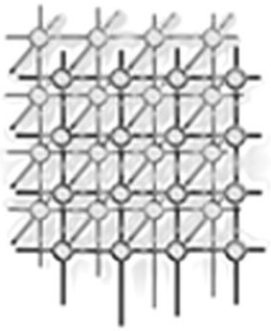
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2007; **19**:721–742
Published online 21 November 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1149

Productivity and performance through components: the ASCI Sweep3D application



Young Yoon*,†, James C. Browne, Mathew Crocker, Samit Jain and Nasim Mahmood

Department of Computer Sciences, Taylor Hall 2.124, 1 University Station C0500, Austin, TX 78712-0233, U.S.A.

SUMMARY

This paper is a case study of the effectiveness of component-oriented development for enhancing both productivity and performance for parallel programs. A process for converting monolithic applications into semantically composable components is described. The supporting software, the P-COM^2 compositional compiler, is briefly described. The componentized version of Sweep3D is described. Productivity is illustrated by composing different instances of the Sweep3D code through automated composition of components using P-COM^2. These instances, each of which targets improving performance for some execution environment or problem case, are examples of a family of instances which are composable from a modest set of components. It is found that customization of componentized codes by component-level adaptation may yield substantial performance improvement for specific execution environments. We identify and explain some of the benefits of component-oriented development for high-performance parallel systems. Copyright © 2006 John Wiley & Sons, Ltd.

Received 11 September 2005; Revised 20 October 2006; Accepted 22 October 2006

KEY WORDS: compositional development; P-COM^2; CODE; parallel programming; Sweep3D

1. INTRODUCTION

1.1. Motivation and goal

Component-oriented software development is one of the most active and significant threads of research in software engineering [1,2]. Use of component libraries has enabled a reduction in effort of several

*Correspondence to: Young Yoon, Department of Computer Sciences, Taylor Hall 2.124, 1 University Station C0500, Austin, TX 78712-0233, U.S.A.
†E-mail: agitato7@cs.utexas.edu

Contract/grant sponsor: Defense Advanced Research Projects Agency (DARPA); contract/grant number: NBCH30390004
Contract/grant sponsor: National Science Foundation (NSF); contract/grant number: ACI-0305644

orders of magnitude in application areas such as GUI development. However, there has been relatively little research on component-oriented development in the context of high-performance parallel programming and almost no detailed case studies comparing the performance of componentized codes to conventionally structured codes.

There are many motivations for raising the level of abstraction of program composition from individual statements to components with substantial semantics, and we give examples below.

- It is often the case that there is a family of applications that can be generated from a modest number of appropriately defined components. Thus, components are reusable.
- Multiple versions of components enable ready adaptation of parallel programs to different execution environments and application instances with minimal effort.
- Componentization allows the specialization of functionality and optimizations local to each component to be focused on.
- Programs generated and maintained as compositions of components have an understandable hierarchical structure and are, thus, readily modifiable and maintainable.
- The clean simple code structure engendered by a componentized structure approach leads to better performance even for sequential versions of the program.

The goal of this project is to demonstrate these benefits by applying concepts of component-oriented development to the Sweep3D [3] neutron transport code to enable the automated composition of applications that perform efficiently with respect to different execution environments and different cases of input data and structural models.

1.2. Sweep3D

Sweep3D [3] is a three-dimensional (3D) particle transport code that has been identified as an Accelerated Strategic Computing Initiative (ASCI) benchmark for evaluating high-performance parallel architectures. In this paper, the Sweep3D ASCI benchmark code is used to demonstrate the productivity and performance gains obtainable by the application of the compositional approach to the development of families of parallel and distributed programs. Further details on Sweep3D are given in Section 2.

1.3. Approach

The Sweep3D benchmark code has been re-factored into a set of components that are then encapsulated in the P-COM^2 [4] component specification language. The P-COM^2 compiler [4] composes parallel programs from components with interfaces in its component specification language. Interfaces specified in P-COM^2 incorporate information on behaviors and implementations of components. Inclusion of implementation information in the interface enables the compiler to automatically select components appropriate for specific execution environments or problem cases. Examples of these (associative) interfaces with descriptions are given in Section 3.2 The P-COM^2 compilation system can generate any of the following as final output: a serial program, an MPI program or a multi-threaded program for a shared memory multiprocessor.

The development process has two phases: component development and program instance development. The first phase begins with a domain analysis to identify the components and their

DOI: 10.1002/cpe

attributes and the relationships among the components. Serial code for each component is extracted by partitioning the original code into logical functional components and then turning these functional components into self-describing composable components by encapsulating them in the P-COM^2 component specification language. In the second phase, the programmer analyzes a particular problem instance and its execution environment to determine the attributes and attribute values that characterize them, and constructs an appropriate initial component from which the P-COM^2 compiler can generate the program for the application family instance given an appropriate component library. Section 3 illustrates the interfaces for several components and explains how the interfaces specify component behaviors and enable automated composition.

1.4. Experimental evaluation

We identify and report on experiments in composing systems that are appropriate for different execution environments and problem instances, including performance improvements by choosing components to optimize behavior in terms of performance or stability. The performance of the componentized code is compared with the original code in terms of speedup, efficiency and scalability. Detailed descriptions of the experiments are given in Section 4.

1.5. Related research

Section 5 summarizes related research on component-based development in the context of high-performance computing.

1.6. Conclusions

The experiments in composing instances of Sweep3D targeting different execution environments and data inputs demonstrate enhanced productivity in that many instances of the code which are efficient for a given execution environment and/or data input are readily generated with little effort. It is also found that componentization exposes opportunities for optimization which are difficult to find in the original monolithic code. Section 6 summarizes what has been demonstrated and learned as a result of this case study of compositional development from self-describing components.

2. SWEEP3D AND DOMAIN ANALYSIS

2.1. Overview of Sweep3D

The Sweep3D application [3] has received considerable attention as an important wavefront application. The benchmark code, written in Fortran and MPI, represents the heart of a real ASCI application. It solves a 1-group time-independent discrete ordinates (S_n) 3D Cartesian (XYZ) geometry neutron transport problem. The XYZ geometry is represented by a 3D rectangular grid of cells indexed as IJK. The angular dependence is handled by discrete angles with a spherical harmonics treatment for the scattering source. The solution involves two main steps:

(i) the streaming operator is solved by sweeps for each angle;
(ii) the scattering operator is solved iteratively.

