

XCAT-C++: Design and Performance of a Distributed CCA Framework

Madhusudhan Govindaraju, Michael R. Head, and Kenneth Chiu

Grid Computing Research Laboratory (GCRL),
State University of New York (SUNY) at Binghamton, NY 13902, USA,
mgovinda@cs.binghamton.edu,
head@acm.org,
kchiu@cs.binghamton.edu

Abstract. In this paper we describe the design and implementation of a C++ based Common Component Architecture (CCA) framework, XCAT-C++. It can efficiently marshal and unmarshal large data sets, and provides the necessary modules and hooks in the framework to meet the requirements of distributed scientific applications. XCAT-C++ uses a high-performance multi-protocol library so that the appropriate communication protocol is employed for each pair of interacting components. Scientific applications can dynamically switch to a suitable communication protocol to maximize effective throughput. XCAT-C++ component layering imposes minimal overhead and application components can achieve highly efficient throughput for large data sets commonly used in scientific computing. It has a suite of tools to aid application developers including a flexible code generation toolkit and a python scripting interface. XCAT-C++ provides the means for application developers to leverage the efficacy of the CCA component model to manage the complexity of their distributed scientific simulations.¹

Key Words: CCA, XCAT-C++, component, performance, multi-protocol

1 Introduction

The software engineering benefits of component based software have been widely described in the literature: components foster code re-usability and provide high level abstractions to shield users from low level details. They provide a manageable unit for software testing, distribution and management, and reduce the complexity of building large scale scientific applications, which often require the integration of multiple numerical libraries into a single application. The plug-and-play characteristic of component architectures provides the ability to reuse components in multiple applications, and serve performance needs by allowing components to be swapped at run-time with others that meet the required Quality of Service (QoS) metrics.

A consortium of university and national laboratory researchers launched the “CCA Forum” [1] in 1998, to develop a Common Component Architecture (CCA) specification for large scale scientific computation. The CCA specification defines the roles and functionality of entities necessary for high performance component-based application

¹ Supported in part by NSF grants CNS-0454298 and IIS-0414981.

development. The specification is designed from the perspective of the required behavior of software components. However, the design and implementation of the framework, choice of communication protocol, and component discovery mechanisms have not been formally specified. This has facilitated different research groups to design, develop and evaluate the use of the same CCA specification to support a wide variety of applications [2].

XCAT-C++ is tailored for distributed scientific applications and is designed to meet the following goals: (1) the framework should have a modular design so that specialized modules can be easily loaded to extend the capabilities of the system; (2) applications should have the capability of seamlessly and dynamically switching to a suitable communication protocol to maximize effective throughput; (3) the overhead due to component layering should be minimal and not impact the overall performance of the distributed system; (4) each XCAT-C++ component should be capable of interacting with endpoints that are compliant with Grid Web services standards; (5) and a flexible, extensible and powerful code generation toolkit should be provided that can generate the transport protocol specific code and shield away the complexity of the run-time specific details in *stubs* and *skeletons*.

The remainder of this paper is organized as follows. In Section 2 we provide a brief introduction of the CCA specification and highlight its key concepts. In Section 3 we discuss in detail the design and implementation of the key features in XCAT-C++. Section 4 describes some utility modules in XCAT-C++. We present performance of XCAT-C++ in Section 5. Section 6 discusses related work, and we conclude with a summary and pointers to future work in Section 7.

2 The Common Component Architecture

The Common Component Architecture (CCA) [2] specification is an initiative to develop a common architecture for building large-scale scientific applications. CCA places minimal requirements on components to facilitate the integration of existing scientific libraries into a CCA framework and also to minimize the impact of the component layer on performance. The specification does not mandate the use of any specific form of distributed or parallel technology as the underlying communication architecture, thereby ensuring that it does not preclude applicability to serial, parallel, distributed or grid systems. CCA promotes interoperability by requiring all components to define their interfaces via a Scientific Interface Definition Language (SIDL) [3]. The Babel toolkit [4] can be used to generate glue code from SIDL to many programming languages including C, C++, Java, Fortran and Python. SIDL has been specifically designed for high performance scientific applications. It explicitly supports complex numbers, dynamic multi-dimensional arrays, parallel attributes, and communication directives.

