A POWER-AWARE, SELF-ADAPTIVE MACRO DATA FLOW FRAMEWORK

MARCO DANELUTTO

Computer Science Department, University of Pisa,
Pisa, Italy
marcod@di.unipi.it

and

DANIELE DE SENSI

Computer Science Department, University of Pisa,
Pisa, Italy
desensi@di.unipi.it

and

MASSIMO TORQUATI

Computer Science Department, University of Pisa,
Pisa, Italy
torquati@di.unipi.it

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 the entire 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 multicore shared memory platforms, that dynamically selects the optimal number of cores to be used as well as their clock frequency according to either the workload pressure or to explicit user requirements. We implement the algorithm for both structured and unstructured parallel applications and we validate our proposal over three 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 [21], with an energy demand equal to the 3% of the overall energy production. Nevertheless, according to [24], 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 multicore shared memory architectures, based on a dataflow programming model [18, 9] that, by constantly monitoring the application throughout its execution, is able to automatically use at any time the least power consuming amount of resources sufficient for processing all the input elements or to satisfy explicit user requirements. We will consider as resources both the number of cores used by the application and their clock frequency. As opposed to some existing solutions [27, 15], which are limited to a specific class of applications (e.g. data-parallel or data stream processing), we will manage any streamable 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 three 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. Moreover, we show that our algorithm is also able to satisfy user requirements on maximum execution time.

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.

2. Macro Data Flow

Macro Data Flow (MDF) [9] 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) (also known as *operations* in other similar programming models), 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.
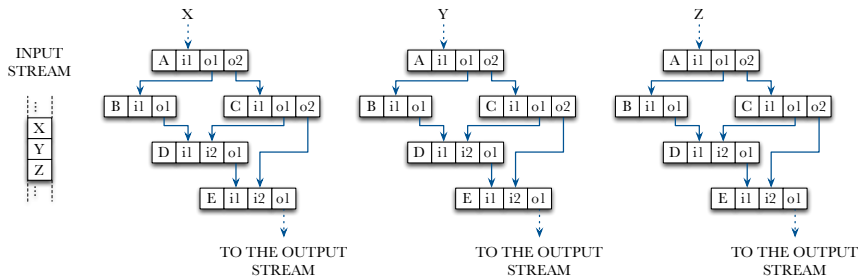

Fig. 1.   Relationship between stream elements and Macro Data Flow graph instances.

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 (also known as *workers* in other similar programming models). 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 interpreters 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.

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 limit to the maximum number of graphs (i.e. stream elements) that can be present inside the graph pool at any moment. The higher the limit, the more we shift towards a solution where we always prefer to read from the input stream. Moreover, to avoid starvation, the scheduler will not block on any of the two inputs if no elements are present on such inputs. This will be better clarified in Section 2.1.

Algorithm 1 shows the full scheduler pseudocode.

ALGORITHM 1: Reconfiguration algorithm - General case

```
1  Function Scheduler()
2  │   numGraphs ← 0;
3  │   while true do
4  │   │   if numGraphs < maxNumGraphs then
5  │   │   │   if t ← receiveFromStream() then
6  │   │   │   │   numGraphs ← numGraphs + 1;
7  │   │   │   │   g ← pool.createGraph(t);
8  │   │   │   │   sendToInterpreters(g.getFirst());
9  │   │   │   end
10 │   │   end
11 │   │   if r ← receiveResult() then
12 │   │   │   if isOutputResult(r) then
13 │   │   │   │   numGraphs ← numGraphs - 1;
14 │   │   │   │   sendToOutputStream(r);
15 │   │   │   else
16 │   │   │   │   instructions ← pool.updateTokens(r);
17 │   │   │   │   for i in instructions do
18 │   │   │   │   │   if isFireable(i) then  sendToInterpreters(i) ;
19 │   │   │   │   end
20 │   │   │   end
21 │   │   end
22 │   end
```

In addition to arbitrary graphs, we also provide the possibility to specify skeleton applications. Currently, we support *pipeline*, *farm*, *map* and *reduce* skeletons [8]. These skeletons will then be actually compiled into suitable Macro Data Flow graphs.

