in index service nodes supporting real time search. The second one applies event driven simulation of timed colored Petri nets to model the system described in Figure 1. These techniques, as applied following the guidelines we suggest above, are capable of predicting quite well performance of real search engines (error is less than 10%).

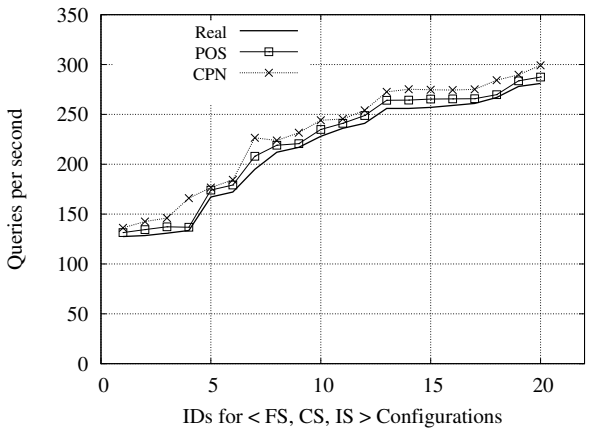


Figure 2: Simulation versus real implementations

For instance, Figure 2 shows results from sequential simulations of the system described in Figure 1. The curves labeled POS and CPN stand for process-oriented simulation and timed colored Petri nets respectively. The curve labeled "Real" shows results from an actual implementation of the same system. The figure shows query throughput results for different configurations given by specific values for the number of replicas of the FS, and the number of partitions and replicas of the CS and IS. In total, these configurations range from 115 to 240 processors. Overall, the results clearly indicate that simulations are able to predict performance trend of the service configurations. POS achieves better precision than CPN as it simulates the fat-tree communication infrastructure whereas CPN models it stochastically. However, a typical POS execution takes more than 4 hours to complete whereas running times for CPN are of the order of 15 minutes. Even for stochastic modelling of communication, we observed that POS running times are much larger than CPN running times since POS must cope with costly elements such as co-routine management.

Our parallelization proposal for the POS and CPN sequential simulators is generic and follows a standard late 90's trend for distributed discrete event simulation. Computations are organized in a bulk synchronous manner and there exists a rule to prevent processors from going too far into the simulation future without considering the state of event causality related processors. Ours are like optimistic distributed simulations where no rollback mechanism is employed and thereby they approximate sequential simulations of the same systems. Simulation objects are expected to be uniformly distributed at random on the processors.

In this paper we suggest that, given the nature of our application, a properly restrained bulk synchronous parallel simulation is capable of efficiently producing statistical results which are very similar to the results produced by a respective sequential simulation. We illustrate this point for process oriented and event driven simulation by means of our two use cases. We propose specific methods to parallelize the respective sequential simulators.

2. BACKGROUND AND RELATED WORK

Sequential discrete-event simulation techniques for performance evaluation studies in Web search engines and related systems are being applied by the information retrieval research community (c.f. [10]). To our knowledge, parallelization of these simulators have not been mentioned in the technical literature. Note that parallel simulators can be used not only in the context of research and development but also rutinarely in production systems operating as auxiliary components for search engines. An use case is as a capacity planning tool for data centers where engineers must periodically determine the amount of hardware resources assigned to search engine services in accordance with the observed query traffic. A challenging use case in terms of running time is as an automatic control support component in which the effects of user driven events in overall performance are simulated in real-time in order to tune the operation of the actual search engine.

Efficient strategies for parallel discrete-event simulation (PDES) have been widely studied in the late 90's and early 2000's literature [11, 25]. This topic has deserved recent attention in the context of clusters of multi-core processors [19, 15]. The work in [20] proposes a multi-grained parallelism technique based on the discrete Event System Specification (DEVS) methodology. The work in [6] presented a global schedule mechanism approach for improving the effectiveness based on a distributed event queue. In [8] the authors presented a distributed approach based on the migration of simulated entities. The work presented in [29] evaluated symmetric optimistic simulation kernels for a multi-core cluster which allows dynamic reassignment of cores to kernel instances.

In PDES, parallelism is introduced by partitioning the system into a set of concurrent simulation objects called logical processes (LPs). Events take place within LPs, their effect is the change of LP states, and LPs may schedule the occurrence of events in other LPs by sending event messages to them. Once the LPs are placed onto the processors, one is faced with the non-trivial problem of synchronizing the occurrence of events that take place in parallel so that the final outcome is equivalent to that of a sequential simulation of the same events. The Time Warp (TW) strategy [11] solves this problem by optimistically processing the occurrence of events as soon as they are available at the LPs and locally re-executing events whenever the simulation of earlier events is missed in one or more LPs.

In TW, the simulation time may advance at differing time intervals in each LP. Whenever a LP receives a "straggler" message carrying an event e with timestamp in the "past", the LP is "rolled back" into the state just before the occurrence of e and the simulation of the LP is resumed from this point onwards. That is, all the events with timestamps later than e's timestamp are re-simulated. The LPs save periodically their states to support roll-backs. In addition, any change propagated to other LPs as a result of processing erroneous events is corrected with similar roll-back method. To this end, special (anti-)messages are sent to the affected LPs in order to notify them of the previous sending of erroneous messages. During simulation, a quantity called global virtual time (GVT) is calculated periodically. Its value is such that no LP can be rolled-back earlier than it, and thereby the processed events and anti-messages with timestamps less than GVT are discarded to release memory. GVT steadily advances forward regardless the amount of roll-backs. When the rate of roll-backs is kept small, performance becomes very efficient.

In contrast, to achieve competitive performance, conservative synchronization [3] requires developers to enhance model definition with additional constructs such as lower bounds on the time the effects of events are propagated across the model. Instead, optimistic simulation makes it transparent for the developer to execute the simulation model either in parallel or sequentially. Another important advantage of optimistic simulation is that it does not impose any restriction on the communication topology among the simulation objects. That is, model developers are not required to be aware of parallelization details.

There are also recent works on parallel simulation of Petri nets models. A parallel simulator tailored to GPUs is presented in [13]. The work in [17] presents a strategy for distributed timed colored Petri net simulation. This scheme introduces a logical time and a priority vector to decide the exact ordering of all incoming and local events in an atomic unit of time.

A timed colored Petri Net (CPN) [16] is a high-level formalism that provides primitives to model latency, concurrency and synchronization of complex systems. It introduces the notion of token types (colors) which may be arbitrary data values. CPNs are represented as directed graphs with nodes containing places and transitions, and connecting arcs to indicate how tokens may flow through places and transitions. Places are graphically represented as ovals and transitions as rectangles. A place may have an initial set of tokens (initial marking). Arcs may contain expressions on token types/values used to indicate the condition by which one or more tokens may move from one transition to one place, and viceversa. Tokens passing through transition nodes may suffer time delays. There can also be expressions signaling tokens movement from several places (transitions) to several transitions (places). An expression is evaluated to true when all input places contain the required tokens.

Parallelization of process-oriented simulators has been considered in the literature as an involved task. Work in this topic has been presented in [21]. In our case, parallelization is simplified as we do not consider the execution of rollbacks to correct erroneous computations.

