RC24811 (W0906-078) June 18, 2009
Computer Science

IBM Research Report

Interoperable Model Graph Simulator for High-Performance Computing

James Kozloski, Maria Eleftheriou, Blake Fitch, Charles Peck
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Interoperable Model Graph Simulator for High-Performance Computing

James Kozloski, Maria Eleftheriou, Blake Fitch, Charles Peck, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

Abstract.

We designed a system for rapidly composing simulations of networks characterized by extreme heterogeneity and scale. Our target domain, computational neuroscience, requires flexibility for rapid and iterative extension and revision of each modeled system. Here we present a graph simulator, which employs one language for defining interoperable models and one for declaring graphs whose vertices are instances of these models. Together these constitute a standard programming model for specifying simulations of complex networks such as those found in neural tissue. Our graphs comprise heterogeneous collections of models whose connections are initialized at runtime for communicating specific model state at specific phases of model execution using efficient collective communication. We demonstrate the applicability of our graph declaration languages for parallel programming, with examples of the 3-dimensional Fourier transform. Finally we show preliminary scaling and performance on the Blue Gene/L supercomputer comparable to that of a stand-alone, optimized implementation of the same algorithm.

1. Introduction

The field of computational neuroscience models neural tissue with simulations of large networks of communicating elements. Computational neuroscience presents challenging requirements for any simulation system that aims for applicability to the broadest range of models. We designed an interoperable model graph simulator (“the Simulator”) for executing large-scale network simulations [1]. Here we extend the system to support executing network simulations in parallel on a variety of massively parallel supercomputers, such as the Blue Gene family. Our design points balance a flexible and extensible environment for specifying arbitrary networks with run time efficiency. Flexible domain modeling requires a generic architecture, while run time efficiency requires a specific mapping onto high-performance computing architectures for maximal exploitation of these resources. In addition, our design satisfied certain key requirements we identified as specific to this field and central to meeting the goal of broad applicability. In part because of these demanding requirements, we are can now apply the tool to other problems in parallel programming.

Heterogeneity is a central property of neural systems and therefore a key requirement for neural system modeling. Heterogeneity refers to the diversity of neural elements found in the brain, both in terms of different neuron types, as well as different processes and components that comprise individual neurons and their connections (synapses). Computational neuroscience also models neural systems at levels of abstraction above that of the single neuron (for example, models of neural microcircuits, columns, brain areas, tissue volumes, and brain subsystems). The diversity of elements found in the brain and the diversity of abstractions used to model them create a need for simulation systems that can support arbitrary and heterogeneous network elements. To meet this requirement, the Simulator can now specify and implement any component of a computational algorithm representable as a communicating network element.

Brain networks have massive scale. The human neocortex, for example, comprises about 28×10^9 neurons connected by about 10^{12}-10^{13} synapses [2]. The largest scale simulations to date are orders of magnitude smaller [3]. Some argue that the need for still larger scale neural simulations may provide better understanding of how real networks function and give rise to biological capabilities [4]. Many requirements for our simulation system derive from these large scales: efficient initialization of large numbers of computational elements, their connections, and distribution across high performance computing platforms; balancing computation and communication across distributed computing networks; and communicating state efficiently in order to minimize run time are just a few. Our system provides multiple user-specified and automatically generated components for addressing and manipulating large numbers of network elements, their computation, and communication as single units. These aggregations support efficient initialization, distribution, and runtime performance.

Many efforts in computational neuroscience currently aim to integrate models from different levels of abstraction into “consistent” simulations [5]. Here, consistent refers to the property of a simulation that allows its elements to perform the same computations regardless of configuration. We designed the Simulator to encapsulate network elements and to provide generic, user defined interfaces for integrating new instances of existing or newly defined network elements into existing simulations. The Simulator thus supports arbitrary configurations with correct runtime communication through these interfaces.

The Simulator addresses the requirement of supporting networks that are extensible both in scale and in heterogeneity by supporting interoperability between existing network elements and newly defined elements. In meeting this requirement, the Simulator can be used for developing, maintaining, and extending any parallel

program that can be represented as a complex network (i.e., a graph) of communicating elements. Graph here refers to a set of vertices joined by edges (Fig. 1). Unlike the simple concept of network, comprising nodes joined by precisely zero or one connection(s) per node pair, a graph comprises vertices joined by any integer number of edges per vertex pair greater than or equal to zero.

Figure 1: A simple network is defined by nodes and connections (one or zero per node pair), whereas the graphs simulated by the Simulator consist of vertices joined by any number of directed edges.

The paper is organized as follows: Section 2 describes design concepts for the Simulator and discusses its target compute platforms. Section 3 describes an approach for specifying parallel programs as graph declarations and gives an overview of the Simulator system of graph declaration: graph specification, model definition, model connection, graph partitioning, and simulation phase sequencing. Section 4 presents two case studies of graph declarations; each graph computes the 3-dimensional Fourier transform, the first by implementing the 3-dimensional direct Fourier transform (3D DFT) algorithm, and the second by implementing the 3-dimensional fast Fourier transform (3D FFT) algorithm. Section 5 presents a more complex graph declaration implementing a previously published model of neocortical self-organization using the Simulator. Section 6 describes the Simulator execution architecture, detailing how a graph is partitioned, its elements executed, and its state communicated on phase boundaries. Section 7 reports the performance run times for our various graph declarations using the Simulator on the Blue Gene/L supercomputer.

