High Performance Publish-Subscribe Matching Using Parallel Hardware

Alessandro Margara and Gianpaolo Cugola

Abstract—Matching incoming event notifications against received subscriptions is a fundamental part of every publish-subscribe infrastructure. In the case of content-based systems this is a fairly complex and time consuming task, whose performance impacts that of the entire system. In the past, several algorithms have been proposed for efficient content-based event matching. While they differ in most aspects, they have in common the fact of being conceived to run on conventional, sequential hardware. On the other hand, parallel hardware is becoming available off-the-shelf: the number of cores inside CPUs is constantly increasing, and CUDA makes it possible to access the power of GPU hardware for general purpose computing.
In this paper, we describe a new publish-subscribe content-based matching algorithm designed to run efficiently both on multicore CPUs and CUDA GPUs. A detailed comparison with two state of the art sequential matching algorithms demonstrates how the use of parallel hardware can bring impressive speedups in content-based matching. At the same time, our analysis identifies the characteristic aspects of multicore and CUDA programming that mostly impact performance.

Index Terms—C.1.2 Multiple Data Stream Architectures (Multiprocessors), C.2.4 Distributed Systems, C.4 Performance of Systems, D.1.3 Concurrent Programming [Parallel Programming], H.3.4 Systems and Software [Distributed Systems, Performance evaluation (efficiency and effectiveness)]

1 Introduction

Most distributed applications involve some form of event-based interaction, often implemented using a *publish-subscribe infrastructure* that enables distributed components to *subscribe* to the *event notifications* (or simply "events") they are interested to receive, and to *publish* those they want to spread around.

The core functionality realized by a publish-subscribe infrastructure is *matching* (sometimes also referred to as "forwarding"), i.e., the action of filtering each incoming event notification e against the received subscriptions to decide the components interested in e. This is a non trivial activity, especially for *content-based* systems, whose subscriptions filter events based on their content [15]. In such cases, the matching component may easily become the bottleneck of the system. On the other hand, several scenarios depend on the performance of the publish-subscribe infrastructure. For example, in financial applications for high-frequency trading [19], a faster processing of incoming event notifications may produce a significant advantage over competitors. Similarly, in intrusion detection systems [22], the ability to timely process the huge number of events that results from monitoring a large network is fundamental to detect possible attacks before they could compromise the system.

This aspect has been clearly identified in the past and several algorithms have been proposed for efficient content-based matching [2], [5], [10], [16], [27]. Despite their differences, they have a key aspect in common: they were all designed to run on conventional, sequential hardware. If this was reasonable years ago, when parallel hardware was the exception, today this is no more the case. Modern CPUs integrate multiple cores and more will be available in the immediate future. Moreover, modern Graphical Processing Units (GPUs) integrate hundreds of cores, suitable for general-purpose (not only graphic) processing.

Unfortunately, moving from a sequential to a parallel architecture is not easy. Often, algorithms have to be redesigned from the ground to maximize the operations that can be performed in parallel, and consequently to fully leverage the processing power offered by the platform. This is specially true for GPUs, whose cores can be used simultaneously only to perform data parallel computations.

Moving from these premises, we developed *Parallel Content Matching (PCM)*, an algorithm explicitly designed to leverage off-the-shelf parallel hardware (i.e., multicore CPUs and GPUs) to perform publish-subscribe content-based matching. We present two implementations of this algorithm: one for multicore CPUs using OpenMP [14], the other for GPUs that implement the CUDA architecture. We study their performance comparing them against SFF [10], the matching component of Siena [7], and BETree [27], a novel, high-performance matching algorithm. This analysis demonstrates how the use of parallel hardware can bring impressive speedups in content-based matching. At the same time, by carefully analyzing how PCM performs under different workloads, we also identify the characteristic aspects of multicore

The authors are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, P.zza L. da Vinci, 32, 20133 Milano, Italy. E-mail: margara,cugola@elet.polimi.it.

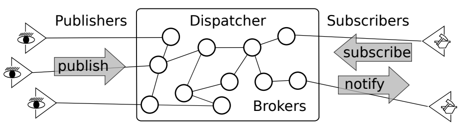


Fig. 1. A typical publish-subscribe infrastructure

CPUs and GPUs that mostly impact performance.

The remainder of the paper is organized as follow: Section 2 introduces the event model we adopt. Section 3 describes PCM and its implementation in OpenMP and CUDA. The performance of these implementations in comparison with SFF and BETree is discussed in Section 4, while Section 5 presents related work, and Section 6 provides some conclusive remarks. Two additional appendixes provide the required background on OpenMP and CUDA together with additional tests.

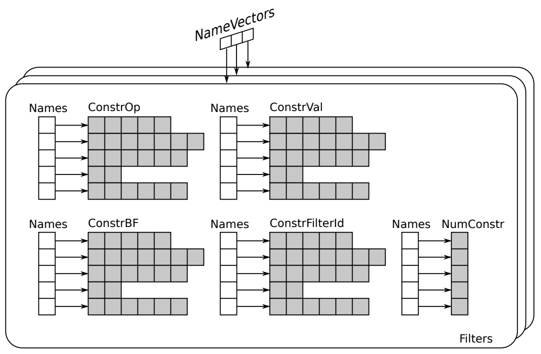
2 Events and Predicates

To be as general as possible we assume a data model which is very common among event-based systems [10]. We represent an *event notification*, or simply *event*, as a set of *attributes*, i.e., $\langle name, value\rangle$ pairs. Values are typed and we consider both numbers and strings. As an example, $e_1 = [\langle area, \text{“}area1\text{”}\rangle, \langle temp, 25\rangle, \langle wind, 15\rangle]$ is an event that an environmental monitoring component could publish to notify about the current temperature and wind speed in the area it monitors. The interests of components are modeled through *predicates*, each being a disjunction of *filters*, which, in turn, are conjunctions of elementary *constraints* on the values of single attributes. As an example, $f_1 = (area = \text{“}area1\text{”} \wedge temp > 30)$ is a filter composed of two constraints, while $p_1 = [(area = \text{“}area1\text{”} \wedge temp > 30) \vee (area = \text{“}area2\text{”} \wedge wind > 20)]$ is a predicate composed of two filters. A filter f *matches* an event e if all constraints in f are satisfied by the attributes of e. Similarly, a predicate matches e if at least one of its filters matches e. In the examples above p_1 matches e_1.

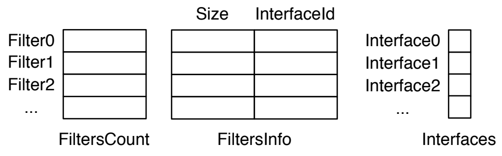
The problem of content-based matching we address here can be stated as follow: given an event e and a set of *interfaces*, each one exposing a predicate, find the interfaces relevant for e, i.e., those that expose a predicate matching e. In a centralized publish-subscribe infrastructure, it is the *dispatcher* that implements this function, by collecting predicates that express the interests of subscribers (each one connected to a different "interface") and forwarding incoming event notifications on the basis of such interests. In a distributed publish-subscribe infrastructure the dispatcher is realized as a network of *brokers*, which implement the content-based matching function above to forward events to their neighbors (other brokers or subscribers).

3 The PCM Algorithm

This section describes our PCM algorithm and its implementations in OpenMP (*OCM*) and CUDA (*CCM*).



(a) Filters and Constraints Tables



(b) Filters and Interfaces Tables

Fig. 2. Data structures

The key goal in designing PCM was to maximize the amount of data parallel computations, minimizing the interactions among threads. This is obtained by organizing processing in three phases: a *filter selection* phase, a *constraint selection* phase, and a *constraint evaluation and counting* phase.

The first phase partitions the set of filters based on their attributes' names: each attribute name is mapped into a bit of a (small) bit vector, called `NameVector`. Filters having the same `NameVector` become part of the same partition. When an event e enters the engine, we compute its `NameVector` NV_e and use it to retrieve the filter partitions whose `NameVector` is included into NV_e. The remaining filters have no chance to match e, as their constraint names are not part of e's attribute names.

In the second phase, PCM selects, for each attribute a in e, the set of constraints (part of the filters selected by the previous phase) having the same name as a.

The selected constraints are evaluated in the third phase, using the value of a. In particular, when a constraint c is satisfied, we increase the counter associated to the filter f belongs to. A filter f matches an event when all its constraints are satisfied and so does the predicate p it belongs to. When this happens, the event can be forwarded to the interface exposing p.

