SAND--96-8610C

POET (Parallel Object-oriented Environment and Toolkit) and Frameworks for Scientific Distributed Computing

CONF-970112--

Rob Armstrong

Alex Cheung

Distributed Systems Research

Sandia National Laboratories

Livermore, CA 94551

rob@ca.sandia.gov

RECEIVED
JAN 21 1997
OSTI

Abstract

Frameworks for parallel computing have recently become popular as a means for preserving parallel algorithms as reusable components. Frameworks for parallel computing in general, and POET in particular, focus on finding ways to orchestrate and facilitate cooperation between components that implement the parallel algorithms. Since performance is a key requirement for POET applications, CORBA or CORBA-like systems are eschewed for a SPMD message-passing architecture common to the world of distributed-parallel computing. Though the system is written in C++ for portability, the behavior of POET is more like a classical framework, such as Smalltalk. POET seeks to be a general platform for scientific parallel algorithm components which can be modified, linked, "mixed and matched" to a user's specification. The purpose of this work is to identify a means for parallel code reuse and to make parallel computing more accessible to scientists whose expertise is outside the field of parallel computing.

The POET framework provides two things: 1) an object model for parallel components that allows cooperation without being restrictive; 2) services that allow components to access and manage user data and message-passing facilities, *etc.* This work has evolved through application of a series of "real" distributed-parallel scientific problems. The paper focuses on what is required for parallel components to cooperate and at the same time remain "black-boxes" that users can drop into the frame without having to know the exquisite details of message-passing, data layout, *etc.* The paper walks through a specific example of a chemically reacting flow application. The example is implemented in POET and we identify component cooperation, usability and reusability in an anecdotal fashion. The following conclusions are drawn:

1) Specifically, POET components not only execute their assigned tasks, but provide data dependency information allowing components to cooperate on data decomposed across distributed machines.

2) Generally, frameworks for parallel scientific computing have an advantage over parallel tools using a more imperative style of programming like compilers or subroutine libraries. Because frameworks are "aware" of components they contain, information can be gathered ahead of actual execution of the component task and more sophisticated latency for bandwidth tradeoffs can be made.

19980406 081

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER

1. Why Frameworks for Parallel Computing?

Frameworks for parallel computing have recently become popular as a potential way to encapsulate and preserve parallel numerical algorithms. Because parallel numerics are orders-of-magnitude more complex and machine dependent than their serial counterparts, the motivation for preserving working implementations is strong. Currently, most practitioners of parallel computations write all of the component numerical implementations customized for each application. If a new application is to be written then all of the component numerical operations must be rewritten to suit the new application. In the past scientific subroutine libraries, written in, and for, use in FORTRAN, were able to provide the functionality necessary to encapsulate numerical algorithms for serial machines. For a variety of reasons, this familiar model for computational code preservation will not do for parallel computing:

1. The data needed by the subroutine is decomposed on the participating processors. All of the data cannot be passed through the argument list, as on a serial machine. This means the user must be cognizant of the data decomposition and provide this information to the routine. Minimally, the user may not attempt to access data present on another processor.
2. The subroutine cannot "tell" the system what data has to be present on a processor in order for it to perform its function. "Asking" for the data after the subroutine is called is usually inefficient, especially on distributed systems.
3. Because parallel computation is essentially real-time computing and is message-passing event driven, the user can either unwittingly stall event handling or, at least, must be made aware of the details going on below his/her code.

To paraphrase Albert Einstein, code reuse for parallel computing should be as simple as possible but not simpler. A model of parallel computing that makes a parallel machine look serial or casts the application as a sequence of subroutine calls is too simple to put the unnecessary details of processor/data decomposition out of the user's way.

1.1 Defining a Framework

The word framework is so overloaded to mean so many different things that it often conveys only a vague idea of a software "system." For the purposes of this work, framework will be defined as a system that provides services to components. The components are created *via* a template designed for cooperation among themselves and the framework system. Frameworks have long been accepted in areas where the user does not want to acquire the knowledge necessary to be in total control of the software system. A framework provides an environment that "takes care" of the top-level and provides a frame within which user-modified components can be placed. In addition to this environment, the framework typically will come with an assortment of pre-made components with which the user can base his application. Examples of frameworks are GUI's: X Window System Toolkit, and programming languages: Smalltalk and Visual Basic. Note that a framework need not be object-oriented: the X toolkit and Smalltalk are object-oriented, Visual Basic™ is not. The key to each of these frameworks is that components nested in the frame are easily modifiable by the user and new components are easy to create from old components or from scratch. [An exception: Visual Basic™ actually does not have a way, within the language, of creating new components.]

An application implemented in a framework consists of the framework itself, providing predefined services to the components, and a user-chosen set of components linked together by a means provided by the framework. There are two distinct phases to a frame-based application:

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

1. *The framing phase*: in which the components are selected and instantiated and the links between them created. It is at this point that the user creates or modifies components that will specialize their application. This phase can be thought of as "programming" the framework: like creating a script.
2. *The execution phase*: when control is handed over to the framework itself which in turn drives the framed components to do what they are framed to do.

