Design and Analysis of Parallel Content Matching Algorithm for Pub-Sub Systems using Different Parallel Paradigms

M. A. Shah[1], Dr. D. B. Kulkarni[2]

[1] Department of Computer Science and Engineering, Walchand College of Engineering, Sangli (MS) India

[2]Department of Information Technology, Walchand College of Engineering, Sangli (MS) India

Abstract: ***The key challenge in the performance of Pub-Sub system is the design of matching algorithm. The content matching takes place on each broker system along the path from publisher to subscriber in broker overlay network. Matching time is significant as compared with a network delay of message forwarding. In the case of content-based systems, matching is time-consuming task, whose performance affects the entire system. Efficient content-based event matching is considered as challenging research problem from past few years. All algorithms proposed earlier are inherently sequential and does not exploit parallel architecture which is easily available in current generation computers. This paper describes a new Pub-Sub content-based matching algorithm designed using principles of shared and distributed memory program running efficiently on multicore processor architecture. Hybrid parallel programming approach shows 4 times reduction in average matching time and an improved throughput of over 4000 events/s when using 32 processors which are almost double of events processed using only shared memory approach or only distributed memory approach. Paper also presents the result of the content matching algorithm using GPU.***

Keywords: Throughput, matching time, distributed memory architecture, Shared memory architecture, CUDA, GPU, Content-matching Algorithm

1. Introduction

Nowadays the majority of information is available on the World Wide Web. Besides system for searching, querying, retrieving information from the web, there is a need for systems being able to capture the dynamic aspects of the web information by notifying users of interesting events A tool that implement this functionality should have features of efficiency and scalability. Indeed, it should manage demands for millions of subscriptions to get matched with the events published. It should handle high rates of events and notify the interested users in short delay. For inter-object communication generally event-driven, or Notification-based, interaction pattern is used. This notification pattern is increasingly being used in a Web services context [11]. Another good example of Pub-Sub system is the set of auction sites on the internet e.g. eBay, Amazon, Yahoo. Every day a large number of items are put up for auction by each of those auction sites.

Over the last couple of years, another reliable Pub-Sub system called Wormhole is designed. The wormhole has become a critical part of Facebook's software infrastructure. At a high level, Wormhole propagates changes issued in one system to all systems that need to reflect those changes – within and across data centers. Low latency and efficiency are the prominent properties of Wormhole [20]. Figure 1 shows an overview of Pub-Sub system.

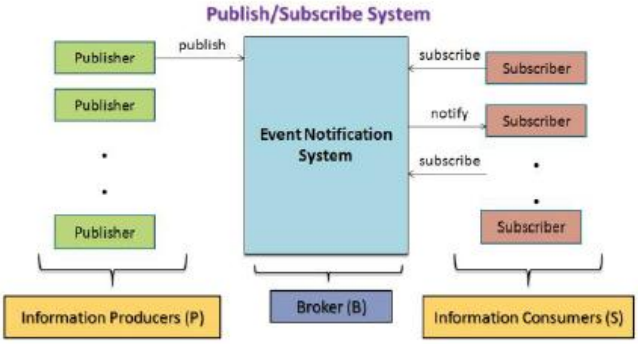

Figure 1: Overview of Pub-Sub System

Pub-Sub systems [6]; establishes a connection between publishers (producers) and subscribers (consumers) of events. Pub-Sub system is asynchronous in time and space. This way, publishers are decoupled from subscribers; they do not need to be aware of each other. Publishers submit events to the Pub-Sub system while Subscribers through subscriptions express their interest in the event. Pub-Sub system does the work of matching and notification of events to interested subscribers. The core functionality implemented by Pub-Sub system is matching. For large number of events and subscriptions matching component must work at attainable performance.

This work concentrates on making this Pub-Sub system efficient and scalable by leveraging commodity multicore processors and accelerators. The contributions of this work are as follows:-

1) Achieved reduction in matching time
2) Proposed a data distribution mechanism for subscription management & designed hybrid approach of parallelism to achieve high throughput in the matching process.
3) Behavioral analysis of parameters such as high throughput and low latency for varying size of processors and workload.