DOI: 10.1002/cpe

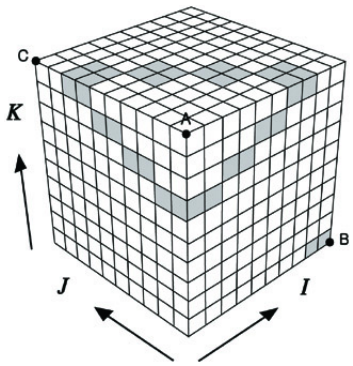


Figure 1. Example of the wavefront process in a $10 \times 10 \times 10$ data cube. This figure depicts a wavefront, shaded in gray, that originated from the unseen vertex in the cube, and is about to finish at vertex A. At the same time, a further wavefront is starting at vertex B and will finish at vertex C. Note that the example shows the use of a 5×5 grid of processors, and in this case each processor holds a total of $2 \times 2 \times 10$ data elements (data set of $10 \times 10 \times 10$).

The Sweep3D problem exploits parallelism through a wavefront process. The data cube is decomposed so that a set of processors, indexed in a 2D array, hold part of the data in the I and J dimensions, and all of the data in the K dimension. The sweep processing consists of pipelining the data flow from each cube vertex in turn to its opposite vertex. It is possible for different sweeps to be in operation at the same time but on different processors. The sweeps are computed for all octant directions in each iteration until convergence occurs, which may take many iterations. Figure 1 is a schematic of the decomposition and the wavefront processing.

2.1.1. Parallelism in Sweep3D

Sweep3D exploits parallelism via a wavefront process. There is a further type of parallelism that can be exploited in the Sweep3D problem by multitasking the loops in various components in the inner routine on shared memory processors. Particularly, in the 'ComputeFlux' component, multitasking parallelism can be exploited within each block of work (computational grid block) although this requires a special ordering of that work. Multitasking parallelism is not exploited in this case study because it requires a thread-safe MPI that was not available in our principal parallel execution environment.

2.1.2. Pseudo-code for control flow

Figure 2 gives pseudo-code for the control flow of the original parallelized code. Note for future reference that the loop nest is five deep with another level of single loops inside the five nested loops.

DOI: 10.1002/cpe

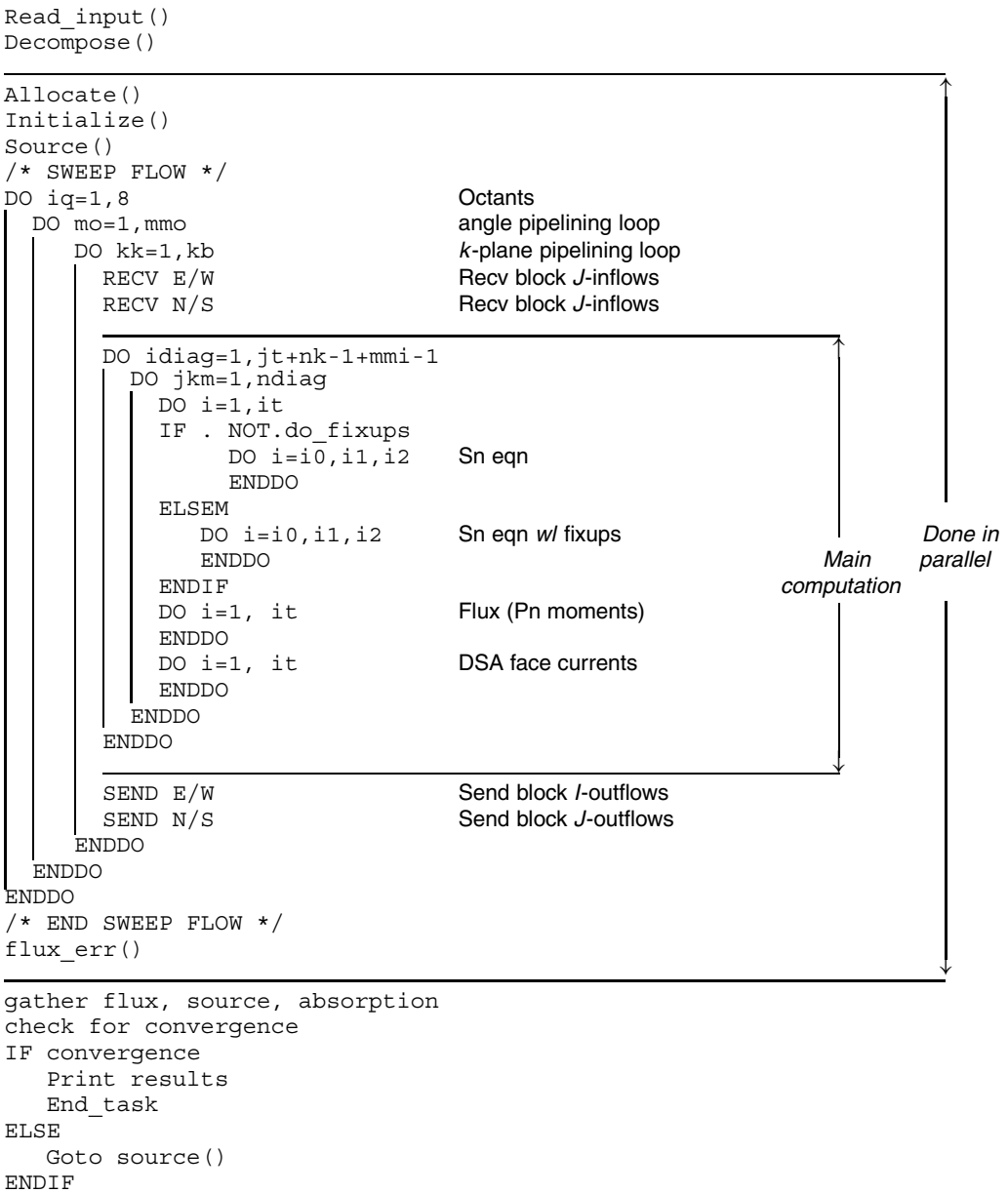


Figure 2. Pseudo-code for control flow in a Sweep3D application, depicting the part that can be parallelized and the main computation.

DOI: 10.1002/cpe

2.2. Domain analysis

Domain analysis [5] is the basis for identifying components from which a family of programs can be constructed and defining properties and behaviors of the components in terms of behavioral and architectural attributes. We have performed a domain analysis for the Sweep3D problem. There are three main types of components. The first type covers the *initialization*, *decomposition* and *allocation* components of the 3D grid and each are typically executed once at the beginning of the program. The second type of component arises in the main inner loop that includes the flux computation (which takes most of the computational time). Most of the communication between processors also takes place among these components. These components include computing source, octant loop, angle block loop, k-plane loop, receiving inflows from neighboring processes, computing flux and faces, sending outflows to the neighboring processes and checking for flux convergence. The third category includes components for gathering computational results, checking for convergence, printing diagnosis results and terminating the program.

