Parallelization of Complex Event Processing on GPU

Patel Kuldip L.* Savalia Jay M.* Jaya Sreevalsan-Nair*

* *International Institute of Information Technology, Bangalore, India.*
{kuldipkumar.patel, jay.savalia}@iiitb.org, jnair@iiitb.ac.in

Abstract

Publish/subscribe is a paradigm used in real-time enterprise applications, such as job portals, where an event processing system is the key element. It allows applications to process the incoming event streams and applies relevant techniques in real-time for making efficient and quick decisions. Achieving scalability and high performance under excessive scale and load is a challenging problem. The most computationally intensive module in the event processing system is the processing and matching engine which handles the important task of connecting decoupled entities. We propose an algorithm and its parallel implementation to utilize the GPU for event matching algorithm which results in high throughput.

1. Introduction

A job portal consists of three significant components: a job provider, a job subscriber and the processing logic. The job provider posts a job with certain specific criteria, the job subscriber subscribes to the job-portal with a specific skill set and the processing logic runs a matching algorithm to match available jobs to available subscribers. Current implementations of such a system restrict the jobs posted to the job portal to confirm to a particular set of subscribers who fulfill the requirement for that job. The existing approach to building the processing logic is a sequential search of a qualification from a query for all registered job subscribers, and repeat the process for all required qualifications. This approach requires several disk accesses owing to high frequency of querying to a database, and hence results in low throughput. We present an alternative model which reduces the computation time in the processing logic module, thus resulting in higher throughput, in comparison to existing approaches.

Publish/subscribe is a powerful paradigm for information dissemination from publishers (event producers) to the subscribers (event consumers) in the real world [1]. A data event specifies values pertaining to a set of attributes associated with the event. Subscribers register their interests in future events by using a set of predicates over the event attributes. Upon receiving an event published by a publisher, the system matches the event to the subscriptions, which serve as filters, and deliver results to the event subscribers [2], [3].

A job portal is an event processing problem that is to some extent similar to publish/subscribe model. In the case of a job portal, the job provider acts as a publisher of events and the job subscriber is equivalent to a subscriber for that particular event. An ideal implementation of a job portal implies efficient and scalable implementation of its various modules, such as, subscription management, event matching, and event delivery to the set of job subscribers.

The graphics processing unit (GPU) is a multi-threaded processor containing hundreds of processing elements known as Scalar Processors (SPs). Eight SPs together form a group called the Streaming Multi-processors (SMs). These eight SPs execute in Single Instruction Multiple Thread (SIMT) fashion. Hence, all the SPs in an SM execute the same instruction at the same time [4]. As the memory access pattern is known in advance and there is massive data reuse in the algorithm, we can take advantage of the faster shared memory to hide latency of global memory access. A job portal application being heavily task-parallel, is an ideal application for parallelization using the GPU.

We had implemented a prototype of a distributed

publish/subscriber model for a job portal using the GPU. We analyz our work and report the priliminary results in the following sections.

2. System Design

In this section, we elaborate on the overview of a job portal system based on the publish/subscribe model. We briefly discuss each component of the system.

2.1. Event (Job)

A job in the job portal application is equivalent to an event in a publish/subscribe system and hence, can be treated as an event. We shall be using job and event interchangeably throughout our paper. A job, posted in the job portal by a job provider, specifies certain requirements for a potential match for that job. We can safely assume that the requirement parameters of any job are the descriptive attributes of the event. These parameters include educational qualifications, skills, experience, etc.. We express these parameters in bit values, that is, 1 or 0, which indicate the presence or absence of an attribute for a given job, respectively. Thus, a bit string representation of an event or a job takes into consideration all possible requirement parameters that is relevant to the job portal. Figure 1 shows the vector representation of a job, where each element corresponds to a particular requirement. As

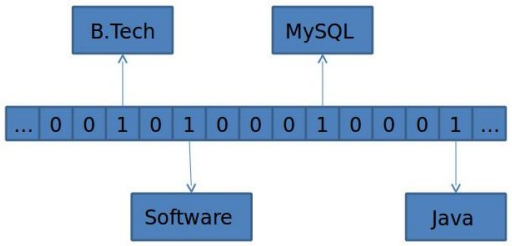


Figure 1: Vector representation of job or an event.

shown in Figure 1, this particular job or event has the requirements for a candidate holding a B.Tech. degree who is working in software development and maintenance, with proficiency in Java and MySQL. So in the event parameters rather than storing entire keyword, we make the keyword position fixed and indicate its presence or absence of the keyword using 1 or 0, respectively, thus representing an event using a bit-string representation. Figure 1 shows a simple representation of the parameter attributes used for a job. The disadvantage of such a representation is that the length of the representative bit-string increases in size exponentially with addition of jobs, and also the bit-string representations need not be unique for a job description. Since our implementation is a prototype, we do not address these issues in the scope of this work.