2. System Design Concepts and Applicability

Interoperable Models for Composable Graphs

Object oriented programming languages such as C++ support the class construct in order to encapsulate data of specific type, declare methods for accessing and manipulating these data, and specify what access other users of the class have to data and methods [6]. Classes in these languages do not provide constraints necessary for their

immediate use in parallel programs as definitions of communicating sequential processes (CSPs; as defined by Hoare [7]). In Hoare's simple scheme, assignment to one sequential process from another is equivalent to communication in a parallel program; in practice however, the task is more complicated. Parallel programmers often leverage object oriented design concepts and the class construct to organize communication between sequential processes [8]. Classes are also used to encapsulate sequential processes, while providing access to data through the class application program interfaces (APIs), whether the access is through local memory or a communications subsystem such as the Message Passing Interface (MPI). These approaches, while convenient, nevertheless require that a programmer design class syntax to support the instantiation and coordination of CSPs within the parallel program. Hoare's vision of simple primitives for composing and coordinating CSPs has yet to be realized in a widely-accepted programming model, though it has influenced the design of many special purpose tools, communication interfaces, and parallel programming languages (for example, see [9]).

The Simulator employs a more constrained concept than class for encapsulating network elements during graph declaration. All network elements in a simulation are "models." A model includes local data, communication interfaces, and a set of sequential processes captured as execution phases. The conjunction of a model and its phase is then equivalent to Hoare's concept of a CSP. Models also implement:

1) How a model receives and parameterizes connections from other models, and what interfaces are exercised for each connection type
2) How references to data presented by these connections are organized and stored
3) The number and name of distinct phases a model requires to perform its computation
4) The communication of phase boundary-specific model data
5) A scalar estimate of the computational and communication costs associated with each phase
6) Any additional user-defined methods required for model initialization, connection, or execution

In this way, model definition provides a programming construct sufficient in its constraints to define a CSP, and model graph declaration provides a construct sufficient to integrate models into a network of CSPs.

Parallel Algorithms as Model Graph Simulations

To demonstrate how the Simulator and its system of graph declaration may be used as a programming model for high performance parallel computing, we will present two examples of parallel algorithms (computing the 3-dimensional Fourier transform), each specified using the Simulator's graph declaration languages. In the Simulator

system, all graph vertices are model instances, all edges are connections between model instances (declared individually or in aggregate), and all graphs are directed graphs. When a vertex pair is joined by multiple edges, each connection represents a different data relationship or connection for communication between the model pair. A declared graph is then executed in parallel according to rules of an implicit runtime system. These rules make the execution of the simulation consistent, and allow a graph simulation to implement a range of parallel programs:

1) A phase of computation defined for a set of models and declared for a graph is executed in parallel across multiple threads referencing shared memory, multiple processes referencing distributed memory, or both. Models that share a phase have no data dependencies. We refer to this rule as the "Acyclic Rule".
2) Within a phase, models reference the data of other models directly through the models' APIs, applied to either the model or a proxy, such that access is always to locally stored data. We refer to this rule as the "Data Locality Rule".
3) Communication occurs on phase boundaries and only between models and model proxies; output model data that changes within a phase is marshalled and communicated to all processes instantiating proxies of model; data is subsequently demarshalled into these proxies. We refer to this rule as the "Proxy Communication Rule".

Our graphs therefore comprise heterogeneous collections of models, joined together by connections and initialized at runtime for communicating specific state at specific phases of model execution. Model and graph declarations are sufficient then to specify complex network simulations of structures such as neural tissue, as well as to specify many parallel algorithms or programs.

Target Platforms

Graph declaration in the Simulator is platform independent. Therefore, the system may be used to declare graphs once for execution on either standard serial architectures or high performance parallel architectures. The Simulator execution engine is designed to exploit a variety of high performance platforms. Parallel execution of the graph can occur in a multi-threaded, shared-memory architecture such as a symmetric multiprocessor (SMP) or multi-core processor system. Parallel execution of the graph can also occur in distributed memory architectures such as the massively parallel processor system Blue Gene/L. For the latter case, the graph is partitioned, and each sub-graph is allocated to a processor in a manner that exploits Blue Gene/L's 3-dimensional torus [10]. A combination of these two execution strategies will allow the Simulator to exploit massively parallel systems that support multiple parallel threads on each system node (such as Blue Gene/P [11]).

3. Overview of Graph Declaration

Graph Design

The Simulator provides two declarative languages (Model Definition Language or "MDL", and Graph Specification Language or "GSL"), which together create a standard syntax for graph declaration. These languages together introduce the programming construct of a "model," which constrains the class concept and supports rapid composition of network simulations as declared graphs. We have previously used the Simulator to specify large-scale simulations of self organization in the primate visual cortex [12]. Here we show that these languages are also sufficient for specifying parallel algorithms and their constituent sequential processes that can be represented as graphs [13].

The most important consideration in graph design is the Acyclic Rule. This rule states that no two models in a simulation can share both a simulation phase and data dependencies. A data dependency occurs when one model writes to its data and another reads that data in the same phase. For each simulation phase, a specific phase view of the overall simulated graph may be considered, which captures these data dependencies (Fig. 2). Analyzing the topology of each phase view of a simulation graph and ensuring no loops exist is sufficient to ensure that the Acyclic Rule is satisfied.

