CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:2073–2094
Published online 22 January 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1549

Adaptive structured parallelism for distributed heterogeneous architectures: a methodological approach with pipelines and farms

Horacio González-Vélez[1,*,†] and Murray Cole[2]

[1]*School of Computing and IDEAS Research Institute*, *Robert Gordon University*, *Aberdeen*, *U.K.*
[2]*School of Informatics*, *University of Edinburgh*, *Edinburgh*, *U.K.*

SUMMARY

Algorithmic skeletons abstract commonly used patterns of parallel computation, communication, and interaction. Based on the algorithmic skeleton concept, structured parallelism provides a high-level parallel programming technique that allows the conceptual description of parallel programs while fostering platform independence and algorithm abstraction. This work presents a methodology to improve skeletal parallel programming in heterogeneous distributed systems by introducing adaptivity through resource awareness. As we hypothesise that a skeletal program should be able to adapt to the dynamic resource conditions over time using its structural forecasting information, we have developed adaptive structured parallelism (ASPARA). ASPARA is a generic methodology to incorporate structural information at compilation into a parallel program, which will help it to adapt at execution. ASPARA comprises four phases: programming, compilation, calibration, and execution. We illustrate the feasibility of this approach and its associated performance improvements using independent case studies based on two algorithmic skeletons—the task farm and the pipeline—evaluated in a non-dedicated heterogeneous multi-cluster system. Copyright © 2010 John Wiley & Sons, Ltd.

Received 21 September 2008; Revised 7 October 2009; Accepted 19 October 2009

KEY WORDS: structured parallelism; algorithmic skeletons; parallel patterns; parallel programming; concurrent programming structures; cluster computing

*Correspondence to: Horacio González-Vélez, School of Computing and IDEAS Research Institute, Robert Gordon University, St Andrew Street, Aberdeen AB25 1HG, U.K.
†E-mail: h.gonzalez-velez@rgu.ac.uk

1. INTRODUCTION

Algorithmic skeletons abstract commonly used patterns of parallel computation, communication, and interaction [1]. While computation constructs manage logic, arithmetic, and control flow operations, communication and interaction primitives coordinate inter- and intra-process data exchange, process creation, and synchronization. Skeletons provide top-down design, composition, and control inheritance throughout the program structure. Parallel programs are expressed by interweaving parameterized skeletons analogously to the way in which sequential structured programs are constructed.

Known as structured parallelism, this design paradigm provides a high-level parallel programming methodology that allows the abstract description of programs and fosters portability by focusing on the description of the algorithmic structure rather than on its detailed implementation. This provides a clear and consistent behaviour across platforms, with the underlying structure depending on the particular implementation. That is to say, the skeleton behaviour refers to the outcome sought by the application programmer, and the skeleton structure concerns the resource to functionality correspondence established at the infrastructure level.

Therefore, by decoupling the behaviour from the structure of a parallel program, structured parallelism benefits entirely from any performance improvements in the system infrastructure, while preserving the program results.

Such behaviour–structure decoupling has allowed the structured parallelism paradigm to be seamlessly deployed on different multi-processor systems with distinct architectural models. Nonetheless, large non-dedicated heterogeneous multiprocessing systems have long posed a challenge to known distributed systems programming techniques, as a result of the dynamic nature of their resources. The conglomeration of dozens of cores per processor has just increased the complexity of the challenge at hand. Berkeley's view report has further highlighted the importance of not only producing realistic benchmarks for parallel programming models based on patterns of computation and communication, but also that of developing programming paradigms which efficiently deploy scalable, task parallelism [2].

It is widely acknowledged that one of the major challenges in programming support for large heterogeneous distributed systems is the prediction and improvement of the performance [3–6]. Such systems are characterized by the dynamic nature of their heterogeneous components, due to shifting patterns in the background load which are not under the control of the individual application programmer. In principle, it is expected that efficient parallel applications must be aware of the system conditions, and adapt their execution according to variations in the available computation and communication resources. The challenge is, therefore, to produce and support applications that can respond automatically to this variability.

Moreover, scant research has been devoted to the adaptive exploitation of the structure of a parallel application to improve the overall resource usage. Since workloads in distributed systems must be divided into tasks in order to minimize the communication costs, little attention is paid to partitioning using the application structure. We argue that the intrinsic coordination characteristics of structured parallelism place this paradigm in a preponderant position to explore this area. Based on the central premise of application adaptiveness to resource availability, we would like to research their actual correlation and provide a methodology to enable skeletons to conform to the heterogeneity of large distributed systems.

Furthermore, we consider it relevant to explore this correlation by employing the forecasting information about the structured parallelism model and exploiting its intrinsic adaptiveness through generic conventions. The main difference between this and other performance approaches is that it is intended to be focused on algorithmic skeletons, adaptable by construct, and empirically evaluated on a non-dedicated multi-cluster infrastructure.

Instead of developing a specific skeletal framework, this work presents ASPARA (*Adaptive Structured Parallelism*), a generic methodology to optimize the performance in heterogeneous distributed systems. ASPARA can forecast and enhance the performance of a skeletal application by exploiting the skeleton structure while preserving the skeleton behaviour.

ASPARA *comprises a set of rules to be embedded within skeletons*, *where every rule essentially defines an application scheduling scheme which is parameterized in terms of the existing system resources.*

As opposed to compile-time optimization, ASPARA capitalizes on the fact that the behaviour of a skeletal application is known prior to its execution. In contrast to run-time optimization, ASPARA exploits the structural characteristics of skeletons.

From this perspective, ASPARA can be categorized as a scheduling methodology for parallel programs executing in heterogeneous distributed systems, which is:

dynamic since the correct selection of resources and the adjustment of algorithmic parameters are performed at execution time;

adaptive due to the provision of intrinsic mechanisms to dynamically adjust to the system performance variations;

application-level because all the decisions are based on the particular requirements of the application at hand;

task-oriented as it assumes that the application is arbitrarily divisible into independent tasks, with or without precedence relations; and

heuristic because it comprises a set of rules intended to increase the probability of enhancing the overall parallel program performance.

This paper significantly extends our initial approach outlined in [7], describing in detail the introduction of adaptiveness into skeletal parallel programming through resource awareness driven by the program structure. This adaptiveness rests on the resource availability-performance premise, that is to say, the assumption that certain parallel programs can perform more efficiently based on a wise selection from the available system resources. We support our claims concerning performance enhancement by presenting positive empirical results based on two task-parallel skeletons: the task farm—first discussed in [8] and later extended in [9]—and the pipeline—introduced in [10] and generalized in [11]—. For convenience, we have implemented simple skeletal application programmer interfaces (APIs) in C with MPI for both skeletons, but stress that they are merely prototype vehicles to support the investigation of application scheduling schemes, which forms our main contribution.

2. MOTIVATION

As a matter of theory, the resource availability-performance premise is widely accepted as the determining factor of application adaptiveness in heterogeneous distributed systems [12–14]. According to this premise, parallel applications must be able to transform, evolve, or extend their behaviour to conform to the resources present at a certain time to improve their efficiency. The major problem in empirical research based on this premise has arisen from the automatic deployment of generic applications.

Based on the central premise of application adaptiveness to resource availability, we would like to research their actual correlation and provide a methodology to enable parallel patterns (algorithmic skeletons) to conform to the heterogeneity of a given system. Besides, we consider it relevant to explore this correlation by employing the forecasting information about a certain skeleton and fostering adaptiveness through the exploitation of its structure. Since workloads in a large heterogenous distributed system should preferably be divided into independent tasks in order to minimize communication costs, little attention is paid to partitioning using the application structure. Therefore, we shall concentrate on task parallelism and explore two distinct scenarios:

1. The workload is arbitrarily divisible into totally independent tasks (or groups of tasks), and these are scheduled to any available node in a multi-round fashion. This case is examined in Section 4.1 using the task farm skeleton.
2. The workload is decomposed into a sequence of independent computational stages, where the data is passed from one computational stage to another, and each stage, allocated to a different processing element, executes concurrently. We explore this case in Section 4.2 using the pipeline skeleton.

