High Performance Composition Operators in Component Models

Julien Bigot, Christian Pérez

High Performance Composition Operators in Component Models

Julien BIGOT [a] and Christian PÉREZ [b,1]
[a] *INSA Rennes/LIP, France*
[b] *INRIA/LIP, France*

Abstract. Scientific numerical applications are always expecting more computing and storage capabilities to compute at finer grain and/or to integrate more phenomena in their computations. Even though, they are getting more complex to develop. However, the continual growth of computing and storage capabilities is achieved with an increase complexity of infrastructures. Thus, there is an important challenge to define programming abstractions able to deal with software and hardware complexity. An interesting approach is represented by software component models. This chapter first analyzes how high performance interactions are only partially supported by specialized component models. Then, it introduces HLCM, a component model that aims at efficiently supporting all kinds of static compositions.

Keywords. Component, Connector, Genericity, High Performance Computing, Parallel Computing.

1. Introduction

As computing and storage capacities increase, numerical scientific applications would like to benefit from them to improve accuracy—by refining their level of discretization for example—and/or realism—by integrating more phenomena in their models. Therefore, applications are becoming more and more complex.

In addition to the complexity coming from applications, application developers also have to face to the fast evolution of resource complexity. The complexity ranges from nodes which can have multiple processors with many cores, some of which might be GPUs [2], to node infrastructures which can be a supercomputer, a cluster, a federations of clusters, a grid, a cloud, or a federation of clouds—also known as sky computer. Hence, achieving efficient executions of an application on a wide range of infrastructure is challenging and is a burden of application developers.

Existing High Performance Computing (HPC) oriented programming models, however, usually provide models very close to the machines such as MPI [21] for homogeneous clusters/supercomputers, threads for multi-core nodes, or OpenCL [24] for nodes equipped with GPUs.

[1] Corresponding Author: Christian Perez, LIP, ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex, France E-mail: christian.perez@inria.fr.

[2] Graphical Processing Uning

While partitioned global address space (PGAS) languages, e.g. OpenMP [31], aim at hiding such resource complexity, they do not address application complexity as they are usually variations of an imperative or object oriented sequential language, like Fortran and C/C++ for OpenMP. Therefore, they are not primarily targeting the simplification of the design of complex applications on complex resources [35].

An important challenge is hence to design programming abstractions which have to be simple enough to let application developers concentrate on the functional parts of their applications by hiding resource complexity and let applications be able to efficiently make use of a wide range of resources.

An old trend to deal with application complexity is proposed by software component models [27,35]. This approach advocates for composition: it promotes creating applications by *assembling* rather than *developing*. Different composition kinds exist such as spatial and temporal compositions. Two great success stories of the software component approach are the pipe operator in operating systems and data flow models.

This chapter surveys various approaches that have been proposed for using component models for HPC applications. It also describes the High Level Component Model (HLCM), a model aiming at efficiently enabling all kinds of spatial compositions within a unique model. The remainder of the chapter is organized as follows. It first begins by analyzing requirements of HPC applications and whether such requirements can be achieved by standard component models. Then, it studies various works that have brought partial support of parallel programming paradigms in component models. Last, it describes HLCM, a component model that aims to achieve a synthesis of these works.

2. Component Models from HPC's Perspectives

2.1. Overview of Classical Component Models

Component models are an important topic in distributed computing. Many models have been proposed either by industry—EJB [18] by Sun Microsystems, CCM [29] by OMG, SCA [30] by OSOA, or .NET by Microsoft—or by research groups—Fractal [15], GCM [9], HOCs [20], etc.

A consensus on component models seems to emerge from all these works and it is consistent with the definition given by Szyperski et al. [35]. A *component* is a black box that interacts through *ports* that describe what it provides and what it requires. Ports express a kind of interaction, such as remote method invocation or message passing. An *assembly* is a set of interconnected components that can be statically described through an *architecture description language* and/or it can be built dynamically through a model-dependant API. In all cases, a third entity is able to manage connections between components. A *primitive* component is a component whose implementation is in a native programming language such as C++, FORTRAN, JAVA, etc. A *composite* component is a component whose implementation is an assembly of components. Hence, composite components can produce hierarchies of components.

2.2. Analysis of HPC Application Requirements

Two features of classical component models are of importance for scientific applications.

- the black box behavior of components eases their re-use;
- the separation of concern between functional codes (e.g. a numerical method) and non functional codes (e.g. a load balancer) is well suited for their development by domain experts.

In addition, scientific applications also require the four following HPC-related features: performance, adaptability and support for long-lifespan. A first requirement is *performance*. Overhead coming from the programming model has to be as low as possible. For example, a method invocation between two locally connected components should be the same as a native method invocation, provided the two components are using the same programming language.

A second important requirement is that HPC-oriented component models should be able to *exploit hardware* such as massive parallelism, GPUs, or high performance networks. Therefore, they should support adequate parallel programming paradigms such as collective communications, $M \times N$ invocations, and algorithmic skeletons [17]. This support should come without incurring overhead such as useless memory copy.

A third requirement is to support the long lifespan of HPC applications. Hardware evolution is hard to anticipate when designing an application. To solve this issue, component models should let application development be as independent as possible from execution resources. This constraint applies to primitive and composite components.

Legacy implementations of parallel programming paradigms are able to take into account some of these constraints with the help of parameters—for example, the number of processes for an MPI application. Such parameters have to be set when execution resources are known.

However, having one single integer parameter is not always enough. When porting an application from one parallel architecture to another, such as a grid, some parts of the structure of the application also need to be changed to adapt the implementation of the used programming paradigms. For example, porting an MPI application to a grid may require to add several communicators to differentiate local low latency communications from global high latency communications [23].

Thus, the main point to evaluate the relevance of component models for scientific applications appears to be how a component model supports parallel programming paradigms that abstract resources.

Classical component models fail to provide an adequate basis for HPC applications as they are unable to abstract parallel paradigms. An application developer has to manually implement them with the provided concepts—usually remote method invocation with some light support for group communications. Therefore, the next section deals with component models specialized for HPC applications.

3. On the Support of Parallel Paradigms in Specialized Component Models

This section analyzes the support of some common parallel paradigms in component models: parallel components (SPMD), inter parallel component communications ($M \times N$), collective communications, data sharing, workflow and algorithmic skeletons (in particular the task-farm skeleton).