The remainder of the paper is organized as follows: Section 2 covers the survey related to traditional Pub-Sub systems, sequential matching algorithms and discuss work related to parallel high-performance event processing. Section 3 describes event and data model. Section 4 presents parallel programming data model and their features. Section 5 explains content matching algorithm on various parallel platforms. In section 6 extensive performance evaluations are reported.

2. Related Work

Let's review the related work on distributed content based Pub-Sub systems. This elaborates traditional Pub-Sub system as well as newly designed high-performance Pub-Sub systems.

2.1 General Pub-Sub Research

Most of the earlier work on scalable Pub-Sub has relied on networks of brokers, which are dedicated machines, each performing the whole range of operations that compose the content routing task viz.: (1) management of subscriptions from users and other brokers (2) filtering of incoming publications against stored subscriptions and dispatching to local interested subscribers and (3) filtering of incoming publications against routing tables for dispatching to other brokers. Brokers are typically organized in a broker overlay, with subscriptions and publications flowing between brokers according to its logical structure, typically a tree or a mesh. Well-known examples of broker-based Pub-Sub middleware are SIENA [8], Gryphon [9] and PADRES [10].

2.2 Event Processing Algorithms in Pub-Sub Systems

Two main categories of matching algorithms have been proposed: counting-based [13, 14, 15] and tree-based [16, 17, 18] approaches. These approaches can further be classified as either key-based, in which for each expression a set of predicates are chosen as identifier [12], or as non-key based [13,17,15]. Counting-based methods aim to minimize the number of predicate evaluations by constructing an inverted index over all unique predicates. The two most efficient counting-based algorithms are Propagation [14], a key-based method, and the k-index [15], a non-key-based method. Likewise, tree-based methods [17,18] are designed to reduce predicate evaluations and to recursively divide the search space by eliminating subscriptions on encountering unsatisfied predicates. The most prominent tree-based method, Gryphon, is a static, non-key based algorithm [16]. BE-Tree [2] is a novel tree-based approach, which also employs keys, that outperform existing work [13,14,15,16]. The latest advancement of counting-based algorithms is k-index [15], which gracefully scales to thousands of dimensions and supports equality predicates and non-equality predicates. k-index partitions subscriptions based on their number of predicates to prune subscriptions with too few matching predicates; however, k-index is static and does not support dynamic insertion and deletion. BE-Tree is distinguished from k-index is that BE-Tree is fully dynamic, naturally supports richer predicate operators (e.g. range operators), and adapts to workload changes. BE-Tree, however, is limited to attributes whose values are discrete and for which the range in discrete attribute values is pre-specified. So, BE-tree is unable to cope with real-valued attributes, string-valued attributes, and discrete-valued attributes with unknown range. Additionally, BE-tree [1] [2] employs a clustering policy that is ineffective when many subscriptions have a range predicate such as low $\leq a_i \leq$ high, where a_i is an attribute and the clustering criterion p that is used lies between low and high. Here p is the clustering technique used to collect the subscriptions into the same cluster. In this case, all such subscriptions fall into the same cluster and event processing is considerably slowed. PUBSUB [19], which is a heterogeneous system, offers a variety of data structures to keep track of the buckets in an attribute structure enabling the user to select data structures best suited for each attribute. Because of PUBSUB's heterogeneity in data structures for each attribute, PUBSUB permits all attribute data types.

2.3 High-performance event processing in Pub-Sub systems

In [7], the event matching algorithm proposed is parallelized leveraging chip multi-processors, increasing the throughput to over 1600 events/second with eight cores and reducing the processing latency by 74%. In [3] author has proposed CUDA based Content Matcher (CCM) on GPU to accelerate matching in content-based Pub-Sub systems. In [4] author has proposed high performance massively parallel architecture for content-based Pub-Sub system while [5] proposed multi-core message broker with Quality of Service support.