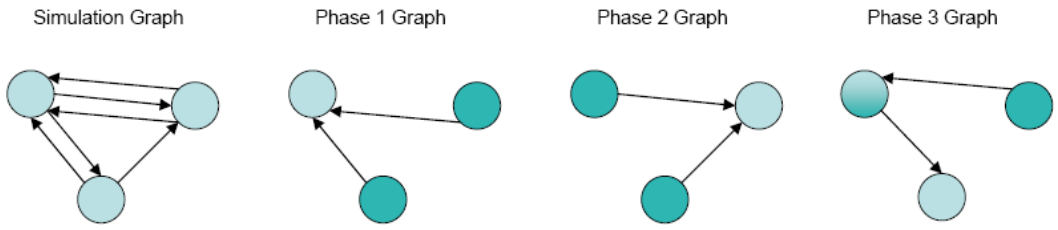

Figure 2: A single simulation graph (left) can be decomposed into phase views. For each phase the data that a model executed in the phase (light blue) is represented by an edge. Models that provide this data (dark blue) must not modify the data in the same phase (Acyclic Rule).

When decomposing a parallel algorithm or network simulation into a graph, it is necessary to identify which sequential portions of the algorithm can be executed independently, then to map these portions to phases encapsulated within the model definitions. Different models implementing an algorithm may be executed in the same simulation phase only if they do not have data dependencies. Other considerations in graph design include the

amount of state communicated between models on phase boundaries. As we will explore below, often the same algorithm may be decomposed into more than one graph representation. The considerations for choosing between different decompositions then include not only which graph best distributes the work of computation to the most model instances (and thus achieves the greatest degree of parallelism), but also which minimizes the amount of data communicated between model instances on phase boundaries.

Model Definition

Model definition with MDL results in C++ code generation. A model therefore comprises multiple C++ classes automatically generated during MDL parsing. Writing MDL may include first defining derived data types, which any model may then reference during compilation of MDL generated C++ code. Primitive data types supported by MDL include: `bool`, `char`, `int`, `unsigned int`, `long int`, `float`, and `double`. Derived data types are defined using the `Struct` keyword and curly brackets `{}`, within which primitive data type or other Struct instances may be declared. Both primitive data types and Struct data types may be followed by square brackets `[]` and a declared instance name to indicate that the data type of the instance is an array of the declared type (for example `foo [] f;`). Array lengths are not defined within MDL and are determined dynamically during graph initialization based on how the array is used. The Simulator provides a thread-safe Array container for this purpose. Square brackets in Array declarations are therefore always empty. In addition, any of these methods of data type declaration can include the standard symbol (*), indicating that the declared type is a pointer to the data type; in the case of pointers to Arrays, a data type is followed by square brackets, then the pointer symbol, then the declared instance name (for example `foo []* f;`).

Prior to model definition, MDL may also specify Interfaces that any model may reference during compilation of MDL generated C++ code. Interfaces are declared similarly to Structs, substituting the `Interface` keyword for `Struct`. The designation of a data structure as an Interface indicates that models can either provide or accept these data during graph connection initialization. A Model definition includes the keyword `Model` followed by the Model type name. To denote that a model will provide data of a specific Interface, the Model type name is followed by the keyword `Implements`, followed by the name of the Interface type it provides. MDL may also define

a Model as implementing more than one Interface by providing a list of comma delineated Interface types after the `Implements` keyword.

Interfaces are used to define how a Model provides access to its data and how it uses data from other models. In either case, the data or state referenced by Interfaces must be defined within the main Model definition body, which is delineated by curly brackets. State that is included with each instance of a model is called model instance state. Model instance state declaration in the Model definition body consists of a semicolon delineated list of data types and declared instance names. In addition, state that is shared by all Models of a specific type is declared using the `Shared` keyword followed by additional set of curly brackets. Shared data is referenced using the `Shared` keyword followed by a period. Interface implementations and examinations use these Model state names in the remaining Model definition.

An Interface implementation declares which model state corresponds to an Interface attribute. It is defined using the name of the Interface, followed by a period, a declared instance name found in the Interface definition, the double left shift operator <<, and the name of the Model instance state that is provided by the interface. For example `FooProducer.foo << myFoo` represents an implementation of the `FooProducer` Interface, which one can infer includes an attribute `foo` provided by the Model instance state of the same type, `myFoo`. Often an Interface's declared data type is a pointer to the corresponding Model state data type. In this case, the Model state instance following the double left shift operator is preceded by the ampersand (&) symbol, indicating that the address of the Model state instance is to be provided by the Interface's pointer type instance. For example, `FooProducer.foo << &myFoo` maps the address of `myFoo` to the `FooProducer` Interface instance's member `foo`. Data type checking during MDL parsing ensures that `foo` is of the type pointer to `myFoo`'s data type.

During initialization, a Model will establish connections to access another Model's state (through its interfaces) if the other model satisfies a predicate. Simple predicates can be specified in MDL. More complex predicates can be implemented by user implemented functions declared in MDL using the keyword `PredicateFunction`. Such a declaration results in C++ code generation of a function stub of the same name. An MDL user completes these functions using the C++ programming language prior to Model compilation. Model and connection parameters specified in graph declaration are also available for implementing this test function. The definition of model Interface examination then involves the declaration of a Connection, using the keyword

Connection, followed by parentheses () containing a simple test (for example Pset.foo==0.5) or a Predicate Function name, indicating this Predicate Function is called at connection initialization time to test if Interfaces from the connecting model should be examined. In addition, Boolean expressions that include multiple equality tests and Predicate Function names may be used. The keyword Expects followed by a comma delineated list of Interface names then indicates which Interfaces are expected from the connecting model. If any of these are not supported, it is rejected. For example, Connection (testInput()) Expects FooProducer, indicates that the defined model will execute the Predicate Function testInput for each connection received. If the return value of testInput is true, the connecting model Interfaces are then examined for FooProducer. If this Interface exists the Connection is processed, otherwise, the Connection from this particular model is rejected.