Communication between CCA components takes place via their ports, which follow a *uses/provides* design pattern. A *provides* port is the public interface implemented by a component. It can be referenced and used by other components. It can also be viewed as the set of services that are exported by the component. A *uses* port is a connection endpoint that represents the set of functions that it needs to call. Port descriptions for CCA

components are provided using the SIDL specification. CCA applications are composed by connecting the *uses* port of one component to the *provides* port of one another. The mechanism by which calls are transferred from the *uses* port to the *provides* port of the connected component is handled differently by each underlying framework.

3 Design and Implementation Features of XCAT-C++

3.1 Mapping CCA Concepts in XCAT-C++

Components in a distributed application often span multiple address spaces and are seldom co-located. Applications are developed by wiring components together into a component assembly. To facilitate this approach, distributed CCA frameworks need to provide support for remote invocation, wherein calls between components seamlessly cross machine boundaries. Components also need the capability to instantiate other components on remote machines. We list a few important CCA concepts and describe how they are designed and implemented in XCAT-C++.

- **Services Object:** Each CCA component contains a *Services* object that is responsible for managing the component's ports, including the ones that are dynamically added during the execution of the distributed application. Application developers can retrieve handles to a component's ports or just inspect its current state via the standard API of the *Services* object. In XCAT-C++, the *Services* object is designed to encapsulate the framework specific bindings for the *provides* and *uses* ports. Whenever a *uses* port is requested, a pointer to a local object is returned, while for a *provides* port a global (serializable) reference is returned that can be sent to components in remote address spaces. The serializable form of the reference contains information necessary to communicate with the provides port from any component. This information includes details such as host name, port number, communication protocol and a globally unique ID for the provides port.
- **ComponentID:** The CCA specification states that each component should design the `ComponentID` as an opaque handle, but does not require any standard format. The motivation for this approach is to allow each framework to design the handle according to its application requirements. In XCAT-C++, the handle has been designed as an object that is serialized to a string format whenever it is transported to another component. The idea is for the remote handle to be compatible with emerging standards in Grid Web services, which have adopted the Web Services Description Language (WSDL) document to represent distributed services. A WSDL document is an XML document that is commonly stored in a string format. This design also allows an XCAT-C++ *ComponentID* to be used for component assembly via work-flow engines [5].
- **Builder Service:** The CCA *Builder Service* presents a standard API for all components to instantiate, connect and disconnect other components. Once a component has been instantiated, the service returns a *ComponentID* to the new component. This *ComponentID* can then be used to directly communicate with the component. In XCAT-C++, the builder service instantiates new components from a set of name-value pairs that encapsulate the remote environment details such as command line

arguments, executable location, target machine name, and creation protocol. Currently, XCAT-C++ supports the use of SSH and we are testing the incorporation of the Grid Resource Allocation and Management (GRAM) service for authenticated launch of components on Grid resources.

- **Component Communication:** In distributed scientific computing, components are instantiated on remote machines and wired together dynamically with running components. As a result the choice of protocol depends on dynamically changing factors including the data type and size that needs to be transferred, security policies, and the list of common protocols supported by a pair of interacting components. Also, a component is typically connected to several components at any given time, each connection probably optimized for a different protocol. We discuss the communication system of XCAT-C++ in detail in Section 3.2.

3.2 XCAT-C++ and Grid Web Services

Web services have emerged as the architecture of choice for grid systems. Standards such as Open Grid Services Architecture (OGSA) [6] and Web Services Resource Framework [7] define a set of Web services based specifications for accessing Grid resources. These standards share many design features with the CCA specification [8]. We briefly discuss how XCAT-C++ components can be used with Grid Web services.

