Checkpoint and Run-Time Adaptation with Pluggable Parallelisation

Bruno Medeiros
Departamento de Informática/CCTC
Universidade do Minho
Braga, Portugal
brunom@di.uminho.pt

João L. Sobral
Departamento de Informática/CCTC
Universidade do Minho
Braga, Portugal
jls@di.uminho.pt

***Abstract*— Enabling applications for computational Grids requires new approaches to develop applications that can effectively cope with resource volatility. Applications must be resilient to resource faults, adapting the behaviour to available resources. This paper describes an approach to application-level adaptation that efficiently supports application-level checkpointing. The key of this work is the concept of pluggable parallelisation, which localises parallelisation issues into multiple modules that can be (un)plugged to match resource availability. This paper shows how pluggable parallelisation can be extended to effectively support checkpointing and run-time adaptation. We present the developed pluggable mechanism that helps the programmer to include checkpointing in the base (sequential). Based on these mechanisms and on previous work on pluggable parallelisation, our approach is able to automatically add support for checkpointing in parallel execution environments. Moreover, applications can adapt from a sequential execution to a multi-cluster configuration. Adaptation can be performed by checkpointing the application and restarting on a different mode or can be performed during run-time. Pluggable parallelisation intrinsically promotes the separation of software functionality from fault-tolerance and adaptation issues facilitating their analysis and evolution. The work presented in this paper reinforces this idea by showing the feasibility of the approach and performance benefits that can be achieved.**

Keywords-application-level checkpointing; run-time adaptation; pluggable parallelisation; aspect oriented programming

I. INTRODUCTION

The first stage of enabling an application to run on computational Grids (a.k.a, *gridification*) relies merely on adapting the application to use Grid services, without further application-specific improvements. A pragmatic example is the MPICH-G2 [1] that enables applications written in MPI to run on computational Grids, using the Globus toolkit. The next stage in application *gridification* is the adaptation of application behaviour to resources effectively committed to application execution [2]. In Grid systems resources committed to the application can change during application execution. Examples of such variability include: resource failure, requests to release allocated resources for use by higher priority jobs and availability of new resources. Thus, the application might have to increase or decrease its resource usage during execution. At an extreme case the application can be forced to restart on a different set of resources. Grid applications can take a long time to complete, becoming essential to address the failure of resources. One way to address these failures is to periodically save application data to disk and, in the case of a failure, restart the application from the last checkpoint.

An essential requirement for fault-tolerance and adaptability mechanisms in Grid systems is portability. Fault-tolerance mechanisms should avoid changes to current Grid middleware and the information should be saved in a portable manner to allow an easy application migration across the heterogeneous set resources typical of a Grid environment. Moreover, the amount of saved information must be minimal, as Grids have dedicated remote storage elements, which increase the latency to store and retrieve data, when compared with traditional cluster environments.

System-level checkpointing mechanisms are intrinsically non-portable, as they require changes to the underlying middleware; they save information on a machine dependent format and tend to save unnecessary data since they do not take advantage of application specific knowledge. On the other hand, application level mechanisms avoid these drawbacks but they require an additional effort from the programmer to insert code for checkpointing.

Self-adaptive systems require strategies for resource selection and malleable applications. The former involves the selection of the most appropriate set of resources to assign to the application. For instance, in [3] a strategy is presented to decide how many computing resources should be allocated to the application by periodically collecting performance data from the application processors. The latter is concerned with reshaping the application to effectively use the given set of resources. Current approaches to reshaping are based on over-complete decompositions, where parallel tasks are coalesced when the resources committed to the application are less than the number of potential parallel tasks [4]. These works are mainly concerned with providing low cost implementations of excess of parallel tasks and/or with performing data redistributions. They can be regarded as providing a dynamic mapping of applications to resources, according to Foster design phases [5]. Thus, the application structure remains basically the same, only the mapping to resources

Published in the 40th International Conference on Parallel Processing (ICPP'11)

changes. This can limit the adaptability to only a few tens of resources, since adapting an application with thousands of potential parallel tasks can introduce high costs, especially when running on a small set computing resources.

In this paper we address the second issue, by exploring a strategy to effectively write malleable parallel applications that can reshape the parallelism structure. For instance, we address the reshaping of a sequential execution mode to concurrent execution based on shared memory. We assume that the adequate set of resources committed to the application is identified with other tools/methodologies (e.g., [3]).

The key insight presented in this paper is that using an approach based on pluggable parallelisation, which localises parallelisation issues into well-defined modules, enables an effective way to adapt the application to the resources committed to application execution. Moreover, fault-tolerance is automatically provided in parallel execution environments by requesting the programmer to specify fault-tolerance in the sequential base code, using the provided pluggable mechanisms.

Pluggable parallelisation intrinsically promotes the separation of software functionality from fault-tolerance and adaptation issues facilitating their analysis and evolution. In this paper we extend previous work on using aspect oriented programming to modularise parallelisation and *gridification* issues [6][7][8] to support pluggable application-level checkpointing and dynamic adaptability of parallelism.

The next section compares this work against other approaches. Section III briefly presents the pluggable parallelisation approach and section IV describes its extension to support application-level checkpoint and adaptation. Section V presents performance results and the last section concludes this paper and outlines future work.

