September 2016

State-Aware Concurrency Throttling

Daniele DE SENSI [a,1], Peter KILPATRICK [b] Massimo TORQUATI [a]
[a] *Computer Science Department, University of Pisa*
[b] *Computer Science Department, Queen's University Belfast*

Abstract. Reconfiguration of parallel applications has gained traction with the increasing emphasis on energy/performance trade-off. The ability to dynamically change the amount of resources used by an application allows reaction to changes in the environment, in the application behavior or in the user's requirements. A popular technique consists in changing the number of threads used by the application (*Dynamic Concurrency Throttling*). Although this provides good control of application performance and power consumption, managing the technique can impose a significant burden on the application programmer, mainly due to state management and redistribution following the addition or removal of a thread. Nevertheless, some common state access patterns have been identified in some popular applications. By leveraging on this knowledge, we will describe how it is possible to simplify the state management procedures following a Concurrency Throttling operation.

Keywords. Power-Aware Computing, Concurrency Throttling, Data Stream Processing

1. Introduction and Motivation

In recent years power consumption management has become a major concern for data center operators due to economic cost, reliability problems and for environmental reasons [16]. To dynamically control the power consumption of an application, a commonly used approach consists in changing the amount of resources allocated to the application (i.e. its *configuration*). This can be achieved by acting on the number of cores allocated to the application [3], on the clock frequency of these cores [2], on the types of cores used by the application [18] or on a combination of those [11]. The application is monitored throughout its execution and the configuration is adapted by considering the user preferences, the interference caused by external factors (e.g. other applications) and possible workload fluctuations.

Concurrency throttling is one of the most commonly used techniques [7,4,17,19] and consists in dynamically changing the number of threads used by the application. By reducing the number of threads, if we had one thread for each core, the application may release some cores, thus allowing the operating system to shut them down and so reduce the overall power consumption of the computing architecture. In some cases, like for example in OPENMP, this can only be done between two parallel sections (e.g. between two different *parallel for*s). On the other hand, other approaches apply this technique

[1] Corresponding Author: Daniele De Sensi, Computer Science Department, University of Pisa, Italy; E-mail: desensi@di.unipi.it.

inside a single parallel section, requiring, however, a proper management of the internal state [11].

An alternative approach, is *thread packing* [2]. Instead of changing the number of threads, they are simply moved onto a smaller set of cores, not requiring any user/programmer intervention. However, as we have shown in [6] this technique is usually less efficient. For example, consider the scenario depicted in Figure 1, outlining the performance of the CANNEAL [5] application when a different numbers of cores are used. In this case, due to contention on shared resources, activating the maximum number of threads is not the best solution, since the best performance is obtained by using 128 threads. However, the maximum performance peak is only achievable by using *concurrency throttling*, i.e. by reducing the number of threads from 256 to 128. The same cannot be achieved by thread packing, i.e. by placing 256 threads on 128 cores, as shown in Figure 1, since there would still be high contention. In addition to that, there are situ-

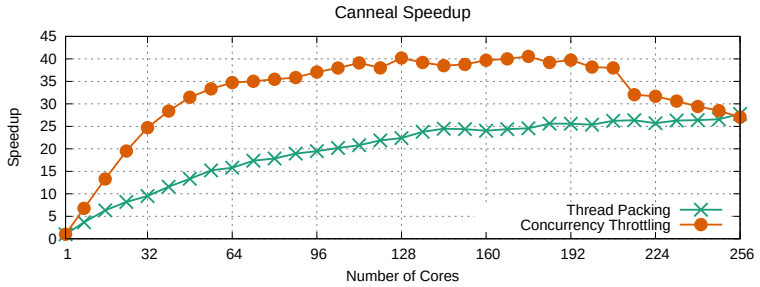

Figure 1. Comparison of *concurrency throttling* and *thread packing* on `canneal` application.

ations where the thread packing *black-box* approach is not sufficient, and we need to operate on individual parts of the application. Consider for example Figure 2, showing the structure of the FERRET application, distributed with the PARSEC benchmark [1]. This application is structured as a pipeline, where some of the stages are replicated between multiple threads (*tread pool*). In this situation, we need to be able to operate on each individual stage. For example, if the *Index* stage is too fast and is spending most of its time waiting for data, we may wish to remove a thread from its pool, thus releasing a core. However, if *thread packing* is used, we would just remove a core from the application, without any guarantee that this would be removed from the *Index* stage. The ability to distinguish different parts of the application is important and can only be achieved by directly changing the number of threads of (a part of) the application.