3.1. SPMD in Component Models

A very common and useful feature for scientific applications is the support of parallel components as they can handle the Single Program Multiple Data (SPMD) paradigm. A parallel component has a cardinality that represents the number of its instances, each of which is assumed to be allocated to a distinct (logical) processor. The instances usually communicate with an external library that offers message passing and collective communications. The number of processors to use for each parallel component as well as their placement depends heavily on resources. Similarly, collective communication algorithms as well as communication between components also depend on resources. Let us analyze four component model examples that provide SPMD support.

3.1.1. Common Component Architecture (CCA)

Parallel components belong to the core of CCA [4] which distinguishes the concepts of Single Component Multiple Data (SCMD) and Multiple Component Multiple Data (MCMD). However, by default, CCA only deals with local interactions. It means that parallel components in CCA are assumed to use an external mechanism such as MPI to manage communications that are internal to a parallel component.

The SCMD approach distributes all instances of parallel components on the same set of processors and by default replicates connections on every processor as shown in Figure 1. On the opposite, the MCMD approach enables to distribute each parallel computing component on a distinct set of processors and to make use of a coupling component distributed over all processors. For example, computing components can be connected to a coupling component as illustrated in Figure 2.

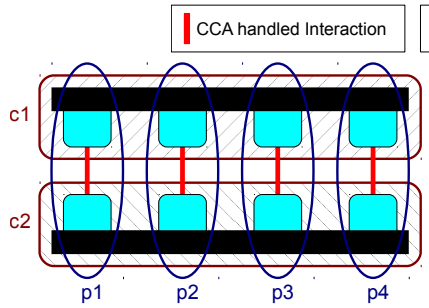


Figure 1. SCMD coupling of two parallel components (cohorts) `c1` and `c2` on four processors `p1-p4`.

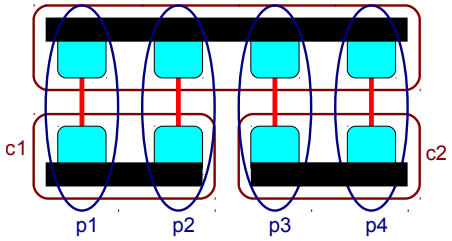


Figure 2. MCMD coupling of two parallel components (cohorts) `c1` and `c2` on two processors each.

At runtime, parallelism is fully managed by the selected communication library, usually MPI. Moreover, this library is used to deploy components. CCA runtime is only used for local interactions. More recently, the support of remote method invocation has been added to CCA [25]. As far as we know, such a feature is not used within a parallel component. However, it is very useful to handle communication between MCMD components without the need for a coupling component.

The CCA approach is interesting because it simplifies porting SPMD application to CCA. Parallel code coupling in CCA enables to hide complex issues such as language interoperability (thanks to Babel). However, by default, CCA does not handle data redis-

tribution issues between parallel components. The MCMD approach enables to delegate such a redistribution to a dedicated component, even though such a component is specific to a coupling. But, the problem of controlling the mapping of components on processors in function of the resource structure is left to application developers.

3.1.2. SCIRun2

SCIRun2 [37] is an implementation of CCA that extends the model by supporting parallel method invocations ($M \times N$) between parallel components. To benefit from this extension, one has to specify which ports of the components are parallel. Parallel ports have to be connected to another parallel port. Hence, it is not possible to connect a parallel `provides` port to a sequential `uses` port and vice-versa.

To make use of a parallel `uses` port, all the members of a parallel component have to perform a collective call to a method of such a port. Each member sends a part of the data to its associated member of the parallel component that provides the service. The associated member is the component instance with the same rank than the sender in the destination parallel component (modulo its cardinality) as shown in Figure 3. Thus, the actual data redistribution is the burden of application developer.

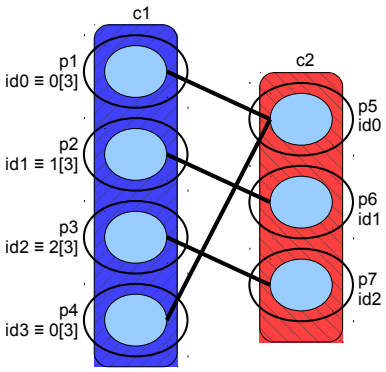


Figure 3. $M \times N$ coupling of parallel components in SCIRun2.

As an extension to CCA, SCIRun2 inherits its main benefits and drawbacks. It eases the coupling of parallel code by enabling the description of parallel interactions. However, data redistribution remains the responsibility of functional components. SCIRun2 does not simplify a lot component development or reuse because there is very limited support to take into account all data redistribution issues.

3.1.3. GridCCM

GridCCM [32] extends CCM with support for parallel components and for data redistribution in $M \times N$ method invocations. A GridCCM parallel component is made of two parts as shown in Figure 4: a data distribution descriptor and an SPMD code implementation whose cardinality is controlled by an attribute. Similarly to CCA, instances of a parallel component need to use a mechanism outside GridCCM for communications inside a parallel component. It is often a message passing library such as MPI.

Parallel components can interact with sequential or parallel components through `uses` or `provides` ports. Internally, GridCCM manages two interfaces for each par-

allel port: one to interact with sequential components and one to interact with parallel components. Interfaces differ because parallel interactions need more meta-data such as data-redistribution information. Implicit $1 \times N$ or $N \times 1$ data distributions –derived from the general data redistribution support of GRIDCCM– enable a transparent binding between parallel and sequential ports.

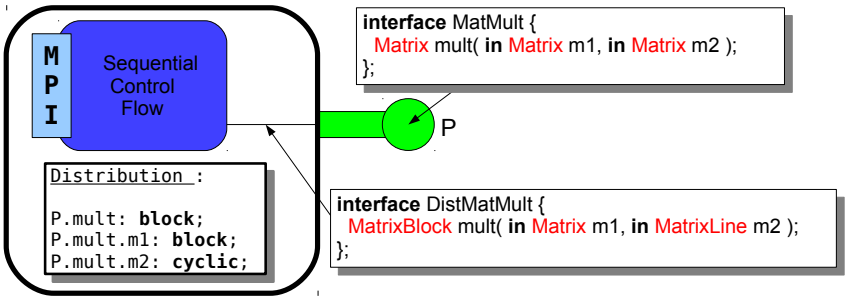


Figure 4. Parallel implementation of a component in GRIDCCM.

During $M \times N$ communications, distinct data redistributions may be required for the parameters and for the return value of a method invocation. Such redistributions are handled by parallel stubs and skeletons that are generated by a GRIDCCM compiler. When connecting components, internal introspection mechanisms are used to configure them with information such as the number of `uses` and `provides` component instances.

