Aspect-based patterns for grid programming

Luis Daniel Benavides Navarro, Rémi Douence, Fabien Hermenier, Jean-Marc Menaud, Mario Südholt
OBASCO group, EMN - INRIA, LINA, Département Informatique,
École des Mines de Nantes. 44307 Nantes cedex 3, France
{lbenavid,douence,fhermeni,menaud,sudholt}@emn.fr

Abstract

The development of grid algorithms is frequently hampered by limited means to describe topologies and lack of support for the invasive composition of legacy components in order to pass data between them. In this paper we present a solution to overcome these limitations using the notion of invasive patterns for the construction of distributed algorithms, a recent extension of well-known computation and communication patterns. Concretely, we present two contributions. First, based on a study of how patterns are instantiated in NAS Grid, a well-known benchmark used for evaluating performance of computational grids, we show how invasive patterns can be used for the declarative definition of large-scale grid topologies and checkpointing algorithms. Second, we qualitatively and quantitatively evaluate how our approach can be used to implement the checkpointing on top of grid applications.

1 Introduction

In recent years, grids have become a powerful system architecture that allows to execute large-scale applications as diverse as scientific applications or large-scale information systems. This kind of architecture that is composed of multiple local federations provides a highly heterogeneous environment to users [9]. To overcome this heterogeneity issue, grid architectures and applications are typically built using special purpose middleware that allows to bridge between existing, often component-based, infrastructures.

Currently, the development of grid applications using such middleware is frequently hampered by two issues: limited means to describe topologies and lack of support for the invasive composition of legacy components. For instance, grid topologies that underlie grid applications are mostly defined only implicitly through message passing as part of a grid application or using low-level means for topology definition, such as graph constructors whose links to the grid application have to be defined once and for all. As to the composition of legacy grid components, it often requires significant rewriting of the involved legacy components, for example, because the composition requires data to be passed that is not exported by the legacy components.

To overcome these problems in this paper we evaluate the applicability of invasive patterns [4]. Invasive patterns are a recent extension of well-known computation and communication patterns for the construction of distributed algorithms. This approach relies on Aspect Oriented techniques to deal with crosscutting (*i.e.*, non modular) accesses to execution states that are needed for the composition of distributed entities. Concretely, we present two contributions: (i) motivate the need for modularization techniques for crosscutting accesses within legacy grid applications: concretely, we show how invasive patterns and their aspect-oriented features for explicitly distributed programming can be used to modularize crosscutting accesses in the context of the NAS Grid Benchmark (NGB) [10], and thus provide effective support for the pattern-based implementation of grid algorithms over large topologies; (ii) we evaluate the approach qualitatively and quantitatively for a non-trivial extension of NGB that extends it by error recovery in form of a checkpoint algorithm.

The paper is structured as follows. First, we present common patterns found in legacy grid applications, in particular NGB (Sec. 2). In Section 3, we introduce the pattern language on which our solution is based and present its implementation using the AWED system for Aspect-Oriented Programming with explicit distributed features. Sec. 4 presents the evaluation and benchmarks of our approach. Related work is discussed in Sec. 5. Finally, Sec. 6 gives a conclusion and discusses future work.

2 Motivation

To motivate our approach we have analyzed the NAS Grid benchmark (NGB) framework [10] for grid infrastructures. The NGB framework is a benchmark suite for computational grids that addresses one of the most important features of grid computing, the ability to execute dis-

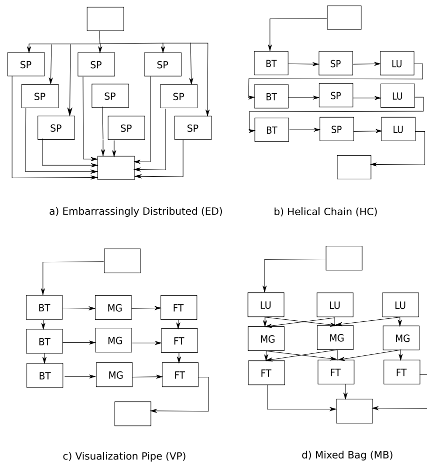


Figure 1: Patterns for NAS Grid

tributed, communicating processes. NGB is frequently used for testing programming tools and compiler optimizations. Furthermore, it provides a real-world example of the use of computational and communication patterns in real-world grid applications.