II. Related Work

OpenMP [9] introduces directives to specify parallelisation issues that can be ignored in a strict sequential execution. Thus, in OpenMP we can unplug the parallel code, but this is only possible if no explicit calls to the OpenMP API are performed. Moreover, parallelisation can only be unplugged at compile-time. OpenMP presents strong limitations when specifying efficient applications for distributed memory machines due to its centralised execution model. Application-level checkpointing mechanisms for OpenMP were proposed in [10].

MPI and its Grid enabled version, MIPCH-G2 [1] imposes a fixed parallelism structure, i.e., the structure cannot change during execution. MIPICH-G2 can use specific communication middleware among nodes and supports the development of configuration aware applications. Although, it is hard to support a high degree of adaptability as the parallelism-related code in mixed with the domain specific code. Thus, with MPI it is only possible to use over-decomposition to support adaptive applications, leading to an additional overhead when multiple processes are mapped into the same physical resource. Checkpointing and adaptability mechanisms for MPI were proposed in [4][11].

Skeleton-based approaches [12] present a higher degree of adaptability, as only the high-level parallelism pattern is specified in the application (e.g., a Farm or a Pipeline), giving flexibility to the skeleton implementation to find the best implementation for each running conditions. This approach can also encapsulate fault-tolerance issues in the skeleton implementation. Although there are some approaches that support Grid systems [13], these approaches do not yet support the dynamic reconfiguration of the parallel structure associated to an application. Skeleton based approaches have similarities to the proposed work as they also explicitly separate domain-specific code from parallelisation issues and they give more freedom to find the best running strategy for each pair skeleton/target platform (e.g., the parallelism degree).

Previous work on adaptability relies on optimising the mapping of a *fixed* parallel structure into a given set of resources (which might change dynamically). These are optimisations of the *mapping* of application level tasks into available resources. For instance, optimisation of a skeleton farm in [14] is concerned in performing the best scheduling of tasks on a specific set of resources. Similarly, [15] deals with reconfiguring ASSIST applications to effectively leverage the available resources, by dynamically changing the mapping of virtual processors into processing elements. Work in [16] presents a system where applications are reconfigured if the performance contract is not meet. Reconfiguration is performed by checkpointing application state to disk and restarting on a different set of resources. A similar strategy for MPI-based applications is presented in [4], where a set of MPI processes can be restarted on a different platform. Overall these approaches can adapt the mapping of processes to processors but they cannot change the amount of parallelism within an application to match a particular target platform, resulting in some overhead when the parallelism degree largely surpass the number of available compute resources.

One way to avoid the overhead of over-decomposition, when running on a small amount of resources, is to promote more malleable applications by dynamically creating tasks. Divide and conquer pattern of parallelism was proposed for that goal [17][3], avoiding the need to migrate running tasks as only newly created tasks are used to adapt the parallelism degree of the application. This approach imposes additional burden to programmers, as tasks should be dynamically created during application execution, a model that is not adequate for most applications and might impose additional overheads due to dynamic task creation.

Our previous work addressed the modularisation and decomposition of parallelisation issues into several modules [6] and on pluggable modules to Grid-enable existing applications [7], with fewer changes than current approaches. This previous work focused on *separating*

parallelisation issues from domain specific codes by means of pluggable parallelisation [8], that *at compile-time* (or load-time), rewrite the application-specific code to deliver Grid-enabled codes. In this approach, the same base code can be used for a strict sequential execution, shared memory systems and distributed memory systems, by plugging different parallelisation modules. Thus, a single code base can be *statically* adapted for a wide range of platforms.

In this article we extend our previous work to also modularise fault-tolerance and adaptability issues and show that modularising parallelisation helps to develop adaptive Grid applications. Our model is extended to support checkpointing and reconfiguration, thus achieving malleable Grid-enabled codes.

III. PLUGGABLE PARALLELISATION

This section presents the programming model underlying the pluggable parallelisation approach. The next subsections give an overview of the programming model, present the programming constructs for shared and distributed memory and show an illustrative example. Additional details can be found in [8][18].

A. Programming Model

Our programming model resemble to the OpenMP model, as in OpenMP the parallelisation process can be separated from the writing of domain specific code. In OpenMP programmers can start by developing the domain specific code and later introduce OpenMP directives to specify parallel execution. These directives can be seen as user specified application rewritings to derive the parallel version of the code (actually, an OpenMP compiler rewrites the code to generate a parallel version). OpenMP fails, however, to provide alternative parallelisation for the same domain specific code as the parallelisation process requires invasive changes to the domain specific code.

The key of pluggable parallelisation is to regard the parallelisation process as an optimisation phase where domain specific code is rewritten to execute in parallel, according user specified pluggable modules. In this approach, like in OpenMP, programmers start by writing the domain specific code. Parallel programming abstractions specify how to rewrite the base code to enable parallel execution. The key difference is that these rewriting are provided in separated modules and we also support programming abstractions for distributed memory. This approach introduces several key benefits:

- *Modularity*: the code that specifies parallel execution is confined to well defined modules;
- *Incremental development*: it is possible to start with simpler (or "sequential" like) versions and later to develop more complex parallel versions by improving or adding more parallelisation modules;
- *Pluggable*: the domain specific code can run without parallelisation and it is possible to develop alternative parallel versions and can be selected according to the target platform/applications.