After the domain analysis, we generated components as C functions (by converting the original Fortran), generated multiple implementations of some of these components and found other components in existing libraries. The components were then encapsulated in the P-COM^2 component specification language using the attributes identified in the domain analysis to specify associative interfaces for the components.

2.2.1. Logical components

The main logical components of the Sweep3D code were identified as follows.

- ReadInput: reads input given by the user.
- Allocate: allocates all of the arrays which represent the computational data needed by each process during its computation.
- Initialize: performs all of the initialization required before the start of the main outer iteration loop. In particular, it initializes cross sections, source, S_n directions and geometry. It also decomposes the problem grid onto the 2D processor grid.
- Source: computes the source moments in the beginning of each inner iteration. It also zeroes out the flux for the next iteration and saves the flux of the previous iteration to compute flux error.
- Octant: initializes data before beginning the sweep of angles for an octant. It coordinates sweep direction parameters and angle weights for an octant.
- AngleBlock: manages the angle blocks pipelined through the 2D processor array.
- K-planeBlock: manages the k-plane blocks pipelined through the 2D processor array.
- ReceiveInflows: receives flux inflow(s) from the neighboring processor(s) depending on the direction of the sweep.
- ComputeFlux: computes scattering of flux and is responsible for 70–80% of computational cost of Sweep3D problem.
- SendOutflows: sends flux outflow(s) to the neighboring processors depending on the direction of the sweep.
- FluxError: computes the flux error between the scalar flux computations of the previous and current iteration. Different algorithms can be used to compute flux error, e.g., relative max, least squares, etc. Checks convergence of the flux.

DOI: 10.1002/cpe

- GatherData: gathers results from each processor at the end of each iteration. It accumulates and prints results including flux error.

Each component represents a logical function. For example, communication is localized to the 'ReceiveInflows' and 'SendOutflow' components. This partitioning of functionality enables easy adaptation of the program's parallel structure.

2.2.2. *Attributes and properties*

Space limitations preclude a comprehensive specification and description of all of the attributes used to specify the behaviors and properties of Sweep3D components. Each component has a type attribute originating in the component analysis, a function within the type and one or more properties describing its behavior and implementation. For example, components involved in complex communication have an attribute 'communication mode' with the two values 'synchronous' or 'asynchronous'. The use of attributes and examples of interfaces are given and discussed in Section 4.2.

2.2.3. *Program structure and parallelism*

An important part of this procedure is to formulate parallelism in the componentized code. The data flow graph (dependence graph among the components) for the componentized Sweep3D code is shown in Figure 3. As discussed in Section 2.1.1, there are two types of parallelism possible in the Sweep3D program: (1) parallelism via domain decomposition and (2) parallelism via diagonal multitasking. The former is the main source of parallelism, scales well across distributed memory systems and, as mentioned previously, is the only form of parallelism used in this study. The system must be load-balanced to ensure optimum parallel processor utilization. The compositional approach allows us to implement different schemes for the mapping of components to processors to obtain load-balance. Most of the allocation of data and mapping of components is isolated to the 'Allocate' component, but can be dynamically modified. The memory requirement for the program and the various components depends on the size of the problem instance. Most of the computation in the program takes place in the 'ComputeFlux' component, which computes flux for a pipelined block of data; therefore, it becomes the main focus of optimizations local to components for different problem data sets and systems. The 'SendOutflows' and 'RcvInflows' components handle all of the communication in the iterative loop of the program, and are crucial to parallel performance. Localization of communication operations facilitates modification and thus optimization of the communication operations. The 'K-planeBlock' and 'AngleBlock' components handle the pipelining of blocks of work. Pipelining is crucial to parallel efficiency and the specialization of pipelining in a separate component allows a flexible and adaptable pipelining scheme for different cases. These components can utilize specific algorithms that are time- or space-efficient.

The degree of parallelism (number of component replicas) for a given single process multiple data (SPMD) component is specified via an index parameter in P-COM^2. The mapping in the code is done in a way so that all of the components on the same replication line (a replication line is a sequence of SPMD components where the replica for each component in the vertical direction is mapped to the same processor; see Figure 3, the data flow graph) are mapped to the same processor which ensures that communication between components on the same replication line are function calls and do not

DOI: 10.1002/cpe

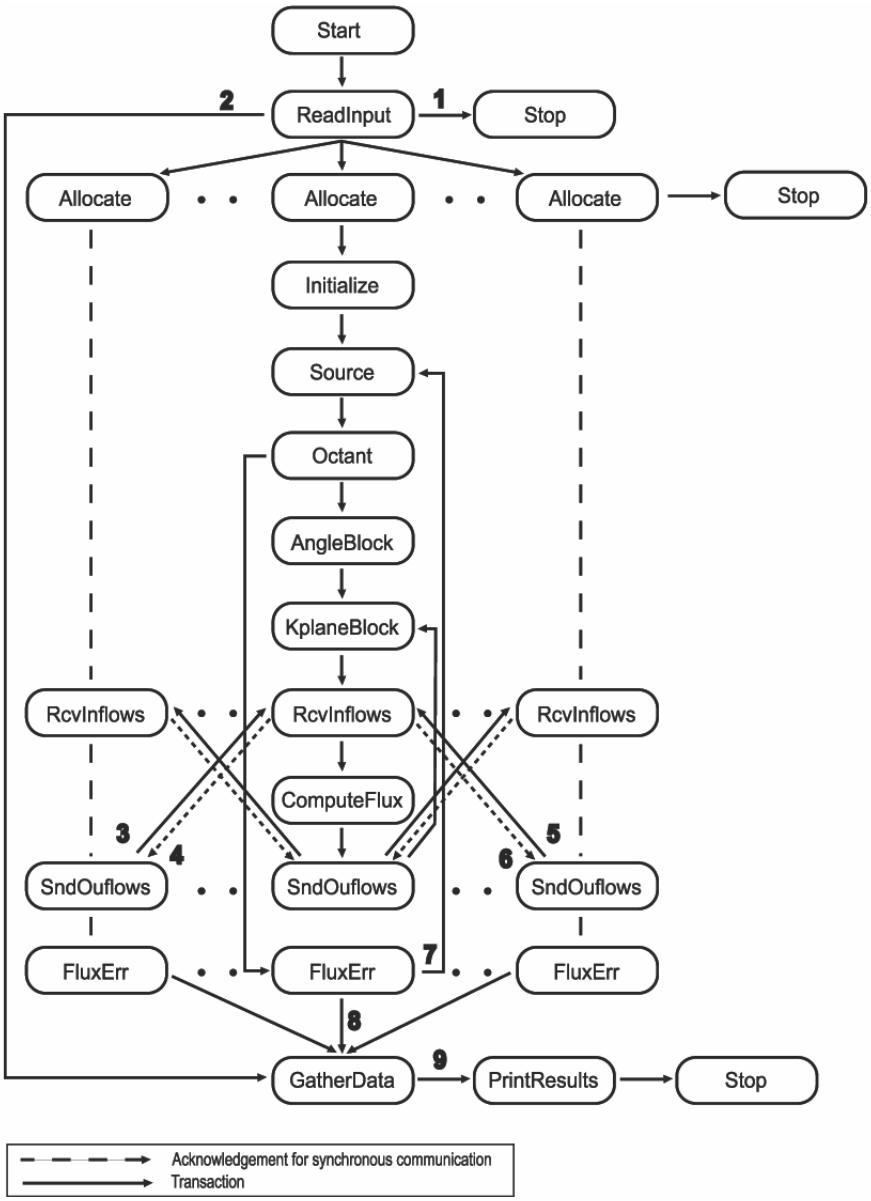


Figure 3. Data flow graph for the componentized Sweep3D application.