3. Events and Predicates

A subscription is a set of predicates. Each predicate consists of 3 characteristics [35]: An attribute name, a value and a relational operator (<, =, !=, >,) are the components of predicate. An event is attribute , value pair. An event's pair, say (<attribute name> x, <value> y), matches a subscription predicate (<attribute name> a, <value> b, <operation> c) only when x = a and y <operation c> b.

Sample event is defined as follows:
- string class=travel/airlines/offer;
- date starts = Jun;
- date expires = Aug;
- string origin = LA;
- string destination = AUS;
- string carrier = United

A Filter is defined as the conjunction of attribute constraints. Each attribute constraint has a name, a type, an operator, and a value. A constraint defines an elementary condition over an event or message. Sample filter is defined as follows:

- string class >*travel/airlines;
- date starts<Jul;
- date expires>Jul;
- string origin = LA;
- string destination = AUS

This is a valid filter matching the event of the earlier example. So a filter matches an event if all the attribute constraints in a filter are satisfied by the attributes in an event.

A predicate is defined as a disjunction of filters. A Predicate matches an event if, at least one of its filter matches an event.

4. Programming Models

Programming paradigm which is best suited for the underlying computer architecture is to be chosen correctly. Here in this section we describe various parallel programming paradigms with their features

4.1 Programming Shared Address Space (OpenMP) Platform

OpenMP [21][22] is an Application Programming Interface for implementing explicit shared-memory parallelism. API provides an incremental path for developing parallel code from existing serial code. OpenMP offers programmers a simple and flexible interface for parallel application development on different platforms which ranges from a desktop computer to supercomputer. This API is designed for multicore shared memory machine. Parallelism in OpenMP is accomplished exclusively through the use of threads. A thread is a lightweight process and smallest unit of execution which is scheduled by the operating system. Threads exist within the resources of a single process. Typically there is one to one association between thread and process but the actual use of thread is decided by the application. In this programming, the model programmer has full control over parallelization. Parallelism is as simple as inserting compiler directive in the sequential program and as complex as insertion of subroutines to form multiple levels of parallelism and locks.

Fork -join model of parallel execution is used by OpenMP. Every OpenMP program begins with master thread and until the first parallel region construct is encountered, the master thread executes sequentially. The master node creates a pool of threads to achieve parallelism. Block of the code which is enclosed by the parallel region constructs is executed in parallel by using a number of threads. After compilation of execution of statements in a parallel region, all threads synchronize and eventually terminate. Then again master thread takes the control of the program. Compiler directives which can be embedded in C / C++ and FORTRAN are used to achieve OpenMP parallelism. The threads involved in the execution of parallel region can be altered dynamically at runtime to promote efficient resource utilization.

The distinct component comprises OpenMP API are compiler directives, runtime library routines, and environment variables. Compiler directives are used for various purposes like distribution of loop iterations between threads, a block of code to be divided among threads and synchronization of works among threads. Runtime library routines are used for various purposes like setting and querying a number of threads, getting thread unique identifier and setting and querying nested parallelism etc. Environmental variable is used for controlling the execution of parallel code at runtime. The environmental variable set the number of threads, binds threads to processors and also enable/disable dynamic threads. In our work, we have used this construct extensively to achieve parallelism.

4.2 Programming Distributed Address Space (MPI) Platform

The message-passing programming paradigm is one of the oldest and most widely used approaches for programming parallel computers. Two key attributes that characterize the message passing programming paradigm are partitioned address space and support for explicit parallelization. Message-passing programs are often written using asynchronous or loosely synchronous paradigms. In the asynchronous paradigm, all concurrent tasks execute asynchronously. This makes it possible to implement any parallel algorithm. In loosely synchronous programs tasks or a subset of tasks synchronize to perform interactions. However, between these interactions, tasks execute completely asynchronously. Most message-passing programs are written using the single program multiple data approach (SPMD).

4.3 Programming Hybrid Platform

To exploit parallelism beyond a single level, MPI and OpenMP programming models can be combined. Domain decomposition is the key to obtain coarse grain parallelism and we can achieve fine grain parallelism at loop level with the help of threads.