Pluggable parallelisation [8] is based on a set of well know parallel programming abstractions. Currently the focus is on object-oriented applications, as they provide a richer set of programming abstractions to support modular programming. We developed programming abstractions that support execution models similar to OpenMP (for shared memory systems) and MPI (for distributed memory systems), but those abstractions act as program rewrites (more specifically, they rewrite object implementations). This allows the deployment of multiple versions of the same application:

1. Sequential version, based on the domain-specific code;
2. Parallel version for shared memory systems, by plugging parallelisation modules for shared memory.
3. Parallel version for distributed memory by plugging parallelisation modules for distributed memory systems.

Overall, pluggable parallelisation addresses the complexity of the development of parallel applications by promoting an incremental development. Developers start with simple parallel versions and progressively improve the code by developing/extending modules that specify parallel execution issues. The modules can also be composed to attain complex forms of parallelisation (e.g., hybrid shared/distributed memory parallelisation). The concept of pluggable parallelisation has been applied to develop parallel versions of many applications, including all JGF benchmarks [19][8], a Java framework for evolutionary computation [20] and a framework for molecular dynamics simulations [21].

B. Programming Abstractions for Shared Memory Systems

The programming model for shared memory systems follows an execution model similar to the OpenMP model. Execution starts in a main thread that can spawn a team of threads to execute a block of code. We provide the concept of *parallel method*: a method that is executed by all newly created threads in the team. Synchronisation occurs when the parallel method finishes. Data sharing constructs can protect shared data (object fields or other objects) from concurrent accesses. For this purpose we provide *synchronised*, *master* and *single* methods that provide functionality similar to the OpenMP directives with identical names. For instance, *synchronised* methods are executed in mutual exclusion when executed by a team of threads. A *barrier* construct inserts a barrier before or after a method execution. Recently two new mechanisms where added: *for* methods and *thread local* fields. The former provides functionality similar to the *for* work sharing construct and *thread local* fields can be used to avoid synchronisation by providing a local object field to each thread in the team.

C. Programming Abstractions for Distributed Memory Systems

The programing model for distributed memory slightly differs for the MPI model. Like MPI, it is based on a SPMD model but it mainly relies on the concept of object aggregate. An object aggregate is a class of objects that have a single instance on each node and transparently replaces a single object instance in the domain specific code. In our model this is specified by the *Replicate* abstraction. Aggregate members are identified by their Id. Calls to the original object instance are executed by the aggregate element with Id 0 (i.e., the object instance transparently replaced by the aggregate). Several primitives control the way method calls are executed by the aggregate. These calls can be executed in parallel by all elements in the aggregate, using the same or a specific parameter for each element, or delegated to a specific aggregate element. When the original call returns a value, a special function can be specified to combine the return result of each method execution to a single value.

Data structures (and objects) created by aggregate elements are replicated on each node of an aggregate. There is an exception for object data fields consisting of primitive data. This primitive data can be partitioned among aggregate elements, according to a pre-defined partition (block, cyclic and hybrid). Since we start from domain specific code, using centralised view of the data, we specify the points in execution where data is partitioned and scattered, gathered and updated. Moreover, user defined data partition and data updates are also possible.

D. Illustrative Example

We illustrate this programming model by presenting a distributed memory parallelisation of the JGF Series benchmark [19] (Figure 1). In this case, the domain specific code is presented in black and the distributed memory parallelisation is presented as comments (in red/italic). This programming model is inspired in OpenMP, although we use a more powerful template-based notation in order to overcome some composition limitations of annotations. For understandability purposes we inserted the parallelisation code as comments in the domain specific code, but usually they are specified in a separate module (e.g., file).

```
...
// Partitioned<TestArray,BLOCK>
double TestArray[][] = ...
...
// ScatterBefore<Do(),TestArray>
void Do() {
  ...
  for (int i = 1; i < TestArray[0].length; i++) {
    TestArray[0][i] = TrapezoidIntegrate(/*.. */);
    TestArray[1][i] = TrapezoidIntegrate(/*.. */);
  }
}
// GatherAfter<Do(),TestArray>
```

Figure 1. Distributed memory parallelisation of the JGF Series benchmark

The *Partitioned<TestArray,BLOCK>* declares that *TestArray* object field will be distributed block-wise among aggregate elements. *ScatterBefore<Do(),TestArray>* declares that each partition should be updated (using the data from the aggregate id 0) before the execution of method *Do*. The reverse operation is performed after execution of method *Do*: data is again collected in the master aggregate member.

In this programming model it is possible to express most common types of parallel applications in a way such that the parallelisation code can be unplugged for a strict sequential execution ([18] presents an example of Farming and an Heartbeat parallel applications; more recently we re-implemented all JGF parallel benchmarks [19] in this programming model [8]). This also enables the development of alternative parallelisations. For instance, a shared memory parallelisation could be implemented by declaring the *Do* method as parallel (*ParallelMethod<Do()>*) and by using the *for* construct to schedule calls to the *TrapezoidIntegrate* method among threads in the team.

IV. Checkpointing with Run-time Adaptation

This section describes the extensions performed to support checkpointing and run-time adaption. We start by describing the approach used for checkpointing and later describe the run-time adaptation. The key aspect of this paper is to show that there is a minimal effort to improve our model to support checkpointing and run-time adaption in a modular way. We also show that there is a performance advantage of providing different versions that can better match the target architecture and running conditions.

A. Checkpointing

Checkpointing techniques periodically save application state into a permanent storage to be able to recover the application state in the case of a failure.