In general terms, NAS Grid provides facilities for the benchmarking of grid applications that are based on the following four patterns [7] (see Fig. 1): Embarrassingly Distributed (ED), Helical Chain (HC), Visualization Pipe (VP) and Mixed bag (MB).

Benchmarks are produced using the NAS Grid framework by defining graphs of nodes that represent calculations and edges that indicate how results of computations have to be calculated, ordered and passed between nodes. An instance of a benchmark is specified by a static data flow graph (DFG). The DFG consists of nodes connected by directed arcs. NAS Grid includes an imperative and low-level language to describe such graphs by enumerating all nodes and edges. Communication between nodes is asynchronous. A DFG node receives input data from other nodes through its input arc(s); this data is used by the target node(s) to set initial conditions and to perform the target nodes' calculations. A DFG node starts its computation only after it receives all data from its predecessor(s) in the graph. After performing its calculation, it sends the computed result along all of its output arcs.

Example: Global Checkpoint Error Recovery

In order to illustrate the problems in the implementation of distributed algorithms over grid architectures and evaluate our solution to them we have investigated a fundamental service in grids, global checkpoint error recovery.

Checkpoint recovery is a service that facilitates the recovery and the continuation of an interrupted computation. This service is essential for large, long-running computations to minimize downtime and other costs incurred by system or applications failures that stop the computations. A checkpointing service periodically saves the state of the applications and the manipulated data. For a distributed application, a distributed checkpoint is a set of local checkpoints, one from each process constituting the overall distributed computation. In this situation, the service must ensure the global consistency of the captured state. Consequently, a checkpointing service has to inspect and modify the local computations in an invasive manner.

In grid environments, global checkpoint recovery is particularly important to facilitate migration and continuation of incomplete computations in the context of temporarily unavailable resources. However, for large scale applications, checkpointing is subject to two specific problems. First, some specific applications embody theoretically and experimentally validated algorithms, whose correctness must not be endangered through source code modification. In this situation, a generic approach that does not require any code modification can be used but impacts the memory footprint and thus frequently is not viable for performance reason [17]. Our solution that uses an aspect based approach allows a checkpointing service to be implemented while transparently modifying the interaction between legacy components with a negligible impact on the memory footprint and other performance characteristics. Second, defining grid algorithms concisely over large-scale topologies depends on the application at hand and is, for instance, very error-prone and tedious using NGB's low-level means for topology definitions. In the case of global checkpoint recovery, a complete representation of the communication state is needed to ensure the necessary global coordination for the capture of a coherent state: a coordination algorithm that may be complex and specific to the application communication model thus has to be developed. Invasive patterns support the concise description of modifications to computations and communications between components and thus enable, in particular, checkpointing to be added modularly to the NGB.

3 Aspect-based invasive patterns for grid programming

In order to address the limitations of current grid implementation methods, we propose to harness invasive patterns [4] to flexibly define grid applications over arbitrary topologies and enable the modularization of crosscutting invasive accesses (*i.e.*, scattered and tangled code) using aspects.

As illustrated in Fig. 2, invasive patterns allow invasive

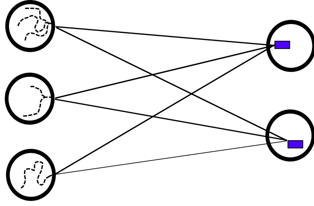


Figure 2: Invasive patterns

data accesses on one machine (dashed lines on the left represent a sequence of local events) and trigger executions on other machines (on the right). By supporting the application of specific behavior to groups of hosts (instead of individual hosts), grid applications can easily quantify over large topologies. This way, it is possible to invasively modify the intrinsic structure of legacy code in grid applications by creating different topologies for different concerns over one application. In this section, we first revisit the essentials of invasive patterns and then present our current implementation of invasive patterns in terms of a transformation from invasive patterns into the AWED system [5], a language and execution platform for aspects with features for explicitly distributed programming.

3.1 Pattern Language

Our language for invasive patterns is based on Aspect Oriented Programming (AOP) [1, 15] and relies on two key concepts: aspects and sequences of aspects.

