A Power-Aware, Self-Adaptive Macro Data Flow Framework

Marco Danelutto · Daniele De Sensi ·
Massimo Torquati

June 17, 2016

Abstract The dataflow programming model has been extensively used as an effective solution to implement efficient parallel programming frameworks. However, the amount of resources allocated to the runtime support is usually fixed once by the programmer or the runtime, and kept static during execution. While there are cases where such a static choice may be appropriate, other scenarios may require to dynamically change the parallelism degree during the application execution. In this paper we propose an algorithm for shared memory architectures, that dynamically selects the optimal number of cores to be used as well as their clock frequency according to the workload pressure. We implement the algorithm for both structured and unstructured parallel applications and we validate our proposal over two real applications, showing that it is able to save a significant amount of power, while not impairing the performance and not requiring additional effort from the application programmer.

Keywords Power-aware computing · Self-adaptive computing · Dataflow · Structured Parallel Programming

1 Introduction

In the recent years, researchers pay increasing attention on finding mechanisms and techniques for implementing energy-efficient applications. This interest is motivated both by environmental and economical reasons. Estimations report that, during 2010, data centers in the US produced more CO_2 than an entire country like Argentina or Netherlands [23], with an energy demand equal to the 3% of the overall energy production. Nevertheless, according to [27], [6],

Marco Danelutto, Daniele De Sensi, Massimo Torquati
University of Pisa
Computer Science Department
E-mail: {marcod, desensi, torquati}@di.unipi.it

the average utilisation of many systems is usually in the range $10\% - 50\%$. This opens many possibilities for energy saving by increasing the average utilisation of these systems. This solution is also supported by manufacturers, which provide architectural mechanisms to control and adapt the amount of used physical resources to the real application needs, for example by scaling the frequency of the CPUs or by turning off cores, cache or RAM modules when they are not used. However, the management of such mechanisms should be completely transparent to both the application programmer and to the end user. Indeed, the programmer should only deal with the functional correctness of his application, while this additional complexity should be hidden inside an appropriate runtime system.

In this paper we focus on streaming applications, usually characterised by significant workload fluctuations during their execution. We propose a runtime system for shared memory architectures, based on a dataflow programming model [19], [10] that, by constantly monitoring the application throughout its execution, is able to use at any time the least power consuming amount of resources sufficient for processing all the input elements. We will consider as resources both the number of cores used by the application and their clock frequency. As opposed to some existing solutions [29], [16], which are limited to a specific class of applications (e.g. data-parallel or data stream processing), we will manage any parallel application where data dependencies can be expressed as a direct acyclic graph (DAG). These include data-parallel and stream-parallel applications but also "unstructured" computations.

The main contributions of this paper are:

- A methodological solution for implementing power-aware, workload-sensitive dataflow runtime supports. By using such programming model, it will be possible to define both structured (e.g. based on algorithmic skeletons or parallel design patterns) and unstructured parallel applications. The autonomic management of the resources will be completely transparent to both the application programmer and to the end user, and will not require any additional programming effort.
- The implementation of a prototype runtime system that uses the proposed solution.
- The validation of our solution over two different streaming applications, showing that it is able to maintain an high resources utilisation, independently from the workload pressure. This leads to a significant reduction in the power consumption of the applications, while not impairing their performance.

This paper is structured as follows. In Sec. 2 we describe the Macro Data Flow model, then, in Sec. 3 we outline the strategy we use to monitor the application and to choose the most appropriate amount of resources to be used. Sec. 4 describes a simple example. In Sec. 5 the experimental validation is described and the main achieved results are shown. Eventually, we describe the related work in Sec. 6 and in Sec. 7 we draw some conclusions, proposing also some possible future directions.
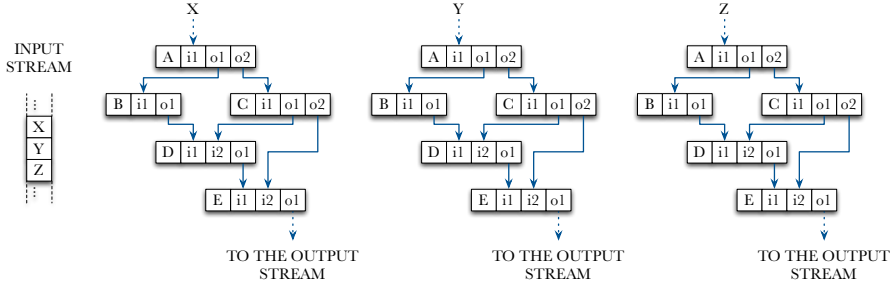
Fig. 1 Relationship between stream elements and Macro Data Flow graph instances.

2 Macro Data Flow

Macro Data Flow (MDF) [10] is a parallel programming model that allows the user to specify parallel computations by expressing them as direct acyclic graphs (DAG). In such graphs, each node represents a sequential code fragment, while the edges represent the flow of the data computed inside the graph. The nodes are also called "*Macro Data Flow instructions*" (MDFi), with the *Macro* term underlining the fact that each instruction actually represents a consistent part of the computation. Such structure is depicted in Figure 1, where A, B, C, D and E represent sequential code fragments, the arrows represent the data dependencies between such code blocks, and i* and o* represent the inputs and the outputs of each instruction.

Each instruction may receive/send data from/to one or more instructions, except for the first and last instruction of the graph. Indeed, the first instruction can only have one input, corresponding to the input data. Similarly, the last instruction has only one output, i.e. the result of the computation. By adding this constraint, we provide the possibility to seamlessly compose the graphs between each other, i.e. to use a graph in place of an instruction. As we will show later this can be used to implement the composability of skeleton-based applications.