With respect to CCA, GRIDCCM offers a higher level of abstraction that maintains the properties of `uses` and `provides` ports for parallel components and hides most of the data distribution issues. However, as for CCA/SCIRUN2, parallel GRIDCCM components are restricted to resources natively supported by an external communication library and by data distribution libraries. Moreover, there is a hidden dependency that the same communication library implementation has to be used by all parallel components.

3.1.4. Replicating Components

Replicating components have been proposed as an extension to CCM [10]. They aim to provide a clear definition for parallel components—a set of component instances—and to intra-parallel component communications. It is based on three concepts: replicating component, `AnyToAny` connection and collective communication component.

In [10], a replicating component is defined as an algorithmic skeleton whose content is another component and that has a parameter: the number of replicas. When instantiated, its content is replicated into replicas. Replicas can be interconnected with `AnyToAny` connections, which interconnects a set of `AnyToAny` ports. An `AnyToAny` port is made of two object interfaces of the same type: one that is provided—similarly to a `provides` port— and one that is used—similarly to a multiple `uses`. Thus, all components taking part to an `AnyToAny` are fully interconnected.

A collective communication component can be used to provide a higher abstraction than just the point-to-point abstraction offered by an `AnyToAny` port. For example, a collective communication can provide a port whose interface is derived from MPI.

An assembly with replicating component is abstract: it needs to be transformed into a concrete assembly once the number of replicas is known as shown in Figure 5. Repli-

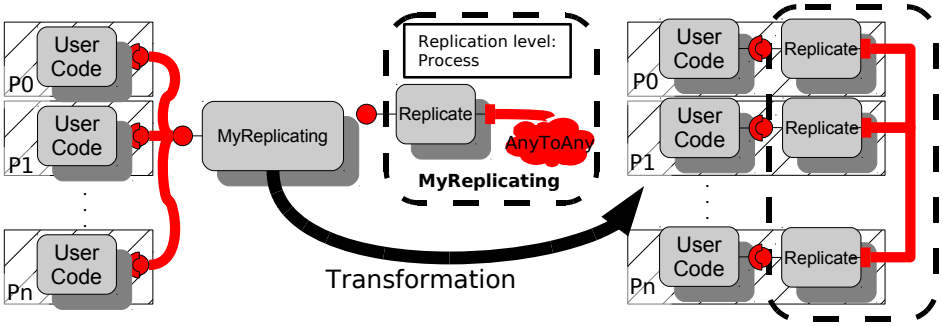


Figure 5. Example of transformation of an assembly with a replicating component.

cating components are replaced by an assembly of replicated component instances and `AnyToAny` connections are replaced by classical `uses` and `provides` connections. Hence, the resulting concrete assembly can be deployed by standard CCM tools.

By allowing to choose run-time parameters (the number of replicas) for the assembly, this approach offers a finer control than GRIDCCM or SCIRUN2. However, parallel interactions between replicated components are not considered by this approach such as with plain CCA.

In conclusion, this work enables to express collective communications within a parallel component. It is then easier to select the best communication interactions with respect to a given infrastructure. Moreover, it also enables to define hierarchical parallel components, that is to say collective communications spread over several composites.

3.2. Data Sharing Between Components

A second feature that may be useful to scientific applications and that has been integrated into component models is the sharing of data between components. Data sharing enables a set of components to access a common piece of data, either in read-only or in read-write mode. This is particularly useful for large data which requires highly irregular accesses. The implementation of such a paradigm highly depends on the component placement: sharing memory is easy between components located inside the same process, while a middleware is required for achieving this on grids [6].

3.2.1. Data Sharing Ports

Data sharing between components has been proposed as an extension to CCM [7] and to CCA and FRACTAL [8]. This extension introduced two new kinds of ports: `share` and `access` ports that respectively enable to share a piece of data and to access such a shared piece of data. An `access` port can be connected to a unique `share` port.

`share` or `access` ports are visible from the component implementation through dedicated interfaces, respectively `SharePort` and `AccessPort`, illustrated in Figure 6 and Figure 7. These interfaces offer both access methods, to obtain a local memory address to the shared data, and consistency-related methods, such as acquiring or releasing a lock.

Components making use of data sharing ports need to be compiled with a dedicated compiler. Similarly to GRIDCCM, such a compiler generates stubs and skeletons to pro-

```
interface AccessPort {
  void* get_pointer();
  long  get_size();
  void  acquire();
  void  acquire_read();
  void  release();
}
```

Figure 6. Interface offered by an `access` port.

```
interface SharePort : AccessPort {
  void set_pointer( void* data,
                    long size );
}
```

Figure 7. Interface offered by a `share` port.

vide the implementation of `SharePort` and `AccessPort` interfaces with respect to the actual shared data types that are specified in an extended OMG IDL for the CCM implementation. Stubs and skeletons are generated to use a particular shared memory interface: it can directly be the process memory for components inside the same process, a shared memory segment if components are on the same node, a distributed shared memory (DSM) for clusters or a shared data service such as JUXMEM for grids.

By mapping the memory of a component to shared data, ports such as `share` and `access` offer a simple implementation of the data sharing paradigm. However, they require a dedicated compiler and the employed middleware needs to be specified at compile time.

3.3. Workflow and Component Models

Adding dataflow and/or workflow capabilities to a spatial component model enables to support both spatial and temporal compositions within a unique and coherent model. Several works have been pursued into that direction. For example, SCA enables a primitive component implementation to be a workflow (BPEL) but only spatial composition is possible between components [30]. On the contrary, several workflow models enable a spatial description for a task implementation such as Triana [36] or ASKALON AGWL [19]. But, spatial composition is not possible between tasks or between sub-workflows. The remainder of this section presents STCM which does not have these limitations.

3.3.1. Spatio-Temporal Component Model (STCM)

STCM [13] extends the Grid Component Model (GCM) [9] with workflow concepts borrowed from the ASKALON Grid Workflow language (AGWL) [19]. It defines task-components as well as a new kind of port, the input/output ports, which are used to describe data dependencies between tasks.

The assembly description language supports both spatial and temporal compositions. It is derived from the FRACTAL description language with new constructs to describe temporal relationships between components inspired from AGWL. An example of an STCM application is given in Figure 8.

In STCM, a task-component has only one task. A task-component differs from a component as it has to implement a specific interface—`TaskControler`—that contains the prototype of the method to execute when a task has to be executed by the component. Such a method can access input data through its input ports and it can set its results by setting the output ports.

