TOWARDS THE DEPLOYMENT OF FASTFLOW ON DISTRIBUTED VIRTUAL ARCHITECTURES

Sonia Campa
Marco Danelutto
Massimo Torquati

Department of Computer Science
University of Pisa

Horacio González–Vélez
Alina Mădălina Popescu

Cloud Competency Centre
National College of Ireland

KEYWORDS

Parallel Patterns; Algorithmic Skeletons; Language Constructs and Features; Parallel Programming; Distributed Architectures; Performance Analysis; Virtualisation; Cloud Computing

ABSTRACT

In this paper we investigate the deployment of FastFlow applications on multi-core virtual platforms. The overhead introduced by the virtual environment has been measured using a well-known application benchmark both in the sequential and in the FastFlow parallel setting. The overhead introduced for both the sequential and the parallel executions of CPU and memory-intensive applications is in the range of $2-30\%$, while the execution speedup is almost preserved. Additionally, we have ported the FastFlow benchmark to a cloud-based distributed environment in which a task-intensive application has been tested and the performance compared with the corresponding run on a smaller cluster of multi-core machines without virtualisation.

From a parallel programming perspective, we have demonstrated how a unique programming framework based on the structured parallel programming paradigm can cope with very different kind of target architectures without any (or minimal) code intervention.

INTRODUCTION

A declarative description of a parallel activity, a *pattern* focuses on the parallel behaviour of the application rather than on its implementation, expressed in terms of communication channels and hardware/software features of the target architecture.

Patterns can either be:

- 'RISC' type: describe basic and well-assessed patterns of parallelism such as *pipeline, farm, map* and *reduce*; or,
- domain-specific: more significant to the techniques used in a specific domain and express complex parallel computations such as *divide-and-conquer* and *dynamic programming*.

A pattern is implemented via a parallel activity graph (*algorithmic skeleton* or, simply, 'skeleton') which defines a pattern in terms of computational nodes and data and control dependencies among nodes. The overall parallel activity graph of a given application is thus represented by a composition (nesting) of one or more skeletons.

Skeletons can be implemented through higher-order functions, libraries, or syntactic primitives (González-Vélez and Leyton, 2010). In our case, skeletons are implemented through a template library in FastFlow, a C++ parallel programming framework for multi-core platforms (Aldinucci et al., 2013b). Currently, FastFlow supports the execution of skeleton templates on shared memory environments through non-blocking lock-free/fence-free synchronisation mechanisms.

As part of the EU FP7 ParaPhrase project (Hammond et al., 2011), research has been conducted on the distributed (shared-nothing) implementation of skeleton-based applications using FastFlow (Aldinucci et al., 2012a). As FastFlow supports the use of external communication channels from one skeleton graph node to another, FastFlow applications can be executed on different hosts, potentially targeting heterogenous distributed architectures.

Contribution

In this paper, we focus in understanding the behaviour of the FastFlow programming environment with respect to those target architectures for which FastFlow has not been explicitly designed. We have investigated its exploitation on new platforms ranging from single virtual machine to cloud virtual cluster using a base benchmark.

We start with the evaluation of the sequential overhead of the well known matrix multiplication benchmark using different matrix sizes. Then, performance figures have been obtained by executing the parallel version of the application on two different single multi-core machines using the KVM (Kernel Virtual Machine) and Oracle VirtualBox virtual machines respectively. The results have been compared with the ones obtained on the corresponding physical machine without any virtualisation. The overhead measured in a KVM-based virtual environment seems to be predictable and bounded in the range $2-30\%$. The same does not apply, for the considered benchmark and physical platform, to the Oracle VirtualBox environment where we have obtained a much higher overhead.

Since cloud computing elastically exploits resources using location, pay-per-use, and usability constraints, it can potentially represent a very promising deployment platform for FastFlow applications. Employing Amazon EC2, we have run a *farm of farms* (nested farm skeletons) FastFlow application in which the workers of the outer farm are distributed among virtual multi-core nodes of the EC2 cloud. We have then measured scalability and completion time correlated with the number of nodes involved.

Our overall results show that the application performance obtained through the distributed scenarios of FastFlow match similar deployments in the literature, whilst exhibiting substan-

Proceedings 27th European Conference on Modelling and Simulation ©ECMS Webjørn Rekdalsbakken, Robin T. Bye, Houxiang Zhang (Editors)
ISBN: 978-0-9564944-6-7 / ISBN: 978-0-9564944-7-4 (CD)

tially superior portability.

However, the intention of this paper is not to focus on a "yet-another" application that scales, but rather on a programming environment that guarantees the programmer to reach predictable scalability results across platforms. Thus, as a parallel programming environment, FastFlow is able to guarantee the performance boundaries documented in the scientific literature, and, if properly exploited, it can support in this sense not only a single application but a class of them in distributed scenarios.