The processing of Connections from models is defined within curly brackets following the Connection declaration. Here, the double right shift operator >> is used to define Interface assignments, similarly to the above use of the left shift operator to define mappings from Model state onto Interface data. The Interface assignments describe how all Interface reference data will be stored in the defined model state at connection time. In the example above, curly brackets following the Connection declaration may include the following: FooProducer.foo >> fooInput; , indicating that the discovered FooProducer interface's data member foo should be copied into the Model's state data fooInput. Consistent with the example above, fooInput should be of type pointer to myFoo's data type. Finally, arbitrary functions may be declared outside the Connection definition using the keyword UserFunction, and listed inside the Connection definition before or after these Interface assignment statements, indicating that user defined C++ code should be executed before or after the defined Connection Interface assignments. Parameters passed in during graph declaration are available for implementing these functions. User Functions are useful, for example, if manipulation of Interface data is required prior to receiving a subsequent connection.

Finally, Model definition requires declaration of Model phases. Phases can be declared using the keyword InitPhase, RuntimePhase, or FinalPhase, to indicate at which stage of the simulation they are to be executed. Init and Final Phases are executed once at the beginning and end of a simulation, respectively, while Runtime Phases are executed iteratively until a stop condition is satisfied. A Phase declaration then includes a name for the internal Model Phase, followed by parentheses. Within these parentheses, model state is provided in a comma delineated list

indicated which state is modified during this phase. Only state that is part of an Interface implementation is listed within the Phase declaration. Thus any state that must be communicated on the phase boundary must be identified in the Phase declaration. For example, `RuntimePhase generateFoo(foo);` indicates that the Runtime Phase `generateFoo` will complete with the communication of all `foo` state, according to the Proxy Communication Rule. MDL provides a warning if state that is part of an Interface implementation is not listed for communication in the definition of any Model Phase. Similar to Predicate Functions and User Functions, Phase declarations result in MDL generation of a C++ function stub that an MDL user subsequently completes prior to Model compilation. This C++ code defines the Model's behavior in the overall graph simulation during this Phase, and represents a runtime sequential process.

Connection Functors

After defining Models using MDL, a user may declare a graph that incorporates any model type using the graph declaration language GSL. As described, MDL defines Interfaces and how a Model gains or provides data access during graph initialization. GSL then specifies which connections are attempted. Central to this specification is the Simulator programming construct known as a Connection Functor. Connection Functors are defined in MDL and declared in GSL; they express how a set of models may connect to another set of models. In addition, Connection Fuctors specify how to parameterize Connections. First, GSL declares an n-dimensional Model structural component known as a Grid. A Grid allows Models to be assigned to locations and connections to be specified according to Grid location. Within a Grid, Models of a given type are allocated within Layers. Model Sets are then declared in GSL, constraining Grid Layers to smaller sets of Models to be connected, such as ranges of Grid locations. These Model Sets are then passed as arguments into Connection Scripts. These employ Connection Functors, which test the predicates and execute the routines necessary to provide access to the appropriate data. Connection Functors are stateful and are implemented in C++ with a reference to a Connection Context. With this state and contextual reference they can support elaborate Grid location sampling methods that depend on statistics and patterns of previously established Connections. Connection Functors are also composable, such that one declared Connection Functor may be passed as an argument to another, allowing for even more elaborate sampling methods. In this way, very compact descriptions of massive patterns of connectivity between Models can be

expressed in GSL, allowing one Connection Script to easily scale out to larger Grids and networks through at most a minor adjustment to Connection Functor parameters.

Granules

Once a model graph is specified, it must be partitioned across the processors that will collectively execute the simulation or parallel program. Because of the large number of Models and Connections the Simulator supports for any graph, graph partitioning may be computationally intractable. To reduce the computational complexity, the Simulator introduces another grouping construct, the Granule. A Granule is an arbitrary grouping of Model instances that, unlike a Grid or Model Set, is not declared directly in GSL. Instead, Granules are populated with Model instances by Granule Mapping Elements, which themselves are declared in GSL. Each simulation Grid Layer is evaluated by an associated Granule Mapping Element, and each Granule is assigned to a particular processor within a distributed architecture. Models are assigned to Granules by the Granule Mapping Element upon instantiation. Mapping proceeds according to a procedure implemented by the Granule Mapping Element, which can include an implicit or parameter-based mapping from Grid coordinates to the distributed system's processor mesh coordinate system. Like Connection Functors, Granule Mapping Elements are stateful and implemented in C++ classes prior to graph declaration. Granules provide a means to represent the graph more coarsely for subsequent graph partitioning. In support of Granule graph partitioning, Granules are also responsible for aggregating statistics on the computational costs and communication costs associated with Granules and the declared Connections between Models within any given Granule pair. Within a memory space, a Model proxy is created whenever a Connection is made to a Model assigned to another memory space.