The framing and execution phases may often be mixed together, some components may as part of their execution do some additional framing to be executed later, but the two phases are distinct and is what gives frameworks their distinctive advantage to parallel computing. As opposed to a standard imperitive style of computing, frameworks are to a greater or lesser degreee "aware" of what its constituent components are going to do before they are asked to do it. There is no opportunity for the user to stall message-passing events. The framework system lives along side the components during execution so that data dependencies can be queried and re-queried. In Section 4 an example will present that shows how components for parallel computing describe their data dependencies to the managing framework *before* an computing takes place.

1.2 Review of Existing Frameworks for Parallel Computing

POOMA[1] (Parallel Object Oriented Methods and Applications) and POET[2] (Parallel Object-oriented Environment and Toolkit) are examples of parallel frameworks from Los Alamos National Laboratory and Sandia National Laboratories respectively. Currently, these frameworks are being merged so as to share components. The discussion here is general enough to apply to both of these systems, but the author's experience and familiarity with POET will necessitate its use for the examples. While sharing the same philosophy, POET seeks to be a general framework for parallel algorithms, POOMA's approach is more toward specific applications. Though both packages are written in C++, POOMA tends to exploit and rely more heavily on the C++ concepts of operator overloading and templates. POET looks more like a "classic" framework in the spirit of Smalltalk. Both frameworks are oriented toward building applications in a modular fashion for code reuse. While POOMA is oriented toward particular applications implemented on parallel machinery, POET is oriented toward parallel algorithm implementations that are important to a variety of applications. POOMA focuses on numerical component reuse across problem domains. It can be said, for example, that POOMA has a component for Molecular Dynamics (MD) and POET has a component for equation solving, while it is simultaneously true that an MD application is implemented in POET and POOMA contains equation solvers for the work it needs to do. Though POET and POOMA are the closest to objective of this work there are related parallel computing tools that are useful to mention here.

The Portable, Extensible Toolkit for Scientific Computation[3] (PETSc) is a software library written at Argonne National Laboratory for the solution of partial differential equations on high-performance computers. As a complete rewrite of previous versions, PETSc 2.0 is written in ANSI C, employs the MPI standard for all message-passing communication, and is usable from Fortran, C, and C++. PETSc 2.0 incorporates a hierarchical set of abstractions in the form of software modules (e.g., matrices, linear and nonlinear solvers, time steppers, etc.), which are organized via encapsulation and

---

[1] To appear in *Parallel Programming in C++*, MIT Press.

[2] http://glass-slipper.ca.sandia.gov/~rob/poet.

[3] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith, *PETSc 2.0 Users Manual*, Technical Report ANL-95/11, Argonne National Laboratory, November, 1995.

polymorphism. In contrast to the framework approach of POOMA and POET, PETSc application programs are procedural based, where the user interface is a uniform set of routine calls for each library component. PETSc provides many facilities related to linear algebra and matrix manipulation and, similarly to POET, seeks to provide a platform for parallel algorithms.

Legion[4] is an OO framework that orchestrates parallel computing on a wide-area network, providing an object model for scheduling (as a superset of load balancing), object naming and location, object security, *etc.*, on world-wide metacomputer. Legion does *not* concern itself with parallel algorithms and application domains as do the other frameworks mentioned above. It is probably most similar to a CORBA (Common Object Request Broker Architecture) but built with high-performance computing in mind. Though Legion is based on C++, its object model, like POET, is similar to a "classical"

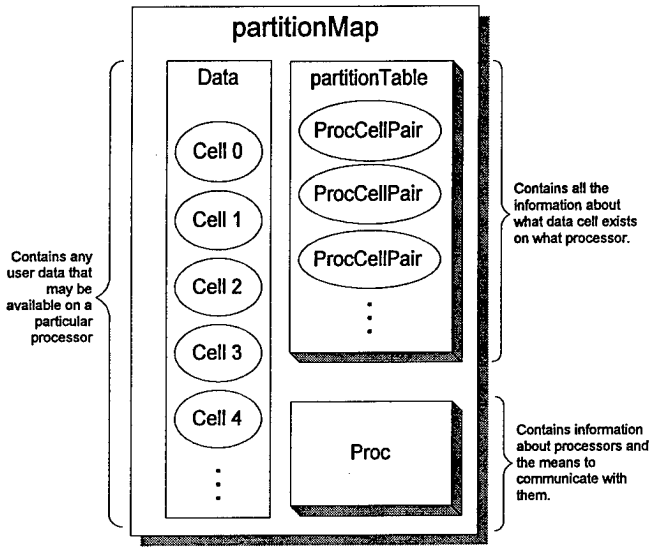

Figure 1

Diagram of the important services the POET Framework provides. The services are available through a single C++ class called *partitionMap*. Data of arbitrary type and dimension are contained in the data *Cell*'s. Not all *Cell*'s will be available on all processors. The *partitionTable* contains the decomposition relating which cells are on what processor. The *partitionTable* contains *procCellPair* entries, identifying which cell exists on which processor. Every component can expect that each processor has an up-to-date version of the *partitionTable*. The *partitionMap* also provides communication and processor information.

SmallTalk-like framework. Legion is complementary to the other frameworks to the extent that any of PetSC, POET, and POOMA could themselves be "framed" inside of Legion, exploiting Legion services to facilitate computing in the large.