We have decided to deploy these two task-parallel skeletons using an imperative programming paradigm, in order to capitalize on its efficiency while providing a higher-level abstraction. It is crucial to emphasize that task parallel skeletons, and in particular the task farm and the pipeline, have been selected for our evaluation, as they represent a significant set of problems in computational science [15]. Moreover, as described by Brinch Hansen [16], they cover two out of four of the main paradigms in parallel computing:

1. **pipelining** and ring-based applications;
2. divide and conquer;
3. **master/slave**;
4. and cellular automata applications.

By means of these two case studies, this paper intends to demonstrate the applicability of the ASPARA methodology and its pragmatic approach, which incorporates scheduling rules at compilation time and, based on resource utilization, adapts at execution time.

Hence, the main objective of this work is to address the open question: how much can the structural forecasting information about structured parallelism help to improve the performance of parallel applications executing in a non-dedicated distributed heterogenous system?

3. THE ASPARA METHODOLOGY

ASPARA is a generic methodology to incorporate structural information into a parallel program at compile time that helps it to adapt at execution time. It instruments the program with a series of

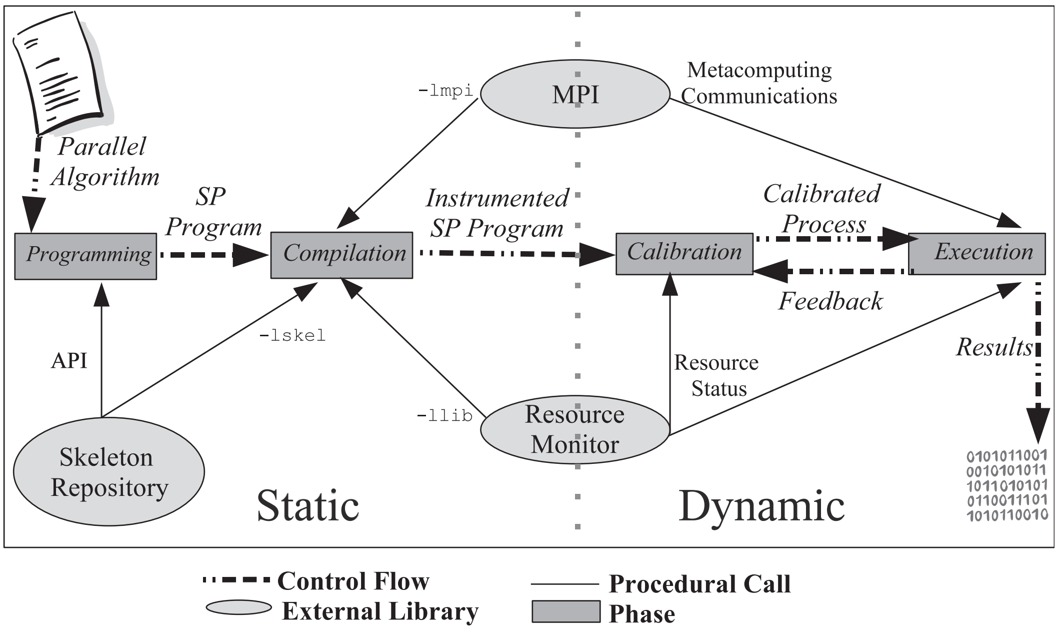

Figure 1. The ASPARA methodology. It consists of four phases: programming, compilation, calibration, and execution. The main emphasis of this work is placed on the latter two, which are considered dynamic.

pragmatic rules embedded in the algorithmic skeletons, which depend on particular performance thresholds based on the nature of the skeleton, the computation/communication ratio of the program, and the resources. As illustrated in Figure 1, ASPARA comprises four phases: programming, compilation, calibration, and execution.

3.1. Programming and compilation

Programming is a design phase in which the application programmer implements a parallel algorithm by selecting a skeleton through an API. Since structured parallelism provides a high-level approach to programming, the programmer is only required to include the initialization and termination calls for the parallel environment, and to parameterize the API calls to ASPARA. This parameterization specializes the generic meaning of the algorithmic skeleton to fit the given problem instance, and to decide on the appropriate scheduling strategy to enact adaptivity at runtime.

The compilation phase instruments the structured parallel program. The source parallel program is compiled and linked with the ASPARA library, the parallel environment, and, if statistical calibration is to be employed, a specialized resource monitoring library with historical/statistical profiling. In any case, basic monitoring capabilities, namely granularity and process execution timing, are always incorporated in this phase. Such capabilities are typically instrumented through calls to the different software infrastructure layers, such as the operating system, distributed resource monitoring environment, or independent scheduling libraries.

It is assumed that the parallel environment handles entirely the multiprocessing infrastructure, including node initialization and termination, resource co-allocation, inter-domain scheduling, and other related matters. Our current implementation does not provide for skeleton nesting.

The programming and compilation phases are both static, since they do not require any online interaction or feedback from the underlying platform. Moreover, while important in functional terms, their operation is hardly different from that of other skeletal frameworks and, therefore, will be tangentially discussed in the remainder of this work.

3.2. Calibration

The calibration is an automatic stage, which executes the skeletal program on a sample of the input data on every allocated node, and extrapolates the node performance in order to rank the fittest nodes and, consequently, select the most adequate one for the given application.

In this context, the *fitness* is defined as the quality of a node of being the most suitable computational entity to execute a task, or a series of them, on a non-dedicated basis. We then say that 'node a is fitter than node b' if, and only if, the measured execution time for node a is less than the measured execution time of node b for a given task, or a series of them.

Nodes can be selected by extrapolating their performance based on their fitness only. This is known as *times-only calibration*. This calibration schema has been employed in the case studies presented in Section 4.1—the task farm—and in Section 4.2—the pipeline. Optionally, a univariate or multivariate linear regression—correlating the fitness, the node capacity (e.g. node clock frequency or a system benchmark), the processor load, and/or the bandwidth utilization—can be applied to statistically adjust the execution time. This is known as *statistical calibration*, and has been previously employed with a single-round scheduling task farm [9].

Algorithm 1 details the calibration procedure. This procedure applies the computing function, f, over an input data subset ds in order to select the fittest nodes, $Chosen$, from the processor pool, P. Given the existing resource-usage conditions, a pre-defined calibration node collects the execution times from all processors in the pool and ranks them accordingly. As this ranking involves the actual execution of the application with a reduced data set on the complete processor pool, its temporal complexity is bound by the execution time of the slowest processor.

Measured on a non-dedicated basis in the nodes, the fitness is a transient characteristic. The selection of the fittest nodes from a pool depends not only on the hardware capabilities of each processor, but also on the system software, the existing resource usage, and the parameterized skeletal application at hand. Note that the overall execution time of this phase is determined by the execution time of the slowest node in the pool, as the correct selection depends on knowing the fitness of all the nodes.

Nonetheless, despite its positive qualities, there are two main issues associated with the calibration phase: overhead and transience. First, the calibration phase introduces a delay in the normal control flow, as it stops the execution by imposing a barrier. The performance impact of the delay has been minimized in our case studies by employing calibration functions with similar complexity for all workers/stages. Second, in terms of transience, the calibration outcome—the set of the fittest nodes—is completely valid only for as long as the resource conditions remain the same. Assuming that steady resource conditions is not only unrealistic, but also against our initial assumptions.