DOI: 10.1002/cpe

add to the communication cost. Most of the communication takes place between different processors during the sweep routine while receiving inflows/sending outflows from/to adjacent processors in the grid.

Communication can be overlapped with certain computations which can be moved outside the main diagonal loop, but this requires making copies of data and adding several loop nests which also increases code complexity. There is a space–computation tradeoff involved in this, but when memory is scarce, adding extra buffers might decrease performance due to increased memory traffic. Therefore, the system needs to be balanced with respect to the characteristics of the execution environment to give optimum performance.

2.3. Component adaptations and optimizations

There are a number of space–computation tradeoffs which can be applied to optimize the code for a given execution environment and problem specification. Some of them include the following.

- Double buffering the send/receive arrays for asynchronous communication.
- Overlapping communication with certain computations that can be moved outside the main diagonal loop, requiring multiple copies of data, thus more memory.
- Loop restructuring within each component for more efficient memory access.
- Duplicate invariant data as a local state in components or communicate through transactions between components.
- Size of angle blocking factor `mmi` and k-plane blocking factor `mk`.
- Use of specialized algorithms that are time- or space-efficient.
- Use of alternative caching mechanisms depending on available memory.

The deeply nested loop structure in the original program makes manual construction of optimized loop structures complex while parameterization for adaptation to execution environments and problem instances may lead to a complex code structure. Building specialized components and automating the composition of parallel programs by letting the compiler select efficient components for specific execution environments and problem instances is not only an effort-efficient approach, but also gives significant performance gains and scalability, as we will see in Section 4. Moreover, having specialized components allows for better management of the program. The compositional approach targets ready generation of many program instances and many execution environments.

3. COMPONENTS AND COMPOSITION

3.1. The P-COM^2 interface definition language and components

The P-COM^2 [4] component specification language incorporates information on behaviors and implementations of components to enable automated qualification of components for effectiveness in specific application instances and execution environments. A component is a serial program that is encapsulated by an associative interface that specifies the properties of the component. (Composition is recursive: composed components can be composed into larger components.) The composition implemented by the compiler is based on the matching of component specifications. (Section 3.2.2

DOI: 10.1002/cpe

gives examples of component specifications (associative interfaces) and a formal specification can be found in [4].) The interfaces and composition are illustrated in Section 3.2. The compiler can generate as output a program in the CODE [6] data flow graph format which can be mapped to any of the following: a serial code, an MPI code or a multi-threaded code for a shared memory multiprocessor.

3.2. Example components

3.2.1. Example 1: `accepts` and `requests` interface

A component specification consists of an `accepts` interface and a `requests` interface. An `accepts` interface specifies the set of interactions in which a component is willing to participate. It incorporates a profile which describes the properties and behaviors of the component including the differences between implementations of a given functionality, signatures for the set of transactions (functions) which it implements and a state machine to sequence the receipt of messages and initiation of interactions. A `requests` interface specifies the set of interactions that a component must initiate if it is to complete the interactions it has agreed to accept. It incorporates a set of selectors defining the profiles and transactions (function signatures) for the components it requires and may incorporate a state machine for sequencing the interactions.

Figures 4 and 5 show the interfaces implemented in 'RcvInflows' and 'SndOutflows' components. The profile in the `rcv_inflows` accept clause specifies: that this component instance belongs to the subdomain `sweep3d_inner_sweep`, that it implements the function `rcv_inflows` and that its communication with other components is synchronous. The transaction section specifies that this component implements three functions, `get_data`, `get_EW_flux` and `get_NS_flux`, and will accept invocations of these functions in any order. The first selector in the requests interface of `rcv_inflows` specifies that it requires a component implementing the function `compute_flux` in the subdomain of `sweep3d_inner_sweep` which does not implement multitasking. The transaction `get_data` will be invoked only when the condition on the local state variable `state`, '`state==3`', is true. The second and third selectors of `rcv_inflows` are composed with instances of `snd_outflows` whose profiles satisfy the selectors as shown in Figure 5.

3.2.2. Example 2: state machine implementation

There are often precedence or sequencing relationships between (1) transactions implemented (or accepted) by the component and (2) transactions requested by the component. Sequencing relationships among messages can make the correct implementation of asynchronous communication complex and difficult. The component structure where communication is isolated in a few interacting components simplifies the formulation of asynchronous communication.

Sequencing relationships among transactions accepted by the component are implemented through coding an implicit state machine by combining guards (expressions in propositional logic over attributes and state variables) on transactions with a precedence operator '>'.

Figure 6 shows the synchronous communication between `rcv_inflows` and `snd_outflows` in the inner sweep computation, where sequencing control is required. `snd_outflows` waits for an acknowledgement from `rcv_inflows` after every iteration so that coordination among the processes is maintained. The numbers on the arcs of Figure 6 are carried forward from Figure 3 (the data

DOI: 10.1002/cpe

```
computation node rec_inflows {
profile:
           string domain = "sweep3d_inner_sweep";
           string function = "rcv_inflows";
           string comm_pattern = "synchronous";

transaction:                                                          Accept Interface
           int get_data ( in int_array data1, ... );
           ||
           int get_EW_flux (in real_array phiib, in int sync);
           ||
           int get_NS_flux (in real_array phijb, in int sync);
var {
}
initial {
}
comp{
}
{selector:
           string domain == "sweep3d_inner_sweep";
           string function == "compute_flux";
           string multi_tasking == "false";
transaction:
        %{ state == 3}%
                 int get_data (out int_array data1);
}index [pid pid]

{selector:
             string domain == "sweep3d_inner_sweep";
             string function == "snd_outflows";
             string comm_pattern == "synchronous";
transaction:
        %{ state == 3, ns_rcv ! = -1}%
               int get_sync (out int sync);
}index [ns_rcv ns_rcv]                                                Request Interface

{selector:
              string domain == "sweep3d_inner_sweep";
              string function == "snd_outflows";
              string comm_pattern == "synchronous";
transaction:
        %{ state == 3, ew_rcv != =1}%
                 int get_sync (out int sync);
}index [ew_rcv ew_rcv]
}
```

Figure 4. Example code for the `accepts` and `requests` interface of the `rcv_inflows` component.