2. The POET Framework

Though the purpose of this paper is to make the case for parallel computing frameworks in general, what follows is a sketch of the POET framework as an example. The purpose of POET is to encapsulate parallel algorithms similar to the way scientific subroutine

---

[4] *Legion: The Next Logical Step Toward a Nationwide Virtual Computer*, Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, Paul F. Reynolds Jr., Proc. of High Performance Distributed Computing 5, Aug. 5, 1996.

libraries are used in serial architectures. As was mentioned, a framework must provide services to constituent components. As part of the design these components must also follow a "template" interface to interact with the framework services and peer components. For object-oriented frameworks, like POET, this translates into an object model for the components to be framed. The user is expected to instantiate components and link them together using this object model to create a parallel application. The POET model is entirely SPMD, though it is useful to identify a "host" and "worker" processors for most applications. Many pre-made POET components support this paradigm. As opposed to other parallel tool-kits, POET makes no attempt to hide the fact that the application is running on a parallel machine.

2.1 POET Services

In POET the services are provided by the *partitionMap*[5] class (see Figure 1). This class packages user-defined data into "cells." A *Cell* is the smallest division of data with which POET will concern itself. *Cell*'s can contain arbitrary amounts and types of data and represent the smallest granularity of the problem. The *Cell* should contain enough data such that the work needed to process the data is large enough to amortize the framework overhead. This has to be defined by the user, who is the only expert on what their code is doing. *Cell*'s are then distributed to various processors either by means of a default decomposition or by components particularized by the user for the user's task. Objects are particularized by the use of C++ virtual functions. The *partitionMap* also contains information about which *Cell*'s are on what processor (*partitionTable*) and the means to communicate *Cell*'s or parts of *Cell*'s to other processors that require data contained in them (*proc*).

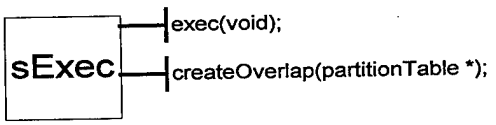

Figure 2

The *sExec* is an object, signified by a rectangle, that is expected to operate on the data. Everything a POET application does will be done by a derived *sExec* overloading the two methods the *sExec* class has: *exec(void)* and *createOverlap(partitionTable*)*. *sExec* inherits *Exec* to provide information to the framework system about what data is needed for the component to do its work. The *createOverlap* method adds entries to the partitionTable identifying data that has to be on a particular processor.

2.2 POET Object Model

In addition to these services, the framework provides various pre-made components that act on the data and provide ways for user code to act on the data. The fundamental component in POET is called an "*Exec.*" This is a pure virtual class that has one method on it called "*exec(void),*" which means: "do something; its your turn." All components that form the application, and participate in the framework, will inherit from this class and overload the *exec()* method. Because components almost always operate on user data, information must be provided by the *Exec* about the type and quantity of data needed to perform its function. Indeed, if the framework could access

[5] Italics denote the names of actual POET data types.

only this *exec()* method alone, then POET could do nothing more than the standard imperative scientific subroutine libraries of the past. A derived *Exec* is provided for this purpose called *sExec*; so-named because it associates or "scopes" cells or parts of cells to perform its task. It adds a new virtual method to be overridden called *createOverlap*. This method examines the given *partitionTable* and adds entries needed by the *sExec* to perform its *exec()* task. The *sExec* is the fundamental building block for POET applications.

Another important kind of *sExec* is one that exists to message and maintain other *sExec*'s. A common example is the *bulletinBoardExec*, which maintains an array of *sExec*'s and executes them in sequence when the *exec()* method is called (see Figure 3). Other useful pre-made components include an arbitrary dimensional *cellPkgExec* for processing user-defined callbacks over a stencil of *Cells* and an equation solver (biCG with various preconditioners). Nested sequences of these *Exec*'s will be created during the framing phase of the application. During the framing phase, the user is expected to instantiate selected *sExec* components. These can be modified by replacing constituent objects derived from base classes used by the *sExec* or by inheriting directly off of a particular kind of *sExec*, or both. The execution phase begins by simply calling the top-level object's *exec* method: this object is usually a container of *sExec*'s such as *bulletinBoardExec*.