- Two choices for mapping XCAT-C++ components to Web services are (1) every XCAT-C++ component can be a Web service, with the endpoint in the WSDL document for the service pointing to the *ComponentID* of the component; or (2) every *provides* port of a component can itself be a Web service, as it has a well defined interface and endpoint. Unlike in Web services, two ports of the same type belonging to the same CCA component can exhibit semantically different behavior. To keep this flexibility we have chosen to map each *provides* port to a different Web service (and hence a different WSDL document) that can be uniquely identified and separately accessed by Web service clients.
- The Open Grid Services Infrastructure (OGSI) specification (precursor to the WSRF specification) required each Grid service to have a standard Grid Service Port. This requirement can be trivially met in CCA by defining a standard *provides* port with all the operations of the Grid Service Port. We are currently working on mapping the collection of five specifications (WS-Resource Properties, WS-Resource Lifetime, WS-RenewableReferences, WS-ServiceGroup, WS-BaseFaults) of the WSRF framework to standard CCA *provides* ports. The idea is to make these services available by default to all XCAT-C++ components. In the current implementation, each component has access to a *Builder Service* by default.
- Resource Lifetime Management: the life cycle of each component is managed via the *Builder Service* and a standard *go* port. The *Builder Service* design is based on the factory model. The *go* port of each component can be used to stop or kill the current execution of a component. As the *go* port of a component is a *provides* port, it is described in a WSDL document that can be used by clients to manage the component's lifetime.

- Service Handles: Each CCA component has a unique (opaque) handle represented as a *ComponentID* object within the component address space and some native representation for the on-the-wire format. This concept maps directly to that of a Grid Service Handle (GSH) used by grid services. However, to enable interoperable communication between different CCA frameworks, we have proposed [9] that a standard CCA registry service be defined to convert the GSH of each framework to a WSDL format, which can serve as a service reference pointing to the endpoint of the *provides* port. This idea directly corresponds to the two level naming scheme adopted by grid Web services.

Multi-Protocol Approach for Component Interactions The imperative for multi-protocol design is clear when we consider the diverse communication characteristics of various distributed applications. There is no single best protocol that can meet the requirements for all data types and communication patterns. We have successfully incorporated the *Proteus* [10] multi-protocol library as the communication substrate for XCAT-C++. Proteus currently has support for two protocols: (1) XBS [11], an efficient streaming binary protocol; and (2) XSOAP, a C++-based implementation of the SOAP specification. We list some features of the multi-protocol approach employed in XCAT-C++:

- For communication between two XCAT-C++ components, both enabled by Proteus, communication can switch to an optimized communication protocol on a per-call basis. Proteus provides an API, which the XCAT-C++ framework can call, to select the communication library to be used for subsequent calls. Also, communication modules can be dynamically loaded for each component. An example scenario is shown in Figure 1, in which three entities (A, B and C) are connected to form a distributed application. For communication between components A and B, the efficient XBS protocol is used. When interaction with the Web Service (entity C) is required, the components can automatically switch to XSOAP, as SOAP is the only protocol supported by the Web service.
- The use of Proteus can serve as the basis for interoperability with components running in a different CCA framework. For example, to communicate with a component running on the Legion framework [9], a common-denominator protocol can be used to negotiate the use of the most optimized protocol available with both the frameworks. The negotiation and switch to the appropriate protocol can be handled by the framework and remain transparent to the application.
- The use of a multi-protocol approach provides a fail-safe mechanism for data transfer in XCAT-C++. If a particular protocol results in errors or has an unexpected loss in performance, the framework can dynamically switch to another communication protocol. Moreover, it allows error reporting to take place via a protocol that is different from the one that generated it.

A central issue in any system that relies on multiple protocols is finding break-even points – to know when one method is preferable to another. In numerical computing, such approaches are called *polyalgorithms*, and the break-even points can often be specified in terms of a few parameters giving problem characteristics independent of the

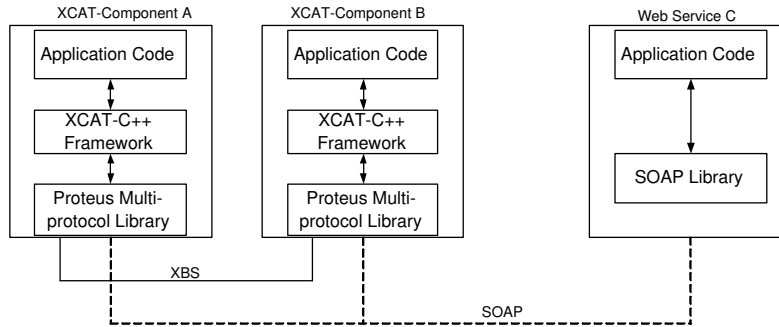


Fig. 1. The Figure shows a simple architecture of XCAT-C++ components using the Proteus Multi-protocol communication library. For communication between two XCAT-C++ components, the XBS (a streaming binary serializer) protocol is used, while for interaction with Web services, the XSOAP protocol is used.

computing environment. In communication systems, however, the issue is significantly more complex because of dependence on hardware, networks, and software implementations. For scientific applications, with widely varying communication characteristics, an extensive testing framework is required for each application.

