# A FORMAL CSP FRAMEWORK FOR MESSAGE-PASSING HPC PROGRAMMING

J. Carter, W.B. Gardner
*Department of Computing and Information Science*
*University of Guelph*
*Guelph, ON, N1G 2W1, Canada*
{jcarter,gardnerw}@uoguelph.ca

## Abstract

*To help programmers of high-performance computing (HPC) systems avoid communication-related errors, we employ a formal process algebra, Communicating Sequential Processes (CSP), which has a strict semantics for interprocess communication and synchronization. Verification tools are available for CSP-specified programs to prove the absence of failures such as deadlock, and to explore potential multiprocess interactions. By introducing a CSP abstraction layer on top of the popular MPI message-passing primitives, we create a framework, called CSP4MPI, designed to largely hide the complexity of parallel programming for HPC.*

*CSP4MPI is comprised of a C++ class library that provides a CSP-based process model, and a "cookbook" of candidate solutions for HPC programmers not trained in CSP. Developers can prototype their systems using CSP, and use verification tools to examine possible points of failure before implementing via the CSP4MPI library. Alternatively, they may choose an existing, verified solution from a number of common parallel application archetypes. By using CSP4MPI, HPC developers leverage the benefits of formal specification and verification in their work, in addition to obtaining an alternate method to developing HPC applications.*

***Keywords:*** *HPC, MPI, parallel patterns, CSP, selective formalism.*

## 1. Introduction

High-performance computing (HPC) is typically based on clusters of computer workstations. Clustering makes available computational resources on par with the world's biggest supercomputers. However, researchers wishing to tap this power face a considerable challenge. It involves mastering the programming tools for the creation and control of parallel processes on distributed hosts. A popular package for HPC is Message Passing Interface (MPI) [1, 2], whose application program interface features over 150 functions—a significant learning curve. A seasoned concurrent programmer has experience that assists in recognizing and minimizing overhead in parallel computations, such idleness, extraneous computation, and unnecessary communication. But time spent translating a sequential algorithm to a parallel algorithm is greater for a researcher lacking background in algorithms and parallel programming. Such researchers do not always have the benefit of a computer science background, and so may be ill-equipped to confront common multiprocessor programming hazards such as deadlocks and race conditions.

We present an alternate route to development of parallel applications for MPI clusters by means of the process algebra Communicating Sequential Processes (CSP) [3]. CSP as a specification language offers a comfortable level of abstraction when placed alongside syntax-rich programming languages. The ability to formally verify correctness (such as the absence of deadlock) in CSP is a virtue, saving execution time spent running problematic solutions on machines with many users vying for scheduling.

Our tool for HPC programming, called CSP4MPI, is built upon a library of previously verified communication patterns [4] to which users can connect custom functions that perform useful calculations. These efficient patterns help to maximize the computation to communication ratio [5], the metric by which one judges the effectiveness of a parallel application against its sequential counterpart. For novice HPC programmers, this saves development efforts spent on inefficient implementations, and time spent tracking down difficult communication and synchronization errors.

Alternatively, users knowledgeable in CSP have the ability to create their own patterns. In either case, the CSP-specified code constitutes a control backbone that invokes user-coded functions which carry out the desired calculations. This arrangement—mixing a formally-specified control structure with non-formal program code—is called *selective formalism,* and is key to our approach [6].

In the following section, we motivate our solution by focusing on the problem of communication errors in parallel applicatons. Then we explain how our solution is based on selective formalism, as specifically applied in the context of MPI programming. We show how an HPC program utilizing CSP4MPI works, and explain how both novice HPC programmers and developers experienced with CSP can take advantage of our approach. Finally, we discuss the prospects for further automating programming with CSP4MPI in the future.

## 2. Errors in Parallel Applications

In presenting arguments for the application of CSP to HPC

computing, we observe two types of errors that arise through incorrect parallel programming: computational errors and communication errors.

### 2.1. Computational Errors

A computational error is a flaw in a section of code that yields an incorrect result. Culprits for computational errors range from using the assignment ('=') operator to test for equality, operator precedence issues, overflow errors, or numerical precision errors due to improper floating point calculations. In most cases, computational errors cause similar (if not identical) results during multiple executions of the program, making traditional debugging techniques, such as the introduction of print statements or inspection via a debugger, an effective means for diagnosis.