Moreover, as in the best structured parallel programming tradition, we will demonstrate that an application implemented on top of FastFlow could run on (even extremely) different programming environments without asking for additional efforts to the user programmer: once the distributed/multi-core/virtualized environment has been properly set up, no or minimal adjustments to the code are needed in order to run the application. *To the best of our knowledge,* FastFlow *is the first parallel skeleton programming environment to be efficiently running on clouds and clusters without modification.*

This paper is structured as follows. Firstly, this work critically contrasts the concepts of cloud and virtualisation with those of distributed systems and HPC to substantiate the introduction of a homogenous virtualised environment. Secondly, it provides a background overview of FastFlow, followed by experiments showing the analytical results of our benchmarks. Finally, this paper draws our conclusions and provides future directions to this work.

VIRTUALISATION AND CLOUD COMPUTING

The ICT evolution during years has registered the amalgamation of distinct platforms such as virtualisation, High Performance Computing (HPC), and cloud computing to enable more robust, larger computational environments. Even when a single HPC system contains substantial (physical) computational resources, virtualisation allows the abstraction and efficient utilisation of those physical resources, while cloud computing typically adds flexibility and connectivity to the overall environment (Wang and von Laszewski, 2008; Lonea, 2013).

That is to say, a cloud computing ecosystem is composed of HPC systems or servers, typically virtualised. It provides common, location-independent, online, utility and on-demand services to users. Resource limits, static workload allocation, and on-premise management are the three main differences between traditional virtualisation and cloud computing. Thus, cloud computing extends virtualisation by providing elasticity, dynamic workload allocation using an Application Programming Interface (API), and on-premise and off-premise management (Lonea, 2013).

The on-demand provision of resources in cloud computing increases system reliability and flexibility by delivering more abstract resources and services via a pay-as-you-go formula (Foster et al., 2008; Rings et al., 2009) and, in this way, the delayed allocation of resources is eliminated (Wang and von Laszewski, 2008).

Even if some computational resources are effectively clusters, they are nominally distributed and located in the cloud resource pool, which differentiate cloud from cluster computing (Letaifa et al., 2010; Gong et al., 2010). Another significant distinction between cloud and cluster is the utilisation factor: clusters are mainly used for load balancing and for providing high availability, while cloud computing is used for providing services.

Nevertheless scant research has been conducted to provide seamless structured parallel programming frameworks, which can efficiently scale from tightly-coupled clusters up to clouds, exposing the same programming abstractions (parallel patterns). Typically used for embedding the computational resources within software environments, virtualisation can arguably be used to abstract the instruction sets of parallel software in order to allow—its divided units—to be mapped/remapped onto alternative guest virtual machines in different distributed environments.

In this work, we hypothesise that the efficient use of virtualisation can enable the scaling of pattern-based parallel applications mapped into cloud environments. We have empirically tested this mapping process using hosted and hypervisor virtualisation architectures.

A hosted virtualisation architecture consists of an application (e.g. Oracle VirtualBox, VMware Player, ACE) that runs on top of an operating system to enable the virtualisation layer. In a hypervisor-based architecture, the virtualisation layer is installed directly onto the 'bare metal' environment, which seemingly increases the scalability, robustness, and performance (VMware, 2007).

Nonetheless, virtualisation entails a performance overhead. Substantial research has been devoted to investigate the overhead magnitude in multi-tiered systems using Xen and OpenVZ (Padala et al., 2007); for applications that run in virtual based Xen environment (Cherkasova and Gardner, 2005); for HPC benchmark applications (Tikotekar et al., 2008); and, for parallel applications using a private cloud environment based on Xen hypervisor (Ekanayake and Fox, 2010). Xen-based systems have been more intensely analysed as their architecture is geared towards HPC systems. It contains enhanced scheduling policies, faster access to hardware drivers running first a simple device driver (Cherkasova and Gardner, 2005; Tikotekar et al., 2008; Ekanayake and Fox, 2010). Xen adopts the paravirtualisation solution. Besides paravirtualisation, two others types of CPU virtualisation can be realized: full virtualisation and hardware virtualisation. Full virtualisation combines the execution of binary translation and direct execution, without modifying the user level instructions at the running time, compared with the paravirtualisation technique which is a OS assisted virtualisation. Moreover, the third virtualisation technique is the hardware assisted virtualisation, which can be executed within Intel and AMD processors, that have incorporated virtualisation support (i.e. Intel-VT and AMD-V) (VMware, 2007).

THE FastFlow PROGRAMMING FRAMEWORK

A structured parallel programming environment written in C++ on top of POSIX thread library, FastFlow provides programmers with predefined and customisable patterns such as *task-farm*, *pipeline* and recently also *map* patterns working on streams (Aldinucci et al., 2013a). It has been initially designed and implemented to be efficient in the execution of fine grain parallel applications on general purpose multi-core architectures (Aldinucci et al., 2013b).

