High-Performance Publish-Subscribe Matching Using Parallel Hardware

Alessandro Margara and Gianpaolo Cugola

1 Introduction

Most distributed applications involve some form of event-based interaction, often implemented using a *publish-subscribe infrastructure* (see Fig. 1) 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 nontrivial 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 CPUs and GPUs that mostly impact performance.

The remainder of the paper is organized as follows. Section 2 introduces the event model we adopt. Section 3

• *The authors are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy. E-mail: {margara, cugola}@elet.polimi.it.*

Manuscript received 5 June 2012; revised 11 Jan. 2013; accepted 16 Jan. 2013; published online 14 Feb. 2013.

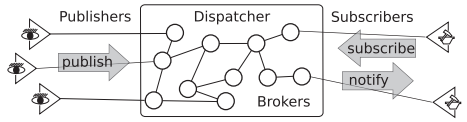


Fig. 1. A typical publish-subscribe infrastructure.

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 appendices provide the required background on OpenMP and CUDA together with additional tests.

2 Events and Predicates

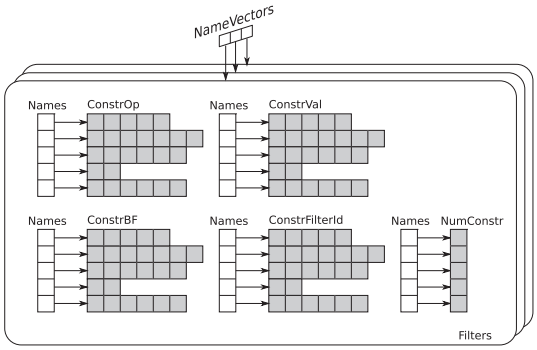
To be as general as possible we assume a data model which is very common among event-based systems [10]. We represent an *event notification*, or simply *event*, as a set of *attributes*, i.e., $\langle name, value \rangle$ pairs. Values are typed and we consider both numbers and strings. As an example, $e_1 = [\langle area, \text{``}area1\text{''} \rangle, \langle temp, 25 \rangle, \langle wind, 15 \rangle]$ 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 follows. 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).

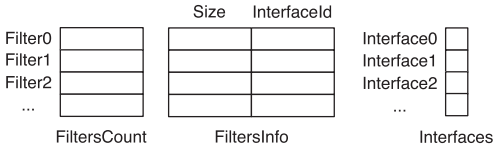
3 The PCM Algorithm

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



(a) Filters and Constraints Tables



(b) Filters and Interfaces Tables

Fig. 2. Data structures.

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 with 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. 2a, 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 five 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 multicore CPUs.

Fig. 2b 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 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. 2a, 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.

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 with c. This is an additional, quick check (a bitwise 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 percent of the times using 64 bit integers as Bloom filters) and it does not impact the correctness of results.

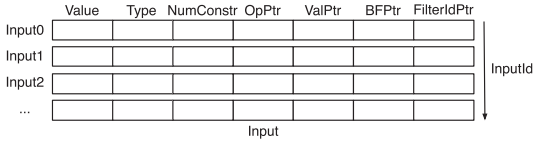


Fig. 3. Input data.

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. 2a), 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, 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) have 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

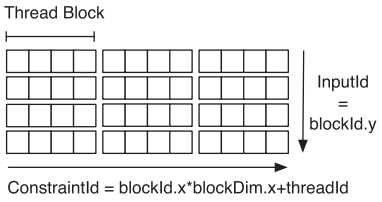


Fig. 4. Organization of blocks and threads.

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 four threads (in real cases we have 256 or 512 threads per block).

The pseudocode of the CCM kernel is presented in Algorithm 1. At the first two lines, each thread determines its x and y coordinates in the bidimensional 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, for example, 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)
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 pseudocode 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. 2a 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: 1) 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); 2) 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 eight 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 percent.

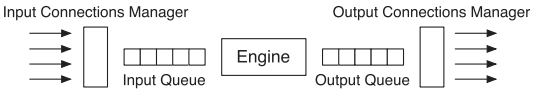


Fig. 5. Architecture of a publish-subscribe system.

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 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.

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%),$\neq$(25%),$>$(25%),$<$(25%)
Number of values	100

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.

