Targeting Distributed Systems in FastFlow[⋆]

Marco Aldinucci[1], Sonia Campa[2], Marco Danelutto[2]
Peter Kilpatrick[3], and Massimo Torquati[2]

[1] Computer Science Department, University of Torino, Italy
[2] Computer Science Department, University of Pisa, Italy
[3] Computer Science Department, Queen's University Belfast, UK

Abstract. FastFlow is a structured parallel programming framework targeting shared memory multi-core architectures. In this paper we introduce a FastFlow extension aimed at supporting a network of multi-core workstation as well. The extension supports the execution of FastFlow programs by coordinating–in a structured way–the fine grain parallel activities running on a single workstation. We discuss the design and the implementation of this extension presenting preliminary experimental results validating it on state-of-the-art networked multi-core nodes.

Keywords: structured parallel programming, multi-core, fine grain

1 Introduction

In a scenario with more and more cores per socket available to the application programmer it is becoming increasingly urgent to provide programmers with effective parallel programming tools. Programming tools and frameworks are needed to efficiently target the architectures hosting inter networked, possibly heterogeneous, multi-core devices, which appear to be "the" reference architecture ferrying programmers from the mainly sequential to mainly parallel programming era [1]. The urgency is even more crucial given that both grids and clouds provide application programmers with the possibility to reserve collections of multi-core to support parallel applications eventually presented as, or orchestrated by, web services.

Shared memory multi-core and clusters/networks of processing elements, however, require quite different techniques and tools to support efficient parallelism exploitation. The *de facto* standard tools in the two cases are OpenMP [2] and MPI [3] used either alone or in conjunction. Despite being very efficient on some classes of applications, OpenMP and MPI share a common set of problems: poor separation of concerns among application and system aspects, a rather low level of abstraction presented to the application programmers and poor support for really fine grained applications are all considerations hindering easy use of MPI and OpenMP. Actually, it is not even clear yet if the mixed MPI/OpenMP programming model always offers the most effective mechanisms for programming clusters of SMP systems [4].

⋆ This work has been partially supported by ParaPhrase (`www.paraphrase-ict.eu`)

The algorithmic skeleton community has proposed various programming frameworks aimed at providing the application programmer with very high level abstractions completely encapsulating parallelism exploitation patterns and solving most of the problems mentioned above [5,6]. Initial skeleton based programming frameworks targeted only cluster/network of workstations. More recently, some of the existing frameworks have been extended in such a way clusters of multi-core may also be exploited. SkeTo [7] provides data parallel skeletons as proper C++ abstractions. It has been recently extended to target multi-core clusters using a two-step dynamic task scheduling strategy, which enables balancing of the load both between nodes of the cluster and cores of the single node [8]. The Muesli programming framework is provided as a C++ library offering both data and stream parallel skeletons [9]. The original version of Muesli generated code for MPI platforms. Recently, Muesli has been extended in such a way that both multi-core architectures and distributed multi-core workstations may be targeted by generating OpenMP code in addition to the MPI code [10]

The contribution of this paper is twofold. First, we discuss an extension of FastFlow targeting clusters of multi-core workstations. The extended version supports a two tier parallel model with a lower tier exploiting fine grain parallelism on single multi/many core workstation and the upper layer supporting structured coordination–across internetworked workstations–of medium/coarse grain parallel activities. Second, we present experimental results validating the proposed approach and showing: i) how real applications may be structured using the proposed model, ii) how different interconnection networks may be seamlessly exploited depending on the communication bandwidth required by the application, and iii) how our programming framework is able to fully exploit state-of-the-art architectures.

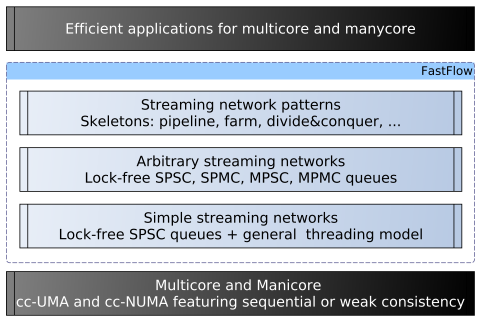
2 The FastFlow programming framework

FastFlow[4] is a structured parallel programming environment implemented in C++ on top of Pthreads library and targeting shared memory multi-core. FastFlow provides programmers with predefined and customizable task farms and pipelines parallel patterns. It has been initially designed and implemented to be very efficient in the execution of fine grain parallel applications [11].