When an instruction is executed, it produces one or more results, called *tokens*. These tokens will be used as inputs for the instructions that depend on the current one. For example, in Figure 1, when instruction A terminates its execution, it will produce two output tokens, one used as input for instruction B and one used as input for instruction C. An instruction become *fireable* (i.e. it is ready to be executed), when it receives all input tokens. For example, instruction E can be executed only after it has received the results from instructions C and D. The execution progress is orchestrated by an instruction *scheduler*, that works as follows:

1. A fireable instruction is located in the graph and sent to one of the *interpreters* in a pool of interpreters. The interpreters can execute different instructions in parallel, thus allowing to exploit the parallel execution between instructions that do not have data dependencies. Each of the inter-

preters in the pool is capable of executing any fireable instruction. This is possible because the instruction code is stored inside the instruction itself.

2. The results of instruction executions are received from the interpreters, and used by the scheduler as input tokens for the corresponding destination instructions. As soon as one of these instructions become fireable, it is sent to the interpreters.

These two steps are iterated until there are no more fireable instructions available (i.e. up to program termination).

In general, there may be a *stream* of data input tokens to be computed. We call each element on the input stream a *task*. In this case, for each received task, a new instance of the graph is created and stored into a *graph pool*. When the last instruction of a graph is executed, the final result is sent over an *output stream* and the corresponding graph instance is destroyed. Accordingly, there is a 1-to-1 association between stream elements and graphs instances. For this reason, the scheduler has to be able to receive data from the input stream as well as to receive the results from the interpreters in order to update the graphs corresponding to the already received stream elements. A tradeoff in the priority given to these two steps must be found. Indeed, if the priority is always given to the input stream, then the computations of the graphs already present in the system will never advance in their execution. On the other hand, if we read from the input stream only when there are no more fireable instructions, then we are not fully exploiting the available parallelism. In our implementation, we face this problem by putting a threshold to the maximum number of graphs (i.e. stream elements) that can be present inside the graph pool at any moment. The higher the threshold, the more we shift towards a solution where we always prefer to read from the input stream.

Formally, a generic instruction instance (i.e. a node of a graph instance), can be viewed as a 5-tuple:

$$Gid \times Id \times Code \times I^n \times O^m \tag{1}$$

where Gid is the identifier of the specific graph instance, Id is an identifier used to distinguish an instruction from the other instructions of the same graph instance and $Code$ is a pointer to the code to be executed in the instruction. I^n and O^m represent respectively the n instruction that will produce the input tokens for this instruction and the m instructions that will receive the tokens produced by this instruction.

In addition to arbitrary graphs, we also provide the possibility to specify skeleton applications. Currently, we support *pipeline*, *farm*, *map* and *reduce* skeletons [9]. These skeletons will then be actually compiled into suitable Macro Data Flow graphs.

2.1 Implementation

As common in many dataflow environments [2] [28], we implemented the runtime support through a master-worker pattern, with the master running the

GRAPHS POOL
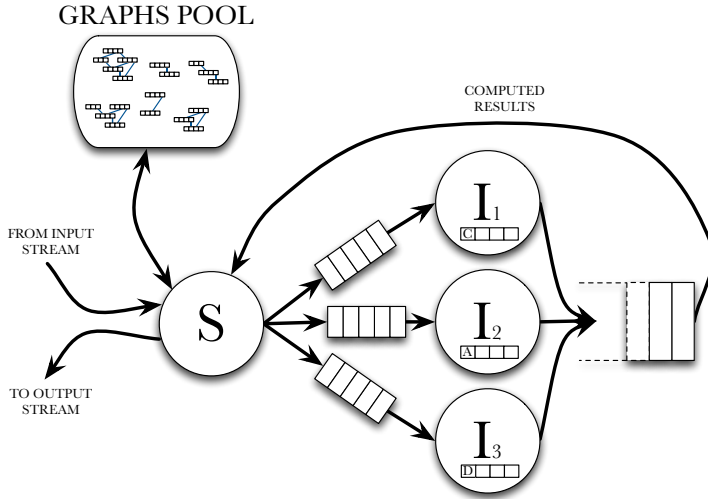
Fig. 2 The runtime architecture. Node 'S' is the scheduler. Nodes 'I' are the interpreters.

scheduler of the instructions and the workers running the interpreters. The master and the workers run in separate threads, as depicted in Figure 2. In principle having a single master could be a bottleneck, thus limiting the scalability of the system. However, it simplifies the possibility of defining custom scheduling policies improving the flexibility of the approach. Moreover, the scheduler implementation we used has been proven efficient and feasible in similar contexts for state-of-the-art multicore architectures up to 32/64 cores using a variety of different applications [13].

For the implementation of our dataflow interpreter, we used FASTFLOW, a parallel programming framework for multicore platforms based on non-blocking lock-free/fence-free synchronisation mechanisms [14]. The FASTFLOW framework is composed of a stack of layers that progressively abstracts out the programming of shared-memory parallel applications. The goal of the stack is twofold: to ease the development of applications and to make them very fast and scalable. FASTFLOW is particularly targeted to the development of streaming applications and we have chosen it for two main reasons: i) it already provides the possibility to dynamically change the number of workers used by a master-worker pattern, ii) it provides full flexibility for scheduling stream elements to workers.

When one of the interpreters in the pool finishes the execution of an instruction, it inserts the results into a lock-free multi-producer, multi-consumer queue [24]. The results will then be read by the scheduler and used to update the corresponding dataflow instructions stored in the graph pool.