Algorithm 1: The ASPARA Calibration Algorithm

```
Data: $f$: Set of Functions;
$ds$: data subset;
$P$: Processor Pool;
$t$: Execution times vector;
Result: $Chosen$: Table of fittest processing elements;

Execute $f(ds)$ over $P$ nodes and collect centrally execution times;
Set $t \leftarrow$ execution times($f(ds)$);
if Statistical Calibration then
    Collect processor and bandwidth values;
    Adjust $t$ statistically;
end
Rank $P$ based on $t$;
Select $Chosen$ from $P$;
Broadcast/Share $Chosen$ to/with all nodes;
Return($Chosen$);
```

Therefore, we have experimented with calibration under two execution adjustment modes, periodic and reactive, to extend its temporal validity.

Sudden variations in the resource conditions render the calibration invalid, but increasing the frequency of calibrations leads to performance penalties. Therefore, a balance between transience and validity must be preserved to avoid thrashing and keep the set of fittest nodes effectual.

In summary, the calibration phase essentially drives the task-to-node allocation based on the intrinsic properties of the algorithmic skeleton and the resource characteristics and usage. It provides the optimized conditions for the execution of the skeletal parallel program.

3.3. Execution

As the set of fittest nodes is initially determined by the calibration phase, the execution phase is responsible for keeping this set valid. By monitoring the performance throughout the course of the program execution, this phase:

periodically reassesses the relative nodes' fitness. The set of fittest nodes is dynamically adjusted by ranking the nodes according to their relative fitness within the set. Thus, if a node increases its availability during the latest task allotment, not only is its relative fitness affected, but also that of the other nodes. This periodic reappraisal uses any subset of tasks on all processors, relying on the assumption of the iso-complexity of the tasks. That is to say, the API tracks the execution times of every subset of the workload, allowing the nodes to be correctly ranked at all times. Periodic adjustment is applied for the multi-round scheduling task farm case study presented in Section 4.1 or

reactively triggers a re-calibration to adjust the set of nodes once a threshold is reached. In this execution mode, we formally define a *threshold* as a percentage value of performance fluctuation. This threshold typically expresses the dispersion of the calibration times in the node pool. The more dispersed the times are, the lower the threshold value should be. ASPARA then makes the application adapt to the infrastructure by allowing performance variations up to this threshold. That is to say, the threshold determines how permissible a performance variation is. Once the threshold is surpassed, the API takes action, e.g. feeding back to the calibration phase and modifying the task scheduling according to the properties of the skeleton in hand. By setting the right threshold for a given platform, one can avoid thrashing due to frequent re-scheduling. Reactive adaptiveness is deployed for the pipeline case study presented in Section 4.2.

Ideally, the parametric variations of the periodic and reactive execution modes must be automatically determined according to the existing system load conditions at the start of the execution, rather than requesting any human intervention or feedback. Nonetheless, both execution modes imply an overhead.

- In the periodic mode, the dynamic adjustment is carried out by timing the execution of a set of tasks in a given node and, subsequently, calculating the relative fitness for every node. Both the timers and the arithmetic calculations require processing cycles.
- In the reactive mode, the re-scheduling of tasks requires to stop the normal execution to migrate processes with a consequent idle system time. The full extent of the performance implications of this migration are difficult to estimate since they involve four steps: stop and checkpoint the input stream, pipeline draining, re-calibration, and pipeline filling-up.

Intended as a proof of concept rather than as an industrial-grade framework, our methodology has produced performance improvements in our evaluative case studies. Nevertheless, extremely noisy applications with a large number of processing peaks and/or short-lived cyclical variations might produce idle times and/or thrashing, which would eventually render the adjustments counterproductive. Under those atypical circumstances, the full extent of the overheads in the different execution modes deserves further examination.

It is crucial to note that both stages, calibration and execution, must be dynamically determined, since their actual execution varies according to the application—defined by the skeleton parameterization and the algorithm itself—the size of the overall workload, and the resource conditions. It is precisely these two stages that represent the main differentiator of the ASPARA methodology, and, therefore, the case studies will examine in detail their instantiation in the context of the task farm and pipeline skeletons.

The overall functionality of the execution phase is illustrated by Algorithm 2.

4. EVALUATION

We present two evaluation scenarios: one for the multi-round task farm and another for the pipeline, which are discussed in Sections 4.1 and 4.2, respectively. Consequently and as a proof of concept, ASPARA has been furnished with two algorithmic skeletons, the task farm and the pipeline, programmed in ANSI C as APIs, as shown in Figure 2.

In terms of the workload, the underlying assumptions for these scenarios are:

1. the selected parallel algorithms have been pre-qualified to be divisible workloads of independent nature or stage-decomposable, that is to say, they deploy embarrassingly parallel (task farm) or pipelined computations;

Algorithm 2: The ASPara Execution Algorithm

```
Data: $f$: Set of Functions;
$Chosen$: Table of fittest nodes;

Map workload to the $Chosen$ nodes;
Set $M$ as the monitor node;
while ¬ end_of_workload do
    Execute $f$ over $Chosen$ nodes concurrently;
    if $M$ then
        Set $t \leftarrow$ execution times(f);
        Adjust $Chosen$ according to $t$/* This adjustment can be carried out either
            periodically or reactively                                    */
    else
        Send time from this node to monitor node;
    end
end
Return();
```

```
void pfarm(void(*worker)(), void *in_data, int in_length, MPI_Datatype in_type,
     void *out_data, int out_length, MPI_Datatype out_type, MPI_Comm comm,
     enum scheduling sched);
```
(a)

```
void pipeline(stage_t *stages, int no_stages, MPI_Datatype in_data[],
      MPI_Comm comm);
```
(b)

Figure 2. Application programming interfaces to: (a) the task farm and (b) the pipeline algorithmic skeletons.

2. the computational complexity of each task in the workload is identical, in the sense that all tasks would take the same time to process one item if executed on a dedicated reference processor. In effect, this is to assume that the programmer has done a sound abstract job of balancing complexity; and
3. the communication time is not significant. Note that this is not to assume that the communication is negligible, but rather to assume that the communication costs hinder all consumer–producer pairings similarly.

The first assumption assures the applicability of the ASPara methodology, and is therefore *sine qua non*. Nevertheless, it is important to assert that there is a significant number of real problems in computational science, which can be modelled as divisible workloads of independent tasks, with or without precedence relations [17,18].

The second assumption defines the complexity of the calibration phase. Undoubtedly a crucial component of the methodology provides a standardized initial execution foundation, the calibration phase is bound in its complexity by the slowest processing of all the nodes, and introduces an overhead in the total processing of the workload due to its inherent control barrier. We acknowledge that this is a significant constraint, but there are many real applications for which it holds (e.g. the computational biology code used in this paper). Such applications are considered well balanced and would not normally need dynamic farming; hence, they are not usually seen as candidates in need of anything more than a trivial $\lfloor N/P \rfloor$ distribution of the N tasks on a fully homogeneous system with P nodes.

For the sake of argument, let us consider that this assumption is relaxed, such that there are $\jmath$ different task functions with distinct complexities. For a heterogeneous system with P nodes, this will imply the calibration of the P nodes for the $\jmath$ functions which, in itself, will be a hard optimization problem. Since there exists a sequential constraint to avoid contention, and the execution of each function is bound by that of the slowest node, one would expect, in the worst case, the total time to be defined as the sum of the slowest times in P for each of the $\jmath$ functions. Furthermore, let us assume that the node availability changes during the calibration of the nodes for the ℓ function ($\ell \leq \jmath$). This will render invalid the previous $\ell - 1$ calibrations, which rely on a ranking based on the availability conditions present. As it stands, our constraint of a single function complexity for each task allows us to maintain an acceptable balance between system dynamism and usability.