Parallel patterns in FastFlow implement structured synchronisation among concurrent entities (graph nodes) via shared memory pointers passed in a consumer-producer fashion. The FastFlow run-time support takes care of all the required synchronisation relating to communication among the different nodes resulting from the compilation of the high level FastFlow pattern(s) used in an application. The entire FastFlow graph describing the application is implemented us-

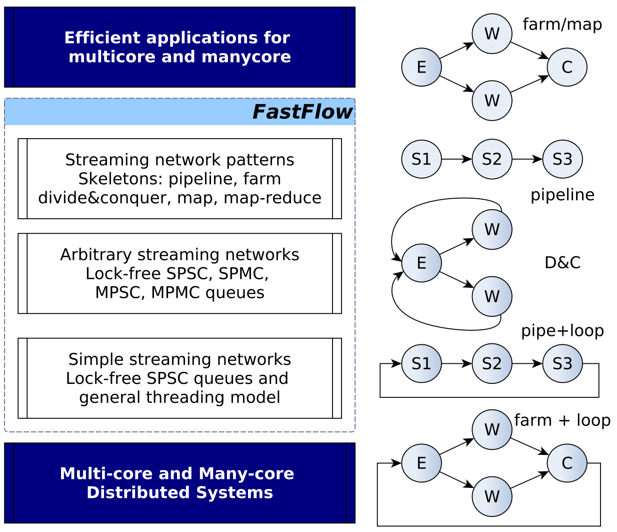
Efficient applications for multicore and manycore
FastFlow
Streaming network patterns
Skeletons: pipeline, farm
divide&conquer, map, map-reduce
Arbitrary streaming networks
Lock-free SPSC, SPMC,
MPSC, MPMC queues
Simple streaming networks
Lock-free SPSC queues and
general threading model
Multi-core and Many-core
Distributed Systems
farm/map
pipeline
D&C
pipe+loop
farm + loop
E
W
C
S1
S2
S3

Fig. 1: Layered FastFlow design together with some possible parallel patterns

```
class ff_node {
protected:
   virtual bool push(void* data) { return qout->push(data);}
   virtual bool pop(void** data) { return qin->pop(data);}
public:
   virtual void* svc(void * task) = 0;
   virtual int    svc_init () { return 0; };
   virtual void   svc_end()  {}
   ...
private:
   SPSC* qin;
   SPSC* qout;
};
```

Fig. 2: FastFlow's `ff_node` class schema

ing non-blocking concurrent threads inside a single process abstraction.

The FastFlow framework provides two abstractions of structured parallel computation: i) the *standalone parallel* mode which basically provides the possibility to write full parallel applications as (compositions of) FastFlow parallel design patterns; ii) the *accelerator* mode, supports the self-offloading of parallel computations from within standard C++ sequential code to a software accelerator programmed as a parallel design pattern composition which is running on the "spare" cores of the architecture. FastFlow is being currently extended to offload data parallel computation to many-core GPGPUs (Goli et al., 2012; Goli and González-Vélez, 2013) and to hardware accelerators (Buono et al., 2013).

The FastFlow design is layered (see Fig. 1). The lower layer implements a lock-free and wait-free Single-Producer, Single-Consumer queue (Aldinucci et al., 2012b). On top of this mechanism, the second layer provides Single-Producer Multiple-Consumers and Multiple-Producers Single-Consumer queues using arbiter threads. This abstraction is designed in such a way that arbitrary networks of activities can be expressed while maintaining the high efficiency of the synchronisation. Eventually, the third layer provides, in the form of standard C++ classes, parallel programming patterns. The possibility to efficiently handle both stream parallel and data parallel computations using the same programming model represents an advantage of FastFlow with respect to other frameworks that only (efficiently) support either stream or data parallel computations.

The key concept in the implementation of FastFlow is the `ff_node` class. It is used to encapsulate sequential portions of code implementing functions as well as higher level parallel patterns such as pipelines and farms. The `ff_node` class structure is outlined in Fig. 2.

Predefined FastFlow patterns may be arbitrarily nested, and so we can have pipelines with farm stages and vice versa. Also, the policies used to schedule the input tasks–as well as portions/partitions of the input tasks–to farm workers, and also to gather results from workers onto the farm output queue may be customised via C++ overloading of methods implementing standard policies (round-robin and on-demand). Using the customisation features, different patterns may be implemented in terms of the pipe and farm building blocks.

Recently FastFlow has been extended to target also loosely-coupled distributed systems (Aldinucci et al., 2012a), thus providing the user with a two-tier programming model. At a *lower tier*, a shared-memory implementation of skeletons inside a single multi-core workstation; at an *upper tier*, structured coordination among a set of distributed nodes executing the lower tier computations. More specifically, at the lower tier the user designs a typical FastFlow skeleton graph, employing stream parallelism and the shared memory skeletons offered by the original FastFlow framework. Multiple lower tier FastFlow graphs can be connected together using the mechanisms of the upper tier, that is, using a suitable communication pattern which implements a network channel (i.e. point-to-point, broadcast, scatter, etc.). At this level, the programming model exposed to the programmer can be either SPMD or MPMD.

In order to send and receive tasks from and to other FastFlow graphs, the edge-nodes of the FastFlow application have to be defined as `ff_dnode`. A `ff_dnode` is actually a `ff_node` with an extra communication channel (henceforth *external channel*) which connects the edge-node of the graph with one or more edge nodes of other FastFlow application graphs running on the same or on a different host.