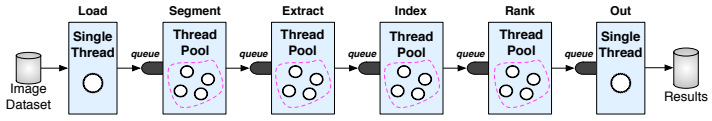

Figure 2. Ferret application structure.

A similar structure can be found in other co-running applications (e.g. DEDUP in the PARSEC benchmark). Thread packing, however, can still be effectively used on some applications, e.g. on *master-worker* or *thread pool* applications where (almost) all the threads are homogeneous.

Dynamically changing the number of threads used by the application is a complex and costly operation, and may require some support from the application programmer. Consider for example a parallel application for computing the sum of two matrices. In this case, we would split the matrices between the threads and then each thread would compute the sum over its submatrix. If we dynamically add a thread during the computation, we need to redistribute the two matrices to let the new thread have some work to perform. This means that the application must be paused, the thread must be added, and the remaining part of the computation needs to be redistributed among the new number of threads. Eventually, the application can be resumed. Thus, even for such a simple case, applying concurrency throttling requires extra effort from the programmer (to compute the new partitions) and may also lead to performance issues (due to the pause/resume of the application). These problems are exacerbated for more complex applications where interactions and synchronization among threads is not so trivial. If the application is programmed by using a dataflow execution model, the number of threads is usually always reconfigurable since the reconfiguration occurs at the runtime level and the structure of the application specified by the programmer is not modified. However, in this case the problem is simply shifted from the application programmer to the programmer of the dataflow support, as shown in [8].

Recently there has been an attempt to establish a firm basis for developing stateful parallel applications by classifying them according to their *state access pattern* [9]. In a nutshell, six patterns have been identified for modelling typical stateful stream parallel applications. They can be classified by the extent and way in which the state can be structured and accessed by the application. By optimizing each individual pattern, the optimization will automatically benefit all applications characterized by a given pattern. Although they have been presented as patterns for streaming applications, they could actually be applied to any iterative application, by considering as a stream element the element processed at a specific iteration. State access patterns are orthogonal to parallelism exploitation patterns (i.e. *parallel patterns* or *skeletons*). Indeed, while the former express the way in which the state is accessed by the different modules composing the parallel application, the latter express how these modules are organized and connected to each other.

The main contribution of this work is to provide a set of techniques and appropriate programming interfaces for managing state access and concurrency throttling in stateful streaming applications (i.e. classic streaming applications with non-trivial internal state), thus reducing the effort required to the programmer. A trivial solution would be to just stop the application, perform all the actions required to redistribute the internal state and then resume the application. However, this is a costly operation and we would like to exploit the knowledge about the structure of the application to implement pattern-specific reconfiguration procedures to minimize the performance impact. Concurrency throttling hints for stateful streaming patterns were described briefly in [9]. In the current work we will analyze more deeply the implications of applying concurrency throttling, providing pattern-specific reconfiguration designs, together with an appropriate C++ interface. We believe that this is an important step in providing appropriate reconfiguration mechanisms, due to the increasing effort carried out by both research [11,2,10] and industrial communities [2] [3] in building applications able to dynamically adapt themselves.

[2] `https://aws.amazon.com/autoscaling/getting-started/`

[3] `https://cloud.google.com/compute/docs/autoscaler/`

The rest of this paper is structured as follows. Section 2 describes some existing state access patterns and provides an appropriate API to reduce the burden of state reconfiguration on the programmer. Section 3 outlines techniques that can be applied for managing concurrency throttling and Section 4 presents some experimental results. In Section 5 we outline related work and, finally, in Section 6 we draw conclusions and outline some possible future directions for this work.

2. State Access Patterns

In this Section we will briefly outline some of the state access patterns described in [9]. For each of these patterns, we will propose in Section 3 some possible strategies for applying *concurrency throttling*.