AOP is a new programming paradigm that investigates the modularization of crosscutting concerns that are separated and implemented in independent units of modularization, the so-called aspects. An aspect is a class-like construct with two additional elements: pointcuts and advice. Pointcuts are language constructs that match sets of join points (events in the execution of a program). Advice is a method-like construct with an associated pointcut definition. When a specific join point is matched during the execution of the base application by a pointcut, the body of the corresponding advice is executed (depending on the advice definition) before, after or instead of the matched join point. Technically, the code of the base application and aspect code are weaved (*i.e.*, compiled) into an executable program. This can be done either at compiling time, using a static weaver, or at execution time, using a dynamic weaver.

An *aspect* is applied to two groups of hosts (see Fig. 3-a). When its pointcut designator matches execution events on the hosts of the left hand side group, its remote advice is executed on the hosts of the right hand side group. If the first group is restricted to containing a single host only we get the farm pattern (Fig. 3-b). If a single host forms the second group we get the gather pattern (Fig. 3-c). Finally,

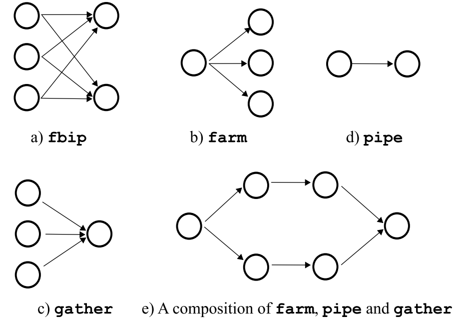


Figure 3: Basic patterns and their composition

a single host in both groups yields a pipe pattern (Fig. 3-d). Note that we do not need to specify all edges of the graph but only the flow from one group of components to another.

Sequences of aspects provide a means to synchronize several pattern-defining aspects: the second aspect in a sequence is activated only once the first has started (or ended) executing its advice. This enables us to compose patterns and realize complex data flow graphs. Figure 3-e, for instance, illustrate a composition of aspects realizing a farm pattern that is followed by two pipe patterns and a gather pattern.

The grammar in Fig. 4 defines the syntax of our language. A sequential pattern P is a sequence of host groups $G_1 \ldots G_n$ separated by aspects $A_1 \ldots A_{n-1}$. A group of hosts G is defined by its hosts H or by sub-patterns P (patterns can be hierarchically composed). A pattern $P = G_1 A_1 \ldots G_n$ denotes the hosts G_n when P occurs on the left hand side of an aspect, and P denotes the hosts G_1 when it occurs on the right hand side of an aspect. Each aspect A defines an around pointcut PCD a $SourceAdvice$ to be executed locally where the joinpoint is matched and a $TargetAdvice$ to be executed remotely. This remote advice can be synchronous but is asynchronous by default. Advice consist of method bodies, *i.e.*, essentially Java statements. Source advice may additionally call the matched base call using the `proceed` construct. Target advice must not call `proceed`, because the corresponding pointcut is matched on the target hosts. When a pointcut matches a joinpoint, it can bind identifiers to values such as the localhost H or method arguments Id. A single pointcut can match several joinpoints on different hosts, so there is a set of bindings $(H, Id*)$ for each host. A pointcut PCD matches method call specified by their signature $MSig$. It can extract context values by binding identifiers to the receiver `target` or its arguments `args`. It can be logically composed, but it can also define a sequence Seq of joinpoints. Following the paradigm of stateful pointcuts [8, 5] (and unlike AspectJ [3, 16]) pointcuts may explicitly specify finite-state automata for matching of sequences of execution events.

$$
\begin{array}{lll}
P & ::= & \texttt{patternSeq}\ G_1\ A_1\ G_2\ A_2 \ldots G_n \\
G & ::= & H\ G \ \mid P\ G \ \mid \epsilon \\
A & ::= & \texttt{aspect}\ \{\ \texttt{around}((H, \mathrm{Id}^*)^*)\text{: } PCD \quad SourceAdvice \quad [\texttt{sync}]\ TargetAdvice\ \} \\
PCD & ::= & \texttt{call}(MSig) \ \mid \texttt{target}(Id) \ \mid \texttt{args}(Id+) \\
 & \mid & PCD\ \&\&\ PCD \ \mid PCD\ ||\ PCD \ \mid\ !PCD \\
 & \mid & Seq
\end{array}
$$

Figure 4: Pattern language

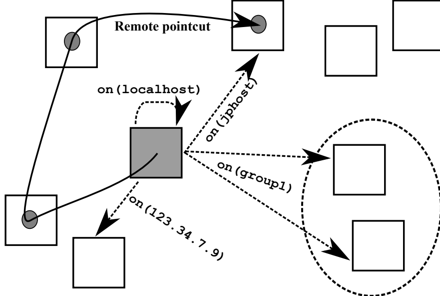


Figure 5: Remote pointcuts and advice in AWED

Such automata are constructed from transitions that may be labelled, advice may then be triggered by the occurrence of a transition with label `l` of the automaton `a` using the pointcut term `step(a, l)`.