At the second tier no memory is shared among processes and so all iterations have to be implemented using explicit communications, which are the responsibility of the FastFlow run-time support and are completely transparent to the user (see Fig. 3).

External channels are implemented using the ZeroMQ (Aldinucci et al., 2012a) messaging framework. The ease-of-use of ZeroMQ together with the asynchronous communication model offered were the key factors for choosing ZeroMQ for the implementation. The communication patterns currently implemented, are summarised in the following:

unicast	unidirectional point-to-point communication between two peers
broadcast	sends the same input data to all connected peers
scatter	sends different parts of the input data (typically partitions) to all connected peers
onDemand	the input data is sent to one of the connected peers, the choice of which is taken at run-time on the basis of the actual work-load (typically it is implemented using a request-reply protocol)
fromAll	(also known as all-gather) collects different parts of the data from all connected peers combining them in a single data item
fromAny	collects one data item from one of the connected peers

Nonetheless, scant research has been conducted to substantiate the seamless portability of distributed FastFlow applications across platforms, from local clusters and multi-node virtualised environments to fully-virtualised public cloud infrastructures.

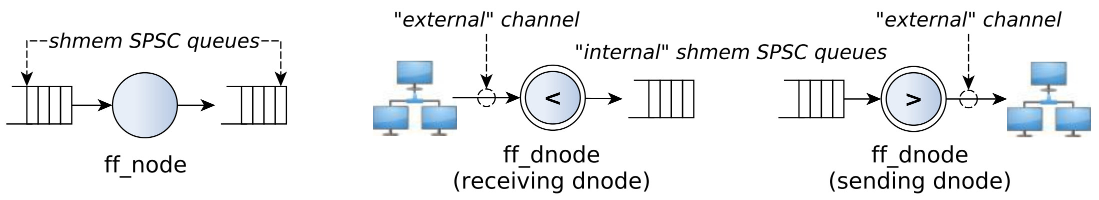

Fig. 3: FastFlow's node vs receiving and sending dnode(s).

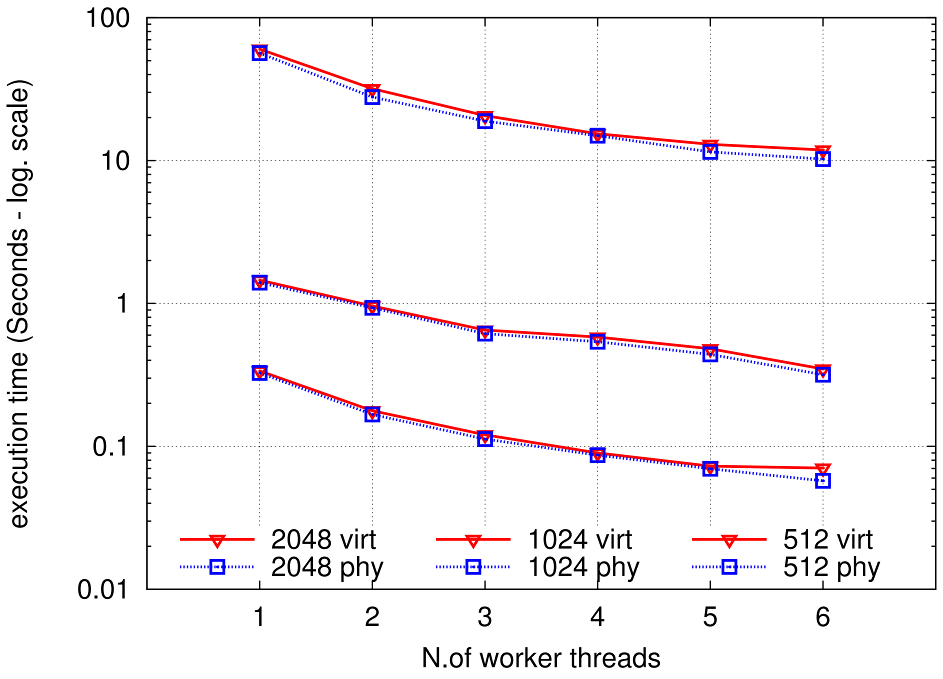

Fig. 4: Virtual vs. Physical execution of the FastFlow MatMul implementation: completion time varying the number of worker threads.

EXPERIMENTS

In this section we have first measured the overhead of running sequential and parallel CPU-intensive applications on virtual multi-core machines, then we extend our tests considering also a cluster of virtual multi-core machines running in a public cloud environment.

The sequential application considered is a simple matrix multiplication algorithm (MatMul). We tested two versions of the sequential algorithm: the standard naive algorithm and a cache oblivious version which computes matrix elements (double precision) in a different order (`for(i) for(j) for(k) C[j][k] += A[j][i]*B[i][k].`).

First we consider a Linux KVM-based virtualisation environment as part of a Eucalyptus private cloud computing-based architecture (Nurmi et al., 2009; Lonea et al., 2012).