### 2.2. Communication Errors

The second, and more insidious, type of error in parallel programming is the communication error. We define this to be an error arising in interprocess communication or synchronization leading to premature termination of execution. Communication errors are the result of undesirable conditions such as deadlock, livelock, or race conditions, resulting from incorrectly designed process synchronization, unrealistic assumptions about communication buffer sizes, or incorrect use of blocking operations.

In our experience, communication errors require greater time and knowledge to identify and debug. A challenge to communication error debugging is that such errors may not be present in successive executions due to variables such as buffering or scheduling. The nondeterministic nature of communication errors causes them to be difficult to pinpoint, and frustrates developers unaccustomed to concurrent programming. Our thesis is that by building HPC applications with the aid of a formal semantics for interprocess synchronization and communication, developers should be able to build programs whose communications are “correct by construction.”

## 3. CSP and Previous Work

CSP is a process algebra for describing systems composed of a number of parallel components, and specifying the ways in which they may communicate and synchronize. CSP is formally verifiable using a number of software tools. For example, Formal Systems Europe's FDR2 [7] is able to prove the absence of deadlock, livelock, and non-determinism, and make assurances about process behavior by examining a system’s traces and failures for safety properties. Our research has used CSP as an input language for C++ software synthesis via the framework CSP++ [8, 9].

Aside from our own prior experience, years before the advent of MPI, Mazzeo et al [10] suggested using CSP as a means to program a cluster of heterogeneous workstations. Explaining the development of their DISC system, they argued in favour of CSP as follows:

- CSP is simple but powerful.
- CSP is not restricted to any usage patterns, such as master-slave.
- CSP allows for an object-oriented style of component composition and reuse, and abstracts away details such as scheduling or resource contention.
- CSP constructs can be created using existing programming languages.

The DISC system implements clustering using the CSP computational model as a means for programming parallel applications on a network of heterogeneous UNIX workstations. DISC allows for an application to be developed on a single workstation and be executed using computational resources of other DISC-enabled machines on its local area network. DISC development is conducted via a custom parallel compiler, linker, run-time environment, system monitor, profiler, debugger, makefile generator and graphical user interface.

The DISC system was developed with the intention of supporting a developer through the entire development cycle, as opposed to merely offering a parallel compiler and run-time environment. DISC computation is performed through a set of processes, in turn composed of events, each representing a step in a calculation. Interprocess communication is performed exclusively via channels, noting that DISC made use of many-to-one channels (an extension of the CSP channel definition). DISC also offers channel inheritance whereby a child process has access to the channels of its parent process.

DISC is an undeniable ancestor of contemporary clustering and grid computing packages, of special interest due to the usage of CSP. In that period, predating joint standards such as PVM or MPI, researchers were creating parallel programming languages based on their platform and computing model of choice.

DISC contrasts with CSP4MPI in that it does not partition between computation and communication, but rather combines the two through its syntax. Instead of our selective formalism approach, DISC merely provides an infrastructure through which communication and synchronization must take place with calculation interspersed throughout program code. Compared to contemporary parallel programming methods—PVM, MPI, OPENMP—DISC is considerably more heavyweight and platform dependent.

## 4. Applying CSP to HPC

### 4.1. Selective Formalism

After settling on a formal method, the next question is how to inject the formalism so as to obtain the benefits of its semantics, but without creating fresh drawbacks. That is, given the learning curve that novice HPC programmers already face, we do not want to add the burden of learning a formal notation.

Formal methods are frequently applied in industries building mission-critical systems where insufficient verification can risk

