Highly Scalable Distributed Component Framework for Scientific Computing

Kostadin Damevski, Ashwin Deepak Swaminathan, and Steven Parker

Scientific Computing and Imaging Institute
University of Utah, Salt Lake City, Utah 84112, USA
{damevski,ashwinds,sparker}@sci.utah.edu

Abstract. As scientific computing experiences continuous growth of the size of simulations, component frameworks intended for scientific computing need to handle more components and execute on numerous hardware resources simultaneously. Our distributed component framework CCALoop presents a novel design that supports scalability in both number of components in the system and distributed computing resources. CCALoop also presents several other beneficial design principles for distributed component frameworks such as fault-tolerance, parallel components, and support for multiple users. To provide scalability it distributes the responsibility for data queries and updates equally to all nodes in the system through a distributed hash table mechanism, while providing low latency in these operation through a method that guarantees one-hop routing of framework messages.

1 Introduction

In recent years, component technology has been a successful methodology for large-scale commercial software development. Component technology encapsulates a set of frequently used functions into a component and makes the implementation transparent to the users. Application developers typically use a group of components, connecting them to create an executable application. The components are managed by a component framework that exists on each computing node where components may be instantiated or executed (see Figure 1). A component framework provides a set of services to components: locating other components in the system, instantiating, connecting, executing, reporting error messages or results, etc. It can also provide a user interface, often a Graphical User Interface (GUI), to compose, execute and monitor components. In order to manage a large component application that uses many components and utilizes sets of distributed computing resources, one or more component frameworks have to exist on each separate computing resource. This requires that multiple frameworks cooperate in some fashion to manage and monitor a large component application.

Component technology is becoming increasingly popular for large-scale scientific computing in helping to tame the software complexity required in coupling multiple disciplines, multiple scales, and/or multiple physical phenomena. The

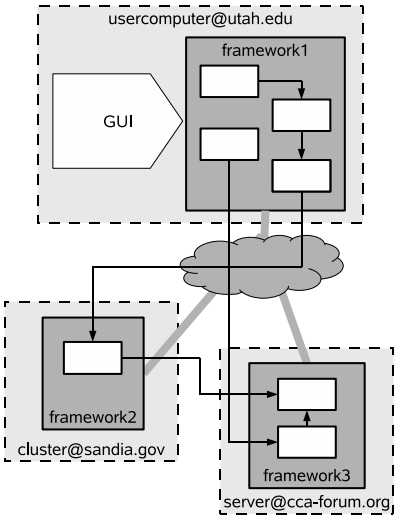


Fig. 1. Three distributed framework nodes located on separate computing resources.

Common Component Architecture (CCA) [1] is a component model that was designed to fit the needs of the scientific computing community by imposing low overhead and supporting parallel components. The CCA standard also provides for the inclusion of Single Program Multiple Data (SPMD) parallel components. These components exists in several address spaces and are internally managed by message passing (e.g. Message Passing Interface (MPI)). A compliant framework needs to provide the facilities to instantiate, manage, and execute this novel type of component. The CCA model selects a lightweight component framework that optimizes execution efficiency. Several software frameworks are targeted at the CCA component model, including SCIRun2 [2], CCAFFEINE [3], XCAT [4] and others. These systems enable creation of complex high-performance simulations through the assembly of software components. Component frameworks aimed at scientific computing need to support a growing trend in this domain toward larger simulations that produce more encompassing and accurate results. The CCA component model has already been used in several domains, creating components for large simulations involving accelerator design, climate modeling, combustion, and accidental fires and explosions [5]. These simulations are often targeted to execute on sets of distributed memory machines spanning several computational and organizational domains [6]. To address this computational paradigm a collection of component frameworks that are able to cooperate to manage a large, long-running scientific simulation containing many components are necessary.

In each of the CCA-compliant component frameworks, facilities are provided to support the collaboration among several distributed component frameworks.