2.1. *Implementation*

As common in many dataflow environments [3, 25], we implemented the runtime support through a master-worker pattern, with the master running the 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 enables the possibility to define 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 [12]. This is also confirmed by this work (Section 5.2, application characterised by very fine-grained computation).

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 [13]. 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.
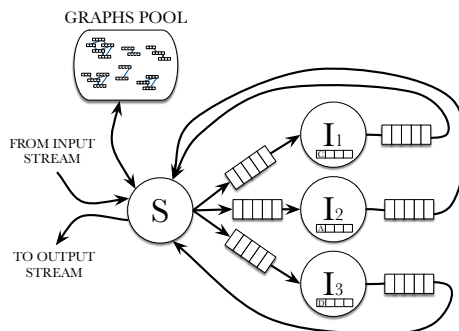

Fig. 2.   The runtime architecture. Node 'S' is the scheduler. Nodes 'I' are the interpreters.

When one of the interpreters in the pool finishes the execution of an instruction, it inserts the results into a lock-free single-producer, single-consumer FIFO queue [2][a]. The results will then be read by the scheduler and used to update the corresponding dataflow instructions stored in the graph pool. If no elements are present on the input stream, the scheduler must not wait for new elements to arrive. Indeed, meanwhile new results could arrive on the feedback channels from the interpreters and those results need to be managed. Similarly, the scheduler must not wait on the feedback queues if they are empty. To implement a non-deterministic read from these channels, the interactions with both channels has been implemented through non-blocking mechanisms. Accordingly, if no elements are present on the input stream the *receive* call will immediately return and the scheduler will check the feedback channels (and vice versa). This execution model is inspired to Kahn Process Networks [18].

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 that interpreter 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,

---

[a]To simplify the exposition, from a logical point of view, we often refer to these queues as if they where a single multi-producer, single-consumer queue.

it will send it to the first available interpreter (I_1). 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.
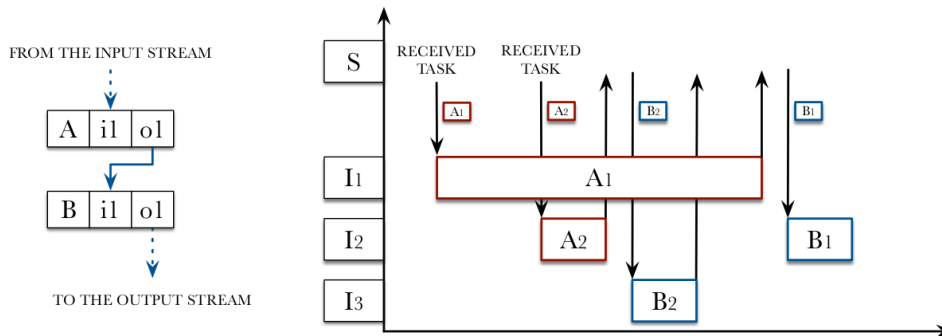

Fig. 3.   Reordering of the instructions.

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 with 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. Moreover, since the feedback channels are FIFO as well, even the next instruction in the graph will be processed in the correct order. 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. Moreover, note that the FIFO constraint on the queues between the scheduler and the interpreters is only needed in this case and that it could be relaxed in scenarios where the input stream elements do not need to be processed in the same order they are received.

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 a priority queue, where the priority is the identifier of the result (lower identifiers represent higher priority). The head of the queue is periodically checked and sent on the output stream only if it is the next element to be sent.

Macro Data Flow support could also be a part of a bigger 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. In this work we focus on minimising power consumption rather than energy consumption. Indeed, energy is obtained by multiplying the average power consumption by the execution time of the application. However, since we are considering streaming applications and since they are often characterised by an infinite input stream and thus an *infinite* execution time, it would not be possible to optimise energy consumption but only instantaneous power consumption. To face this problem, we used the algorithm described in [10]. This work differs from [10] for the following reasons: i) In [10] the algorithm was only applied for a very specific, non dataflow, application. In this paper we exploit the algorithm over the runtime of a programming model, analysing the problems of

using such reconfiguration algorithm to manage general dataflow applications. ii) We improved the algorithm to consider explicit performance requirements in terms of bandwidth or execution time, while in [10] it was only used to express utilisation requirements. iii) We studied the effectiveness of the algorithm in presence of an external disturbance (Section 5.3). iv) The power consumption model is improved to explicitly consider the voltage, while in [10] the voltage was approximated by using a cubic function.