FastFlow design is layered (see Fig. 1). The lower layer implements a very efficient, lock free and wait free single producer, single consumer queue [12]. On top of this mechanism, the second layer provides single producer multiple consumer and multiple producer single consumer queues. Eventually, the third layer provides the *farm* and *pipeline* parallel patterns as C++ classes [13].

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. 1. Each `ff_node` will be used to run a concurrent

[4] Project site `http://sourceforge.net/projects/mc-fastflow/`


Efficient applications for multicore and manycore
FastFlow
Streaming network patterns
Skeletons: pipeline, farm, divide&conquer, ...
Arbitrary streaming networks
Lock-free SPSC, SPMC, MPSC, MPMC queues
Simple streaming networks
Lock-free SPSC queues + general threading model
Multicore and Manicore
cc-UMA and cc-NUMA featuring sequential or weak consistency

```
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. 1. Left: Layered FastFlow design. Right: FastFlow's `ff_node` class schema

activity in a thread, and it has associated two (shared memory) message queues: one used to receive input data (pointers) to be processed and one to deliver the (pointers to) computed results. The `svc` method encapsulates the computation to be performed on each input datum to obtain the output result. `svc_init` and `svc_end` methods are executed when the application is started end before it is terminated. The three methods constitute the only thing the programmer has to provide to instantiate an `ff_node`.

The predefined pattern provided by FastFlow may be customized in different ways. For example default patterns may be arbitrarily nested and so we can have pipelines with farm stages and vice versa. Using the customization features, different patterns may be implemented in terms of the pipe and farm building blocks, such as divide&conquer, map and MISD[5] pattern.

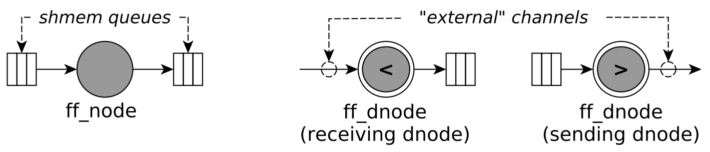
FastFlow is being currently extended to also support data parallel patterns and to offload data parallel computation to GPUs, where available.

3 From single to multiple nodes

FastFlow was originally designed to target efficiently shared-cache multi-core platforms and only those platforms. The FastFlow stream semantics guarantees correct sequencing of activation of the concurrent activities modeled through `ff_node`s and connected through streams. The stream implementation ensures *pure data flow* semantics.

In order to scale to thousands of cores in terms of computational power, or to be able to use huge amounts of memory, the only viable solution is to exploit more and more multi-core workstations together. To this end, the streaming network model provided by FastFlow to describe concurrent activities, turns out to be suitable to be used also in loosely-coupled systems such as clusters, cloud and grids. The idea has been therefore to extend the FastFlow framework to provide the user with a two-tier programming model:

[5] different computations on the same input, providing a vector result (each position hosting the result of one computation)



```
template<typename CommImpl>
class ff_dnode: public ff_node {
protected:
 bool
 push(void* data){... return com.put(data);}
 bool
 pop(void** data){... return com.get(data);}
public:
 int
 init(std::string& name,std::string& address,
  int peers,CommImpl::TransportImpl* transp,
  bool p,int nodeId,dnode_cbk_t cbk=0) {
     ...  return com.init(address,nodeId);
 }
 // serialization/deserialization methods
 // used by the sender dnode
 virtual void
 prepare(svector<iovec>& v,void* ptr);