3.1 Data Structures

Fig. 2 shows the data structures we create and use during processing. As shown in Fig. 2(a), PCM aggregates filters having the same `NameVector` into a `Filters` table, with several of such tables indexed by `NameVector` values. Each `Filters` table organizes the constraints of the included filters into 5 data structures (in gray in Fig. 2): `ConstrOp`, `ConstrVal`, `ConstrFilterId`, `ConstrBF`, and `NumConstr`. We collectively refer to them as the `Constraints` tables.

They are organized by constraint names: each row is implemented as an array, storing information on constraints having the same name into contiguous memory regions. Each table is indexed using an STL map: given a name n, PCM can efficiently get the pointer to the first element of the array that includes information on constraints with name n.

In particular, for each constraint c belonging to a filter f, `ConstrOp` and `ConstrVal` store the operator and value of c; `ConstrFilterId` stores the id of f; `ConstrBF` contains a Bloom filter that encodes in a 64 bit integer the set of constraint names appearing in f. Finally, `NumConstr` stores the number of constraints having name n. In this organization, if different filters share a common constraint c we duplicate c. This simplifies memory layout and minimizes the size of each element of table `ConstrFilterId`, two key aspects for hardware meant to perform data parallel computations, which compensate the additional work required to evaluate duplicate constraints.

Similar considerations motivate the choice of separately storing information about each constraint c (i.e., operand, value, Bloom filter, and filter id). Indeed, this layout maximizes the throughput of memory accesses performed by the group of threads that evaluate constraints in parallel. This is specially true for GPUs, which may leverage their large data-path toward memory, but it is also true for multi-core CPUs.

Fig. 2(b) shows the additional data structures containing global information about filters and interfaces. Array `FiltersCount` stores the number of constraints currently satisfied by each filter f, while `FiltersInfo` contains static information: the number of constraints part of f (`Size`) and the id of the interface f belongs to (`InterfaceId`). Finally, `Interfaces` is an array of bytes, one for each interface, which, at the end of processing, contains a 1 at position x if the processed event must be forwarded through interface x. Both `FiltersCount` and `Interfaces` are cleared before processing each event and they are updated (in parallel) during processing.

In the OpenMP version of PCM, all data structures are stored in the main (CPU) memory. In the CUDA version, they are permanently stored into the GPU memory; a choice that minimizes the need for CPU-to-GPU communication during event processing. Only the maps that associate a name (or a `NameVector`) to the corresponding structures, as shown in Fig. 2(a), are stored on the main memory and accessed by the CPU during processing.

3.2 Implementing PCM in OpenMP (OCM)

Using OpenMP to parallelize the PCM algorithm is rather easy. When an event e enters the engine, OCM computes its `NameVector` NV_e and uses it to determine the `Filters` tables whose `NameVector` matches NV_e (discarding the others). These tables are evaluated in parallel using a *parallel for* loop.

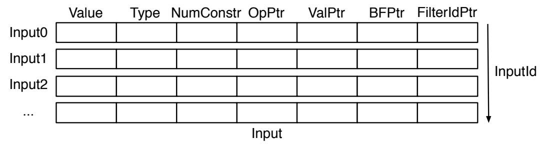


Fig. 3. Input data

For each attribute a in e, OCM uses the name of a to determine the row of each table that includes the constraints relevant for a. These constraints are evaluated sequentially. For each constraint c, OCM first compares the 64 bit Bloom filter that encodes the names in e with the Bloom filter associated to c. This is an additional, quick check (a bit-wise and) that allows to discard a lot of constraints, i.e., those belonging to filters that have no chance to be satisfied as they include at least one constraint that does not find a corresponding attribute in e. Notice that this check, as the preliminary filtering based on the `NameVector` values, may generate false positives, which means that in some cases we evaluate constraints that are part of filters that actually have no chance to be satisfied. Fortunately, this happens rarely (less then 1% of the times using 64 bit integers as Bloom filters) and it does not impact the correctness of results.

When a constraint c is satisfied, OCM has to update the `FiltersCount` for the filter c belongs to (retrieved using the `ConstrFilterId` table). Since a filter f cannot be part of two different `Filters` tables, OCM may update the `FiltersCount` structure without synchronizing, being sure that each thread processing a different `Filters` table will access a different element of such structure.

After updating `FiltersCount`, each thread checks whether it became equal to the `Size` of the filter, as stored in Table `FiltersInfo`. In that case, it sets to one the corresponding element in Array `Interfaces` (retrieved using the `InterfaceId` field). This array represents the result of our algorithm: the set of interfaces that match e. It is transferred to the caller at the end of processing and reset, together with the `FiltersCount` structure, before processing the next event.

3.3 Implementing PCM in CUDA (CCM)

The implementation of PCM on CUDA GPUs deserves more discussion. As for OpenMP, when an event e enters the engine, it is the CPU that computes the `NameVector` (NV_e) and the Bloom filter (called `InputBF`) of e. It then uses NV_e to isolate the relevant `Filters` tables (see Fig. 2(a)), and the names of attributes in e to determine the rows of such tables (i.e., the constraints) relevant for e.

As a result of this first phase, the CPU builds table `Input` shown in Fig. 3. It includes one line for each `Filters` table t identified through NV_e and for each attribute a in e. Each line stores the `Value` of a,

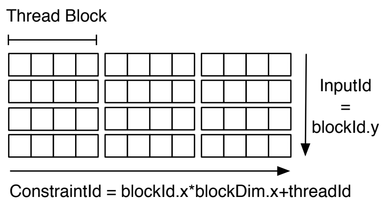


Fig. 4. Organization of blocks and threads

its `Type`, the number of constraints in t having the same name as a (`NumConstr`), and the pointers (in the GPU memory) to the rows of tables in t that are relevant for a. This information is transferred to the GPU together with the Bloom filter `InputBF`. Notice that this information is not modified by CCM during processing. Accordingly, both `InputBF` and the entire `Input` structure are stored into a specific region of the GPU's global memory called *constant memory*, which is cached for fast read-only access.

After building these structures and transferring them on the GPU, CCM launches a kernel (i.e., the CUDA function executed on the GPU), which uses thousands of GPU threads to evaluate constraints in parallel. Each thread evaluates a single attribute a of e against a single constraint c. After that, the results of the computation (i.e., the `Interfaces` array) has to be copied back to the CPU memory and the `FiltersCount` and `Interfaces` structures have to be reset for the next event. We will come back to these steps later in this section, while here we focus on the main CCM kernel.

In CUDA, threads are organized into blocks. At kernel launch time the developer must specify the number of blocks executing the kernel and the number of threads composing each block. Both numbers can be in one, two, or three dimensions (see Appendix A for further details on CUDA). CCM organizes threads inside a block over a single dimension (x axis) while blocks have two dimensions. The y axis is mapped to event attributes, i.e., to rows of table `Input` in Fig. 3, while the x axis is mapped to set of constraints. Indeed, the number of constraints with the same name may exceed the maximum number of threads per block, so CCM must allocate multiple blocks along the x axis. Fig. 4 shows an example of such organization in which each block is composed of 4 threads (in real cases we have 256 or 512 threads per block).

The pseudo-code of the CCM kernel is presented in Algorithm 1. At the first two lines, each thread determines its x and y coordinates in the bi-dimensional space presented above, using the values of the blockId and threadId variables, initialized by the CUDA runtime. Since different rows of the tables encoding constraints may have different lengths, we instantiate the number of blocks (and consequently the number of threads) to cover the longest among them. Accordingly, in most cases we have too many threads. We check this possibility at line 3, immediately stopping unrequired threads. This is a common practice in GPU programming, e.g., see [28]. We will analyze its implication on performance in Section 4.

Algorithm 1 Constraint Evaluation Kernel

```
 1: x = blockId.x·blockDim.x+threadId
 2: y = blockId.y
 3: if x ≥ Input[y].NumConstr then
 4:     return
 5: end if
 6: if ! covers(InputBF, Input[y].BFPtr[x]) then
 7:     return
 8: end if
 9: constrOp = Input[y].OpPtr[x]
10: constrVal = Input[y].ValPtr[x]
11: attrVal = Input[y].Value
12: constrType = Input[y].Type
13: if ! sat(constrOp, constrType, constrVal, attrVal) then
14:     return
15: end if
16: filterId = Input[y].FilterIdPtr[x]
17: count = atomicInc(FiltersCount[filterId])
18: if count+1==FiltersInfo[filterId].Size then
19:     interfaceId = FiltersInfo[filterId].InterfaceId
20:     Interfaces[interfaceId] = 1
21: end if
```

At line 6, each thread compares the `InputBF` with the Bloom filter associated with the constraint it has to evaluate. As for OCM, it immediately stops the evaluation if the former does not cover the latter.