However, existing designs do not scale to larger applications and multiple computing resources. This is due to a master-slave (server-client) communication paradigm. In these systems, the master framework manages all the components and their metadata while also handling communicating with the user through the GUI. The slave frameworks act only as component containers that are completely controlled by the master. This centralized design is simple to implement and its behavior is easy to predict. As component and node numbers grow however, the master framework is quickly overburdened with managing large quantities of data and the execution time of the entire application is affected by this bottleneck. Moreover, as simulation size grows even further, the master-slave design can inhibit the efficiency of future scientific computing applications. We present an alternative design that is highly scalable and retains its performance under high loads. In addition to the scalability problem, the master framework presents a single point of failure that presents an additional liability for long-running applications and simulations.

A component framework's data is queried and modified by a user (through a GUI) and by executing components. In both cases it is imperative that the framework provides quick responses under heavy loads and high-availability to long running applications. The goal of our work is to present a solution to several key issues is distributed component framework design. The system described in this paper is architected to: (1) scale to a large number of nodes and components, (2) maintain framework availability when framework nodes are joining and leaving the system and be able to handle complete node failures, (3) facilitate multiple human users of the framework, (4) support the execution and instantiation of SPMD parallel components. Our distributed component framework, CCALoop, is self-organizing and uses an approach that partitions the load of managing the components to all of the participating distributed frameworks. The responsibility for managing framework data is divided among framework nodes by using a technique called Distributed Hash Tables (DHT) [7]. CCALoop uses a hash function available at each framework node that maps a specific component type to a framework node in a randomly distributed fashion. This operation of mapping each component to a node is equally available at all nodes in the system. Framework queries or commands require only one-hop routing in CCALoop. To provide one-hop lookup of framework data we keep perfect information about other nodes in the system, all the while allowing a moderate node joining/leaving schedule and not impacting scalability. We accommodate the possibility that a framework node may fail or otherwise leave the system by creating redundant information and replicating this information onto other frameworks.

We organize the discussion of our distributed component design as follows. Section 2 contains a discussion of related work. In Section 3 we show a detailed view of our design and implementation of the CCALoop distributed component framework, while in Section 4 we discuss some of the preliminary benchmarks we have taken of our system. Finally, we finish with conclusions and future work of the project in Section 5.

2 Related Work

The work of this paper is largely motivated by the lack of attention to scalability in current CCA framework implementations. The SCIRun2 [2] and XCAT [4] CCA-compliant frameworks enable distributed components existing on a variety of resources. Both of them follow a paradigm of a master framework connected to a number of slave frameworks and, as we mentioned before, this design does not support large simulations and further growth. The CCAFFEINE [3] framework is different in that it is not distributed, but only a parallel framework. It communicates internally by using MPI communication, but is unable to communicate with frameworks not part of the same MPI communicator. The CCAFFEINE framework is an SPMD application that minimizes interoperation between the parallel cohorts and does not follow the master/slave approach of the other CCA-compliant frameworks mentioned. We argue that CCAFFEINE has similar problems scaling as the other approaches because it creates an architecture where each framework process contains all of the framework data (all nodes are master nodes). This incurs the same scalability problems, while also paying even larger performance penalties when the framework contains a large number of nodes. Additionally, the redundancy of data in CCAFFEINE does not mean fault tolerance: if one MPI process fails the rest of them will usually hang.

The design of our CCALoop component framework follows previous work on peer-to-peer communication and systems. Specifically, the use of Distributed Hash Tables (DHT) [7] for distributed systems that was pioneered by systems such as Chord [8]. Several other systems apply these design principles to applications such as file-sharing, although we are unaware of a component framework application. The work by Gupta et al. [9, 10] argues for the use of a one-hop lookup scheme in distributed Chord-like systems that do not have an extremely dynamic membership. Our domain of high-performance computing does not have high node turnover and requires that we provide low-latency communication within the framework which was best accommodated through a one-hop design. Therefore, we adopted this approach in our component framework.