3. PARALLELIZATION METHOD

We use a bulk-synchronous multi-threading and message-passing strategy to automatically conduct simulation time advance. The practical implementation of these ideas comes from combining open-source software such as MPI [31], OpenMP [32], BSPonMPI [30], and LibCppSim [24] operating on top of the `C++` programming language.

To explain the parallelization method we resort to BSP parlance [28]. In BSP the computation is organized as a sequence of *supersteps*. During a superstep, the processors (master threads) may perform computations on local data and/or send messages to other processors. The messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors. The underlying communication library ensures that all messages are available at their destinations before starting the next superstep.

We assume a cluster of P processors where each processor contains T threads so that simulation objects are evenly

distributed on the $T \times P$ threads. Each thread has its own instance of a sequential simulation kernel (e.g., LibCppSim for process-oriented simulation) which includes a local event list and functions related to event processing and message handling. In each processor there is one master thread that synchronizes with all other $P-1$ master threads to execute the BSP supersteps and exchange messages. In each processor and superstep, the remaining $T-1$ threads synchronize with the master thread to start the next superstep, though they may immediately exchange messages during the current superstep as they share the same processor main memory.

As it is well-known, processing events in parallel during periods of time where no messages from other processors are received can lead to the problem of missing the arrival of event messages at the right simulation time. We simply ignore such situations though (when possible) proper care is taken to correct measures such as response time of individual queries. The challenge is to limit the extent in simulation time of these straggler messages [11] by making no assumptions on the event time increments and communication topology among the simulation objects run by the $T \times P$ sequential kernels. We achieve this by means of a windowing strategy.

3.1 Sliding time window strategy

In each superstep the simulation time is advanced W units of time. A new value for W is obtained every n supersteps to follow variations in the evolution of the simulation model. The method extends ideas from [22]. Each processor stores tuples (t, s, d) for each message sent to other processor, where t is the occurrence time of the event, s is the source processor and d is the destination processor. At the end of the n-est superstep, all processors send their arrays of tuples (t, s, d) to one master processor. The master contains an array C holding P supersteps counters initialized to zero and an event list that is initialized with the incoming tuples (t, s, d). This allows the master to determine the minimum number of supersteps S required to process the (t, s, d) tuples. To this end, the master processor extracts them from event list in chronological order and for each tuple (t, s, d) it sets $C[d] = \max\{C[s] + 1, C[d]\}$ where $s \neq d$. Once all tuples have been extracted, the total number of supersteps S is the maximum value in the array C. Therefore, if Δ is the total simulation time increment from the first to the last (t, s, d) tuple, the window W is defined by $W = \Delta/S$.

In each superstep, all threads process events with time less than or equal to B and the time barrier B is increased by $B = B + W$ in the next superstep, and so on. The global virtual time (GVT) of the distributed simulation is defined as the event with the least time across all threads [11]. This value is calculated at the end of any superstep after all messages have been delivered and inserted in their respective kernel event lists. Thus assuming that a new value of GVT is produced every n_g supersteps, the time barrier B is periodically re-initialized to $B = \text{GVT} + (n_g + 1) \cdot W$.

3.2 Self-increment of supersteps

The above windowing scheme may be prone to pathological cases in which the minimum number of supersteps S actually degenerates event processing into something similar to a sequential simulation. For instance, assume a two processor system where in one processor we place simulation objects that only send messages to be consumed by simulation objects located in the second processor. In this case, the values of S calculated by the master processor is just 2 and the simulation becomes sequential. Namely, in the first superstep, processor 1 generates all messages, and in the second superstep, processor 2 causes the occurrence of all events caused by those messages whilst processor 1 performs no event processing. Patterns similar to this could randomly take place in a very large simulation model during unpredictable periods of simulation time.

To solve this problem, during the chronological processing of tuples (t, s, d) we impose an upper limit to the number of tuples processed in each superstep and processor. This forces the self-increment of the superstep counters $C[s]$ for processors that have not received messages during a long period of simulation time. We define a processor global event list as the union of the event lists of all threads co-located in the same processor. We use the average size L_E of the global event list maintained in each processor to assume that in each superstep, each processor should consume no more than one of its respective global event list. The time elapsed in the consumption of these events indicates the average size of an auxiliary sliding time window W_2. This window is used to force the self-increment of the counters $C[s]$ when required in the master processor and it can be calculated locally in each processor. In particular, for a processor s that has simulated $N_E[s]$ events in the period of n supersteps, the value of its window $W_2[s]$ can be calculated as $W_2[s] = f \cdot (\Delta/N_E[s]) \cdot L_E[s]$ where $f \geq 1$ is a slackness factor like 1.3. The window $W_2[s]$ along with tuples (t, s, d) is sent to the master processor.

The average value of L_E observed during the n supersteps is calculated as in [26], that is, it is efficiently calculated by modelling the problem as a G/G/∞ queuing system. In this case the average number of active servers is L_E, which is defined as the ratio of the arrival rate of events λ to the service rate of events μ, namely $L_E = \lambda/\mu$. At steady state, for n large enough, N_E events are inserted in and N_E events are extracted from the event list. Thus the arrival rate is given by $\lambda = N_E/\Delta$. In addition, in this model, each event e_i requires a "service time" δ_i that is given by $\delta_i = t_a - t_b$, where t_b is the simulation clock at the time in which e_i is inserted in the event list, and t_a is the simulation time at which the event is scheduled to take place. Thus the average service rate μ is given by $\mu = N_E / \sum_{i=1}^{N_E} \delta_i$, and thereby $L_E = \sum_{i=1}^{N_E} \delta_i / \Delta$. The simulation kernels calculate these statistics as they process events during the n supersteps.

4. USE CASE 1: APPLICATION FOR PROCESS ORIENTED SIMULATION

The index services of current search engines are expected to be designed to accept concurrent updates to support real time search. The objective is to include in the query results Web documents published in the very recent past. This leads to the problem of efficiently organizing multi-threading so that read and write conflicts are prevented from happening as query processing (read only transactions) and index updates (write only transactions) take place concurrently in each multi-core processor. A number of concurrency control strategies can be borrowed from the OS and DB literature, and the problem is to determine which of them are more efficient and scalable in the search engine domain [2].

Web search engines use inverted indexes to determine the top-k results for queries. The index is composed of a vocabulary table and a set of inverted or posting lists. The vocabulary contains the set of relevant terms found in the text collection. Each of these terms is associated with an inverted list which contains the document identifiers where the term appears in the collection along with the term frequency in the document. Frequencies are used to calculate document scores. To solve a query, it is necessary to get from the inverted lists the set of documents associated with the query terms and then to perform a ranking of these documents based on their score values. Updates in the index take place when a new document is stored in the respective processor. This implies modifying the posting lists associated with the terms present in the new document.

People tend to favor reduced subsets of terms both in queries and documents, which means that some terms are much more popular than others in the inverted index. This has the potential of causing R/W conflicts among running threads in search engines serving workloads of hundred thousand queries per second.