On Grid systems the main requirements are portability and minimisation of the information saved. Application-level mechanisms accomplish these requirements but they increase the burden of the programmer. Our idea is to minimise this extra burden for programmers that develop applications based on pluggable parallelisation.

Application-level checkpointing mechanisms require solutions for three key issues:

1. Indication of data to be saved/recovered;
2. Identification of the points in execution where checkpoint can be taken;
3. How to save/recover the call stack;

These are addressed in our approach by:

1. Monitoring a set of data fields (object *allocations*) that are to be saved into the checkpoint data;
2. Specifying a set of *safe points* that are points in execution where the checkpoint can be taken;
3. Rebuilding the call stack on application restart, replaying the application by using *safe points* and *ignorable methods* (described later).

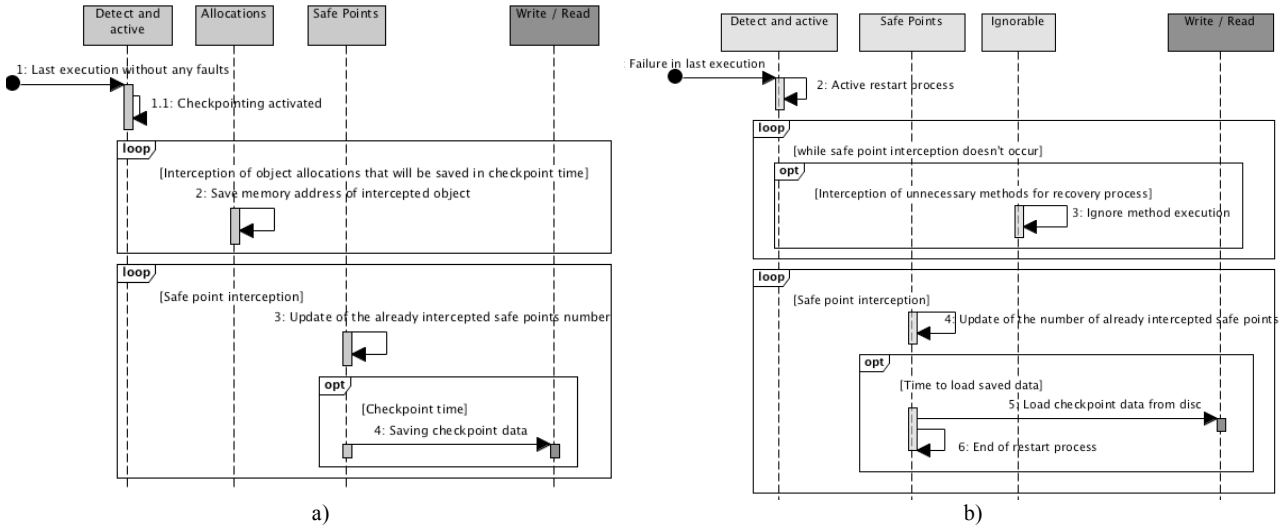

Figure2. a) Checkpoint and b) restart phases

Checkpointing applications is performed as follows (Figure 2a): 1) at application start-up, the *pcr* module verifies if the last execution was concluded without failures; this is accomplished by rewriting the "main" application method; 2) if the last execution completed successfully, the application runs normally and the *allocations* module keeps track of the address of data that must be saved; this is accomplished by monitoring all data allocations; 3) when a safe point in execution arises the *safepoints* module increments the number of executed safe points and 4) when a predefined set of safe points is executed the data in addresses gathered by *allocations* module is saved along with the number of executed safe points.

Application restart in the case of a failure relies on a set of *ignorable methods* that can be skipped during restart. Application restart proceeds as follows (figure 2b): 1) at application start-up, the *pcr* module identifies a failure in the last execution activating the replay mode; 2) the *ignorablemethods* module skips the execution of methods that can be safely ignored. 3) the *safepoints* module increments the number of executed safe points and 4) when the number of safe points saved in the checkpoint file is accomplished the checkpoint data is loaded and execution proceeds normally from that point.

In this approach the programmer must provide:

1. Application data fields to save
2. A set of safe points
3. A set of ignorable methods

The *SafeData<T.field>* template is used to express object data field(s) that should be saved. Frequently these fields are the same that are declared as *partitioned* in the distributed memory parallelisation.

Our approach relies on a replay mechanism to reconstruct the stack at application restart in the case of a failure. With this strategy we have a portable solution since the restart mechanism is completely implemented at application level.

Conceptually to restart the application at the same execution point we need to save every method call and its parameters. We avoid this additional overhead with *safe points* and *ignorable methods*. With safe points we only need to keep track of method executions that are in the call stack when safe points are reached.

IgnorableMethods template allows the programmer to specify method executions that can be safely ignored (i.e., during restart the execution of these methods is skipped). This approach also provides an additional benefit: we actually only need to keep track of the number of safe points executed. Thus, to rebuild the call stack we only need to replay the application until the number of safe points executed is reached.

The *SafePoints* template specifies points in execution where a checkpoint can be taken. The selection of the set of safe points is a trade-off between checkpointing overhead and computation lost when a failure occurs. Note that a checkpoint might be taken only after a set of safe points.