2.1. All-or-None Pattern

This pattern occurs in streaming applications where some input stream elements can modify the internal state while some other elements trigger a read-only operation. The rationale is that elements triggering read-only operations can be executed in parallel, while elements modifying the internal state must be executed sequentially in order to preserve the semantics of the application. For example, in a financial auto-quoting system some operations (typically less frequent) may modify the market state (i.e. buy and sell proposals), which must be executed in their arrival order, while more frequent operations are queries that do not modify the market state (e.g., search for the best proposals for a given stock symbol). Accordingly, the user needs to specify a predicate $\mathscr{P}$ such that $\mathscr{P}(x_i) = true$ if the stream element x_i modifies the internal state and $\mathscr{P}(x_i) = false$ otherwise. This pattern may be implemented with a *master-worker* parallel pattern, and by sharing the internal state between the master and the workers. When a stream element x_i is received, if $\mathscr{P}(x_i) = false$, it is scheduled to one of the workers. If $\mathscr{P}(x_i) = true$, the computation on the element will be executed by the master. However, since this element will modify the internal state, such computation can only start after all the previous stream elements have been executed (since they need to be executed on the old state). To ensure this it is sufficient to keep a global atomic counter of the elements currently in the application (both enqueued to the workers or currently processed). The master will increment this counter every time that an element is scheduled to a worker. Each worker will decrement the counter after it terminates the execution of an element. When the counter is equal to zero, the master can start the execution of the element that modifies the internal state, since all the previous stream elements have already been processed in the proper order.

The pattern interface would be as follows:

```
1 template <typename S, typename IN, typename OUT> class AllOrNone{
2     S _state;
3 public:
4     AllOrNone(bool (*predicate)(const IN& element),
5               OUT (*readOnly)(const S& state, const IN& element),
6               OUT (*readWrite)(S& state, const IN& element));
7 };
```

The pattern is instantiated by providing the `predicate` $\mathscr{P}$ the `readOnly` operation and the operation that can modify the internal state (`readWrite`). When instantiating the pattern, the types of the input (`IN`) and output (`OUT`) streams are specified, as well as the state type.

2.2. Fully Partitioned Pattern

In applications characterized by this state access pattern, the state can be modeled as a vector v[4], where each element accesses only a specific location of the vector. To decide which element $v[i]$ is accessed by the stream element x_i, a function $\mathscr{H}$ (effectively a hash function) must be provided by the user, such that element $v[\mathscr{H}(x_i)]$ is accessed. Another condition holding for such applications is that if two stream elements access the same vector location, they must be executed in the order they are received by the application. This pattern can be implemented with a *master-worker* parallel pattern. Each worker manages a contiguous partition of the state, such that state element $v[i]$ is managed by worker $\lfloor \frac{i}{z} \rfloor$, where z is the size of a partition of the vector. If s is the size of the vector, we have $z = \frac{s}{n_w}$. By doing so, the master can schedule an element x_i to the worker with identifier $\lfloor \frac{\mathscr{H}(x_i)}{z} \rfloor$. In typical implementations [7], the master is connected to each worker through a shared FIFO message queue. The FIFO property ensures that two elements accessing the same state element will be processed in the order in which they have been received by the application. The state will actually be shared between all the workers, but since the master properly schedules the stream elements, each worker will always access only a contiguous partition of the state.

A possible programming interface for this pattern is shown in the following code snippet.

```
1 template <typename S, typename IN, typename OUT> class FullyPartitioned{
2     std::vector<S> _state;
3 public:
4     FullyPartitioned(uint (*hash)(const IN& element),
5                      OUT (*compute)(S& state, const IN& element));
6 };
```

Even in this case, to instantiate the pattern it is necessary to specify the types of the input (`IN`) and output (`OUT`) streams and of the element of the state (in this specific example we consider one-dimensional state). A `hash` function must be specified, taking as input an element of the input stream and producing the hash value for that element. The function `compute` performs the computation on a stream element, producing an output value to be sent to the output stream.

2.3. Successive Approximation State Access Pattern

Applications characterized by this pattern manage a global state which, for each element of the stream, is subsequently approximated. The new state approximation is computed as a predicate $s_{new} = \mathscr{S}(x_i, s_{curr})$, where the function $\mathscr{S}$ is specified by the user and could, in some cases, return the current state without performing any modification[5]. The

[4] In the most general case it is modeled as a multi-dimensional array.