To illustrate modelling and simulation of this use case we focus on a simple concurrency control strategy. There exists a processor input queue that contains the arriving read and write transactions, and any thread requesting a transaction (1) locks the queue, (2) removes the transaction, (3) requests locks for all terms involved in the transaction, (4) when all locks are granted it releases the lock on the input queue, (5) then it safely starts processing the transaction, and (6) once processing is completed, it releases all locks.

The simulation model uses a processes and resources approach. Processes represent threads in charge of transaction processing. Resources are artifacts such as inverted lists and global variables like exclusive access locks. The simulation program is implemented using LibCppSim [24], where each process is implemented by a co-routine that can be blocked and unblocked at will during simulation. The operations *passivate()* and *activate()* are used for this purpose. Co-routines execute the relevant operations of threads/transactions. The operation *hold*(t) blocks the co-routine during t units of simulation time. This is used to simulate the different costs of transactions and the cost of the different tasks performed by the simulated multi-core processor. The dominant costs come from operations related to ranking of documents, intersection of inverted lists and update of inverted lists. These costs are determined by benchmark programs implementing the same operations. Figure 3 illustrates the main ideas where we placed the *rank()* operation to emphasize that these simulations can also be driven by the outcome of the actual process being simulated.

To include the effects of the processor architecture in performance we use the Multi-BSP model of parallel computation [27] proposed recently for multi-core processors. Using process oriented simulation we extend the model with processor caches and special lock variables. The processor cache entries (cache lines) are managed with the LRU replacement policy, where each entry is a memory block of 64 bytes like Intel processors, and we use a directory based coherence protocol for cache entries.

In the Multi-BSP model a multi-core processor is seen as a hierarchy of memories (caches and main memory) that form a tree with root being the main memory and leaves being pairs (core, cache L1). To simply description let us assume

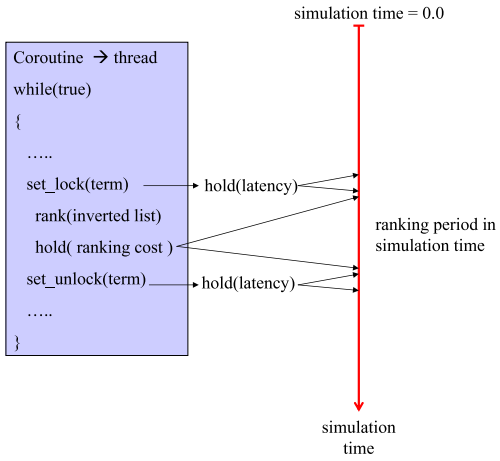


Figure 3: Concurrent routines simulating the steps taken by a thread to process a query (read-only transaction) and generating cost in simulation time.

a processor composed of eight cores, where four pairs (core, L1 cache) are connected to a single L2 cache, and two L2 caches are connected to the main memory. Cores (CPUs) can only work on local data stored in their respective L1 caches. When a core references data out of its L1 cache, the data is look at the respective cache L2. If found, a cache line transfer (or several lines if data pre-fetching is enabled) is performed between caches L2 and L1. The cost of this transfer is given by a parameter g_1. If not found, a cache line sized piece of the data is transfered from main memory to cache L2 at cost g_2, and then from L2 to L1 at cost g_1.

Computations in Multi-BSP are organized as sequences of three supersteps where each superstep is ended with a synchronization barrier. During the first superstep, all data transfers take place between main memory and the two L2 caches. During the second superstep, all data transfers take place between the L2 and L1 caches. During the third superstep, all cores are allowed to perform computations on the data stored in their respective L1 caches. Certainly this bulk-synchronous structure of computations is intended to make mathematically tractable the analysis of algorithms. We do not require this structure but it helps during (1) simulation program debugging, (2) benchmark program construction to measure g_1 and g_2, and (3) understanding performance. Process oriented simulation enables us to simulate these actions asynchronously. In addition, the LRU policy and the read/write coherence protocol on caches provide a more realistic setting to our processor model.

Lock variables are forced to be global by keeping them updated in main memory at all time. This accounts for its fairly larger ($g_1 + g_2$) cost than standard local and global program variables.

Figure 4 shows the steps followed during the simulation of a thread that executes a cost dominating transaction operation that takes `time_cpu` units of simulation time and works on a piece of data whose space address is given by `[base_address:offset_bytes]`. To this end, the respective simulator co-routine executes the function `run( time_cpu, base_address, offset_bytes )`. The cost of data transfers and computation performed by cores is simulated by calls to functions of the form `thread->causeCost*(...)`, which con-

```
Core::run(  double time_cpu,
            string base_address, int offset_bytes )
{
   int nblocks = offset_bytes / cache_block_size;
   time_cpu = time_cpu / nblocks;

   for( int i= 0; i < nblocks; i= i+1 )
   {
      sprintf(str,"%s %d", base_address.c_str(), i);
      if ( cacheL1->hit( str ) == true )
      {
         cacheL1->update( str );
         thread->causeCost_CPU( time_cpu );
      }
      else
      if ( cacheL2->hit( str ) == true )
      {
         cacheL2->update( str );
         thread->causeCost_TransferL1L2( Latency_g1 );
         cacheL1->insert( str );
         thread->causeCost_CPU( time_cpu );
      }
      else
      {
         thread->causeCost_TransferL2Mem( Latency_g2 );
         cacheL2->insert( str );
         thread->causeCost_TransferL1L2( Latency_g1 );
         cacheL1->insert( str );
         thread->causeCost_CPU( time_cpu );
      }
   }
}
```

Figure 4: Simulating the execution of a cost dominating operation defined by *run*(...), which is executed by a simulator co-routine.

tain corresponding hold(t) operations. Cache entries that get replaced by the LRU policy and have been modified by threads, are copied to the lower memory in the hierarchy (L1 $\rightarrow$ L2, L2 $\rightarrow$ main memory). The coherence protocol maintains track of cache entries that contain copies of the same data so that when a thread modifies an instance of a replicated data, the remaining copies are invalidated.

The implementation of exclusive locks associates a name string with a lock variable along with a queue for co-routines (threads) waiting to get access. Lock administration takes a latency $g_1 + g_2$ in simulation time, as they are always read and written in main memory.

4.1 Parallelization

The LibCppSim library ensures that the simulation kernel grants execution control to co-routines in a mode of one co-routine at a time. Co-routines are activated following the sequential occurrence of events in chronological order. The co-routines implement simulation object (called processes in LibCppSim) and we communicate each other through message passing. When an object x places a message in object z the object x can activate z if z has been passivated earlier in the simulation.

Assuming that every time a simulation object is executing code no other simulation object is allowed to execute code, then we define an auxiliary BSP simulation object (co-routine) to make it in charge of communicating and synchronizing with other LibCppSim instances running in other processors or even running in other cores of the same processor. Each LibCppSim instance is a separate Unix process and BSPonMPI enables them to communicate and synchronize each other in determined points of the simulation time.