The physical machine used is a Linux CentOS x86_64 and has two 6-core CPUs Intel Xeon E5-2540 2.5 GHz with 8MB L3 cache. We run the sequential algorithm both on the physical and virtual machine varying the matrix size. The virtual machine runs the same OS and has 6 virtual cores. The overhead measured ranges between 6% and 27% for the size tested (see Tab. I). We observed an higher overhead for the cache oblivious version of the algorithm since the absolute time measured are much lower w.r.t. the ones obtained using the standard algorithm thus the overhead as a greater incidence.

Then, we considered the execution of a parallel version of the matrix multiplication algorithm on both physical and virtual machines. The parallel version is designed to use the FastFlow accelerator feature using the task-farm skeleton as described in (Aldinucci et al., 2011). In order to make the results comparable, when the parallel application is executed on the physical machine, the threads of the farm skeleton have been forced to be executed in the same 6 physical cores used by the 6 virtual cores of the virtual machine (We used the *taskset* command to set process's CPU affinity).

The completion time obtained varying the number of worker threads are shown in Fig. 4. Table II reports the maximum over-

TABLE I: Overhead between physical and virtual execution for the sequential MatMul algorithm.

	matrix size		
Seq. MatMul	512	1024	2048
Standard Algorithm	5.96%	6.01%	7.8%
C. Oblivious Algo.	27.43%	11.52%	10.29%

TABLE II: Virtual vs. Physical execution of the FastFlow MatMul implementation: maximum overhead obtained.

	matrix size		
Par. MatMul	512	1024	2048
Max. Overhead	21%	10%	16%

head observed among all runs. As it can be seen, the trend of the execution time is almost the same in all tested cases. We performed other tests (not reported in this paper for the sake of conciseness) for different matrix sizes in the range 384 and 3072. In all cases, the overhead introduced by the KVM virtual environment ranges between 2% and 30% maximum.

The same kind of experiments—both sequential and parallel—have been also run on a different virtual and physical environment. We have considered the Oracle VirtualBox application environment (version 4.0.10) installed on as physical system with 2 CPUs Intel Sandy Bridge Xeon E5-2650 2.0GHz. In this case, the measured overhead for both versions of the algorithms is much higher when compared with the one related to the KVM environment. The parallel speedup trend of the physical and virtual environment is almost the same also in this case. The higher differences are related to the fact that the VirtualBox environment is less integrated with the low-level Linux host system, the VirtualBox environment running as a standard application on the host OS.

As expected, executing CPU and memory intensive application on a virtual environment introduces non-negligible overhead. However, when optimised virtualisation environments are used, such overhead is somehow predictable within a bounded range allowing to know in advance the maximum overhead that will be introduced in the execution.

We wanted to evaluate if the above considerations are still valid when considering distributed application running on cluster of virtual multi-core machines in a public cloud platform. We have therefore considered the Amazon EC2 public cloud, where we deployed 7 Linux Ubuntu 12.04 LTS x86_64 virtual machines: one VM with 8 cores and 6 VMs with 4 cores. The VMs with 4 cores has an Intel CPU E-2670 with 4 virtual core 2.6 GHz with 20MB of L3 cache and 14GB of RAM.

The application taken into account is the one sketched in Fig. 5, which is actually a 3-stage FastFlow distributed pipeline. The first stage generates a stream of square matrices, the second stage computes, on each input, the matrix multiplication with a constant local matrix. The second stage is internally parallel implemented using the FastFlow task-farm pattern so that each worker thread computes an entire matrix multiplication sequentially using the cache-oblivious algorithm.

The last stage collects the results sending them back to the first stage of the distributed pipeline. The communication patterns that link the second stage with the first and with the third are the *onDemand* and the *fromAny* patterns, respectively. They allow replication of the second stage without touching the code for a given number of times, so that its throughput can be increased.

The three involved stages are mapped on the virtual cluster using the following schema: the first and the third stages are mapped in one 8-cores VM whereas the middle stages are mapped each one in a separated 4-cores VM. This mapping is sub-optimal w.r.t. the single node available bandwidth, but allow us to maximise the number of middle stages.

Even in this case we tested three different size of matrices: 512×512, 1024×1024 and 2048×2048 (double elements). In Fig. 7 are shown 3 performance metrics obtained varying the number of middle stages when the 1024×1024 matrix size is considered. The scalability is almost linear up to 7 nodes, whereas the time speedup is more than linear since each node is internally parallel (4 worker threads are used for each middle stage). The maximum time speedup obtained is thus $\sim 8\times$ when compared to the overall sequential time of the application.