A task-component has an extended life-cycle whose state diagram contains a new state—`running`—to represent a task-component whose task is currently being executed, as shown in Figure 9.

```
<component name ="exApp">
 <input name="vI"  type="Vect"/>
 <body><component name="Init">
   <input  name="ii1" ... set="vI"/>
   <output name="io1" ... />
   <output name="io2" type="double" />
  </component>
  <component name="A">
   <input  name="a" ... set="ci.io1"/>
   <client name="pA" ... set="B.pB"/>
  </component>
  <component name="B">
  // in: double b, out: double outB
  <server name="pB" type="GetRes"/>
 </component>
 <sequence name="seq1">
  <instance name="ci" compRef="Init">
    // block is on the right
  </sequence></body>
</component>
```

```
<parallel name="ParallelCtrl">
 <instance name="a" compRef="A"/>
 <instance name="b" compRef="B"/>
 <section>
  <exectask nameInstance="a"/>
 </section>
 <section>
  <while name="LoopCtrl">
   <input name="c" type="double"
           set="ci.io2"
           loopSet="B.outB"/>
   <condition> c<0.1 </condition>
   <loopBody>
    <exectask nameInstance="b">
     <input name="b" set="c"/>
    </component>
   </loopBody>
  </while>
 </section>
</parallel>
```

Figure 8. Main elements of an application description in STCM: Instance `a` is executed in parallel of multiple executions of Instance `b` (while loop); they can communicate through the connection of the ports `pA` and `pB`.

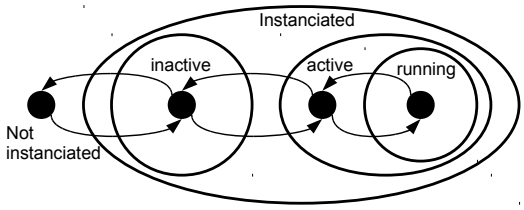


Figure 9. State diagram describing the life-cycle of a task-component in STCM.

A dedicated workflow engine has been developed to execute an STCM assembly: it instantiates components, launches tasks and manages data transfers. The instantiation of a task-component can be delayed until either the flow of control reaches it, or one of its provide ports is used.

STCM enables both the description of spatial and temporal dependencies between primitive and composite components. Its runtime is more complex since it has to contain a specialized workflow engine. However, STCM does not support parallel components nor data sharing.

3.4. Algorithmic Skeletons and Task-Farm in Component Models

Algorithmic skeletons [17] can be defined in component models as parameterizable patterns for composites. Skeletons usually have a first set of parameters to specify their behavior and a second set of parameters that adapt them to resources. A replicating component is an example of an algorithmic skeleton. This section focuses on the support of the task-farm skeleton in component model as it is a very common skeleton in HPC applications. Then it deals with more general support for skeletons.

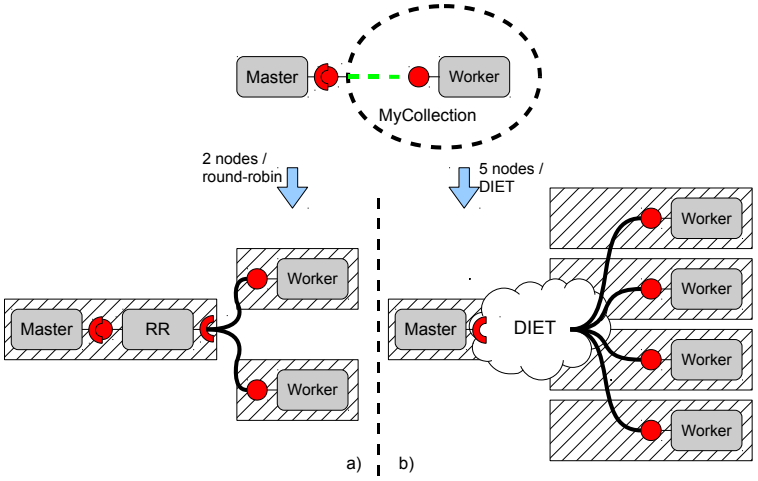


Figure 10. Examples of collection transformation based on a round-robin policy (a) and on a DIET policy (b).

3.4.1. Collection Based Task-Farm Skeleton

An implementation of the task-farm skeleton (also known as master-worker relationships) has been proposed as an extension to CCM, FRACTAL and CCA [14]. This extension introduced the concept of collection to describe a task-farm content as an assembly of components.

A collection has two parts: a component assembly to replicate (the worker part) and an association of worker ports to the collection's ports (the dispatch and collect part). There are two strategies for such associations: i) all worker ports are accessible through the collection's ports or ii) a single collection port aggregates the worker ports. In the latter case, a mechanism is needed to dispatch an invocation of a collection port to a worker port: the request transport policy. Such a policy could be implemented with components or with an existing middleware such as DIET [16].

Being abstract, a collection needs to be concretized to be deployed. This step typically occurs at deployment time when resources are known. Two parameters have to be set: the initial number of worker instances and a request transport policy whose instantiation will actually connect collection's ports to workers' ports.

Although this approach is limited to the task-farm skeleton, it is interesting as it enables the use of farms of tasks in component models without modifying the components at runtime. It is easy to use since it maintains classical use/provide interactions for collections. Eventually, by delaying transformation to deployment, an application can be adapted to available resources. This approach has been studied in [5], however, automatically making efficient parameter choices was not considered in the general case.

3.4.2. Spatio-Temporal-Skeleton Component Model (STKM)

A more general approach to provide algorithmic skeletons in component models has been studied in STKM [2]. STKM extends STCM with (behavioral) skeletons [3] such as pipeline or farm. Such skeletons can be used as regular components in an assembly. Data stream ports are also defined to enable interactions between skeletons.

A skeleton in STKM offers an external interface which is defined by a set of ports (similarly to regular components) and an internal interface which presents parameterization points. When instantiating a skeleton, each of the parameterization points needs to be fulfilled, for example by a component type, and its ports need to be connected. Ports of a component used to fulfill a parameterization point can also be connected to components outside the skeleton. Figure 11 illustrates the task-farm skeleton.

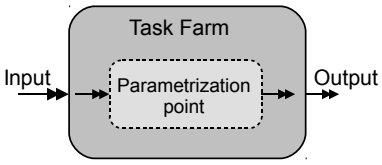


Figure 11. Task-farm example in STKM that exposes two data stream ports and a parameterization point.

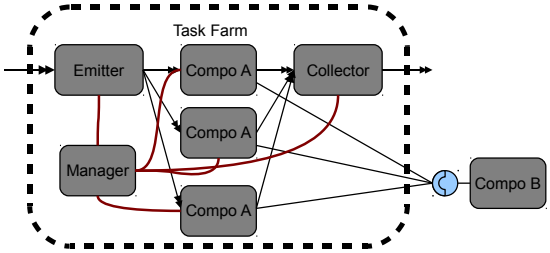


Figure 12. Deployed task-farm with 3 workers (Compo A).