To this end, we use the BSPonMPI library that provides primitives to send and receive messages among Unix processes deployed on different processors or cores, and it provides a primitive to barrier synchronize Unix processes so that after the synchronization point all messages are delivered at their destinations (i.e., there are no messages in transit after the synchronization point). The Unix processes can be executed on different physical processors by means of the MPI library. Thus the auxiliary BSP object running in each LibCppSim instance can execute the following steps:

```
void inner_body( void )
{
   while( time() <= END_SIM_TIME )
   {
      hold( Sim_Time_Window );

      Send all messages buffered during the period.

      bsp_sync();// synchronize all Unix processes.

      Receive all messages from Unix processes and
      place them in their destination objects.
   }
}
```

The value of `Sim_Time_Window` is periodically adjusted to fit with the automatic windowing scheme described in Section 3. A new superstep starts after the `bsp_sync()` operation finishes. This operation (1) waits for all Unix processes to reach a synchronization point, (2) causes the sending and reception of all messages, and (3) let each Unix process continue computations. In the BSP simulation object the associated co-routine retains control of the execution until the call to the next hold(t) operation.

The relevant part of the cost in running time comes from the execution of the operation run(...) presented in the Figure 4. This operation acts on a set of L1 and L2 cache objects, and simulation objects representing threads cause cost in the simulation time depending on the contents of these cache objects. Thus for object distribution, a straightforward approach is to store related simulation objects (threads and caches) in the same processor to exploit locality.

5. USE CASE 2: A FRAMEWORK FOR CPN EVENT DRIVEN SIMULATION

The parallelization strategy of the process oriented simulation application presented in the previous section was specific to the problem at hand. To this end, it was necessary to define (1) a BSP simulation object and related housekeeping for message passing, (2) a particular distribution of simulation objects among processors, and (3) a strategy for approximating the sequential simulation. In addition, no shared memory was exploited in each processor due to the single process design of LibCppSim. That is, cores in processors are treated as individual processors by placing LibCppSim Unix process on them by means of BSPonMPI and communication is through message passing.

In this section we present a framework to model and simulate general Web search engines using timed colored Petri nets. This illustrates the fact that the parallelization strategy proposed in Section 3 used in combination with the *key* based approach and framework described in this section enable to hide the complex details of parallelization to the user.

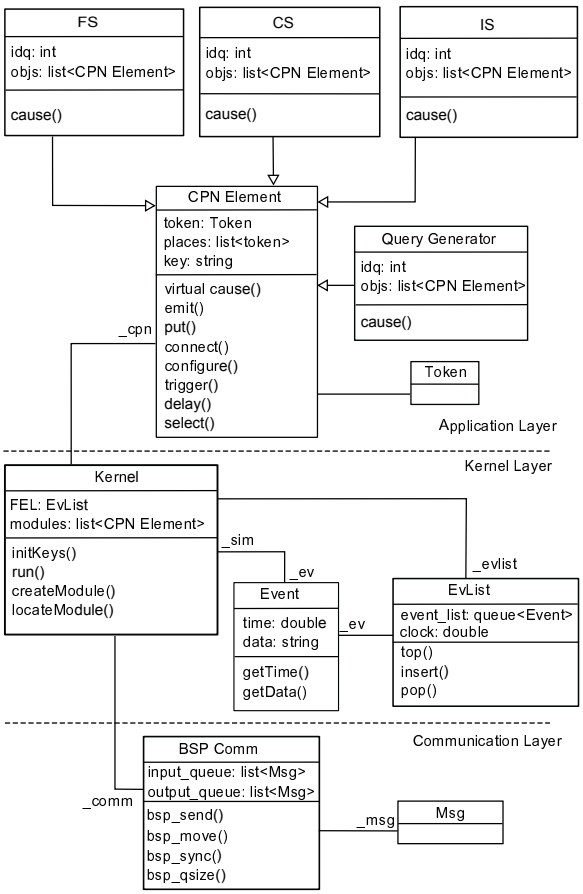
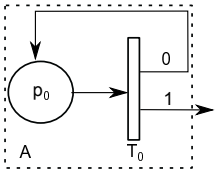


Figure 5: Abbreviated class diagram.

Figure 5 shows a class hierarchy for our software framework applied to the example of Figure 1.

The communication layer (at the bottom) uses the *BSP comm* class. Messages move Petri net tokens from one processor to the another so that tokens can be circulated through the distributed graph composed of transitions and places. The kernel layer defines the simulation kernel classes used to sequence events and administer simulation objects belonging to classes derived from the class *CPN element*. There is one simulation kernel per running thread, and one thread per processor core. One master thread per processor has access to the BSP communication layer and the remaining threads are barrier synchronized to wait for the start of the next superstep. During supersteps co-resident kernels may exchange tokens by using exclusive locks to prevent from R/W conflicts. We use OpenMP to administer the set of threads deployed in each processor.

The application layer consists of a set classes used to specify the user model. To this end, there is a *CPN Element* class with a set of pre-defined attributes and methods required to implement a timed colored Petri net (CPN) model. Figure 6 shows an example for a simple CPN Element named A. It is composed by one place P_0 and one transition T_0 with two out-links. This is specified by *configure("Places=1; Transitions=1; Ports=2")*. The *connect("Port=0; Place=0" ,A)* operation connects the first out-link of the transition T_0 to the place P_0. The function *connect()* is used to set connections between pairs of CPN elements. Namely, *connect()*




```
cpnElement *A= new cpnElement();
A->configure("Places=1;
              Transitions=1;
              Ports=2");
A->connect("Port=0;Place=1",A);

A->transition( Delay(),
               Trigger(),
               Select() );
```

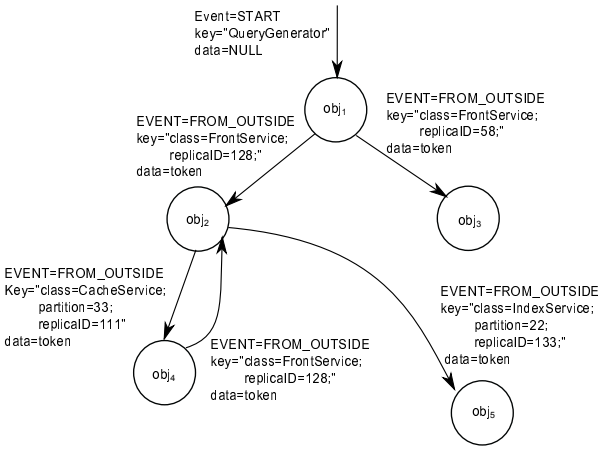
Figure 6: CPN Element configuration code.

defines which out-link of transition i is connected to which in-link of a place j, so that tokens can flow from i to j. The `C++` template binary functions *Trigger()*, *Delay()* and *Select()* are defined by the user to specify the behavior of the transition. In general, these functions perform the following: (1) *Trigger* determines if the corresponding transition must be fired, (2) *Delay* returns the duration of the transition in simulation time units, and (3) *Select* defines the out-link to be followed by a token after the transition delay has finished. The framework provides pre-defined version of these functions with standard behavior for CPN elements.

In general, the idea is that the user defines CPN modules which are composed of one or more objects of class *CPN Element*. Each simulation kernel simulates sequentially the occurrence of events in the CPN modules hosted by the kernel. The *cause()* operation is defined by the user to handle the events received and internally generated by the respective module. These events may indicate the arrival and departure of tokens. The user has to explicitly define the route followed by those tokens as shown in the `C++` fragment code located in the bottom part of Figure 7. In this case, the *put()* operation is used to place the token in one of the CPN elements of the module whereas the *emit()* operation is used to send a token to another module. The destination module may be located in the same simulation kernel, or it may be located in a co-resident simulation kernel or located in a remote simulation kernel hosted by other processor.