At lines 9 – 10 each thread reads the operator and value of the constraint it has to process from the `ConstrOp` and `ConstrVal` tables (which are stored in the global memory) to the thread's local memory (i.e., hardware registers, if they are large enough), thus making subsequent accesses faster. Also notice that our organization of memory allows threads having contiguous identifiers to access contiguous regions of the tables above, and consequently contiguous regions of the global memory. This is particularly important when designing an algorithm for CUDA, since it allows the hardware to combine different read/write operations into a reduced number of memory-wide accesses, thus increasing performance.

At line 11 – 12 each thread reads the value and type of the attribute it has to process and evaluates the constraint it is responsible for (at line 13). We omit for simplicity the pseudo code of the sat function that checks whether a constraint is satisfied by an attribute.

If the constraint is not satisfied the thread immediately returns, otherwise it extracts the identifier of the filter the constraint belongs to (line 16) and updates the value of the corresponding field in table `FiltersCount` (line 17). Differently from OCM, CCM evaluates all constraints in parallel, including constraints having different names. Accordingly, it is possible that two or more threads try to access and modify the same element of `FiltersCount`. To avoid clashes, CCM exploits the atomicInc operation offered by CUDA, which atomically reads the value of a 32 bit integer from the global memory, increases it, and returns the old value.

At line 18 each thread checks whether the filter is satisfied, i.e., if the current number of satisfied constraints (old count plus one) equals the number of constraints in the filter. If this happens, the thread extracts the identifier of the interface the filter belongs to (line 19) and sets the corresponding position in `Interfaces` to 1 (line 20). Again, it is possible that multiple threads access the same position of `Interfaces` concurrently, but in this case, since they are all writing the same value, no conflict arises.

CUDA provides best performance when threads in the same warp (see Appendix A) follow the same execution path. After all unrequired threads have been stopped (lines 3 and 6), there are two conditional branches where the execution paths of different threads may diverge. The first one, at line 13, evaluates a single attribute against a constraint, while the second one, at line 18, checks whether all the constraints in a filter have been satisfied before setting the relevant interface to 1. The threads that follow the positive branch at line 18 are those that process the last matching constraint of a filter. Unfortunately, we cannot control the warps these threads belong to, since this depends from the content of the event under evaluation and from the scheduling of threads. Accordingly, there is nothing we can do to force threads on the same warp to follow the same branch. On the contrary, we can increase the probability of following the same execution path within function sat at line 13 by grouping constraints according to their type, operator, and value. This way we increase the chance that threads in the same warp, having contiguous identifiers, process similar constraints, thus following the same execution path into sat. Preliminary experiments, however, showed that this approach provides a negligible improvement in performance, while it makes creation of data structures (i.e., the tables in Fig. 2(a) that needs to be properly ordered) much slower. Accordingly we did not included it into the final version of CCM.

Reducing Latency. As we have seen, during event processing there are a number of tasks to be performed. First the CPU has to compute the `InputBF` Bloom filter and the `Input` data structure, which are subsequently copied to the GPU constant memory. Then the CPU has to launch the CCM main kernel, waiting for the GPU to finish its job. At the end the CPU has to copy the content of the `Interfaces` array back to the main memory, and it has to reset both the `FiltersCount` and `Interfaces` structures.

Since copies between CPU and GPU memories and kernel launch are asynchronous operations started by the CPU, a straightforward implementation of the workflow above could force the CPU to wait for an operation involving the GPU to finish before issuing the following one. However, this suffers from two limitations: *i.* it does not exploit the possibility to run tasks in parallel on the CPU and on the GPU (e.g., the CPU cannot prepare the input data for the next event while the GPU resets the `FiltersCount` and `Interfaces` structures); *ii.* for every instruction sent to the GPU (e.g., a kernel launch) it pays the communication latency introduced by the PCI-Ex bus.

To solve these issues, the current implementation of CCM uses a *CUDA Stream*, which is a queue where the CPU can put operations to be executed sequentially and in order on the GPU. This allows us to explicitly synchronize the CPU and the GPU only once for each event processed, when we have to be sure that the GPU has finished its processing and all the results (i.e., the whole `Interfaces` array) have been copied back into the main memory before the CPU can access them. This approach maximizes the overlapping of execution between the CPU and the GPU, and let the runtime issuing all the instructions it finds on the Stream sequentially, thus paying the communication latency only once. We will come back to this choice in Section 4, where we measure its impact on performance.

Reducing Memory Accesses. In real applications, the number of constraints in each filter is usually limited. For this reason, using a 32 bit integer for each element of the `FiltersCount` array wastes hardware resources. On the other hand, the elements of the `FiltersCount` array need to be accessed and updated atomically and CUDA offers atomic operations (like the *atomicInc* used by CCM) only for 32 bit data.

To work around this situation, CCM operates in eight stages, starting from stage 1, moving to the next stage at each event, and going back to stage 1 after processing the event at stage 8. While in stage s, CCM only considers the value stored in the sth half-byte of each element of the `FiltersCount` array. In practice the atomicInc operation at line 17 of Algorithm 1 increments FiltersCount[filterId] by 16^{s-1}, not 1. This way CCM may reset `FiltersCount` only before entering stage 1, using it 8 times before resetting it again. This optimization proves to significantly reduce the accesses to the GPU memory (the `FiltersCount` structure can be quite big), providing an average improvement in processing time of about 30%.

Finally, our tests have demonstrated that using an ad-hoc kernel for resetting the `FiltersCount` and `Interfaces` data structures may bring some improvement over the use of two separate memset operations. This is what CCM uses.

4 Evaluation

Our evaluation has several goals. First, we want to compare our work with state of the art algorithms to understand the real benefits in parallelizing the matching process. Second, we want to analyze our prototypes to better understand the implementation choices that mostly impact on performance. Finally, we want to test the behavior of our solutions when

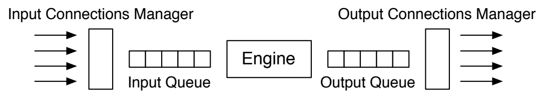


Fig. 5. Architecture of a publish-subscribe system

deployed on a complete system. To this extent, we considered a generic publish-subscribe infrastructure as shown in Fig. 5. It includes three components: the `Input Connections Manager` handles links with publishers. It receives events from the network, as streams of bytes, decodes them, and stores them into the `Input Queue`. The `Engine` executes the matching algorithm: it picks up events from the `Input Queue`, computes the set of destinations they have to be delivered to, and stores the results in the `Output Queue`. Finally, the `Output Connections Manager` handles links with subscribers, by reading events from the `Output Queue`, serializing them, and delivering them. The rest of the section is organized in two parts. In the first, we concentrate on the `Engine` and analyze its latency in processing a single event, under various workloads. In the second part we study the maximum throughput of the whole system. For space reasons, additional experiments are moved to Appendix B.

Existing matching algorithms (see Section 5) can be divided into two main classes: counting algorithms and tree-based algorithms. In our analysis, we selected one sequential algorithm for each class.

As a counting algorithm we chose SFF [10] (v.1.9.4), the matching algorithm used inside the Siena event notification middleware, which is known in the community for its performance. Similarly to PCM, SFF runs over the attributes of the event under consideration, counting the constraints they satisfy until one or more filters have been entirely matched. Differently from PCM, it combines identical constraints belonging to different filters. Moreover, when a filter f is matched, SFF marks the related interface, purges all the constraints and filters exposed by that interface, and continues until all interfaces are marked or all attributes have been processed. The set of marked interfaces represents the output of SFF. To maximize performance under a sequential hardware, SFF builds a complex, strongly indexed data structure, which puts together the predicates (decomposed into their constituent constraints) received by subscribers.

As a tree-based algorithm, we chose BETree, which in a recent publication [27] has demonstrated significant performance benefits when compared to several existing algorithms under a large number of workloads. It organizes constraints into a tree structure; intermediate nodes contain expressions to be evaluated against the content of events to determine the path to follow. Leaf nodes store the subscriptions satisfied by an event that reaches them. Differently from SFF and PCM, BETree only supports constraints on numeric values, and not on strings. While the source code of SFF is available for download, we could only obtain an executable of BETree explicitly compiled for our hardware by contacting the authors.

