RC23797 (W0511-090) November 11, 2005
Computer Science

IBM Research Report

Intentional MPI Programming in a Visual Development Environment

Donald P. Pazel, Beth R. Tibbitts
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

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

Intentional MPI Programming in a Visual Development Environment

Donald P. Pazel, Beth R. Tibbitts
IBM T.J. Watson Research Center
30 Sawmill River Road
Hawthorne, NY 10532
pazel@us.ibm.com, tibbitts@us.ibm.com

Abstract

BladeRunner is a research project that explores the use of intentional and code generative approaches to facilitate programming in MPI. The BladeRunner development environment provides an interactive visual approach based on diagrams with figures representing higher level abstractions of MPI artifacts. This, along with direct manipulation of visual abstractions, allows the user to both view and express coding intentions clearly and rapidly. As a constructive programming paradigm, direct manipulation of diagrams translates into coding actions reflected into generated program code. Thus, programming focus is higher level, on visual representations for MPI programming concepts, and evolution of program state reflected in diagrams, instead of syntax or language structure details.

We describe the vision for this work, and a discussion of the BladeRunner prototype tool, built upon the ***Eclipse*** *open-source platform. This paper specifically focuses on applicability of this tool to MPI communicator construction. We discuss the conceptual interface of this tool, along with design issues regarding this intentional programming model and visual infrastructure.*

Keywords: parallel programming, intentional programming, MPI, code generation, model-driven programming, Eclipse, grid, cluster computing

1. Introduction

Grid and cluster technologies have driven and attracted significant levels of interest in parallel, concurrent, and distributed programming. These technologies enable parallel and scale-out computing architectures for applications, allowing them to capitalize through concurrency-based performance gains. High-performance programming, however, remains a high-skill art. Practitioners in high-performance middleware such as MPI[12], OpenMP[1], or PVM[6] are difficult to find in industry. Furthermore, programming tools that facilitate programming with high performance middleware are difficult if impossible to find.

To increase high-performance middleware accessibility for the general programming community, programming tools are essential. That being said, the scope of tool coverage is admittedly wide, including coding, debugging, testing, and monitoring. The problem addressed in this paper concerns facilitating parallel programming with conceptually difficult parallel programming concepts, such as those found in MPI. Therefore we argue that program construction tools provide important first steps in helping the programming community adjust to the current shift towards parallel and distributed programming.

Towards a solution, this paper presents an intentional approach [2] to effective and accurate parallel and distributed programming, focused on MPI programming. The intentional tool presented provides a diagrammatic means to efficiently and effectively assert coding intentions at a higher conceptual level. The resultant diagrams translate into lower-level MPI and programming language constructs. Our prototype program development system, BladeRunner, is presented with specific focus on specification and generation of MPI communicators, which concisely elucidate the approach and direction. BladeRunner is implemented as a plug-in under the Eclipse [13] tools platform, which includes other complementary program development tools.

This paper is organized as follows. After a review of background and related work, a discussion of MPI programming from an intentional programming viewpoint is given. This is followed by a detailed description of our tool's front-end concepts and

interface for deriving MPI communicators and code generation. This is followed by a discussion of the tool's design. A concluding summary offers thoughts on future work.

2. Background and Related Work

Programming tools' research for high-performance middleware has generally focused on distributed and parallel monitoring and debugging tools. The multiplicity of processes and hardware greatly magnifies the amount of tracing, and the complexity of sorting event interleaving is an important part of isolating and solving deep programming problems. Far less focus has been given to coding tools to help with coding phases. However, a few instances can be cited. VisualMPI[5] is a knowledge-based code generation tool with expert systems to assist in the design of process topologies. Net-Console[9] is described as a comprehensive MPI development suite using an MPI-aware editor for code development.

Related to this, usage of sophisticated static analysis techniques on MPI programming has been studied [11]. PARSE-DAT[8], while not an MPI tool, provides a sophisticated graphical process specification editor, using the pi-calculus for design and verification.