To guarantee the execution progress even when no elements are present on the input stream or when no results have been computed by the interpreters,
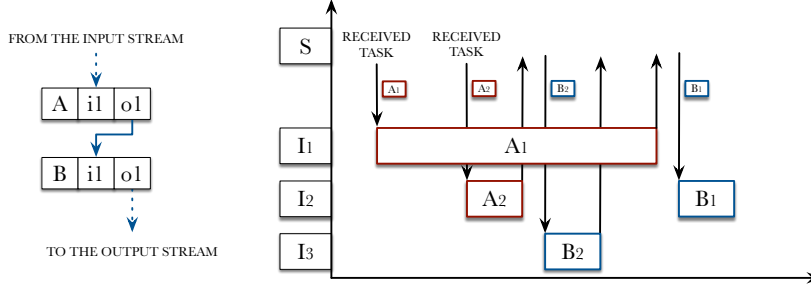
Fig. 3 Reordering of the instructions.

the interaction with both the input stream and the queue of computed results has been implemented in a non-blocking fashion.

In general, the scheduler sends fireable instructions to the interpreters (workers) by using an *on-demand* strategy (i.e. the instruction is sent to an interpreter if it has no other instructions to process). However, in some cases different scheduling strategies may be more appropriate. For example, consider the case depicted in Figure 3 where we have 3 interpreters, a graph composed by two pipelined instructions (A and B) and an element arrives on the input stream. The scheduler will generate a new graph instance. Since the first instruction (A_1) becomes fireable, it will send it to the first available interpreter (I_1). In the meanwhile, a new element arrives on the input stream, the corresponding graph is generated and its first instruction A_2 is sent to interpreter I_2. When A_2 terminates its execution, B_2 became fireable and it is sent to I_3. Note that, since execution of A_1 takes too long to terminate, B_2 is executed before B_1.

Albeit this can be acceptable in some applications, there are situations where the application programmer needs instructions belonging to different stream elements to be processed in the same order they are received. Note that this cannot be enforced by data dependencies alone. Indeed, dataflow dependencies ensure the correctness of the execution *for a specific input* but they do not constrain instructions associated to different stream elements. To solve this problem, we can schedule the instructions with the same identifier to the same interpreter. Since the enqueued instructions are processed by the interpreter in FIFO order, we guarantee that instructions with the same identifier and associated to different stream elements are processed in the same order they are received. However, this type of scheduling may lead to workload imbalance between the interpreters, since some of them may receive more (or more costly) instructions. To clarify this, let's consider the case where we have 3 interpreters, a graph composed by 7 instructions with the same average latency and a scheduling function that assigns the instructions with

identifier x to the interpreter $I_{x\%7}$. This scheduling function assigns all the instructions with the same identifier to the same interpreter. However, for each element received from the input stream, the interpreter I_1 will process 3 instructions while interpreters I_2 and I_3 will process 2 instructions each. This workload imbalance could be present even if different scheduling function are used and even if the instructions have a different latency. Indeed, since a given instruction can only be executed by a specific worker, an optimal scheduling could not always be found and, as we will show in Section 5, this could lead to inefficiencies.

Another problem concerns the order in which the results are sent on the output stream. When instructions associated to different stream elements can be executed in any order, the last instruction of the graphs (which will produce the output stream elements), could be executed in any order as well. Therefore, the elements could appear on the output stream in the wrong order. Since an increasing identifier is assigned to the graphs when they are created, we can keep track of the last output sent over the stream. When a new result to be sent appears, if the identifier of his graph is the next to be sent on the output stream, then it is sent, otherwise the result is stored into an hash-table. This hash-table is periodically flushed on the output stream in the correct order.

Macro Data Flow support could also be a part of a bigger skeleton application. For example, we could have a 3-stage pipeline where the middle stage uses the dataflow runtime. In this case, the input stream will correspond to the output channel of the first pipeline stage, and the output stream will correspond to the input channel of the last pipeline stage. This is possible because we can guarantee the order of the stream elements flowing through the dataflow stage.

3 Power aware adaptivity

The main idea to implement power aware adaptivity consists in dynamically changing the number of computational resources used by the runtime support according to the rate of instructions to be processed. This rate may change during the execution due to external factors (for example because the arrival rate of the input elements changes) or even because the application exhibits different behaviours during its execution. In general, we would like to avoid under-utilisation or over-utilisation of the computational resources. Indeed, under-utilisation implies that there are resources that are kept active but are not fully utilised, thus leading to a power-inefficient execution. On the other hand, over-utilisation implies the impossibility to manage all the instructions produced by the dataflow scheduler. To face this problem, we used the algorithm described in [12]. That algorithm was only used on a specific application and, to the best of our knowledge, this is the first attempt to use it in a dataflow runtime.

To check if the interpreter is under-utilised or over-utilised, at regular time intervals, we compute the average utilisation factor of the application, defined

as:

$$\rho = \frac{T_S}{T_A}$$

where T_S is the mean time required by the set of interpreters to process an instruction and T_A is the average time between the issue of two fireable instructions. The system will be able to compute all the instructions only if $\rho < 1$. If such condition is not verified, the length of the input queues of the interpreters will constantly increase until the system memory is completely exhausted.

To optimise system utilisation, we would like to keep the utilisation as close as possible to 1. In general, we want to keep ρ between two thresholds ρ_{min} and ρ_{max}, with ρ_{max} very close to 1. For example, we may want to guarantee that $0.8 < \rho < 0.9$. The rationale is that, when the utilisation is lower than ρ_{min}, the system is under-utilised and we could decrease the used resources while still being able to manage the same instructions rate. Similarly, when ρ is greater than ρ_{max}, the system starts to become over-utilised and we should increase the resources in order to manage all the generated instructions.