To make CPN module location transparent to the user and also to enable scaling to a very large number of modules, we use a *key* based approach. Each CPN module is associated with an unique *key*. Any departing token must specify the module destination key. The concept is that the token is *emitted* to the "outside" of the module so that it is left to the kernel the responsibility of routing the token to the right destination module. Keys are strings and thereby char arrays can be defined to assign identifiers to modules in a flexible manner. Modules are automatically instantiated and deployed on processors by the simulation framework upon the emission of a new key during simulation.

The typical content of a *key* is a string like "class= *ClassName*; instance=*ID*", say "class= FrontService; replica=128", or a more convenient definition for a search engine such as "class= IndexService; partitionID=22; replicaID=133". The only requirement is to specify the class identification field to let the simulation framework instantiate the right object. The whole simulation is started by emitting one or more keys which identify the class names of the modules in charge of generating the initial tokens of the simulation. The simulation framework applies a hash function on the keys to decide in what processors and threads are the respective modules deployed and subsequently located.




```
cause( Token token ) // token arrival
{
    switch( token->type() )
    {
      case FROM_OUTSIDE:
           A->put("Place=0",token);
           break;
      case FROM_INSIDE:
           emit(key,token); // token departure
           break;
    }
}
```

Figure 7: *Key* **based approach example.**

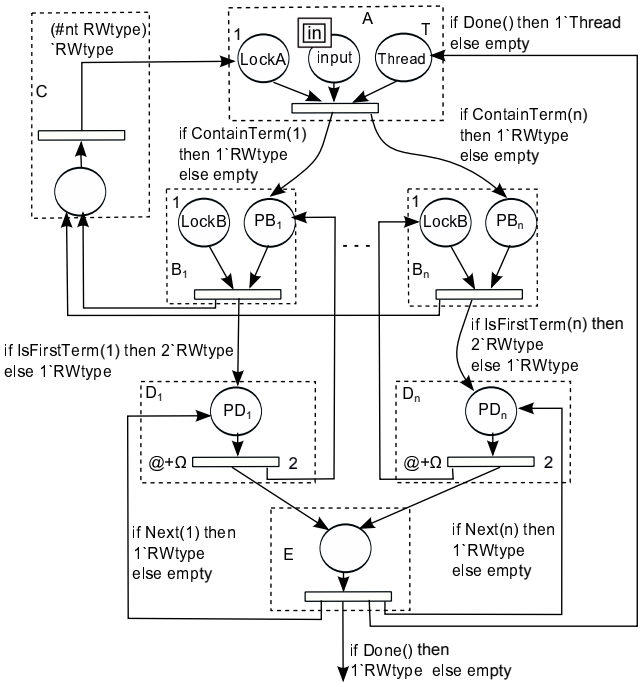


Figure 8: CPN module for an index service node.

The upper part of Figure 7 shows an example where the initial key is generated with the *START* event which creates a *Query Generator* object. The other objects are created as soon as the query generator sends tokens to them. In the figure, obj_1 contains the query generator module, objects obj_2 and obj_3 are front-service modules, and obj_4 is a cache service node whereas obj_5 is an index service node. All these objects are expected to be complex CPN modules as they must also simulate the cost of executing service operations in the respective cluster processor. They stand for timed CPN models specifically designed to represent the cost of query processing in search engines as described in [7].

5.1 CPN modelling example

For lack of space we are not able to provide more details on CPN modelling. Nevertheless, to provide an idea of how a CPN module, with all its CPN element members, might look like, we present in Figure 8 a model of the multi-threading strategy presented in Section 4 for process oriented simulation. The diagram uses CPN tools notation [33]. For an index node operating on a processor with T threads, the input place in element A accepts arriving tokens representing read and write operations which we identify as tokens of type "RWtype". The place *Lock A* is used to lock the input queue so that a new RWtype token can be removed from the queue. Then, for each term i contained in the R/W operation, the *ContainTerm(i)* function triggers a copy of the token be stored in place PB_i of element B_i. The place *Lock B* in each element B_i is used to lock the associated posting list so that one token at a time is sent to the place PD_i of the succeeding element D_i. Thus for each term i, a pair (B_i, D_i) is involved in token displacement. In total there are n pairs (B_i, D_i), one per term i in the inverted index. Elements D_i represent either the cost of performing document ranking in the posting list (read operation), or the cost of updating the posting list (write operation). The service time required by these operations is denoted by @+Ω.

Notice that the elements D_i must be visited sequentially and the lock on each element B_i must be released immediately after visiting each D_i. To this end, the transition in D_i is triggered with two tokens which occurs when either (1) the value i refers to the first term in the operation as determined by the *IsFirstTerm(i)* function or (2) the second token arrives from another D_j (through element E) that contains a term located just before term i in the R/W operation being solved. Element E is used to decide the next D_i to be visited as determined by the *Next(i)* function or to finish the token movements when the *Done()* function returns true.

Consider as an example a query operation with two terms 1 and 2. One token of type query enters into the place PD_2 of D_2 and two tokens of type query enter into the place PD_1 of D_1 which triggers the transition of D_1. After the first query term 1 has been processed by the pair of elements (B_1,D_1) a token of type query enters into the place of the element E. The next term to be visited is set to 2. Then, only the arc connecting element E with element D_2 is evaluated to true, and a second token of type query enters into the place PD_2 of D_2. This last action triggers the transition of D_2. After this, the token enters into the place of element E. Now the *Done()* function returns true and the token of type query is sent to the remitting FS processor.

At all time, at most T R/W operations may go beyond element A. This is solved with place *Thread* in A so that each time a RWtype token departs from element E, a token of type "Thread" is returned to place *Thread* in A. The variable *#nt* contains the number of terms in each R/W operation and element C is used to release the input queue when the current R/W operation has obtained all posting list locks for its terms.

6. EXPERIMENTS

We performed experiments with a log of 36,389,567 queries submitted to the AOL search service between March 1 and May 31, 2006. This query log exposes typical patterns present in user queries [1, 12]. We pre-processed the query log following the rules applied in [12]. The resulting query log has 16,900,873 queries, where 6,614,176 are unique queries and the vocabulary consists of 1,069,700 distinct query terms. We also used a sample (1.5TB) of the UK Web obtained in 2005 by the Yahoo! search engine, over which a 26,000,000 terms and 56,153,990 documents inverted index was constructed. We executed the queries against this index to get the cost of the query processing operations. The document ranking method used was WAND [4] (index service) and the cache policy used was LRU (caching service). All benchmarks were run with all data in main memory to avoid access to disk. Results were obtained on a cluster with 256 nodes composed of 32-cores AMD Opteron 2.1GHz Magny Cours processors with 16GB of main memory per node. Simulations are driven by query traces containing information such as query terms, ranking cost, and posting lists size.