3.2 Implementation

The pattern language introduced in the previous section supports the pattern-based construction of grid algorithms by two main concepts: algorithms formulated in terms of conditions about hosts that trigger executions on other hosts and flexible assembly of such (basic) algorithms over topologies expressed using groups of hosts. We have employed AWED [5], a language and execution system for AOP of distributed applications, to implement such grid algorithms. In the following we first introduce the main concepts of the AWED language followed by a description of the implementation of our pattern language in termstransformation into AWED.

3.2.1 Implementing invasive patterns for grids with AWED

The AWED language has been designed as an aspect language for the modularization of crosscutting concerns in distributed systems. In general terms, AWED allows to match sequences of execution events in a distributed system that then trigger remote (advice) executions.

Figure 5 illustrates the two main features of the language: remote pointcut and advice. Pointcuts essentially allow to match sequences of execution events that occur on different hosts. Hosts can be referred to using absolute addresses but can also be defined relative to the host on which an aspect is deployed (term `localhost`), the host colored in gray in the figure. Remote advice can be triggered on other hosts using the `on` specifier. Besides the host specifications available for pointcut definitions, advice execution can also be specified to take place on the host where the pointcut has been matched (term `jphost`).

Pointcuts and remote advice execution may depend on explicitly defined groups of hosts. In pointcuts, such groups may limit matching of execution events to sets of hosts; as to advice executions, groups allow to execute advice on several hosts. Finally, AWED supports synchronous and asynchronous communication between a pointcut and triggered pieces of advice.

Transforming Invasive Patterns into AWED. The AWED language matches the requirements of the language for invasive patterns quite well. Most importantly, AWED aspects can be used to directly implement the modularization of crosscutting accesses for invasive patterns. Modularization of such concerns using finite-state automata in the pattern language (`seq` in Fig. 4) is directly expressible as part of AWED's pointcut language. Furthermore, AWED's notion of remote aspects, together with argument passing from pointcuts to advice, fits some of the communication patterns shown in Fig. 3 quite well. Concretely, execution of remote advice on several hosts in a group corresponds to a `farm`-like communication topology, while execution of an advice on one machine realizes a `gather`-like topology if it has been triggered by events occurring on different hosts.

Fig. 6 presents the implementation of the pattern shown in Fig 2 as an example of the fundamental mechanisms used for the implementation of the four basic patterns underlying NAS Grid benchmarks shown in Fig. 1. In the figure, the aspect matches a sequence of local events on each one of the left hosts by means of the pointcut `invasiveSequence()`. In the sequence definition each state is defined by an atomic pointcut, *e.g.*, the pointcut `call(* *.*(..)) && host("LeftHosts")&&host("localhost")` match all the calls to any method of any class (`call(* *.*(..))`), that occur in a host that is a member of the group of hosts `LeftHosts` and that happens in the host where the as-

```
1  aspect DualGatherSync {
2   public void sendData(BMRequest req, BMResult res){}
3
4   pointcut invasiveSequence():
5     seq(
6       Ti1: call(* *.*(..)) && host("LeftHosts")
7            && host("localhost") > Ti2 || Ti3,
8       Ti2: call(* *.*(..)) && host("LeftHosts")
9            && host("localhost") > Ti2 || Ti3,
10      Ti3: call(* *.*(..)) && host("LeftHosts")
11           && host("localhost"));
12
13  asyncex around(): step(invasiveSequence(), Ti3) {
14    proceed();
15    //get results
16    sendData(req, res);}
17
18  pointcut syncSequence(BMRequest req, BMResults res):
19    seq(
20      T1: call(* *.sendData(..)) && host("LeftHosts")
21          && on("RightHosts") > T2,
22      T2: call(* *.sendData(..)) && host("LeftHosts")
23          && on("RightHosts") > T3,
24      T3: call(* *.sendData(..)) && host("LeftHosts")
25          && on("RightHosts") && args(req, res) > T1);
26
27  asyncex after(BMRequest req, BMResults res):
28                  step(syncSequence(req, res), T3){
29    IS isComp=
30        new IS(req.clss, req.numthreads, req.serial);
31    isComp.runBenchMark();}}
```