In this paper, we consider as resources the number of physical cores used by the interpreter and their clock frequency. As common in similar works [22], we assume to have at most one thread running on each core. Accordingly, since each interpreter runs in a different thread, we change the number of cores used by the runtime by changing the number of interpreters it uses. We call *configuration* a specific ⟨CORES, FREQUENCY⟩ pair. When the monitored utilisation is outside the specified range, we need to change the current configuration. To decide how many cores to use and at which clock frequency they should run we need to predict how the utilisation changes when the amount of used resources changes. For simplicity, in this work we ignore the utilisation of the scheduler, by only considering the utilisation of the workers. However, the model we used can be easily extended to consider that factor too. Consequently, we consider the utilisation of the runtime to be the average of the utilisations of its interpreters. We define the current configuration as a pair $< \overline{\omega}, \overline{\pi} >$ where $\overline{\omega}$ is the number of interpreters and $\overline{\pi}$ is the frequency of the cores on which the interpreters are running. We then define a reconfiguration as a change of the used resources from $< \overline{\omega}, \overline{\pi} >$ to a different generic configuration $< \omega, \pi >$. To select the destination configuration we need to predict all the utilisations $\rho(\omega, \pi)$ for any ω and π and select one characterised by a utilisation that falls inside the specified range. For this purpose, we used the following equation, better described in [12]:

$$\rho(\omega, \pi) = \rho(\overline{\omega}, \overline{\pi}) \times \frac{\overline{\omega} \times \overline{\pi}}{\omega \times \pi} \tag{2}$$

$\rho(\overline{\omega}, \overline{\pi}), \overline{\omega}$ and $\overline{\pi}$ are constants since they are obtained from the current configuration. This equation implies that the utilisation proportionally decreases when the number of workers and/or the frequency of the cores they use is increased. This is in general true for computations which spend most of their

execution time on CPUs. However our approach is generic and, if needed, more complex models could be used.

Since in general there will be many configurations characterised by a good predicted utilisation, we would like to select the one with the lowest power consumption. Accordingly, we also need to predict the power consumption in different configurations. Note that we are not interested in knowing the exact power consumption but only a proportional estimation such that we can compare two different configurations in order to pick the one consuming less. According to [8], [3], [21], the power consumption can be estimated as:

$$P(\omega, \pi, \gamma) \propto \omega \times \pi \times \gamma^2$$

where π is the operating frequency and γ is the supply voltage. Since the voltage depends on the frequency of the processor, we can rewrite it as $P_{\omega,\pi} \propto \omega \times \pi \times V(\pi)^2$, where V is a function that associates a voltage to a specific frequency. Consequently, from the restricted set of configurations, we will pick the one such that $\omega \times \pi \times V(\pi)^2$ is minimum.

This reconfiguration strategy has been implemented by using NORNIR[1], an autonomic support for FASTFLOW applications. The reconfiguration decisions are taken by an external *manager* thread, that interacts with the master-worker pattern, as typical in autonomic applications [1]. The manager monitors the interpreters and, when they become under- or over-utilised, predicts performance and power consumption of all other possible configurations and then selects the best one. After that, it sends a configuration change request to the scheduler. When such request is received by the scheduler, it sends a pause command to all the interpreters. Eventually, the scheduler will (re-)start only the interpreters needed according to manager's prediction. The pause and restart commands are only necessary when a custom scheduling function is used, in order to keep it consistent (e.g. when the programmer requires the tasks to be processed in the arrival order). Concerning the clock frequency, it can be directly modified by the manager by using the tools available on the target platform. Note that the configuration changes are completely transparent both to the application programmer and to the final user. This is possible since, even if the number of interpreters is changed, the structure of the computation as specified by the programmer is not modified, thus not requiring any state redistribution.

The code of the dataflow runtime and the provided applications have been integrated into NORNIR framework and can be downloaded from its website[1].

4 Programmability aspects

In this Section we describe the steps to be performed by a user to specify a generic DAG dataflow computation and how to specify constraints on performance or power consumption. To define a generic graph, it is sufficient to

---

[1] NORNIR website: http://danieledesensi.github.io/nornir/

define the instructions (wrapping the sequential code or other already existing graphs) and to link them together. The following code snippet shows how to do that in order to build the graph in Figure 1.

```cpp
Computable *A, *B, *C, *D, *E;
class InputStreamApp: public InputStream{
public:
    inline void* next(){...}
    inline bool hasNext(){...}
};
class OutputStreamApp: public OutputStream{
public:
    void put(void* a){ /* Send the element. */ }
};
... /** Code for other instructions. **/
class DCode: public Computable{
public:
    void compute(){
        // Get data from the 2 instructions linked to D.
        receiveData(...); receiveData(...);
        ... // Compute
        // Send result to the instruction linked to D.
        sendData(..., result);
    }
};
int main(){
    A = new ACode(...); B = new BCode(...); C = new CCode(...);
    D = new DCode(...); E = new ECode(...);

    Mdfg graph;
    graph.link(A, B); graph.link(A, C); graph.link(B, D);
    graph.link(C, D); graph.link(C, E); graph.link(D, E);

    InputStreamApp isa; OutputStreamApp osa;

    Parameters p;
    p.rhomin = 80.0; p.rhomax = 90.0;
    p.dataflow.orderedOutput = true;
    Dataflow d(&p, &graph, &isa, &osa);
    d.start(); d.wait();
    return 0;
}
```