Highly distributed systems aimed to support middleware have been developed with the advent of the Internet. The Grid [11] is one of them that is also directed toward scientific computing applications. A system designed for the Grid that is based on DHT is the approach for a range queried grid information service [12]. This approach extends the DHT design principle to enable range queries (e.g. memory $>$ 250 Mbytes) and apply them to a grid information service. A CCA framework has similarities to a grid information service as they both answer system information queries. However, in this task a CCA framework has stricter latency and efficiency requirements. A CCA framework is additionally responsible for a number of other tasks like connecting, instantiating and executing components. Therefore, we directed the focus of our work to low latency, scalability, multi-user capabilities, and parallel components.

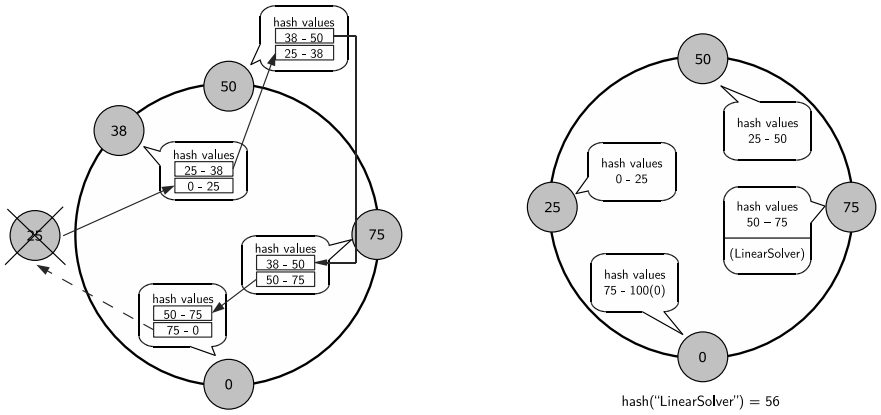
3 Design

Current distributed framework design is inappropriate in accommodating component applications with numerous components that use many computing resources. We implemented a CCA-compliant distributed component framework called CCALoop that prototypes our design for increased framework scalability and fault tolerance. CCALoop scales by dividing framework data storage and lookup responsibilities among its nodes. It is designed to provide fault-tolerance and uninterrupted services on limited framework failure. CCALoop also provides the ability to connect multiple GUIs in order for users to monitor an application from multiple points. While providing these capabilities CCALoop does not add overwhelming overhead or cost to the user and satisfies framework queries with low latency. In this section we will examine the parts that form the structure of this framework. We begin by looking more closely at the tasks and roles of a CCA-compliant component framework.

The main purpose of a component framework is to manage and disseminate data. Some frameworks are more involved, such as Enterprise Java Beans [13], but in this work we focus on the ones in the style of CORBA [14] that do not interfere with the execution of every component. This kind of a component framework performs several important tasks in the staging of an application, but gets out of the way of the actual execution. Executing components may access the framework to obtain data or to manage other components if they choose to, but it is not usually necessary. CCA-compliant frameworks also follow this paradigm as it means low overhead and better performance.

CCA-compliant frameworks store two types of data: static and dynamic. The majority of the data is dynamic, which means that it changes as the application changes. The relatively small amount of static data describes the available components in the system. In a distributed setting, static data consists of the available components on each distributed framework node. The dynamic data ranges from information on instantiated components and ports to results and error messages. A significant amount of dynamic data is usually displayed to the user via a GUI. In our design, we distribute management of framework data without relocating components or forcing the user to instantiate components on a specific resource. The user is allowed to make his or her own decisions regarding application resource usage.

One of the principal design goals of CCALoop is to balance the load of managing component data and answering queries to all participating frameworks. This is done by using a DHT mechanism where each node in the system is assigned a unique identifier in a particular identifier space. This identifier is chosen to ensure an even distribution of the framework identifiers across the identifier space. We provide an operation that hashes each component type to a number in the identifier space. All metadata for a given component is stored at the framework node whose identifier is the successor of the component hash as shown in Figure 2(b). Given a random hash function the component data is distributed evenly across the framework nodes. The lookup mechanism is similar to the stor-



(a) **(b)**

Fig. 2. (a) Framework data replication across node's two successors. Shown as node 25 is leaving the system and before any data adjustments have been made. (b) The data responsibilities of the CCALoop framework with four nodes.

age one: to get information about a component, we compute its hash and query the succeeding framework node.