TABLE 1
Parameters in the Default Scenario

Number of events	1000
Attr. per event, min-max	3-5
Number of interf.	10
Constr. per filt., min-max	3-5
Filt. per interf., min-max	22500-27500
Number of names	100
Distribution of names	Uniform
Numerical/string constr.	100% / 0%
Operators	=(25%),≠(25%),>(25%),<(25%)
Number of values	100

All tests of this section were executed on a AMD Phenom II PC, with 6 cores running at 2.8GHz, and 8GB of DDR3 Ram. The GPU was a Nvidia GTX 460 with 1GB of GDDR5 Ram. We used the GCC compiler, version 4.7, and the CUDA runtime 5.0 for 64 bit Linux. Nowadays both the CPU and the GPU we adopted are considered mid-low level hardware. On the other hand, they were top level one year ago, and they have a similar price, so the comparison is fair.

4.1 Latency of Matching

To evaluate the latency of pure matching we defined a default scenario whose parameters are listed in Table 1, and used it as a starting point to build a number of different experiments, by changing the various parameters one by one and measuring how this impacts the performance of CCM, OCM, SFF, and BETree. In our tests we let each algorithm process 1000 events, one by one, and we compute the average processing time. To avoid any bias, we repeated all tests (including those reported in Section 4.2) 10 times, using different seeds to randomly generate subscriptions and events, and we plot the average value measured. The 95% confidence interval of this average was always below 1% of the measured value, so we omitted it from all the plots.

Default Scenario. Table 2 shows the processing times measured by the algorithms under analysis in the default scenario. This is a relatively easy-to-manage scenario. It includes one million constraints on average, which is not a huge number for large scale applications. Under this load, SFF requires 1.353ms to process a single event, while BETree requires 0.326ms. If we consider our algorithms, CCM requires 0.0205ms and OCM 0.0091ms, providing respectively a speedup of 66× and 148.7× w.r.t. SFF and 15.9× and 35.8× w.r.t. BETree. Finally, we included in our evaluation a new, unpublished, version 1.3 of BETree. It borrows some of the ideas of PCM (e.g., the use of Bloom filters to reduce the number of constraints to evaluate), providing a significant improvement over the original release. Despite these advancements, PCM

TABLE 2
Processing time in the default scenario

CCM	OCM	SFF	BETree	BETree 1.3
0.0205ms	0.0091ms	1.353ms	0.326ms	0.031ms

still provides better results, while offering a more expressive subscription language (e.g., it also includes constraints on strings).

Number of Attributes. Fig. 6 shows how performance changes with the average number of attributes inside events. All the algorithms under analysis exhibit higher matching times with a higher number of attributes. This is especially true for the counting algorithms (i.e., SFF and PCM) that need to explicitly evaluate all the event attributes. The impact is more evident on the CPU (SFF and OCM) then on the GPU (CCM). Indeed, as described in Section 3, OCM processes the different attributes sequentially, while CCM processes all of them in parallel, exploiting the large number of cores available on the GPU. As a result, the speedup of CCM over SFF increases with the number of attributes, moving from 24.9× to 58.4×. Similarly, the performance of OCM w.r.t. CCM decreases from 2.4× to 0.46× (i.e., CCM becomes twice as fast as OCM). Differently from counting algorithms, BETree uses a tree-based structure for matching events as a whole and not attribute by attribute. For this reason, we do not observe any significant change in the performance of the algorithm. The speedup of PCM over BETree decreases with the number of attributes, moving from 20.7× to 8.2× for CCM, and from 48.8× to 3.7× for OCM. Finally, BETree 1.3 shows a level of performance that is comparable (albeit slower) to CCM: in the extreme case of 9 attributes per event, it also outperforms OCM.

Number of Constraints per Filter. Fig. 7 shows how performance changes with the average number of constraints in each filter. During this test, we fixed the overall number of constraints to 1M, while changing the overall number of filters.

When increasing the number of constraints per filter, the optimization derived from the filter selection phase becomes more effective, thus reducing the processing times of PCM. BETree and SFF suffer this situation, showing an increased matching time as the number of constraints per filter grows. The only case in which PCM is outperformed by SFF is when we consider only one or two constraints per filter. This is a very special (and quite unrealistic) case in which the chance to find a matching filter for a given interface is very high, such that at the end all events are relevant for all interfaces. The pruning techniques of SFF work at their best in this case, while OCM and CCM always process all constraints, albeit in parallel. The BETree algorithm, which does not include any optimization of the matched interfaces, performs much worse in this region. BETree 1.3 has better performance in this area, being comparable to CCM, then it looses when the scenario becomes more complex. This proves the benefits of using parallel hardware (both multi-core CPUs and GPUs) when a large number of complex subscriptions are deployed on the system.

Number of Filters per Interface. Fig. 8 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. Accordingly, all the algorithms show growing processing times. This scenario emphasizes the advantages of parallel processing: CCM registers a 60.4× speedup w.r.t. BETree, a 697.7× speedup w.r.t. SFF, and a 4.16× speedup w.r.t. BETree 1.3 with 250k filters. The advantage is visible also for OCM: with 250k filters it registers a speedup of 102.3× w.r.t. BETree, a speedup of 1180.7× w.r.t. SFF, and a speedup of 7.1× w.r.t. BETree 1.3. These results are particularly interesting if we consider that increasing the number of filters also increases the number of shared constraints, which are considered only once in SFF and BETree, but multiple times in PCM. Finally, observe how a very small number of filters favors SFF and BETree: they perform better than PCM with less than 1000 filters. Under such circumstances the matching is very fast, with all algorithms (including the slowest SFF) registering an average processing time below 0.02ms, and the (almost fixed) overhead of the parallel architectures becomes relevant.

Number of Interfaces. Another important aspect that significantly influences the behavior of a matching algorithm is the number of interfaces. In Fig. 9 we analyze its impact on SFF, BETree, and PCM, moving from 10 to 100 interfaces. Notice that 100 interfaces may represent a realistic scenario, in which several clients are served by a common event dispatcher that performs the matching process for all of them. As in the previous experiments, increasing the number of interfaces also increases the number of constraints, and thus the complexity of matching. Accordingly, all algorithms show growing processing times as the number of interfaces grows. Also in this case the speedups of PCM constantly increases, both for CCM (up to 658.7× w.r.t. SFF, 60.7× w.r.t. BETree, 4.1× w.r.t. BETree 1.3), and for OCM (up to 1193.3× w.r.t. SFF, 109.9× w.r.t. BETree, 7.4× w.r.t. BETree 1.3).

4.2 Throughput

Consider the reference architecture shown in Fig. 5. With a low input rate events do not accumulate in the `Input Queue`: as soon as the `Input Connections Manager` enqueues an event, it is immediately dequeued and processed by the `Engine`. However, in case of bursts, the rate at which events are put in the `Input Queue` may temporarily become higher than the processing rate of the `Engine`. In this case, many events may be waiting in the `Input Queue`, ready

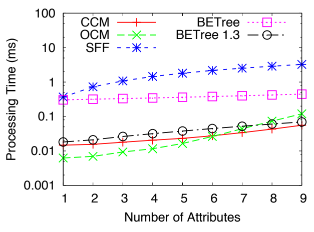


Fig. 6. Number of Attributes

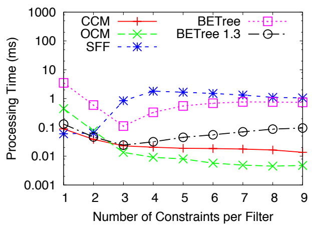


Fig. 7. Number of Constraints per Filter

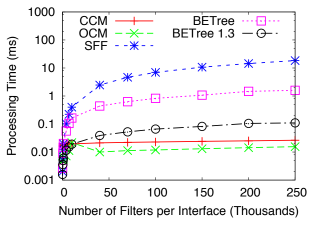


Fig. 8. Number of Filters per Interface

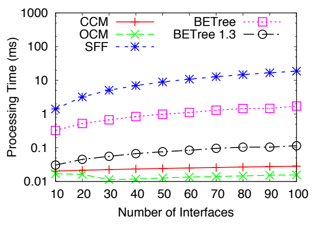


Fig. 9. Number of Interfaces

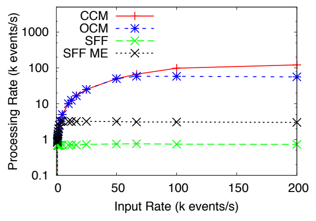


Fig. 10. Overall System Performance

to be processed. Processing them in parallel does not impact the average latency for matching each of them, but it may reduce the total time needed to evaluate all of them, thus improving the throughput of the system. We now investigate this aspect in details.