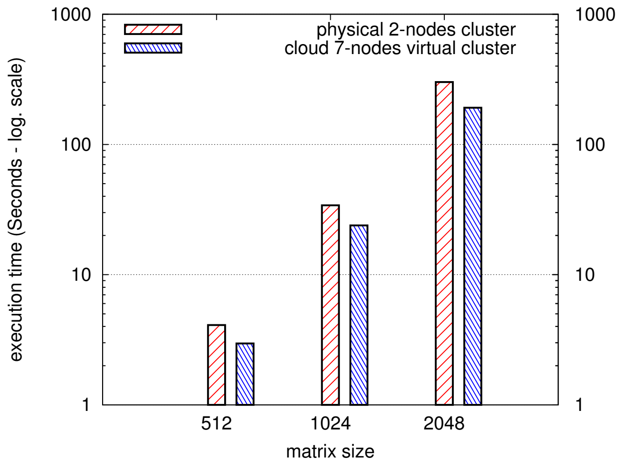

Fig. 6: The minimum completion time obtained running the application in the 32 cores EC2 virtual machines vs. 32 cores physical cluster.

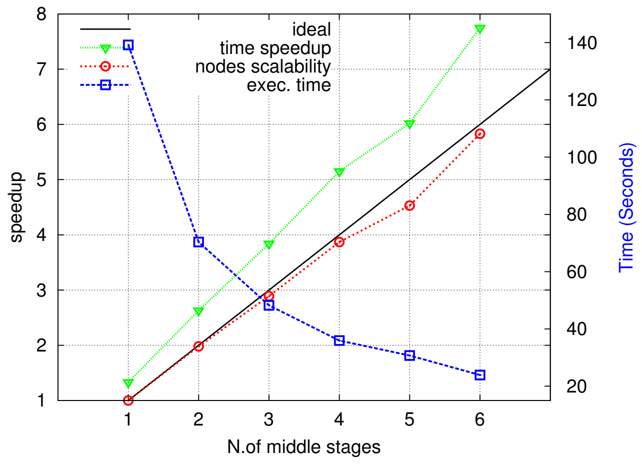

Fig. 7: Completion time, time speedup and node scalability varying the number of middle stages (Number of matrices: 128. Matrix size 1024×1024)

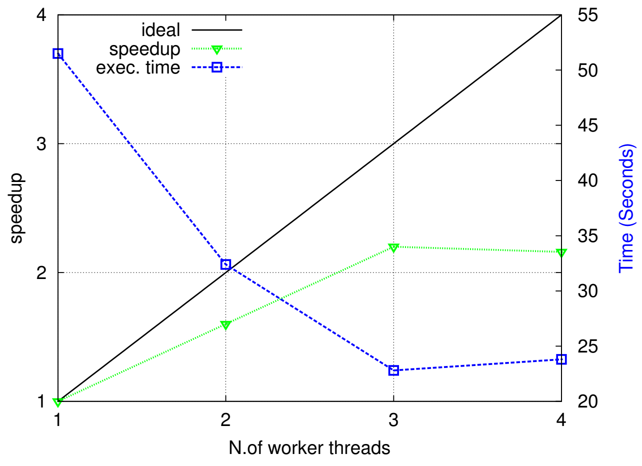

Fig. 8: Completion time and time speedup varying the number of worker threads used inside each middle stage (Number of matrices: 128. Matrix size 1024×1024).