3.1 Loop Structure

CCALoop's framework nodes are organized in a ring structure in topological order by their identifier numbers. Each framework node has a pointer to its successor and predecessor, allowing the ring to span the identifier space regardless of how the system may change or how many nodes exists in a given time. CCALoop also facilitates a straightforward way of recovering from node failure, by naturally involving the successor of the failed node to become new successor to the queried identifier. Use of a loop structure with a DHT lookup is commonly found in several peer-to-peer systems such as Chord [8].

Adding framework nodes to the system splits the responsibility for framework data between the joining node and its current owner. It is a two step process that begins at any already connected node in the system. The first step is assigning an identifier to the joining node that best distributes the nodes across the identifier space. The identifier in the middle of the largest empty gap between nodes is selected based on the queried framework node's information. Later, we will explain how every node in the system contains perfect information about the existence of other nodes in the distributed framework. Apart from a well-chosen identifier, the node is given the address of its predecessor and successor. The second step is for the joining node to inform the predecessor and successor that

it is about to join the network so that they can adjust their pointers. This step also resolves conflicts that may occur if two joining frameworks are assigned the same identifier in the first step by two separate nodes. If this conflict occurs, one of the nodes is assigned a new identifier and forced to repeat the second step. Removing nodes in the system has the opposite effect as adding nodes: the successor of the leaving node becomes responsible for the vacated identifier space and associated framework data.

Periodically, nodes invoke a *stabilize()* method that ensures that the successor and predecessor of that node are still alive. If one or both has failed, the node adjusts its predecessor or successor pointer to the next available node. Since perfect loop membership information is kept at every node, finding the next node in the loop is straightforward. The process of updating the predecessor and successor pointers ensures that the loop structure is preserved, even when framework nodes leave the system. When a node leaves and this readjustment takes place, the identifier distribution may become unbalanced. This will last until a new framework joins; when it will be instructed to fill the largest current gap in the identifier space.

In order to enable seamless functioning of CCALoop during framework node failure we replicate the data across successor nodes, so that if a framework fails, its successor is able to assume its responsibilities. To be able to handle multiple simultaneous failures we can increase the number of successors to which we replicate the framework data. This incurs a bandwidth cost proportional to the number of replicas. If a node joins or leaves the system, some data readjustment is performed to ensure that the replication factor we have chosen is restored. CCALoop targets the high-performance scientific computing domain, which uses dedicated machines with high availability. Machine failures or intermittent network failures are possible, but infrequent. Because of this, we are content with providing two or three replicas for a particular framework node's data. Figure 2(a) shows an example of data replication across two successors as a node leaves the framework.

3.2 One-hop Lookup

An advantage of our distributed framework design is the ability for nodes to contact other nodes directly, or through "one-hop". The one-hop mechanism was initially designed as an alternative to Chord's multi-hop query design [9]. One-hop lookup enables low latency querying, which is important in maximizing performance to components in a distributed framework. In order to support one-hop lookups, full membership awareness is required in the framework; every node needs to keep updated information about all other nodes in the system. There is certainly a cost to keeping this information current in the framework and it is proportional to the joining and leaving (turnover) rate. As with data replication, our expectation is that framework nodes comprising a CCA-compliant distributed framework will not have the same framework turnover rate as one of the popular file sharing networks, where turnover is an issue. Therefore, our design is unlikely to encounter very high levels of node turnover. When node

turnover does occur, our distributed framework would provide graceful performance degradation.

