High Performance Communication Between Parallel Components

Chaudhry Hassan Afzal and Alan Sussman
Department of Computer Science, University of Maryland
{hassan, als}@cs.umd.edu

Abstract

The Common Component Architecture (CCA) is a standard for building high performance component-based applications, and CCaffeine is a CCA framework that supports SPMD style parallel components, in addition to standard sequential components. However, commmunication between parallel components is limited when using CCaffeine, only allowing communication between components in the same process. The Maryland InterComm library, on the other hand, is designed to enable efficient communication between parallel SPMD-style applications. We present the component version of InterComm, to allow CCA compliant application components to efficiently and flexibly transfer data. We describe two approaches to building components from a complex library such as InterComm within the Ccaffeine framework. More specifically, we discuss in detail how to enable CCaffeine to support a multiple component multiple data (MCMD) model using the InterComm components. Since the CCaffeine framework uses threads to implement components, and the InterComm components use a library that is not thread-safe, we describe how we have modified the basic MCMD model to avoid multi-threading thread problems. To manage multiple parallel and sequential components within the CCaffeine framework, we have designed a Manager *component that performs various initialization and bookkeeping functions to enable MCMD component execution. We also present experimental results to show that the componentized version of InterComm performs as well as the original library version.*

1. Introduction

As physical models become better understood and advances in computer hardware continue, complex computer simulations of those physical models are becoming ubiquitous. Often a single research group is only able to focus on a particular aspect of a real world phenomenon due to limited resources. In order to get a holistic understanding of the real world phenomenon it becomes necessary for these different simulations, each simulating a particular aspect of the entire problem, to interact so that the phenomenon or model can be fully analyzed. One common way for the simulations to interact is through data transfers at physical boundaries or in overlapping regions. Such *coupled* simulations create many difficulties in modern computing environments, because the different simulation components are often written in different programming languages, run on multiple processes as parallel programs and their data is distributed across those multiple processes. The problem of data transfer among high performance simulation components has even greater complexity when each application component runs on a different number of processes, also known as the MxN problem [1].

The Maryland InterComm library [2] provides functionality that enables parallel applications running in a high performance computing environment to exchange data efficiently and flexibly, supporting MxN process topologies. Since there has been widespread interest in the high performance computing community in moving from libraries and monolithic, highly complex application-specific frameworks to components based applications and lightweight component frameworks, a component-based implementation of InterComm functionality would components running in a framework to exchange data, with efficient communication of paramount importance, especially for parallel components. In this work, we focus on the Common Component Architecture [3], a lightweight component architecture targeted at high performance computing environments. More specifically, we target the Ccaffeine CCA framework [4], which supports both parallel and sequential components. While the Ccaffeine framework supports communication between parallel components, such support is very limited, only enabling communication between components running in the same framework process (so only among corresponding component processes for 2 or more parallel components). We address this limitation by creating CCA compliant components that encapsulate the InterComm library functionality, and then run those components within the Ccaffeine framework to enable parallel components to exchange data with other components (parallel or sequential) within the Ccaffeine framework. This work can benefit any CCA-compliant components by providing them with generic MxN data exchange capabilities, and also serves as a guide to implementing such functionality in any component framework that requires MxN capabilites to efficiently support parallel components.

In addition to describing the basic funcionality of the InterComm components, we also provide feedback to component framework developers by identifying several issues related to creating and running components created from libraries or applications that are not necessarily thread-safe. These issues stem from the way that component frameworks manage the components. Component frameworks often use threads, and not separate processes, to efficiently implement applications containing large numbers of components. This can cause problems for libraries or applications with shared state that may not allow those components to be used by more than one other component at the same time.

The rest of the paper is structured as follows. Section 2 provides additional details on the InterComm library, the Common Component Architecture and the Ccaffeine framwork. Section 3 describes how CCA components were created from the InterComm library, and also elaborates on the problems encountered in running the components in CCaffeine. We con-

clude and describe directions for future research in Section 4.

