Developing a component-based framework for subsurface simulation using the Common Component Architecture

Bruce Palmer[1], Yilin Fang[1], Vidhya Gurumoorthi[1], Glenn Hammond[1], James Fort[1] and Tim Scheibe[1]

Pacific Northwest National Laboratory,
PO Box 999 Richland WA 99352

E-mail: bruce.palmer@pnl.gov

Abstract. This paper will provide a brief overview of two frameworks being developed for subsurface simulations. The first is based on the Smoothed Particle Hydrodynamics algorithm and is designed to model flow and transport at the pore scale. The second is based on the Subsurface Transport Over Multiple Phases subsurface continuum code (STOMP) and is designed to simulate flow at the Darcy scale. Both frameworks have been built using the Common Component Architecture toolkit. These frameworks will eventually be combined into a single framework for performing hybrid multiscale simulations that seamlessly integrate both the particle and continuum simulations together.

1. Introduction

The increasing complexity of computer simulations, coupled with continuing changes in libraries, algorithms, computer architectures, and operating systems, is making the process of developing and maintaining code to perform high-end computations increasingly difficult. The traditional model of monolithic code development, in which all aspects of programming are handled by a relatively small and homogeneous group of programmers, is breaking down in the face of increasingly complex simulation requirements related to communication, I/O, linear algebra, grid management, visualization, solvers, etc. This paper will describe the decomposition of two codes into components using the Common Component Architecture toolkit[1]. This includes a Smoothed Particle Hydrodynamics (SPH) code, based on a Lagrangian formulation of the hydrodynamic equations [2], that is used for pore scale simulations of fluid flow in topologically complex configurations and a Darcy-scale continuum code used to simulate subsurface flow using finite volume numerical techniques[3,4]. The long term goal of this project is to couple the Lagrangian and continuum approaches into a single hybrid multiscale simulation of subsurface flow.

The components themselves are patterned after principles developed in object-oriented programming. Individual components represent a collection of subroutines or functions along with some internal data representing state. Data is communicated between components using explicit calls, no data is shared between components by sharing common data structures. The CCA toolkit itself supports this encapsulation, as well as providing other user services such as runtime configurability and language interoperability[5].

2. Reducing dependencies between components

A key feature of our strategy for creating components with minimal dependencies between them is to use a data dictionary and associated metadata to allow individual components to identify necessary data from a common data pool without explicit exchanges of information. At the start of the simulation, all components that generate data arrays that might be used by other components allocate memory for these arrays and deposit pointers to them in a data manager, along with associated meta-data. The meta-data consists of attributes that allow other components to determine whether the data is something that they can use. The combination of a data pointer and associated meta-data comprises an entry into the data dictionary. All entries in the data dictionary are interpreted in the same way by components in the application. For example, a data element representing the x velocities of particles located on a processor will be interpreted as such by all components. Some components, such as the component responsible for performing the time integration of particle coordinates, may recognize the x velocity as something it particularly needs to execute its function, but other components, such as the component for exporting a system snapshot, may treat the x velocity as just another particle property that should be included in the output.

A second mechanism for reducing dependencies is to provide input to each component directly from the user, instead of having all the input directives read by a single "driver" program and then delivering them to other components by passing them through the call stack. Alternatively, the driver can store the directives in globally accessible variables, but both methods can lead to dependencies between otherwise unrelated components. Having each component read its input directly from an external source eliminates these dependencies, although it may be cumbersome to require multiple independent files to run a single simulation. To simplify the input and allow users to include all input directives in a single file, a "read input" component was created that reads an input file that has been divided up into blocks with each block directed at a single component. The individual blocks have the form

```
input component1
  ….
  ….
end
```

Component 1 can retrieve the input data between the delimiters "`input`" and "`end`" by making a request to the input component for the block of input text corresponding to "`component1`". This text block can then be parsed using whatever mechanisms the author of component 1 wishes to use. The input supplies a number of functions that can be useful for parsing input, but if other code already exists for reading the input, it can be used instead. The input component itself is fairly neutral with respect to input formats and, apart from requiring the delimiters `input` and `end`, supports any kind of input (e.g. keyword-value, formatted, etc.).