Currently *SafeData, IgnorableMethods* and *SafePoints* are specified by the programmer, but we are developing a tool to help the programmer to identify those. Although there are some proposals in the literature to automate this process, the main issue here is that the programmer only needs to focus on specifying checkpointing of the (base) sequential version of the code.

To summarise, in our approach, the programmers' burden to introduce checkpoint in their code is the identification of safe data fields, ignorable methods and safe points. All the additional code required to take application

snapshots and to restart the application is provided by our system. Two important benefits arise from this approach: 1) the base code (domain-specific code) remains unchanged following the philosophy of pluggable parallelisation, by providing an additional set of templates that localise fault-tolerance related issues and 2) we automatically provide mechanisms to perform checkpointing in shared and distributed memory systems.

Checkpoint in shared memory systems is performed as follows. When a checkpoint is to be taken (i.e., on a safe point) we introduce a barrier before and another after the safe point. When all threads have reached the first barrier the master thread saves the data specified by the *SafeData* template and the number of safe points executed. Restart is preformed by replaying the application as on a sequential execution, but *parallel methods* are still executed to rebuild the number of threads and their corresponding call stack. A barrier is introduced after the safe point where the checkpoint was taken. The master thread reads the saved data when reaching that safe point and then releases the other threads waiting at the barrier.

Checkpoint in distributed memory systems is performed as follows. We perform checkpoint on each process as in the sequential case, only special care must be taken to ensure that every process takes the snapshot on the same safe point. We provide two implementation alternatives to save *partitioned* data fields. In the first case, each process takes a local snapshot. In that case we need to introduce two global barriers, as in the case of the shared memory. In the second alternative we collect the partitioned data on the master node, which avoids the need for barriers (this is possible in our programming model, since we know how the data is partitioned among processes).

Collecting the data and taking the snapshot at the master process has the advantage of making it possible to restart the application on any of the execution modes supported by pluggable parallelisation: 1) sequential execution; 2) parallel execution in shared memory systems and 3) parallel execution in distributed memory systems. This is possible since the checkpoint data is the same in all environments. Thus, adaptation can be performed by saving the checkpoint data and restarting the application on a different execution mode. An additional benefit of this approach is that we can also checkpoint a hybrid shared/distributed memory parallelisation. The next section describes how adaptation can be performed without restarting the application.

B. Run-time Adaptation

The model presented in the previous section assumes that the program was developed in such a way that we can introduce parallel methods and object aggregates with program transformations. This is close to the OpenMP philosophy of introducing annotations to specify parallel execution, but it can cover a wider set of parallelisations. Although, we assume that these transformations are statically applied. With this static approach we can only delay the selection of a particular parallelisation up to load time (e.g., using some load-time program transform technique). To support run-time adaptability on computational Grids we need to extend this approach to plug these transformations during run-time. Next we describe how to extend each of our programming abstractions to support their application at run-time.

Data sharing constructs are the simplest to apply at run-time, as it is ensured by design that these transformations can be (un)plugged without affecting the program correctness. *Parallel Methods* are a bit more complex, as we need to spawn or destroy a team of threads depending if we are plugging or unplugging the mechanism.

The aggregate abstraction is more complex to apply at run-time, as the application state may be distributed across the aggregate. We use the partitioning information to deal with this issue. Thus each class field must be is marked as *Replicated*, *Partitioned* or *Local* (by default, fields are considered *Local*). This information is used by the run-time system to decide how the state of the aggregate is merged into a single instance and how to transform an instance of a class into an aggregate. *Replicated* fields are duplicated on all aggregate elements. When we transform an instance into an aggregate we set this type of field to the same value as the original instance. *Partitioned* fields usually correspond to arrays that are partitioned across aggregate instances, so we can use the corresponding scatter or reduce primitive. *Local* fields are only local to each aggregate element, so they are not subject to any transformation.

Adaptability protocol. The adaptability protocol relies on application execution points where changes can be made to the parallelisation (e.g., safe points). Thus, requests to adapt the application parallelism structure are managed on these safe points.

To describe the adaptability protocol we differentiate between the expansion phase (e.g., the application will use more resources) and the contraction phase (e.g., the application will use less resources). In both cases we discuss how we can change from a sequential execution into a concurrent execution (shared memory based) to a cluster execution (distributed memory based) and vice-versa.

Expansion of Resource Usage. The first type of resource usage expansion is to move from sequential execution to a concurrent execution (i.e., multiple concurrent activities in a single node). For *data sharing constructs* we can simply activate the corresponding implementation, but when the adaptation is done on the context of a *Parallel Method* (i.e., parallel region) we need to spawn multiple threads to execute the given method. For this purpose, when running inside a potential parallel region, we track the beginning of the parallel region to be able to replay the parallel region for threads beside the master thread. Thus, when the move from sequential to concurrent execution occurs inside a parallel region, we replay the execution inside parallel region for

each new thread, in a manner similar to the restart of the application, but just from the beginning of the parallel region. This is done to build the correct calling stack on each thread in the team. A similar strategy can be used to increase the number of running threads. Using this strategy, based on replay, allows the mechanism to be highly portable as it is implemented at application-level. The *for* work sharing is addressed by activating the corresponding implementation during the replay. In this way, each thread will get the call stack that it would have if the program ran with concurrency activated from the start. *Thread local* variables are updated with the value of the main thread.