As for collections, an assembly with skeletons is abstract. To be made concrete, a skeleton needs to be replaced by its implementation, usually a composite. The content of such a composite can therefore change in function of execution resources. For behavioral skeletons, managers aiming to deal with adaptation issues belong to this composite. An example of a deployed task-farm is given in Figure 12.

STKM extends STCM with behavioral algorithmic skeletons based on data streams. This behavior is closer to classical skeleton models than the model proposed by the collection based approach. STKM abstracts a programmer from many issues and enables skeletons to be transparently adapted to available resources. However, the number of skeletons in STKM is fixed and there is not any mechanism for a user to specialize their implementation. Moreover, STKM does not support other HPC features.

3.5. Discussion

This section has presented various works that aimed to extend component models with features generally required for HPC applications. There is no single component model that supports all of them. These features have various impacts on component models, which can be classified along two axis:

1. Some features introduce new kinds of interactions between components while some introduce new kinds of component implementations;
2. Some features are static—they only impact the assembly used for the deployment—while some are dynamic—they impact the behavior of a component assembly during runtime.

Hence, Table 1 classifies the studied extensions based on these two axis. Except for data sharing, each feature relies on the introduction of several concepts which belong to both categories—new interactions and new implementations.

Most of the studied features are static with the exception of dataflow and workflow that are dynamic, as they impact components' life-cycles. However, algorithmic skeletons may also have a dynamic part if they provide adaptability capability such as behavioral skeletons.

Table 1. New interactions and implementations of component models coming from HPC extensions.

	Interaction	Implementation	Static†/Dynamic
SPMD	$M \times N$ method invocation between parallel components, message passing, collective communication	parallel components	static†
Data sharing	data sharing	-	static†
Workflow and dataflow	control flow, data flow	task-component	dynamic
Task-farm	method invocation with load-balancing	task-farm	static†
Algorithm Skeletons	data stream	generic composites	static†or dynamic*

†: Dynamic adaptation is not considered.
*: Dynamic adaptation based is considered through behavioral skeletons.

3.6. Conclusion

Various HPC related features have already been studied with respect to component models. They increase the abstraction level provided to programmers, hence easing the development of applications. Moreover, they also simplify the adaptation of components to new resources by enabling to change the parameters of a component implementation and/or by promoting a separation of functional and non-functional concerns.

Using these abstraction introduces two main drawbacks. First, as usually, when increasing the level of abstraction, performance relies on the optimization of some application external codes such as the runtime, the code generator—for GRIDCCM—and/or the assembly transformers—for algorithmic skeletons. Hence, it is important that a programmer be able to replace such piece of code for a particular application on a particular resource if needed. Second, all these extensions are a priori incompatible at the model level as well as at the implementation level. Sometimes, even extensions applied to the same component model (such as to CCA or CCM) are not compatible between them, and may introduce incompatibilities with the original component model.

The next section presents a work that aims at providing a framework to define, implement and optimize any kind of static extensions within a unique and coherent component model.

4. High Level Component Model

In this section, we first introduce HLCM, stating its purpose and providing an overview. Next, we present the model and HLCM$_i$, a proof-of-concept implementation. Finally, the usefulness of HLCM is evaluated based on how well it expresses the previously presented HPC features.

4.1. Introduction

The previous section has shown that two kinds of concept are used to support HPC features: i) new kinds of component implementations and ii) new kinds of interactions.

Component implementation kinds can be seen as various algorithmic skeletons. Hence, a mechanism that enables higher order assembly description is needed to let users define new skeletons. Such a mechanism can be based on genericity [28]. However, as the value of a generic parameter could also be used to define composite content, the model needs to support genericity through meta-programming similar to templates in C++ [1].

For the support of new kinds of interactions, a solution based on connector—introduced in architecture description languages—seems very promising. A connector exposes roles that can be fulfilled by ports. They are intrinsically generic as ports are used as parameters. However, connectors have not been previously defined in hierarchical component models. Based on this analysis, a model named *High Level Component Model* (HLCM) has been proposed [12].

4.2. HLCM Overview

HLCM is an abstract component model that supports hierarchy, genericity and connectors [12]. It is an abstract model as it does not specify the primitive elements of the model (primitive component implementations, generators and port types which are introduced hereafter); primitive elements are instead specified by *specializations* of HLCM such as HLCM/CCM. In HLCM/CCM, primitive components are CCM components and primitive connectors are the CCM connections (use/provides and events). This makes it possible to take advantage of HLCM using various underlying execution models or *backends*.

Genericity has been introduced in HLCM using the approach described in [11]. All types of the model are generic (i.e. accept other types as parameter). The implementations of these types can either implement the whole generic type or be restricted to a given set of generic parameters. HLCM supports meta-programming with constructs such as static conditionals and loops evaluated at compilation time.

The meta-model of HLCM has been described in the ECORE language of the Eclipse Modeling Platform (EMF). As for any instances of ECORE meta-models, HLCM applications can be described in the OMG XML Metadata Interchange (XMI) dialect. This syntax is however not human-friendly: examples in this section are described in a dedicated HLCM textual syntax as well as in an informal graphical syntax.

4.2.1. Model Elements

The basis of HLCM is a generic hierarchical component model with connectors. The main elements of HLCM are components, connectors, port types, bundles and connection adaptors. Components and connectors are implemented respectively by component implementations and generators.

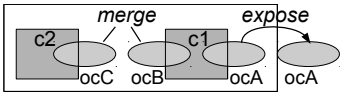
A **component** in HLCM is a black-box, locus of computation. It exposes a set of named interaction points and has one (or more) implementation(s). These points of interaction are not ports but *open connections*: an open connection is a connector instance with some roles fulfilled by the component and some roles remaining open to let the component be connected to other components.