All tests of this section were executed on a AMD Phenom II PC, with six cores running at 2.8 GHz, and 8 GB of DDR3 Ram. The GPU was a Nvidia GTX 460 with 1 GB 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 1,000 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) ten times, using different seeds to randomly generate subscriptions and events, and we plot the average value measured. The 95 percent confidence interval of this average was always below 1 percent 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,

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

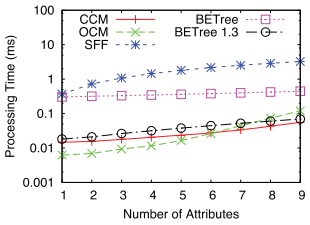


Fig. 6. Number of attributes.

SFF requires 1.353 ms to process a single event, while BETree requires 0.326 ms. If we consider our algorithms, CCM requires 0.0205 ms and OCM 0.0091 ms, providing respectively a speedup of $66\times$ and $148.7\times$ w.r.t. SFF and $15.9\times$ and $35.8\times$ 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 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 $24.9\times$ to $58.4\times$. Similarly, the performance of OCM w.r.t. CCM decreases from $2.4\times$ to $0.46\times$ (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 $20.7\times$ to $8.2\times$ for CCM, and from $48.8\times$ to $3.7\times$ for OCM. Finally, BETree 1.3 shows a level of performance that is comparable (albeit slower) to CCM: in the extreme case of nine 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

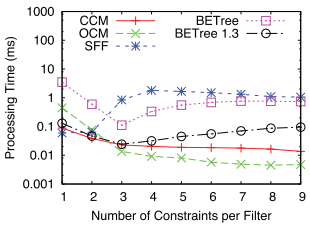


Fig. 7. Number of constraints per filter.

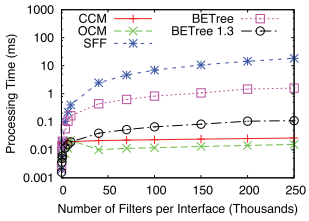


Fig. 8. Number of filters per interface.

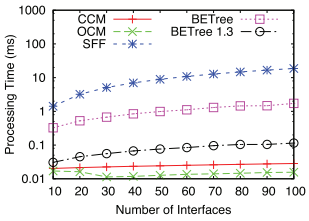


Fig. 9. Number of interfaces.

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 multicore 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 $60.4\times$ speedup w.r.t. BETree, a $697.7\times$ speedup w.r.t. SFF, and a $4.16\times$ 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 $102.3\times$ w.r.t. BETree, a speedup of $1180.7\times$ w.r.t. SFF, and a speedup of $7.1\times$ 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 1,000 filters. Under such circumstances the matching is very fast, with all algorithms (including the slowest SFF) registering an average processing time below 0.02 ms, 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 $658.7\times$ w.r.t. SFF, $60.7\times$ w.r.t. BETree, $4.1\times$ w.r.t. BETree 1.3), and for OCM (up to $1193.3\times$ w.r.t. SFF, $109.9\times$ w.r.t. BETree, $7.4\times$ 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 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`. These 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` has 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

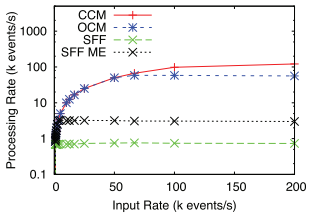


Fig. 10. Overall system performance.

versions, 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 marshaling/unmarshaling 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 six 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 0.0091 ms 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 0.0205 ms, 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 marshaling/unmarshaling and communication. The same holds for SFF but this cannot be considered as a positive aspect. Indeed, SFF, with its purely sequential approach, is underutilizing 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 1,000 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 BETree, 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 against 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 multicore CPUs both to speedup the processing of a single event and to parallelize the processing of different events. Unfortunately, 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 multicore 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.

Acknowledgments

The authors would like to thank Professor Hans-Arno Jacobsen and Dr. Mohammad Sadoghi for giving them access to their BETree prototype and for helping them in using it during our tests. This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The Design and Implementation of An Intentional Naming System," *Proc. 17th ACM Symp. Operating Systems Principles,* pp. 186-201, 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," *Proc. 18th Ann. ACM Symp. Principles of Distributed Computing,* pp. 53-61, 1999.

[3] T.S. Axelrod, "Effects of Synchronization Barriers on Multiprocessor Performance," *Parallel Computing,* vol. 3, pp. 129-140, 1986.

[4] R. Baldoni and A. Virgillito, "Distributed Event Routing in Publish/Subscribe Communication Systems: A Survey," technical report, Dipartimento di Informatica e Sistemistica, 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," *Proc. 23rd Int'l Conf. Software Eng.,* pp. 443-452, 2001.

[6] A. Carzaniga and C.P. Hall, "Content-Based Communication: A Research Agenda," *Proc. Sixth Int'l Workshop Software Eng. and Middleware,* 2006.