[5] The definition of this pattern is more compact and slightly different than the definition in [9].

function $\mathscr{S}$ must be *monotonic* and must converge towards a specific value. For example, it could be monotonically decreasing, i.e. $s_{new} = \mathscr{S}(x_i, s_{curr}) < s_{curr}$. Updates to the global state may occur in any order but are executed under mutual exclusion. Different implementations are possible. In this paper, we consider an implementation which is different from that presented in [9]. In this implementation, we use a *master-worker* pattern, where the state is globally shared and where each *worker* executes the function $\mathscr{S}$ on its stream element and on the global state. In case of monotonically decreasing $\mathscr{S}$, if the function evaluation returned a value s_{new} such that $s_{new} > s_{curr}$, the state is not updated. s_{curr} is a global variable shared by all the workers and accessed in mutual exclusion.

A possible programming interface for this pattern could be:

```
1 template <typename S, typename IN, typename OUT>
2 class SuccessiveApproximation{
3     S _sCurr;
4 public:
5     SuccessiveApproximation(OUT (*compute)(S& state, const IN& element));
6 };
```

The `sCurr` variable contains the current state approximation, which can be modified by the `compute` function, and which will return the element to be sent to the output stream. The pattern will ensure that the state will be rolled back to the previous state if the function `compute` did not improve the approximation.

3. State Aware Concurrency Throttling

In this Section we describe some possible techniques to be adopted for concurrency throttling on applications using the described state access patterns. In all cases, it is implicit that each part of the parallel application knows the number of currently active threads.

3.1. All-or-none pattern

To increase the number of threads, it is sufficient to communicate this decision to the master. It will start a new thread and will begin scheduling the new elements to it. This can be done at any time during the execution, without any additional programmer intervention. To remove a worker, the master will send it a special stream element (e.g. a NULL pointer) and will not send any other element to that worker. Since the elements are processed in FIFO order, when the worker receives this special message we are sure it has already processed the previous elements which were scheduled to it, so it can stop itself, thus releasing a core.

This reconfiguration strategy preserves the correctness of the application since the updates to the internal state still occur in the correct order. Indeed, when a new worker is added, it will start updating the shared variable counting the elements received by the application but still to be processed. Similarly, when a worker is removed, the master will only update the state when the counter is zero, and this can only happen after the worker to be removed terminates the processing on its incoming elements. It is worth noting that it is possible to apply this reconfiguration strategy only because of the knowledge about how the internal state is accessed by the different parts of the application and could not be applied in an arbitrary application.

3.2. Fully Partitioned Pattern

When dynamically changing the number of workers, we must ensure that the state is repartitioned among the new number of workers. To do so, it is sufficient to change the variable n_w, thus reflecting the new scheduling[6]. However, before changing this variable, we must ensure that all the elements which were already scheduled have been processed. Consider the simple case where we have a state vector composed by 4 elements and two active workers. In this situation, the first worker will access the first 2 elements of the vector while the second worker will access the remaining two elements. Assume also that the master has already enqueued an element with $H(x_i) = 3$ on the message queue of the second worker. However, it still has not extracted the message from the queue. In the meantime, a request to add two workers arrives and a new element with $H(x_{i+1}) = 3$ is received by the master. This element would be now scheduled to the fourth worker, which will execute the computation on it. In the meantime, the second worker extracts element x_i from the queue and executes the computation on it. As is evident, this violates the requirement of ordered execution. To avoid this, it is sufficient to enforce that the previous elements present in the application are processed before changing the number of workers. This would introduce a performance penalty since all the elements already present in the message queues must be processed before starting scheduling new elements. We will show experimental results relating to this issue in Section 4.

3.3. Successive Approximation State Access Pattern

To add or remove a worker, no additional operations need to be carried out. Indeed s_{cur} is globally shared among all the threads.

4. Experiments

In this Section, we present experimental results for the performance of some of the state-aware concurrency throttling strategies.

4.1. Fully Partitioned Pattern

As anticipated in Section 3, when an application characterized by a fully partitioned state needs to be reconfigured, we first need to process all the elements already enqueued to the different workers. This introduces a penalty, which we verified to be proportional to the queue size, by evaluating it on a real network monitoring application [7]. Results are not shown here due to space constraints. Having small queues would be beneficial for reconfiguration latency. However, larger queues can better accommodate workload fluctuations. Finding the optimal message queue size is a complex problem and outside the scope of this work. In Figure 3 we give some results presented in [11], showing that by using a state-aware concurrency throttling we are able to have a power consumption which is proportional to the application performance. In that case the application was characterized by a fully partitioned state access pattern, albeit it was implemented

[6] On distributed memory systems, we also need to move parts of the state between workers. Some solutions to this problem exist [10].

from scratch. Having a proper abstraction would significantly reduce the time required to develop such an adaptive application.