We have conducted experiments with three different CPN models in order to test the behavior of our approximate parallel discrete-event simulation under different scenarios:

Log: This model describes a Web search engine composed of a set of front-end (FS) nodes, index service (IS) nodes and caching service (CS) nodes (6,047 nodes in total). Queries are injected in the FS nodes by using the above AOL query log. Queries are not actually stored in the caches of the CS service. Instead, each CS node uses a probabilistic function where 15% of queries processed are signaled as cache hits (i.e., they are found in the cache). The CPN models for FS and IS are described in Figure 9. The model for CS is similar to IS but with different cost for query processing operations. The model for the multi-threaded processor module is a simplified version of the model presented in Figure 8 since in this case it is not necessary to lock posting lists as all transactions are read only.

LogCache: This model is similar to the above Log model but it includes actual implementations of two levels of caches in the CS. The first level consists of a results cache which stores the most frequent queries with their top-k results. In the second level we include a location cache which stores the partition IDs of the index service capable of placing documents in the top-k results for frequent queries. The idea is that when a query is not found in the results cache, a second look-up is performed in the location cache to decide whether to send the query to all partitions of the IS or to a subset of them. The memory space used by location cache entries is less than 1% of the result cache entries. Frequent queries requiring a small number of partitions to be solved are good candidates to be kept in the location cache. The total number of cache entries in the results cache is adjusted to achieve a similar hit ratio than in the Log simulator (15%).

AsyncRW: This is the multi-threading model described in Figure 8. The model represents a single multi-core processor that must process read and write transactions given by user query and index update operations respectively. We simulate the CPN model described in Figure 8 and the process oriented simulation model described in Section 4. Read transactions are user queries whose answers must be calculated by reading all items of the posting lists associated with the query terms. Write operations insert new items in the posting lists. We perform experiments with 100,000 terms. That is, the model contains CPN elements $B_1 \dots B_n$ and $D_1 \dots D_n$ with $n = 100,000$ in Figure 8. It simulates $T = 128$ threads which means instantiate the model with 128 tokens in place *Thread* of the CPN element labeled A in Figure 8. No processor caches are considered in the CPN model. On the other hand, the process oriented model is also simulated for 128 threads and it contains the organization of processor caches L1 and L2 described in Section 4 (Figure 4).

6.1 Evaluation of Performance

Figure 10 shows how the window size W dynamically adapts it-self as the simulation time advances forward. With a larger number of processors the window size tends to be smaller since simulation objects are distributed in more processors and thereby message passing among them increases the minimum number of supersteps S required to process the (t, s, d) tuples (Subsection 3.1). However, the figure shows curves in which the number of processors P is doubled and the window size grows in a much smaller proportion. This is independent of the number of threads (simulation kernels) executed in each processor. The reason is that the minimum number of supersteps does not grow linearly with the number of processors. This is observed in Figure 11 for different number of processed events in the simulations. In these experiments the number of processed events ranges from one million to eight millions and the relative increase in S with P remains fairly stable. In general, the total number of real supersteps executed by the simulator is about 30% larger than the minimum number of supersteps required to simulate the same period of simulation time.

In Figure 12 we show results for the percentage of events that should be re-simulated if our parallelization strategy were implemented with a roll-back mechanism. In some cases the fraction of events that must be re-simulated reaches 30%, though it could be reduced by reducing the size of the window W. As expected this is correlated with the number of simulation objects hit by queries. In the case of the Log simulator after each cache miss the respective query is sent to all index service partitions whereas in the LogCache simulator the number of partitions is reduced due to use of the location cache. Thus in the LogCache simulator less simulation objects are hit by queries than in the Log simulator.

In Figure 13 we show speed-up results for the CPN models of the search engine described in Figure 1. The results show some advantage of performing asynchronous multithreading in each processor during supersteps. For instance, the speed-up for (P=1, T= 32) is better than for (P=8, T= 4) where P is the number of processors and T the number of running threads in each processor.

In Figure 14 we show speed-up results for the AsyncRW model simulated using the CPN model described in Figure 8 and the process-oriented (PO) model described in Section 4. The results show that PO simulations scale up better than CPN simulations. However, PO simulations are much more demanding in running time than CPN simulations. Granu-

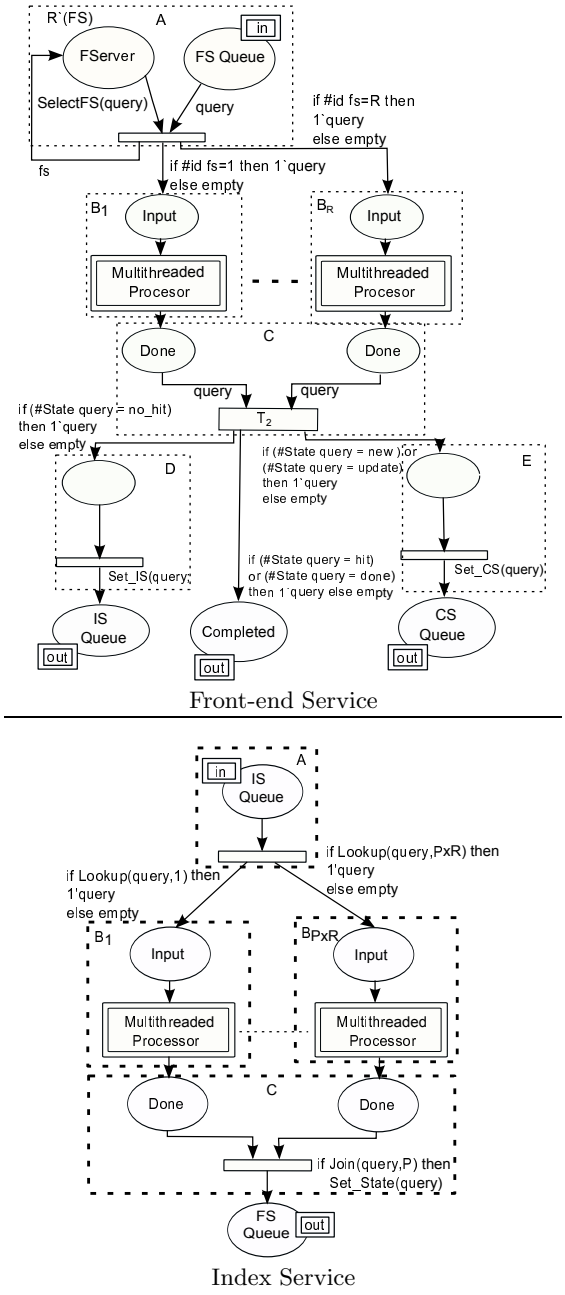


Figure 9: CPN models of services.

larity of computation per event is large and load balance is near perfect and thereby speed-ups are better in PO simulations. The LibCppSim library is not thread safe so only Unix processes were distributed on the processor cores.