4.4 GPU Programming with CUDA

Ten years ago performance of single core CPU has essentially stagnated and a solution devised was exploiting Multi-core to increase parallelism. An application which works on Big data or scientific simulation require increased performance so the question raised was are there faster alternative to CPU?

The rise of GPU in 1990's for graphics processing games and visualization, solved this problem at same extent. CPU has always been slow earlier and GPU was only used for graphics processing. GPU benefited from Moore's law. GPU architecture was evolved from hardwired logic which has fixed-function to flexible programmable ALU's. GPU become fully programmable in 2006, NVIDIA invented parallel programming model and parallel computing platform named CUDA, in Nov 2006. Computing performance is dramatically increased by utilizing the power of Graphic Processing Unit (GPU). General purpose programming is possible with the help of this programming model. Nowadays GPUs are widely deployed as accelerators. To interact with CUDA compliant device different languages can be used. We used CUDA C [23], which is explicitly devoted to programming GPUs. Five key characteristics are described below on which CUDA programming model is based.

Thread Group Hierarchical Organization

In parallel programming, the problem is partitioned into different sub-problems which can be solved independently, by blocks of threads in a concurrent fashion. Each sub-problem is further decomposed into finer pieces which can be solved in parallel by all threads within a block. This hierarchical decomposition helps the algorithm to scale across an available number of cores.

Shared Memories

During execution of the thread, it may access data from multiple memory spaces. Each thread is having access to the private local memory where automatic variables are stored. Each block has a shared memory accessible to all threads residing in the same block. Finally, all thread may access same global memory.

Barrier Synchronization

There is no need to offer synchronization between thread blocks as they are allowed to execute independently from each other. On other hand, threads within the same block synchronize to execute and thus their memory access is coordinated. Through barriers, thread synchronization is achieved in CUDA.

Separation of Device and Host

GPU act as a coprocessor of a host (the CPU) running a C / C++ program according to CUDA programming model CUDA threads gets executed on separate device (the GPU).

Two separate memory spaces are maintained by host and device. Before execution, data should be copied from the memory of host to device memory allocated at the start of execution. Device performs the execution using thread and results are copied back to the memory of host and device memory is de-allocated.

Kernels

A single flow of execution for multiple threads is decided by a special function called Kernel. When calling kernel function, the programmer specifies the number of block and number of threads within each block that must execute it. CUDA runtime provides two special variables ThreadID and BlockID which are accessible inside the kernel and them together uniquely identify individual thread among those who execute the kernel.

5. Matching Algorithm

The matching algorithms developed for distributed memory architecture is presented here with the parallel matching engine that makes use of cores as well as threads.

5.1 MPI Content-based Matching Approach

The MPI Content- based matching algorithm consists of three phases: 1) Decomposition phase 2) Matching phase 3) Reduction phase. Subscriptions are divided equally among a number of processes. Events are available with every process, they are kept globally. Every process calculates the matched subscriptions for given block of events using constraint evaluation and counting algorithm, and send to the root process. Thus every process works parallel towards the building of solution. As shown in Figure 2, in Output decomposition method, calculation of output is divided into a number of processes.