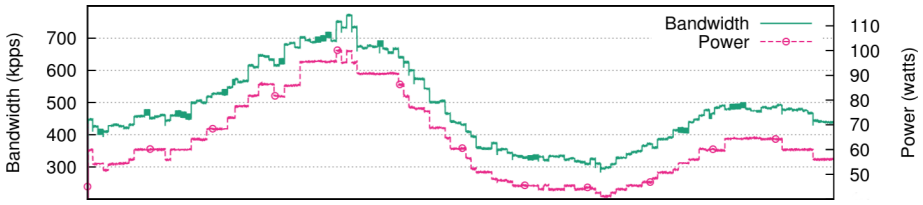

Figure 3. Benefit of using a state-aware concurrency throttling.

4.2. Successive Approximation State Access Pattern

For this experiment, we take as example the `canneal` application, which minimizes the routing cost of a chip design. Specifically, we consider the implementation described in [5]. This is not a streaming application but, since it is an iterative application, the presented state access patterns can still be applied. In this implementation, each iteration must be performed by all the *workers*. Since there is no stream, the *master* will simply ask the *workers* to perform a specific iteration, until there are no more iterations to perform. The *master* decides that there are no more iterations to perform if a convergence condition is satisfied or if a specified number of iterations have been performed. The function $\mathscr{S}$ performed by the *workers* is also parametric in their number. This is not a problem since, as described in Section 3, we assume that all the parts of the parallel application always know how many threads are currently active. After each iteration, a *barrier* is performed and then the convergence conditions are checked by the *master*. The *barrier* is implemented by the *master*, which waits for an acknowledgment message from all the *workers*. When the number of threads is changed, this is reflected on the condition checked by the *master*, which will always wait for the correct number of acknowledgments.

We now show an example where an external *manager* controls the application to enforce a maximum power consumption. To perform this experiment we used NORNIR [11], a power-aware runtime support for parallel applications. Power consumption has been measured by using MAMMUT [12].

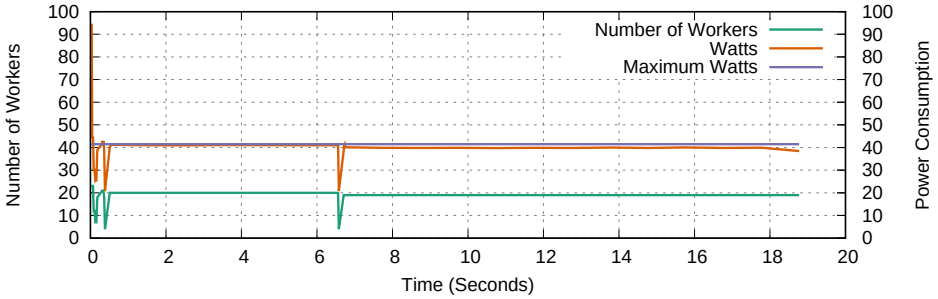

Figure 4. Benefit of using a state-aware concurrency throttling to enforce power consumption constraints.

In this specific experiment, we set a maximum power consumption of 41 Watts. As we can see from Figure 4, the runtime support at the beginning of execution activates a different number of *workers*, to select the optimal one. After this first exploration phase, the application stabilizes its power consumption at 41 Watts, by using 20 workers. Around 6 seconds from the start, this requirement is violated[7] and the runtime support does another small exploration and, eventually, activates 19 workers.

5. Related Work

With the emerging *Internet of Everything* paradigm, streaming computations have gained momentum. Data streams are often characterized by fluctuation on the input arrival rate with the presence of data bursts and cyclic phases [15].

In the context of long running streaming applications, managing such variability is paramount to provide a given Quality of Service (QoS) level using the smallest possible number of resources to reduce energy consumption and improve efficiency.