The approach to coding tools described here follows closely to that found in intentional [2] and model-driven programming schemes [14]. Our tool vision focuses on providing visual means for describing concurrent programming at a conceptual level, with the tool generating low-level coding details. We do not strive to provide a complete visual language covering all aspects of programming, but instead focus on concurrency aspects. The parallel programming model behind our work is MPI [12]. This work follows onto the state-based constructive themes found in [10] and [3].

Generally-available programming tools for parallel development and runtime monitoring on the Eclipse framework are beginning to emerge, specifically in PTP (Eclipse Parallel Tools Platform) [4], and integration with these tools is possible.

3. Intentional MPI Programming

The attraction of applying intentional techniques to MPI programming draws from the unique functionality and complexity that MPI offers. MPI, amongst the dominant middleware for high-performance computing, has the following significant features:

- Dynamic process clustering (communicators)
- Peer-to-peer messaging
- 1-N and M-N process message broadcast
- Data distribution (data typing)

That being said, MPI's programming model has the following complexities:

- An assumed SIMD process model (Single Instruction, Multiple Data) executed on MIMD (Multiple Instructions, Multiple Data) hardware.
- Byte-format and size parameter specification with overlay data typing
- Strong synchronization at the MPI communicator (process group) level

The spirit of intentional programming as found in [10] and [3] strives for clarity of coding intentions in complex programming, using visual techniques or other means. Restated, the underlying motivation is clarity in presenting and manipulating program semantic state. By *program semantic state*, we mean, for a given point in a program, its statically derivable state defined in terms of program elements (variables, fields, etc) and any assertions concerning state or values, e.g. x>=0, b==true, c points to d. *Constructive program development* involves first considering an initial program semantic state consisting of a subset of program elements and their relationships, and manipulating the program elements to form a new semantic state. The manipulation followed by the consequent state transition is *de facto* programming, which could be expressed as generated code regarding the manipulated elements. This is the essence of constructive programming: abstract program state visualization along with code generative program state manipulation.

The vision of BladeRunner is to elucidate MPI-related data and their relationships through visualizations, and by manipulating those visualizations, generate MPI code. At this level, MPI itself becomes something of a higher level abstraction, whose manipulation results in programmatic details left to code generation.

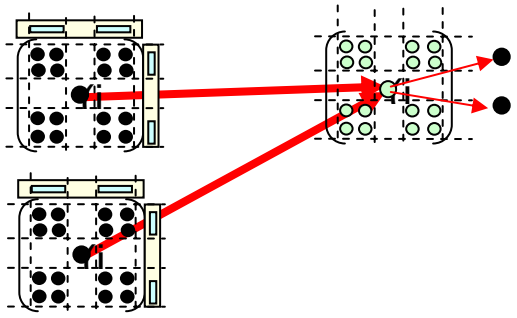


Figure 1 - Matrix broadcast on a process array

As a demonstration of what we envision for this tool, consider Figure 1, depicting an intentional specification of the distribution of two matrices (on the left) among the process members of a communicator (on the right) configured similarly as a matrix, by a master process. In MPI, a *communicator* represents, in the simplest sense, a group of mutually-aware processes than can communicate. The matrices are partitioned into equal parts as indicated by the adjoining partitioning boxes. The broad arrows indicate that each matrix partition is "sent" to a process in the communicator, which moves the data to local matrices (the two dots on the far right). With this drawing, sufficient MPI coding could be generated to fulfill the intentions of the diagram.

The spirit of intentional programming, however, is to use higher levels of abstraction than that defined by MPI; instead using domain-specific abstractions which would translate to MPI. For example, visual semantic assists for finite-element methods, Fourier transforms, and other advanced domain-specific technologies would be accessible at their own conceptual terms, and not at the level of MPI or other low-level protocol, per se. The programmer would work at that conceptual level over the lower-level. This speaks to future goals of this work.

4. A Visual Tool for Constructing MPI Communicators

BladeRunner provides a visual editor for MPI communicator creation, allowing a user to create and make choices on how to partition an MPI process space into "derived" sub-communicators. Visual representations of communicators and related artifacts form a medium not only for assisting the programmer in making partitioning choices, but also in clarifying what has already been done. This is especially helpful in illustrating how one communicator is derived from another. Visualizations help elucidate the relationships amongst communicators. Visual editing also enables derivation "experimentation", speeding the formulating of different process partitioning scenarios to see which works best for one's needs.