The elimination of dependencies using these two concepts is best illustrated by looking at the remapping component in the SPH framework. The remapping component starts with data associated with particles, which may be randomly distributed in space, and interpolates the data onto a regular grid so that it can be exported in a format used by standard visualization software. To execute the remapping function, the component needs the x, y, z coordinates of all particles and it gets these from the data manager by looking for entries called '`xcoord`', '`ycoord`', '`zcoord`'. These arrays are *required* by the remapping component in order for it to work so it looks for them specifically in the data manager. Everything else that the remapping component uses is specified by the user. The user may specify in the remap input block that they want to visualize the velocity vector with components '`xvel`', '`yvel`', and '`zvel`' as well as the concentration field '`conc`'. After parsing the input block and extracting these names, the remap component will then check the data manager to see if there are fields with these names available. The remap component then caches pointers to these fields and will write the corresponding data to a visualization file whenever called upon to do so.

Most components contain an initialization method, which is called at calculation startup and allows each component to find the data it will use in the remaining simulation. Most components also contain

a “do something” method that causes the component to execute its function whenever it is called. In the case of the remap component, this method causes the remap component to interpolate the user specified data onto a regular grid and then write the values out to a file for subsequent analysis and visualization.

3. The Smoothed Particle Hydrodynamics framework

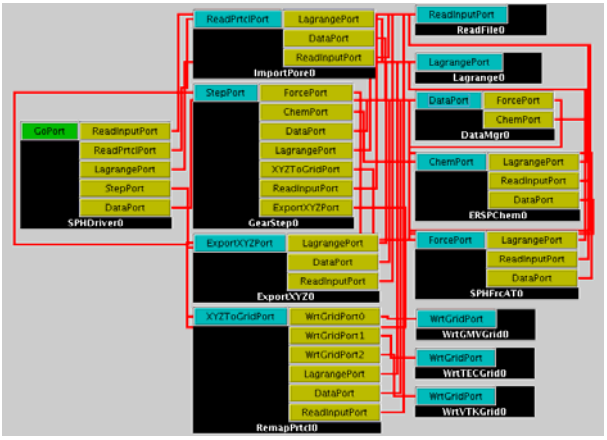

Figure 1. Schematic diagram of SPH application built from components.

The SPH framework is fairly well-developed and supports considerable functionality. It is also mature enough that new physics and chemistry can be added to it by focussing on modifying only a few routines. The remainder of the components in the framework can remain untouched. A “wiring diagram” of an SPH application is shown in Figure 1.

The diagram shows a driver component at the left that is responsible for controlling overall program flow, a set of components in the middle that are responsible for reading and writing particle configurations and integrating the particle equations of motion and a variety of components on the right that provide support functions and define the model physics and chemistry. The Lagrange component is the communication layer used to parallelize the SPH application. It is built with the Global Arrays communication library[6] and is used by a large number of the other components, the ReadFile and DataMgr components supply the input and data manager capabilities described above, and the ERSPChem and the SPHFrcAt components are responsible for implementing the chemistry and physics models that the application is simulating. Adding new models to the framework should only require modifications or new versions of the force and chemistry components.

An example of a simulation performed using the SPH framework is shown in Figure 2. This simulation contains approximately 14 million particles and represents transport of a non-reactive contaminant through a porous medium generated from a Monte Carlo simulation of 250 hard spheres. The figure shows an isosurface of concentration after the contaminant has traversed approximately halfway through the simulation cell (flow is in the z-direction). These simulations are being used to determine the effective transport constants for contaminant transport in subsurface simulations. Additional models that are under development include a simulation of uranium transport in porous media that includes the effects of intragranular diffusion.