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. Changes in ρ may be caused by fluctuations of T_A (Sections 5.1 and Section 5.2) as well as changes in T_S (Section 5.3). T_S may change due to internal factors (e.g. a phase change of the application), or by external interferences (e.g. other applications executed on the system).

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 [20], 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. The scheduler runs on a separate core when its ρ is close to 1. When the scheduler utilisation is low, it may run on a physical core shared with one of the interpreters (if there are no available cores). In this case, since the utilisation is low, the scheduler will not interfere with the interpreter execution. In fact, to further reduce the contention on this shared core, the scheduler properly relax its loop execution to avoid continuous active wait on empty input channels.

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 interpreters. 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 [10]:

$$\rho(\omega, \pi) = \rho(\overline{\omega}, \overline{\pi}) \times \frac{\overline{\omega} \times \overline{\pi}}{\omega \times \pi} \tag{1}$$

$\rho(\overline{\omega}, \overline{\pi})$, $\overline{\omega}$ 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. If this is not the case, the algorithm will move to a suboptimal configuration. However, this will be detected and the model will be re-evaluated, selecting a new configuration. At each step the algorithm will get closer to the optimal configuration, eventually reaching it. However our approach is generic and, if needed, more complex models could be used in order to let it converge faster.

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 [7, 4], 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. To simplify the exposition, we are assuming to run all the cores at the same frequency/voltage. However, the model could be applied to each frequency/voltage island separately. The algorithm would then consider as estimation the sum of the individual estimations.

This reconfiguration strategy has been implemented by using NORNIR [14][b], an autonomic support for parallel 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 con-

---

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

sumption 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. 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. Configuration changes are completely transparent both to the application programmer and to the final user 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[b].

Explicit performance requirements Beside being used to optimise utilisation of the resources, the presented algorithm can also be used to require explicit performance requirements in terms of minimum bandwidth or maximum execution time. Indeed, denoting with B the bandwidth of the application, starting from Equation 1 we can predict the bandwidth in different configuration as: $B(\omega, \pi) = B(\overline{\omega}, \overline{\pi}) \times \frac{\omega \times \pi}{\overline{\omega} \times \overline{\pi}}$. This equation holds under the same assumptions and conditions described for Equation 1. In this case, if B_{req} is the minimum bandwidth required by the user, the algorithm will find the lowest power consuming configuration $< \omega, \pi >$ such that $0 < \frac{B(\omega,\pi) - B_{req}}{B_{req}} < 0.05$, i.e. a configuration able to sustain a bandwidth that is at most 5% higher from the target one. If no solutions fall in this range, the algorithm will select the closest to the user requirement (preferring those with $B(\omega, \pi) > B_{req}$). If s is the number of elements in the input stream, the user may set a maximum execution time requirement by using the equation $T(\omega, \pi) = \frac{s}{B(\omega,\pi)}$. An example of this scenario will be presented in Section 5.3.

4. Programmability Aspects

In this Section we describe the steps to be performed by the programmer to transform an existing sequential application into a parallel dataflow and to specify performance constraints, taking as example the STRASSEN algorithm for multiplying two matrices.