Phase Sequencing

To facilitate interoperability between Models specified with different Phases and Phase names, GSL provides the means to map arbitrary internal Model Phases onto graph specific simulation Phases. This is accomplished by GSL syntax for declaring and ordering simulation Phases, and for associating each Model Phase with a simulation Phase upon declaration of the Model type in GSL. For example, a GSL specification may begin with the following simulation Phase declarations and ordering: `RuntimePhases = { run1, run2 };`. This syntax names two

simulation Phases, and orders them for sequential execution during runtime. Now, when three different Model types are subsequently declared, their internal Model Phases are mapped to the Simulation Phases as follows:

```
ModelType X { computeX->run1 };
ModelType Y { computeY->run2 };
ModelType Z { computeZ->run2 };.
```

Clearly, in order to satisfy the Acyclic Rule, Model Types `Y` and `Z` must not share data dependencies between their internal Phases `computeY` and `computeZ`, since both internal Phases are mapped to the same simulation Phase. Using Phase sequencing, Model definition can proceed with no explicit knowledge of another Model's internal Phases (for example, if the other Model has not been conceived or defined yet), yet at a subsequent time, both Model types may be integrated consistently into the same graph declaration, provided their data dependencies are understood, and the proper ordering of their internal Phases is determinable.

4. Parallel Algorithm Case Studies: 3D DFT and 3D FFT

To illustrate graph declaration using the Simulator, we provide examples of two implementations of the 3-dimensional Fourier transform in MDL and GSL. The 3D-DFT includes a Simulator Model type that computes a single wave number of a direct Fourier transform (Fig. 3).

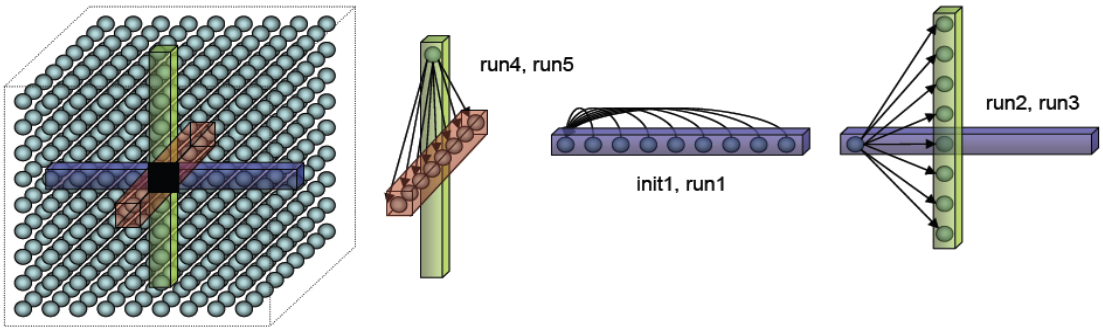

Figure 3: A representation of the 3D-DFT graph, declared and executed by the Simulator with the simulation phase names shown (init1, run1, run2, run3, run4, run5). For each transpose, all data in a 1-dimensional pencil is made available to all nodes in the subsequent pencil dimension so that each may compute its wave number of the Fourier transform directly. Despite the apparent connection redundancy, communication of data between memory spaces is not redundant, due to the Proxy Communication Rule and the Data Locality Rule.

Despite the advantage of parallelizing wave numbers calculations, this graph requires identical data be communicated to all Model graph partitions spanning the same 1-dimensional data projection (or "pencil"), and is

therefore inferior in performance to a second graph that implements the 3D-FFT similarly to a previously published implementation [14]. In this graph data is communicated uniquely to graph partitions comprising Models that compute a complete FFT (Fig.4).

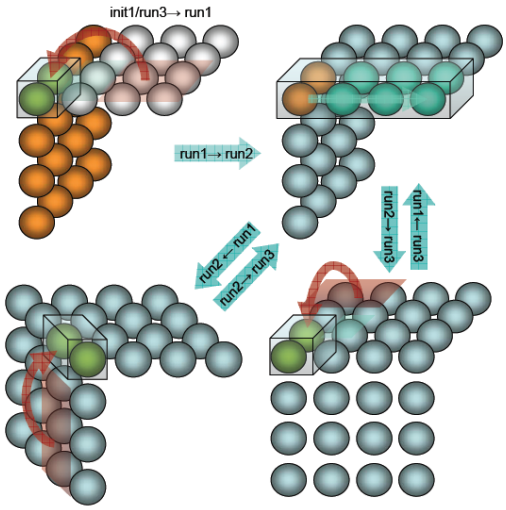

Figure 4: A representation of the 3D-FFT graph, declared and executed by the Simulator. Each 1D-FFT is computed by FFT Models represented by orange vertices. Inter-process communication (red arrows) from input Models (white vertices) or from previous 1D-FFT results (FFTelement Models; blue vertices) to FFT Models (highlighted in green) occurs on each phase boundary. Two steps for each runtime phase boundary (labeled blue arrows) are provided, since FFTelement Models are reference nodes and perform no computation. Instead, FFT Models pass data through the reference nodes (blue arrow, upper right).