A **connector** in HLCM represents a kind of interaction. It exposes a set of named roles and has one (or more) implementation(s). Connector instances are called **connections** and connector implementations are called **generators** [26]. Interactions in assemblies are described by **merging** two or more connections of the same type (connector). The result of a merge is a new connection whose roles are fulfilled by the union of

i) Textual representation

```
composite aCompositeImplem
implements aComponent
{
  AnotherComponent c1;
  AThirdComponent c2;
  merge ( c1.ocB, c2.ocC );
  merge ( this.ocA, c1.ocA );
}
```

ii) Expanded view (connections are not merged)



iii) Compact view (connections are merged)

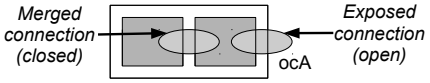


Figure 13. Three representations of a composite implementation of a component `aComponent`. It contains two internal component instances `c1` and `c2` connected through the merging of the two open connections `c1.ocB` and `c2.ocC`. It exposes the open connection `c1.ocA` as its own `ocA`.

```
generator LoggingUP<UI,PI> implements
  UseProvide< provider={Facet<PI>}, user={Receptacle<UI>} >
when ( UI super PI ) {
  LoggerComponent<UI> proxy;
  proxy.client.user += this.user;
  proxy.server.provider += this.provider;
}
```

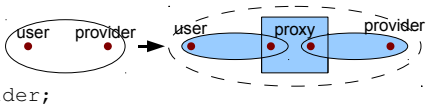


Figure 14. Example of a generator implementing the `UseProvide` connector by inserting a proxy component for logging purposes. Ports fulfilling the roles of a `UseProvide` connection are used to fulfill roles of the `client` and `server` exposed connections using the += operator.

their corresponding roles in the merged connections. Connections can be closed or open. **Closed connections** are connections that can not be merged anymore, i.e. connections internal to an assembly. **Open connections** are connections that are or can be further merged, *i.e.* connections exposed in the interface of components.

A generator can impose constraints on the generic parameters of the connector it implements (i.e. the type of the ports) as any implementation of a generic type. In addition, it can also impose constraints amongst a set specific to the specialization. This can for example be used to constraint locality of the component instances exposing the ports.

A **bundle** groups multiple connections into a single entity. **Connection adaptors** enable (open) connection polymorphism. A connection adaptor can adapt an (open) connection exposed by a component whose actual type does not match the type declared in the component interface.

4.3. Behavior of HLCM Elements

The specifications of HLCM are based on a model-driven engineering (MDE) approach [34]: the behavior of HLCM applications is based on the specification of a model transformation that puts HLCM applications in equivalence with applications of the underlying execution model. The behavior of an HLCM application is defined as being that of the equivalent application of the underlying model.

An HLCM application is defined by the set of HLCM elements it contains: components, connectors, generators, port types and connection adaptors and by the component used as the root of the application. To map it into a primitive application, which can then be executed, it should be transformed into an assembly which only contains primitive components, primitive ports, and primitive connections. In the case of the HLCM/CCM

specialization for example, this means that HLCM/CCM applications are transformed into plain CCM applications.

As a first step of the transformation, the approach described in [11] is applied, to support genericity. The rest of the transformation is straight forward: i) the implementations of the various component instances and connectors are chosen amongst the available choices; ii) in the case of composite implementations, their content is exposed so as to form a flat assembly. The process is repeated until all non-primitive elements are implemented. Since composite implementations are opened and only their content is used, ultimately all elements have primitive implementations and thus form an application of the underlying execution model.

This transformation is however non-deterministic as it does not specify how the choices of connections and implementations are made. There can therefore be multiple distinct applications of the underlying execution model in equivalence with a single HLCM application. In this case, each of these primitive applications defines a valid behavior of the HLCM application. If a transformation does not result in such a primitive application, the initial HLCM application is declared invalid.

4.4. HLCM$_i$, an Implementation of HLCM

In order to evaluate HLCM and its specializations, a set of proof-of-concept implementations of HLCM have been developed. These implementations are themselves built as assemblies of the Low Level Component Model JAVA (LLCM$_j$), a plain JAVA backend for HLCM$_i$. To solve the bootstrap problem, these assemblies are hard-coded in JAVA and do not take advantage of the HLCM assembly language. The components shared by the various HLCM implementations form a framework known as HLCM$_i$. The implementation relies on the tools provided as part of the Eclipse Modeling Framework (EMF).

In addition to the LLCM$_j$ specialization of HLCM, HLCM$_i$ supports HLCM specializations (i.e. backends) to LLCM$_{cpp}$, a `C++` variation of LLCM$_j$, CCM, and Charm++ [22]. The remaining of this section deals with some evaluations based on the CCM specialization of HLCM.

4.5. Evaluation: Defining Parallel Paradigms in HLCM

Section 3 has dealt with four types of parallel extensions to component models: SPMD with $M \times N$ and collective communications, data sharing, workflow and algorithmic skeleton (task-farm). HLCM enables meta-programming. Hence, HLCM by construction supports algorithmic skeletons. For example, it is straightforward to define a task-farm in HLCM by letting the number of instances as well as the component implementing the request transport policy be parameters of a composite.

On the contrary, HLCM is not dynamic. Hence, workflows are not directly supported in HLCM. However, it seems possible to apply the same approach as for STCM [13] to HLCM with the difficult point of handling model adaptation on the fly.

This section provides an overview on the two remaining parallel extensions: SPMD with $M \times N$ and collective communications, and data sharing. More details can be found in [12]. Let us start with data sharing.

```
component MemoryAccessor exposes {
   SharedMem<access={ LocalReceptacle<DataAccess> }> memory;
}
```

Figure 15. Declaration of a component `MemoryAccessor` exposing a connection `memory` that can be used to access the shared memory.

i) Textual representation

```
composite CompositeAccessorImpl
implements MemoryAccessor {
  MemoryAccessor c1;
  MemoryAccessor c2;
  merge(this.memory, c1.memory, c2.memory);
}
```

ii) Graphical representation

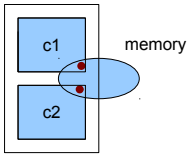


Figure 16. The `CompositeAccessorImpl` composite merges the `c1.memory` and `c2.memory` into a single connection exposed as `this.memory`.

4.5.1. Data Sharing in HLCM

The shared memory interaction has been defined in HLCM/CCM using an approach inspired by [7]. It is provided by `SharedMem`, a connector with a single role: `access`.

For accessing the shared memory from primitive components, an object interface `DataAccess` is defined: it contains methods to retrieve a pointer to the shared memory and to acquire/release read/write privileges. This interface is not aimed to be remotely accessed as it returns a local pointer to manipulate to data. As a result this interface can be correctly used only by components located in the same address space (process). To support such constraints, two primitive ports— `LocalReceptacle` and `LocalFacet`—of the HLCM/CCM backend are used. These port types behave exactly like the usual CCM `Receptacle` and `Facet` except that they impose a process collocation constraint between the involved component instances in the resulting assembly.

A component can therefore safely access the shared memory through a port of type `LocalReceptacle<DataAccess>` used to fulfill the `access` role of a `SharedMem` connection as shown in Figure 15. A composite component implementation containing multiple instances of such a component, as shown in Figure 16, can both specify that i) its internal instances access the same shared memory and ii) external members of the composite can also access the same shared memory by exposing the connection.