First, the programmer analyses the sequential code (Figure 4 (1)), extracting the dataflow dependencies in the algorithm (Figure 4 (2)). After that, he can create the corresponding dataflow graph, which instructions will be executed in parallel by the runtime (Figure 4 (3)). 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 the stream and
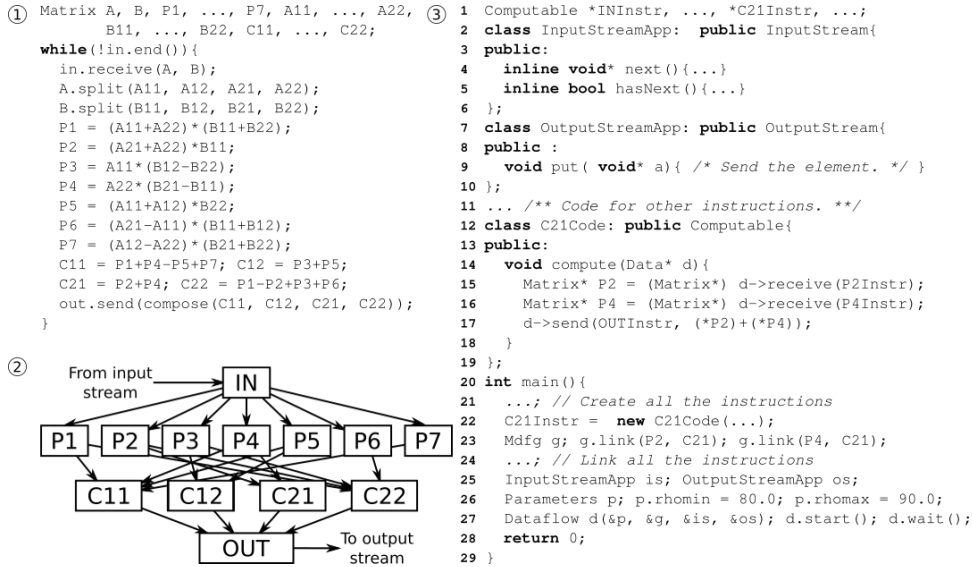
12   *Parallel Processing Letters*

①
```
Matrix A, B, P1, ..., P7, A11, ..., A22,
        B11, ..., B22, C11, ..., C22;
while(!in.end()){
  in.receive(A, B);
  A.split(A11, A12, A21, A22);
  B.split(B11, B12, B21, B22);
  P1 = (A11+A22)*(B11+B22);
  P2 = (A21+A22)*B11;
  P3 = A11*(B12-B22);
  P4 = A22*(B21-B11);
  P5 = (A11+A12)*B22;
  P6 = (A21-A11)*(B11+B12);
  P7 = (A12-A22)*(B21+B22);
  C11 = P1+P4-P5+P7; C12 = P3+P5;
  C21 = P2+P4; C22 = P1-P2+P3+P6;
  out.send(compose(C11, C12, C21, C22));
}
```

②

③
```
1  Computable *INInstr, ..., *C21Instr, ...;
2  class InputStreamApp:  public InputStream{
3  public:
4    inline void* next(){...}
5    inline bool hasNext(){...}
6  };
7  class OutputStreamApp: public OutputStream{
8  public :
9    void put( void* a){ /* Send the element. */ }
10 };
11 ... /** Code for other instructions. **/
12 class C21Code: public Computable{
13 public:
14   void compute(Data* d){
15     Matrix* P2 = (Matrix*) d->receive(P2Instr);
16     Matrix* P4 = (Matrix*) d->receive(P4Instr);
17     d->send(OUTInstr, (*P2)+(*P4));
18   }
19 };
20 int main(){
21   ...; // Create all the instructions
22   C21Instr =  new C21Code(...);
23   Mdfg g; g.link(P2, C21); g.link(P4, C21);
24   ...; // Link all the instructions
25   InputStreamApp is; OutputStreamApp os;
26   Parameters p; p.rhomin = 80.0; p.rhomax = 90.0;
27   Dataflow d(&p, &g, &is, &os); d.start(); d.wait();
28   return 0;
29 }
```

Fig. 4.   Steps required to parallelise the sequential STRASSEN application.

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 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 11-19). 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 21-22) and linked together (lines 23-24). Eventually, optional parameters may be specified (line 26) and the runtime can be created and started (line 27).