Although this treatment considerably simplifies the calibration, there are a significant number of problems in computational science, e.g. parameter sweeps, which are modelled as similar-complexity tasks [19]. Nonetheless, different heuristics have been suggested to circumvent this hindrance [20], such as the definition of a generic cost function in terms of a weight-based computation-to-communication ratio; a random stage/worker-to-node allocation; and, the initial calibration of all nodes on a dedicated basis. All the same such approaches may still increase the overall complexity of the calibration process, imply a loss of generality, and/or deactivate our non-dedicated systems approach, affecting particularly the reactive adjustment mode of the execution. Nevertheless, we strongly believe that the results are worthwhile as heterogeneity and resource dynamism tend to be prevalent in distributed systems.

The third assumption reflects the fact that our performance improvement strategies have not concentrated on node-specific communication issues for simplicity purposes, but could eventually be added.

4.1. Task farm

The first evaluation scenario examines the application of the ASPara methodology for the task farm skeleton using times-only calibration and periodic adjustment at execution. The programming phase requires the API to be parameterized with the worker function, the input and output vectors, the MPI communicator and the scheduling, as shown in Figure 2(a).

In addition to the subject of this work, the self-scheduling multi-round mode, the scheduling parameter in the API allows the task farm skeleton to operate in naïve and adaptive single-round modes, as well as simple multi-round mode that distributes one task at a time to each node.

Let us examine how the task farm operates, when employing the self-scheduling multi-round mode. It quantifies the worker resources at a given time on a certain node topology from an application-specific perspective, by means of a fitness index F. Defined during the calibration phase, F is to be used to determine the task size on a per node basis and, consequently, define the scheduling. Moreover, its value is also adjusted during execution.

Let S denote the workload assigned to the farmer, expressed as the total number of tasks in the workload, and N the number of participating workers (typically $N \ll S$). Thus, our objective is to calculate $\alpha_\imath$, the task size for each worker:

$$\alpha_\imath \ \forall\ \imath \in [1, N] \quad \text{subject to} \sum_{\imath=1}^{N} \alpha_\imath = S$$

If we construct F as the sum of the relative fitness $F_\imath$ of each node as $\sum_{\imath=1}^{N} F_\imath = 1$, we can determine $\alpha_\imath$ as:

$$\alpha_\imath = S \times F_\imath \ \forall \imath \in [1, N] \tag{1}$$

Note that $\alpha_\imath$ is the total number of elements assigned to node $\imath$ and the key differentiator for the scheduling lies in how this amount is distributed, as $\alpha_\imath$ can be distributed to node $\imath$ in one or more *installments*. If distributed in one installment, then the scheduling will be considered single-round, otherwise it will be multi-round. That is to say, the number of installments clearly defines the scheduling and we can therefore extend Equation (1) to consider an *installment factor* k to determine such a distribution:

$$\alpha_\imath = \frac{S}{k} \times F_\imath \quad \forall \imath \in [1, N] \ \wedge \ k \geq 1 \tag{2}$$

The values k and F are determined using Equations 3 and 4. In particular, they are calculated using the coefficient of variability (CV) of the nodes' execution times at calibration:

$$F_\imath = \frac{1/t_\imath}{\sum_{\jmath=1}^{N} 1/t_\jmath} \tag{3}$$

$$\text{Given that } \bar{t} = \frac{1}{N}\sum_{\imath=1}^{N} t_i, \quad \sigma = \sqrt{\frac{1}{N}\sum_{\imath=1}^{N}(t_i - \bar{t})^2} \quad \text{and} \quad CV = \frac{\sigma}{\bar{t}}$$

$$k = \ln(S)^{CV} \tag{4}$$

In order to justify the cogency of k as an installment factor, let us examine its behaviour schematically. Figure 3 plots k in terms of the workload S and the coefficient of variability CV. As k grows logarithmically with respect to S, the overall number of installments reflects the size of the workload while minimizing the communication exchanges. Furthermore, since the CV reflects the difference in the activity of the participating nodes, noisy systems will yield larger values of k in order to satisfy their inherent changes but, by construction, will not be greater than 1. Thence, k reflects both the size of the workload and the activity of the nodes while lessening the overall number of installments.

Moreover, the initial calculation of F abstracts *ab initio* the resource availability in a given system, but its temporal validity is not necessarily assured as the load conditions frequently vary over time. Thus, during the execution, we propose to adapt F periodically according to the latest

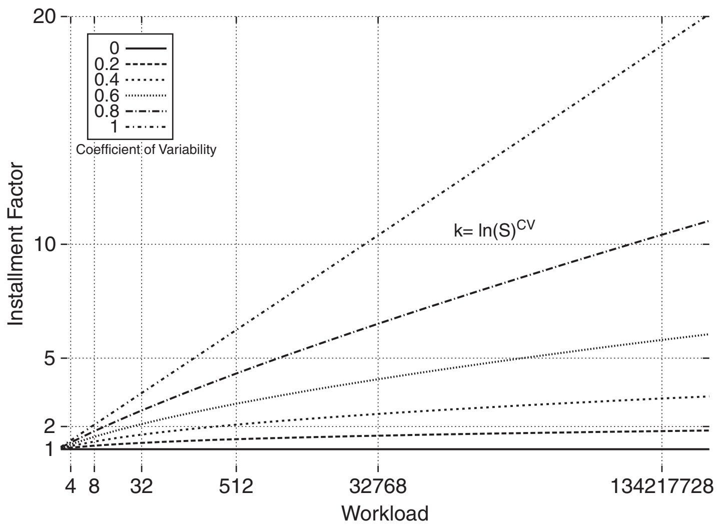

Figure 3. The installment factor, k is a function of the workload, S, and the coefficient of variability, CV, defined as $k = \ln(S)^{CV}$. The six different lines of k are defined by the variation of CV from 0 to 1 in intervals of size 0.2.

performance reading for a node, i.e. we always re-calculate the F_ι every time a node reports the execution time t_ι, and therefore its subsequent α_ι is always up-to-date. Note that this re-calculation also affects the relative fitness of all the other nodes.

Hence, our self-scheduling task farm can be conceived as a generic heuristic for multi-round scheduling of divisible workloads in heterogeneous systems, as the number of rounds for the scheduling is dynamically defined based on the general node activity.

4.2. Pipeline

The second evaluation scenario explores the application of ASPARA to the pipeline skeleton. In this case, the programming phase parameterizes the API with the stage function, the input vector, and the MPI communicator, as illustrated in Figure 2(b).

Internally, the implementation of the API works as follows. Assuming an N-stage pipeline, at calibration, the API allocates stages to processors, ranking the processors by descending fitness, and selecting the fittest N. Then, it iteratively tries to improve its stage allocation by moving stages from the slowest nodes to the fastest ones in a greedy fashion.

In addition to the mapping, this phase performs the calculation of the performance threshold, key to the adaptivity mechanism of the pipeline. We propose a performance threshold for our adaptive pipeline as the inverse standard deviation of the calibration times of all nodes in the processor pool. Assuming a normal distribution of the calibration times, their standard deviation, σ, can be calculated using Equation (5), where $|P|$ and $\bar{t}$ are the number of nodes in the processor pool and the average calibration time, respectively. As σ quantifies the dispersion of the calibration times, its inverse, presented in Equation (6), can arguably be used as a measure of adaptivity:

$$\sigma = \sqrt{\frac{1}{|P|}\sum_{\imath=1}^{|P|}(t_i - \bar{t})^2} \tag{5}$$

$$threshold = \frac{1}{\sigma} \tag{6}$$

Then, the execution monitors the correct functioning based on an intrinsically defined performance threshold, and reactively triggers a calibration if the threshold is surpassed.