First of all, the input stream is defined (lines 2-6). This class must provide two member functions. The function (next()) must return the stream element if available, or NULL if no elements are currently present on the stream. The function (hasNext()) returns true if there are still elements to receive from
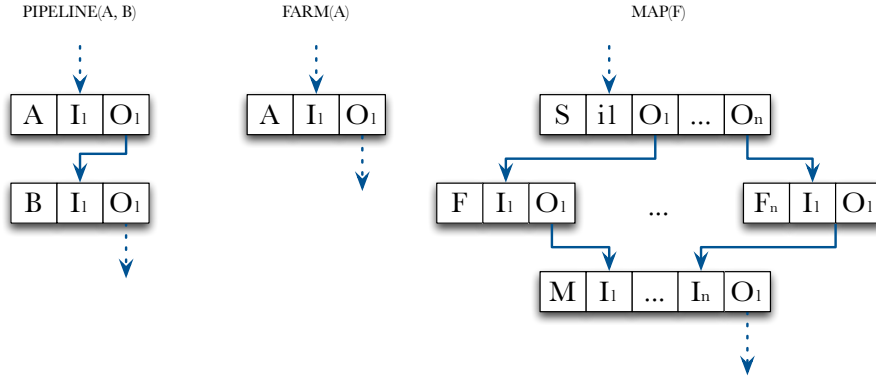
Fig. 4 Macro Data Flow Graph corresponding to the skeletons provided.

the stream and `false` when there are no more elements on the stream. Then, the output stream should be defined (lines 7-10), by implementing the function `put()` to manage the computed results. To define the Macro Data Flow instructions, the application programmer needs to wrap each business logic fragment (i.e. the actual code performing the computation) into a class that extends the `Computable` class and implements the `compute()` function (lines 14-20). In the `compute()` function the application programmer can use the data received from the linked instructions (line 16) and send the results to the output linked instructions (line 19). After that, the instructions are created (lines 23-24) and linked together (lines 27-28). Eventually, optional parameters may be specified (lines 33-34) and the runtime can be created and started (line 36).

For skeleton-based computations, the graphs are already known and well defined. Accordingly, it is possible to avoid to specify the individual instructions and their linkage, by only defining the type of skeleton and the business code. In Figure 4 we show the graphs corresponding to the provided skeletons. The farm skeleton is compiled into a graph composed by a single Macro Data Flow instruction. Indeed, since a graph is created for each element on the input stream and since the instructions are executed in parallel by a pool of interpreters, this leads to a behaviour corresponding to the one of a farm skeleton. The graph of the *reduce* skeleton is not shown since it is very similar to the *map* one. Moreover, skeletons (but also unstructured graphs) can be composed. In the code fragment below we show a pipeline where the first stage receives a `double` and produces an `int` and the second stage receives an `int` and produces a `float`. Each stage is parallelised by using a farm.

```
1  int* w1(double* t){/** Process t and return an int*. **/}
2  float* w2(int* t){/** Process t and return a float*. **/}
3  Pipeline p(getFarm<double, int>(w1), getFarm<int, float>(w2));
```

5 Results

In this Section we describe the applications we used to validate our approach
and we analyse the obtained results. All experiments were conducted on an
Intel workstation with 2 Xeon E5-2695 @2.40GHz CPUs, each with 12 2-
way hyperthreaded cores, running with Linux x86_64. This machine has 13
possible frequency levels: from 1.2GHz to 2.4GHz with 0.1GHz steps. Since
we plan to have at most one thread per physical core, we will perform our
tests by using at most 24 threads (22 for the interpreters + 1 for the scheduler
and 1 for the manager). On the considered platform, we used RAPL power
management interface to measure the power consumption of the CPUs. For
all the experiments, we set $\langle \rho_{min}, \rho_{max} \rangle = \langle 0.8, 0.9 \rangle$.
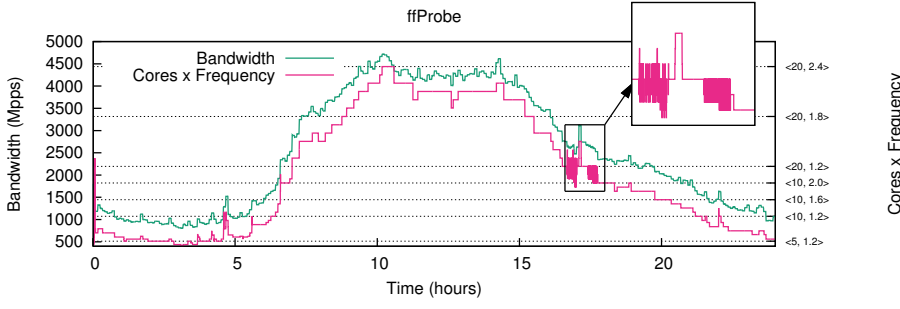
5.1 ffProbe

The first application we used is FFPROBE [11], a parallel implementation of a
NetFlow probe, i.e. an application responsible for network traffic monitoring.
Network packets are aggregated in *flows*, created when the first packet of the
flow has been received, and destroyed when some *expiration* conditions is ver-
ified. In FFPROBE, the application is logically structured as a pipeline, where
each stage manages a partition of the active flows. When a packet is received
by the input stream, it is inserted into a stream task and sent to the first
worker of the pipeline. If the packet belongs to a flow managed by the worker,
the corresponding flow is updated. After that, the worker checks if some of
the flows it manages are expired and, if this is the case, these expired flows
are added to the task. Eventually, the task is forwarded to the next pipeline
stage. Such kind of implementation has been proven [11] to have performance
comparable or better to those obtained with commercial solutions. However,
this is the first attempt to implement it over a power aware dataflow runtime.
By implementing FFPROBE by using a dataflow model, we obtained the same
peak performance obtained by the original implementation.