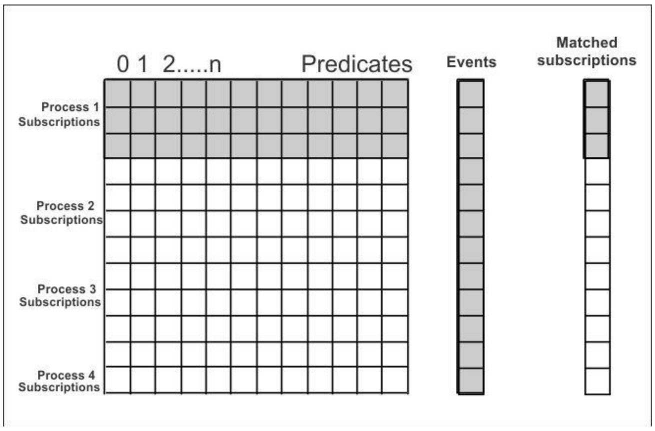

Figure 2: Output Decomposition Algorithm for Matching using MPI

```
Begin
Input: - Subscriptions, Events, Number of Processes
Output:: - Matched Subscriptions
1. Dynamically divide the subscription file among a number of processes taking part into the computation.
2. Place the events into global memory.
3. For each process do in parallel
    For each event do
        For each predicate in event do
            Compare with predicates in subscriptions
            If all predicates in the events matched with all predicates in subscription
    Count=count+1
End if
End for
End for
End for
Return Count (matched subscriptions) to root process
Output the Total matched subscriptions by root process.
End.
```

5.2 Hybrid (MPI + OpenMP based) Content-based Matching Approach

The matching algorithm is based on the two-phase algorithm presented in [7]. The algorithm works in two phases. Incoming messages are evaluated against predicates in the subscription in first phase. First phase named as H phase generates intermediate results. Subscriptions are traversed and evaluated in next phase called C phase. In C phase we evaluate formed clusters according to a number of predicates in the event. For this clusters are formed based on the number of predicates in the subscriptions. This increases the pre-processing time but reduces the matching time. Figure 3 describes the content matching engine to be executed by every process.

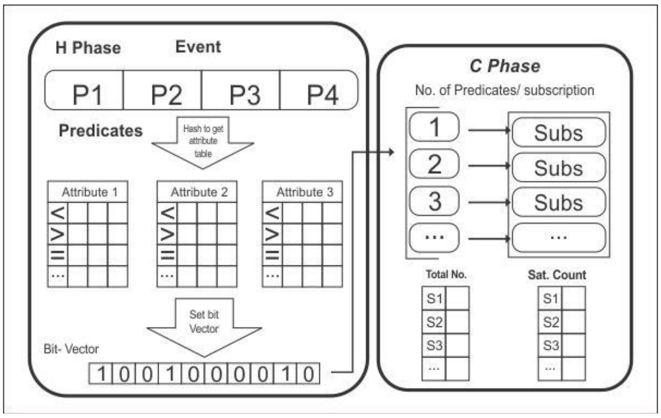

Figure 3: Matching Engine

In hybrid approach again, decomposition of subscriptions takes place according to a number of processes. Each process performs the matching function asynchronously. Here a block of events is allocated to every process. Adding one more level of parallelism, events within the pool are further distributed to threads allocated to processes and each thread matches event with subscriptions. Figure 4 explains the working of a hybrid approach.

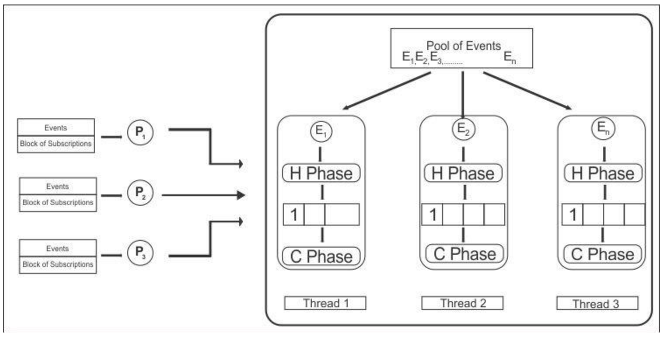

Figure 4: Hybrid Approach of Parallelism

5.3 Matching Algorithm using GPU

We have also implemented CUDA Content Matching algorithm presented in [3] which run efficiently on GPU. For this, we created data structures as mentioned in [3].It is observed that CCM performs well as compared to sequential matching algorithm and achieves good speedup.

All the tests were executed on the machine with the following configuration.