2. Background

2.1. InterComm Library

The InterComm library provides software support for simulating complex physical systems that require multiple physical models, potentially at multiple scales and resolutions and implemented using different programming languages and distinct parallel programming paradigms [2]. The individual models are coupled to allow them to exchange information either at boundaries where the models align in physical space or in areas where they overlap in space. InterComm enables these coupled simulation components to easily and efficiently exchange data, with minimal changes to the existing simulation code.

InterComm requires the application developer to specify an XML Job Description (XJD) file to determine which simulation components are to be run and which data objects will be transferred between pairs of components. It also provides calls for specifying what data should be transferred between the coupled components at runtime, no matter how the data is distributed across processes for a parallel program. Decisions about when a data transfer would actually take place are different between the two available versions of InterComm. In the earlier version, the data exchange times are completely specified by corresponding export and import calls in the two participating components. In the latest version of InterComm a coordination policy is specified in the XJD file to allow a component to specify its data transfer requirements independent of other components. It becomes the responsibility of the application developer connecting the components to ensure that the data transfer requirements of each component are met. InterComm is responsible for performing the data transfers properly according to the coordination policy. The coordination specification includes a range of matching policies which match *timestamps* a user provides for each import and export requests, resulting in a variety of options for the application developer to determine when data is transferred on each connection. The policies allow for matching timestamps within a given window, rather than an exact match, giving more flexibility in deciding when to transfer data.

2.2. Common Component Architecture

The Common Component Architecture [5] is designed to cope with the increasing complexity of large scale scientific software systems. As supercomputers on which these software systems are run grow more powerful, simulations with greater complexity and sophistication become more common. The need to manage the development of these simulations has been alleviated somewhat by using libraries or computational engines that provide parameterized algorithms to suit each individual target computer system [3]. However those efforts do not sufficiently address many issues related to building and running large, complex software systems. One reason is that many legacy scientific application codes can't for technical or practical reasons be rewritten to use these higher level solutions. Exploring other venues to tackle this problem leads us to component based programming which has already gained foothold in other areas of computing.The Object Management Group's CORBA [6], Microsoft's COM and DCOM [7], and Sun's Enterprise JavaBeans [8] are examples of very popular component environments in the business and internet areas.

Components may be thought of as objects that encapsulate useful units of functionality and interact with other components only through well-defined interfaces. The component approach facilitates reuse and interoperability of code. The crux of CCA is the concept of ports through which the components interact with eachother. Ports can be thought of as implementation independent interfaces. A component wishing to offer some functionality to other components does so through the functions defined on the port it provides. Similarly, a component wishing to utilize the functionality of another component does so by declaring that it uses the port provided by the other component. The CCA mandates a Uses/Provides connection design pattern for a component to invoke another component's methods.

Components get connected to other components to form a whole application within a framework. Given this approach constructing a complex scientific application could be a matter of minutes by joining all the appropriate connections amongst components. Furthermore, the required components can already be made available through a component repository where each component has been written by the experts in their respective fields.

2.3. Ccaffeine Framework

Ccaffeine is a framework implementation compliant with the CCA core specification [4]. A number of parallel scientific software frameworks already exist such as POOMA [9], Sierra [10], Overture [11], HYPRE [12], GrACE [13], [14], PetSC [15], CCAT [16] and Uintah [17], all of which seek to automate parallel computations for a particular scientific or numerical domain. Among all these frameworks the closest to Ccaffeine is Uintah, however Uintah is a more heavyweight framework. CCAT is a CCA compliant framework that focuses on distributed computing.The Ccaffeine framework's forte include its ease of use. This ease of use however comes at a cost of Ccaffeine having to provide specialized components for additional functionality. This less automatic approach that Ccaffeine adopts is made up for by the framework's extensibility. Ccaffeine is fast and lightweight providing framework services by using external portable components instead of integrating all services into a single heavy framework core [4]. The Ccaffeine framework extensibility, ease of use, and support for parallel components makes it a good choice for componentizing InterComm library.