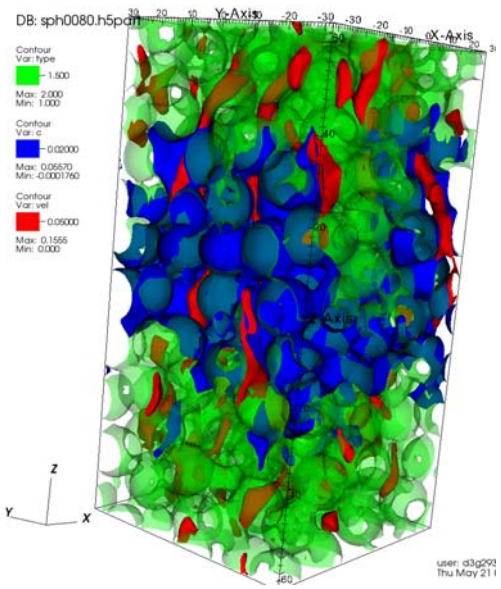

Figure 2. Simulation of non-reactive contaminant transport using SPH framework. Porous media surface is in green, isosurfaces of velocity are red and isosurfaces of concentration are blue. The visualization is done using the VisIt parallel software package.

4. The STOMP framework

Work has also been underway to develop a component version of the Subsurface Transport Over Multiple Phases (STOMP) subsurface simulation code[3,4]. This is a continuum code designed to perform simulations at the Darcy scale. The degree of componentization is not nearly as advanced as for the SPH framework, but a major objective has been achieved by splitting out the grid from the rest of the STOMP code as a separate component. At present, the STOMP framework contains only two components. One is the grid component and the second is a physics component that contains the rest of the STOMP code.

The grid component contains much of the input associated with problem setup and is responsible for creating the grid and adding all the fields required by the simulation to the grid. The grid component supports an unstructured syntax, although currently it is only implemented for a structured grid. The data mapping block refers to the necessity of mapping from a structured grid to an unstructured syntax. At some point in the future, if the grid component is reimplemented using an unstructured grid library, it should be possible to plug this into the existing framework with only modest modifications to the remaining components. Several additional components in the remaining STOMP code have been identified and work is underway to separate these from the main code. They include components for physics, chemistry, solvers, input, and output. The existing STOMP framework with possible future components is illustrated schematically in Figure 3.

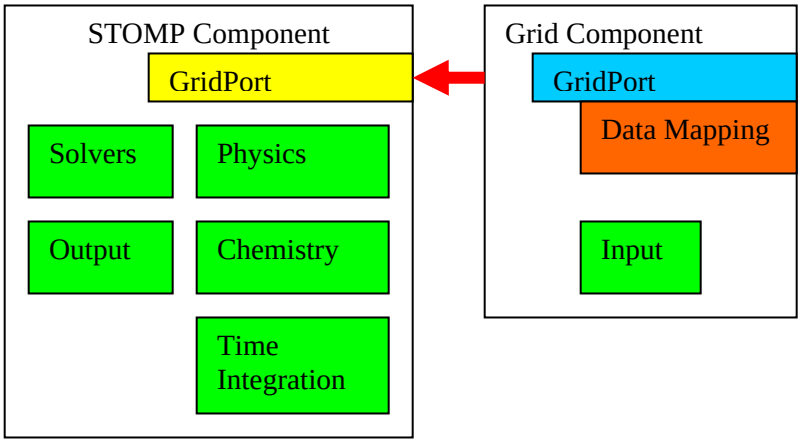

Figure 3. Schematic diagram of STOMP framework. Green boxes indicate future components that are currently contained within existing components.

A simulation of contaminant transport at the Hanford IFRC site performed with the existing STOMP framework is shown in Figure 4. The figure shows the migration of a plume through an array of monitoring wells.