1) NVIDIA Tesla C1060
2) 4GB Global Memory Each
3) 30 SMs
4) Max. 512 threads i.e. 16 warps per SMP
5) (32 threads = 1 warp)
6) 8 Cores in Each and so 8 active blocks in each SM
7) 1024 threads i.e. approx. 30 warps can be active simultaneously

6. Experimental Results

The performance of the content matching algorithm is evaluated based on the scalability, matching time and throughput. Specifically, The gains achieved in both throughput and average matching time are shown, as the matching engine scales from a sequential system of one processor to a fully parallel system of thirty-two processors. The algorithms designed using shared memory; distributed memory and hybrid memory model are compared.

Experiments were run on a machine HP Proliant DI 785 65 servers with 8 CPU quad core AMD processor. Secondary storage is 584 GB and each core has 2 GB RAM.

To generate subscriptions and events workload generator was used. Both subscriptions and events were input to the system in a single batch (first subscriptions were loaded followed by events). Table 1 specifies the workload

Table 1: Workload Specification

Parameter	Subscription Workload
Number of subscriptions	100000-500000
Average Predicates per Subscription	18
Subscription Predicate Value Range	1-25
Number of Events	100000
Average Attribute per Event	50
Event Attribute Value Range	1-25
Number of Distinct predicates	1000
Number of Distinct attributes	100

6.1 Scalability

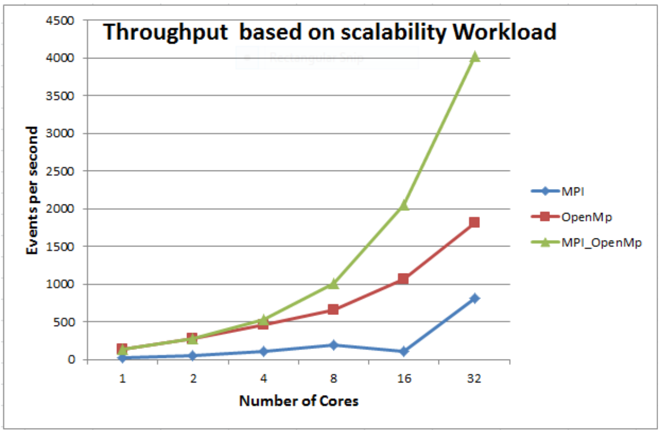

Figure 5: Throughput Based on Scalability Workload

Figure 5 shows the number of events processed per second using shared memory, distributed memory, and hybrid approach. All above approaches are implemented by using parallel programming APIs OpenMP, MPI, and MPI + OpenMP respectively. As the graph shows, increasing the number of processes from 1 to 32 results in near linear increase in throughput. This is not true for MPI approach because it does the sequential matching, which takes more time and so less throughput. Near double increase in throughput is observed for a hybrid approach. This is obvious because subscriptions get divided among the processes, and also threads works in parallel to match the events.

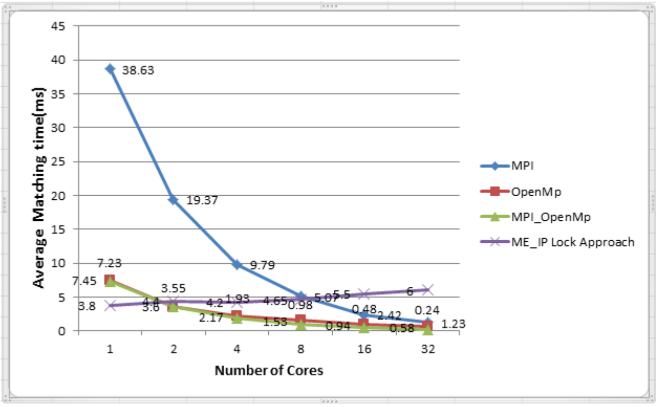

Figure 6: Average Matching Time for Parallel Approaches

Figure 6 shows the average matching time of the single event as the number of cores increase for all the three implementations. The Linear reduction is observed in matching time for OpenMP and MPI + OpenMP implementation. Compared to lock based ME_IP approach presented in [7] we get 4 times reduction in matching time for shared as well as a hybrid matching approach. Developed hybrid matching algorithm shows significant performance improvement in average matching time