3. Steps To Componentizing InterComm

The problem that we are trying to solve is enabling data exchange among high performance computing applications

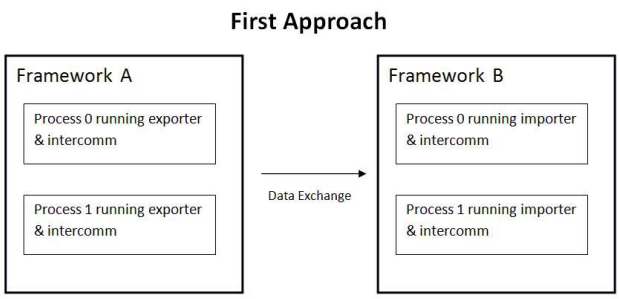

Fig. 1. 2x2 Exporter & Importer Running in Two Seperate Frameworks

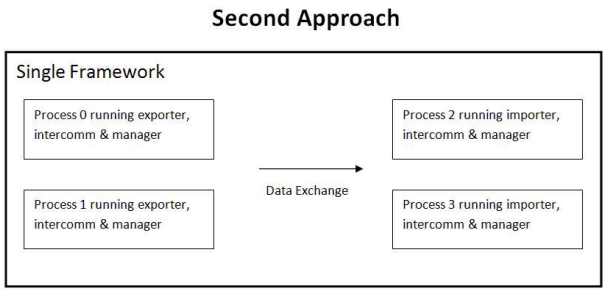

Fig. 2. 2x2 Exporter & Importer Running in a Single Framework

efficiently and flexibly. The InterComm library provides just that ability for HPC applications however in the component based programming world there is an absence of a component providing data exchange abilities similar to the ones that the InterComm library provides. Thus we seek to fill this void by creating a component version of the InterComm library. Converting a library into a component isn't a trivial task because a library can have global states and parallel libraries can complicate matters further by use of PVM or MPI which may or may not be thread safe.

3.1. InterComm Library As a Component

InterComm library isn't thread safe and the component version of the library too suffers from the same problem. The most straightforward solution to resolve thread safety issues for the InterComm component is to run the applications participating in the data transfer in separate instances of the Ccaffeine framework thus ensuring that only one component instance is running in a process at one time and calls to the InterComm library are made by a single thread running in a process. We experimented with two application components; one exporter and the other importer both running in two separate Ccaffeine frameworks. Though the InterComm library's componentized version successfully works in this scenario however the solution in addition to being inelegant is cumbersome for the user who now has to start two separate Ccaffeine frameworks. This would turn into a nightmare for the user if the number of components using the InterComm component is higher for instance five or ten where now the user would be obliged to run five or ten separate instances of the Ccaffeine framework. Note that this straightforward solution doesn't require the use of a parenting or managing component as is required in our next approach.

The first approach can be better understood with Figure 1. This is a trivial example of an importer and an exporter each running with two instances in seperate frameworks. This is in contrast to Figure 2 where each of the application component runs in a single framework. Figure 2 depicts our second approach. Note that in Figure 2 each application component is running within a seperate processes even though they have been instantiated within the same framework.

The InterComm library isn't monolithic and consists of

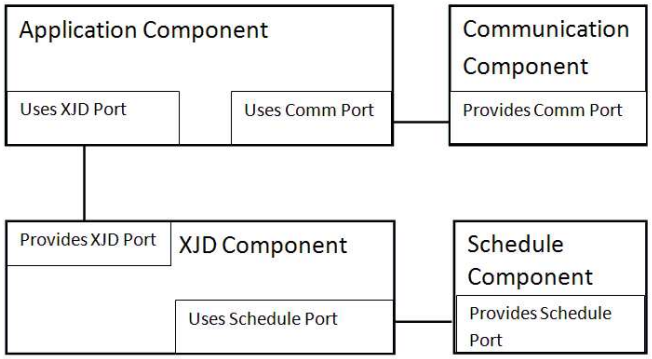

Fig. 3. Connections amongst InterComm Components and Application Component

different portions of code performing distinct functionalities. In keeping with this view of the library we have broken down the corresponding InterComm component into three distinct sub-components. Two of the components relate to initialization routines and the third is involved in the actual communication of data. The three components are the XJD Component, the Schedule component and the Communication component. Any application component in Ccaffeine wishing to use InterComm capabilities needs to connect to its own set of these three components.