[7] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, "Achieving Scalability and Expressiveness in An Internet-Scale Event Notification Service," *Proc. 19th Annu. ACM Symp. Principles of Distributed Computing,* pp. 219-227, 2000.

[8] A. Carzaniga, M.J. Rutherford, and A.L. Wolf, "A Routing Scheme for Content-Based Networking," *Proc. IEEE INFOCOM,* 2004.

[9] A. Carzaniga and A.L. Wolf, "Content-Based Networking: A New Communication Infrastructure," *Proc. NSF Workshop Infrastructure for Mobile and Wireless Systems,* pp. 59-68, 2001.

[10] A. Carzaniga and A.L. Wolf, "Forwarding in a Content-Based Network," *Proc. SIGCOMM,* pp. 163-174, 2003.

[11] G. Cugola and A. Margara, "High-Performance Location-Aware Publish-Subscribe on GPUs," *Proc. 13th Int'l Middleware Conf. (Middleware '12),* pp. 312-331, 2012.

[12] G. Cugola and A. Margara, "Low Latency Complex Event Processing on Parallel Hardware," *J. Parallel Distributed Computing,* vol. 72, no. 2, pp. 205-218, 2012.

[13] G. Cugola and G. Picco, "REDS: A Reconfigurable Dispatching System," *Proc. Sixth Int'l Workshop Software Eng. Middleware (SEM),* pp. 9-16, 2006.

[14] L. Dagum and R. Menon, "Openmp: An Industry-Standard API for Shared-Memory Programming," *IEEE Computer Sci. Eng.,* vol. 5, pp. 46-55, Jan.-Mar. 1998.

[15] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys,* vol. 35, pp. 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," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '01),* pp. 115-126, 2001.

[17] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On Power-Law Relationships of the Internet Topology," *Proc. SIGCOMM,* vol. 29, pp. 251-262, 1999.

[18] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen, "Parallel Event Processing for Content-Based Publish/Subscribe Systems," *Proc. Third ACM Int'l Conf. Distributed Event-Based Systems (DEBS '09)*, pp. 8:1-8:4, 2009.

[19] L. Fiege, G. Mühl, and A.P. Buchmann, "An Architectural Framework for Electronic Commerce Applications," *GI Jahrestagung 2*, pp. 928-938, 2001.

[20] Z. Jerzak and C. Fetzer, "Bloom Filter Based Routing for Content-Based Publish/Subscribe," *Proc. Second Int'l Conf. Distributed Event-Based Systems (DEBS '08)*, pp. 71-81, 2008.

[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," *Proc. Workshop Parallel Programming Patterns (ParaPLoP '10)*, pp. 9:1-9:8, 2010.

[22] C. Krgel, T. Toth, and C. Kerer, "Decentralized Event Correlation for Intrusion Detection," In K. Kim, ed., *Information Security and Cryptology, ICISC*, vol. 2288, pp. 59-95, Springer, 2002.

[23] A. Margara and G. Cugola, "High Performance Content-Based Matching Using GPUs," *Proc. Fifth ACM Int'l Conf. Distributed Event-Based System (DEBS)*, pp. 183-194, 2011.

[24] G. Mühl, L. Fiege, F. Gartner, and A. Buchmann, "Evaluating Advanced Routing Algorithms for Content-Based Publish/ Subscribe Systems," *Proc. IEEE 10th Int'l Symp. Modeling Analysis and Simulation of Computer and Telecomm. System (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*, vol. 26, pp. 80-113, 2007.

[27] M. Sadoghi and H.-A. Jacobsen, "Be-Tree: An Index Structure to Efficiently Match Boolean Expressions over High-Dimensional Discrete Space," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 637-648, 2011.

[28] S. Schneidert, H. Andrade, B. Gedik, K.-L. Wu, and D.S. Nikolopoulos, "Evaluation of Streaming Aggregation on Parallel Hardware Architectures," *Proc. Fourth ACM Int'l Conf. Distributed Event-Based Systems*, pp. 248-257, 2010.

[29] K.H. Tsoi, I. Papagiannis, M. Migliavacca, W. Luk, and P. Pietzuch, "Accelerating Publish/Subscribe Matching on Reconfigurable Supercomputing Platforms," *Proc. Many-Core and Reconfigurable Supercomputing Conf.*, 2010.

[30] T.W. Yan and H. García Molina, "Index Structures for Selective Dissemination of Information under the Boolean Model," *ACM Trans. Database System*, vol. 19, no. 2, pp. 332-364, June 1994.