DOI: 10.1002/cpe

```
computation node snd_outflows {
profile:
            string domain = "sweep3d_inner_sweep";
            string function = "snd_outflows";
            string comm_pattern == "synchronous";

transaction:
            int get_data (in int_array data1, ...);
            ||
            int get_sync (in int sync);
.
.
.
```

Accept Interface

Figure 5. Example code for the accepts interface of the `snd_outflows` component.

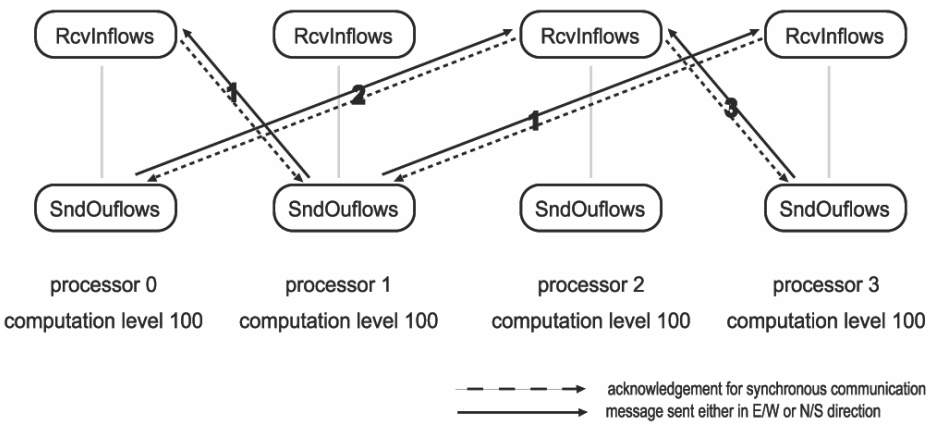


Figure 6. Synchronous communication in the inner sweep process.

flow graph). This synchronization ensures that all 'ComputeFlux' components remain in the lock step with respect to the iteration count (the iteration count is equal to the computation level in Figure 6).

On the other hand, if communication is asynchronous, SndOutflows may continue to another iteration in the flux computation and deadlocks can occur if messages are received out of order. Message ordering under asynchronous communication can be readily specified and enforced with a simple P-COM^2 state machine specification as shown in Figure 7. This state machine is for an asynchronous RcvInflows component that is to first receive data from the K-planePipeline component and then to receive two messages from east/west and north/west directions in sequential order. Figure 8 gives the implementation of this state machine in the asynchronous RcvInflows component in the P-COM^2 component specification language.

DOI: 10.1002/cpe

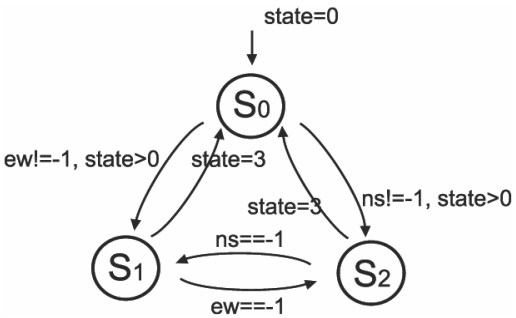


Figure 7. State machine to set the precedence of transactions and avoid deadlock.

```
computation node rcv_inflows {
profile:
               string domain = "sweep3d_inner_sweep";
               string function = "rcv_inflows";
               string comm_pattern = "asynchronous";
transaction:                  enabling condition
            %{state == 0}% int get_data (in int_array data1, ...);
            ||
            %{state > 0, ew_rcv != -1}% int get_EW_flux (in real_array phiib);
                  {% ew_rcv = -1; %}      actual transaction
            ||
            %{state > 0, ns_rcv != -1}% int get_NS_flux(in real_array phijb);
                  {%ns_rcv = -1; %}
                  action after transaction
```

Figure 8. State machine in Figure 7 implemented in the accept interface of the `rcv_inflows` component.

4. ADAPTATIONS AND OPTIMIZATIONS

4.1. Overview

This case study focuses primarily on optimizations that became readily visible when the code was componentized. Componentization provided the unexpected benefit of enabling ready identification of a number of serial code optimizations which were difficult to identify in the original code which has a six-deep loop structure and quite complex index computations. Each component has no more than two deep loop nests (the other loops become arcs in the data flow graph) so that the loop structures, index expressions and access patterns are easy to identify. This visibility of loop behavior enables ready adaptation and optimization of components. Obviously these optimizations could have been readily identified in the original code if it had been developed with a well-partitioned structure. However, it is sometimes the case that high-performance codes are not componentized because of a belief that the

DOI: 10.1002/cpe

extra overhead imposed by structures with separate address spaces leads to inefficient code. The results obtained here suggest that well-partitioned structures may lead to better performance.

Efficiency in computation

- Moving loop invariant computations outside the loops. These cases turned out to be easier to identify in the componentized code.
- Replacing complicated expressions with simple and cheaper expressions (or operators). These cases were also more readily identified in the componentized code where the loop structures are simple.
- Beneficial loop transformations are more readily identified in the simple loop structures of the components.

Efficiency in memory use

- Keeping frequently used items together in cache to exploit the spatial and temporal locality of reference. Locality is much easier to identify in simple loops with simple indexes.
- Marching forward instead of backward while accessing memory. Many systems have caches of predictive type that tend to read in successive cache lines.
- Column-major accessing with minimum stride length. Rewriting access patterns is simpler when the loops are simpler and the access patterns are simpler.

Efficiency in communication

- Sending/receiving the minimum amount of data since communication is the largest overhead. Thus, avoid sending constants or change variables infrequently.
- Using asynchronous communication by double buffering send/receive messages. Given the complexity of the communication patterns and the sequencing relations among messages, implementing asynchronous communication would have been difficult for the original code but was simple for the componentized code.

Basic loop restructuring as shown in Figure 9 has a substantial impact on the performance of the compute bound components (primarily ComputeFlux) since most of the computation and memory accesses in Sweep3D take place inside multi-dimensional loops in these components. Some of the optimizations are execution environment specific; in such cases, we can have separate implementations of the same component for different execution environments. The different implementations of a given functionality are differentiated in terms of the attributes defined in the domain analyses.