We now present a number of examples of communicator derivation using BladeRunner. By way of illustration, these introduce our visual construction artifacts and techniques, and convey the semantic power of this tool in practical usage.

4.1. Communicator Derivation

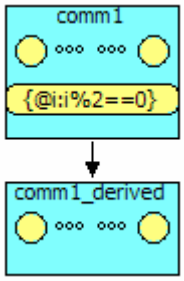
A communicator represents a set of processes of unspecified size and membership. It is visually represented as follows:

The above figure represents an abstract linearly ordered group of processes, with the first and last process designated by the large circles, and with ellipsis indicating that the number of processes is unspecified.

The derivation process involves constructing a new communicator from a given communicator, using membership qualifications to determine the new members based on the members of the given communicator. Qualifications are made through SIN expressions (Simple Index Notation, described more fully later), through segmentation, or through a combination of both.

In the following example, a SIN expression is used to derive from an existing communicator, a communicator comprised of the even-numbered processes, ala MPI rank. An expression indicates the set of even-numbered (rank) processes: `{@i: i%2==0}`. This expression representing the process subset is inserted into the communicator's visual body for later use. This is represented like this:

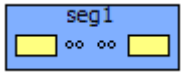
The new communicator based on that subset is derived interactively. The user initiates the derivation of a new communicator through a context menu option. This produces the new communicator from the original one, visually shown as:



Multiple derivations can be made using different subset expressions on the same communicator. For example, the complementary communicator containing the odd-numbered processes, through an odd-membership SIN expression is equally simply derived.

When deriving communicators from large numbers of processes, it may be more practical to work with the

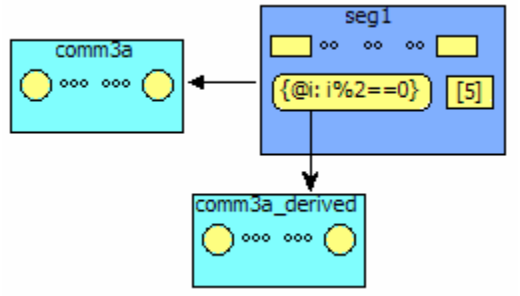
process space partitioned into a coarser set of sub-groups of processes. We introduce the concept of segmentation to facilitate partitioning a process space into process sub-groups, *segments*, of roughly equal size. Segmentation is an operator that defines a partitioning on a set of objects. In the case of a communicator, it partitions the set of processes; in the case of a segmentation, it partitions the set of segments. Visually, segmentation is represented as a block with a set of sub-blocks, representing an unspecified number of segments of unspecified sizes. Visually, it is displayed:



The number of segments and element membership per segment is determined by two numbers. One is the element population of the target. The other is either the number of segments desired, or the number of elements for each segment. These latter parameters are called the *dimension* of the segmentation, and at least one of those numbers must be identified using a SIN expression (a single SIN expression, described below).

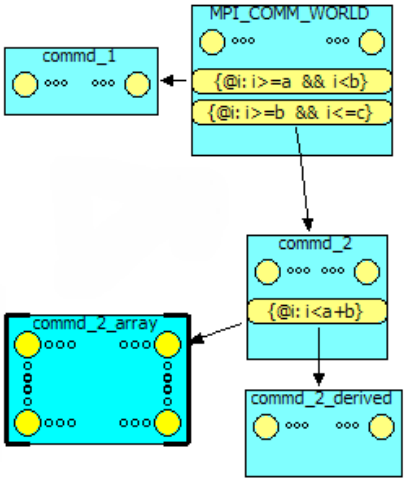
A segmentation can be qualified as either *even balanced*, meaning that the number of elements in each block is as close as possible, or as *odd balanced*, meaning that all blocks have the same number of elements, except the last one.