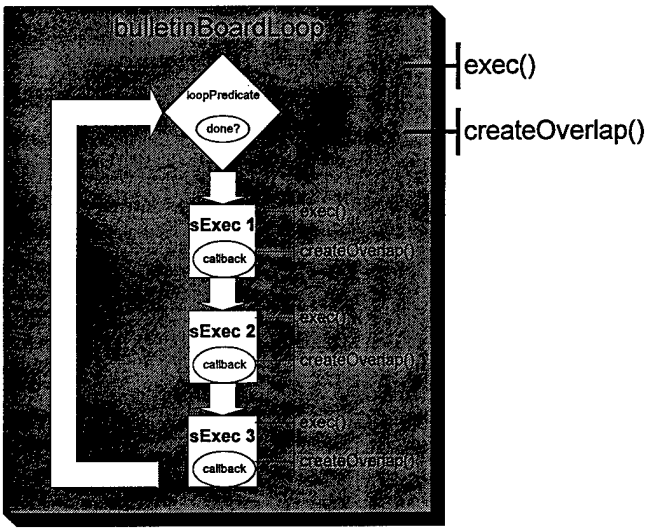

Figure 3

BulletinBoardExec's are a derived from *sExec* and serve as a container. Rectangles denote POET tool-kit code and ovals denote user code add-on. This particular bulletinBoard, *bulletinBoardLoop*, tests a user predicate before each iteration over it's constituents. The bulletinBoard also manages the data dependency of each *sExec* through the *createOverlap* method on its contained *sExec*'s. User-supplied callbacks (indicated by the ovals) particularize the *sExec*'s function to user's task. Most types of *sExec*'s have these callbacks and, in this case, so does the predicate object, *loopPredicate*. The block represented here is executed on all processors simultaneously in a SPMD fashion. No synchronizations are made by the framework, though it is likely that the constituent *sExec*'s will.

As the name implies, the *bulletinBoardExec* is a container one can tack *sExec*'s onto and, when the *bulletinBoardExec*'s *exec* method is called, will call its constituents' *exec()* methods. The *bulletinBoardExec*'s *createOverlap()* method also conveys dependency

information *via* the contained *sExec*'s *createOverlap()* method. A variety of bulletinBoardExec's exist as well: providing for iteration over its constituents until a predicate test returns false (*bulletinBoardLoop*). Another kind does a gather communication to an host processor, an *exec()* for each of the bulletinBoard's constituent components, and a scatter back to the worker processors (*standAloneBB*[6]). In general, an application implemented in the POET framework will look like nested bulletinBoardExec's within bulletinBoardExec's as in Figure 3.

Note that little in the way of a predefined model for parallel computing is required by the POET object model. No *a priori* constraint is placed on the parallel numerical method. The two methods from *sExec*: *exec* and *createOverlap()*, can be overridden to allow any numerical method to coexist in the POET framework.

3. Interfaces for Parallel Algorithms

Broadly, numerical algorithms are categorized as either implicit or explicit. Implicit methods, as the name implies, are treated as a black box: given the needed data, it computes independent of user intervention and returns the answer. Examples of implicit methods are an equation solver or matrix inverter. Explicit methods require the needed data supplied to a user-created computation updating the data with the result. Stencil problems are the most common explicit algorithms. Though the implicit methods are usually difficult to implement in their own right, especially on parallel-distributed systems, encapsulating and preserving implicit schemes as components is relatively straightforward.

For example, the POET sparse conjugate-gradient solver component allows the user to enter the coefficients row-wise *via* a method. It performs the solution and returns the solution vector, with the same partitioning as the original matrix. For the explicit *cellPkgExec* component, user-specified data has to be marshalled to a user-specified callback and images of the user-specified data existing on another processor (*i.e.* ghost data) must be maintained. This rich interplay between component code and user code is what makes the user interface to stencil methods difficult.

3.1 Components for Implicit Methods

Though implicit methods present a straightforward user interface, often performance, especially on parallel distributed systems, is lacking. The most significant feature of the componentized equation solver is that it must accept a variety of data/processor decompositions. Indeed all components in POET must be flexible to a wide range of decompositions. The numerical method component might have an optimum decomposition, and for an equation solver component on a distributed system, might well force the decomposition. Each component however, cannot dictate the decomposition. The interface for the POET bi-conjugate gradient (biCG) solver, *CGExec*, looks like:

```
class CGExec : public sExec {
private:
\\...
public:
// ...
CGExec(CGAnswerCallback *, proc *);
// ...
// execStatus is an enum type that provides success or failure information.
  virtual execStatus exec(void);
  virtual void createOverlap(partitionTable*);
};
```

[6] *standAloneBB*: *i.e.* allows a component to run by itself on a single processor.

```
// This interface is exported by CGExec, to sExec's upstream from it.
class CGInput {
private:
// ...
public:
// a array are the coefficients, size is the number of coefficients (always the
// same), b is the RHS column vector.
  void addRow(double *a, int size, double *b);
// ...
};

// This is used by CGExec to convey the answer to where it needs to go.
class CGAnswerCallback {
private:
// ...
public:
// Overridden by the user.
  virtual void useAnswer(float *) = 0;
};
```

Here *CGInput* is exported by *CGExec* and can be used by another component to load the matrix into *CGExec*. When *CGExec's exec()* method is invoked, the linear system of equations is solved. *CGExec* then uses *CGAnswerCallback* to convey the answer through the user overridden *useAnswer()* method.

Though many applications would benefit from equation solvers like *CGExec*, the high latency of TCP networking on networked distributed systems prevents the solvers from working well enough to be practical. Recent developments both in the networking[7] and algorithmic[8] sides promise to improve equation solvers on distributed systems in the future.