2.2. Subscriber (Job Subscriber)

Similar to an event, a subscriber can also be described by representing its skill set and qualifications in a bit-string, using 1 or 0 to indicate presence or absence of that skill set in the subscriber profile, respectively. For consistent representations, in our prototypical implementation, we assume that the describing keywords are same for the subscriber as well as the job. As shown in the example of a job in Figure 1, a subscriber with the skill set <B.Tech,Software,JAVA,MySQL> may be represented similarly.

2.3. Event-Subscriber Relationship

We use the notation b_{Ai} for bit-value, which can take values 0 or 1, for subscripts A indicating job ($A = J$) or subscriber ($A = S$), and i indicating the location in the bit-string representation. We define the set $B_{J1} = \{i | b_{Ji} = 1, \text{ for } B_J = b_{J1} b_{J2} \ldots b_{Jn}\}$ for a bit-string representation of a job B_J, with n parameters, where B_{J1} is the set of all locations of 1's in the bit-string B_J. Similarly for the subscriber represented by the bit-string B_S, we define the set $B_{S1} = \{i | b_{Si} = 1 \text{ for } B_S = b_{S1} b_{S2} \ldots b_{Sn}\}$.

There can be three cases of subscriber-job relationship, considering the bit positions containing 1's in the vector representations of a subscriber and of a job, respectively.

1) A subscriber S is "aptly qualified" for a job J, if the vector representation of subscriber exactly matches for all the bit positions that are 1's in that of the job, that is, $B_{J1} = B_{S1}$.
2) A subscriber S is "underqualified" for a job J, if the vector representation of subscriber does not match for all the bit positions that are 1's in that of the job, that is, $(B_{S1} \cap B_{J1}) \subset B_{J1}$.
3) A subscriber S is "overqualified" for a job J, if the vector representation of subscriber matches for all the bit positions that are 1's in that of the job as well as has additional 1's, that is, $B_{J1} \subset B_{S1}$.

Our system is designed to handle matches when a subscriber is either "aptly qualified" or "overqualified"

for a job. The information of the subscribers is stored in a database and we assume that all subscribers are available prior to the arrival of events.

2.4. Cluster

For scheduling, we had to find the set of subscribers that satisfies a set of criteria specific to the job. To do that we had to check the absence or presence of all the parameters required for every registered subscriber. This leads to a large number of comparisons and thus the matching process becomes highly computationally intensive. To alleviate this problem, we had used clustering. Once similar types of subscribers are clustered, we can easily reduce the computation by applying search to specific clusters which are relevant to the given job. Another acceptable advantage of clustering is reduction of data transfer between the GPU and the CPU, which the clustering methods achieve by removing subscribers which are outliers.

In our prototype, we use a very rudimentary clustering algorithm. We divided all subscribers based on their skill sets, and we clustered them if they contained the same set of skills. Thus we clustered the subscribers based on the 1's in the bit-string representation of the subscribers, for e.g. presence of same parameters for certain professions, e.g. medicine, law, engineering, etc. causes the subscribers to be naturally clustered. In effect, we have clustered the subscribers based on their profession or specialization, viz. engineers, doctors, lawyers, pharmacists. For e.g., for matching a job suitable for doctors (in the medical field), we eliminated a set of subscribers, who are not related to the medical field, thus reducing the number of potential subscribers we need to consider. We assume that in our job portal, the set of the subscribers is not dynamic and hence the overhead to create and update the clusters will be minimal.

3. Implementation

In this section, we explain the actual implementation of each component used for the event matching algorithm. Our system is divided into two parts: *Scheduler* and *EventMatchingAlgorithm*.

3.1. Scheduler

In a job portal, the job arrival rate is not predictable. We can assume that the arrival of different jobs are independent events. Hence, whenever a job is available, we first place it in the database where incoming jobs are stored. At a certain instant of time, the scheduler will start processing the database to find similar jobs in the context of its description parameters.

After finding similar events, the scheduler will dispatch them to the event matching algorithm along with the specific cluster of subscribers to initiate processing. Figure 2 shows an outline of the working of scheduler.