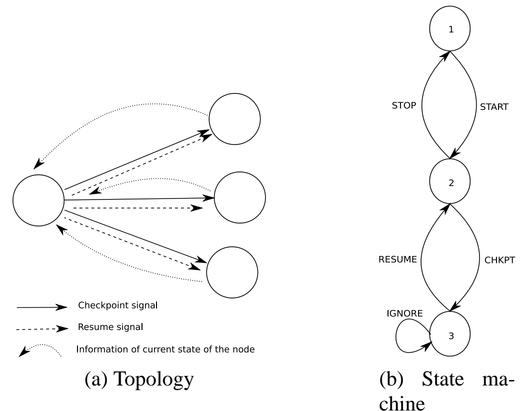
Figure 6: Implementation of a multiple farm-gather pattern

pect is deployed (`host(localhost)`). The advice that is triggered on the last step of such sequence (by means of pointcut `step(invasiveSequence(), Ti3)`) is used to gather specific data and generate an event (`sendData`) to trigger the remote execution in the right hosts. The other sequence pointcut (`syncSequence`) implements a rendezvous: it ensures that the remote calculation is only launched after the three triggering events have occurred. Note that explicit references to hosts are avoided, instead only groups are referred to.

4 Evaluation

To evaluate our approach we have implemented an extension to the NAS Grid benchmarking framework by adding checkpointing support. This implementation coexists with the native communication mechanism of the NAS Grid framework (we used its Java RMI instance); AWED is exclusively used to implement the checkpointing concern. This concern uses a different distributed topology than that directly implementable by the different configurations of NAS Grid. Figure 7a shows the topology structure and distributed messages of the checkpointing algorithm that we have used in the experiment. In the experiment, any node, even an external node, can generate a *Checkpoint* signal: upon reception of that signal, a node stops its current computation, stores a consistent state, sends that state to the centralized checkpointing monitor and waits for a *resume* signal. Thus, the application will use a composition of the farm and gather topologies as presented in section 3 figures 3b and 3c. This simple algorithm does not require synchronization between nodes but needs to weave the underlying application with joinpoints that have to be propagated in the (distributed) grid: this algorithm allows to evaluate the actual overhead of the runtime infrastructure imposed by the AWED implementation of invasive patterns. The algorithm is fully distributed and any node can serve as coordinator of the checkpointing protocol.

Figure 7b shows a representation of the state machine controlling the checkpointing algorithm in the distributed nodes. In the native infrastructure, we have identified two joinpoints per node that are relevant for our implementation of checkpointing. The first joinpoint (START transition) corresponds to the execution point when data from previous calculations is received by a node; the second (STOP) corresponds to a node having just sent calculation results to the next node in the calculation graph. When a checkpoint signal (CHKPT transition) is received a consistent local state (state before calculation) is stored locally and also, by our checkpointing implementation, remotely in the checkpoint data structure. Thus, after a failure, recovery will be carried out locally by each node depending on the state of the node. If it determines that it has been in the third state, it will relaunch the calculation, otherwise no specific action is needed (because the result of the last computation has already been sent).



(a) Topology (b) State machine

Figure 7: Topology and state machine representation of the protocol implementation for check pointing.

Figure 8 shows the implementation of this checkpointing algorithm using AWED. The aspect defines two local fields to store the checkpoint image. The image is created when the first event defining the `START` transition is received by the advice triggered by the pointcut `step(chkptSequence(), START)`. As a second step, the aspect waits for a checkpoint event (transition la-