CCALoop uses a multicast mechanism to provide easy and quick dissemination of membership information. This mechanism creates a tree structure to propagate node leaving and joining information to all nodes in the system. We divide our loop structure into a number of slices, and assign a "slice leader" node to each slice. In CCALoop, the slice leader is the node with the smallest identifier in the slice. When a node joins the framework, its successor contacts the slice leader. The slice leader distributes this information to all other slice leaders as well as all the other nodes in the slice. Finally, each slice leader that received the message propagates it to all members of its slice. This hierarchy enables faster membership propagation which in turn enables CCALoop to reach a steady state faster. Additionally, this reduces errant queries as well as providing the means for low-latency access to framework nodes.

3.3 Multiple GUIs

Providing a graphical user interface is an important role of a scientific component framework. A framework's user is interested in assembling a simulation, steering it, and analyzing intermediate and final results. In large, cross-organizational simulations several users may need to manage a simulation, and several others may be interested in viewing the results. State of the art CCA-compliant scientific computing frameworks provide the capability to attach multiple GUIs and users. However, each of these frameworks provides that capability only at the master node, which hinders scalability as previously discussed. One of the opportunities and challenges of a scalable distributed framework like CCALoop is to handle multiple GUIs.

CCALoop allows a GUI to attach to any cooperating framework node. A user is allowed to manage and view the simulation from that node. When multiple GUIs are present, we leverage the slice leader multicast mechanism to distribute information efficiently to all frameworks with GUIs. We establish a general event publish-subscribe mechanism with message caching capability and create a specific event channel for GUI messages. Other channels may be created to service other needs. GUIs on which some state is changed by the user are event publishers, while all the other GUIs in the system are subscribers. We route messages from publishers to subscribers through the system by passing them through slice leaders while ensuring that we are not needlessly wasting bandwidth. All messages are cached on the slice leader of the originating slice of nodes.

The reason we use the hierarchical mechanism to transfer GUI messages over a more direct approach is to cache the GUI state of the framework. CCALoop provides a mechanism that is able to update a GUI with the current state when this GUI joins after some user operations have already occurred. To prepare for this scenario, we continuously cache the contribution to the GUI state from each slice at its slice leader. This is advantageous since we expect GUIs to often join at midstream, then leave the system, and possibly return.

An additional concern with multiple distributed GUIs is the order of state-changing operations. We use a first-come-first-serve paradigm and allow every GUI to have equal rights. A more complex scheme is possible and would be useful, but that is outside the scope of this paper.

3.4 Parallel Frameworks

To support CCA's choice of SPMD-style parallel components, a framework needs to be able to create a communicator, such as an *MPI Communicator*, which identifies the set of components that are executing in parallel and enables their internal communication. This internal (inter-process) communication is embedded in the algorithm and it is necessary for almost any parallel computation. To produce this communicator a framework itself needs to be executing in parallel: In order to execute the component in parallel we first execute the framework in parallel. A parallel framework exists as a resource onto which parallel components can execute. Parallel component communication and specifically parallel remote method invocation can be quite complex and it has been studied extensively [15].

A concern of this work is the inclusion of parallel frameworks in a distributed framework environment involving many other parallel and non-parallel frameworks. The parallel framework can be handled as one framework entity with one identifier or it can be considered as a number of entities corresponding to the number of parallel framework processes. We choose to assign one identifier to the entire parallel framework to simplify framework-to-framework interaction. This needs to be done carefully, however, to ensure that communication to all parallel cohorts is coordinated. By leveraging previous work in the area of parallel remote method invocation we gain the ability to make collective invocations. We use these collective invocations to always treat the parallel framework as one entity in a distributed framework.

Even though we choose to treat all parallel framework processes as one parallel instance, the component's data that the framework stores is not limited to one entry per parallel component. To enable a more direct communication mechanism, we need to store an amount of data that is proportional to the number of parallel processes of the parallel component. Large parallel components are a significant reason that more attention should be directed toward scalable distributed component frameworks.