That is to say, our self-scheduling pipeline implements a generic heuristic for stage mapping in heterogeneous systems [21]. Based on the node dispersion, our pipeline allows the automatic mapping and consistent monitoring of stages, without requiring pre-existent application knowledge or benchmarks.

5. RESULTS

The performance figures reported in this section have been obtained employing two interconnected, non-dedicated Beowulf clusters, co-located across the University of Edinburgh, and configured as shown in Table I.

The complete configuration includes a non-dedicated network, a storage sub-system, and individual cluster management software, enabling the interconnection and storage virtualization while maintaining both clusters as two separate entities. The versions of the software are Linux Red Hat FC5 (kernel 2.6), gcc Compiler 4.1.1, LAM/MPI 7.1.2, and the GNU Scientific Library (GSL) 1.7.

This multi-cluster environment is considered as the representative of the dynamism of large heterogenous systems, as it spans different administrative domains, comprises heterogeneous nodes and links, and does not have dedicated resource co-allocation or reservation.

All variabilities in the system are due to the external load and, to a lesser extent, due to the difference in the performance among processors. All execution times reported in this work sum up the average of several measurements, with a standard deviation of less than 1%.

Table I. The hardware configuration of our experimental multi-cluster environment.

	`bw240`	`bw530`
Hardware	64 nodes	16 nodes
CPU	Uni-core Intel P4	Uni-core Intel Xeon
Memory	1 GB node^{-1}	2 GB node^{-1}
Network	2×100 Mb s^{-1}	1×100 Mb s^{-1}, 1×1 Gb s^{-1} Myrinet
BogoMips	3350–3555	3326–3359

5.1. Task farm

Our experiments have been designed to take advantage of the task farm intrinsic task parallelism—which presents virtually no inter-process communication—and the ability to access different data sources—inherent to any distributed system. As a result, they deploy a parameter sweep for a series of independent executions of a stochastic simulation algorithm of voltage-gated calcium channels on the membrane of a spherical cell. Parameterized in terms of the number of channels and time resolution, the algorithm calculates the calcium current and generates a calcium concentration graph per run.

A spherical cell possesses thousands of voltage-gated channels, and simulating their stochastic behaviour implies the processing of a large number of random elements with different parametric conditions. Such parameters describe the associated currents, the calcium concentrations, the base and peak depolarizing voltages, and the time resolution of the experiment. This process can be modelled stochastically, defining a threshold based on voltage and time constraints, and aggregating individual calcium currents for a given channel population [22].

Furthermore, as the voltages and the peak duration can be varied without affecting the complexity, the parameter space can be explored while preserving the complexity constant at each run. The model has been abstracted as the function f where the number of channels ($channels$) and time resolution, defined as the number of steps ($steps$), determines its temporal complexity on a per-experiment basis as follows:

$$Time(model) = O(channels \times steps) \tag{7}$$

A typical experiment involving the simulation of 10^4 channels for a second in 10 μs intervals (10^5 steps) will have $O(10^9)$ temporal complexity. Each experiment generates two result files: a data file that records the calcium currents values over time and a gnuplot script to automatically produce graphs for these values. The physiological interpretation of the algorithm is beyond the scope of this work, nonetheless it is interesting to underscore its relevance to the biomedical community. A complete description of the simulation algorithm, a comprehensive parameter sweep, and the physiological interpretation of the results are reported in [23].

Here we have run a series of experiments incorporating three different settings: light, medium, and heavy.

At the single experiment level, while maintaining the number of channels, the time interval, and the base voltage constant at 10 000, 10 μs, and −80 mV, respectively, we have varied the simulation time for each experiment using 0.01 s, 0.1 s, and 1 s, i.e. a time resolution of 10^3, 10^4, and 10^5 steps, respectively. Note that the complexity of the experiments in the medium case is similar to that of those employed in the preceding section for the single-round scheduling.

At the parameter space level, the parameter sweep looks upon peak voltages in [−60, 60 mV]. Employing steps of 0.0125 mV, 0.03125 mV, and 0.125 mV, this range is evenly divided, producing an associated number of experiments of $S = 9600$, 3840, and 960, respectively. Note that the variation in the value of S has no bearing on the complexity of the experiments as described in Equation (7). Table II shows the three instances of the parameter space.

We have assembled nine different scenarios by varying the number of workers to 8, 16, and 32 executing the light, medium, and heavy problem instances, and benchmarked our adaptive scheduling against the self-scheduled work queue [24]. In the work queue scheduling, the farmer

Table II. Parameter space for the multi-round scheduling task farm.

Parameter	Light	Medium	Heavy
Experiment			
Number of channels	10^4	10^4	10^4
Time resolution	10^3	10^4	10^5
Peak duration (s)	0.006	0.06	0.6
Sweep			
Peak voltage steps (mV)	0.0125	0.03125	0.125
Number of experiments	9600	3840	960
8-worker time (s)	868.4	3199.6	7720.8
16-worker time (s)	470.4	1730.2	4205.1
32-worker time (s)	235.3	869.2	2125.1

The final time rows show the average execution time, in seconds, for whole parameter sweep on 8, 16, and 32 workers using adaptive multi-round scheduling.

supplies one task to any available worker at a given time—therefore, we have used the one-by-one designation—and it is widely accepted that such a strategy provides an acceptable load-balancing strategy for large workloads of undetermined size. The greedy nature of self-scheduling allows the assign-to-idle-node scheme to balance the system load over time.

While the results have demonstrated a modest performance improvement of 3% for the light case and a slightly negative decrement for the medium −0.1% and heavy −0.3% cases, the automatic calculation of the installment factor and the periodic refinement of the task size should be considered as important contributions for self-scheduling parallelism. A summary of the results is presented in Figure 4.

5.2. Pipeline

We have here employed as the pipeline stage function the synthetic `whetstones` procedure from the 1997 version of the Whetstone benchmark [25] with parameters (256, 100, 0). It accounts for some 5 s of double-precision floating-point processing in an unloaded node of our experimental computational environment.

We have assembled an empirical evaluation using three node pools, $n = \{8, 16, 32\}$, handling five input data sizes $S = \{32, 64, 128, 256, 512\}$. The 15 scenarios were staged on three different days, based on the number of nodes, with each scenario yielding to the execution of five experiments. To make the execution of each of the 25 experiments independent in a day, we have interleaved their executions according to the input data size, ensuring that only one pipeline experiment was executed at a time.

Each entry in Table III presents the average of three experiments along with its coefficient of variation (CV), the average threshold for the series, and the naïve time estimate, i.e. the processing time assuming no re-scheduling and a direct mapping in which one stage is assigned to each of the n fittest nodes. Our adaptive method employs a greedy algorithm to optimize the stage-to-processor mapping and allows re-scheduling according to the defined threshold. A complete discussion on the greedy algorithm and the mapping itself is presented in [11].

The obtained figures attest to the performance improvements attributed to the use of our pipeline.