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. 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. 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. The compiler will ensure that the output type of a stage is compatible with the input type of the successive stage.

```
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 $\langle \rho_{min}, \rho_{max} \rangle = \langle 0.8, 0.9 \rangle$. Different values could have a different impact on the performance of the algorithm. A narrower range would cause the algorithm to be more sensitive to changes, while at the same time being more unstable. Similarly, shifting the range closer to 1.0, will lead to a lower power consumption. However, we could experience some performance degradation due to temporary performance fluctuations characterised by $\rho > 1$. We experimentally found the range $\langle 0.8, 0.9 \rangle$ to be a good tradeoff between sensitivity, power efficiency and performance.

5.1. *ffProbe*

The first application we used is FFPROBE [11], a parallel implementation of a Net-Flow 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 verified. 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. As shown in [11], this parallel implementation significantly improves the performance in terms of processed network packets per time unit, achieving 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 applications [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

14   *Parallel Processing Letters*

of a real network[c], covering a 24 hours span. To model future network scenarios, this rate has been linearly scaled up in order to increase the 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 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. 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. Since we have a graph composed by 20 instructions, the runtime can activate 1, 2, 4, 5, 10 or 20 interpreters. In this application, the instructions have an average latency of 3 milliseconds.
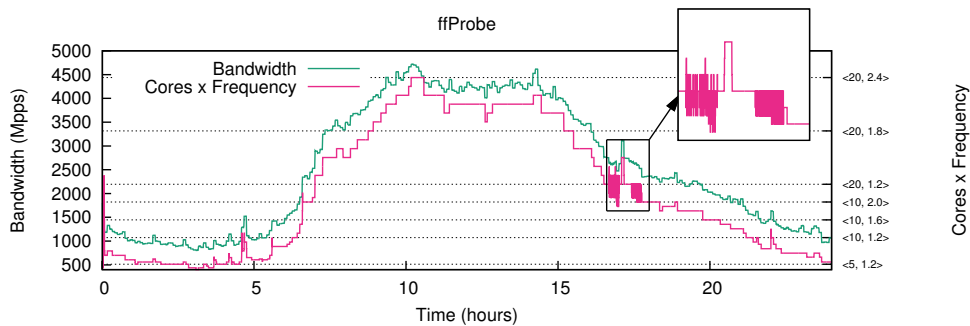

Fig. 5.   Input bandwidth (Millions Packets Per Second) and used resources for FFPROBE.

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

[c]http://bit.ly/1RY7fEt - The used rate is the one collected on 08 Jun 2005
[d]Processed bandwidth is not shown since the application is always able to process all the data

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.
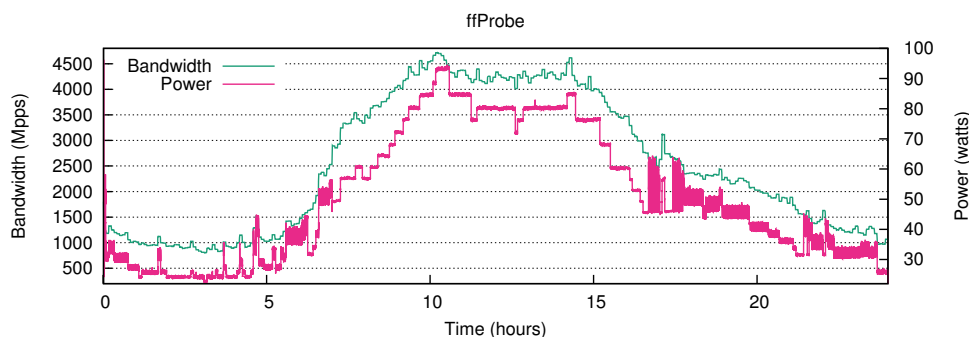

Fig. 6.   Input bandwidth (Millions Packets Per Second) and power consumption for ffProbe.

An alternative solution 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 average power with respect to our solution (which uses the `userspace` governor). This happens because we operate both on the number of cores and their frequency, 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, avoiding the unbalancing problems that characterise the ffProbe application. In our experiments we used the PARSEC `native` input and the input workload of a trading day of the NASDAQ market[e].

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

In this application, the instructions have an average latency of 0.3 milliseconds. 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. Both OPENMP and INTEL TBB implementations use the `parallel for` construct, while the PTHREAD implementation is an hand-written *map* skeleton. In the following table we show the average normalised value (between 0 and 1) obtained over 10 different executions of the experiment.

DATAFLOW	PTHREADS	OPENMP	INTEL TBB
1	0.87	0.83	0.95

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.
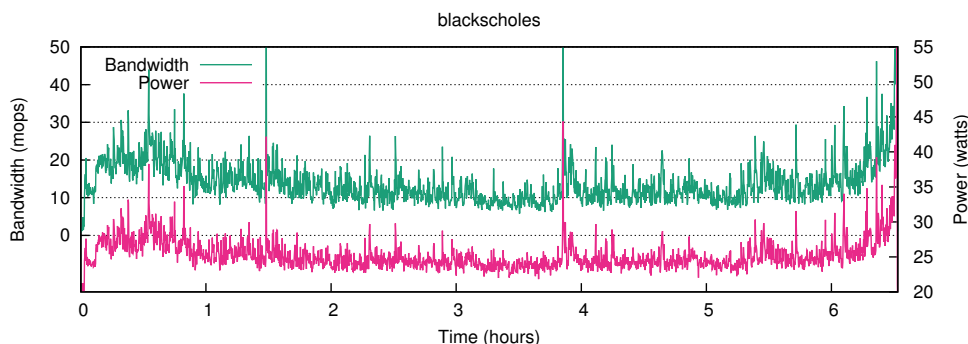

Fig. 7.   Input bandwidth (Millions Opt. Per Second) and power consumption for BLACKSCHOLES.