 // used by the receiver dnode
 virtual void
 prepare(svector<msg_t*>*& v,size_t len);
 virtual void
 unmarshalling(svector<msg_t*>* v[],
  int vlen,void*& task);
private:
  CommImpl com;
};
```

Fig. 2. Top)FastFlow's node vs dnode(s). Bottom) FastFlow's `dnode` class schema.

– at a *lower tier*, a shared-memory implementation of skeletons inside a single multi-core workstation is supported;
– at an *upper tier*, structured coordination among a set of distributed nodes executing the lower tier computations is supported, by providing all the mechanisms needed to implement the low tier skeletons in the distributed multi-core scenario.

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. Parallel patterns implement structured synchronization among concurrent entities (graph's nodes) via shared memory pointers passed in a consumer-producer fashion. The FastFlow run time support takes care of all synchronization needed and related to communication among the different nodes resulting from the compilation of the high level FastFlow pattern(s) used in an application. At this level, the entire FastFlow graph describing the application is implemented using non-blocking concurrent threads inside a single process abstraction. Then, multiple lower tier FastFlow graphs can be connected together using the mechanisms of the second 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 (from now on we refer to it as *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, thus all iterations have to be implemented using explicit communications, which are the responsibility of the FastFlow run-time support and completely transparent to the user due to the careful design of the `ff_dnode`.

As shown at the top of Fig. 2, a FastFlow node is a concurrent activity with an input and an output channel that is implemented using concurrent queues, whereas a `ff_dnode` is a concurrent activity where one of the 2 channels (either the input or the output one) is an external channel. A `ff_dnode` cannot have both external input and output channels at the same time since the minimal pure FastFlow application is composed of at least 2 nodes (a pipeline of two sequential nodes or a farm with an Emitter node and a sequential worker node). The interface of the `ff_dnode` class is sketched at the bottom of Fig. 2. The template parameter (`CommImpl`) represents the type of the communication pattern that the programmer wishes to use to connect different `ff_dnode`s. The `ff_dnode::init` method has to be called in order to initialize the external channel and it is typically called within the `ff_node::svc_init` method in order to perform parallel initialization among multiple `ff_dnode`[6]. If the `ff_dnode::init` method is not called the dnode behaves like a pure FastFlow node. The `ff_dnode` class overwrites the *pop* and *push* methods of the `ff_node` class so that they work on external input and output channels if activated by the `ff_dnode::init` method.

ZeroMQ as external transport layer. ZeroMQ is an LGPL open-source communication library [14]. It provides the user with a socket layer that carries whole messages across various transports: inter-thread communications, inter-process communications, TCP/IP and multicast sockets. ZeroMQ offers an asynchronous communication model, which allows construction of complex asynchronous message-passing networks very quickly and with reasonable performance. The message API offers the possibility to perform zero-copy sends and and non-blocking calls to the socket layer.

In FastFlow, we used ZeroMQ as the external transport for the `ff_dnode` concurrent entity. In particular, we built on top of ZeroMQ all the communication patterns using the `DEALER` and the `ROUTER` sockets offered by the library. The `ROUTER` socket allows routing of messages to specific connections provided that the peer identifier is known; the `DEALER` socket instead can be used for fair-queuing on input and for performing load-balancing on output toward a pool of connections. Within FastFlow we do not use the load-balancing feature; instead we use it to connect a `ff_dnode` to a `ROUTER` socket of another `ff_dnode`. The ease-of-use of ZeroMQ was the factor for choosing it for the implementation of the distributed transport layer.