3.2. Event Matching Algorithm

Matching algorithm accepts two inputs.
Input−1: Similar jobs (event pool) which come from the scheduler.
Input−2: Cluster of relevant subscribers for given *Input-1*.

We can have two implementations of the event matching algorithm: (a) sequential, and (b) parallel.

3.2.1. Sequential Implementation. The sequential implementation of the matching algorithm can be done in two ways:

1) Per-event match for all subscribers.
2) Per-subscriber match for all events.

If there are m events and n subscribers, the sequential implementation using any of the two approaches, will have a time complexity O(n*m). Algorithm 1 shows pseudo code for first approach.

We could interchange the *for*-loops in algorithm 1 and obtain the algorithm for the second approach, i.e. per-subscriber match for all events.

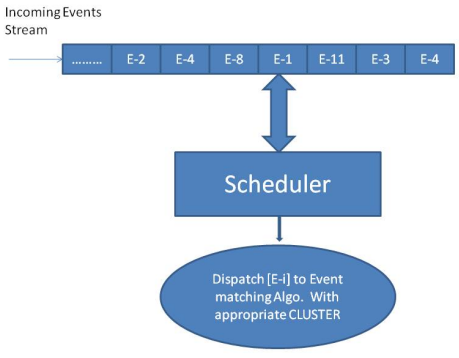


Figure 2: Outline of Scheduler.

```
for event from event pool do
    for subscriber from cluster do
        if Ismatch(event,subscriber) then
            eligible subscriber for event.
        end
        else
            ineligible subscriber for event.
        end
    end
end
```

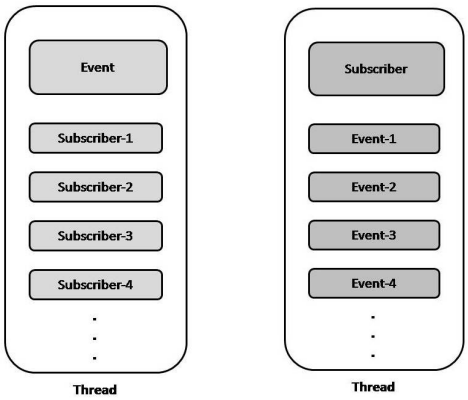
Algorithm 1: Per-event match all Subscribers.

3.2.2. Parallel Implementation. In the parallel implementation approach, matching algorithm can be implemented using one of the following models:

A. Thread per event.
B. Thread per subscriber.
C. Thread per event-subscriber pair.

We had implemented these models on the GPU using CUDA. We explain the implementation details of each of the models:

- **Model A: Thread per event**
  This model creates a thread per event, that is, if there are m events and n subscribers, then this model creates m threads. Each thread corresponds to one particular event and a list of subscribers (*Input-2*). The thread compares skill set of an event with the skill set of all the subscribers. This comparison results in a set of subscribers which satisfy the event criteria, eligible for that event. An abstract view of the composition of such a thread is as shown in Figure 3(a).
- **Model B: Thread per subscriber**
  As opposed to the event-centric model A, in model B we create threads per subscriber. Thus, if there are m events and n subscribers then this model creates n threads. Each thread has one particular subscriber and list of events. The thread compares the skill set of a subscriber to the skill set of all the events. This comparison results in set of events for which subscriber is eligible. An abstract view of the composition of such a thread is as shown in Figure 3(b).
- **Model C: Thread for each event-subscriber pair**
  For efficient utilization of GPU resources, large number of threads should be created and deployed. Hence we use a model which creates thread for each pairing of an event and a subscriber. If there are m events and n subscribers, then this model creates $(m*n)$ pairs, and hence, $(m*n)$ threads. Each thread compares the skill set of a particular event with that of a particular subscriber. Each comparison makes a decision for eligibility of subscriber for that event. An abstract view of the composition of such a thread is shown in Figure 4.



(a) (b)

Figure 3: Models for Parallel Implementation of Job Matching Algorithm: (a) Model-A: Thread-per-event Model, (b) Model-B: Thread-per-subscriber Model.