We implemented the system architecture shown in Fig. 5, using one thread inside the `Input Connections Manager` and one inside the `Output Connections Manager`, while a single remote client was used as a source and sink of events. The `Input Queue` had a finite size of 100 events. The remote client was used to generate events at increasing input rate and we measured the processing rate, i.e., the rate at which events were processed by the `Engine`. This two rates initially coincide. However, when the input rate begins overcoming the capabilities of the `Engine`, incoming events accumulate on the `Input Queue`. When it is completely filled, the `Input Connections Manager` have to drop incoming events, and the two rates start diverging. Notice that, since we are adopting an unbounded `Output Queue`, the processing rate measured is proportional to the throughput of the system. During this experiment, we modified CCM to process multiple events in parallel when they are available in the `Input Queue`. In particular, we use a different copy of the `FiltersCount` and `Interfaces` structures for each event to be processed in parallel. This way we could use a single kernel for multiple events, thus avoiding the overhead of launching multiple kernels.

Fig. 10 shows the results we measured. We could not include BETree in this test, since we had only access to the binary of the algorithm and it was not designed to process multiple events in parallel. For SFF we considered two version, one processing one event at a time, and one (`SFF ME`) processing multiple events in parallel. Ideally, the processing rate should be the inverse of the matching latency of the `Engine`. However, some CPU resources are used by the two `Connections Managers` to perform marshalling/unmarshalling of events and to handle the communication with sources and sinks, thus reducing the processing resources available to the `Engine`. This is confirmed by the results we measured for OCM. Indeed, it requires all 6 CPU cores of our test system to reach the peak performance measured in the previous sections. When less cores are available (some of them being allocated to other tasks) the overall performance drops. More precisely, OCM exhibits a processing time of 0.0091ms per event in our reference scenario, meaning that it should be able to process more than 109k events/s. However, when used within a complete system its processing rate hardly reaches 60k events/s. On the other hand, CCM does not suffer this problem. With a processing time of 0.0205ms, it could enable a maximum throughput of about 48k events/s. However, by processing multiple events in parallel, it reaches and overcomes a maximum throughput of 100k events/s. This highlights a key, very positive aspect of CCM: since it is entirely executed on the GPU, it only requires one CPU thread to start memory copies and to launch kernels inside the `Engine`, while most of the processing resources on the CPU are free for other system tasks, in particular marshalling/unmarshalling and communication. The same holds for SFF but this cannot be considered as a positive aspect. Indeed, SFF, with its purely sequential approach, is under-utilizing available resources (i.e., CPU cores).

4.3 Final Considerations

The results presented so far allow us to draw some general conclusions about publish-subscribe content-based matching on parallel hardware. First of all we may observe that the matching problem is relatively easy to parallelize. Indeed, using an appropriate algorithm, only a few operations (updates of filter counters) need to be performed atomically. On the other hand, a sequential algorithm like SFF or BETree can add a number of optimization that, albeit introducing much more synchronization points, may reduce the gap with OCM under particular scenarios.

Our experience in developing CCM let us draw some conclusions about CUDA. First of all, programming CUDA is (relatively) easy, while attaining good performance is (very) hard. Memory accesses and transfers tend to dominate over processing (at least for the matching problem) and must be carefully managed, while having thousands of threads, even if they are created to be immediately destroyed, has a minimal impact. Also, the fixed cost to pay to launch a kernel makes (relatively) simple problems not worth being demanded to the GPU (see the case of less than 1000 filters). Fortunately, it is easy to determine whether the set of subscriptions installed in the system is large enough to take advantage of CCM. In practical terms, we implemented a translator that allows us to switch dynamically between the CPU (running SFF or OCM) and the GPU (running CCM) to always get the best performance. Focusing on pure performance, we notice how using a GPU may provide large speedups w.r.t. using sequential algorithms on the CPU. More importantly, these speedups grow with the scale of the problem to solve. However, when considering latency, running a parallel algorithm on the CPU (OCM) provides similar and sometime better results than running on the GPU. This is mainly due to the fixed overhead required for moving information between the CPU and the GPU memory.

Finally, using the GPU has the additional, fundamental advantage of leaving the CPU free to focus on those jobs (like I/O) that do not fit GPU programming, as demonstrated by our analysis on the maximum throughput in Section 4.2. In this context CCM outperforms OCM thanks to the number of parallel resources available on the GPU, which allow multiple events to be processed in parallel.

For additional experiments, please refer to Appendix B, where we study the impact of further parameters on the latency of matching, we consider the time for deploying new subscriptions, we decompose and analyze in great details the processing times of CCM, and we evaluate the benefits of processing multiple events in parallel.

5 Related Work

The last decade saw the development of a large number of content-based publish-subscribe systems [25], [4], [15], [24], [13] first exploiting a centralized dispatcher, then moving to distributed solutions for improved scalability. Despite their differences, they all share the need of matching events against subscriptions. Two main categories of matching algorithms can be found in the literature: *counting* algorithms [16], [10], [30] and *tree-based* algorithms [2], [5], [27]. SFF and PCM are counting algorithms: they maintain a counter for each filter that records the number of constraints satisfied. Tree-based algorithms, like BE-Tree, organize subscriptions into a rooted search tree. Inner nodes represent an evaluation test, while leaves represent the received predicates. Given an event, the search tree is traversed from the root to the leaves. At every node, the value of an attribute is tested, and the satisfied branches are followed until the fully satisfied predicates (and corresponding interfaces) are reached at the leaves. To the best of our knowledge, no existing work has demonstrated the superiority of one of the two approaches. However, in a recent publication, BETree has been compared agains many state of the art matching algorithms, showing its performance advantages in a wide range of scenarios.

Despite these efforts, content-based matching is still considered a complex and time consuming task [9]. To overcome this limitation, researchers have explored two directions: on the one hand they proposed to distribute matching among multiple brokers, exploiting covering relationships between subscriptions to reduce the amount of work performed at each node [8]. On the other hand, they moved to probabilistic matching algorithms, trying to increase the performance of the matching process, while possibly introducing evaluation errors in the form of false positives [6], [20]. The use of a distributed dispatcher is orthogonal w.r.t. our work. Indeed, the brokers that build a distributed dispatcher have to perform the same kind of matching analyzed in this paper. Accordingly, PCM can be used in distributed scenarios, contributing to further improve performance. At the same time, some of the ideas behind PCM can be leveraged to speedup probabilistic algorithms through parallel hardware. Indeed, probabilistic matching usually involves encoding events and subscriptions as Bloom filters reducing the matching process to a comparison of bit vectors. This is a strongly data parallel computation, which would perfectly fit OpenMP and CUDA. We plan to explore this topic in the future.

The idea of parallel matching has been recently addressed in a few works. In [18], the authors exploit multi-core CPUs both to speedup the processing of a single event and to parallelize the processing of different events. Unfortunatly, the code is not available for a comparison; however, the processing delays appearing in the paper seem to be much worse than those obtained by PCM. Other works investigated how to parallelize matching using ad-hoc (FPGA) hardware [29], while we focus on off-the-shelf hardware. To the best of our knowledge, PCM is the first matching algorithm to be implemented on GPUs. A preliminary version of the algorithm, with an analysis of its performance (in terms of latency, only) when implemented on GPUs, has been published in [23], and later extended to location-aware publish-subscribe systems [11]. Along the same line, in [12] we explored the possibility to use GPUs to detect complex events as a combination of primitive ones in a temporal pattern.

The adoption of GPUs for general purpose programming is relatively recent and was first enabled in late 2006 when Nvidia released CUDA. Since then, commodity graphics hardware has become a cost-effective parallel platform to solve many general problems, including image processing, computer vision, signal processing, and graphs algorithms. See [26] for an extensive survey on the application of GPUs.

6 Conclusions

In this paper, we presented a parallel publish-subscribe content-based matching algorithm and its implementation both on a multi-core CPU, using OpenMP, and on CUDA GPUs. We compared it with SFF and BETree, two state of the art sequential algorithms. Results demonstrate the benefits of parallelism in a wide spectrum of scenarios. Moreover, delegating to the GPU the effort required for the matching process brings additional advantages when considering the whole system, by leaving the main CPU free to perform other tasks. Although our presentation focuses on the case of content-based publish-subscribe systems, the problem of matching is more general. Indeed, as observed by others [10], [15], several applications can directly benefit from a content-based matching service. They include intrusion detection systems and firewalls, which need to classify packets as they flow on the network; intentional naming systems [1], which realize a form of content-based routing; distributed data sharing systems, which need to forward queries to the appropriate servers; and service discovery systems, which need to match service descriptions against service queries.

Acknowledgment

We would like to thank Prof. Hans-Arno Jacobsen and Dr. Mohammad Sadoghi for giving us access to their BETree prototype and for helping us in using it during our tests. This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom, and by the Dutch national program COMMIT.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *SOSP*, pages 186–201. ACM Press, 1999.