beled CHKPT) or a finalization event (transition STOP). In case of a checkpoint event the aspect waits for a resume event (see transition RESUME) to reinitialize the calculation. Finally, the transition IGNORE ensures that terminated calculations are ignored and avoid gathering data or restarting computations in this case. This last transition guarantees that no further events are sent after a checkpoint image is captured: checkpointing thus conforms to the notion of consistency introduced in section 2 (a checkpoint is taken between reception of data and the end of the corresponding calculation).

```
1  aspect ChkPtAsp perobject {
2   BMRequest req;
3   BMResults res;
4
5   public BMRequest requestChkPtInfo(){}
6
7   pointcut chkptSequence():
8   seq(
9   START: call(* BenchUnion.startBenchmark(..))
10        && host(localhost) > STOP || CHKPT;
11  STOP: call(* BenchServer.PutArcData(..))
12       && host(localhost);
13  CHKPT: call(* chkimpl.MainConsole.stopCalculNow(..))
14      && !host(localhost) > RESUME || IGNORE;
15  IGNORE: call(* BenchServer.PutArcData(..))
16       && host(localhost) > RESUME || IGNORE;
17  RESUME:
18        call(* chkimpl.MainConsole.startCalculNow(..))
19        && !host(localhost) > STOP || CHKPT);
20
21  before(): step(chkptSequence(), START)
22  { System.out.println("Asp:Iniciando...");
23     BenchUnion comp =
24        (BenchUnion) thisJoinPoint.getCalledObject();
25     req =
26      (new BenchUnionInspector(comp)).getRequest();
27  }
28
29  around(): step(chkptSequence(), IGNORE){
30     return new Object();}
31
32  after(): step(chkptSequence(), RESUME)
33  { BenchUnion comp = new BenchUnion(req);
34     comp.startBenchmark();
35 }}
```

Figure 8: Checkpoint concern implemented using AWED

4.1 Qualitative evaluation

We have first performed a qualitative evaluation by comparing how concise grid applications can be expressed in terms of the native topology configuration language provided by NAS Grid and our pattern language.

A comparison between the native NAS Grid language for the definition of DFGs (fig. 9) and our pattern language (fig. 10) shows that the abstraction of host groups we introduce make the declaration of grid topologies much more concise. A large-scale grid application is frequently composed of over 1,000 processes. Without patterns and pattern composition, the task of defining a grid application relying only on the NAS Grid language is very tedious. Typically one needs to write one line to define a node and one for the link between two nodes. Our language is concise (a few lines define groups and connect them with patterns), and it supports pre-established properties (*e.g.*, synchronization, topology). For the sake of readability, we directly use pattern names such as farm and gather. These terms can be formally defined as syntactic sugar in forms of macros that are expanded into plain aspect definitions: farm(G1,Afarm,G2), for example, becomes G1 {aspect Afarm ...} G2 in terms of the pattern language introduced in Sec. 3.1. (In our prototype implementation, we have used a straightforward script to expand intensional group definitions, such as G2={h1..h7}).

```
1  graph: {
2  title: "ED.A"
3  node:{title: "0" label: "SPTask.A.h0"}
4  node:{title: "1" label: "SPTask.A.h1"}
5  node:{title: "2" label: "SPTask.A.h2"}
6  node:{title: "3" label: "SPTask.A.h3"}
7  node:{title: "4" label: "SPTask.A.h4"}
8  node:{title: "5" label: "SPTask.A.h5"}
9  node:{title: "6" label: "SPTask.A.h6"}
10 node:{title: "7" label: "SPTask.A.h7"}
11 node:{title: "8" label: "SPTask.A.h8"}
12 edge:{sourcename: "0" targetname:"1"}
13 edge:{sourcename: "0" targetname:"2"}
14 edge:{sourcename: "0" targetname:"3"}
15 edge:{sourcename: "0" targetname:"4"}
16 edge:{sourcename: "0" targetname:"5"}
17 edge:{sourcename: "0" targetname:"6"}
18 edge:{sourcename: "0" targetname:"7"}
19 edge:{sourcename: "1" targetname:"8"}
20 edge:{sourcename: "2" targetname:"8"}
21 edge:{sourcename: "3" targetname:"8"}
22 edge:{sourcename: "4" targetname:"8"}
23 edge:{sourcename: "5" targetname:"8"}
24 edge:{sourcename: "6" targetname:"8"}
25 edge:{sourcename: "7" targetname:"8" }}
```

Figure 9: Farm-Gather topology with NAS language

```
1 G1={h0}
2 G2={h1..h7}
3 G3={h8 }
4 gather(farm(G1,Afarm,G2),Acalc,G3)
```

Figure 10: Farm-Gather topology with pattern language

We have implemented the functionality for checkpointing and recovery using AWED by two classes and one aspect accounting for a total of 93 lines of code (LOC). Achieving the same functionality in native NAS Grid using Java RMI will require the modification of the current framework for distribution that amounts to 3939 LOC. We could also create an additional framework for the distribution, concurrency and coordination of the checkpoint functionality, but in both cases we still have to modify the original framework. It is clear that our proposal is much better

understandable and more maintainable, in particular if further evolution or refactoring is required.