6.2 Evaluation of Accuracy

In this section we evaluate the accuracy of our approximate parallel simulations. That is, how well they approximate the performance metrics determined by the sequential simulation of the same systems. Regarding global metrics such as query throughput there is no relevant differences, just small fluctuations less than 1%. Therefore we first focus on a much more demanding case. We compare the relative values of the response time of individual queries. To this end, we compute the Pearson correlation among query

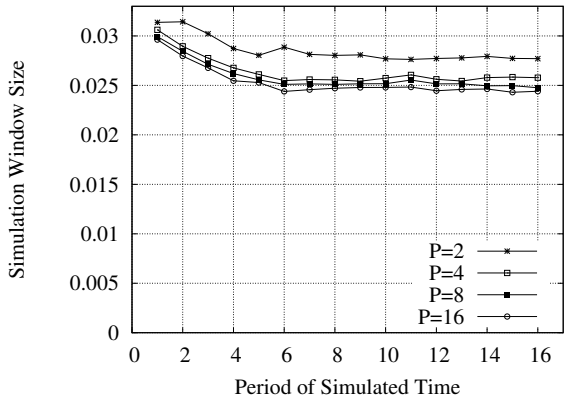


Figure 10: Window size along simulation time.

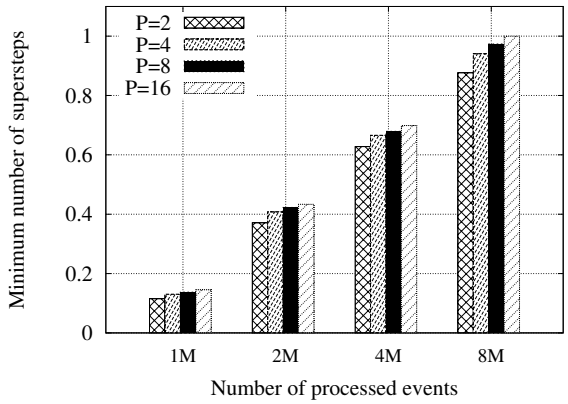


Figure 11: Minimum number of supersteps.

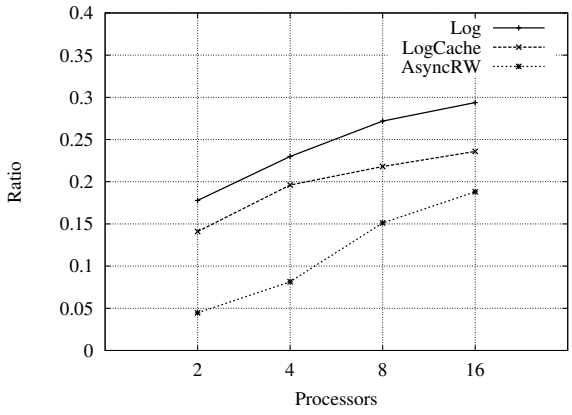


Figure 12: Ratio of number of events that should be re-simulated to total number of processed events.

response times reported by the sequential simulation and the response times for the same queries as reported by our approximate parallel simulations.

In Figure 15 we show the Pearson correlation ($y-$axis values) achieved for different random samples of queries ($x-$axis values) and curves for different number of queries N included in the samples. Each sample is composed by N pairs $< id\_query, response\ time >$ obtained as follows. We identify a query q_i which produces a straggler event in a

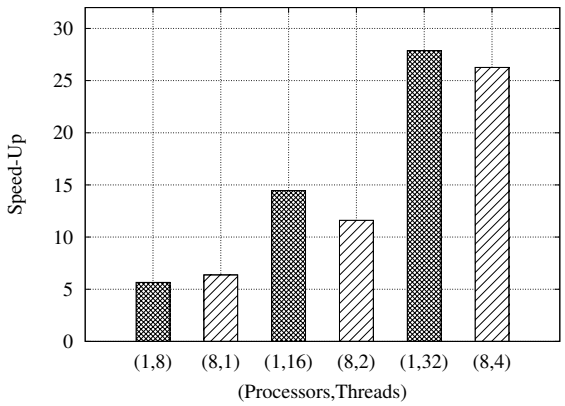


Figure 13: Speed-up for the Log simulator (similar results are obtained for the LogCache simulator).

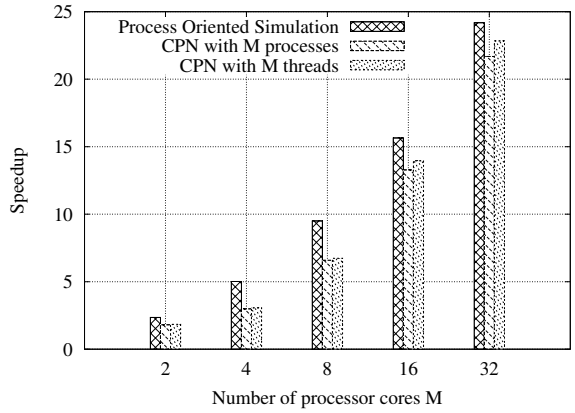


Figure 14: Speed-ups for the CPN and process-oriented AsyncRW models.

CPN module and we build the sample by taking $N/2$ queries whose events take place before and $N/2-1$ queries whose events take place after q_i in the simulation time in the same CPN module. Namely, each sample contains the neighborhood of a query that would have caused a roll-back in an equivalent Time Warp simulation of the same system. The results shown are for the Log simulator as it is the simulator that achieves the largest number of potential roll-backs as compared with the LogCache and AsyncRW simulators.

Figure 15 shows that even with a small sample size of $N = 10$ the parallel simulations have a Pearson correlation of almost 0.9 (with 1.0 meaning perfect correlation). In other words, there is a strongly positive correlation among the results from the sequential and approximate parallel simulations. We expanded these results for a single sample containing the whole set of 16,900,873 queries, for different number of processors and threads per processor, and for the Log, LogCache and AsyncRW CPN simulators. All of the results were above 0.998.

We also executed the Kolmogorov–Smirnov (KS) test over the query response time data which is a non-parametric test that allows comparing the cumulative distributions of two samples. It is used for verifying whether two samples are drawn from the same distribution. Like in the Pearson test, one sample is given by the query response times obtained

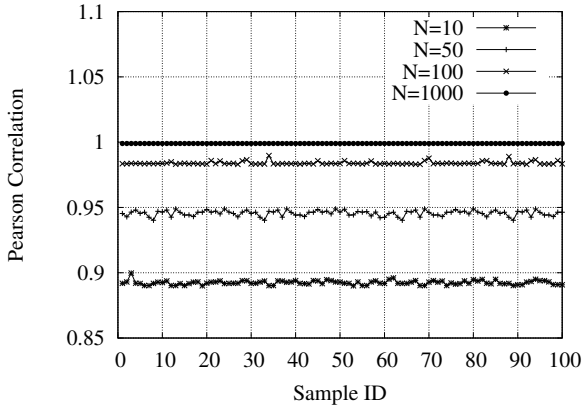


Figure 15: Pearson correlation obtained with different sample sizes.

with sequential simulations and the second sample is given by the query response times obtained with approximate parallel simulations for different values of the pairs (processors, threads per processor). In Figure 16 we show the ρ-value of the KS test which reports if the samples differ significantly (ρ= 1 means no discrepancy). As the ρ-values close to 1 we cannot reject the hypothesis that the two samples come from the same distribution. Only the Log simulator reports a few ρ values less than 1 but they are above 0.8 which is a value large enough to claim that both distributions are very similar. The values are for the whole query log (N=16M) and the average taking small samples of 100 queries (N=100).