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 ρ between the specified bounds ($80\% < \rho < 90\%$).

5.3. *Strassen*

The last application we used to validate our solution, consists in computing the Strassen matrix multiplication algorithm [26] over a stream of matrices. This algorithm divides both the input matrices in 4 parts and applies several sum, differences and multiplications to these submatrices to get the final result. The application can be expressed as a non-structured dataflow graph (Figure 4 (2)). In this application, the instructions have an average latency of 60 milliseconds.

In this experiment we have a stream of 4000 matrices and we required a maximum execution time of 200 seconds. Moreover, after 100 seconds from the beginning,
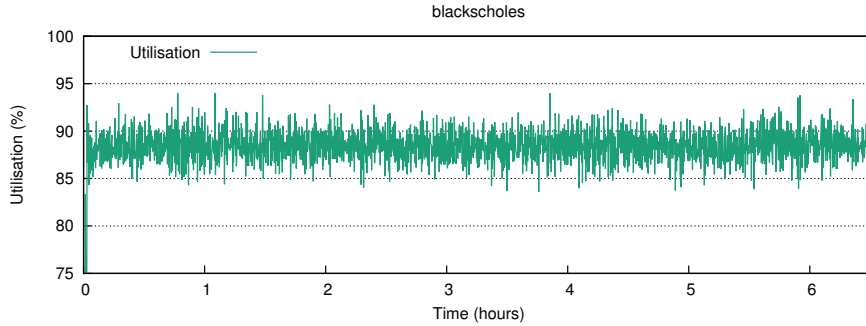
Fig. 8.   Resources utilisation for BLACKSCHOLES.

an external application is started (parallel data compression with 24 threads).
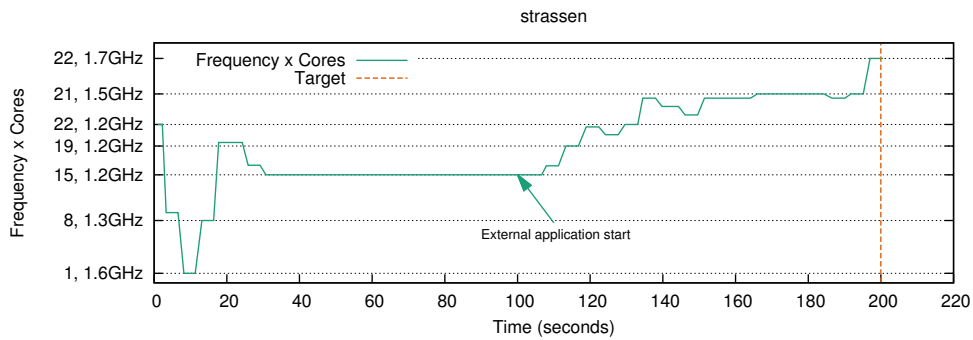

Fig. 9.   Resources allocated to Strassen application with a required completion time of 200s.

As shown in Figure 9, around second 20, the runtime automatically selects a configuration satisfying the user requirement. At second 100, the performance of STRASSEN application decrease and the runtime the amount of resources used by the application are increased, allowing it to terminate its execution within the target.

6. Related Work

Dataflow is a computing model that has been around since the earliest days of computer science research activities [18]. Many programming environments currently use dataflow concepts in the implementation of different parallel programming models. StreamIT [28] is a programming language and a compilation infrastructure, specifically engineered for modern streaming systems. However, power-awareness is still not provided. OpenMP [6] and some recent extensions [23] 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. OpenMP 4.0 also introduces mechanisms for synchronising tasks

by dataflow dependencies. StarPU [5] supports parallel activities graphs in essence similar to Macro Data Flow graphs and executes these graphs on heterogeneous architectures (multi-core + GPU) by using an homogeneous programming abstraction. More recently, Microsoft TPL[f] and Google's TensorFlow[g] have been proposed and extensively used in commercial solutions. However, many of these programming environments do not provide any structured parallel programming abstraction.

S-Net [22] is a mature dataflow model, providing static soundness guarantees and recursive graphs. Despite such features are not provided by our framework, 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 have been built over S-Net, they have some limitations. For example, in [19], 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. Similarly, in [16] the author proposes an algorithm to allocate resources to applications according to their real needs. Different from our approach, they can mediate resource demands between different applications. However, the impact of frequency scaling is not modelled and it is not possible to express explicit performance requirements.

Similar to our work, in [17] a solution to minimise power consumption under a given performance constraint for dataflow applications is presented. However, only explicit performance requirements (i.e. minimum bandwidth) can be specified and there is no way to implicitly optimise utilisation of resources according to input bandwidth fluctuations. Moreover, this solution requires information extracted through a static analysis of the application structure, by annotating each phase of the application with the maximum parallelism degree achievable. However, even inside the same phase the maximum amount of parallelism that can be exploited could significantly change due to fluctuations in the input data arrival or due to interferences caused by other applications. Our approach consider these dynamic scenarios (as shown in Section 5.3) and, since it is agnostic from the specific appli-