The XJD component provides the XJD port which is used by the application components. The XJD component reads in the XJD file which is similar to the one written for InterComm library version. The XJD component in turn uses the Schedule port which is provided by the Schedule component. The Schedule component computes the schedules for the data exchange, which only need to be computed once at startup. Finally the application component uses the communication port provided by the Communication component to import or export data. The application component doesn't need to use the XJD component for second or any subsequent data transfer rather it directly uses the Communication port. The connections among components for a typical InterComm scenario are shown in Figure 3.

3.2. Need for a Manager Component and the Resulting MCMD Design

Any data transfer operation within Ccaffeine would involve at least two components both of which would be running in their respective processes at the time of the transfer. That is neither of the two components can be blocked on a uses port call or be inactive at the time of the transfer. Unlike a typical application model in Ccaffeine where only one component is active or is blocked while it makes a uses port call to another component we require both the exporter and the importer to be able to send and receive data using the Communication component's port. This necessitates that a overseeing component be created which starts up both the exporter and the importer components as threads and then waits for them to finish processing. We call this parenting component the Manager Component and we envision it to be responsible for a whole lot of other bookkeeping and connection management activities in the future when we introduce dynamic connections within the InterComm library.

The Manager Component is started up with the go port and runs in every instance of the framework. Each instance of the Manager Component decides which application component would it run on the same framework instance as the one on which it is running itself. In our case the Manager Component instance could run either the exporter or the importer by calling the functions defined on the port that the importer or the exporter provides and the Manager Component uses. Since the same code is executed by all the instances of the Manager Component each instance is able to figure out if it is required to start up the exporter or the importer on the same framework instance as it is running on. Note that the Manager Component instance would only run one of the importer or the exporter but not both. The call to the application component by the Manager Component isn't blocking that is the Manager simply invokes a function on the application component where a thread is spawned with the main function of the application component as the target. This allows the Manager to get back control and perform other functions if need be while the application component also keeps running. Finally the Manager waits upon the application components to signal it that they have completed their processing thus allowing the Manager to exit too. For sequential components the Manager Component starts up the component on a single instance of the framework.

Another reason to use the Manager component is to realize the MCMD model within Ccaffeine. The real power of the InterComm library lies in enabling transferring data between two applications when each of them is running on a different number of processes also known as the MxN problem. Ccaffeine trivially launches every component on every process requiring additional intervention by the application developer to ensure that components run on only a subset of processes. This additional supervision in our case is provided by the Manager component which makes sure that the components are only started on the specified number of processes in the XJD file.The MCMD model is illustrated in Figure 4.

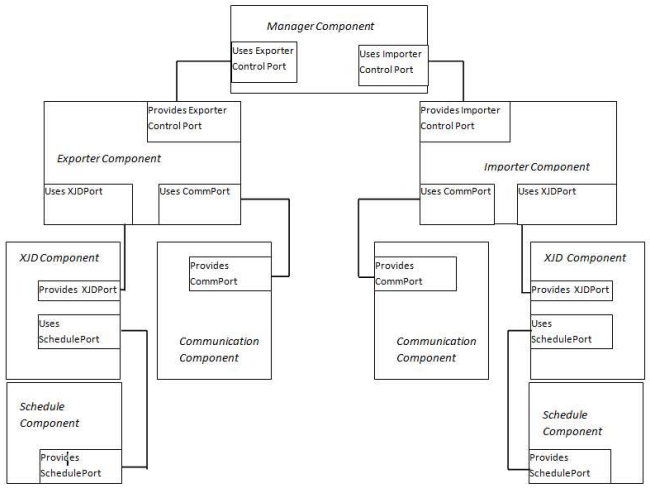

Fig. 4. Multiple Component Multiple Data (MCMD) model for a possible scenario

3.3. Overcoming Thread Safety And Implementation Issues

A further subtlety in the design of the InterComm component lies in the use of PVM by the underlying implementation of the InterComm library. Unfortunately PVM isn't thread safe. This means that if the importer and the exporter are running in the same process and make calls to their respective InterComm components, the entire framework would crash because of PVM failure. In order to address this issue the Manager component ensures that the importer and exporter run on different subsets of processes i.e. disjoint subsets of processes. Note that this requirement mandates that there must be atleast M+N processes assuming the exporter and the importer run on M and N processes respectively. Thus no two components using InterComm component would run on the same process or for that matter no two components making PVM calls can run in the same process.