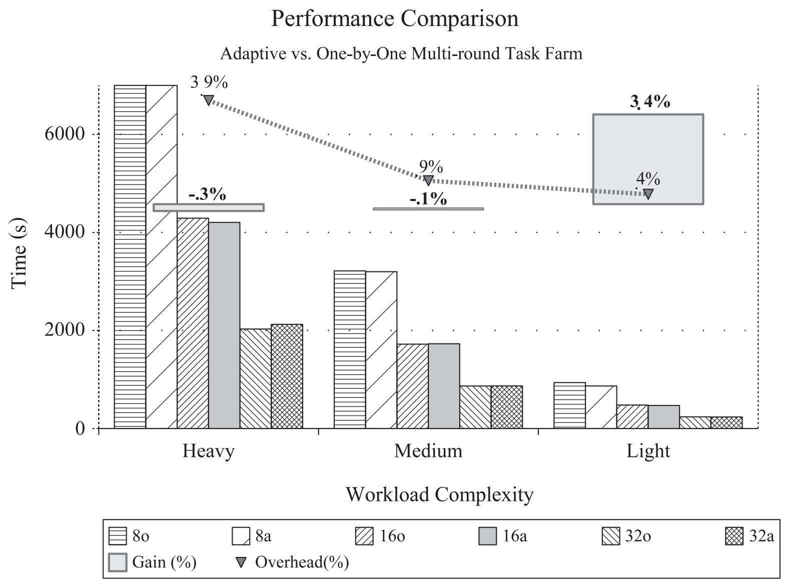

Figure 4. Execution time summary for the task farm using adaptive multi-round scheduling on 8, 16, and 32 workers with light, medium, and heavy experimental workload settings, as described in Table II. The hatched bars represent the execution times for each combination of worker-scheduling-setting. The top dashed line represents the overhead incurred by the initial calibration, whereas the thick-line rectangles plot the average gain from using the adaptive scheduling for all executions in a setting. Key: [processors no.][scheduling], e.g. 8a means 8 workers and adaptive scheduling model and, analogously, 8o represents 8 workers and one-by-one (work queue) scheduling.

Table III. Listing of the execution times of the adaptive pipeline using different pipeline ($n = \{8, 16, 32\}$ and input $|S| = \{32, 64, 128, 256, 512\}$) sizes.

n	S	**32**	**64**	**128**	**256**	**512**
8	*Adaptive* (s)	347.69	672.69	1289.56	2501.27	5193.20
		$CV = 4.82\%$	$CV = 5.51\%$	$CV = 4.62\%$	$CV = 3.18\%$	$CV = 2.75\%$
	Threshold	0.47	0.38	0.33	0.36	0.36
	Naïve (s)	344.43	778.58	1665.33	2994.19	5891.82
16	*Time* (s)	399.09	714.18	1384.75	2617.53	5311.50
		$CV = 5.03\%$	$CV = 5.89\%$	$CV = 0.56\%$	$CV = 0.87\%$	$CV = 1.33\%$
	Threshold	0.45	0.50	0.44	0.42	0.40
	Naïve (s)	436.10	711.38	1532.29	2734.57	6304.72
32	*Time* (s)	495.84	821.76	1467.12	2774.43	5560.20
		$CV = 2.34\%$	$CV = 0.57\%$	$CV = 0.60\%$	$CV = 7.57\%$	$CV = 2.79\%$
	Threshold	0.53	0.55	0.56	0.47	0.51
	Naïve (s)	493.02	774.13	1381.95	3323.52	6533.42

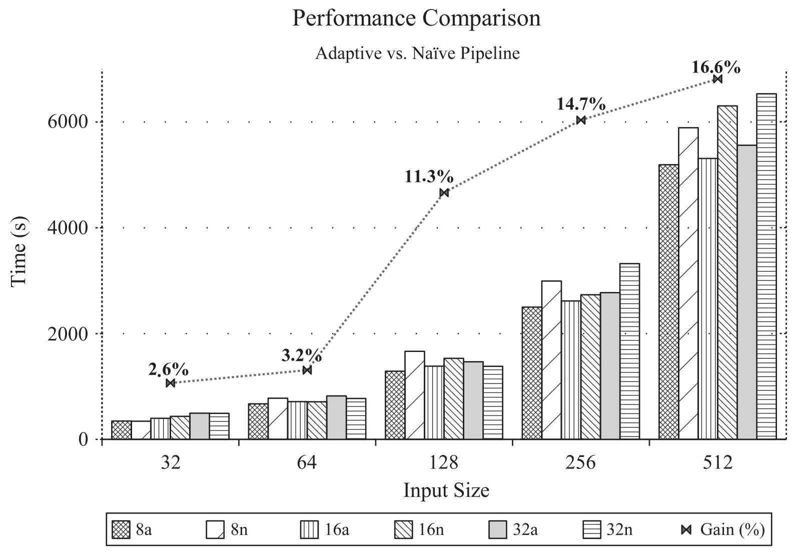

Figure 5. Execution time summary and overall improvement due to the use of our adaptive pipeline. The x-axis indicates the number of elements processed by the pipeline (input size). Each hatched bar plots the execution time for a given pipeline size and scheduling method (adaptive or naïve). The top dashed line represents the aggregated gain as a percentage. Key: [processors no.][scheduling], e.g. 8a means 8-stage pipeline and adaptive scheduling model and, analogously, 8n represents 8-stage naïve scheduling.

Figure 5 presents a summary of the overall performance gain due to its use. Our empirical evaluation demonstrates a correlation between the size of the data input, S, and performance gains, as load variations impact more as the processing increases.

In the course of the 3-day empirical evaluation, our multi-cluster configuration entertained different workloads from a multiplicity of users. Such workloads provided a realistic environment in which to assert the adaptivity of our pipeline.

6. DISCUSSION

In terms of the performance, our results have shown modest improvements, nonetheless encouraging, as they have allowed to support the claim that the skeletal structure information can be used to improve the performance in non-dedicated open grid environments. These improvements varied from −0.3 to 3.4% for the task farm, and from 2.6 to 16.6% for the pipeline.

Furthermore, while this evaluation has assumed similar complexity for all tasks in the workload, this is not *sine qua non*. Although it is arguable that generic workloads are composed of tasks with different complexities, we wanted to isolate the system from function variations and just deal in this work with variations due to load conditions. Different heuristics have been suggested to circumvent this inconvenience [20], such as a random allocation, the definition of a weight-based computation-to-communication ratio, or the initial calibration of all nodes on a dedicated basis, but such approaches may deactivate our non-dedicated systems approach, increase the overall complexity of the calibration process which is assumed to be lightweight, and/or imply a loss of generality, also affecting the reactive execution. Previous work on skeletal-based load balancing has indicated that different scheduling strategies for mapping tasks on demand may yield to performance improvements [26]. Our initial experiments with a single-round task farm using weight-based determination have also confirmed such implications [9]. As it stands, our constraint of a single function complexity for each task allows us to maintain an acceptable balance between system dynamism and usability.

7. RELATED WORK

Traditional strategies for task scheduling in distributed systems [17,27–30] rely on system simulators, dedicated configurations, and/or performance estimators to model the general system, particularly to characterize the background load in terms of its job arrival rate. While much can be said about the reproducibility of their results, one may argue that they artificially create tractable evaluation scenarios for their scheduling policies. Accordingly, the ASPARA methodology cannot be simplistically compared with any task scheduling policy in terms of algorithmic optimality and complexity, but ought to be evaluated in terms of the makespan for a certain workload. To this end, our case studies report execution times, as well as the relative performance improvement with respect to the baseline for a given workload.

On the other hand, platform-oriented approaches have deployed software extensions to the existing frameworks in order to enact application-level scheduling.