---

[f]https://msdn.microsoft.com/en-us/library/hh228603(v=vs.110).aspx
[g]https://www.tensorflow.org/

cation structure, could potentially work on graphs with a structure that dynamically changes through the execution.

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 three 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, for example by using C++ *pragmas* or *attributes*. Moreover, we will investigate the possibility to have quantitative power consumption estimations, in order to express explicit power consumption constraints. Lastly, we would like to analyse possible solutions for balancing the load, particularly when instructions are statically assigned to interpreters and the number of interpreters changes during application execution.

References

[1] Aldinucci, M., Danelutto, M., Kilpatrick, P.: Autonomic management of non-functional concerns in distributed and parallel application programming. In: Proc. of IPDPS 2009, pp. 1–12. IEEE, Rome, Italy

[2] Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: Proceedings of the 18th International Conference on Parallel Processing, Euro-Par'12, pp. 662–673. Springer-Verlag, Berlin, Heidelberg (2012)

[3] Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in Java. Future Gener. Comput. Syst. **19**(5), 611–626 (2003)

[4] 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)

[5] 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)

[6] 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)

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

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

20 *Parallel Processing Letters*

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

[10] Danelutto, M., De Sensi, D., Torquati, M.: Energy driven adaptivity in stream parallel computations. In: Proc. of PDP 2015, pp. 103–110 (2015)

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

[12] Danelutto, M., Torquati, M.: Loop parallelism: A new skeleton perspective on data parallel patterns. In: Proc. of PDP 2014, pp. 52–59 (2014)

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

[14] De Sensi, D., Torquati, M., Danelutto, M.: A reconfiguration algorithm for power-aware parallel applications. ACM Trans. Archit. Code Optim. **13**(4) (2016)

[15] 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)

[16] Grelck, C.: Active resource management for multi-core runtime systems serving malleable applications. In: Proc. of PARCO 2015 (2015)

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

[19] 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)

[20] Li, J., Martínez, J.F.: Dynamic power-performance adaptation of parallel computation on chip multiprocessors. Proc. - Intl. Symposium on High-Performance Computer Architecture **2006**, 77–87 (2006)

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

[22] 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)

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

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

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

[26] Strassen, V.: Gaussian elimination is not optimal. Numer. Math. **13**(4), 354–356 (1969)

[27] 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)

[28] Thies, W., Karczmarek, M., Gordon, M.I., Maze, D.Z., Wong, J., Hoffman, H., Brown, M., Amarasinghe, S.: Streamit: A compiler for streaming applications. Technical Report MIT/LCS Technical Memo LCS-TM-622, MIT, Cambridge, MA (2001)