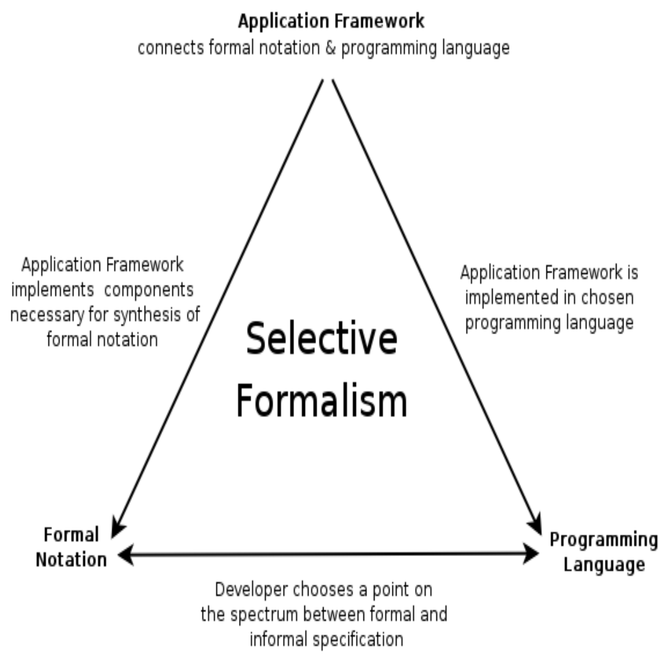


**Fig. 1. Ingredients of Selective Formalism**

human injury or large-scale financial loss. Software developers working in such areas must prove a system to be correct and safe. In contrast, most software development is carried out unfettered by formal specification and verification, in a process that is likely faster and cheaper than the formal route, but not necessarily more correct.

Figure 1 portrays a spectrum of software design rigour that ranges between two poles. On the right is non-formal software development in a conventional programming language. The polar opposite, on the left, is development using rigorous specification and verification methods based on a formal notation. Striking a balance between these two poles is possible via "selective formalism."

Selective formalism allows developers to select critical components of a system to be specified formally, and secondary or non-critical components to be coded using a conventional non-formal programming language. Developers select a point on the continuum between total formal specification and non-formal development that is appropriate to their project.

The three ingredients of selective formalism, illustrated in Figure 1, are:

- a formal notation capable of expressing the system.
- a programming language of interest to developers and practical for the application.
- an application framework to connect constructs from the formal notation to the chosen programming language.

Through selective formalism we partition the realm of communication and computation, and off-load the responsibility of designing communication patterns from the novice HPC developer.

## 4.2. Anatomy of a CSP4MPI Application

Referring to Figure 2, the first level of a CSP4MPI application is the physical interconnect. Interconnects range from off-the-shelf Ethernet networks to low-latency direct memory access (DMA) products. Given that CSP4MPI is not tied to any particular interconnect type, a discussion of interconnects is outside the scope of this work. Next to the interconnect layer is the MPI implementation, which may have special configurations or optimizations for the interconnect or processor architecture. Many MPI implementations are available, but our development efforts have focused on LAM/MPI [11], due to our available in-house HPC systems.

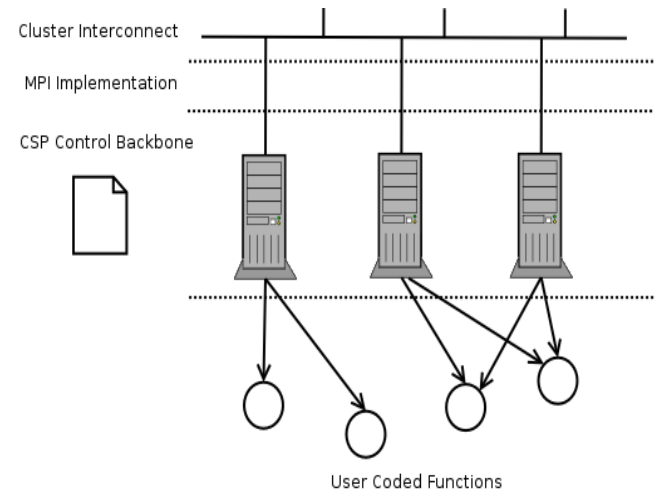


**Fig. 2. CSP4MPI Architecture**

CSP4MPI builds on the previous two layers, common to any message passing cluster. On top of the MPI implementation is the "CSP Control Backbone" layer comprised of a C++ class library, implementing CSP primitives such as processes, events, channels and their respective operations. A process describes a sequence of events to take place, including interprocess communication, synchronization or the calling of user-coded functions (discussed next). An event is a single atomic operation executed by a process, and multiple processes may rendezvous on the same event, as a form of barrier synchronization similar to MPI_Barrier(). A channel is an unbuffered, unidirectional, point-to-point communication mechanism with "producer and consumer" end points. Channels are the sole interprocess communication mechanism used by CSP, and likewise the only interprocess communication method available through CSP4MPI.