3.3 Wormhole Routing

In wormhole routing, a message is divided into a sequence of (fixed size) data units, called flits. As the header flit moves, the remaining flits follow in a pipeline fashion. As opposed to the store-and-forward policy, worm-hole routing allows parts of a message to be forwarded to the next node even before the entire message has been received. All parts of a single message follow the same route. The overlapping of transmission with reception of data sets, when done for fixed sized chunks tuned for each system, can also maximize the benefits of cache hits. The wormhole routing feature is included as part of the Proteus communication library [10]. The use of wormhole routing in XCAT-C++ components will allow them to efficiently function as gateways for some distributed applications that just require efficient streaming of large data sets via network storage depots [12]. The XCAT-C++ gateway components do not have to store the entire data in memory at any given time and can start forwarding data chunks even before the entire data set has been received.

4 Utility Modules for Application Development

In this section we briefly describe a few tools that facilitate in providing ease-of-use for application scientists. While these don't directly address performance requirements they contribute to the rich experience of using component based technology, wherein each component is a binary unit of composition, and just the interfaces are needed for plug-and-play application development.

4.1 A Flexible Code Generation Toolkit

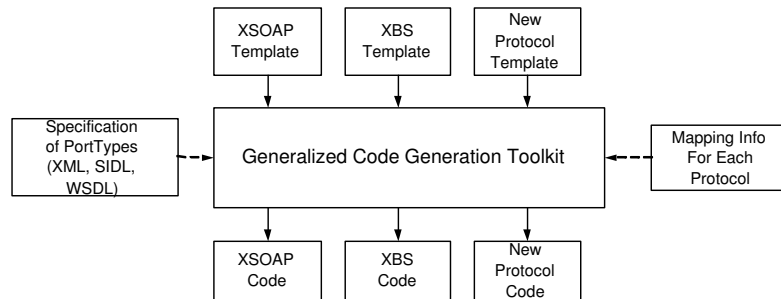


Fig. 2. The Figure shows the architecture of a flexible code generation toolkit that can generate the required glue code for many different communication libraries. The toolkit needs to be provided with the interface description and a template for the required communication library. Additionally, mappings can be provided to steer the code generation process.

The use of the Proteus multi-protocol library with XCAT-C++ allows applications to be built by composing components living in disparate heterogeneous environments using various communication protocols. Each communication substrate has low level details that are shielded from the user by isolating them in a library that is generated by a specialized code-generator. However, the use of several code generators to compose a distributed application is tedious and inconvenient. It imposes a burden on the user. It is desirable to have a single code generation toolkit that can be used for all communication protocols available to the framework.

Figure 2 shows a simplified design of our code generation toolkit that can be used to generate *stubs* and *skeletons* for a wide-variety of communication protocols. The same code generator can also be used to generate framework specific code. The common patterns in the generated code for distributed object systems are captured in a grammar, that can be used to specify *templates* for each communication protocol. Apart from these design patterns, the template can specify control structures and mappings for variables in code-templates. These templates need to be written only once by the designer of the communication library. A user needs to specify a CCA port type interface in XML (we will add support for SIDL in the near future) and pick a template from the available protocol-templates with the toolkit. The code generation toolkit understands the grammar used to define the templates, and can generate the required code accordingly.

4.2 Composition Model

In scientific computing, one often needs to run a distributed computation multiple times with minor variations. This makes a scripting language interface for building such applications invaluable. To accomplish this, we have developed a simple Python interface

to XCAT-C++ by using the Simple Wrapper Interface Generator (SWIG) [13] to translate calls between Python and the XCAT-C++ library. Scientists can use the CCA API directly from the python script and do not have to be concerned about the details of the XCAT-C++ implementation. The design of XCAT-C++ does not preclude the use of other composition models such as Matlab or Graphical Uses Interfaces (GUIs). We plan to add a convenient GUI interface so that users can visually drag and drop components from a repository to compose an application. This will also allow end-users to save configurations of successful runs as python scripts.