4 Results

In this section, we intend to show that scalability of component frameworks is a practical concern in large-scale simulations consisting of many components and computational resources. The CCALoop component framework implementation is designed to scale gracefully under heavy loads due to its design that distributes the load across the distributed framework. In contrast, other frameworks that are

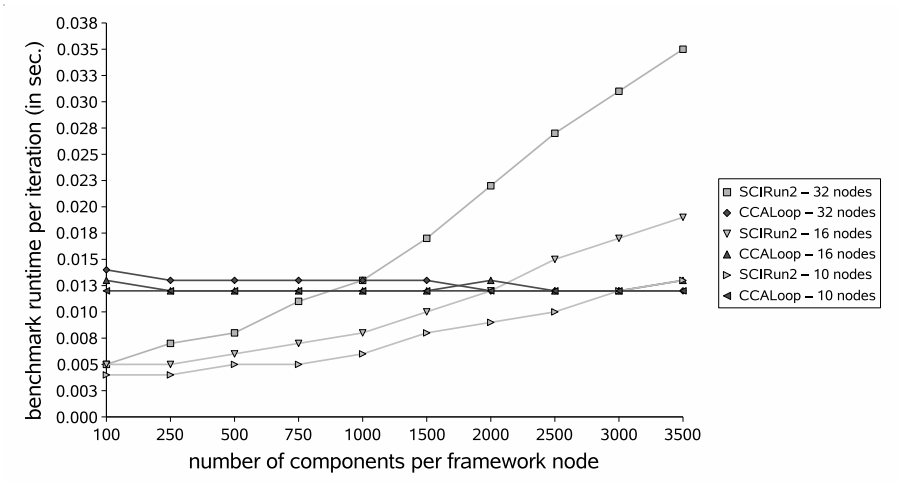


Fig. 3. Scalability comparison of SCIRun2 and CCALoop for 10, 16, and 32 nodes.

not designed with scalability in mind, such as SCIRun2, ought to significantly degrade in performance after the number of components reaches some threshold.

To understand the scalability constraints of CCA-compliant frameworks we benchmarked and compared CCALoop and SCIRun2. The CCALoop framework implements the distributed framework design discussed in this paper, while SCIRun2 uses a master/slave hierarchy. In each of these frameworks, we implemented a simple real-world scenario: instantiate a large number of components and then attempt to connect some of their ports. This benchmark is a simplified group of the tasks any CCA framework would be asked to perform in the setup stage of a simulation (in pseudo-code):

```
framework.createInstance(‘‘Hello’’);
framework.createInstance(‘‘World’’);

ComponentID hello_id = framework.getComponentID(‘‘Hello’’);
ComponentID world_id = framework.getComponentID(‘‘World’’);

hello_ports = framework.getProvidedPortNames(hello_id);
world_ports = framework.getUsedPortNames(world_id);

framework.connect(world_ports[0], world_id,
                  hello_ports[0], hello_id);
```

We measure the time it takes to execute this benchmark in SCIRun2 and CCALoop and present the results in Figure 3. We executed the benchmark on a cluster of dual-processor P4 Xeon nodes with 2GB of memory per node. The graph shows the the average time it takes to complete one iteration of the benchmark in CCALoop and SCIRun2 as the number of components per framework

node increase. We show multiple lines presenting benchmark execution on a different number of nodes in the cluster.

Each of the lines (in light gray) showing the SCIRun2 benchmark runtime have a steadily increasing slope indicating that the number of components has a negative effect of SCIRun2's ability to perform basic tasks. It may seem that a very large number of components are possible before SCIRun2's performance degrades. We also notice that the performance of SCIRun2 get worse at a faster rate in the lines representing larger number of nodes. On the other hand, the performance of CCALoop (in dark gray) remains steady across executions on different node sizes. The intersection of the corresponding (by number of nodes) SCIRun2 and CCALoop lines moves in the left direction to smaller numbers of components as the node numbers increase.

This indicates that simulations executed on 128 or more nodes, which are common in the scientific computing domain, may be adversely affected by SCIRun2's centralized design at small numbers of component instances. In addition, these large simulations often include parallel components whose effect on the framework is approximately proportional to their number of processes. Therefore we expect that frameworks similar in design to SCIRun2 will be easily overwhelmed in these cases, making CCALoop the necessary framework choice.