In the following diagram, given a segmentation applied to a communicator, and a SIN expression selecting a subset of segments from the segmentation, along with the dimension of the segmentation, a communicator is derived whose process population comprises the union of the segments expressed by the set expression. Note that the dimension is given by a variant SIN expression type, a *single expression*, representing a single value, expressed with brackets [] instead of braces {}, e.g. [5], [a+5].



The derive action is performed on the segmentation's expression, which by association is from the segmented communicator. The figure shows a new communicator, `comm3a_derived`, derived from the segmented communicator, `comm3a`.

Communicator derivations may cascade – meaning that further derivations can be built upon existing derivations. In the following example, two communicators are derived from `MPI_COMM_WORLD`, both via expressions that describe a bounded range (from a to b, and from b to c) of process ranks.



The communicator `commd_1` is derived using the first expression. The second derived communicator `commd_2` is derived from the second expression. `commd_2_derived` is derived from `commd_2` using the expression {@i: i<a+b}. An array communicator, `commd_2_array`, is derived from `commd_2`. Being a default 2-dimensional array, it uses the MPI default dimensioning protocol to determine the dimensions. These can be overridden using the Eclipse properties page relevant to this array communicator [not shown here]. Arrays are derived through a visual array operator accessible near the drawing canvas.

4.2. Simple Index Notation

An expression language called Simple Index Notation (SIN) was developed for concise designation of process or segment sets. SIN expressions denote sets of integers identifying processes by rank in a communicator, or segment indices in a list of segments. Set expressions are principally used to derive new communicators whose process collection is a subset of a source communicator.

There are two basic types of SIN Expressions. A **Single expression** signifies an integer or boolean value. Single expressions use square brackets [] and take the form `[a+b]` or `[true]`. A boolean single expression may also be defined through a relational expression such as `[!(a+b>c+d)]` (where ! indicates "not")

Examples:

- `[5]`
- `[true]`
- `[false]`
- `[!(a+b>c+d)]`

A set expression designates a collection of ordinals. Its members are integer typed with values >=0. Set expressions use curly braces {} and are of two basic forms:

- **abstract collections:** These contain the universal qualifier "for all/each", and have the form `{@i, @j : i<=j+3}` , which reads "the set of all i and all `j` such that `i<=j+3"`, (equals is `==`).
- **discrete collections:** These contain an explicit ordinal membership, and have the form `{1, 3, a+c}`, which reads "the set containing 1, 3, and `a+c"`, where a and c are defined ordinal/integer symbols. Two reserved symbols may be used in discrete collections, `first` and `last`, where `first` designates the first element of a process set, and `last` designates the last element. For example, we have `{2,4,last}` or `{first+a,5,last-3}` as discrete collections from a communicator or segment list.

Examples:

- `{1}` - this expression indicates the second (0 being first) of a set of processes or entities from a set.
- `{@i: i%2==0}`
- `{@i, @j : i<=last-(j+3)}`
- `{1, 3, a+c}`
- `{ 2, 4, last-3}`
- `{first+a, 5}`
- `{@i: i%5==3}\/({@i:i+3==d}/\{a,b,c})` – allowing unions `\/` and intersections `/\`

4.3. Code Generation

At any point in visually editing communicator derivations, the MPI code for producing them may be generated. The resulting code is effectively one or more routines whose invocations generate all the communicators, and a set of variables or access methods for each of the communicators. A sample of some of the generated code follows:

```
int routine testCES_DeriveSECD_1(MPI_Comm comm3a)
{
    //
    // Get group information
    //
    MPI_Group group0 = 0;
    int       gsize1 = 0;
    MPI_Comm_Group(comm3a, group0);
    MPI_Group_Size(group0, &gsize1);

    Segmentation seg0 = new Segmentation();
    //        Given population of gsize1, split into
columns each of depth 1
    seg0.partitionBySegmentSize(1, gsize1);
    BitVector exprV = null;

    {       // generate the process list from the
expression: {@i: i%2==0}
      BitVector set0 = null;
      //  generate the process list from the
expression: {@var0:var0%2==0}
      {
        BitVector v = new BitVector();
        for(int var0=0; var0 < gsize1; var0++)  {
          if (var0%2==0)
            v.add( var0 );
        }
        set0=v;
      }
      exprV=set0;
    }     // end of expression eval

    // Given exprV indicating the columns in segment
domain, map to columns in range
    exprV = seg0.mapToTarget(exprV);
    int [] pList = exprV.toList();
    MPI_Group newGroup2 = null;
    MPI_Group_Incl(group0,
                   pList.size(),
                   pList,
                   &newGroup2);
    MPI_Comm_Create(group0,
                    newGroup2,
                    &comm3a_derived);

    return comm3a_derived;
}
```