1. AppLeS [31] includes a parameter sweep template (which can be thought of as an instance of task farming) which implements four different heuristic scheduling policies based on an initial estimate of a task computational cost [32]. Their main disadvantage is the need for instrumenting the application with AppLeS primitives.
2. Nimrod/G [33] incorporates not only the time but also the cost of the resources as the basis for its parameter sweep scheduling algorithm [34]. It relies on historical data to determine a task computational cost, but receives no guidance from the application structure.
3. Condor [35] has been extended with a master–worker runtime library (again, in effect a task farm) primarily intended to optimize the number of workers [36]. Based on a usage threshold, it optimizes the number of workers by keeping track of the resource consumption over time and returns idle workers to the processor pool. Nonetheless, Condor does not possess any information about the application itself, so all load balancing is carried out using generic on-demand farming guidelines, as Condor itself does not allocate resources based on the application-specific tasks at hand, as it is the case with the ASPARA calibration.
4. Workflow systems: Concerned with the automation of scientific processes based on a multiplicity of different data, control, and planning dependencies [37], a scientific workflow is structured using sequential, parallel, choice, and iterative tasks [38]. It is representable through directed acyclic graphs (DAG) and non-DAGs, which include not only communication, computation, and interaction characteristics, but also application planning requirements and domain knowledge. On the other hand, algorithmic skeletons, and undoubtedly the

ASPARA methodology, concentrate on well-defined patterns of computation, communication, and interaction, and, consequently, the structure can be used to guide scheduling decisions. Therefore, workflow systems address a different, more generic problem, where the overall application presents more complex control, data, and planning dependencies—which cannot necessarily be representable as a skeletal construct—and, ultimately, scheduling decisions are driven by performance and structural considerations as well as market/economical and security/trust considerations.

While the four cited approaches provide more mature software frameworks with dozens of applications deployed, they typically require:

- the application to be previously instrumented and modified to execute under the specific framework;
- the user to supply application performance estimations or benchmarks; and/or
- the framework to support complex control, data, and planning dependencies.

That is to say, they do not completely decouple the behaviour from the structure of the application, disallowing the clear advantages of ASPARA, which are its application agnosticism through the use of structural information and its heuristic resource model derived from the system activity.

Finally, recent work on adaptive high-level parallel systems has only reinforced the importance of platform adaptation for the automatic optimization of parallel codes in heterogeneous distributed systems. Cunha *et al.* [39] cite a series of component-based problem-solving environments that 'allow a clear separation between computation and interaction.' While the list is far from comprehensive, it provides clear guidance on the need for enhanced high-level parallel programming tools, and several skeletal libraries now furnish support for heterogeneous distributed systems.

ASSIST provides load-balancing mechanisms through an application manager [40]. This manager uses configuration-safe points within the program to enable load balancing when a bottleneck is encountered. While the reported results on reconfiguration overheads are interesting, scant reference is made to the allocation policies employed, either at the start of the application or at the reconfiguration. Additionally, under the umbrella of the European projects of CoreGRID and GridCOMP, ASSIST has reported attractive results on automatic performance management of applications [41] and the use of behavioural skeletons [42] has been suggested to address the autonomic management of resources.

eSkel has employed reactive process algebra methods to predict the performance and incorporate the reactive scheduling into a pipeline skeleton [43].

Lithium has been extended to provide future-based Java RMI optimization mechanisms to enhance load balancing through a statically defined thread interval [44]. This interval, typically of two–six threads, preemptively controls the node load. Although the reported results provide favourable guidance, it is unclear how the interval is defined.

Mallba deploys the Netstream middleware to instrument the skeleton structure in the library, providing high-level network communication services [45]. Resource selection is based on the readings of network links and node load, but according to the authors, they are 'still at the stage of developing intelligent algorithms to use this [network] information to perform a more efficient search [of resources].'

Hence, in contrast to our approach, these four skeletal approaches apply *ab initio* resource-node matching strategies, based on theoretical performance cost modelling (eSkel and Lithium), current resource usage only (Mallba), or reactive modification of the resource allocation (ASSIST).

It is also important to stress the fact that ASPARA uses online information, permitting the calibration to automatically feed the node status to the execution, which in turn uses the information as the starting point for its operation, allowing a more accurate feedback process.

8. CONCLUSIONS

In this work, we have investigated the feasibility of using the information provided by the structure of an algorithmic skeleton to enhance the performance of the corresponding parallel program during execution. Being agnostic to the skeleton behaviour, our 4-step methodology, ASPARA, has deployed a pragmatic approach in order to instrument the parallel application at compilation and allows the resulting parallel program to adapt at execution.

Having been conceived to function under the dynamic conditions of a non-dedicated heterogeneous distributed system, we have evaluated ASPARA under non-controlled circumstances in an open computational grid. This paper has employed two crucial task-parallel skeletons, the task farm and the pipeline, to illustrate the feasibility of ASPARA.

Our main findings can be summarized as:

1. the introduction of resource-awareness to a skeleton, using its structural information and, without altering its behaviour, has improved the performance of the resulting parallel program; and
2. the generation of autonomic scheduling strategies for parallel patterns—without the need for application foreknowledge, e.g. benchmarks or execution traces—is not only feasible, but also efficient.

The former effectively deals with our initial open question, and the latter implies that the calibration–execution phase pairing has, in essence, become a scheduling scheme for parallel patterns, which is, both, application-agnostic and autonomic. While the application independence is inherent to the skeletal paradigm, the autonomy has been achieved through sampling and statistical functions, which reflects the state of the nodes and the actual requirements of the application at hand.

REFERENCES

1. Cole M. *Algorithmic Skeletons*: *Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman/MIT Press: London, 1989.
2. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA. The landscape of parallel computing research: A view from Berkeley. *Technical Report UCB/EECS-2006-183*, EECS Department, University of California, Berkeley, December 2006. Available online at: http://view.eecs.berkeley.edu/ [Version: 17/Nov/08].
3. Buyya R. Process scheduling, load sharing, and balancing. *High Performance Cluster Computing*, *Volume 1*: *Architectures and Systems*, ch. 20–25. Prentice-Hall: Upper Saddle River, 1999; 499–624.