3.2 Components for Explicit Methods

The *cellPkgExec* class is derived from *sExec* and implements the notion of a generalized stencil computation. It is so-named because it is designed to call a package of cells to user code, dictated by the stencil. *cellPkgExec* manages and interprets a *stencil* class. The *stencil* implements the idea that, to advance the solution for a particular target cell, a certain number of other cells, or parts of cells, are needed. It is used to encapsulate data dependency information alone. POET regards a *stencil* to be an object that encapsulates the answer to the question: "what data do you need to accomplish your task?" The *stencil* class is given a target *procCellPair* from the current partitioning scheme (*partitionTable*) and is expected to produce a table of cells representing the region required for the computation of that target. Note that this is a much broader interpretation than the usual idea of a stencil as a template mask of cells moving over grid. Here each cell on the entire distributed machine can have a different region associated with its computation and there is no need for associated cells to be spatially "close" to the target cell. Indeed the POET framework does not have any concept of 1D, 2D or any dimensionality built-in, though many components are designed to suit a particular dimensional grid.

The *stencil* class is a pure virtual class that can be user inherited but is more likely to be used as a pre-made, inherited component that comes with POET. POET has many

---

[7] "*Myranet Fast Networking Hardware*", IEEE-Micro, pp. 29-36, **15**, (1995)

[8] *Optimistic Active Messages: A mechanism for scheduling communication with computation*, Deborah A. Wallach, Wilson C. Hsieh, Kirk Johnson, M. Frans Kaashoek, and William E. Weihl, 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95), Santa Barbara, California, July 17-22, 1995.

kinds of stencils: *stencil_1d*, for one-dimensional grids; *stencil_2d* for two-dimensional grids, *etc.*

```
class stencil {
private:
// ...
public:
// ...
partitionTable *createRegionAbout(partitionTable *t, procCellPair *target) = 0;
// ...
};
```

Recall the *partitionTable* container class from Figure 1 that holds the global cell/processor partitioning information for the entire distributed machine. It contains *procCellPair*'s that identify what processor a cell is on, whether that cell is a copy (ghost) and what part of the cell's contents that copy has. The *createRegionAbout()* method returns a new *partitionTable* containing just the region the computation for *target* requires. The *partitionTable* region output by the *stencil* can be used by *sExec* to identify what ghost cells need to be freshened from other processors after the user callback has been run. A sketch of a kind of *cellPkgExec:*

```
class cellPkgExec : public sExec {
private:
// ...
public:
// ...
cellPkgExec(sCallback *, stencil *, proc *);
// ...
// execStatus is an enum type that provides success or failure information.
virtual execStatus exec(void);
virtual void createOverlap(partitionTable*);
};

// *Not* derived from sExec.  Provides hook for user code.
class sCallback {
private:
public:
// ...
virtual execStatus exec(cellPackage *cP) = 0;
};
```

Here the methods *exec* and *createOverlap()* override the ones from *sExec*. The constructor for *cellPkgExec* requires the user to supply a callback in the form of a class derived from *sCallback*. The *cellPackage* container class will have the data cells identified by the *stencil*. The *proc* type in the third argument to the *cellPkgExec* constructor contains processor configuration information. The user overrides *exec(cellPackage *cP)* in *sCallback* class to do the work required by the explicit algorithm. It should be emphasized that *cellPkgExec* does not need to know anything about the dimensionality of the computation. All of that is encapsulated in the *stencil*.

The *stencil* class provides a way to associate a *target* cell with other cells so the framework can make provision for them in the decomposition of data cells across processors. *Stencils* also provide finer control by allowing scoping of only parts of cells to reduce communication further.

4. An Example: Chemically Reacting Flow

Many applications have been implemented in POET: Molecular Dynamics, Quantum Monte Carlo, Seismic Ray Tracing, *etc.* Target users for POET can be described as computer literate scientists that are expert in disciplines other than Computational Science. These users are typically familiar with the concept of parallel computing and

already have an idea of how their problems can be mapped to a parallel environment. They are unwilling however, to hand code the communication patterns and algorithms necessary to implement their ideas. The focus here will be the mechanics of creating an application in POET and as an example a Chemically Reacting Flow (CRF) application will be used.

This CRF application models a turbulent fuel jet into stagnant air. Because realistic chemistry is computationally demanding, implementations of this and similar models on serial architectures require a drastically simplified and reduced chemistry. Usually one to three chemical species is the practical limit for these models. Generally, serial implementations for chemically reacting flow predict only gross and cumulative properties accurately, such as temperature and air/fuel mixture fraction. With increasing interest in pollutant formation in combustion systems, a need has developed for detailed chemical calculations involving 50-100 species. Because pollutants, such as Oxides of Nitrogen, are present in only trace quantities in combustion effluent, even minor species must be tracked in order to produce a credible prediction of pollutant formation. Predicting pollutants created by flames necessitates a parallel approach to the problem.

We use a Probability Distribution Function (PDF) approach[9] that models species in terms of an ensemble of statistical instances. Since our intention is to use a workstation cluster where equation solvers function poorly, we chose to compute the explicit chemistry and transport in parallel only, leaving the implicit fluid mechanics to a single node. Because the fluid mechanics deals only with averaged quantities (density, and velocity) the extra communication caused by this host/worker arrangement is kept to a minimum. These presuppositions are born out in the resulting model: for a typical run on 10 SGI Indigo[10] workstations, 3.5 days are needed for the chemistry and during this time only 15 minutes of CPU time are required for the fluid mechanics running on a single processor. At present there exists both steady and unsteady versions of this model. In what follows, the framing operations necessary to create the CRF application in POET will be sketched.