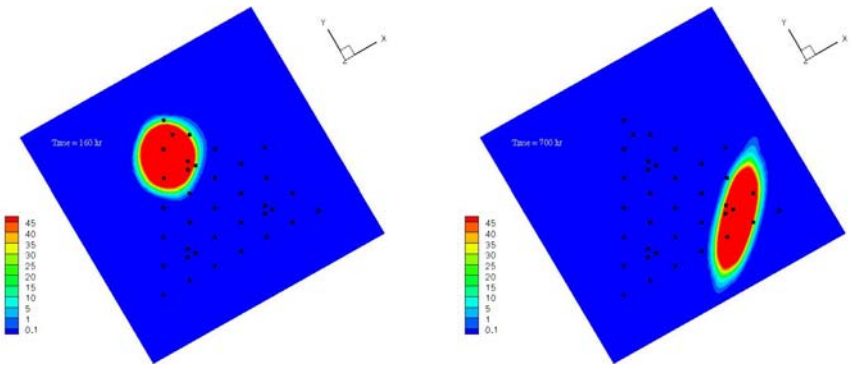

Figure 4. STOMP simulation of contaminant transport through an array of monitoring wells at the Hanford IFRC site.

Although the existing STOMP framework does not make use of an explicit data manager, the grid component can act in a similar way and many of the data concepts outlined above have been incorporated into it.

5. Summary

The frameworks described above demonstrate the feasibility of creating component-based applications that have a high degree of functional encapsulation, minimal dependencies between components, and still maintain high performance. An example of how this can increase programmer productivity is the development in the SPH framework of multiple force and input components that can be interchanged with each other without modifications to other components. The frameworks were implemented using the CCA toolkit, which allows users to configure the application at runtime and supports language interoperability between components. The SPH framework has exploited this feature by having components written in F90, C, and C++.

The componentization of the individual SPH and STOMP frameworks is expected to support the next stage of this project, which is to integrate these two simulation approaches into a single hybrid multiscale simulation that couples a pore scale description using the SPH algorithm with a Darcy scale simulation using STOMP. This will require extensions to the existing data model and a more robust version of the data manager, as well as development of appropriate coupling components that will link the two models together into a single application.

Acknowledgments

The authors are indebted to the CCA development team for its support and help on issues related to building and using the CCA toolkit as well as to Karen Schuchardt for useful discussions. This research is supported by the U. S. Department of Energy's Office of Science under the Scientific Discovery through Advanced Computing (SciDAC) program. A portion of this research was performed using the Molecular Science Computing Facility at EMSL, a national scientific user facility sponsored by the Department of Energy's Office of Biological and Environmental Research and located at Pacific Northwest National Laboratory.

References

[1] Allan BA, R Armstrong, DE Bernholdt, F Bertrand, K Chiu, TL Dahlgren, K Damevski, WR Elwasif, TGW Epperly, M Govindaraju, DS Katz, JA Kohl, M Krishnan, JW Larson, S Lefantzi, MJ Lewis, AD Malony, LC McInnes, J Nieplocha, B Norris, SG Parker, J Ray, S Shende, TL Windus, SJ Zhou. (2006) A Component Architecture for High-Performance Scientific Computing *Int. J. High Perf. Comput. Applications* **20** 163-202.

[2] Monaghan, JJ. (2005) *Smoothed particle hydrodynamics Rep. Prog. Phys*. **68** 1703-1759.

[3] White, M. D., and M. Oostrom (2000), STOMP Subsurface Transport Over Multiple Phases Version 2.0 Theory Guide, PNNL-12030, 235 pp., Pacific Northwest Natl. Lab., Richland, Wash.

[4] White, M. D., and M. Oostrom (2006), STOMP Subsurface Transport Over Multiple Phases Version 4.0 User's Guide, PNNL-15782, 120 pp., Pacific Northwest Natl. Lab., Richland, Wash.

[5] Kohn, S, G Kumfert, J Painter, C Ribbens, Divorcing Language Dependencies from a Scientific Software Library, 10th SIAM Conference on Parallel Processing, Portsmouth, VA, March 12-14, 2001.

[6] Nieplocha, J, B Palmer, V Tipparaju, M Krishnan, H Trease, E Apra. (2006) Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit *Int. J. High Perf. Comput. Applications* **20** 203-231.