5 Performance

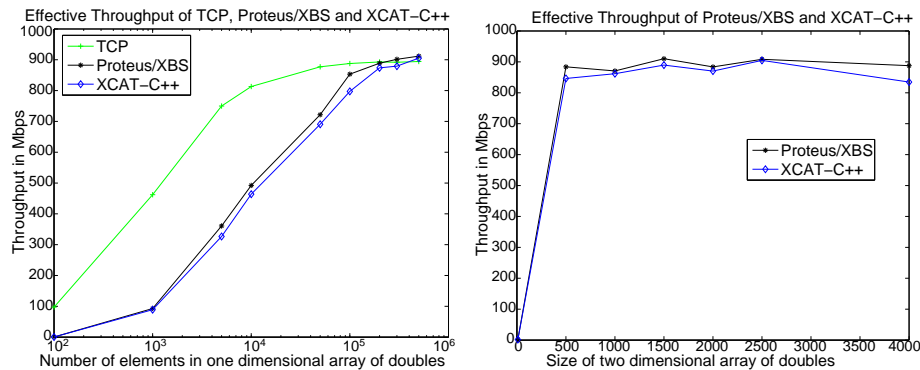


Fig. 3. The figures compare the performance of raw TCP, Proteus/XBS and XCAT-C++ for one- and two- dimensional arrays. XCAT-C++ is layered on top of Proteus and XBS is one of the communication protocols of Proteus. The plot for raw TCP serves as a standard with which we can judge the overhead of the other protocols. For very large arrays of floating point data (doubles), the performance of Proteus/XBS approaches that of TCP. The plot for XCAT-C++ shows that the overhead due to component layering is minimal. For large data sizes, XCAT-C++ achieves 900 Mbps on a Gigabit switched network.

Our test environment consisted of two dual processor machines, each configured with 2.0 GHz Pentium 4 Xeon with 1GB DDR RAM and a 15K RPM 18GB Ultra-160 SCSI drive running Debian Linux 3.1 (“sarge”) with the 2.4.26 kernel. The machines were connected by Gigabit Ethernet. Two XCAT-C++ components with compatible ports were launched on different nodes of the cluster. The port communication involved sending and receiving one and two dimensional arrays of various sizes. The code was compiled with gcc version 3.3.5. Our results reflect the average of multiple measurements for each reported data point.

The primary aim of our tests was to measure the overhead imposed by the XCAT-C++ component layering on data types commonly used by scientific applications.

XCAT-C++ was run by selecting the high performance streaming XBS parser [11] from the Proteus library. Figure 3 shows the performance comparison of raw TCP, Proteus/XBS and XCAT-C++ on top of Proteus/XBS. The performance of raw TCP is better than that of Proteus/XBS. This is expected as Proteus/XBS has additional overhead above pure transmission. The performance of XCAT-C++ closely matches that of XBS for all data sizes. The design of XCAT-C++ ensures that all application calls are transferred to the Proteus communication module without buffer copying and the component layering cost is restricted to just a few virtual method calls. The second plot of Figure 3 compares the performance of XCAT-C++ and Proteus/XBS for two dimensional arrays, and once again for large data sizes both XCAT-C++ and Proteus/XBS achieve an average of 900Mbps on a Gigabit switched network.

6 Related Work

The most widely used component models developed by the industry include CORBA Component Model (CCM), Distributed Component Object Model (DCOM) and Enterprise Java Beans (EJB). These component models have not been explicitly designed to meet the challenges of scientific computing. In particular, scientific applications require the component models to encapsulate parallel and distributed programs sending large, complex, and rapidly changing data objects.

Many CCA systems have been developed for different application domains. SCIRun2 [14] is specialized for parallel-to-parallel remote method invocation in a distributed memory environment. SCIRun2 has mainly been used for visualization applications. XCAT-Java [8], [15] is a Java framework that uses the Web services model as its basic architecture and supports the SOAP communication protocol. LegionCCA [16] uses the Legion object model and run-time system to launch applications on Legion-based grids. The CCAFFEINE [17] framework is specialized for parallel computing and supports both single program/multiple data (SPMD) and multiple program/multiple data (MPMD) models.