4.1 Framing the Chemically Reacting Flow Problem

The task is to reduce what needs to be done into components. Components must be linked together by means provided by the framework and by the components themselves. During this framing phase no calculation is done, or rather no *exec* method is called. Here only the "script" for what will be done at execution time is laid out.

First, the data for the grid must be initialized with the starting values of chemical species, velocities, *etc.*

```
// PVM identifies the message-passing library
// PVM requires the name of the executable
  char *name = "crf";
  proc *p = new proc( numProcs, PVM, name);
  IBP_Callback* cb0 = new initCallback;
// numXCells is the number of cells in the x-direction
// numYCells is the number of cells in the y direction
  partitioner* part = new Partitioner_2d(p, numXCells, numYCells);
  sExec *iBP = new initByPieces(p, part, cb1);
```

The above code fragment is how the CRF application is framed for initialization and is concerned with instantiating an *initByPieces* object. The *initByPieces* class initializes

---

[9] "An Improved Turbulent Mixing Model". S.B.Pope, Comb. Sci. & Technol., **28**, 131-135, (1982)

[10] Silicon Graphics Indigo 1's with R4000 processor, 64Mb memory.

the two-dimensional grid cells in small pieces to reduce memory usage on a user-designated host machine and sends the cells out to worker machines. This code fragment is included for completeness and is not important to a discussion of the POET object model. Briefly, the *proc* class, instantiated first, holds the information defining the distributed machine resources and is needed by the various components (see Figure 1). Here *initCallback* is derived from a pure virtual class *IBP_Callback* that is needed by the initialization *sExec initByPieces,* similar to the case for *sCallback* above. Created by the user, *initCallback*[11] inserts initial velocities, densities, concentrations, *etc.* into the cells that form the grid. This also introduces a new (pure virtual) class called *partitioner.* The derived *Partitioner_2d* class understands that the problem is 2D and will provide the initial partitioning scheme for the problem. Note that the *InitByPieces sExec* is aloof from the dimensionality of the problem: only the *initCallback* needs to be coordinated with the *Partitioner_2D.*

Now that things are initialized, the stencil operations must be framed into components:

```
Stencil_t is[9] =   { Pt_St_NotNeeded, Pt_St_Needed,   Pt_St_NotNeeded,
                      Pt_St_Needed,    Pt_St_Target,   Pt_St_Needed,
                      Pt_St_NotNeeded, Pt_St_Needed,   Pt_St_NotNeeded };

  sCallback *cb1 = new diffCB;
// stencil2d requires a vector of Needed/NotNeeded's and the x and y dimension
// of that vector
  stencil *theStencil = new stencil2d(is, 3, 3, numXCells, numYCells);
  sExec *cPX1 = new cellPkgExec(cb1, p, theStencil);
```

Here *diffCB* is derived from *sCallback* and does a diffusion step. The *stencil2d* defines a 5 point star (using the *Needed*'s and *Not_Needed*'s). Both of these are used by *cellPkgExec* and are given to its constructor. Similar to the above a chemistry/post-processing callback, *cb2* is created by the user (not shown) and a second *cellPkgExec* instantiated with the same stencil:

```
  sExec *cPX2 = new cellPkgExec(cb2, p, theStencil);
```

Now that the chemistry and transport operations are done, the fluid mechanics calculation has to be framed for a single host processor. Recall that this requires data from the grid, residing on the workers, to compute and is to be done on a single processor.

```
  standAloneBB *sABB0 = new standAloneBB(p); // Special gather/scatter BB.
  sExec *dCFD = new doCFD;  // A derived sExec does CFD over the entire grid.
  sABB0->addExec(dCFD);  //Add the CFD sExec to the bulletinBoard.
```

The *standAloneBB* is a *bulletinBoardExec* that has a user-designated host processor that: (1) has a built-in *sExec* component that does a gather operation to the host; (2) runs *exec()* it's constituent *sExec* components on the host alone; (3) has another built-in *sExec* that does a scatter operation back to the workers. New *sExec*'s are added to any bulletinBoard by the *addExec()* method. Exec *doCFD* is derived directly off of *sExec* and operates on the entire grid, prepared for it by *standAloneBB.*

The diffusion, chemistry and CFD constitute the main iterate for advancing the solution in unsteady or steady modes. These will be contained in their own *bulletinBoardLoop* (see Figure 3) with a predicate that decides step sizes and returns false when the calculation is over:

---

11 All of the user-derived class are written by combustion scientists noted in the acknowledgments and are actually encapsulated FORTRAN subroutines and functions. The number of lines of FORTRAN in the application actually exceeds that of POET itself.

```
loopPredicate *lP = new stopOrGo; // user-derived predicate class.
bulletinBoardLoop *loopBoard = new bulletinBoardLoop(lP);
loopBoard->addExec(cPX1);
loopBoard->addExec(cPX2);
loopBoard->addExec(sABB0);
```

Here *stopOrGo* is a predicate class user-derived off of the pure virtual *loopPredicate*. *bulletinBoardLoop* uses *loopPredicate* to tell it when to stop iterating.