6.2 Speedup and Efficiency

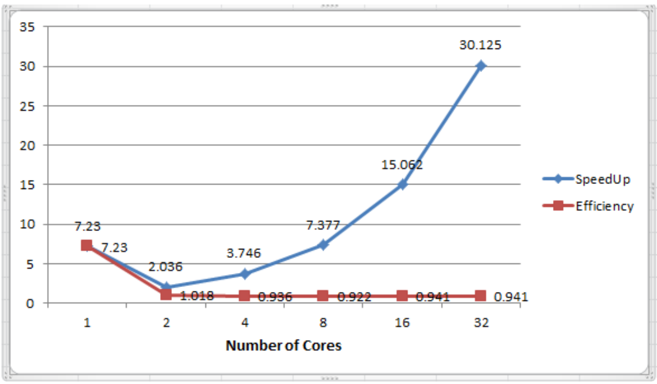

Figure 7: Speedup and Efficiency of MPI_OpenMP Architecture

Figure 7 shows a linear increase in a speedup for approaches of OpenMP and hybrid. Efficiency observed is approximately 1 for 2 to 32 processors. It has been observed that efficiency remains constant as we increase the number of processors. Hence we claim that algorithm is efficient and scalable. The formula for calculating speedup and efficiency is described here.

- Speedup (S) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p identical processing elements.
- Efficiency(E) is a measure of the fraction of time for which a processing element is usefully employed
- Mathematically, it is given by
- E=S/p

Result Analysis for Matching Algorithm using GPU

The dataset used for these experimentations is as mentioned in [3].

The following Figure 8 shows how performance changes with the average number of attribute inside events. The algorithm shows higher matching times with a higher number of attributes. It has become possible because CUDA content matching (CCM) algorithm processes all of the attributes in the events parallel. Available GPU cores are fully exploited by CUDA content matching algorithm. We have observed increased speedup of CCM over its sequential counterpart.

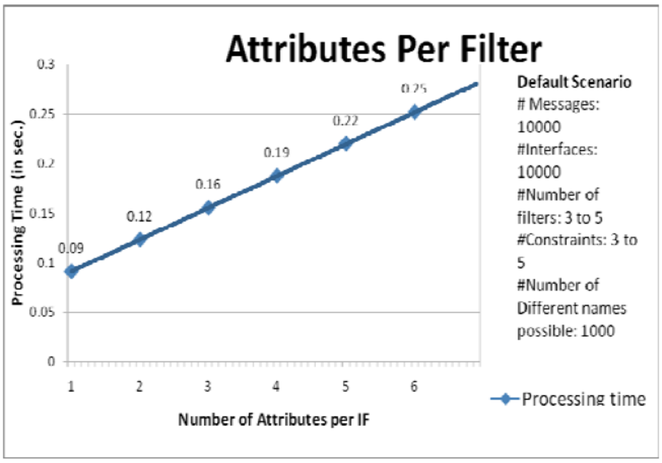

Figure 8: Number of Attributes Vs. Processing Time

The following Figure 9 shows how performance changes with the number of filters per interface. Increasing such number also increases the overall number of constraints, and thus the complexity of matching.