For our experiments, we generated a synthetic traffic dataset, as common
for such scenarios [11]. Since we are interested in analysing the behaviour of
the runtime support when fluctuations in the input rate are present, we receive
the data at a variable rate. To preserve the characteristics of a real scenario,
we used data rates of a real network[2], covering a 24 hours span. To model
future network utilization scenarios, this rate has been linearly scaled up in
order to increase the amount of parallelism need.

In this application, the packets belonging to the same flow are always
processed by the same Macro Data Flow instruction and since they must be
processed in the same order they are received, this requirement is specified in
the parameters of the runtime. As we described in Section 2, this implies that
the scheduling of the instructions to the interpreters is done according to their
identifier. Since the instructions are characterised by a very similar execution

---

[2] http://bit.ly/1RY7fEt - The used rate is the one collected on 08 Jun 2005

Fig. 5 Input bandwidth (Millions Packets Per Second) and used resources for the FFPROBE application.
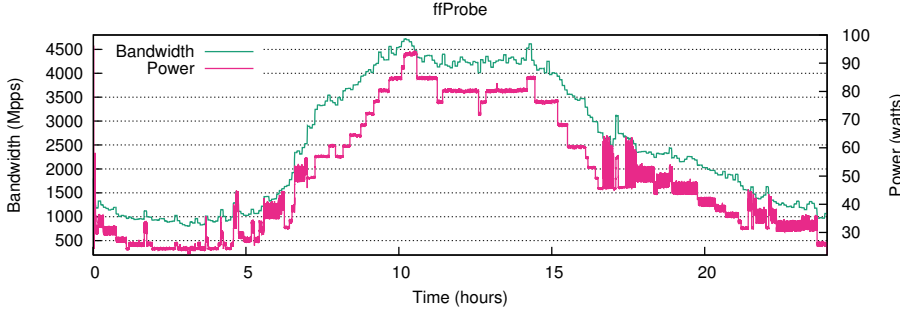
Fig. 6 Input bandwidth (Millions Packets Per Second) and power consumption for the FFPROBE application.

time, the best scheduling strategy is to partition the graph in equal parts among the interpreters in order to keep the load perfectly balanced. However, this is only possible if the number of graph instructions is a multiple of the number of active interpreters. If this is not the case, we would have some interpreters much more loaded than others. Moreover, since we instantiate a new graph for each element received from the input stream, this unbalance would accumulate as more elements are received, thus leading to the impossibility to process all the elements of the stream. In a static scenario, this could be solved by forcing the number of interpreters to be a divisor of the number of instructions in the graph, in order to have a perfectly balanced scheduling. However, this solution is not feasible in our scenario since we are dynamically changing the number of interpreters at runtime. To solve this problem, we forced the runtime to only use a number of interpreters which is a divisor of the number of instructions in the graph. In our experiments, we have a graph composed by 20 instructions. Accordingly, the runtime can activate 1, 2, 4, 5, 10 or 20 interpreters.

In Figure 5 we show the bandwidth of the data sent to the application[3] and the amount of used resources. We plot the product between the number of

---

[3] The bandwidth of the received data is not shown since the application is always able to receive all the sent data

used cores and their frequency since, as anticipated in Section 3, we expect the performance to be proportional to this quantity, as validated in the figure. The labels on the right y-axis report the number of cores and the clock frequency used by the runtime. We would like to point out that around 18 hours from the application start, the runtime starts to oscillate between 10 cores running at 2GHz and 20 cores running at 1.2GHz. This happens since the optimal solution falls in an intermediate value that cannot be used because we restricted the possible choices for the number of interpreters (i.e. cores) that can be used by the runtime. The final effect of the resources scaling is sketched in Figure 6, where we show both the input data rate and the power consumption of the application. The power consumption ranges from 20 to 100 Watts according to the workload conditions. It is worth noting that, if a self-adaptive solution were not used, the system would be underutilised for most of the time, consuming around 100 Watts for its entire duration.

An alternative solution to our approach under Linux OSs, could be to let the runtime use always the maximum number of cores, and to delegate the clock frequency scaling management to the OS by using the `ondemand` frequency governor. This strategy however consumes 12.74 % more power with respect to the solution we proposed. This happens because our strategy can operate both on the number of cores and their frequency, thus extending the range of possible configurations and allowing a more fine grained control.

5.2 Blackscholes

This second application is provided by the well-known PARSEC benchmark. It analytically calculates the prices for a portfolio of stock options by using the Black-Scholes partial differential equation. This application is structured as a farm, where each stock option received from the input stream is scheduled to a different worker. Differently from the previous case, this application does not require the tasks to be processed according to any precise order. In this case, the instructions will be scheduled according to an on-demand policy, thus avoiding the unbalancing problems that characterise the FFPROBE application. To perform our experiments, we used the PARSEC `native` input and the input workload of a trading day of the NASDAQ market[4].

Firstly, we evaluated the maximum performance achieved by our dataflow solution with those obtained by using PTHREAD, OpenMp and Intel TBB implementations that are all distributed with PARSEC. In Table 1 we show the average normalised value (between 0 and 1) obtained over 10 different executions of the experiment. The dataflow implementation obtains a slightly better performance with respect to the other solutions. This is mainly due to the fact that the dataflow runtime pins each interpreter on a specific core. On the contrary, PTHREAD, OpenMp and Intel TBB implementations let the operating system manage the threads allocation.

---

[4] http://www.nyxdata.com - The used trading day is 30 Oct 2014

Table 1 Maximum performance achieved by different Blackscholes implementations. Results are normalised between 1 (best) and 0.