Model Definitions in MDL

DFT

```
Struct Complex {
  double real;
  double imag;
}
Interface ComplexProducer { Complex* complex; }
Node DFTelement Implements ComplexProducer
{
  Complex complex ;
  Complex complexCopy ;
  Complex* [] x;
  Complex* [] y;
  Complex* [] z;
  InAttrPSet { string inputDimension; }
  ComplexProducer << &complexCopy;
  Connection Pre Node (PSet.inputDimension=="x") Expects ComplexProducer {
```

```
    ComplexProducer.complex >> x;
  }
  Connection Pre Node (PSet.inputDimension=="y") Expects DFTelement_ComplexProducer {
    ComplexProducer.complex >> y;
  }
  Connection Pre Node (PSet.inputDimension=="z") Expects DFTelement_ComplexProducer {
    ComplexProducer.complex >> z;
  }
  InitPhase initialize;
  RuntimePhase updateStateX, copyStateX (complexCopy);
  RuntimePhase updateStateY, copyStateY (complexCopy);
  RuntimePhase updateStateZ;
}
```

FFT

```
Interface ComplexArrayProducer { Complex []* complexArray; }
Model SignalElement Implements ComplexProducer {
  Complex complex;
  ComplexProducer.complex << &complex;
  InitPhase initialize();
  RuntimePhase generateSignal(complex);
}
Model FFTelementX Implements ComplexProducer {
  Complex* complex;
  ComplexProducer.complex << complex;
  UserFunction extractArrayElementReference;
  Shared { Complex []* input; }
  Connection () Expects ComplexArrayProducer {
    ComplexArrayProducer.complexArray >> Shared.input;
    extractArrayElementReference();
  }
  RuntimePhase copyStateX(complex);
}
Model FFTelementY Implements ComplexProducer {
  Complex* complex;
  ComplexProducer.complex << complex;
  UserFunction extractArrayElementReference;
  Shared { Complex []* input; }
  Connection () Expects ComplexArrayProducer {
    ComplexArrayProducer.complexArray >> Shared.input;
    extractArrayElementReference();
  }
  RuntimePhase copyStateY(complex);
}
Model FFT Implements ComplexArrayProducer {
  Complex [] fft;
  Complex* [] x;
  Complex* [] y;
  Complex* [] z;
  Shared { Complex* input; }
  ComplexArrayProducer.complexArray << &fft;
  InAttrPSet {
    string inputDimension;
    int pencilDimension;
```

```
  }
  UserFunction extractArrayElementReference;
  Connection (PSet.inputDimension=="x") Expects ComplexProducer {
    ComplexProducer.complex >> Shared.input;
    extractArrayElementReference();
  }
  Connection (PSet.inputDimension=="y") Expects ComplexProducer {
    ComplexProducer.complex >> Shared.input;
    extractArrayElementReference();
  }
  Connection (PSet.inputDimension=="z") Expects ComplexProducer {
    ComplexProducer.complex >> Shared.input;
    extractArrayElementReference();
  }
  RuntimePhase updateStateX();
  RuntimePhase updateStateY();
  RuntimePhase updateStateZ();
}
```

Graph Specifications in GSL

DFT

```
InitPhases = { init1 };
RuntimePhases = { run1, run2, run3, run4, run5 ;
ModelType DFTelement { initialize->init1, updateStateX->run1, copyStateX->run2, updateStateY->run3,
copyStateY->run4, updateStateZ->run5 };
ConnectionScript connectPencils(ModelSet ns, string dimName, int pencilDim) {
  connectSets3(ns, ns, SrcDimensionConstrainedSampler(pencilDim));
};
Dim1Connect connect1D();
DFTGranuleMapper DFTelementsGM( {32,32,32} );
Grid DFT {
  Dimension ( 32, 32, 32 );
  Layer(elements, DFTelement,DFTelementsGM);
  connect1D(.[], "x", 0);
  connect1D(.[], "y", 1);
  connect1D(.[], "z", 2);
};
DFT dft;
```

FFT

```
InitPhases = { init1 };
RuntimePhases = { run1, run2, run3 };
ModelType SignalElement { initialize->init1, generateSignal->run3 };
ModelType FFTelementX { copyStateX->run1 };
ModelType FFTelementY { copyStateY->run2 };
ModelType FFT { updateStateX->run1, updateStateY->run2, updateStateZ->run3 };
ConnectionScript ConnectPencilToPlane(ModelSet ns1, ModelSet ns2, int pencilDim) {
  connectSets3(ns1, ns2, SrcDimensionConstrainedSampler(pencilDim));
};
ConnectionScript ConnectPlaneToPencil(ModelSet ns1, ModelSet ns2, int pencilDim) {
  connectSets3(ns1, ns2, DstDimensionConstrainedSampler(pencilDim));
```

```
};
ConnectPencilToPlane connectPencilToPlane();
ConnectPlaneToPencil connectPlaneToPencil();
VolumeGranuleMapper signalDataGM("Grid SignalData's VolumeGranuleMapper", {3,3,3}, 2);
Grid SignalData {
  Dimension ( 32, 32, 32 );
  Layer(signal, SignalElement, signalDataGM);
};
FFTGranuleMapper FFTelementsGM({32,32,32}, 2);
Grid FFTElements {
  Dimension ( 32, 32, 32 );
  Layer (elementsX, FFTelementX, FFTelementsGM);
  Layer (elementsY, FFTelementY, FFTelementsGM);
};
FFTGranuleMapper FFTPlaneGM("Grid FFTPlane's VolumeGranuleMapper", {3,3,3}, 2);
Grid FFTPlane {
  Dimension ( 3, 3, 1 );
  Layer (plane, FFT, FFTPlaneGM);
};
Composite FFT_3D {
  SignalData signalData;
  FFTElements fftElements;
  FFTPlane fftPlane;
  connectPlaneToPencil(fftPlane[], fftElements[], 0);
  connectPencilToPlane(signalData[],  fftPlane[], "x", 0);
  connectPencilToPlane(fftElements[].Layer(elementsX), fftPlane[], "y", 1);
  connectPencilToPlane(fftElements[].Layer(elementsY), fftPlane[], "z", 2);
};
FFT_3D fft_3D;
```

5. Runtime and Communication Architecture

The Simulator runtime architecture provides solutions to three critical performance challenges: graph partitioning on a distributed system, parallel Model execution, and efficient communication (Fig. 5).