One of the main goals of the project was to demonstrate the performance obtainable from the compositional approach. Our performance analysis can be classified into two main sections:

- performance comparisons between the componentized Sweep3D and the original Sweep3D;
- performance improvements obtainable by replacing components with components tailored for some specific execution environment.

DOI: 10.1002/cpe

```
for (n = 2; n <= i_5; ++n) {
        i_6 = *it;
        for (i_ = 1;  i_ <= i_6; ++i_) {
          phi [i_] += pn[m + (n + iq * pn_dim2) *
                      pn_dim1] * src[i_ + (j + (k + n *
                      src_dim3) * src_dim2) * src_dim1];
        }
}
```

A

```
for (n = 2; n <= i_5; ++n) {
        _i = (j + (k + n * src_dim3) * src_dim2) * src_dim1;
        val = pn[m + (n + iq * pn_dim2) * pn_dim1];
        i_6 = *it;
        for (i_ = 1; i_ <= i_6; ++i_) {
          phi [i_] += val * src [i_ + _i];
        }
}
```

B

Figure 9. Example of a simple loop restructuring for serial optimization. **A** is before optimization and **B** is after optimization.

Most performance experiments were conducted on the Lonestar [7] cluster of Intel Xeon 3.06 GHz dual processors, each with 2 GB memory, 512 kB cache and 4.2 GB/s bus, at the Texas Advanced Computing Center (TACC). One set of experiment was conducted on the Optimonster system in the Computer Science Department which has AMD Opteron^{TM} [8] processors with better cache performance that the Intel processors of Lonestar.

4.2. Components-level serial optimizations

The original and componentized Sweep3D codes (using synchronous communication) are compared for serial and parallel performance. Table I gives the execution time and the parallel speedup for $100 \times 100 \times 100$ and $200 \times 200 \times 200$ problem sizes. Note that the speedups are computed by dividing the parallel execution times in the serial execution time for each version, not by the usual definition of dividing the parallel execution time into the best sequential execution time which in this case is the sequential execution time for the componentized code. If we had used the sequential execution time from the componentized code to compute the speedup of the original code, then the original code would have showed a slowdown for the two-processor case which did not seem fair. The serial version of the componentized Sweep3D program ran much faster than the original Sweep3D because of optimizations local to components, mostly basic loop restructurings. This improvement in serial performance is obtained despite the introduction of thousands of procedure invocations due to the componentizations. The parallel component-based code outperformed the original code for two and four processors for a $100 \times 100 \times 100$ problem size. However, as the number of processors is increased, the overhead from the increased number of procedure calls becomes significant as the amount of computation done by a component becomes small and the overhead from procedure calls is

DOI: 10.1002/cpe

Table I. Performance comparison between *synchronous original* code and *synchronous componentized* code on a fixed problem size of 100 × 100 × 100 and 200 × 200 × 200.

Number of Processors	Runtime (s)				Speedup			
	100 × 100 × 100		200 × 200 × 200		100 × 100 × 100		200 × 200 × 200	
	Original	Componentized	Original	Componentized	Original	Componentized	Original	Componentized
1	162.99	114.22	1712.00	817.00	N/A	N/A	N/A	N/A
2	80.55	61.69	927.74	485.30	2.02	1.85	1.85	1.68
4	34.37	34.36	352.72	257.20	4.73	3.32	4.85	3.18
8	18.70	20.30	179.50	130.70	8.11	5.69	9.54	6.25
16	8.70	12.85	94.52	87.90	18.70	8.89	18.11	9.29
32	5.40	10.30	40.31	46.00	30.13	10.98	42.47	17.76

larger than the saving in computational efficiency. This accounts for the inversion in performance as a fixed amount of computation is spread over more processors. Note that the componentized code retains its advantage for a larger number of processors for the larger 200 × 200 × 200 problem. The larger the problem the more scalable the componentized code becomes.

4.2.1. Synchronous versus asynchronous communication

Synchronous communication can impose a significant execution time overhead, especially if there is a difference in the performance of the processors or if the system is not well load-balanced. For such a situation, asynchronous (non-blocking) communications can be used to an advantage. In this project, we built separate components for asynchronous communications using a simple scheme. We overlap the pair of sends and the pair of receives and overlap communication for the next block with the work on the current block by double buffering the send/receive arrays. Table II shows performance comparisons and overheads between the two communication patterns for 100 × 100 × 100 and 200 × 200 × 200 problem sizes. Note that with asynchronous communication the component-based code is faster than the original code for all of the cases of the 200 × 200 × 200. Table III gives the performance of the asynchronous componentized code on a difference processor configuration, a set of two four-processor clusters with AMD Opteron processors where the two clusters are connected by 100 MB Ethernet. Table IV has the additional memory requirements for the double buffering for the implementation of asynchronous communication.

4.2.2. Memory versus communication tradeoff

A fraction of the data in the program remain constant or change infrequently. In a conventional MPI implementation, all data defined in a main program are automatically instantiated in all processes on all processors. The default approach in a component-oriented architecture is that only the data specifically needed in the component are instantiated in the component. This means that in a parallel

DOI: 10.1002/cpe

Table II. Performance comparison between *synchronous componentized* code and *asynchronous componentized* code on a fixed problem size of 100 × 100 × 100 and 200 × 200 × 200.

Number of Processors	Runtime (s)				Speedup			
	100 × 100 × 100		200 × 200 × 200		100 × 100 × 100		200 × 200 × 200	
	Synchronous	Asynchronous	Synchronous	Asynchronous	Synchronous	Asynchronous	Synchronous	Asynchronous
1	114.22	114.22	817.00	817.00	N/A	N/A	N/A	N/A
2	61.69	53.00	485.30	457.47	1.85	2.15	1.68	1.78
4	34.36	28.76	257.20	226.04	3.32	3.97	3.18	3.61
8	20.30	13.20	130.70	119.36	5.69	8.65	6.25	6.84
16	12.85	10.80	87.90	63.04	8.89	10.58	9.29	12.96
32	10.40	7.16	46.00	36.55	10.98	15.95	17.76	22.35

Table III. Performance of *synchronous* and *asynchronous componentized* code on AMD Opteron processors with larger cache size (1024 kB).

Number of Processors	Runtime (s)				Speedup			
	100 × 100 × 100		200 × 200 × 200		100 × 100 × 100		200 × 200 × 200	
	Synchronous	Asynchronous	Synchronous	Asynchronous	Synchronous	Asynchronous	Synchronous	Asynchronous
1	97.09	97.09	686.28	686.28	N/A	N/A	N/A	N/A
2	51.82	42.98	402.80	375.13	1.87	2.25	1.70	1.83
4	28.87	22.43	205.60	174.05	3.36	4.33	3.34	3.94
8	16.85	10.56	100.1	95.49	5.76	9.19	6.86	7.19

Table IV. Additional memory required theoretically with increasing problem size for asynchronous communication due to double buffering.