Dataflow	Pthreads	OpenMp	Intel TBB
1	0.87	0.83	0.95
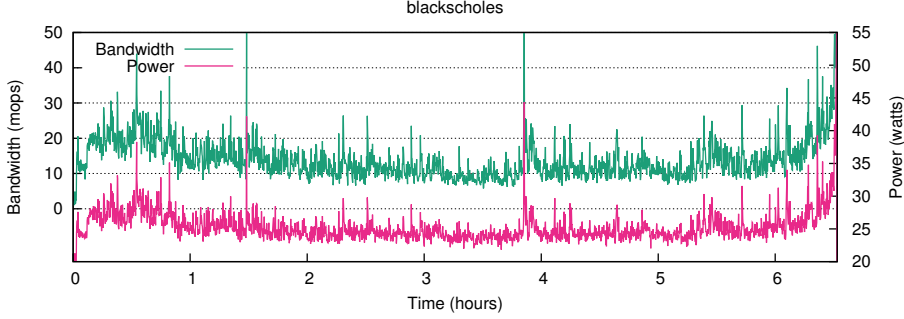
Fig. 7 Input bandwidth (Millions Options Per Second) and power consumption for the Blackscholes application.
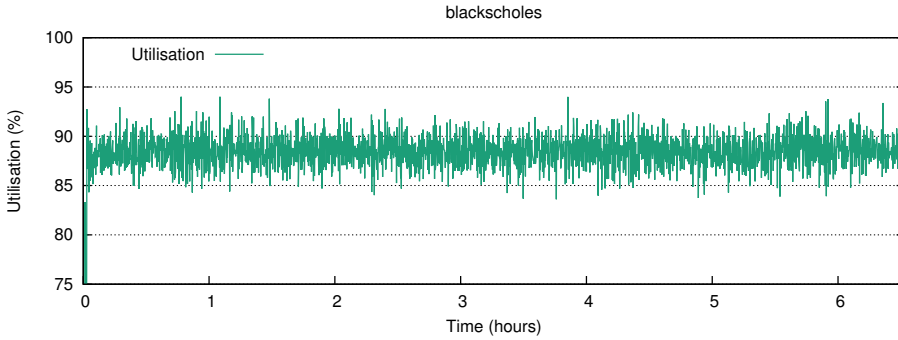
Fig. 8 Resources utilisation for the Blackscholes application.

Concerning the dynamic behaviour, as we can see from Figure 7, the dataflow interpreter scales its power consumption according to the rate of packets received by the application. Furthermore, as shown in Figure 8, the solution always satisfies the requirements by keeping the utilisation factor ρ between the specified bounds ($80\% < \rho < 90\%$).

6 Related work

In this section we describe the main research results both in the area of dataflow models and in the area of power-aware runtime systems.

Dataflow is a computing model that has been around since the earliest days of computer science research activities [19], [18]. Many programming environments currently use dataflow concepts in the implementation of different parallel programming models. For example, different systems provide the

programmer with the possibility to define/identify activities that are subsequently scheduled for parallel execution according to different execution models and scheduling policies. Cilk provides the application programmer with the possibility to spawn computations (C function/procedure calls) and to wait asynchronously for their termination [7]. The execution model is based on the scheduling of the resulting DAG representing instructions and their dependencies. The supporting run time library uses job-stealing to guarantee efficiency and appears similar to the Macro Data Flow interpreter we have implemented. OpenMP [5] and some recent extensions [26] provide the possibility to use code annotations to define independent blocks of code, which are scheduled for execution in a fairly similar manner to that used for our fireable Macro Data Flow instructions. StarPU [4] supports parallel activities graphs in essence similar to Macro Data Flow graphs and executes these graphs on heterogeneous architectures (multi-core + GPU) with interesting results. More recently, Microsoft TPL[5] and Google's TensorFlow[6] have been proposed and extensively used in commercial solutions. However, many of these programming environments do not provide any structured parallel programming abstraction.

Recently, dataflow principles have been proposed through S-Net as a suitable candidate for implementing coordination languages [25]. S-Net is a mature dataflow model, providing static soundness guarantees and recursive graphs. We believe that our work is orthogonal to S-Net for two main reasons. The first one is that in S-Net the nodes of the graph must be stateless, while in our model each node can have its own internal state. This is an important feature for a large class of applications where the presence of a state is necessary. For example, for FFPROBE state is needed to correlate different packets to perform network analysis. The second reason is that, albeit some self-adaptive techniques has been built over S-Net, they have some important limitations. For example, in [20], the authors propose an algorithm to avoid underload and overload of physical resources. However, they only operate on voltage and frequency, while keeping the amount of used physical cores fixed. On the contrary, our algorithm operates on both frequency and physical cores, extending the range of possible configurations and allowing a more fine grained control, which leads to a lower power consumption. Moreover, the algorithm they propose can only increase/decrease frequency one step at a time, thus requiring a longer time to reach the target. In highly dynamic scenarios the input rate may change again before the algorithm reaches the desired frequency level, thus leading to an unstable behaviour. This effect is mitigated in our algorithm since, by predicting the optimal configuration, we can skip the intermediate steps. Similar to our work, in [17] the authors present a solution to minimise power consumption under a given performance constraint for dataflow applications. However, they do not have the concept of *stream* and are mainly targeted towards batch applications. Moreover, their algorithm needs information extracted through a statical analysis of the application structure. On

---

[5] `https://msdn.microsoft.com/en-us/library/hh228603(v=vs.110).aspx`

[6] `https://www.tensorflow.org/`

the contrary, our approach is agnostic from the specific structure and could potentially work on dynamically evolving graphs. Other solutions [16], [15] are targeted towards *Data Stream Processing* (DaSP) applications. However, such solutions lack of generality, since they impose more restrictions on the application characteristics compared to a dataflow approach.