5 Future Work and Conclusions

The future work of this project is to include these features, which we have shown to be implementable and beneficial in CCALoop, into our production-level CCA framework SCIRun2. This will produce an in-depth practical evaluation of the usefulness of this distributed framework design. We also intend to include improved resolution of conflicting user actions in CCALoop. Currently, the framework accepts the first user action and rejects any conflicting actions that arrive later. Parallel components and their communication are also a continuing area of interest to this project.

This paper presents a design that enables scalability and fault tolerance in distributed frameworks. It also provides, through our prototype framework CCALoop, an implementation of the CCA component model that supports multiple GUIs and parallel components. Each of these capabilites is important to the creation of a next-generation component framework supporting the future needs of component-based scientific simulations. As parallel components become more prevalent and as simulations grow larger and more encompassing the need for the design attributes that CCALoop provides will become more apparent.

6 Acknowledgments

The authors gratefully acknowledge support from NIH NCRR and the DOE ASCI and SciDAC programs and would like to thank Ayla Khan and Jeremy Archuleta for helpful suggestions on drafts of this manuscript.

References

1. Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., Smolinski, B.: Toward a common component architecture for high-performance scientific computing. In: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation (HPDC). (1999)
2. Zhang, K., Damevski, K., Venkatachalapathy, V., Parker, S.: SCIRun2: A CCA framework for high performance computing. In: Proceedings of The 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments. (2004)
3. Allan, B.A., Armstrong, R.C., Wolfe, A.P., Ray, J., Bernholdt, D.E., Kohl, J.A.: The CCA core specification in a distributed memory SPMD framework. Concurrency and Computation: Practice and Experience **14** (2002) 323–345
4. Krishnan, S., Gannon, D.: XCAT3: A framework for CCA components as OGSA services. In: Proceedings of The 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments. (2004)
5. McInnes, L.C., Allan, B.A., Armstrong, R., Benson, S.J., Bernholdt, D.E., Dahlgren, T.L., Diachin, L.F., Krishnan, M., Kohl, J.A., Larson, J.W., Lefantzi, S., Nieplocha, J., Norris, B., Parker, S.G., Ray, J., Zhou, S.: Parallel PDE-based simulations using the Common Component Architecture. In Bruaset, A.M., Tveito, A., eds.: Numerical Solution of PDEs on Parallel Computers. Volume 51 of Lecture Notes in Computational Science and Engineering (LNCSE). Springer-Verlag (2006) 327–384
6. Bell, G., Gray, J.: What's next in high-performance computing? Communications of the ACM **45** (2002) 91–95
7. Gribble, S., Brewer, E., Hellerstein, J., Culler, D.: Scalable, distributed data structures for Internet service construction. In: Proceedings of the Symposium on Operating Systems Design and Implementation. (2000)
8. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM '01: Proceedings of the 2001 conference on Applications, Technologies, Architectures,and Protocols for Computer Communication, New York, NY, USA, ACM Press (2001) 149–160
9. Gupta, A., Liskov, B., Rodrigues, R.: One hop lookups for peer-to-peer overlays. In: Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX), Lihue, Hawaii (2003) 7–12
10. Gupta, A., Liskov, B., Rodrigues, R.: Efficient routing for peer-to-peer overlays. In: Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04), San Francisco, CA (2004)
11. Foster, I., Kesselman, C., eds.: The grid: blueprint for a new computing infrastructure. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)
12. Andrzejak, A., Xu, Z.: Scalable, efficient range queries for grid information services. In: Proceedings of the Second IEEE International Conference on Peer-to-Peer Computing (P2P2002). (2002)
13. Enterprise Java Beans: http://java.sun.com/products/ejb (2007)
14. OMG: The Common Object Request Broker: Architecture and Specification. Revision 2.0. (1995)
15. Bertrand, F., Bramley, R., Sussman, A., Bernholdt, D.E., Kohl, J.A., Larson, J.W., Damevski, K.: Data redistribution and remote method invocation in parallel component architectures. In: Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS). (2005) (Best Paper Award).