3.2.3. Comparison of the Models for Parallel Implementation. Parallel implementation of model-A and model-B create heavily loaded threads, that is every thread has a large number of computations to do. If $m \ll n$, given m events and n subscribers, model-A results in less number of threads which are heavily

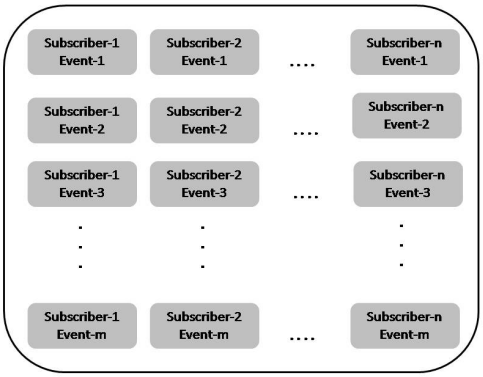


Figure 4: Model-C: Thread-per-(event-subscriber-pair) Model for Parallel Implementation of Job Matching Algorithm.

loaded. In such a case, all the GPU cores are not utilized fully and hence is an inefficient model for parallelization using the GPU. Model-B also suffers from a similar problem, when $m \gg n$.

To overcome the problem of under-utilization of GPU cores, we have introduced model-C which involves creation of several lightly-loaded threads, implying that every thread has less amount of work to do. Each thread has only one comparison to make, which decides the eligibility of a subscriber for the corresponding event. Hence producing an optimum number of lightly loaded aligned threads increases the utilization of the GPU cores. Additionally, this model is designed to use the shared memory as a buffer to hold the data and allow non-coalesced manipulation of data, but coalesced access when writing to global memory. Using the shared memory as a buffer ensures that the accesses to the global memory while writing continue to remain coalesced. The idea is that the memory controller performs efficiently if it accesses a series of memory locations and provide content to the threads of a half-warp, instead of exclusive accesses for each thread. If the threads in a block are accessing consecutive global memory locations, then all the accesses can be *coalesced*, that is, combined to a single request by the hardware. Coalescing accesses to global memory results in low latency. In model-C, a single pair of event-subscriber contributes to a thread and the result of the comparison performed in a thread is written back to the global memory again in a coalesced manner.

4. Results and Analysis

We implemented models A, B, and C, using NVIDIA GeForce GTX 460 processor which has 7 SMs. It has a global memory of 1 GB. Our preliminary results in Figure 5 show that the speedup achieved by the parallel execution of the matching algorithms on a GPU for different datasets. We obtained a speedup of 3 using model-C. The GPU starts to outperform the CPU when n blocks of large threads, where $n > 1$, process the data allowing the hardware to switch between the blocks to hide the latency of the access to the global memory. Figure 5 also shows that peak performance can be achieved by keeping the GPU fully utilized by using an optimum number of threads.

5. Conclusions and Future Work

We implemented both the sequential version of our job-matching algorithm on the CPU as well as its parallel version using the GPU. We have implemented the latter using the three proposed models of the matching algorithm. We have also recorded our preliminary analysis of the performance. The parallel implementation using GPU has a higher throughput when there are a reasonably large number of threads used. In the future, we plan to extend this algorithm to using multiple GPUs. We further propose to optimize the clustering operation in order to optimally reduce the number of comparisons, the amount of data transfer among GPU and CPU and in turn harness the capabilities of the GPUs better.

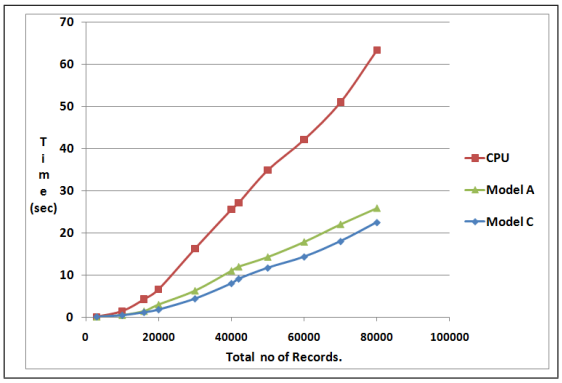


Figure 5: Speedup on GPU Using Parallel Implementation.

Acknowledgments

The authors would like to thank International Institute of Information Technology, Bangalore, for facilitating the research activities for this paper. One of the authors, Mr. Savalia Jay M., unfortunately passed away on October 1, 2011.

References

[1] Y. Zhu and Y. Hu, "Ferry: A P2P-Based Architecture for Content-Based Publish/Subscribe Services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, pp. 672–685, 2007.

[2] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/subscribe," *ACM Comput. Surv.*, vol. 35, pp. 114–131, June 2003.

[3] A. Adi, D. Botzer, G. Nechushtai, and G. Sharon, "Complex Event Processing for Financial Services," *IEEE Services Computing Workshops*, vol. 0, pp. 7–12, 2006.

[4] "NVIDIA CUDA Programming Guide." [Online] Available at: http://developer.nvidia.com/getting-started-parallel-computing, [Accessed Feb 16, 2011].