7 Conclusions

In this work we proposed a general methodology for the implementation of power aware dataflow runtime systems. By constantly monitoring the application and by changing the amount of used resources according to the workload condition, we are able to save a considerable amount of power. We described a concrete implementation of this approach, validating it over two real streaming applications, showing the limitations and the advantages of the design.

As a future work, we would like to improve the expressiveness of our solution, by simplifying the specification of the individual instructions and their linkage, for example by using C++ *pragmas* or *attributes*. Moreover, we will investigate the possibility to express explicit requirements in terms of performance and/or power consumption, both for streaming and for batch computations. To achieve this goal, more accurate models (e.g. machine learning) for the prediction of the utilisation and of the power consumption in different configuration are needed. Lastly, since changing the configuration may be a costly operation, another possible research direction involves the use of control theory techniques to avoid to switch the configuration too often.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Autonomic management of non-functional concerns in distributed and parallel application programming. In: Proc. of Intl. Parallel & Distributed Processing Symposium (IPDPS), pp. 1–12. IEEE, Rome, Italy (2009)
2. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in Java. Future Gener. Comput. Syst. **19**(5), 611–626 (2003)
3. Alonso, P., Dolz, M., Mayo, R., Quintana-Ort, E.: Modeling power and energy of the task-parallel cholesky factorization on multicore processors. Computer Science - Research and Development **29**(2), 105–112 (2014)
4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. : Pract. Exper. **23**(2), 187–198 (2011)
5. Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The design of openmp tasks. IEEE Transactions on Parallel and Distributed Systems **20**(3), 404–418 (2009)
6. Barroso, L., Holzle, U.: The case for energy-proportional computing. Computer **40**(12), 33–37 (2007)
7. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. SIGPLAN Not. **30**(8), 207–216 (1995)

8. Chandrakasan, A., Brodersen, R.: Minimizing power consumption in digital cmos circuits. Proceedings of the IEEE **83**(4), 498–523 (1995)

9. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Computing **30**(3), 389 – 406 (2004)

10. Danelutto, M.: Efficient support for skeletons on workstation clusters. Parallel Processing Letters **11**(01), 41–56 (2001)

11. Danelutto, M., Deri, L., Sensi, D.D.: Network monitoring on multicores with algorithmic skeletons. In: Applications, Tools and Techniques on the Road to Exascale Computing, Proceedings of ParCo 2011 conference, pp. 519–526 (2011)

12. Danelutto, M., Sensi, D.D., Torquati, M.: Energy driven adaptivity in stream parallel computations. In: 23rd Euromicro Int'l Conf. on Parallel, Distributed, and Network-Based Processing, PDP 2015, pp. 103–110 (2015)

13. Danelutto, M., Torquati, M.: Loop parallelism: A new skeleton perspective on data parallel patterns. In: 22rd Euromicro Int'l Conf. on Parallel, Distributed, and Network-Based Processing, PDP 2014, pp. 52–59 (2014)

14. Danelutto, M., Torquati, M.: Structured Parallel Programming with "core" FastFlow. In: Central European Functional Programming School, *LNCS*, vol. 8606, pp. 29–75. Springer (2015)

15. De Matteis, T., Mencagli, G.: Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 13:1–13:12 (2016)

16. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. IEEE Trans. Parallel Distrib. Syst. **25**(6), 1447–1463 (2014)

17. Holmbacka, S., Nogues, E., Pelcat, M., Lafond, S., Menard, D., Lilius, J.: Energy-awareness and performance management with parallel dataflow applications. Journal of Signal Processing Systems pp. 1–16 (2015)

18. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. **36**(1), 1–34 (2004)

19. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress, pp. 471–475 (1974)

20. Karavadara, N., Zolda, M., Nguyen, V., Knoop, J., Kirner, R.: Dynamic power management for reactive stream processing on the scc tiled architecture. Eurasip Journal on Embedded Systems (2016)

21. Kim, N., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J., Irwin, M., Kandemir, M., Narayanan, V.: Leakage current: Moore's law meets static power. Computer **36**(12), 68–75 (2003)

22. Li, J., Martínez, J.F.: Dynamic power-performance adaptation of parallel computation on chip multiprocessors. Proceedings - International Symposium on High-Performance Computer Architecture **2006**, 77–87 (2006)

23. Lucente, E.J.: The coming "c" change in data centers (2010). `http://www.hpcwire.com/2010/06/15/the_coming_c_change_in_datacenters/`

24. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. of the Fifteenth Annual ACM Symp. on Principles of Distributed Computing, PODC '96, pp. 267–275. ACM, New York, NY, USA (1996)

25. Penczek, F., Cheng, W., Grelck, C., Kirner, R., Scheuermann, B., Shafarenko, A.: A data-flow based coordination approach to concurrent software engineering. In: Data-Flow Execution Models for Extreme Scale Computing (DFM), 2012, pp. 36–43 (2012)

26. Planas, J., Badia, R.M., Ayguade, E., Labarta, J.: Hierarchical Task-Based Programming With StarSs. International Journal of High Performance Computing Applications **23**, 284–299 (2009)

27. Ranganathan, P.: Recipe for efficiency: Principles of power-aware computing. Commun. ACM **53**(4), 60–67 (2010)

28. Sridharan, S., Gupta, G., Sohi, G.S.: Holistic run-time parallelism management for time and energy efficiency. In: Proceedings of ICS 2013 conference, p. 337. ACM Press, New York, New York, USA (2013)

29. Suleman, M.A., Qureshi, M.K., Patt, Y.N.: Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. SIGARCH Comput. Archit. News **36**(1), 277–286 (2008)