The next issue we tackle relates to the way InterComm library has been implemented. The InterComm library requires that each application's instances running on different processes be numbered starting from zero i.e. their ranks should start from zero. Say, we have an exporter and an importer running on two and three processes respectively for a total of five processes as depicted in Figure 5. Now for the exporter component running on processes zero and one the library doesn't complain since the numbering of that application component's instances starts from zero however for the importer component which has instances running in processes two, three and four the library complains since the instances are not numbered starting from zero. To address this issue we create disjoint MPI groups and their related MPI communicators in the Manager component to ensure that every component's instances get numbered starting from zero. The manager passes the communicators when starting

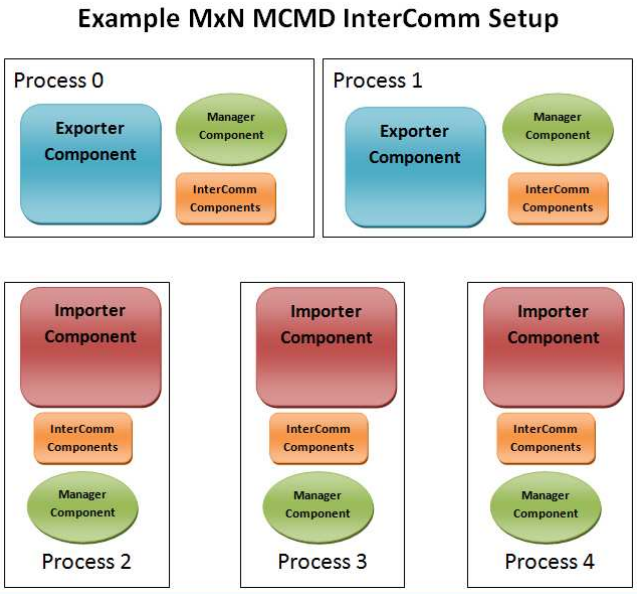

Fig. 5. MxN example of an Importer and Exporter. The Importer runs on 3 processes and the Exporter on 2

up each of the application component so that the instances of each application component running on different processes can communicate amongst themselves if need be using the passed in MPI communicators.

The MCMD model of InterComm can be understood from Figure 5 which shows a 2x3 case where the exporter runs on two processes and the importer on three.The manager and the InterComm components run on every process however the manager starts up the exporter only on processes zero and one. On the other hand, the importer is started up on the remaining three processes numbered from two to four. Note that the manager would create a separate MPI groups and communicators both for the importer and the exporter thus ensuring that the instances of the importer are numbered starting from zero to two. Also notice that the importer and the exporter run on disjoint sets of processes to make sure simultaneous calls to PVM aren't made

The thread-safety issues uncovered while implementing the component version of InterComm offer good feedback for CCA compliant framework design. Each instance of a framework runs as a separate process and the components the framework contains run as threads within that process. Running components as threads within a single process turned out to be an issue with InterComm in particular but this also reveals a more general problem in designing frameworks with this approach. Component based programming advocates ease of use, interoperability and reusability however these ideals might be hard to achieve for developers of application components which aren't thread-safe. Using two or more instances of an application component which either itself isn't thread-safe or makes function calls to a library such as PVM which isn't thread safe can lead to inconsistent states for the components or incorrect results.

Our work suggests that given the present capabilities of the Ccaffeine framework all scenarios where application components aren't thread-safe would require the use of a parenting component similar to the one that we have as the Manager Component to make sure that only a single instance of an application component runs in a single process thus eliminating any potential inconsistent behavior. Ideally, though the component developer shouldn't be bothered about thread-safety issues when designing and developing an application component to truly conform to the goals of component based programming. Thread safety issues can prove to be significant interia in the faster adoption of the component based programming paradigm by the scientific community.