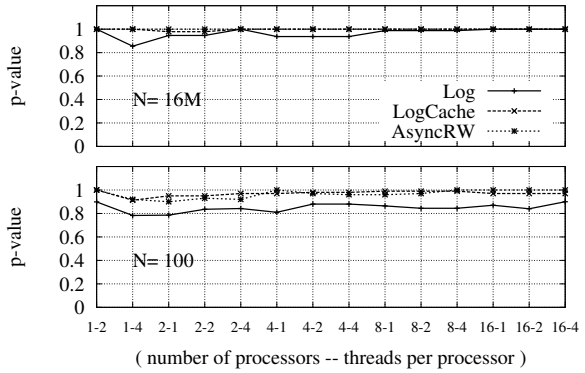


Figure 16: Kolmogorov-Smirnov test.

Finally we calculated the root mean square error of the deviation which is a measure of the differences between values obtained by the sequential simulation and the values reported by the parallel simulations. It is defined as $\epsilon_m = \sqrt{(\sum(x_i - \overline{x})^2/(n \cdot (n-1))}$ and we calculated the relative error (e_r) as $\epsilon_m/\overline{x}$.

In Table 1 we show the relative errors achieved by the parallel simulation for the following global metrics: query throughput, query response time (minimum, maximum and average over all queries) and hit ratio observed in the query caches. The row for query response time shows the biggest error among the minimum, maximum and average response times. In general, the error values reported in the table for the global performance metrics are very small.

Table 1: Relative error for global metrics.

	AsyncRW	Log	LogCache
Throughput (e_r)	0.03%	0.1%	0.3%
Response Time (e_r)	Min. 0.4%	Max. 0.6%	Max. 0.9%
Hits R. Cache (e_r)	-	-	0.2%
Hits L. Cache (e_r)	-	-	0.05%

7. CONCLUDING REMARKS

We have presented approximate parallel discrete event simulation strategies for performance evaluation of search engines. The proposed methods are generic and can be used to evaluate alternative designs for search engine components and support capacity planning studies for data centers.

The experimental evaluation shows that our approximate parallel simulation methods achieve good statistic agreement with results from sequential simulations of the same search engine realizations. The nature of this application makes it suitable for relaxed synchronization of events in the simulation time across processors. These simulations are driven by the processing of individual queries where each query competes for using the hardware resources with other many thousand queries competing for the same resources. As long as queries in the parallel simulation compete against similar number of queries than in the sequential simulation, their simulated processing costs tend to be weakly affected by eventual straggler events with timestamps in the past arriving at the processors.

Overall, the benefits of our proposals are efficient parallel simulations in running time and memory usage, and the possibility of preventing the user from getting involved in the complex details of parallelization. We demonstrated these features by presenting a `C++` framework for event driven simulation of timed colored Petri net models for search engines. We also showed that process oriented simulation models can be efficiently parallelized. Our proposal consisted of combining a sliding time window strategy with bulk-synchronous parallel processing at thread and processor levels.

8. REFERENCES

[1] C. S. Badue, J. M. Almeida, V. Almeida, R. A. Baeza-Yates, B. A. Ribeiro-Neto, A. Ziviani, and N. Ziviani. Capacity planning for vertical search engines. *CoRR*, abs/1006.5059, 2010.

[2] C. Bonacic, C. García, M. Marín, M. Prieto-Matías, and F. Tirado. Building efficient multi-threaded search nodes. In *CIKM*, 2010.

[3] A. Boukerche and S. K. Das. Dynamic load balancing strategies for conservative parallel simulations. *SIGSIM Simul. Dig.*, 27(1):20–28, 1997.

[4] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, 2003.

[5] C. D. Carothers and R. M. Fujimoto. Efficient execution of time warp programs on heterogeneous, now platforms. *TPDS*, 11(3) 2000.

[6] L.-l. Chen, Y.-s. Lu, Y.-p. Yao, S.-l. Peng, and L.-d. Wu. A well-balanced time warp system on multi-core environments. In *PADS*, 2011.

[7] V. Gil-Costa, J. Lobos, A. Inostrosa and M. Marin. Capacity planning for search engines: An approach based on coloured Petri nets. In *ICATPN*, 2012.

[8] G. DAngelo and M. Bracuto. Distributed simulation of large-scale and detailed models. *IJSPM*, 5(2) 2009.

[9] B. Fitzpatrick. Distributed caching with memcached. *J. of Linux*, 2004:72–76, 2004.

[10] A. Freire, F. Cacheda, V. Formoso and V. Carneiro. Analysis of performance evaluation techniques for Large Scale Information Retrieval. In *LSDS-IR*, 2013.

[11] R. Fujimoto. Parallel discrete event simulation. *Comm. ACM*, 33(10):30–53, Oct. 1990.

[12] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *WWW*, 2009.

[13] R. Geist, J. Hicks, M. Smotherman, and J. Westall. Parallel simulation of petri nets on desktop pc hardware. In *WSC*, 2005.

[14] D. W. Glazer and C. Tropper. On process migration and load balancing in time warp. *TPDS*, 4(3) 1993.

[15] S. Jafer and G. A. Wainer. A performance evaluation of the conservative devs protocol in parallel simulation of devs-based models. In *SpringSim (TMS-DEVS)*, 2011.

[16] K. Jensen and L. Kristensen. *Coloured Petri Nets*. Springer-Verlag Berlin Heidelberg, 2009.

[17] M. Knoke and G. Hommel. Dealing with global guards in a distributed simulation of colored petri nets. In *DS-RT*, 2005.

[18] M. Knoke, F. Kühling, A. Zimmermann, and G. Hommel. Towards correct distributed simulation of high-level petri nets with fine-grained partitioning. In *ISPA*, 2004.

[19] J. Liu and R. Rong. Hierarchical composite synchronization. In *PADS*, 2012.

[20] Q. Liu and G. A. Wainer. Multicore acceleration of discrete event system specification systems. *Simulation*, 88(7) 2012.

[21] M. L. Loper and R. Fujimoto. Exploiting temporal uncertainty in process-oriented distributed simulations. In *WSC*, 2004.

[22] M. Marin. Time Warp on BSP. In *ESM*, 1998.

[23] L. Mellon and D. West. Architectural optimizations to advanced distributed simulation. In *WSC*, 1995.

[24] M. Marzolla. LibCppSim: A SIMULA-like, Portable Process-Oriented Simulation Library in C++. In *ESM*, 2004.

[25] D. M. Nicol: Discrete-Event Simulation in Performance Evaluation. Performance Evaluation 2000: 443-457.

[26] K.S. Panesar and R.M. Fujimoto. Adaptive flow control in Time Warp. In *PADS*, 1997.

[27] L. G. Valiant. A bridging model for multi-core computing. J. Comput. Syst. Sci. 1(77), 2011.

[28] L. G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8) 1990.

[29] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *PADS*, 2012.

[30] BSPonMPI. http://bsponmpi.sourceforge.net/

[31] MPI. http://www.mcs.anl.gov/research/projects/mpi/

[32] OpenMP. http://openmp.org/

[33] CPN Tools. http://cpntools.org/