4.2 Performance Evaluation

In order to evaluate the performance impact due to our AWED implementation, we have run the NAS Grid benchmark on Grid'5000, a grid of 5,000 processing units distributed over 9 French sites. Note that the number of resources that can be allocated for an individual experiment depends of the number of request from affiliated laboratories.

In order to present the overhead of our AWED-based implementation, we compare the runtime of two different NASGrid Benchmark configurations that represent typical data-flow application topologies: HC, a fully sequential distributed topology; and FG a typical master/slave distributed topology where an initial node propagates tasks to a farm, and then results are gathered on single node. In both cases, each node is running the same component. For each experiment, we have deployed the components on two different clusters located on two different sites and run the experiment 3 times. Figure 11 shows the average overhead due to the AWED framework and the checkpoint service, using two different application topology and a variable amount of computing nodes. As described earlier, the checkpointing service records a consistent state of each host at two joinpoints: first, when a local node receives data from previous calculations, second when the node just sent calculation results to the next node(s) in the calculation graph.

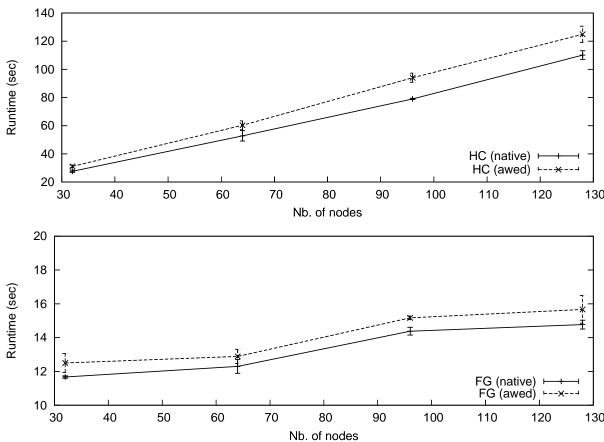


Figure 11: Impact of AWED implementation in NASGrid

The AWED implementation shows an acceptable overhead in the case of the massively parallel (FG) benchmark: in this case the runtimes of the native benchmark and the benchmark with AWED are comparable. On the fully sequential benchmark HC, the global overhead is more important but still acceptable. This is due to the *local overhead* between each node that accumulates over sequential executions. Note that AWED's current implementation generates a distributed message for every call to `startBenchMark` in the class `BenchUnion`: thus the total number of messages including checkpointing is circa double that of the NAS Grid algorithm without checkpointing. It can be expected that further optimization in aspect weaving and message delivery lead to a smaller communication overhead.

5 Related work

While no other approach is directly related to ours in the sense that it investigates language support based on AOP techniques for patterns in the context of grid applications, there is a relevant body of work from several different domains that focuses on some of these characteristics.

Patterns for distributed and parallel programming. Distributed applications are often built using rich middleware structures, which provide basic services for the implementation typical computation and communication patterns. In the domain of grid computing, for instance, Globus, one of the most popular middleware for grid architectures, uses the resource specification language RSL [11] to support the deployment of applications. In contrast to the notion of invasive patterns advocated here, computation and communication patterns have to be programmed in an ad hoc manner, in particular because RSL cannot describe connection constraints between the parts of an application that depend on execution state that is encapsulated by the distributed nodes.

Architectural and programming patterns are quite popular for the programming of (massively) parallel applications. Much work has been done, for instance, on so-called skeletons following Cole's seminal work [6]. More recent work has focused on the application of such pattern-based parallelism to larger-scale imperative applications (see, *e.g.*, [19, 18]). Most of these approaches essentially rely on an underlying regular communication topology and use of a homogeneous synchronization model, two properties that do not hold for the applications we are targeting. Furthermore, crosscutting accesses to execution state is not considered in these approaches.

Aspects for grid applications. Some research and industrial approaches have addressed the use of AOP techniques in the context of grid applications. Sequential aspects have been used to implement monitoring and management of grid applications [12]. Furthermore, they have been employed to address composition in workflow systems for grid services [14]. Finally, recent industrial efforts, such as the Gridgain approach [13] claim to use AOP to enable transparent configuration and modification of grid applications.