CSP stipulates that a process must recurse as another process, including SKIP which describes successful termination, or STOP which is used to denote a state in which no additional progress may be made. For CSP4MPI, SKIP and STOP serve to force a node to wait for its peer nodes to complete, upon which execution of the entire application is complete.

Aside from CSP primitives, the CSP4MPI control backbone layer implements a number of utility classes, the largest being the Environment class that maintains the process table, variable

space, synchronization, and event notification for every computational node in the system.

Under the control of the CSP control backbone are user-coded functions a developer has added to an implementation. user-coded functions are implemented as function pointers attached to events. user-coded functions are called when the corresponding event occurs in the control backbone, reading and writing data from the node's individual process space.

Since CSP4MPI is layered upon the MPI library, a CSP4MPI program has many similarities to an MPI program. Next, we describe what happens when a CSP4MPI program is executed.

The following steps occur for all nodes:

- Step 1. Environment Initialization
  - MPI is initialized with command line arguments.
  - Nodes are assigned individual names.
  - CSP channels necessary for communication are created between the nodes previously named.
  - A separate communicator group is created for system-wide event notification. This offers a “priority lane” for such notification.
- Step 2. Process Construction and Registration
  - Empty processes are created with individual names.
  - Events and channel I/O operators are added to processes.
  - User-coded functions are attached to events.
  - Synchronization events are inserted where required between multiple processes.
- Step 3. Execution
  - Individual nodes begin execution with the process for which they are named. Execution continues until the process (and any following processes) terminate.
- Step 4. Environment Finalization
  - As nodes terminate, destructors are called, and MPI_Finalize() is invoked.

# 5. Using CSP4MPI

CSP4MPI is aimed at two user groups, shown to the left in the use case illustrated in Figure 3.

## 5.1. Novice HPC Programmer

The novice HPC programmer needs no prior CSP experience to benefit from our approach, as CSP is kept “under the hood,” so to speak. From the library of control backbone templates, users select appropriate templates for their problems and these control backbones lay out the interprocess communication and synchronization to be used.

Once a communication template is in place, the novice HPC programmer creates user-coded functions in C/C++ to perform the calculations or “real work.” The developer compiles the functions along with the selected control backbone, creating an HPC application.

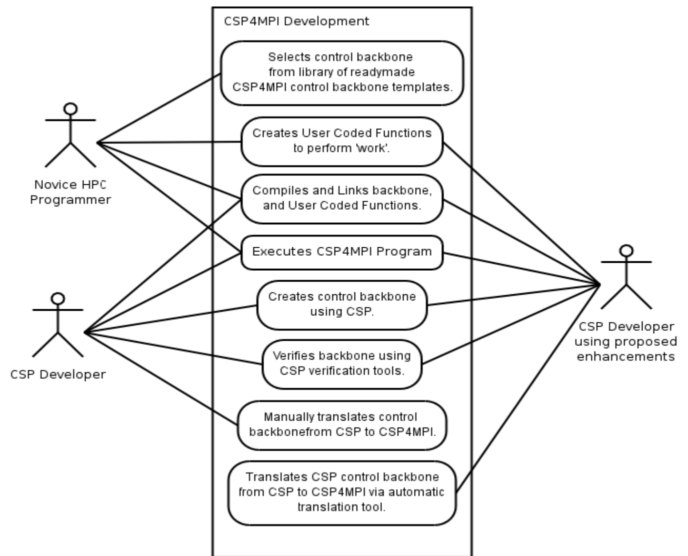


**Fig. 3. CSP4MPI Developer Use Case Diagram**

User-coded functions are written in a manner akin to functions written for a non-HPC program—i.e., sequential execution—with the stipulation that user-coded functions may not “go behind the back” of the control backbone and perform interprocess communication or synchronization on their own. Through this restriction, and in moving all communication and synchronization to a formally-verified control backbone, the novice HPC programmer is relieved from having to debug communication errors. Any remaining computational errors can be found using traditional debugging techniques.

## 5.2. CSP Developer

It is unlikely that a library of control backbones is capable of satisfying the needs of advanced HPC developers, so those wishing to create their own control backbones are free to do so. A previous knowledge of CSP is necessary, making this approach attractive to existing CSP users.