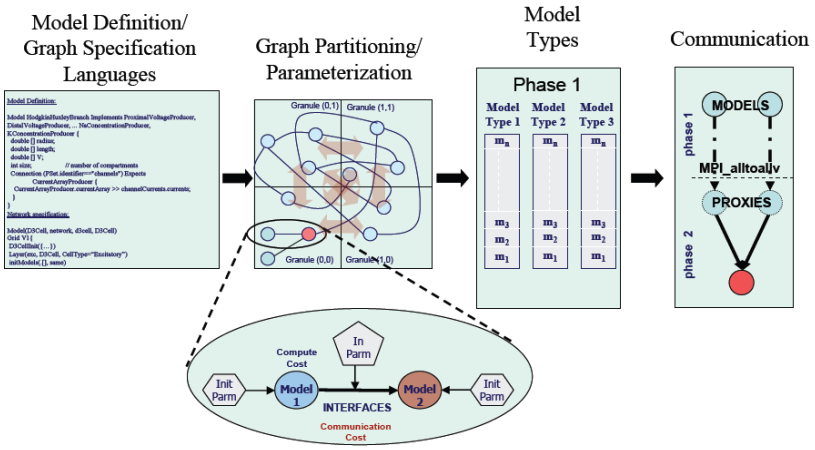

Figure 5: Architectural overview of the Simulator. Shown are representations of graph declaration, partitioning, parameterization, parallel execution, and communication (including the Proxy Communication Rule).

Graph Partitioning

Granule graph partitioning depends on the design of a Partitioner component of the Simulator system. The system supports redesign, extension, or replacement of this component depending on application. For the current work, the Partitioner queried the Blue Gene/L runtime system information in order to map granule sub-graphs onto the coordinates of the 3D communication torus based on sub-graph Grid coordinate centroids. Thus, granules of each Model type were partitioned for efficient layout on the physical system.

Parallel Execution of Model Categories

The Simulator assumes a distributed collection of uniprocessors or shared memory multiprocessor systems. Within each process or memory space, the Simulator has a class for each Model type responsible for executing all instances of that Model type. Each Model's internal phase is implemented as a single sequential process, and Model types preserve locality of reference by allocating Models together. POSIX threads spawned from runtime generated thread pools are managed by the execution engine, which binds or otherwise makes available these threads to all available CPUs. At initialization, Models are partitioned into work units for each phase, and work units assigned to the next available thread during execution.

Data Marshalling and Demarshalling for Efficient Communication

The Simulator's communication engine passes precisely one or zero messages between machine node pairs at each phase boundary. Communication between sub-graphs is managed automatically according to the Proxy Communication Rule. In order to achieve these two capabilities, the Simulator uses a generic strategy for marshalling data from Models and demarshalling data into Model proxies, implemented by MDL-governed C++ code generation. The location of data communicated at each phase boundary for each Model and Model proxy is queried at initialization and represented compactly for each system node. This memory pattern is traversed in order to copy bytes into a single send buffer and from a single receive buffer. Buffers are passed to the MPI collective `MPI_Alltoallv` together with offset and count arguments automatically generated for each phase boundary.

Together, memory pattern traversal and the use of `MPI_Alltoallv` is equivalent to the use of user defined MPI data types and `MPI_Alltoallw` [8]. Each phase's iteration reuses the same arguments.

6. Scaling and Performance

Scaling and performance was analyzed on the full 16-racks of the IBM Blue Gene/L installation at the T.J. Watson Research Center in Yorktown Heights, NY (Fig. 6). We observed scaling to 2048 processors for a dimension 64 3D-FFT and to 4096 for a dimension 128 3D-FFT. Tests were performed using Blue Gene/L's coprocessor mode and a code compiled with blrts_xlc++ and –O2 optimization. The results from the Simulator's implementation were then compared to a reference code benchmark run on the BG/L ADE communication layer instead of MPI [15]. The BG/L ADE was developed by the BG/L hardware group for machine bring-up and diagnostics. BG/L ADE provides direct access to the BG/L network hardware, while the Simulator's use of MPI is generic. Both implementations used the same parallel decomposition and identical communication patterns. Although the performance of the Simulator is far from optimal, we have identified the source of the differences and have developed strategies for mitigating them. First, the Simulator uses the `MPI_COMM_WORLD` communicator to maintain its applicability to simulating arbitrary graphs and managing arbitrary communication networks regardless of their grouping into subcommunicators. The overhead associated with this strategy over the stand-alone implementations of the 3D-FFT [14], which makes use of subcommunicators, accounts for almost all observed performance difference (Fig. 6). Second, the BG/L ADE communication layer has lower overhead than MPI [15].