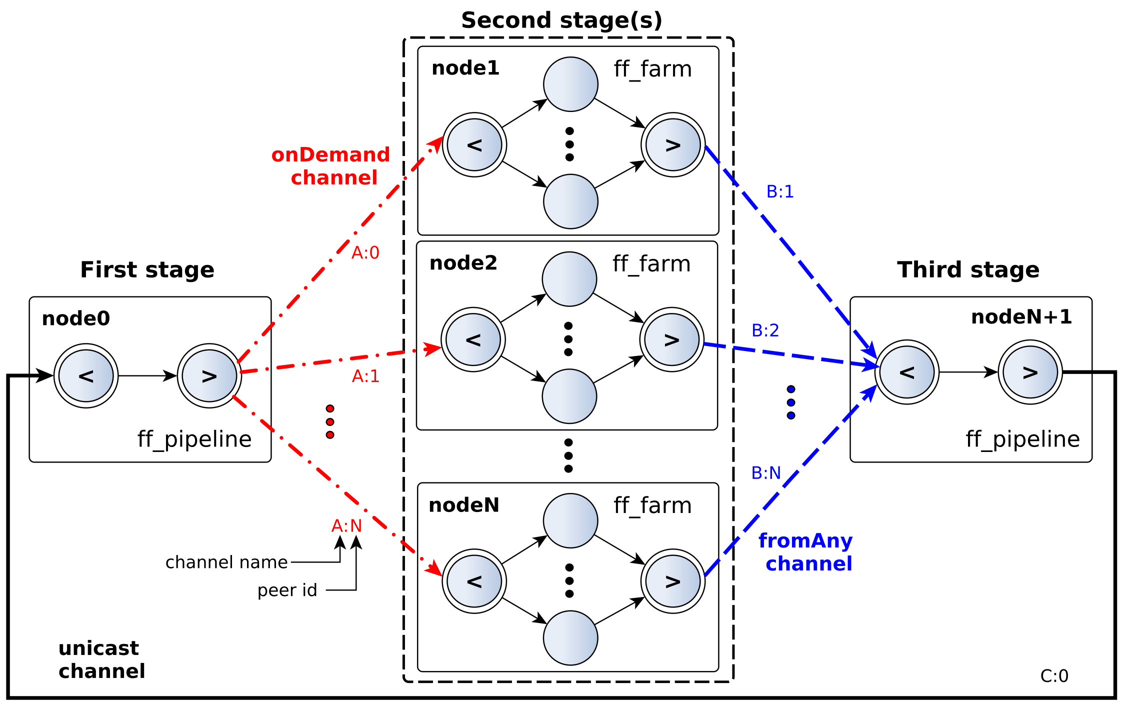

Fig. 5: The schema of the 3-stage pipeline application tested

In Fig. 8 we reported the completion time and the speedup obtained for the case 1024×1024 when 6 middle stages are used and the internal parallelism of each stage is varied between 1 to 4. In this case the performance increase is not particularly good because the first pipeline stage is not able to feed with enough matrices all middle parallel nodes due to the limited available network bandwidth ($\sim$1 Gbit/s bidirectional bandwidth per virtual cluster node).

Finally, in Fig. 6, it is sketched the minimum completion time we obtained running the same 3-stage pipeline in the Amazon EC2 cloud and in a two node physical cluster each one with 2 Intel Sandy Bridge Xeon E5-2650 @2.0GHz with 8 cores each and 20MB L3 shared cache. The physical nodes are connected with an Infiniband Connectx-3 card (40 Gb/s ports). As can be seen using more nodes (and a lower number of cores) we are able to obtain a performance increase from 41% to 57% without additional effort at the application level thanks to the skeleton-based approach used in the implementation.

CONCLUSION

In this paper we have deployed and tested the FastFlow parallel framework on virtual multi-core machines and virtualised public cloud infrastructures. Our experimental evaluation of a distributed application running in the virtual Amazon EC2 public cloud, shows that the FastFlow design is flexible and robust also for such environments. We have demonstrated that applications implemented using FastFlow can run on different parallel and distributed virtualised platforms without additional programming efforts with good performance.

As previously reported in literature, virtualised execution of sequential and parallel applications introduces performance overhead that is affected by many factors. Our tests in a KVM-based virtual environment using a simple micro-benchmark confirm a non-negligible overhead. When measured in both sequential and parallel executions, a performance decrease in the range 2-30% has been observed.

Analogous to the tests performed in Ekanayake and Fox (2010) which use an Eucalyptus Xen-based cloud platform, a part of our tests has been realised in an Eucalyptus KVM-based cloud platform. We have analysed the computational overhead between the local node where the FastFlow-based VM is running and the FastFlow-based VM, using the same hardware and software configuration.

Thus, our tests have dealt with a single FastFlow-based VM in the Eucalyptus private cloud, which was configured to have 6 virtual cores. Unlike our tests, Ekanayake and Fox (2010) have preferred to observe the overhead between the bare-metal and one to multiple VMs, where parallel applications are running as well. They have concluded that increasing the number of VMs per cloud node produces a higher overhead. Likewise, the minimum overhead reported by Ekanayake and Fox (2010) for one VM per hardware node is 8%, while in our results is 6%.

Thus, for both cases the overhead is small and this difference can be explained by the size of the application, by the application itself and by the impact produced by the Xen hypervisor, which interact with the cores of processor using two domains (i.e. Dom 0 and Dom U). Our experimental protocol have extended the performance analysis to include the sequential version of the matrix multiplication application. Both sequential and parallel applications have been tested across different virtual and physical environments.

ACKNOWLEDGEMENTS

This work is based upon research supported as part of ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems `http://paraphrase-ict.eu/`, a research project funded by the European Commission Seventh Framework Programme under contract no.: 288570.

References

Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M., 2012a. Targeting distributed systems in FastFlow, in: Euro-Par 2012 Workshops, Springer-Verlag, Rhodes. pp. 47–56.

Aldinucci, M., Campa, S., Kilpatrick, P., Torquati, M., 2013a. Structured data access annotations for massively parallel computations, in: Euro-Par 2012 Workshops, Springer-Verlag, Rhodes. pp. 381–390.

Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M., 2011. Accelerating code on multi-cores with FastFlow, in: Euro-Par 2011, Springer-Verlag, Bordeaux. pp. 170–181.

Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M., 2012b. An efficient unbounded lock-free queue for multi-core systems, in: Euro-Par 2012, Springer-Verlag, Rhodes. pp. 662–673.

Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M., 2013b. FastFlow: high-level and efficient streaming on multi-core, in: Programming Multi-core and Many-core Computing Systems. Wiley. Parallel and Distributed Computing. chapter 13, p. 13.

Buono, D., Danelutto, M., Lametti, S., Torquati, M., 2013. Parallel patterns for general purpose many-core, in: Euromicro PDP 2013, IEEE, Belfast. pp. 131–139.

Cherkasova, L., Gardner, R., 2005. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor, in: ATEC '05, USENIX, Anaheim. pp. 24–.