4.4. Eclipse Interface

A figure of the complete Eclipse workbench with the described features is show below.

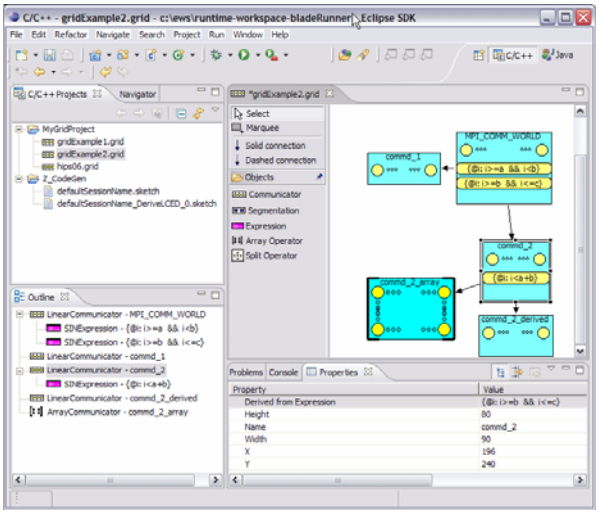


Figure 2 - Eclipse with BladeRunner

The upper-left corner shows the navigator view with file names from the projects. The upper-right shows the GEF-based visual editor. The lower-left is an outline view of all the components show visually, and the lower right is the properties view which shows detailed attributes of the selected component.

5. Architecture and Design

We describe the development infrastructure of BladeRunner, highlighting its differentiating features. An important goal in visual tool design is to maintain flexibility and extensibility in visual representations and interactions. In that regard, our design objectives employ a relatively concrete model for MPI communicators and derivations, and a relatively fluid model for visualization.

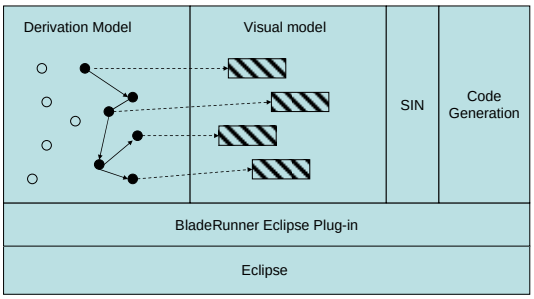


Figure 3 - Design overview

5.1. Tool Structure

The principal components of BladeRunner are shown in Figure 3. BladeRunner is built as an Eclipse extension, a set of plug-ins which extends basic Eclipse platform functionality and builds upon other available features. Eclipse and its extensions provide many tool-centric capabilities, such as editors, other source-related views, and in particular provide the graphical editor used by the BladeRunner visual editing environment.

Our extensions include the following components:

- A model for MPI communicator derivation
- A visual editor for defining and deriving communicators
- A language for process and segment collections called the "Simple Index Notation" (SIN)
- A code generation facility

The above components are created using Eclipse plug-in development frameworks, including:

- CDT – C/C++ Development Tools: For language-specific editor, parser, outlines, building tools, etc.
- GEF - Graphical Editing Framework: Provides MVC (Model-View-Controller) type framework for visual manipulation and interaction among visual objects.
- JET (Java Emitter Template): Provides template-based code generation capabilities, which is part of the EMF (Eclipse Modeling Framework) feature.