The current design flow for a CSP developer begins with the specification of the control backbone in CSP. Given a CSP specification, the developer carries out simulation and refinement. When satisfied with the functionality and properties of the CSP backbone, the developer implements the CSP specification in the CSP4MPI library, and adds the user-coded functions necessary for the application.

The process of hand translating from a CSP specification is the weakest link in the present CSP4MPI design flow, and this step may potentially introduce errors due to improper translation. Our existing CSP++ tool boasts automatic translation from CSPm (a machine readable dialect of CSP) to C++, and this capability will be incorporated as the CSP4MPI library matures.

The use case for this proposed future enhancement is shown on the right side of Figure 3.

# 6. Status of CSP4MPI

## 6.1. CSP4MPI Case Studies

At present two simple case studies exist, implementing two common parallel processing patterns [4]. The first of which is the master-worker pattern. The master node distributes work from an input file among a finite number of worker nodes, collecting and combining worker results as they become available. Communication is provide via the CSP verified control backbone. Values are read from an input file, processed on nodes, combined and outputted using user-coded functions.

The second case study is a pipeline parallel implementation, whereby several stages are composed in sequence. The initial stage reads data from an input file, and the final stage outputs to a results file (both stages make use of user-coded functions for this). Intermediate stages process data using individual user-coded functions to perform their processing tasks. As with the master-worker pattern, communication between stages is derived from a formally verified CSP model.

## 6.2. Future Work

There are three areas that stand to be expanded for the CSP4MPI project:

- CSP4MPI will require an extensive survey of patterns for parallel programming, and a set of relevant communication templates constructed along with documentation and code examples outlining recommended usage of templates. These templates serve as communication backbones for developer projects, and therefore must be formally verified using the existing CSP tools.
- At present, CSP4MPI has little performance optimization in place. As the code base matures, some benchmarking on a production cluster will be performed, followed by necessary code optimization and improvements. A comparison of CSP4MPI solutions vs. 'pure' MPI solutions is also planned.
- As the CSP4MPI implementation solidifies, there are prospects for retargeting CSP++'s CSPm-to-C++ translator to generate code for CSP4MPI. This will eliminate the need for hand-translating parallel patterns from CSP into C++.
- Finally, CSP4MPI is built exclusively using LAM/MPI [11]. Testing with alternate MPI libraries will be conducted.

# 7. Conclusion

We have presented our work to date on the application of communicating sequential processes to message-passing high performance computing. Our selective formalism approach partitions developer errors into two domains, allowing traditional debugging to take place for computational errors, and for burdensome communication errors to be avoided through the use of a template library for novice HPC developers, while offering the assurances of formal specification and verification of communication to advanced users.

# Acknowledgements

The initial development work on CSP4MPI was carried out on SHARCNet (Shared Hierarchical Academic Research Computing Network).

# References

[1] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 878–883, New York, NY, USA, 1993. ACM Press.

[2] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard 1.1. June 1995. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[3] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[4] B.L. Massingill, T.G. Mattson, B.A. Sanders. *Patterns for Parallel Programming*. The Software Patterns Series. Addison Wesley, Boston, MA, USA, 1st edition, Sept. 2004.

[5] B. Wilkinson, M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[6] W.B. Gardner. Converging CSP Specifications and C++ Programming via Selective Formalism. *Trans. on Embedded Computing Sys.*, 4(2):302–330, 2005.

[7] Formal Systems Europe Ltd. FDR2 Homepage. March 2006. http://www.fsel.com/.

[8] W.B. Gardner. CSP++ Homepage. 2000. http://www.cis.uoguelph.ca/ wgardner/. Research Link.

[9] W.B. Gardner. CSP++: How Faithful to CSPm? In *Communicating Process Architectures 2005 (WoTUG-27)*, Concurrent Systems Engineering Series, pages 129–146. IOS Press, 2005.

[10] A. Mazzeo., S. Russo, G. Ventre. Using CSP languages to Program Parallel Workstation Systems. *Future Gener. Comput. Syst.*, 8(1-3):149–163, 1992.

[11] LAM/MPI Parallel Computing Homepage. March 2006. http://www.lam-mpi.org/.