	Problem Grid size			
	50 × 50 × 50	100 × 100 × 100	200 × 200 × 200	300 × 300 × 300
Number of processors	1	2	4	6
Additional memory (kB)	50	200	1200	3150

implementation, data may need to be communicated across several components before being used. It is, however, straightforward to instantiate some or all of the invariant data as local state in components. Thus, the component-oriented approach enables consideration of space–time trade-offs with respect to storing or communicating invariant data.

Storing the data as a state in the components across multiple replications requires that we save multiple arrays in each of the communicating components but transfer them only once during initialization. If we have enough memory for a given problem instance, storing invariants as state

DOI: 10.1002/cpe

Table V. Performance comparison on a fixed problem ($100 \times 100 \times 100$): messaging invariants versus keeping invariants as local variables in each component.

	Runtime (s)		Speedup	
Number of processors	Messaging invariants	Invariants as a local state	Messaging invariants	Invariants as a local state
1	114.22	114.22	N/A	N/A
2	61.69	55.79	1.85	2.04
4	34.36	30.88	3.32	3.69
8	20.30	18.34	5.69	6.23
16	12.85	11.81	8.89	9.67
32	10.40	8.10	10.98	14.10

Table VI. Per processor additional memory required by keeping the invariants as local variables in each component.

	Problem Grid size			
	$50 \times 50 \times 50$	$100 \times 100 \times 100$	$200 \times 200 \times 200$	$300 \times 300 \times 300$
Number of processors	1	2	4	6
Additional memory (kB)	110	270	1400	5500

data can significantly reduce the communication cost, thereby improving overall performance. On the other hand, the memory required to keep all invariants as a state in components may be significant and for Sweep3D the memory requirement increases at an exponential rate as the problem size increases. We will also have to save data in each of the components involved in communication with other processors. As with synchronous versus asynchronous communication, instantiation of invariant data is a space–time tradeoff and we have to balance between communication delay and memory overhead for a given problem instance and execution environment.

Problem size and memory system properties determine the optimum trade-off between keeping invariants as a state in components and sending invariants in communication messages between components.

Table V shows some performance data for a problem size of $100 \times 100 \times 100$ that is small and does not impose a large memory overhead. Therefore, for such a small problem set, we can always use the approach of saving the invariants as a state in communicating components and benefit from less communication and, hence, improved performance. However, for larger problem sets, replicating invariant data may not possible and a more balanced approach may be needed. Table VI shows the increase in memory requirements with an increase in problem size.

DOI: 10.1002/cpe

5. RELATED RESEARCH

There has been relatively little research on component-based programming in the context of parallel and distributed programs and almost no systematic comparison between the performance of componentized and conventionally structured codes.

Darwin [9] is a composition and configuration language for parallel and distributed programs. Darwin uses a programmer-generated configuration script to compose programs from components. In our approach, P-COM^2 generates an initialization component and the compiler can choose the required components automatically.

H2O [10] is a component-oriented framework for the composition of distributed programs based on Web services. Triana [11] is a graphical development environment for composing distributed programs from components targeting peer-to-peer execution environments. G2 [12] composes distributed parallel programs from Web services through Microsoft .Net. Armada [13] composes distributed parallel programs specialized to data movement and filtering.

The Common Component Architecture (CCA) project [14] is a major research and development project focused on the composition of parallel programs from components. However, the goals of CCA are rather different from the goals of this project. The primary goal of CCA is to enable composition of programs from components written in multiple languages. CCA uses the BABEL interface definition language [15] to specify mapping across language specific data formats. CCA has developed interface standards. There are several frameworks including Ccaffeine [16], XCAT [17], SCIRun2 [18] and DCA [19] implementing the CCA interface specification system. The different implementations target different architectures and adopt different programming models. For example, Ccaffeine targets parallel architectures and adopts an SPMD model, XCAT targets distributed architectures and adopts the Grid model, SCIRun2 and DCA targets both distributed and parallel architectures and implement both SPMD and multiple processes multiple data (MPMD) models. Component composition is either graphical or through scripts and make-files. CCA components interact through two types of port. The first type of port is the `provides` port. It is an interface specifying the services components provide to other components. The second type of port is the `uses` port. It is an interface through which components specify other components that they require. These port types exhibit some similarities to the `accepts` and `requests` transaction specifications. However, the details and implementations are quite different, as we have focused on incorporation of the information necessary to enable composition by compilation. Users are responsible for implementing communication between replicated components that is not handled by the framework of CCA. Also, communication among SPMD components is not defined in the CCA standard.

ArchJava [20] annotates ports with provides and requires methods which help the programmer to better understand the dependency relations among components by exposing them to the programmer. The `accepts` and `requests` interfaces of a P-COM^2 component incorporate signatures as do ArchJava provides and requires. The `accepts` and `requests` in P-COM^2 interfaces also include profiles and precedence specifications carrying semantic information and enabling automatic program composition.

ASSIST [21,22] is probably the previously implemented programming system that is most similar to P-COM^2 in terms of the program structures it generates. ASSIST has a programming model in which program structures are defined as programmer specified directed graphs where the nodes can be sequential or parallel programs. Nodes may have an internal state. The semantics of arcs are

DOI: 10.1002/cpe

defined in terms of streams. Later extensions of ASSIST [23,24] have utilized Grid middleware for implementation. It also incorporates a runtime system that supports QoS contracts. P-COM^2 supplies a capability for predicting the performance of a given parallel program instance on a given architecture through performance modeling based on the unification of concrete and simulated execution.

SBASCO [25,26] is a parallel program development environment that integrates skeleton-based and component-oriented technologies. SBASCO has a fixed set of parallel skeletons that can be combined in a program. The goal of SBASCO is to enable the development of parallel programs that combine portability with performance. The central concept is incorporation of both application program concerns (composition) and implementation and configuration information in the Interface Definition Language (IDL). In addition, the SBASCO IDL includes an execution cost model that can be used to guide the degrees of parallelism in the skeleton constructs. The IDL compiler and its runtime system use this cost information to determine an effective execution structure. The SBASCO IDL, like P-COM^2, is based on domain analysis. The program structure and the component relationships are explicitly defined by the programmer while in P-COM^2, the compiler automatically generates the program structure.

PARDIS [27] enables composite SPMD parallel structures of CORBA-defined objects. The CORBA IDL is extended to include SPMD data arguments for objects. The PARDIS compiler translates the extended IDL to generate master–slave parallel program structures through the CORBA request broker.