Communication patterns among ff_dnode(s). A `dnode`'s external channel can be specialized to provide different patterns of communication. The set of communication collectives, allows exchange of messages among a set of distributed nodes using well-known predefined patterns. The semantics of each communication pattern currently implemented, may be summarized as follows:

[6] The `svc_init` method is called once when the node thread has already started

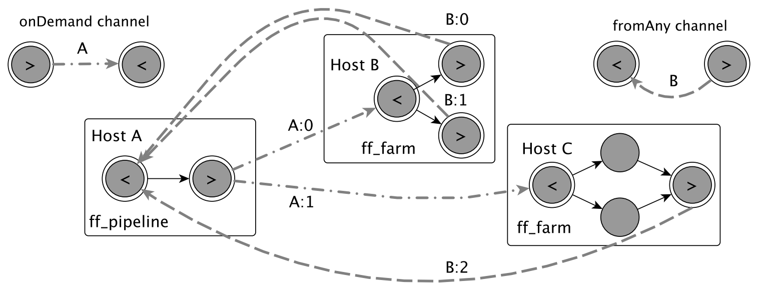


Fig. 3. An example application schema that represents 3 FastFlow applications connected through one *onDemand* and one *fromAny* communication pattern.

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

Each communication pattern has a unique identifier (the channel-name). Each `ff_dnode` has an identifier (not necessarily unique) for the external channel in the range `[0..num-peers[` so that the tag "`channel-name:dnode-id`" is used as a unique identity for the peer connection. Fig. 3 sketches a possible interaction among different FastFlow applications (one Pipeline on the HostA, and two different Farm skeletons on the HostB and on the HostC) connected via an *onDemand* communication pattern, whose channel name is "A", used to feed the 2 farms with input tasks, and a *fromAny* pattern, whose channel name is "B", used to collect back to the HostC the results in a non-deterministic order.

Marshalling and unmarshalling. The `ff_dnode` class provides the user with suitable methods that can be overwritten in order to manage zero-copy marshalling and unmarshalling of data. A `prepare` method allows to serialize non-adjacent data on the sending side, while a `prepare` and a `unmarshalling` methods at the receiving side allow to deserialize the received data in properly castable buffers. We do not give more details on this, due to space limitations.

4 Experiments

All experiments have been run on 2 workstations running Linux x86_64 (named HostA and HostB) equipped with 2 CPUs Intel Sandy Bridge Xeon E5-2650 @2.0GHz. Each CPU has 8 cores double context, 20MB L3 shared cache and

size	8B	8KB	1MB
	Latency (μs)	Bandwidth (Gb/s)	
Eth	69	0.93	0.95
Inf	27	5.1	14.7

(Latency measured using a torus of 2 FastFlow 2-stage pipelines whose edge-nodes are connected with a unicast channel)

Fig. 4. Latency and bandwidth of the *unicast* channel for both 1Gb Ethernet and IP over IB Infiniband networks.

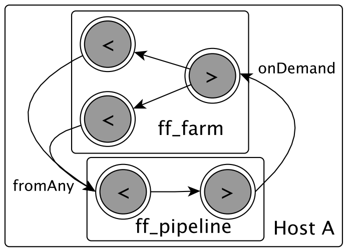
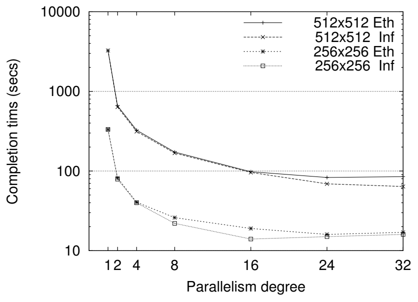


Fig. 5. Parallel schema of the benchmark test when executed on a single host.

32 GBytes of memory. The two workstations are connected both with a Gigabit Ethernet and an Infiniband Connectx-3 card (40 Gb/s ports) using a direct cable. Since ZeroMQ does not have any native support for Infiniband, we used the IP over IB driver. We used two workstations to be able to precisely evaluate the latency and throughput of our FastFlow extension.

We firstly measured the raw performance of the new *unicast* channel (see Fig. 4). The maximum bandwidth obtained running the `iperf` benchmark tool (v.2.0.5)[7] over the Infiniband network using TCP/IP is 4.8 Gb/s and 17.6 Gb/s for 8KB and 1MB messages respectively. This means that the *unicast* channel does not introduce any significant overhead and is able to saturate almost all the available network bandwidth.



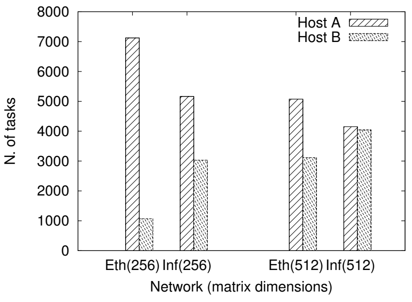
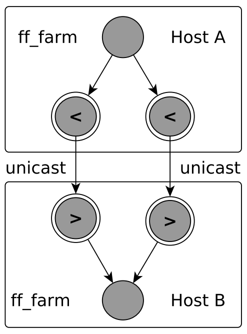


Fig. 6. Benchmark test executed on the two hosts using a stream of 256×256 and 512×512 matrices with the 1Gb Ethernet (Eth) and the Infiniband (Inf) networks. Left: Completion time in seconds. Right: Number of tasks computed in the 2 hosts.

[7] http://sourceforge.net/projects/iperf



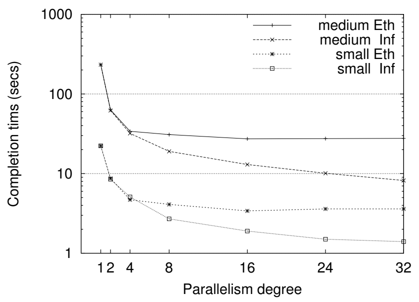


Fig. 7. Image filter application. Left: Application schema. Right: Completion time for small and medium sized images on the 2 networks.