[2] 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*, pages 53–61, New York, NY, USA, 1999. ACM.

[3] T. S. Axelrod. Effects of synchronization barriers on multiprocessor performance. *Parallel Comput.*, 3:129–140, 1986.

[4] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. Technical report, DIS, La Sapienza, 2005.

[5] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *ICSE*, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.

[6] A. Carzaniga and C. P. Hall. Content-based communication: a research agenda. In *SEM*, Portland, Oregon, USA, 2006.

[7] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *PODC*, pages 219–227, Portland, Oregon, 2000.

[8] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *INFOCOM*, Hong Kong, China, 2004.

[9] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in Lecture Notes in Computer Science, pages 59–68, Scottsdale, Arizona, 2001. Springer-Verlag.

[10] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, pages 163–174, Karlsruhe, Germany, 2003.

[11] G. Cugola and A. Margara. High-performance location-aware publish-subscribe on gpus. In *Middleware*, pages 312–331, 2012.

[12] G. Cugola and A. Margara. Low latency complex event processing on parallel hardware. *Journal of Parallel and Distributed Computing*, 72(2):205 – 218, 2012.

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

[14] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5:46–55, 1998.

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

[16] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, pages 115–126, New York, NY, USA, 2001. ACM.

[17] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *SIGCOMM*, 29:251–262, 1999.

[18] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen. Parallel event processing for content-based publish/subscribe systems. In *DEBS*, pages 8:1–8:4, New York, NY, USA, 2009. ACM.

[19] L. Fiege, G. Mühl, and A. P. Buchmann. An architectural framework for electronic commerce applications. In *GI Jahrestagung (2)*, pages 928–938, 2001.

[20] Z. Jerzak and C. Fetzer. Bloom filter based routing for content-based publish/subscribe. In *DEBS*, pages 71–81, New York, NY, USA, 2008. ACM.

[21] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders. A design pattern language for engineering (parallel) software: merging the plpp and opl projects. In *ParaPLoP*, pages 9:1–9:8, New York, NY, USA, 2010. ACM.

[22] C. Krgel, T. Toth, and C. Kerer. Decentralized event correlation for intrusion detection. In K. Kim, editor, *Information Security and Cryptology, ICISC*, volume 2288, pages 59–95. Springer Berlin / Heidelberg, 2002.

[23] A. Margara and G. Cugola. High performance content-based matching using gpus. In *DEBS*, pages 183–194, New York, NY, USA, 2011. ACM.

[24] G. Mühl, L. Fiege, F. Gartner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *MASCOTS*, 2002.

[25] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[26] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A Survey of General–Purpose Computations on Graphics Hardware. *Computer Graphics*, Volume 26, 2007.

[27] M. Sadoghi and H.-A. Jacobsen. Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In *SIGMOD*, pages 637–648, New York, NY, USA, 2011. ACM.

[28] S. Schneidert, H. Andrade, B. Gedik, K.-L. Wu, and D. S. Nikolopoulos. Evaluation of streaming aggregation on parallel hardware architectures. In *DEBS*, pages 248–257, New York, NY, USA, 2010. ACM.

[29] K. H. Tsoi, I. Papagiannis, M. Migliavacca, W. Luk, and P. Pietzuch. Accelerating publish/subscribe matching on reconfigurable supercomputing platforms. In *MRSC*, Rome, Italy, 2010.

[30] T. W. Yan and H. García-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Trans. Database Syst.*, 19(2):332–364, June 1994.

Appendix A
Parallel Programming Models

Attaining good performance with parallel architectures is a complex task. A naive parallelizing of a sequential algorithm is usually not sufficient to efficiently exploit the presence of multiple processing elements, and a complete re-design of the algorithm may be necessary, taking into account the peculiarities of the underlying architecture and its programming model. In this section we present the architectures and programming models considered in this paper, focusing on the abstractions offered and on the aspects that mostly affect performance.

A.1 Multicore CPU Programming with OpenMP

OpenMP (Open Multi-Processing) is an API for shared memory multiprocessing programming in C/C++ and Fortran, consisting of a set of compiler directives and library routines. OpenMP provides thread programming at a high level: the programmer specifies which portions of code should execute in parallel, while the compiler decides low-level details, including the creation of threads and the assignment of tasks to threads. In particular, to develop data parallel algorithms like PCM, OpenMP supports the SPMD (Single Program Multiple Data) implementation strategy [21], through the following primitives:

Parallel regions. The programmer defines regions of code that have to be executed in parallel by different threads, and explicitly specifies the number of threads to be used. Inside a parallel region, each thread is uniquely identified by its *ThreadNum*, which the programmer may access and use to differentiate the execution flows of threads.

Shared memory. Threads in a parallel region can declare private, thread-local variables, but they can also access variables from a common shared memory. Limiting data dependencies among threads is a key aspect to obtain good performance.

Synchronization. To control the access to shared memory, OpenMP provides two different kinds of synchronization primitives: *critical sections* and *barriers*. Critical sections specify portions of parallel regions that must be executed in mutual exclusion by the different threads. Barriers are synchronization points at which all threads must wait before any is allowed to proceed [3].

On top of them, OpenMP also offers some higher level primitives. The most remarkable example (also used in the remainder of this paper) is the *parallel for*, which executes for loops in parallel splitting them among the different cores available.

A.2 GPU Programming with CUDA

Introduced by Nvidia in Nov. 2006, the CUDA architecture offers a new programming model and instruction set for general purpose programming on GPUs. Different languages can be used to interact with a CUDA device: we adopted CUDA C, a dialect of C explicitly devoted to program GPUs. The CUDA programming model is founded on five key abstractions:

Hierarchical organization of thread groups. The programmer is guided in partitioning a problem into coarse sub-problems to be solved *independently* in parallel by *blocks* of threads, while each sub-problem must be decomposed into finer pieces to be solved *cooperatively* in parallel by all threads within a block. This decomposition allows the algorithm to easily scale with the number of available processor cores, since each block of threads can be scheduled on any of them, in any order, concurrently or sequentially.

Shared memories. As in OpenMP, CUDA threads may access data from multiple memory spaces during their execution: each thread has a *private local memory* for automatic variables; each block has a *shared memory* visible to all threads in the same block; finally, all threads have access to the same *global memory*.

Barrier synchronization. Since thread blocks are required to execute independently from each other, no primitive is offered to synchronize threads of different blocks. On the other hand, threads within a single block work in cooperation, and thus need to synchronize their execution to coordinate memory access. In CUDA this is achieved exclusively through *barriers*.

Separation of host and device. The CUDA programming model assumes that CUDA threads execute on a physically separate *device* (the GPU), which operates as a coprocessor of a *host* (the CPU) running a C/C++ program. The host and the device maintain their own separate memory spaces. Therefore, before starting a computation, it is necessary to explicitly allocate memory on the device and to copy there the information needed during execution. Similarly, at the end results have to be copied back to the host memory and the device memory have to be deallocated.

Kernels. Like parallel regions in OpenMP, *kernels* are special functions that define a single flow of execution for multiple threads. When calling a kernel k, the programmer specifies the number of threads per block and the number of blocks that must execute it. Inside the kernel it is possible to access two special variables provided by the CUDA runtime: the *threadId* and the *blockId*, which together allow to uniquely identify each thread among those executing the kernel. As for the *ThreadNum* in OpenMP, conditional statement involving these variables can be used to differentiate the execution flows of different threads.

A.2.1 Architectural Issues

If compared to OpenMP, the CUDA model provides thread programming at a lower level, i.e., there are details about the hardware that a programmer cannot ignore while designing an algorithm for CUDA.

The CUDA architecture is built around a scalable array of multi-threaded *Streaming Multiprocessors* (*SMs*). When a CUDA program on the host CPU invokes a kernel *k*, the blocks executing *k* are enumerated and distributed to the available SMs. All threads belonging to the same block execute on the same SM, thus exploiting fast SRAM to implement the shared memory. Multiple blocks may execute concurrently on the same SM as well. As blocks terminate new blocks are launched on freed SMs. Each SM creates, manages, schedules, and executes threads in groups of parallel threads called *warps*. Individual threads composing a warp start together but they have their own instruction pointer and local state and are therefore free to branch and execute independently. On the other hand, full efficiency is realized only when all threads in a warp agree on their execution path, since CUDA parallelizes them executing one common instruction at a time. If threads in the same warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

Inside a single SM, instructions are pipelined but, differently from modern CPU cores, they are executed in order, without branch prediction or speculative execution. To maximize the utilization of its computational units, each SM is able to maintain the execution context of several warps on-chip, so that switching from one execution context to another has no cost. At each instruction issue time, a warp scheduler selects a warp that has threads ready to execute (not waiting on a synchronization barrier or for data from the global memory) and issues the next instruction to them.