3.4. How InterComm Can Help Other Components

Addition of the InterComm component for Ccaffeine will allow application components to exchange data flexibly and more efficiently within the Ccaffeine framework, however the real benefit comes to application components which require the MxN setting which Ccaffeine has no way of providing trivially. Furthermore, application components within Ccaffeine who don't use MPI have now an alternate way to exchange data using the InterComm component.

4. Conclusion And Future Research

We have thus shown a general MCMD pattern that could be adopted to overcome thread-safety issues for application components by allowing only one application component instance which isn't thread-safe to run in a single process. We apply this approach to the InterComm library and successfully produce a component version of the library which will enable components within Ccaffeine to exchange data in a MxN topology. Lastly, we offer food for thought for component framework developers who design frameworks to run components as threads rather than as seperate processes within the framework as the latter approach makes using and developing components which aren't thread-safe nontrivial. Note that the CCA standard doesn't mandate one or the other approach.

We have progressed from a simple static version of InterComm by adding dynamic timestamp matching and conversion of the library into a component for the Ccaffeine framework. These steps are intermediate milestones along our way to ultimately allowing dynamic connections to enable data transfer among application components which would eliminate the need for defining connections in the XJD file which is read in at initialize time for now. This would entail calculation of the schedules at runtime and the Manager component would play a vital role in coordinating runtime connections among components in the future.

References

[1] F. Bertrand, R. Bramley, D. E. Bernholdt, J. A. Kohl, A. Sussman, J. W. Larson, and K. B. Damevski, "Data redistribution and remote method invocation for coupled components," *Journal of Parallel and Distributed Computing*, vol. 66, no. 7, pp. 931–946, Jul. 2006.

[2] A. Sussman, "Building complex coupled physical simulations on the grid with InterComm," *Eng. with Comput.*, vol. 22, no. 3, pp. 311–323, 2006.

[3] 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. Mclnnes, 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," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 163–202, 2006.

[4] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl, *Parallel Programming using C++*, ser. Concurrency and Computation: Practice and Experience. MIT Press, 2002, vol. 14, no. 5, ch. The CCA core specification in a distributed memory SPMD framework, pp. 323–345.

[5] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. Mcinnes, S. Parker, and B. Smolinski, "Toward a common component architecture for high-performance scientific computing," 1999.

[6] O. M. Group, "OMG's CORBA website http://www.corba.org."

[7] R. Sessions, *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.

[8] V.Matena, M. Hapner, and B. Stearns, *Applying Enterprise JavaBeans: Component Based Development for the J2EE Platform*. The Java Series, Addison-Wesley, 2000.

[9] J. R. et al., *Parallel Programming using C++*. MIT Press, 1996, ch. A framework for scientific simulations on parallel architectures, pp. 553–594.

[10] S. webpage, "http://www.cfd.sandia.gov/sierra.html."

[11] F. Bassetti, D. Brown, K. Davis, W. Henshaw, and D. Quinlan, "Overture: An object-oriented framework for high performance scientific computing," in *In Proceedings of SC98: High Performance Networking and Computing. IEEE Computer Society*, 1998, p. 9.

[12] E. Chow, A. J. Cleary, and R. D. Falgout, "Design of the hypre preconditioner library," in *In SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*. SIAM, 1998, pp. 106–116.

[13] M. Parashar and J. C. Browne, "Systems engineering for high performance computing software: The hdda/dagh infrastructure for implementation of parallel structured adaptive mesh refinement," 1997. [Online]. Available: citeseer.ist.psu.edu/parashar97systems.html

[14] M. P. Department, M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski, "A common data management infrastructure for adaptive algorithms for pde solutions," in *In SuperComputing 97*, 1997, pp. 1–22.

[15] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object-oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*. Birkhauser Press, 1997, pp. 163–202.

[16] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temoko, and M. Yechuri, "A component based services architecture for building distributed applications," 2000.

[17] D. D. St, J. Davison, S. Germain, J. Mccorquodale, S. G. Parker, and C. R. Johnson, "Uintah: A massively parallel problem solving environment," 2000.