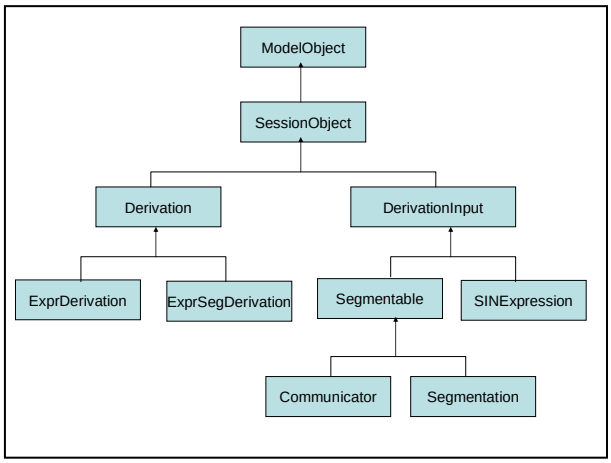


Figure 4 - Model Hierarchy

5.2. Modeling Communicator Derivation

The model for MPI communicator derivation is based on recursive communicator derivation using sub-setting or segmentation operations on communicator process spaces. More precisely, given an initial global communicator, representing all processes in the MPI application, further communicators are derived based on process subsets of that process space, then subsets of those subsets, and so on. Communicator derivation is modeled through an inheritable structure that can be specialized to different varieties of derivation, typically with input objects specifying the source communicator and qualifications for the derivation, and with a single output object, the new communicator. For example, referring to the BladeRunner design model chart in Figure 4, ExprDerivation is a derivation based on a specified subset of processes from an existing communicator, using a SIN expression to qualify the process subset. ExprSegDerivation uses a SIN

expression qualifying a set of segments of processes in the derivation.

Several conceptual entities were added to the derivation model to assist in qualifying the input objects for a derivation, particularly segmentation and specification of sets of processes via SIN expressions. While these are not MPI concepts, they are generally useful concepts which facilitate the specification of process sets, and provide leverage in specifying process partitions. Segmentation is a particularly strong abstraction, not only in that it can be applied to segmenting a communicator process group, but also that it could also be applied recursively with other segmentations, i.e. segmentations of segmentations.

The model structure shown in Figure 4 shows how we abstracted these concepts into the model design. By introducing the abstract concepts DerivationInputObject and SegmentableObject, we provide a fairly general inheritable means for specifying different kinds of input for a derivation.

The model also includes a notification framework used for broadcasting model changes both to objects within the model and to visual objects.

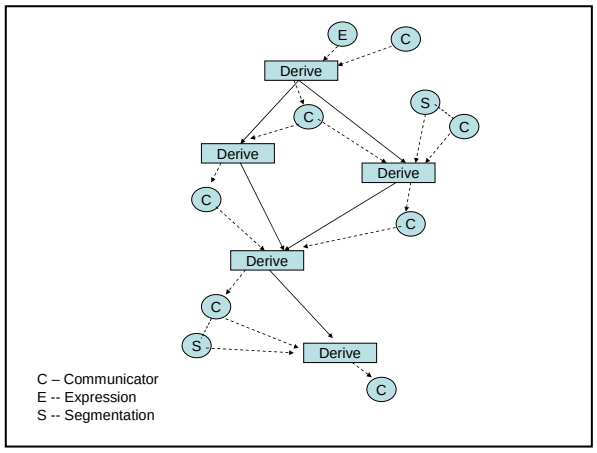


Figure 5 - Derivation Graph

5.3. Visual Interaction and Reflection

The model-visual interaction, based on MVC (Model-View-Controller) architecture [7] is an interaction-notification scheme in which visual objects (View) directly invoke model objects (Model) based on constructive demands (Controller), and visuals respond to notifications reflecting updates to a model. Each model object reflects an edit-state for the object. This state is updated with changes to the model, and reflected back to the visual entities as notifications. Event notifications are central to the communication infrastructure of the Eclipse plug-in hierarchy. Relevant detail pertaining to state is reflected in the visuals: for example, error states when elements have been deleted that were engaged in a derivation. The Eclipse GEF (Graphical Editor Framework), in which this is implemented, provides the MVC basis architecture for BladeRunner.