Two straightforward generators implementing the `SharedMem` connector have been developed. A first generator simply inserts an internal component in charge of the memory. This implementation imposes a process collocation constraint between each component instance accessing the connector and the `store` instance, effectively requiring all accessors to be in the same process.

A second generator supports distributed shared memory by inserting for each accessor an instance of a component delegating the calls to JuxMem [6], a distributed shared memory implementation.

```
bundle PFacet<Integer N, interface I> {
  each(i:[1..N]) { UseProvide<provider={ Facet<I> }> part[i]; }
}
```

Figure 17. Definition of the `PFacet` bundle port type. It contains `N` `UseProvide` connections named `part` whose each `provider` roles has to be fulfilled by a `Facet<I>` port.

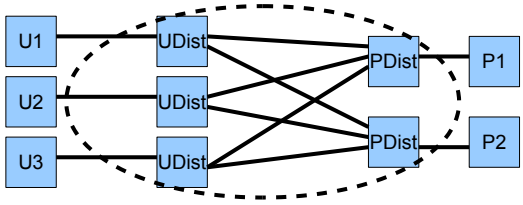


Figure 18. A parallel `UseProvide` connection implemented by the `MxN` generator. Distributor component instances (`Udist` and `Pdist`) are inserted for each participant on the user and provider sides. These instances are connected to all those of the opposite side.

4.5.2. Parallel Method Invocations in HLCM

Unlike shared memory, the support for parallel method invocations [32] does not require the introduction of a new connector: the `UseProvide` connector already supports method invocations. Two bundles are however introduced: `PFacet` whose definition is presented in Figure 17 and the symmetrical `PReceptacle`. We make the choice that supporting parallel method invocations means supporting the case where these bundles fulfill the `user` or `provider` roles of a `UseProvide` connection.

A `MxN` generator that implements `UseProvide` connections whose roles are fulfilled by a `PFacet` and a `PReceptacle` has been implemented. An example of connection implemented by this generator is presented in Figure 18. This enables an efficient support of $M \times N$ connections—as shown in Section 4.6—with data redistribution management on the user side, the provider side or even both.

The support for `UseProvide` connections with only one of the roles filled by a parallel port and the other filled by a sequential port is supported by two connection adaptors. The `Scatter` adaptor whose definition is presented in Figure 4.5.2 supports a connection whose user role is filled by a `PReceptacle` as if they were filled by a sequential `Receptacle`. It contains a component (`Distributor`) in charge of distributing the data connected to all the `part` sub-connections of the bundle and exposing an open connection with a sequential `Receptacle` used as result of the adaptation. A `Gather` adaptor supports the symetrical case for a provider role.

4.6. Analysis

Both shared memory and parallel user/provide connections have been used to compile versions of synthetic applications with a degree of parallelism varying from 1 to 1000 components. The compilation time varies between three and seven seconds on a standard laptop. This is acceptable when compared to the 50 seconds required to compile the primitive components of the examples on the same machine or the deployment time of a distributed CCM application that is usually in tens of seconds on Grid'5000, a distributed testbed.

```
adaptor  Scatter<Integer N>
supports UseProvide<user=
   {PReceptacle<N, MatrixPart>}>
as UseProvide<
    user={Receptacle<Matrix>}>
{   Distributor<N> dist;
    each(i:[1..N]){
       merge(dist.in[i],
           supported.user.part[i]);
    }
    merge(this, dist.out);
}
```

Figure 19. Definition of the `Scatter` adaptor.

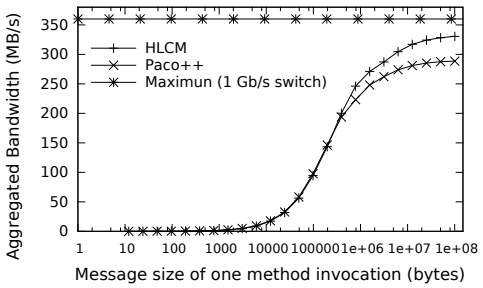


Figure 20. HLCM-based implementation of a 3x4 parallel method invocation vs PACO++, a dedicated environment, on a 1 Gb/s switched cluster.

In the case of shared memory interactions, we have seen that HLCM makes it possible to expose a single open connection whether the component is sequential or a composite containing multiple instances participating in the connection as shown in Figure 16.

Similarly, for parallel method invocation interactions, a component exposing an `UseProvide<user={Receptacle<Matrix>}>` open connection can be sequential or parallel thanks to the mechanism of connection adaptors. In the latter case, a parallel implementation shall expose an open connection whose actual type is `UseProvide<user={PReceptacle<N, MatrixPart>}>`. Thus, a component sequentializing the interaction is automatically inserted only if required by an adaptor.

With respect to performance, there is a unique connection involving (directly after transformation) all the participants. Hence, it is possible to choose the best implementation of the connection with no bottleneck. For example, if both components involved in a method invocation are parallel, the `MxN` generator is used rather than using the two connection adaptors that would sequentialize the interactions.

Therefore, the achievable performance are good as shown in Figure 20. This figure compares HLCM/CCM to PACO++ [33], a dedicated environment efficiently handling $M \times N$ communication used within GRIDCCM. This experiment has been conducted on a cluster of the Grid'5000 French experimental platform. As can be seen, the HLCM version achieves performance comparable to the PACO++ version.

5. Conclusion

To face the increase in complexity of both hardware and software, it is important to propose programming abstraction that will ease development without impacting performance. Software component models have been studied a lot in non-HPC fields. They appear very promising, provided they are able to deal with HPC features. Therefore, several works have proposed component models with some specific extensions towards HPC applications, which includes parallel component, $M \times N$ communications, data sharing, algorithmic skeletons, etc. They confirm the relevance of component models for handling code composition in HPC applications. However, none of them provides all the desired features at the same time.

High Level Component Model (HLCM) is an attempt to coherently combine into a unique model all these features. To be more accurate, it provides concepts such as

components, connectors, genericity, open connections, bundle and connection adaptors that enable to deal with adaptable HPC features. While it should be able to deal with dynamic applications, such as workflows, it has yet to be proved.