4. Norman MG, Thanisch P. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys* 1993; **25**(3):263–302. DOI: 10.1145/158439.158908.
5. Laforenza D. Grid programming: Some indications where we are headed. *Parallel Computing* 2002; **28**(12):1733–1752. DOI: 10.1016/S0167-8191(02)00186-2.
6. Skillicorn DB, Talia D. Models sand languages for parallel computation. *ACM Computing Surveys* 1998; **30**(2):123–169. DOI: 10.1145/280277.280278.
7. González-Vélez H, Cole M. Adaptive structured parallelism for computational grids. *PPoPP'07*. ACM: San Jose, 2007; 140–141. DOI: 10.1145/1229428.1229456.
8. González-Vélez H. An adaptive skeletal task farm for grids. *Euro-Par 2005* (*Lecture Notes in Computer Science*, vol. 3648). Springer: Lisbon, 2005; 401–410. DOI: 10.1007/11549468_47.
9. González-Vélez H. Self-adaptive skeletal task farm for computational grids. *Parallel Computing* 2006; **32**(7–8):479–490. DOI: 10.1016/j.parco.2006.07.002.
10. González-Vélez H, Cole M. Towards fully adaptive pipeline parallelism for heterogeneous distributed environments. *ISPA'06* (*Lecture Notes in Computer Science*, vol. 4330). Springer: Sorrento, 2006; 916–926. DOI: 10.1007/11946441_82.
11. González-Vélez H, Cole M. An adaptive parallel pipeline pattern for grids. *IPDPS'08*. IEEE: Miami, 2008; 1–11. DOI: 10.1109/IPDPS.2008.4536264.
12. Kennedy K, Mazina M, Mellor-Crummey J, Cooper K, Torczon L, Berman F, Chien A, Dail H, Sievert O, Angulo D, Foster I, Gannon D, Johnsson L, Kesselman C, Aydt R, Reed D, Dongarra J, Vadhiyar S, Wolski R. Toward a framework for preparing and executing adaptive grid programs. *IPDPS'02*. IEEE: Fort Lauderdale, 2002; 171–175. DOI: 10.1109/IPDPS.2002.1016570.
13. Schopf JM. Ten action when grid scheduling. *Grid Resource Management*: *State of the Art and Future Trends* (*Operations Research and Management Science*), ch. 2, Nabrzyski J, Schopf JM, Weglarz J (eds.). Kluwer Academic: Boston, 2003; 15–23.
14. Vadhiyar SS, Dongarra J. Self adaptivity in grid computing. *Concurrency and Computation*: *Practice and Experience* 2005; **17**(2–4):235–257. DOI: 10.1002/cpe.927.
15. Dongarra J, Gannon D, Fox G, Kennedy K. The impact of multicore on computational science software. *CTWatch Quarterly* 2007; **3**(1):3–10.
16. Brinch Hansen P. Model programs for computational science: A programming methodology for multicomputers. *Concurrency and Computation*: *Practice and Experience* 1993; **5**(5):407–423. DOI: 10.1002/cpe.4330050503.
17. Bharadwaj V, Ghose D, Mani V, Robertazzi TG. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE: Los Alamitos, 1996.
18. González-Vélez H, Cole M. Adaptive statistical scheduling of divisible workloads in heterogeneous systems. *Journal of Scheduling* 2009; DOI: 10.1007/s10951-009-0138-4.
19. Wilkinson B, Allen M. Embarrassingly parallel computations and 5-pipelined computations. *Parallel Programming*: *Techniques and Applications Using Networked Workstations and Parallel Computers*, ch. 3. Prentice-Hall: Upper Saddle River, 1999; 82–106 and 139–161.
20. Li M, Baker M. Grid scheduling and resource management. *The Grid Core Technologies*, ch. 6. Wiley: Chichester, 2005; 243–300.
21. Benoit A, Robert Y. Mapping pipeline skeletons onto heterogeneous platforms. *Journal of Parallel and Distributed Computing* 2008; **68**(6):790–808. DOI: 10.1016/j.jpdc.2007.11.004.
22. González-Vélez V, González-Vélez H. A grid-based stochastic simulation of unitary and membrane Ca^{2+} currents in spherical cells. *CBMS 2005*. IEEE: Dublin, 2005; 171–176. DOI: 10.1109/CBMS.2005.10.
23. González-Vélez V, González-Vélez H. Parallel stochastic simulation of macroscopic calcium currents. *Journal of Bioinformatics and Computational Biology* 2007; **5**(3):755–772. DOI: 10.1142/S0219720007002679.
24. Hagerup T. Allocating independent tasks to parallel processors: An experimental study. *Journal of Parallel and Distributed Computing* 1997; **47**(2):185–197. DOI: 10.1006/jpdc.1997.1411.
25. Curnow HJ, Wichmann BA. A synthetic benchmark. *Computer Journal* 1976; **19**(1):43–49. DOI: 10.1093/comjnl/19.1.43.
26. Danelutto M. Irregularity handling via structured parallel programming. *International Journal of Computational Science and Engineering* 2005; **1**(2–4):73–85. DOI: 10.1504/IJCSE.2005.009693.
27. Casavant T, Kuhl J. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering* 1988; **14**(2):141–154. DOI: 10.1109/32.4634.
28. Drozdowski M, Lawenda M. Multi-installment divisible load processing in heterogeneous distributed systems. *Concurrency and Computation*: *Practice and Experience* 2007; **19**(17):2237–2253. DOI: 10.1002/cpe.1180.
29. El-Rewini H, Lewis TG, Ali HH. *Task Scheduling in Parallel and Distributed Systems* (*Innovative Technology Series*). Prentice-Hall: NJ, 1994.
30. Majumdar S, Eager DL, Bunt RB. Scheduling in multiprogrammed parallel systems. *SIGMETRICS Performance Evaluation Review* 1988; **16**(1):104–113. DOI: 10.1145/1007771.55608.
31. Berman F, Wolski R, Casanova H, Cirne W, Dail H, Faerman M, Figueira S, Hayes J, Obertelli G, Schopf J, Shao G, Smallen S, Spring N, Su A, Zagorodnov D. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems* 2003; **14**(4):369–382. DOI: 10.1109/TPDS.2003.1195409.

32. Casanova H, Obertelli G, Berman F, Wolski R. The AppLeS parameter sweep template: user-level middleware for the grid. *SC'00*. IEEE: Dallas, 2000; 60. DOI: 10.1109/SC.2000.10061.
33. Abramson D, Buyya R, Giddy J. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems* 2002; **18**(8):1061–1074. DOI: 10.1016/S0167-739X(02)00085-7.
34. Buyya R, Murshed M, Abramson D, Venugopal S. Scheduling parameter sweep applications on global grids: A deadline and budget constrained cost–time optimization algorithm. *Software*: *Practice and Experience* 2005; **35**(5):491–512. DOI: 10.1002/spe.646.
35. Thain D, Tannenbaum T, Livny M. Distributed computing in practice: The Condor experience. *Concurrency and Computation*: *Practice and Experience* 2005; **17**(2–4):323–356. DOI: 10.1002/cpe.v17:2/4.
36. Heymann E, Senar MA, Luque E, Livny M. Adaptive scheduling for master–worker applications on the computational grid. *Grid 2000* (*Lecture Notes in Computer Science*, vol. 1971). Springer: Bangalore, 2000; 214–227. DOI: 10.1007/3-540-44444-0_20.
37. Yu J, Buyya R. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Record* 2005; **34**(3):44–49. DOI: 10.1145/1084805.1084814.
38. Van Der Aalst WMP, Ter Hofstede AHM, Kiepuszewski B, Barros AP. Workflow patterns. *Distributed and Parallel Databases* 2003; **14**(1):5–51. DOI: 10.1023/A:1022883727209.
39. Cunha JC, Rana OF, Medeiros PD. Future trends in distributed applications and problem-solving environments. *Future Generation Computer Systems* 2005; **21**(6):843–855. DOI: 10.1016/j.future.2003.12.015.
40. Aldinucci M, Petrocelli A, Pistoletti E, Torquati M, Vanneschi M, Veraldi L, Zoccolo C. Dynamic reconfiguration of grid-aware applications in ASSIST. *Euro-Par 2005* (*Lecture Notes in Computer Science*, vol. 3648). Springer: Lisbon, 2005; 771–781. DOI: 10.1007/11549468_84.
41. Aldinucci M, Bertolli C, Campa S, Coppola M, Vanneschi M, Veraldi L, Zoccolo C. Self-configuring and self-optimizing grid components in the GCM model and their ASSIST implementation. *HPC-GECO*. IEEE: Paris, 2006; 45–52.
42. Aldinucci M, Campa S, Danelutto M, Vanneschi M, Kilpatrick P, Dazzi P, Laforenza D, Tonellotto N. Behavioural skeletons in GCM: Autonomic management of grid components. *PDP'2008*. IEEE: Toulouse, 2008; 54–63. DOI: 10.1109/PDP.2008.46.
43. Yaikhom G, Cole M, Gilmore S. Combining measurement and stochastic modelling to enhance scheduling decisions for a parallel mean value analysis algorithm. *ICCS'06* (*Lecture Notes in Computer Science*, vol. 3992). Springer: Reading, 2006; 929–936. DOI: 10.1007/11758525_123.
44. Aldinucci M, Danelutto M, Dünnweber J, Gorlatch S. Optimization techniques for skeletons on grids. *Grid Computing and New Frontiers of High Performance Computing* (*Advances in Parallel Computing*, vol. 14), Grandinetti L (ed.). Elsevier: Amsterdam, 2005; 255–273.
45. Alba E, Almeida F, Blesa M, Cotta C, Díaz M, Dorta I, Gabarró J, León C, Luque G, Petit J, Rodriguez C, Rojas A, Xhafa F. Efficient parallel LAN/WAN algorithms for optimization. The MALLBA project. *Parallel Computing* 2006; **32**(5–6):415–440. DOI: 10.1016/j.parco.2006.06.007.