Ekanayake, J., Fox, G., 2010. High performance parallel computing with clouds and cloud technologies, in: Cloud Computing. Springer-Verlag. volume 34 of *LNICST*, pp. 20–38.

Foster, I., Zhao, Y., Raicu, I., Lu, S., 2008. Cloud computing and grid computing 360-degree compared, in: GCE'08/SC'08, ACM/IEEE, Austin. pp. 1–10.

Goli, M., Garba, M.T., González-Vélez, H., 2012. Streaming dynamic coarse-grained CPU/GPU workloads with heterogeneous pipelines in FastFlow, in: HPCC-12, IEEE, Liverpool. pp. 445–452.

Goli, M., González-Vélez, H., 2013. Heterogeneous algorithmic skeletons for FastFlow with seamless coordination over hybrid architectures, in: Euromicro PDP 2013, IEEE, Belfast. pp. 148–156.

Gong, C., Liu, J., Zhang, Q., Chen, H., Gong, Z., 2010. The characteristics of cloud computing, in: ICPPW-2010, San Diego. pp. 275–279.

González-Vélez, H., Leyton, M., 2010. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. Software–Practice & Experience 40, 1135–1160.

Hammond, K., Aldinucci, M., Brown, C., Cesarini, F., Danelutto, M., González-Vélez, H., Kilpatrick, P., Keller, R., Rossbory, M., Shainer, G., 2011. The ParaPhrase project: Parallel patterns for adaptive heterogeneous multicore systems, in: FMCO 2011, Springer-Verlag, Turin. pp. 218–236.

Letaifa, A.B., Haji, A., Jebalia, M., et al., 2010. State of the art and research challenges of new services architecture technologies: Virtualization, soa and cloud computing. International Journal of Grid and Distributed Computing 3, 69–88.

Lonea, A.M., 2013. Private cloud set up using Eucalyptus open source, in: Soft Computing Applications. Springer-Verlag. volume 195 of *Advances in Intelligent Systems and Computing*, pp. 381–389.

Lonea, A.M., Popescu, D.E., Prostean, O., 2012. A survey of management interfaces for Eucalyptus cloud, in: SACI 2012, IEEE, Timisoara. pp. 261–266.

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D., 2009. The Eucalyptus Open-Source Cloud-Computing System, in: CCGRID'09, IEEE, Shanghai. pp. 124–131.

Padala, P., Zhu, X., Wang, Z., Singhal, S., Shin, K., 2007. Performance evaluation of virtualization technologies for server consolidation. Technical Report. HP Laboratories Palo Alto.

Rings, T., Caryer, G., Gallop, J., Grabowski, J., Kovacikova, T., Schulz, S., Stokes-Rees, I., 2009. Grid and Cloud Computing: Opportunities for Integration with the Next Generation Network. Journal of Grid Computing 7, 375–393.

Tikotekar, A., Vallée, G., Naughton, T., Ong, H., Engelmann, C., Scott, S.L., 2008. An analysis of HPC benchmarks in virtual machine environments, in: Euro-Par 2008 Workshops, Springer-Verlag. pp. 63–71.

VMware, 2007. Understanding full virtualization, paravirtualization and hardware assist. white paper.

Wang, L., von Laszewski, G., 2008. Scientific cloud computing: Early definition and experience, in: HPCC-08, IEEE, Dalian. pp. 825–830.

AUTHOR BIOGRAPHIES

Sonia Campa is a Research Fellow of the Department of Computer Science at the University of Pisa. Her main fields of investigation regard models and languages for parallel computation description, with particular emphasis to structured parallel programming environments which she has approached at different levels of abstraction, from formal frameworks to libraries implementations. Email: `campa@di.unipi.it`

Marco Danelutto graduated *cum laudae* from University of Pisa in 1984, where he also got his PhD in Computer Science in 1990. Since 1998, he is an Associate Professor with the Dept. of Computer Science at University of Pisa. His research interests include models and tools for parallel programming, algorithmic skeletons, parallel design patterns, and (parallel) functional programming. Email: `marcod@di.unipi.it`

Massimo Torquati is currently a Research Fellow at the Computer Science Department of the University of Pisa. His main research interests include language, tools and model for parallel and distributed computing. Email: `torquati@di.unipi.it`

Horacio González–Vélez is an Associate Professor and Head of the Cloud Competency Centre at the National College of Ireland. His research interests lie in HPC, cloud computing, and structured parallelism, and their application to the solution of complex computational problems. He earned a PhD in Informatics from the University of Edinburgh. Email: `horacio@ncirl.ie`

Alina Madalina Popescu is a Research Lecturer with the Cloud Competency Centre at the National College of Ireland. She has significant background in Cloud Computing, as a result of her PhD thesis covering the topic of "Security Solutions for Cloud Computing," which was successfully defended at Politehnica University of Timisoara. Dr. Popescu is interested in researching distinct parallel patterns by defining their component states, life-cycle and interfaces to heterogeneous hardware devices from multi-core to cloud computing environment. Email: `Alina-Madalina.Popescu@ncirl.ie`