Finally, something that outputs the computed results must be framed:

```
sExec *dIO = new doIO; // Derived directly from sExec to write results on host.
standAloneBB *sABB1 = new standAloneBB(p) // Gather data host for output.
sABB1->addExec((sExec*)dIO);
```

The *sABB1* component makes sure that the results are written on a known machine in a known place.

At this point we have three main components: for initialization, *iBP*; computing the chemically reacting flow, *loopBoard*; and post-processing and filing the results, *sABB1*. All of these are a polymorphized *sExec*'s. They can be tied together to make the entire application by putting them into a standard *bulletinBoardExec*.

```
bulletinBoardExec *mainBoard = new bulletinBoardExec; // Runs constituent
                                                      // sExec's once.
mainBoard->addExec((sExec*)iBP);
mainBoard->addExec((sExec*)loopBoard);
mainBoard->addExec((sExec*)sABB1);
```

A plain *bulletinBoardExec* does just one iteration of its constituents and returns. *mainBoard* is the entire application and encapsulates everything the CRF code will do. It is important to remember that at this point no calculation has been done. In this framing phase, components have been instantiated, particularized to the CRF application, and linked together, but nothing has actually been computed. The execution phase is begun by calling the top-level *exec()* method:

```
mainBoard.exec(); // End the framing phase, begin the execution phase.
```

Here all that is to be done by the application is accomplished.

Note that the form of the POET framework is largely dictated by the limitations of C++. We might prefer that the final call to *exec()* be automatic. We might prefer, for example, a Java-like language where mainBoard is implicit and created by the framework—where the user "extends" a system-provided, top-level component. C++ has be chosen mainly for portability reasons. POET applications must run on the latest parallel hardware, such as the 9000 processor Intel TFLOPS™ machine, where interpreters or compilers for less-used languages may be absent. Because POET scrupulously avoids using C++ built-in classes, all that is required is an ANSI C compiler and a C++ interpreter available somewhere in close proximity.

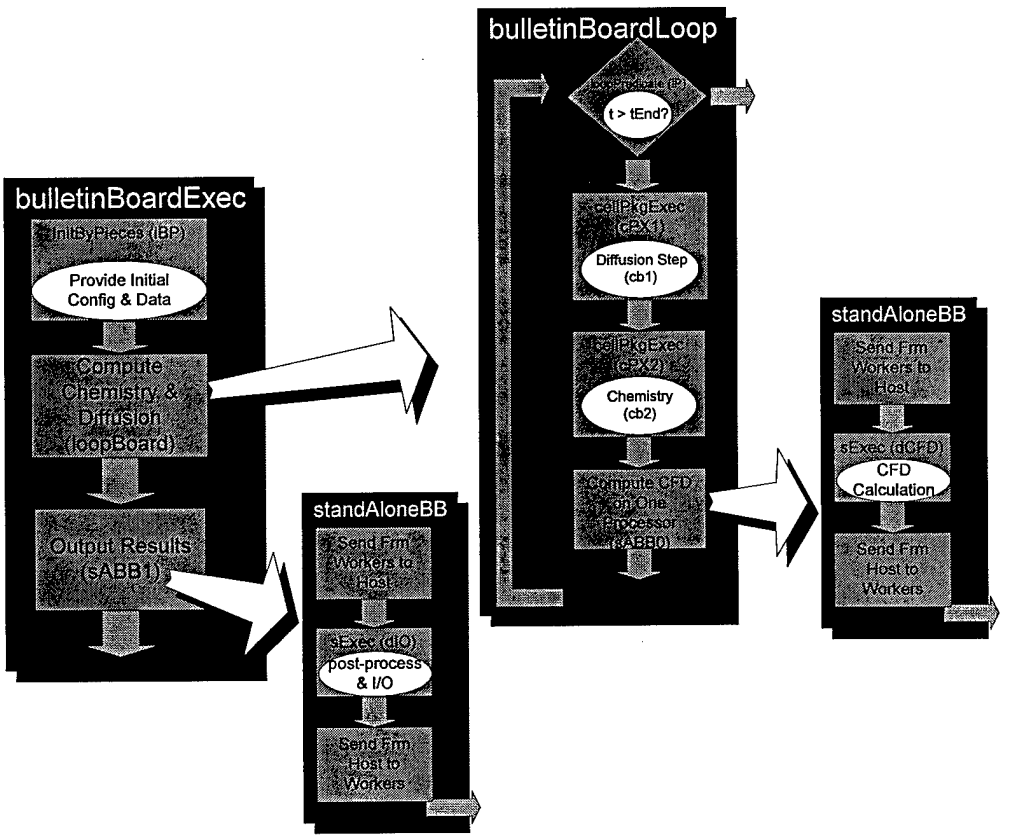

Figure 4

A block diagram of the framing process sketched in Section 4.1. Rectangular elements denote POET system components and ovals denote user-created classes. To orient this diagram to the code fragments given in Section 4.1, names of classes appear as titles for the components and class instance names appear in parenthesis.