The second type of resource expansion is the move from a single node configuration (either sequential or concurrent) to a multi-node configuration. In both cases it involves introducing *aggregates* of objects. The move from sequential execution to a cluster-based one consists on creating an aggregate of objects by replicating all data and the call stack of the sequential program on each node. We perform this task by replaying the application on the additional nodes until they reach the same safe point (using the same implementation strategy as for checkpointing). Data of *partitioned* aggregate object fields is distributed according to the user specified partition strategy in the parallelisation for distributed memory systems. This strategy is used to increase the number of nodes.

Expansions of resource usage for hybrid shared/distributed memory parallelisation require multi-step adaptations. For instance, when moving from a single-node sequential execution to a multi-node of multi-core machines, requires one first step to move from sequential execution to a parallel execution on multiple nodes and a second local step on each machine to move from a local sequential execution to a concurrent execution. We address this issue simply by composing the adaptation protocols, first applying the protocol to move from a single node configuration to a multi-node configuration (each involves global coordination among nodes) and then locally applying the protocol to move from an intra-node sequential execution to a concurrent execution.

Contraction of Resource Usage. The first type of resource contraction is the move from a concurrent execution to a sequential execution. In this case all *data sharing constructs* can be simply deactivated in a coordinated manner. Coordination is required since the parallel region is being run by multiple threads and it ensures that all threads are synchronised in a global barrier before shutting down threads in excess. Parallel methods are simply managed by shutting down the additional threads and disabling work sharing constructs. Shutdown is made gracefully by executing methods with empty operations until the thread gets to the end of the parallel region.

The second type of resource contraction is the move from a multi-node configuration to a single node. The rationale of this adaptation is similar to the move from a concurrent execution to a sequential execution, although, in this case there are remote data that must migrate to the local node.

As in resource expansion, there are cases of resource contraction that require multiple-phases. For instance, when contracting from a cluster of multi-core machines to a single node we need first to contract the execution on each remote node to a sequential execution and then apply the protocol to reduce from multiple nodes to a single node.

V. EVALUATION

The proposed approach was applied to several applications that where previously developed using the concept of pluggable parallelisation: JGF benchmarks [8]; a Java framework for evolutionary computation [20] and a framework for molecular dynamics simulations [21]. In all case studies we found that specifying the safe points, ignorable methods and safe data fields introduces a very small programming overhead, since the required knowledge is acquired during the parallelisation process. More importantly, checkpointing and run-time adaptation is localised into specific pluggable modules.

To evaluate the proposed checkpoint and adaptation mechanisms we present a simple application that illustrates how this approach can be used to adapt parallel applications and we evaluate the benefits and overheads relative to hand written versions. These results were collected on a cluster with two machines, dual Opteron 6174 per node (i.e., 24 cores per machine). This cluster runs Linux x86_64 and Sun Java JDK 1.6.0_13 (more details in about this cluster in search.di.uminho.pt).

We demonstrate the effectiveness of the approach using the JGF SOR benchmark with pluggable parallelisation. This benchmark is a typical scientific application, where a five-point stencil is successively applied to a matrix. This is a benchmark with a very short execution time so it would give an upper bound on the mechanism overhead.

The first test measures the overhead of introducing the code for checkpoint, when 0 or 1 checkpoints are taken. Figure 3 shows the execution time of: 1) the "original" benchmark; 2) when checkpointing is introduced using classic "invasive" techniques and 3) when checkpointing is introduced through pluggable parallelisation (PP). Presented results include sequential execution (seq); execution with 2 to 16 lines of execution (LE) and with 2 to 32 MPI processes (P). These results show that: 1) the overhead of checkpointing is very low, as it would be expected, since the overhead is basically the time required to count safe points, which is less than 1% in most cases; 2) PP does not impose any additional overhead when compared to traditional invasive programming techniques; 3) there is a relevant overhead required to save checkpointing data.

Figure 4 details the cost to save checkpoint data on each environment. It should be stressed that most time overhead is due to the time required to save the application data (seq.) intrinsic to any checkpoint approach.

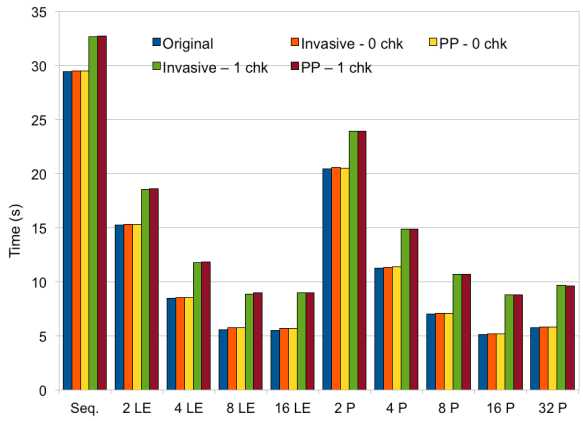

Figure 3. Checkpoint overhead

On a shared memory environment (LE), the time required to save the data slightly increases with the number of threads, since it requires a barrier. The increase on distributed memory systems is higher since the data must be collected at the root node. In the cluster used for these benchmarks this overhead is most noticed with 32 P since the data must move across machines.