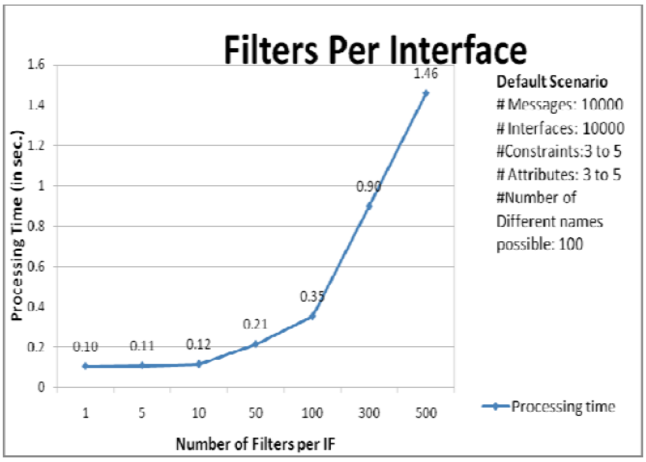

Figure 9: Number of Filters Per Interface Vs. Processing Time

7. Conclusion

This paper presented three parallel approaches to reduce the matching time of a single event and to increase throughput. The result shows that with 32 processors, throughput increased from 250 to over 4000 events/second for a hybrid approach. Matching time is also reduced from 9ms to 0.4 ms. Good speedup is observed & algorithm scales well up to 32 processes. It is observed that matching problem is relatively easy to parallelize. Programming CUDA is (relatively) easy while attaining good performance is hard. Memory accesses and transfers tend to dominate over processing cost and must be carefully managed.

References

[1] Mohammad Sadoghi, Hans-Arno Jacobsen.Analysis and Optimization for Boolean Expression Indexing. ACM Transactions on Database Systems, Vol. 38, No. 2, Article 8, Publication date: June 2013.

[2] M. Sadoghi and H.-A.Jacobsen.BE-Tree an Index Structure to Efficiently Match Boolean Expressions over High dimensional Discrete Space.SIGMOD 2011.

[3] Alessandro Margara, GianpaoloCugol. High-performance content-based matching using GPUs. DEBS '11 Proceedings of the 5th ACM international conference on Distributed eventbased system New York, NY, USA ACM 2011

[4] Raphaël Barazzutti,1 Pascal Felber.StreamHub: A Massively Parallel Architecture for High-Performance Content-Based Publish/Subscribe. DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.

[5] Zhaoran Wang, Xiaotao Chang. Pub/Sub on Stream: A Multi-Core Based Message Broker with QoS Support. DEBS 2012, July 16–20, 2012, Berlin, Germany July 2012.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M.Kermarrec.The many faces of publish/subscribe. ACM Computer Survey, 35:114–131, 2003.

[7] AmerFarroukh , Elias Ferzli , NaweedTajuddin, Hansarno Jacobsen.Parallel event processing for content-Based publish/subscribe systems, in DEBS '09 Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, New York, USA 2009

[8] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf.Design and evaluation of a wide-area event notification service.ACM TCS, 2001.

[9] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra.Matching events in a content-based subscription system.In PODC, 1999.and evaluation of a wide-area event notification service. ACM TCS, 2001.

[10] H.-A. Jacobsen, A. Cheung, G. Lia, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh.The PADRES publish/subscribe system. Handbook of Research on Adv. Dist. Event-Based Sys., Pub./Sub. and Message Filtering Tech., 2009.

[11] J2EE Web Services on BEA Web Logic by Subbarao

[12] G. Cugola and G. Picco.REDS: A Reconfigurable Dispatching System. SEM, pages 9—16, Portland, 2006. ACM Press

[13] T. Yan and H. Garcia-Molina.Index structures for selective dissemination of information under the Boolean model. ACM TODS'94.

[14] F. Fabret, H.-A.Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha.Filtering algorithms and implementation for fast pub/sub systems.SIGMOD'01

[15] S. Whang, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, R. Yerneni, and H. Garcia-Molina. Indexing Boolean expressions.In VLDB'09.

[16] K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra.Matching events in a content-based subscription system.In PODC'99

[17] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. ICSE'01.

[18] G. Li, S. Hou, and H.-A. Jacobsen.A unified approach to routing, covering and merging in [publish/subscribe systems based on modified binary decision diagrams. ICDCS'03.

[19] SartajSahni. PUBSUB: An Efficient Publish/Subscribe System. IEEE Transactions on Computers, , no. 1, pp. 1, PrePrintsPrePrints, doi:10.1109/TC.2014.2315636

[20] Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services Yogeshwer Sharma, Philippe Ajoux, PetcheanAng,

[21] An Introduction to Parallel Programming with OpenMP by Alina Kiessling.

[22] https://computing.llnl.gov/tutorials/openMP/

[23] Introduction to CUDA C by an Jose Convention Center, September 20, 2010.