If parallel patterns such as *pipeline* and *task-farm* [20] are used to model the most critical parts of the streaming application, they have to provide auto-tuning capabilities or at least make available suitable interfaces to the programmer for managing these dynamic changes. Such dynamic behaviors, when targeting multicore systems, are often handled using approaches based on the dynamic adaptation of the parallel run-time system [10,8,7]. The most widely used techniques for optimizing power/performance on multicore platforms are *Dynamic Concurrency Throttling* [17], *Concurrency Packing* [2]. In [6] such techniques have been compared considering the PARSEC benchmark suite revealing that concurrency throttling is usually more efficient in the utilization of the resources, though it requires more programming effort.

In fact, changing the number of resources dynamically usually needs complex state re-organization protocols to move the application state without altering the computation semantics. Some research works, such as [14,13] focus on the state transfer in elastic stateful data streaming applications. However, the main problem of these techniques is that they are often difficult to implement if not directly provided by the runtime system.

Our approach leverages on parallel patterns modelling stateful streaming computation [9]. In this context, the runtime system is aware of the type of the internal state and is able to automatically re-organize the state when the resources change is triggered. Despite the extensive literature describing different techniques for dealing with dynamic resource change, to the best of our knowledge, no previous attempt was made to study state access patterns and concurrency throttling for stateful streaming applications with the objective to reduce the programming effort required by the programmer.

6. Conclusions and Future Work

In this work, we presented techniques to manage state rearrangement when *concurrency throttling* is applied on stateful streaming applications. We provided suitable programmer abstractions and presented some results highlighting the importance of *concurrency*

[7] This is not visible in the figure since the requirement is exceeded by less than 1 Watt.

throttling. As future work, we would like to study state-aware concurrency throttling for other state access patterns and to extend this work to a wider set of applications.

Acknowledgements This work has been partially supported by the EU H2020-ICT-2014-1 project REPHRASE (No. 644235).

References

[1] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[2] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack & Cap: adaptive DVFS and thread packing under power caps. In *Proceedings of the 44th Annual IEEE/ACM Intl. Symposium on Microarchitecture - MICRO-44 '11*, page 175, New York, New York, USA, Dec. 2011. ACM Press.

[3] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1396–1410, Oct 2008.

[4] M. Curtis-maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-based Power-Performance Adaptation of Multithreaded Scientific Codes. *IEEE Transactions on Parallel and Distributed Systems*, 2008.

[5] M. Danelutto, T. De Matteis, D. De Sensi, G. Mencagli, and M. Torquati. P^{3}ARSEC: Towards parallel patterns benchmarking. In *Proc. of the 32nd Annual ACM Symposium on Applied Computing*, 2017.

[6] M. Danelutto, T. De Matteis, D. De Sensi, and M. Torquati. Evaluating concurrency throttling and thread packing on smt multicores. In *Proc. of the 25th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing*.

[7] M. Danelutto, D. De Sensi, and M. Torquati. Energy driven adaptivity in stream parallel computations. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro Intl. Conf. on*, pages 103–110, March 2015.

[8] M. Danelutto, D. De Sensi, and M. Torquati. A power-aware, self-adaptive macro data flow framework. *Parallel Processing Letters*, 27(01):1740004, 2017.

[9] M. Danelutto, P. Kilpatrick, G. Mencagli, and M. Torquati. State access patterns in stream parallel computations. *The Intl. Journal of High Performance Computing Applications*, 0(0), 0.

[10] T. De Matteis and G. Mencagli. 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)*, pages 13:1–13:12, 2016.

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

[12] D. De Sensi, M. Torquati, and M. Danelutto. Mammut: High-level management of system knobs and sensors. *SoftwareX*, 6:150 – 154, 2017.

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

[14] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 23(12), Dec. 2012.

[15] S. Islam, S. Venugopal, and A. Liu. Evaluating the impact of fine-scale burstiness on cloud elasticity. In *Proc. of the Sixth ACM Symposium on Cloud Computing*, pages 250–261. ACM, 2015.

[16] J. G. Koomey. Worldwide electricity used in data centers. *Environmental Research Letters*, 3(3), 2008.

[17] J. Li and J. F. Martínez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. *Proc. Intl. Symposium on High-Performance Computer Architecture*, 2006.

[18] G. Liu, J. Park, and D. Marculescu. Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems. In *2013 IEEE 31st Intl. Conf. on Computer Design*, Oct 2013.

[19] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva. Comparison of Decision-Making Strategies for Self-Optimization in Autonomic Computing Systems. *ACM Transactions on Autonomous and Adaptive Systems*, 7(4):1–32, Dec. 2012.

[20] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.