6. CONCLUSIONS AND FUTURE RESEARCH

This report demonstrates the effectiveness of component-oriented development supported by automated tools in developing efficient implementations of instances of families of components. The significant observations include the following.

- Dozens of instances of a family of codes implementing the Sweep3D functionality were implemented from a modest set of components in a matter of months by two undergraduate students working part-time.
- Performance of the Sweep3D codes were generally improved by componentization.
- Customization to different execution environments can be accomplished by local changes to one or at most a few small components.
- Componentization enables ready identification of optimizations which are difficult to identify in conventionally structured applications.

The obvious conclusion is that componentization supported by automated tools is one of the most important sources for increased productivity for the development of families of codes for high-performance parallel computation.

Future research will include the following areas.

- Employ multi-thread safe MPI to enable implementation and evaluation of diagonal multitasking.
- Consider implementation of compiler transformations to enable coalescing of components after qualification, selection and composition have been completed and the resulting code structure permits.

DOI: 10.1002/cpe

- Extending the case studies to include a detailed study of runtime adaptation which is also supported by the P-COM^2 system.

ACKNOWLEDGEMENTS

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004. This research was also supported in part by NSF grant number ACI-0305644, 'Montage: An Integrated End-to-End Design and Development Framework for Wireless Networks'. Computer services and support were provided by the Texas Advanced Computation Center (TACC). Finally, the authors are grateful to the referees of the earlier versions of this paper for calling attention to related research that needed to be incorporated into the paper.

REFERENCES

1. Szyperski C, Gruntz D, Murer S. *Component Software—Byand Object-Oriented Programming* (2nd edn). Addison-Wesley: Boston, MA, 2002.
2. Heineman GT, Council WT. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley: Reading, MA, 2001.
3. The ASCII Sweep3D code, August 2005. http://www.llnl.gov/asci_benchmarks/asci/limited/Sweep3D/asci_Sweep3D.html [October 2006].
4. Mahmood N, Deng G, Browne J. Compositional development of parallel programs. *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, College Station, TX, October 2003 (*Lecture Notes in Computer Sciences*, vol. 2958). Springer: Berlin, 2003; 109–126.
5. Prieto-Diaz R. Domain analysis: An introduction. *Software Engineering Notes* 1990; **15**(2):47–54.
6. Newton P, Browne J. The CODE 2.0 graphical parallel programming language. *Proceedings of the 6th ACM International Conference on Supercomputing*, July 1992. ACM Press: New York, 1992.
7. Lonestar User Guide, September 2004. http://www.tacc.utexas.edu/services/userguides/lonestar/ [October 2006].
8. AMD Opteron processor, May 2006. http://www.amd.com [October 2006].
9. Magee J, Dulay N, Kramer J. Structuring parallel and distributed programs. *Software Engineering Journal* 1993; **8**(2):73–82.
10. Sunderam V, Kurzyniec D. Lightweight self-organizing frameworks for metacomputing. *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11 (HPDC'02)*, July 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 113–124.
11. Taylor I, Shields M, Wang I, Philip R. Distributed P2P computing within Triana: A galaxy visualization test case. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003.
12. Kelly W, Roe P, Sumitomo J. An enhanced programming model for Internet based cycle stealing. *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2003. CSREA Press: Athens, GA, 2003; 1649–1655.
13. Oldfield R, Kotz D. Armada: a parallel I/O framework for computational Grids. *Future Generation Computer Systems* 2002; **18**(4):501–523.
14. 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. *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, August 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999; 115–124.
15. Govindaraju M, Krishnan S, Chiu K, Slominski A, Gannon D, Bramley R. Merging the CCA component model with the OGSI framework. *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid (CCGrid2003)*, May 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 182–189.
16. Kohn S, Kumfert G, Painter J, Ribbens C. Divorcing language dependencies from a scientific software library. *Proceedings of the 10th SIAM Conference on Parallel Processing*, Portsmouth, VA, 12–14 March 2001. SIAM: Philadelphia, PA, 2001.
17. Allan B, Armstrong R, Wolfe A, Ray J, Bernholdt DE, Kohl J. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience* 2002; **14**(5):323–345.

DOI: 10.1002/cpe

18. Zhang K, Damevski K, Venkatachalapathy V, Parker S. SCIRun2: A CCA framework for high performance computing. *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*. IEEE Computer Society Press: Los Alamitos, CA, 2004; 72–79.
19. Bertrand F, Bramley R. DCA: A distributed CCA framework based on MPI. *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*. IEEE Computer Society Press: Los Alamitos, CA, 2004; 80–89.
20. Aldrich J, Chambers C, Notkin D. ArchJava: Connecting software architecture to implementation. *Proceedings of the 22nd International Conference on Software Engineering*, May 2002. ACM Press: New York, 2002; 187–197.
21. Vanneschi M. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing* 2002; **28**(12):1709–1732.
22. Aldinucci M, Danelutto M, Paternesi A, Ravazzolo R, Vanneschi M. Building interoperable Grid-aware ASSIST applications via WebServices. *Proceedings of the International Parallel Computing Conference (PARCO 2005)*, September 2005.
23. Aldinucci M, Campa S, Coppola M, Magini S, Pesciullesi P, Potiti L, Ravazzolo R, Torquati M, Zoccolo C. Targeting heterogeneous architectures in ASSIST: Experimental results. *Proceedings of the 10th International Conference on Parallel and Distributed Computing (Euro-Par 2004)* (*Lecture Notes in Computer Science*, vol. 3149), Danelutto M, Vanneschi M, Laforenza D (eds.). Springer: Berlin, 2004; 638–643.
24. Aldinucci M, Petrocelli A, Pistoletti E, Torquati M, Vanneschi M, Veraldi L, Zoccolo C. Dynamic reconfiguration of Grid-aware applications in ASSIST. *Proceedings of the 11th International Conference on Parallel and Distributed Computing (Euro-Par 2005)* (*Lecture Notes in Computer Science*, vol. 3648), Cunha JC, Medeiros PD (eds.). Springer: Berlin, 2005.
25. Díaz M, Romero S, Rubio B, Soler E, Troya JM. Using SBASCO to solve reaction–diffusion equations in two-dimensional irregular domains. *Proceedings of the International Conference on Computational Science*, vol. 2. Springer: Berlin, 2006; 912–919.
26. Díaz M, Rubio B, Soler E, Troya JM. SBASCO: Skeleton-Based Scientific Components. *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2004)*, La Coruña, Spain, 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 318–324.
27. Keahey K, Gannon D. PARDIS: A parallel approach to CORBA. *Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC 1997)*, Portland, OR, August 1997. IEEE Computer Society Press: Los Alamitos, CA, 1997; 31–39.

DOI: 10.1002/cpe