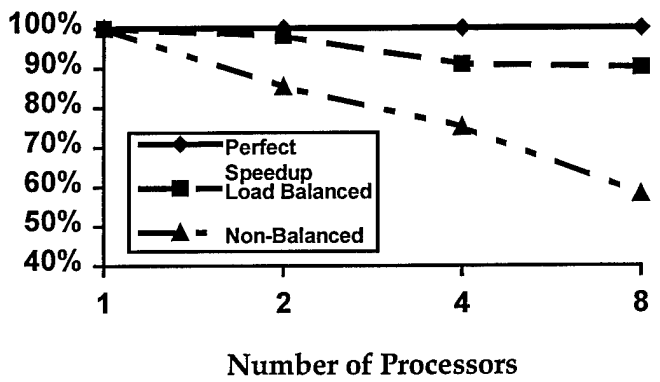

Figure 5

Performance results for the CRF model on a cluster of 8 workstations[12]. Combustion calculations often require load-balancing because CPU work is usually localized to the flame position in motion on the grid. Here a load-balancer component is added to the framing operation of Section 4.1 and comparison is made between load-balanced and non-load-balanced case of Figure 4. Though the load-balancer component is specific to the dimensionality of the problem, it has been reused for an unrelated molecular dynamics application.

The previous is meant to be just a sketch of a real application that produces usable results in the combustion field of research. As time progresses more components are added and existing ones tuned. Improvements can be made in the bare-bones application of Figure 4 by introducing a load-balancing component. The load-balancer is added to *loopBoard* and extracts timings from the *cellPkgExec*'s already in *loopBoard.* When the load-balancer is *exec()*'ed it rearranges the cell/processor decomposition to optimize performance (see Figure 5).

5. Conclusions

The purpose of a parallel framework is ease-of-use and code reuse. Although many applications implemented in POET share the same components, code reuse can best be seen within the CRF example (Figure 4). In the same application components are reused to perform different parallel tasks. The ease-of-use question is better addressed by a previous paper[13] that goes through an actual user experience. Here we are concerned with the mechanics of what components attributes must be in order to cooperate with other components for parallel computing. No single component can dictate the processor/data decomposition. The framework is the arbitrator of the decomposition, must take the needs of all the components into account. The parallel framework must come to a decision that attempts to optimize performance.

The advantage that frameworks have is two-fold:

1. Reusable code. One can afford to invest time on components that you wouldn't normally do for one application alone.

---

[12] DEC ALPHA 3000/600 workstations running DEC OSF/1 V3.2 kernel.

[13] R.C. Armstrong, *Frameworks for Parallel Computing*, Proceedings Of Parallel Object Oriented Methods and Applications (POOMA), December 5-7, 1994, Santa Fe, NM.

2. Control and monitoring the entire calculation. Can monitor progress of individual components in relation to the entire application and make changes on the fly, similar to the load-balancer component previously.

Frameworks permit parallel code reuse by arbitrating the processor/data decomposition and providing uniform services to components, allowing communication and cooperation. Without inter-component cooperation, parallel algorithms cannot be modular, and without modularity there can be no code reuse. Without this framework environment, persisting into the application's execution phase, cooperation would not be possible. The user need only be aware of the interface to a component not the details of the parallel algorithm it represents, nor the data decomposition it requires. This property is shared by all of the frameworks mentioned in Section 1.2. Note that the use of frameworks is a fundamentally different approach than using a compiler; parallelizing or otherwise. The framework participates *both* in the creative, *framing* phase as well as the *execution* phase. Compilers are not involved in what happens during the execution of the programs they compile.

The POET framework introduces an *sExec* class (see Figure 2) that is the embodiment of a component implementing a parallel algorithm. The *sExec* has two methods:

1. *exec()* which says go do what your component is designed to do—presumably a parallel algorithm. By itself *exec()* admits only the imperative style of programming that frameworks seek to supplant.

2. *createOverlap()* which conveys data dependency information needed by the *exec()* to accomplish its function. This allows *sExec* components to cooperate over the same data/processor decomposition.

Ghost overlap data communications, for example, can be piggybacked with among components are unaware of the each other's existence. All of the POET framework is built on this single idea. This one extra concept however, allows for a rich expression of most commonly used implicit and explicit numerical algorithms.

This extra bit of functionality though, is about the minimum that could be done and still have a functioning framework for parallel numerics. The current POET framework can know little or nothing about what communication pattern is produced by the *sExec* component. For load balancing and scheduling purposes, it is necessary for the framework to query, not only data dependencies, but the entire communication and computation graph for the component. Container *sExec*'s, like *bulletinBoard* would then be able to sew together constituent *sExec* graphs into its own graph, and so on recursively. This should provide enough information to create much better load balancing components than that of Figure 5. A design that will accomplish this for the POET tool-kit is currently underway.

Frameworks for parallel computing are a promising avenue to reusable parallel code. This is ultimately because the framework is "aware[14]" of what it has been framed to do and can query components as to what their individual requirements are, making provision for them in the most efficient manner possible.

---

[14] The authors are mindful that computer "awareness" is a topic in the popular media and they use the term in only its mildest sense.

M97051376

Report Number (14) SAND--96-8610C
CONF-970112--

Publ. Date (11) 199701

Sponsor Code (18) DOE/ER, XF

UC Category (19) UC-400, DOE/ER

DOE