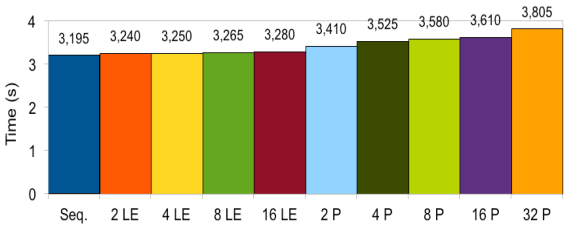

Figure 4. Time to save checkpoint data

The next test measures the time to perform a restart when a failure occurs after 100 safe points (Figure 5). The figure presents separate figures for "reply" and to "load" the checkpoint data.

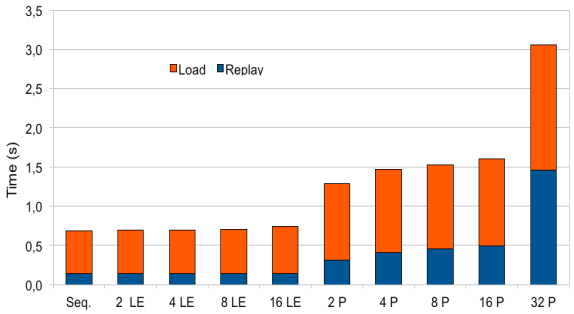

Figure 5. Restart overhead

In all cases the restart overhead is mostly due to the time to load the checkpoint data. In distributed memory this cost is much higher since the data must be scattered across processors after being loaded. Again, this cost is most noticed with 32P.

One important point of the proposed approach is the ability to replay the application on a different environment adapting the execution behaviour to the new environment. Figure 6 illustrates such case by showing the time per iteration. In this case the application started with 2 processes and on iteration 26 it was restarted on 8 processors, shortening the overall application execution to more than half.

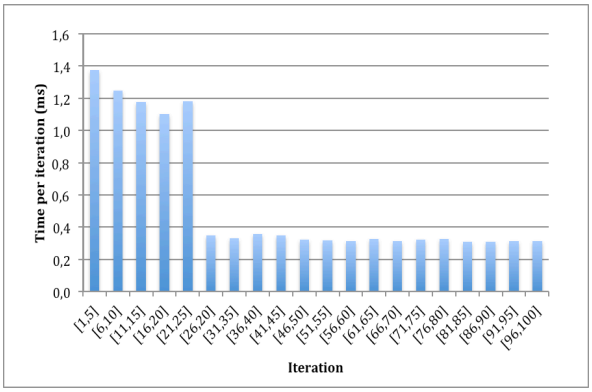

Figure 6. Application restart increasing more resources

Figure 7 compares the benefit of performing adaptation by restarting the application versus using our run-time mechanism to change the number of running threads. On each case the application starts on a set of resources (2, 4 or 8 LE) and during the execution more resources become available (16 LE).

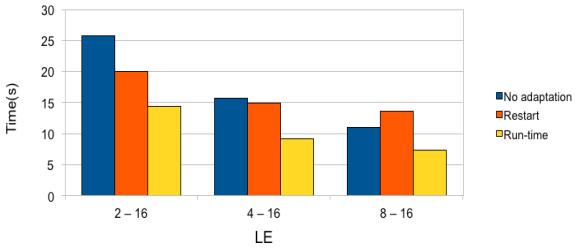

Figure 7. Example of the benefits of resource expansion

In all cases performing adaptation at run-time provides the lowest execution time due to its lower overhead. Actually, the restart overhead increases the execution time when adapting from 8 to 16 LE.

The next benchmark aims to show the overhead of making adaptation through traditional over-decomposition mechanisms (e.g., providing more threads/processes than the number of available resources). Figure 8 shows the overhead of executing the SOR application with several factors of over-decomposition (i.e., number of parallel tasks per processing element).

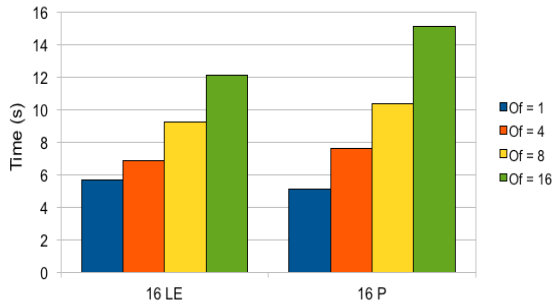

Figure 8. Overhead of over-decomposition

These results show that over-decomposition can impose a high overhead on execution time. For instance, using 256 processes on a 16-processor machine (of = 16) increases the execution time from about 5 seconds to 15 seconds.

The JGF provides 3 versions of the code. A sequential version, a thread-based and a MPI based. Pluggable parallelisation enables changing among these three versions during the execution. Figure 9 compares the running time of the JGF versions against the developed version (on a cluster with eight-core machines), by activating the parallelisation according to resources committed to execution.

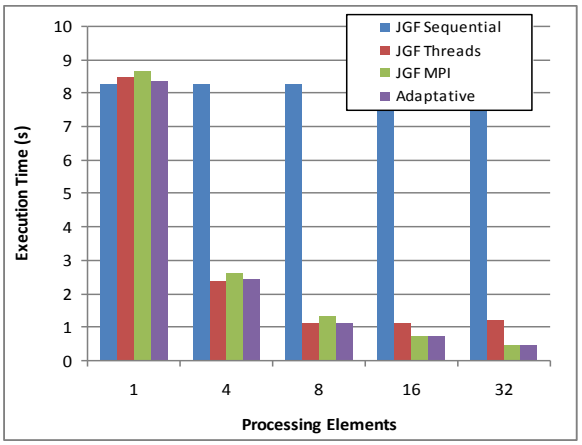

Figure 9. Overhead of adaptability

Execution time of the JGF sequential version does not scale to more than one node, as it does not include support for parallel execution (thus, it always has the same execution time). The JGF Threads version provides the best execution time for a run on 4 and 8 cores (i.e., a single machine). Although, as expected, only the JGF MPI version scales well on a large number of nodes, since the JGF threads version can only use 8 cores (i.e., a single machine). The developed version always attains a performance within 5% of the best version, but, more importantly, it can change from one execution mode to the other during run-time.