5.4. Derivation Graph and Code Generation

Code generation is based on the order dependency of communicator creation/derivation. Since communicators can only be derived from communicators that already exist, the derivations take the form of dependency relations, with some derivations necessarily occurring before others. The dependency relationships impose a directed acyclic graph (DAG) structure, which we call a derivation graph. An example is shown in figure 5. The nodes of the graph are derivations, with directed arrows indicating order precedence of derivations. Other nodes are represented showing other kinds of input, and output from each derivation. Join nodes can result, for example, from union operations on derived communicators. This graph is produced though a topological sort over the model representations of existing derivations.

Code generation is achieved though a simple order traversal of the DAG. Based on the type of each derivation node encountered, a specific code generation template is invoked to encode the derivation in a specific target language. Parameters such as source communicator, SIN expressions, and segmentations are use as input to the templates. Presently, we generate a C-style output using the Java Emitter Template (JET) template technology found in Eclipse EMF (Eclipse Modeling Facility). JET provides a format for expressing direct code generative expressions, using variable substitution in combination with embedded language coding. While we presently are focused on C, it would be relatively easy to produce code generation templates for different target languages, such as Fortran.

6. Summary and Further Work

The BladeRunner tool is presented with a focus on the intentional specification of MPI communicators. We use graphical depictions of communicators representing sets of MPI processes, and derive other communicators from them for application specific tasks. After derivations are made, program code is generated that implements the runtime derivation of

the MPI communicators used in the main tasks. The user can then insert specific code for solving tasks using the communicators.

For the broad vision for the tool, the following directions are of interest:

- Message programming: Visual specification of 1-N and M-N messaging within a communicator, such as send/receive, scatter/gather, broadcast, etc.
- Data typing: Visual specification of MPI data types, and visual manipulation of these into visual message programming
- Data partitioning: Visual means to indicate how application data is to be partitioned amongst a set of processes for scatter messaging
- Shared memory: Visual specification of shared memory and its manipulation
- Templates for code generation in other languages, such as FORTRAN, another common vehicle for MPI programming.

We are also interested in semantic tools for MPI coding, such as tools to assist in viewing and structuring existing MPI code, using code analysis and transformation techniques.

6. References

[1] Chandra, R., Menon, R., Dagum, L., Khor, D., Maydan, D., McDonald, J., Parallel Programming in OpenMP, Morgan Kaufmann, 2000.

[2] Czarnecki, K., and Eisenecker, U.W., *Generative Programming*, Addison-Wesley, 2000.

[3] H. Derby, R.M. Fuhrer, D.P. Pazel, "MindFrames: A Visual Environment for Semantically-Oriented Program Construction", IBM RC22739, 2002.

[4] http://eclipse.org/ptp

[5] D. Ferenc, J. Nabrzyski, M. Stroinski, P. Wierzejewski, "VisualMPI—A Knowledge-Based System for Writing Efficient MPI Applications", PVM/MPI'99, Springer-Verlag, 1999.

[6] Geist, A., Bequelin, A., Dongarra, J., Juang, W., Manchek, R., Sunderam, V.S., PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing, The MIT Press, 1994.

[7] G.E. Krasner, S.T. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System", ParcPlace Systems, 1988.

[8] A. Liu, I. Gorton, "PARSE-DAT: An Integrated Environment for the Design and Analysis of Dynamic Software Architectures", International Symposium on Software Engineering for Parallel and Distributed Systems, Kyoto, 1998.

[9] A. Papagapiou, P. Evripidou, G. Samaras, "Net-Console: A Web-Based Development Environment for MPI Programs", PVM/MPI'99, Springer-Verlag, 1999.

[10] D.P. Pazel, "The Effigy Project – Moving Programming Concepts to a Visual Paradigm", Visual End User Workshop at VL2000, Seattle, 2000.

[11] D. Shires, L. Pollock, S. Sprenkle, "Program Flow Graph Construction for Static Analysis of MPI Programs", PDPTA, Las Vegas, 1999.

[12] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J., MPI—The Complete Reference, Vol I & II, The MIT Press, 1998.

[13] http://www.eclipse.org

[14] http://www.omg.org/mda/