Several issues still need to be resolved. First, while HLCM offers several choices of component and connector implementation, algorithms that will automatically select the best choices with respect to a given optimization criteria have yet to be developed. Second, component models need to be validated not only for handling application dynamicity (such as workflows) but also resource dynamicity. For example, an application could reconfigure itself to adapt to resource failure. Last, component models currently have a quite simple relationship with primitive programming languages as primitive components are mainly assumed to be sequential. Hence, relationships between component models and parallel languages, such as OpenMP, need further research to understand how to efficiently compose codes written with distinct parallel languages.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] M. Aldinucci, H. Bouziane, M. Danelutto, and C. Pérez. Towards Software Component Assembly Language Enhanced with Workflows and Skeletons. In *Joint Workshop on Component-Based High Performance Computing and Component-Based Software Engineering and Software Architecture (CBHPC/COMPARCH 2008)*, 14-17 October 2008.

[3] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonellotto. Behavioural Skeletons in GCM: Autonomic Management of Grid Components. In *Proc. of Intl. Euromicro Parallel Distributed and network-based Processing (PDP)*, pages 54–63, Toulouse, France, feb 2008. IEEE.

[4] B. A. Allan, R. Armstrong, D. E. Bernholdt, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A Component Architecture for High-Performance Scientific Computing. *International Journal of High Performance Computing Applications*, 20(2):163–202, 2006.

[5] F. André, G. Gauvrit, and C. Pérez. Dynamic Adaptation of the Master-Worker Paradigm. In *IEEE 9th International Conference on Computer and Information Technology*, Xiamen Chine, October 2009.

[6] G. Antoniu, L. Bougé, and M. Jan. JUXMEM: An Adaptive Supportive Platform for Data Sharing on the Grid. *Scalable Computing: Practice and Experience*, 6:45–55, Nov. 2005.

[7] G. Antoniu, H. L. Bouziane, L. Breuil, M. Jan, and C. Pérez. Enabling Transparent Data Sharing in Component Models. In *6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 430–433, Singapore, May 2006.

[8] G. Antoniu, H. L. Bouziane, M. Jan, C. Pérez, and T. Priol. Combining Data Sharing with the Master–Worker Paradigm in the Common Component Architecture. *Cluster Computing*, 10(3):265 – 276, 2007.

[9] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. GCM: A Grid Extension to FRACTAL for Autonomous Distributed Components. *Special Issue of Annals of Telecommunications: Software Components – The Fractal Initiative*, 64(1):5, 2009.

[10] J. Bigot and C. Pérez. Enabling Collective Communications between Components. In *CompFrame '07: Proceedings of the 2007 symposium on Component and framework technology in high-performance and scientific computing*, pages 121–130, New York, NY, USA, 2007. ACM Press.

[11] J. Bigot and C. Pérez. Increasing Reuse in Component Models through Genericity. In *Proceedings of the 11th International Conference on Software Reuse, ICSR '09*, LNCS, pages 21–30, Falls Church, Virginia, USA, Oct. 2009. Springer-Verlag.

[12] J. Bigot and C. Pérez. Enabling Connectors in Hierarchical Component Models. Research Report RR-7204, INRIA, August 2010.

[13] H. L. Bouziane, C. Pérez, and T. Priol. A Software Component Model with Spatial and Temporal Compositions for Grid Infrastructures. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 698–708, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] H. L. Bouziane, C. Pérez, and T. Priol. Extending Software Component Models with the Master-Worker Paradigm. *Parallel Computing*, 2010.

[15] E. Bruneton, T. Coupaye, and J-B. Stefani. *The Fractal Component Model, version 2.0.3 draft*. The ObjectWeb Consortium, Feb. 2004.

[16] E. Caron and F. Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *Int. J. of High Perf. Comp. Applications*, 20(3):335–352, 2006.

[17] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[18] EJB 3.0 Expert Group. *JSR 220 : Enterprise JavaBeans Version 3.0*, Dec. 2005.

[19] T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and Grid 2005 (CCGrid 2005)*, volume 2, pages 676–685, Cardiff, UK, May 2005.

[20] S. Gorlatch and J. Dünnweber. From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In *Future Generation Grids*. Springer Verlag, 2005.

[21] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *The MPI-2 Extensions*, volume 2 of *MPI: The Complete Reference*. The MIT Press, Cambridge, MA, USA, Sep. 1998.

[22] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[23] N. Karonis, B. Toonen, and I. Foster. Mpich-G2: A Grid-enabled Implementation of the Message Passing interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, 2003.

[24] The Khronos Group. *The OpenCL Specification*, sep 2010.

[25] G. Kumfert, J. Leek, and T. Epperly. Babel Remote Method Invocation. In *IEEE Intl Parallel and Distributed Processing Symposium*, pages 1–10, Long Beach, CA, March 2007.

[26] S. Matougui and A. Beugnard. Two Ways of Implementing Software Connections Among Distributed Components. In *OTM Conferences (2)*, pages 997–1014, 2005.

[27] D. McIlroy. Mass-Produced Software Components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 138–150, 1968.

[28] D. R. Musser and A. A. Stepanov. Generic programming. In Proc. of the Intl. Symposium on Symbolic and Algebraic Computation (ISSAC'88), pages 13–25, London, UK, 1989. Springer-Verlag.

[29] Object Management group. *Common Object Request Broker Architecture Specification, Version 3.1, Part 3:* CORBA *Component Model*, Jan. 2008.

[30] Open Service Oriented Architecture. *SCA Service Component Architecture: Assembly Model Specification Version 1.00*, Mar. 2007.

[31] OpenMP Architecture Review Board. *OpenMP Application Program Interface version 3.0*, May 2008. Available at http://www.openmp.org/mp-documents/spec30.pdf.

[32] C. Pérez, T. Priol, and A. Ribes. A Parallel CORBA Component Model for Numerical Code Coupling. In M. Parashar, editor, *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, volume 17 of *Lect. Notes in Comp. Science*, pages 88–99, Baltimore, Maryland, Nov. 2002. Springer-Verlag. Special issue Best Applications Papers from the 3rd Intl. Workshop on Grid Computing.

[33] C. Pérez, T. Priol, and A. Ribes. PaCO++: A Parallel Object Model for High Performance Distributed Systems. In *Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of the 37th Hawaii International Conference on System Sciences (HICSS-37)*, Big Island, Hawaii, USA, January 2004. IEEE Computer Society Press.

[34] D. C. Schmidt. Guest editor's Introduction: Model-driven Engineering. *Computer*, 39(2):25–31, 2006.

[35] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2 edition, 2002.

[36] I. Taylor, M. Shields, I. Wang, and A. Harrison. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, September 2005.

[37] K. Zhang, K. Damevski, V. Venkatachalapathy, and S. G. Parker. SCIRUN2: A CCA Framework for High Performance Computing. Intl. Workshop on *High-Level Programming Models and Supportive Environments*, 0:72–79, 2004.