VI. Conclusion

This paper presented a new approach to checkpointing and run-time adaptability in computational Grids. The approach is based on the ability to plug parallelisation at run-time to offer an additional degree of adaptability, relative to traditional implementations that rely on a fixed parallel structure. This approach relies on modular parallelisation than can be enabled and disabled during execution.

In this paper we showed the feasibility of this approach and showed that the performance penalty of this model can be very low, when compared with similar hand written versions.

Current implementation of this approach rely on external tools determinate the optimal set of resources to be used by the applications. A natural evolution is to incorporate mechanisms to find opportunities for self-adaptation to improve execution time, by monitoring the application and the system state.

Acknowledgment

This work was developed under AspectGrid (GRID/GRI/81880/2006), PRIA (UTAustin/CA/0056/2008) and GAsPar (PTDC/EIA-EIA/108937/2008) projects, supported by Portuguese FCT and European funds.

References

[1] N. Karonis, B. Toonen, and I. Foster, “MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface”, Journal of Parallel and Distributed Computing, vol. 63, no. 5, May 2003.

[2] C. Mateos, A. Zunino, M. Campo, “A survey on approaches to gridification”, Software Practice and Experience, vol. 38, no. 5, April 2008.

[3] G. Wrzesinska, J. Maassen, H. Bal, “Self-Adaptive Application on the Grid” ACM Symposium on Principles and Practices of Parallel Programming (PPoPP 07), San Jose, California, March, 2007.

[4] R. Fernandes, K. Pingali and P. Stodghill, “Mobile MPI Programs in Computational Grids”, ACM Symposium on Principles and Practices of Parallel Programming (PPoPP 06), 2006.

[5] I. Foster, “Designing and Building Parallel Programs”, Addison-Wesley, 1995.

[6] J. Sobral, “Incrementally developing parallel applications with AspectJ”, 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 06), Greece, Rhodes, April 2006.

[7] J. Sobral, “Pluggable Grid Services”, 8th IEEE/ACM International Conference on Grid Computing (Grid 07), Austin, Texas, September 2007.

[8] R. Gonçalves, J. Sobral, “Pluggable Parallelization”, 18th ACM international symposium on High Performance Distributed computing, (HPDC 09), Munique, June 2009.

[9] OpenMP architecture review board, OpenMP Application Program Interface, Version 2.5, May 2005, www.openmp.org.

[10] G. Bronevetstsky, K. Pingali, P. Stodghill, Experimental Evaluation of Application-Level Checkpointing for OpenMP Programs, International Conference on Supoercomputing (ICS 06), Australia.

[11] G. Rodríguez, M. Martín, P. González, J. Touriño, R. Doallo, CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications, Concurrency and Computation: Practice & Experience, vol. 22, no. 6, April 2010

[12] M. Cole, Algorithmic Skeletons: structured management of parallel computation, MIT press, 1989.

[13] S. Gorlatch and J. Dunnweber, "From Grid Middleware to Grid Applications: Bridging the Gap with HOCs", Future Generation Grids, Springer, 2006.

[14] H. Vélez, Self-Adptive Skeletal Tasks Farm for Computational Grids, Parallel Computing, vol. 32, no. 7, September 2006.

[15] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi and C. Zoccolo, "Dynamic Reconfiguration of Grid-Aware Applications in ASSIST", Euro-Par 05, Lisbon, September 2005.

[16] S. Vadhiyar and J. Dongarra, "Self Adaptability in Grid Computing", Concurrency and Computation: Practice and Experience, vol. 17, n0. 2-4, February 2005.

[17] G. Wrzesinska, R. Nieuwport, J. Maassen, H. Bal, "Fault-tolerance, Malleability and Migration for Divide and Conquer Applications on the Grid", 19th IEEE International Parallel & Distributed Processing Symposium (IPDPS 05), April 2005.

[18] J. Sobral, C. Cunha, M. Monteiro, "Aspect-Oriented Pluggable Support for Parallel Computing", VecPar'06, Rio de Janeiro, Brazil, June 2006.

[19] J. Smith, J. Bull, J. Obdrzálek, "A Parallel Java Grande Benchmark Suite", Supercomputing Conference (SC 2001), Denver, Nov. 2001.

[20] J. Pinho, M. Rocha, J. Sobral, Pluggable Parallelization of Evolutionary Algorithms Applied to the Optimization of Biological Processes, 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 10), Pisa, Italy, February 2010.

[21] R. Silva, J. Sobral, Optimising Molecular Dynamics with product-lines, 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS 11), Namur, Belgium, January 2011.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, "Aspect-Oriented Programming" European Conference on Object-Oriented Programming (ECOOP 01), 2001.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, "An Overview of AspectJ", European Conference on Object-Oriented Programming (ECOOP 01), June 2001.