mat. size	FF	dFF-1	dFF-2-Eth	dFF-2-Inf
256×256	24.4	18.9	16	14
512×512	203	158	83	64

img. size	FF	dFF-2-Eth	dFF-2-Inf
small	2.4	3.4	1.4
medium	19.2	27.2	8.2

Table 1. Left: Benchmark test (see Fig. 5 for the dFF-1 schema). Right: Image filter application (see Fig. 7 left for dFF-2 schema). Completion time in seconds obtained running different implementation.

In order to test the impact of computation grain and communication bandwidth and latency, we used a synthetic application computing the square of a stream of input matrices. We ran two experiments: one using 256×256 and the other using 512×512 matrices of doubles. We set the stream length to 8192 matrices for a total of 4GB and 16GB of input data, respectively. The parallelization schema adopted is very similar to that sketched in Fig. 3, where 2 FastFlow farms (without the collector) and one 2-stage pipeline are used. The first stage of the pipeline generates the input matrices and then waits for the results from the workers of the two farms. The second stage schedules (using the *ondemand* communication pattern) the matrices to the two farms. The sequential time to compute the square of all 8192 matrices is 333s for the 256×256 and 3260s for 512×512 matrices. As shown in Table 1 (left), the distributed version of the application (dFF-1) running the FastFlow pipeline and just one FastFlow farm on the same HostA (the parallel schema is sketched in Fig. 5), obtains 17.6× speedup for 256×256 matrices and 20× speedup for 512×512 matrices using 16 worker threads. The better result obtained with respect to the shared memory FastFlow version of the same benchmark (FF in the table), is justified by considering that pure FastFlow applications use only non-blocking threads to implement skeletons in order to minimize overheads when executing fine-grained tasks. However, when running medium to coarse-grained computations, non-blocking execution may increase contention and memory pressure not allowing increase of the number of threads to a level greater than the number of physical cores, in order to exploit multi-context platforms. In the distributed version, the *get* operations on the external channel are blocking, thus contention is reduced resulting in better performance. The distributed FastFlow version running on a single host is able to saturate all the available memory bandwidth of the tested platform (51.2 GB/s), reaching a maximum memory throughput of 51.8 GB/s. Figure 6 (left) sketches the performance obtained when 2 farms are used (dFF-2 in Table 1 right), one running on HostA (together with the pipeline) and one on HostB. In this case the speedup is 23$\times$ for 256$\times$256 matrix and $\sim 51\times$ for 512$\times$512 when the Infiniband network is considered. The super-linear speedup obtained is most likely due to the better L3 cache utilization on both hosts which provides the farm skeletons with more memory bandwidth. Figure 6 (right) shows the number of tasks computed by each farm. When the Ethernet network is used, much more tasks are scheduled toward the HostA (the one where the pipeline is mapped) because of the lower latency of the communication. As expected, the smaller the granularity, the bigger the number of task computed on the HostA for both networks.

Finally, we ran another test using a simple streaming application: two image filters (blur and emboss) have to be applied to a stream of input GIF images. The stream can be of any length and images of any size. In our test we fixed the number of images to 256, considering small size images (256KB) and coarser size images (1.7MB) as two separate test cases. The application uses the ImageMagick library[8] to manipulate the images and to apply the filters. The times reported in this paper do not consider the time spent to read and write the images from/to disk. The sequential time to compute all images is 27.5s and 232s for small and medium images, respectively. In Table 1 (right) is reported the minimum completion time obtained for the tested