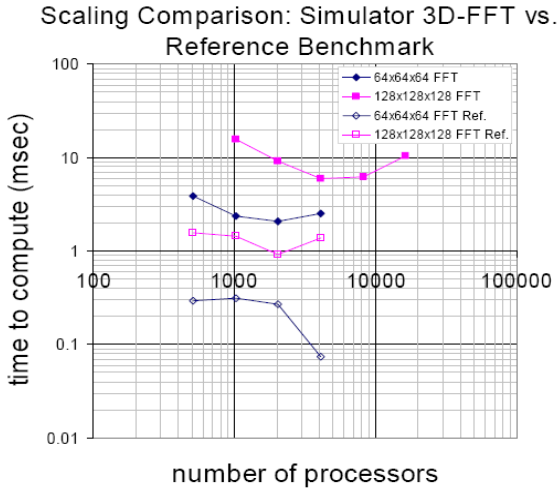

Figure 6: Runtime results for scaling runs of the 3-dimension 64 and 128 FFT on 512-16394 BG/L nodes.

7. Conclusion and Future Work

We have described a simulation system for specifying large-scale network simulations using a graph declaration language that is quite generic. We've suggested that because of it ability to specify arbitrary models and incorporate them into existing graphs consistently (a requirement from the original target domain of computational neuroscience), the Simulator is useful for implementing parallel programming applications. The graph partitioning, Model execution, and communication architectures of the Simulator have been designed to provide maximum flexibility, extensibility, and performance. We have shown two examples of implementations of a parallel algorithm and reported scaling and performance data on Blue Gene/L, comparing it to that of a stand-alone implementation of the same algorithm [14]. Although the performance of the Simulator is less than that of our reference benchmark, the design of the system allows for continued optimization and new strategies for mapping arbitrary graph topologies efficiently onto high-performance system architectures such as Blue Gene/L. We plan to address the observed overhead in our use of `MPI_COMM_WORLD` with custom collective communication based on packet-layer point-to-point messaging and automatically generated, self-optimizing communication patterns. Marshaling and demarshalling of data using automatically generated memory patterns will persist in this newly designed communication layer; however, for any given graph, the use of subcommunicators will be approximated by automatically generated, minimal point to point communications. Finally, the ordering these communications will be subjected to run time optimization.

References

1. C. C. Peck, J. Kozloski, A. R. Rao, and G. A. Cecchi, "Simulation Infrastructure for Modeling Large Scale Neural Systems," Lecture Notes in Computer Science, Computational Science-ICCS, **2660**, 713 (2003).
2. V. B. Mountcastle, *Perceptual Neuroscience: The Cerebral Cortex*, Harvard Press, Cambridge, Mass. (1998).
3. M. Djurfeldt, M. Lundqvist, C. Johansson, M. Rehn, Ö. Ekeberg, and A. Lansner "Brain-scale simulation of the neocortex on the IBM Blue Gene/L supercomputer," IBM Journal of Research and Development **52**, No. 1/2, 31 (2008).
4. H. Markram, "Bioinformatics: Industrializing neuroscience," Nature **445**, 160 (2007).

5. R. C. Cannon, M.-O. Gewaltig, P. Gleeson, U. S. Bhalla, H. Cornelis, M. L. Hines, F. W. Howell, E. Muller, J. R. Stiles, S. Wils, and E. De Schutter, "Interoperability of Neuroscience Modeling Software: Status and Future Directions," NeuroInformatics **5**, No. 2, 127 (2007).

6. B. Stroustraup, *The Design and Evolution of C++*, Addison-Wesley, Reading, Mass. (1994).

7. C. A. R. Hoare, "Communicating Sequential Processes," Communications of the ACM **21**, No. 8, 666 (1978).

8. J. Kozloski, K. Sfyrakis, S. Hill, F. Schürmann, C. Peck, and H. Markram, "Identifying, tabulating, and analyzing contacts between branched neuron morphologies," IBM Journal of Research and Development **52**, No. 1/2, 43 (2008).

9. A. W. Roscoe and C. A. R. Hoare, "The laws of Occam programming," Theoretical Computer Science 60, No. 2, (1988).

10. B. R. de Supinski1, et al., "Blue Gene/L Applications: Parallelism on a Massive Scale," The International Journal of High Performance Computing Applications, **22**, No. 1, 33 (2008).

11. IBM Blue Gene Team, "Overview of the IBM Blue Gene/P Project," IBM Journal of Research and Development **52**, No. 1/2, 199 (2008).

12. J. Kozloski, G. Cecchi, C. Peck, and A. R. Rao, "Topographic Infomax in a Neural Multigrid," Lecture Notes in Computer Science, Advances in Neural Networks, 4492, 500 (2007).

13. J. H. Reif and S. A. Smolka, "Data flow analysis of distributed communicating processes," International Journal of Parallel Programming, **19**, No. 1, 1573 (2005).

14. M. Eleftheriou, B. Fitch, A. Rayshubskiy, T.J.C. Ward, and R.S. Germain, "Performance measurements of the 3D-FFT on the Blue Gene/L supercomputer." Lecture Notes in Computer Science, Euro-Par 2005 Parallel Processing, **3648**, 795 (2005).

15. M. E. Giampapa, R. Bellofatto, M. A. Blumrich, D. Chen, M. B. Dombrowa, A. Gara, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, B. J. Nathanson, B. D. Steinmacher-Burow, M. Ohmacht, V. Salapura, and P. Vranas, "Blue Gene/L advanced diagnostics environment," IBM Journal of Research and Development **49**, No. 2/3, 319 (2005).