An additional issue is represented by memory accesses. If the layout of data structures allows threads with contiguous ids to access contiguous memory locations, the hardware can organize the interaction with memory into several memory-wide operations, thus maximizing throughput. This aspect significantly influenced the design of PCM's data structures, as we discuss in the next section.

In summary, we can say that, similarly to OpenMP, the CUDA programming model ease the implementation of data parallel algorithms using a SPMD implementation strategy. However, while OpenMP runs on hardware architectures where different threads are free to execute different instructions without incurring in additional overhead, the CUDA architecture is designed to execute efficiently only data parallel code that operates on contiguous memory regions.

Finally, to give an idea of the capabilities of a modern GPU supporting CUDA, we provide some details of the Nvidia GTX 460 card we used for our tests. It includes 7 SMs, which can handle up to 48 warps of 32 threads each (for a maximum of 1536 threads). Each block may access a maximum amount of 48KB of shared, on-chip memory within each SM. Furthermore, it includes 1GB of GDDR5 memory as global memory. This information must be carefully taken into account when programming: shared memory must be exploited as much as possible, to hide the latency of global memory accesses, but its limited size significantly impacts the design of algorithms.

TABLE 3
Analysis of Matching Algorithms

	CCM	OCM	SFF	BETree	BETree 1.3
Processing time - Default scenario	0.0205ms	0.0091ms	1.353ms	0.326ms	0.031ms
Subscriptions deployment time - Default scenario	527.5ms/ 4141ms	483.2ms/ 1088ms	992.2ms	n.a.	n.a.
Data structure size - Default scenario	33.9MB (GPU)	33.9MB (CPU)	42.4MB (CPU)	n.a.	n.a.
Processing time - Zipf distribution of names	0.0270ms	0.0198ms	3.662ms	0.410ms	0.044ms

Appendix B Evaluation (Additional Experiments)

We here introduce additional experiments that integrate and complete our evaluation in Section 4. In particular, this section is organized in three parts. First, we study the cost for deploying new subscriptions in PCM. Second, we extend our analysis on the latency of matching by considering additional parameters and by decomposing and analyzing the processing times of CCM in great details. Finally, we investigate the benefits of processing multiple events in parallel with PCM.

B.1 Deployment of Subscriptions

Beside measuring processing time, it is also interesting to study the time required to create the data structures used during event evaluation. Indeed, they need to be generated at run-time, when new subscriptions are deployed on the engine. Even if it is common to assume that the number of publishing largely exceeds the number of subscribing/unsubscribing in any event-based application, this time may become relevant in some scenarios, and thus deserves to be analyzed. Table 3 (second row) shows the average time required by SFF, OCM, and CCM to create their data structures. Since we did not have access to the source code of BETree, we could not instrument it to measure this value.

For CCM and OCM, we consider two separate scenarios, which enable us to better investigate the aspects that influence subscriptions deployment time more. In the first version, we disable the "filters selection" phase, thus storing all constraints into a single table. In the second phase, we run the standard PCM algorithm, including the "filters selection" phase.

In the first case, the data structures adopted by PCM are significantly simpler than the indexed structures used by SFF. This enables OCM to be twice as fast in creating them, with an overall deployment time of less than 500ms. When using CCM, most data structures have to be installed on the GPU memory. Moving data from the CPU to the GPU memory may introduce non negligible delays, caused by the (relatively) limited PCI-Ex performance. In particular, we observed that latency is the most limiting factor w.r.t. bandwidth, accordingly CCM builds all data structures on the CPU memory, and transfer them to the GPU using a single copy. With this solution CCM is less than 50ms slower than OCM.

When introducing the "filters selection" phase, the complexity of data structures increases. This is visible in OCM, which doubles its deployment time. When considering CCM, the impact is much larger: when considering multiple tables for storing constraints, we transfer tham to the GPU separately, thus paying the PCI-Ex latency multiple times. In this settings, the subscriptions deployment time of CCM increases by a factor of 8.

The complexity of the data structures used by SFF also reflects on their size. As shown in Table 3 (third row), OCM and CCM require less memory: the default scenario occupies 33.9MB vs 42.4MB. It is worth mentioning that the maximum occupancy of GPU memory we measured in our tests was below 200MB. Since GPUs nowadays have at least 1GB of Ram, contrary to what happens in other domains, GPU memory occupancy is not a problem for content-based matching.

B.2 Latency of Matching

An Analysis of CCM Processing Times. Table 4 analyzes the cost of the different operations performed by CCM during the matching process in the default scenario. In particular, it splits processing time into five parts: the time required to create the input data structures, the time to copy them from the CPU to the GPU memory, the time to execute the kernel, the time to copy results back to the CPU memory, and the time to clear the data structures used by the GPU. To correctly measure these timings, we synchronized the CPU and the GPU after each of the five steps. By doing this we eliminate the advantages of using CUDA Streams, as described in Section 3. Accordingly, this test also allows us to quantify the benefits of Streams. Interestingly, in the default scenario, the time required to transfer data from the CPU to the GPU and back is more than half of the total time, while the actual executon time is only one third of the total time. This can be explained by observing that our default scenario exhibits a relatively high selectivity of events, allowing most of the threads launched in the kernel to terminate immediately after the comparison of bloom filters or after the constraint evaluation. Only few of them have to access the memory to update the `FiltersCount` or the `Interfaces` arrays. On the other hand, the processing time grows in other scenarios, in some cases even considerably, while the times for data transfer and for data structure reset are independent from the complexity of the scenario and remain almost fixed.

TABLE 4
Analysis of CCM Processing Times

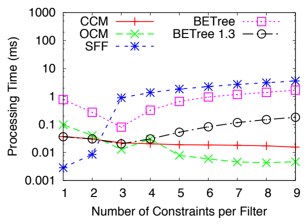
Prepare Input Data (CPU)	Copy Input Data (CPU to GPU)	Kernel Execution (GPU)	Copy Results (GPU to CPU)	Clear Data Struct (GPU)
0.003577ms	0.007768ms	0.01154ms	0.007324	0.002238

Finally, if we sum the five contributions, we obtain a total processing time for a single event of 0.03245ms, as opposed to the 0.0205ms measured with the adoption of CUDA Streams. Since PCM does not need to transfer large chunks of data to the GPU during processing, it suffers from the delay of the PCI-Ex more than from its limited bandwidth. This justifies the significant advantages introduced by Streams, which allow us to enqueue several operations and to pay the delay on the bus only once for all of them.

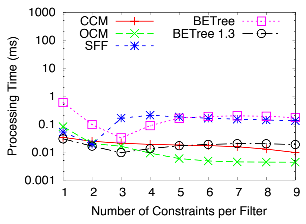
Distribution of Names. The distribution of the names adopted inside constraints is an important aspect for CCM. Indeed CCM creates and launches a number of threads that depends from the size of the longest row in the `Constraints` tables among those selected by the names appearing in the incoming event. In presence of rows with very different sizes, the number of unrequired threads may be relevant. To investigate the impact of this aspect, we changed the distribution of names for both constraints and attributes, moving from the uniform distribution adopted in the default scenario to a Zipf distribution, considered realistic for many application fields [17]. Table 3 (fourth row) shows the results we obtained. First of all we notice how all algorithms increase their matching time w.r.t. the default scenario. The counting algorithms use names to reduce the number of constraints to process, and this pre-filtering becomes less effective with a Zipf distribution. SFF suffers from this problem more than PCM, whose simpler constraint evaluation process and the use of Bloom filters mitigate the cost of considering a larger number of constraints. The speedup w.r.t. SFF increases to 135.6× for CCM and 184.9× for OCM. BETree employs self-adjustment policies to dynamically adapt the processing tree to the specific workload; for this reason, it is the algorithm that suffers less from the use of a Zipf distribution for names. The speedups of CCM and OCM w.r.t. it decrease to 15.2× and 20.7× respectively.

Number of Constraints per Filter. Fig. 11 shows how performance changes with the average number of constraints in each filter.