versions. The pure shared memory FastFlow version (FF) obtains a 11$\times$ speedup and 12$\times$ speedup for the two sizes. The distributed version uses a pipeline of 2 farms running on the 2 hosts (dFF-2). The first farm computes the blur and the second computes the emboss filter. The workers of the 2 farms are directly connected using a *unicast* channel (see the parallel schema in Fig. 7 right side). At the left side of Fig. 7 it is shown the completion time of the distributed version. The maximum speedup obtained is 8$\times$ and 8.5$\times$ speedup when using the Gigabit Ethernet, and 19.6$\times$ and 28.3$\times$ when using the Infiniband network, which represent fairly good results on a 2×16 core cluster.

5 Conclusions

We have proposed an extension of the FastFlow programming framework suitable for targeting cluster of multi-core workstations. Using a small set of applications, we have demonstrated that the extended FastFlow succeeds in exploiting resources in a cluster of workstations with different interconnection networks. We are currently working to implement the higher tier "algorithmic skeletons"

[8] http://www.imagemagick.org

in such a way that application programmers may seamlessly implement extended FastFlow applications much in the same way that they use to implement "single multi-core" applications with the original framework. The whole activity–along with the activities aimed at supporting GPUs within FastFlow [15]–is aimed at providing suitable means to implement the computing model designed within ParaPhrase, an FP7 STREP project whose intent is to use parallel design patterns and algorithmic skeletons to program heterogeneous–multi-core plus GPU–collections of processing elements.

References

1. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Comm. of the ACM **52**(10) (2009) 56–67
2. Park, I., Voss, M.J., Kim, S.W., Eigenmann, R.: Parallel programming environment for OpenMP. Scientific Programming **9** (2001) 143–161
3. Pacheco, P.S.: Parallel programming with MPI. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
4. Cappello, F., Etiemble, D.: Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In: Proc. of the 2000 ACM/IEEE conference on Supercomputing (CDROM). Supercomputing '00, IEEE Computer Society (2000)
5. Cole, M.: Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Computing **30**(3) (2004) 389–406
6. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. Software: Practice and Experience **40**(12) (2010) 1135–1160
7. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A library of constructive skeletons for sequential style of parallel programming. In: Proc. of the 1st Inter. conference on Scalable information systems. InfoScale '06, New York, NY, USA, ACM (2006)
8. Karasawa, Y., Iwasaki, H.: A parallel skeleton library for multi-core clusters. In: Proceedings of the 2009 International Conference on Parallel Processing. ICPP '09, Washington, DC, USA, IEEE Computer Society (2009) 84–91
9. Ciechanowicz, P., Poldner, M., Kuchen, H.: The Munster skeleton library Muesli – a comprehensive overview. In: ERCIS Working paper. Number 7. (2009)
10. Ciechanowicz, P., Kuchen, H.: Enhancing muesli's data parallel skeletons for multi-core computer architectures. In: Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications. HPCC '10, Washington, DC, USA, IEEE Computer Society (2010) 108–113
11. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In: Programming Multi-core and Many-core Computing Systems. Parallel and Distributed Computing. Wiley (2012)
12. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: Proc. of 18th Intl. Euro-Par 2012 Parallel Processing. LNCS, Springer (August 2012) To appear.
13. Aldinucci, M., Danelutto, M., Torquati, M.: Fastflow tutorial. Technical Report TR-12-04, Università di Pisa, Dipartimento di Informatica, Italy (March 2012)
14. ZeroMQ: website. (2012) `http://www.zeromq.org/`.
15. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Targeting heterogeneous architectures via macro data flow. Parallel Processing Letters **22**(2) (June 2012)