These approaches apply directly traditional sequential AOP techniques and do not explore declarative support of aspects to define and implement fully distributed invasive patterns as motivated by our research. As discussed in this paper, such approaches cannot implement crosscutting concerns in distributed applications as concisely as using our proposal if such a concern has to refer to different nodes.

6 Conclusion

In this paper, we have investigated two major difficulties in applying patterns to grid applications on the implementation level: the need for declarative topology definitions and crosscutting access of patterns to non-local execution states. We have harnessed a recent notion of invasive patterns for distributed programming to tackle these issues. Such patterns extend well-known regular computation and communication patterns over arbitrarily large topologies by means for the modularization of access to non-local state that triggers communication between nodes. Finally, we have extended the NAS Grid benchmarking software by a checkpointing service. Our qualitative and quantitative evaluation has shown that our approach is expressive and efficient.

This work offers several opportunities for future work. For instance, AWED supports groups of hosts that evolve at run-time, which should allow to support more dynamic grid applications. Second, invasive patterns introduce structures that could enable analyses of properties over applications (*e.g.*, check topology invariants). Finally, grid applications often depend on their correct execution and synchronization over the underlying system topology: the formal definition of invasive patterns presented in this paper should be useful to define and formally proof such properties of grid applications using invasive patterns.

7 Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see `https://www.grid5000.fr`).

References

[1] M. Akşit, S. Clarke, T. Elrad, and R. E. Filman, editors. *Aspect-Oriented Software Development*. Addison-Wesley Professional, Sept. 2004.

[2] *Proceedings of the 5th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'06)*. ACM Press, Mar. 2006.

[3] AspectJ home page. `http://www.eclipse.org/aspectj`.

[4] L. D. Benavides Navarro, M. Südholt, R. Douence, and J.-M. Menaud. Invasive patterns for distributed applications. In *Proc. of the 9th Int. Symp. on Dist. Objects, Middleware, and Applications (DOA'07)*. Springer Verlag, Nov. 2007.

[5] L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly distributed AOP using AWED. In *Proc. of the 5th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'06)*. ACM Press, Mar. 2006.

[6] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[7] R. F. V. der Wijngaart and M. A. Frumkin. Evaluating the information power grid using the NAS Grid benchmarks. In *High-Performance Grid Computing Workshop at IPDPS'04*. IEEE, Apr. 2004.

[8] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. of the ACM SIGPLAN/SIGSOFT Conference GPCE'02*, pages 173–188. Springer-Verlag, Oct. 2002.

[9] I. T. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Proc. of Euro-Par'01*, pages 1–4, London, UK, 2001. Springer Verlag.

[10] R. Frumkin, M. Van der Wijngaart. NAS Grid benchmarks: a tool for grid space exploration. *High Performance Distributed Computing*, pages 315–322, 2001.

[11] Globus3 resource specification language (rsl). `http://www-unix.Globus.org/toolkit/docs/3.2/ gram/ws/developer/mjsrslschema.html`.

[12] M. Grechanik, D. E. Perry, and D. Batory. Using aop to monitor and administer software for grid computing environments. In *Proc. of COMPSAC'05, Vol. 1*, pages 241–248. IEEE, 2005.

[13] Gridgain, a computational grid framework. `http://www.gridgain.com/product.html`.

[14] N. Joncheere, W. Vanderperren, and R. V. D. Straeten. Requirements for a workflow system for grid service composition. In *Business Process Management Workshops*, pages 365–374, Berlin, Germany, 2006. Springer.

[15] G. Kiczales. Aspect oriented programming. In *Proc. of the Int. Workshop on Composability Issues in Object-Orientation (CIOO'96) at ECOOP*, July 1996. Selected paper published by dpunkt press, Germany.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, pages 327–353. Springer-Verlag, Berlin, June 2001.

[17] L. Silva and J. Silva. System-level versus user-defined checkpointing. *Reliable Distributed Systems, Proc. of the 17th IEEE Symposium on*, pages 68–74, Oct 1998.

[18] S. Siu, M. De Simone, D. Goswami, and A. Singh. Design patterns for parallel programming. In *Proc. of PDPTA'96)*, pages 230–240. C.S.R.E.A. Press, Aug. 1996.

[19] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proc. of the ACM SIGPLAN Symposium PPOPP'03*, pages 203–215, June 2003.