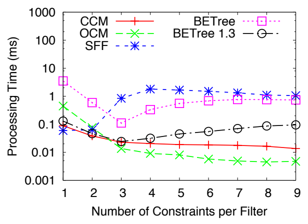
We performed two kinds of experiments: in the first one (see Fig. 11(a)), we set the overall number of filters to 25k; accordingly, increasing the number of constraints per filter also increases the overall number of constraints, and consequently the complexity of matching. In the second one we fixed the overall number of constraints to 200k (Fig. 11(b)), 1M (Fig. 11(c)), and 5M (Fig. 11(d)), while changing the overall num-



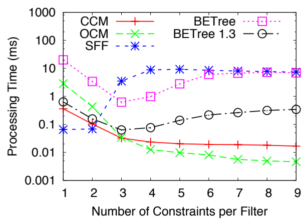
(a) 25k Filters



(b) 200k Constraints



(c) 1M Constraints



(d) 5M Constraints

Fig. 11. Number of Constraints per Filter

ber of filters.

When increasing the number of constraints per filter, the optimization derived from the use of a Bloom filter becomes more effective, thus reducing the processing times of PCM. Remarkably, this is true also in Fig. 11(a), where the number of constraints to be considered increases. BETree and SFF suffer this situation, showing an increased matching time as the number of constraints per filter grows (even when the total number of constraints remains fixed).

The only case in which PCM is outperformed by SFF or BETree is when we consider only one or two constraints per filter. This is a very special (and quite unrealistic) case in which the chance to find a matching filter for a given interface is very high, such that at the end all events are relevant for all interfaces. The pruning techniques of SFF work at their best in this case, while OCM and CCM always process all constraints, albeit in parallel. The BETree algorithm, which does not include any optimization of the matched interfaces, performs much worse in this region.

Finally, even when considering BETree 1.3, we observe how the advantages of PCM significantly increase with the complexity of the scenario. This proves the benefits of using parallel hardware (both multi-core CPUs and GPUs) when a large number of complex subscriptions are deployed on the system.

Number of Names. As discussed in Section 3, PCM makes use of the attribute names in the incoming event to select which constraints have to be evaluated. The same holds for SFF. Accordingly, the total number of names used inside constraints and attributes represent a key performance indicator. Fig. 12 shows how the processing times change with the number of names. Increasing this number allows the "filter selection" and "constraint selection" phases (common to all the counting algorithms, and always performed on the CPU), to discard a higher number of filter and constraints. Accordingly, the cost of the "constraint evaluation and counting" phase (the more expensive in terms of computation and also the one that CCM performs on the GPU) decreases. This is confirmed by Fig. 12: all the counting algorithms perform better when the number of names increases, especially when moving from 10 to 100 names. After this threshold

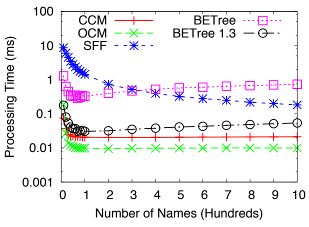


Fig. 12. Number of Different Names

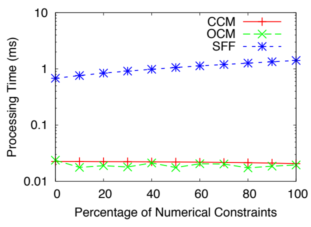


Fig. 13. Type of Constraints

times tend to stabilize. This fact can be explained by observing that over a certain number of names the "constraint evaluation and counting" phase becomes so simple that the processing time cannot decrease anymore.

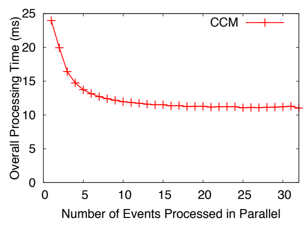
If we consider BETree, it shows two different regions. First, its processing time decreases, when moving from 10 to 100 names; after that, it increases again.

Type of Constraints. Fig. 13 shows how performance changes when changing the type of constraints. In particular, we measured the processing time when changing the percentage of constraints involving numerical values (the remaining ones involve strings). We could not consider BETree in this test, since it only supports numerical constraints.

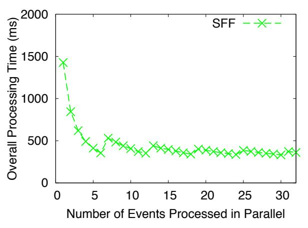
When considering the impact of constraint types, different aspects cooperate in determining the overall processing times. On the one hand, matching numerical values is less expensive than matching strings. On the other hand, the chances of matching a numerical constraint are higher than those of a string constraint, which results in a greater matching effort to increase counters and check if all constraints of a filter have been matched. The second aspect is less relevant for CCM and OCM, where all operations are performed in parallel, while it has a greater impact on SFF. This explains why the matching time of SFF increases with the number of numerical constraints, while CCM and OCM show constant or even decreasing processing times.

B.3 Processing Events in Parallel

With reference to the architecture in Figure 5, this section studies whether processing the events waiting in the `Input Queue` in parallel may introduce some benefits for the algorithms we are considering. To



(a) CCM



(b) SFF

Fig. 14. Processing Events in Parallel

do so we created 1000 events and deployed them in the `Input Queue`. Then we let all algorithms pick up and process all of them k at a time (in parallel), while varying k from 1 to 32, measuring the total time spent. Subscriptions and events were generated using the parameters of our default scenario. For the algorithms running on the CPU we used an OpenMP parallel for to iterate over events. This implementation is possible also with CUDA, since starting from the CUDA Toolkit 4.0 different CPU threads can safely invoke different kernels on the same GPU. However, this would waste CPU resources that may be useful for other purposes, as we will show in the next section. Accordingly, we followed a different route, modifying CCM to use a different copy of the `FiltersCount` and `Interfaces` structures for each event to be processed in parallel. This way we could use a single kernel for multiple events, thus avoiding the overhead of launching multiple kernels. Fig. 14 shows the results we measured. When increasing the number of events processed in parallel, SFF significantly increases its performance, moving from 1350ms to 360ms to process 1000 events. In particular, this time quickly drops moving from 1 to 6 events processed in parallel, where 6 is exactly the number of available cores. The processing time then increases again with 7 events and slowly decreases, becoming almost constant.

OCM already uses all the cores available to process a single event. Accordingly, processing multiple events in parallel does not increase its performance. Conversely, CCM shows a significant benefit: the time to process 1000 events decreases from 23.9ms to 11.0ms. This means that the GPU has enough resources to analyze several events in parallel. Finally, we could not perform this test (and the following one) on BETree, since we had only access to the binary of the algorithm and it was not designed to process multiple events in parallel.

Biographies

Alessandro Margara. Alessandro Margara is currently a post-doctoral researcher in the High Performance Distributed Computing group, at the Vrije Universiteit of Amsterdam. He previously worked as a PhD student and post-doctoral researcher in the DeepSE Group at Politecnico di Milano. In 2012, he received his PhD from Politecnico di Milano, with laude.

His main research interests are in the area of Distributed Systems and, more in particular, Event-Based Middleware. During his PhD studies, he focused on the definition of a Complex Event Processing middleware, with focus on event definition language expressiveness and ease of use, processing algorithm efficiency, and system scalability.

As part of this research project he developed, together with Gianpaolo Cugola, a complex event definition language (TESLA), an event processing system (T-Rex), and a protocol for distributed event detection (RACED).

During the design of T-Rex, moved by the requirement of low-latency processing expressed by applications, he started concentrating on parallel computing, by implementing algorithms for event processing designed to take advantage of the processing power of multi-core CPUs and modern GPUs. These experiences increased his interest in parallel programming, and in particular in programming language support and abstractions for parallelism.

As part of the high performance distributed computing group at the Vrije Universiteit of Amsterdam, he is currently involved in a new research projects that aim at extending the technologies for Semantic Web to capture dynamic, frequently changing information.

Gianpaolo Cugola. Gianpaolo Cugola received his Dr.Eng. degree in Electronic Engineering from Politecnico di Milano. In 1998 he received the Prize for Engineering and Technology from the Dimitri N. Chorafas Foundation for his Ph.D. thesis on Software Development Environments. He is currently Associate Professor at Politecnico di Milano where he teaches several courses in the area of Computer Science.

During his professional life, he has been involved in several projects financed by the EU commission (IDEAS-ERC-227977 SMSCom, IST-034963 WASP, IST-511556 POMPEI, IST-11400 MOTION, ESPRIT-34840 PIE, ESSI-21244 MIDAS), by Microsoft Research (”Network Aware Programming” and ”Virtual Campus”), and by the Italian governor (PRIN D-ASAP, FIRB Insyeme, CNR IS_MANET). He is co-author of tens of scientific papers published in international journals and conference proceedings.

His research interests are in the area of Software Engineering and Distributed Systems. In particular, his current research focuses on middleware technology for largely distributed and highly reconfigurable distributed applications with a special attention to the issue of Content Based Routing and Complex Event Processing as the basic mechanism to develop advanced middleware services like publish/subscribe and data sharing.