7 Summary and Future Work

We presented performance results to show that overhead of component layering on applications is minimal and for large arrays of floating point data XCAT-C++ delivers very high throughput. The features provided by the framework can shield users from the details of managing the scale and complexity of their scientific applications. These include a generalized code generation toolkit, a Python scripting composition model, and the use of a dynamic and efficient multi-protocol library with XCAT-C++ so that inter-component data exchanges can take place via the most optimized communication library for each pair of interacting components.

In future work, we plan to incorporate the use of the Babel toolkit for specialized applications that use multiple components in a single process, each potentially developed in a different language. We also plan to add support for new communication protocols for use with XCAT-C++.

References

1. CCA Forum: Common Component Architecture Forum (July, 2005) <http://www.cca-forum.org>.
2. Bernholdt, D.E., Allan, B.A., Armstrong, R., Bertrand, F., Chiu, K., Dahlgren, T.L., Damevski, K., Ewasif, W.R., Epperly, T.G.W, Govindaraju, M., Katz, D.S., Kohl, J.A., Krishnan, M., Kurfert, G., Larson, J.W., Lefantzi, S., Lewis, M.J., Malony, A.D., McInnes, L.C., Nieplocha, J., Norris, B., Parker, S.G., Ray, J., Shende, S., Windus, T.L., Zhou, S.: A Component Architecture for High Performance Scientific Computing. *International Journal of High Performance Computing Applications*, ACTS Collection Special Issue (2005)
3. Kohn, S., Kurfert, G., Painter, J., Ribbens, C.: Divorcing Language Dependencies from a Scientific Software Library. In: *Proceedings of 10th SIAM Conference on Parallel Processing*, Portsmouth, VA. (March 12-14, 2001)
4. Elliot, N., Kohn, S., Smolinski, B.: Language Interoperability for High-Performance Parallel Scientific Components. In: *International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE 1999)*, San Francisco, CA. (September 29 - October 2nd)
5. Gannon, D., Ananthkrishnan, R., Krishnan, S., Govindaraju, M., Ramakrishnan, L., Slominski, A.: 9, Grid Web Services and Application Factories. In: *Grid Computing: Making the Global Infrastructure a Reality*. Wiley (2003)
6. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: Grid Services for Distributed System Integration. *Computer* 35(6) (2002)
7. Globus Alliance: The WS-Resource Framework (2004) <http://www.globus.org/wsrp/>.
8. Govindaraju, M., Krishnan, S., Chiu, K., Slominski, A., Gannon, D., Bramley, R.: Merging the CCA Component Model with the OGSF Framework. In: *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*. (May 12-15, 2003, Tokyo, Japan.)
9. Lewis, M.J., Govindaraju, M., Chiu, K.: Exploring the Design Space for CCA Framework Interoperability Approaches. In: *Workshop on Component Models and Frameworks in High Performance Computing*. (June 2005)
10. Chiu, K., Govindaraju, M., Gannon, D.: The Proteus Multiprotocol Library. In: *Proceedings of Supercomputing 2002*. (November 2002.)
11. Chiu, K.: XBS: A streaming binary serializer for high performance computing. In: *Proceedings of the High Performance Computing Symposium 2004*. (2004)
12. Swamy, M.: Improving Throughput for Grid Applications with Network Logistics. In: *Proceedings of Supercomputing Conference*. (2004)
13. SWIG: Simplified Wrapper and Interface Generator (1997) <http://www.swig.org>.
14. 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 (HIPS 2004)*, Santa Fe, NM, IEEE Press (2004)
15. Krishnan, S., Gannon, D.: XCAT3: A Framework for CCA Components as OGSA Services. In: *Proceedings of HIPS 2004: 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*. (April 2004)
16. Govindaraju, M., Bari, H., Lewis, M.J.: Design of Distributed Component Frameworks for Computational Grids. *Proceedings of International Conference on Communications in Computation* (June 2004) 160–166
17. 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. *CCPE* 14 (2002) 323–345