Scaling Tuple-Space Communication in the Distributive Interoperable Executive Library

Jason Coan, Zaire Ali, David White and Kwai Wong

August 18, 2014

Abstract

The Distributive Interoperable Executive Library (DIEL) is a software framework designed to configure and execute a series of loosely coupled scientific modules in parallel. The DIEL is capable of running many existing users' codes on high performance computing machines such as Darter and Beacon. It includes an interface for inter-process communication that is controlled via a simple configuration file. Currently provided are direct data exchange using user-defined shared boundary conditions and a prototype of indirect data exchange via a global tuple space. Our task is to improve and extend the tuple space implementation to make it a viable and efficient method of communication.

Chapter 1

Background

1.1 Loosely Coupled Systems

A loosely coupled system is one in which each module solves its part of the overall problem without knowing about the internal workings of any other module. By extension, the module cannot depend on any other module working in any specific way, and should another module be removed, modified, or swapped with another, the first module can still operate. The only dependency a module has on the system is that the system somehow provides it with the type of data that serves as input to its primary computational functioning, but the source of that input does not matter to the module.

In such a system, the points on the shared boundaries between modules represent only computational input and output that is directly related to their primary abstract functions. For example, a loosely coupled module calculating escape velocities of astronomical objects could receive only *mass* and *distance to center of gravity* as inputs and produce only *escape velocity* as output. In this case, we say that the boundaries of this module's domain have three conditions on them to share with other modules.

This program design results in highly reusable code. If our escape velocity module meets the above specifications, it can fit into almost any system that needs escape velocities to be calculated. Over the course of multiple projects using this design, the amount of redundant work is reduced dramatically. Loose coupling also reduces the overall complexity of the system and makes it easier to debug. And, of course, it allows for the problem to be solved efficiently on a parallel computer by producing algorithms that can do their work independently in dedicated threads.

1.2 The Distributive Interoperable Executive Library

The DIEL seeks to encourage and facilitate the creation of loosely coupled systems on high-performance computers. It does this by providing a framework for identifying the modules of the system and their shared boundary conditions. Library functions execute the configured modules and expose to them a small, simple API for passing data along the shared boundary conditions. The system is configured with a simple configuration file that is parsed by the "executive", which compiles and broadcasts the configuration information to the modules.

The library seeks to be as lightweight as possible. It is designed with the assumption that users' modules will themselves spawn parallel processes that pass messages between themselves, and then the DIEL communication API will be used to exchange data at a higher level of abstraction. For this reason, the DIEL should minimize the amount of overhead associated with the use of its communication functions to avoid clogging the HPC interconnect.

The DIEL provides two methods of communication. The first is synchronous, one-to-one message passing via wrappers for MPI_Send and MPI_Recv. These wrappers enforce the loose coupling information in the configuration file and check for and handle any MPI errors. The second is asynchronous, anonymous, many-to-many message exchange via a global tuple space. This report will trace the development of the tuple space from a previously-existing prototype to its current state as a viable method of communication.

Chapter 2

Goals

2.1 Understanding the Requirements

> "A supercomputer is a device for turning compute bound problems into I/O bound problems" – Ken Batcher

A tuple space is associative memory that can be accessed concurrently. It was invented by David Gelernter as the basis for his Linda programming language as an alternative to the message passing and shared memory paradigms[**?**]. There are several mature implementations of tuple spaces, such as JavaSpaces and T-Spaces, but these are designed for distributed systems over wide-area networks and the World Wide Web[**?**]. The requirements for those systems include fault-tolerance and redundancy. Since the compute nodes of these systems are remote from each other and may be under the control of different people, the designers must anticipate nodes occasionally disappearing and define procedures for continuing operation despite this. Geographic location is also important, as there will be very high latencies in accessing a tuple that is geographically far away. For this reason, one would want copies of every tuple spread evenly throughout the system, to minimize the distance between each node and each tuple. These requirements lead to large overhead from using the tuple space. Messages must be continually passed between every tuple space server to ensure the coherency of tuple copies and to check for node disappearance.

Given the truth in the quote by Ken Batcher above, a tuple space for an HPC cluster must meet a different set of requirements. Nodes in a supercomputer do not often fail, and when they do, the whole system should stop so that the problem can be addressed. Also, the latencies associated with I/O cannot be addressed by reducing physical distances with redundant copies, as modern interconnects are already designed to minimize the distance between one node and any other node as much as is currently possible. The way to maximize communication performance in this scenario is to minimize the number and size of messages being passed and to prefer non-blocking procedures over blocking ones. So, fault-tolerance and redundancy are left out.

2.2 Existing Prototype

A prototype of tuple-space communication existed before we arrived. It consisted of a single server processing "get" and "put" requests in sequence, dynamically storing committed tuples in a linked list. It was a special function called on MPI rank 0 by the executive. It was an integral part of the DIEL as a whole, which contradicted our stated goal of developing a modular system. Since a tuple space is, by definition, concurrently accessible, it did not represent a true tuple space implementation. Rudimentary associativity was implemented, but this consisted of the user arbitrarily assigning a tag to each tuple when putting it to the server.

2.3 Desired End

In order to create a fully modular system, we thought it best to convert the tuple space into a DIEL module like any other. This way, the tuple space can be swapped with a modified version or entirely new implementation without having to recompile the DIEL. Once we accomplish this, we need to be able to execute multiple instances of this module as we would any other module. Each of these tuple servers should be able to control an even portion of the overall tuple space. Other module processes should be able to use a hash function to discover the proper server for a specific tuple, creating a distributed hash table from the tuple servers. The input to the hash function should be the associativity data for the tuple, which should correspond to the same shared boundary conditions defined in the configuration file.

Chapter 3

Development

3.1 A Distributed Hash Table

In a hash table, a hash function calculates the proper index for data element based on its associated key. In a distributive hash table, the hash function returns the proper node as well as the index on the node. This means we do not need to pass messages between multiple processes just to find out where our data element is located.

Each of the shared boundary conditions in the configuration file is assigned an integer-value ID. The hash function uses modulus to determine the correct tuple server, and again to determine the correct index:

SBC_ID mod NUM_SERV = server

SBC_ID mod NUM_IDX = index

DIEL modules have two functions for interacting with the tuple space:

Producer: IEL_tput(&data, size, sbc)

Consumer: IEL_tget(&data, &size, sbc)

Since the hash function always returns the same values for the same input, if IEL_tput and IEL_tget both call the hash function, they will get back the same location. Thus, they will look in the same place without directly communicating with each other.

3.2 Anticipating a Stochastic Process

A major challenge with most parallel systems is that they are, from the programmer's point of view, nondeterministic. By this we mean that the actual sequence of events from the point of view of the entire system will usually be different every time the program is run because every process is individually subject to a large number of uncontrollable variables. A robust tuple server

algorithm must be able to anticipate and handle all possible sequences short of a catastrophic hardware failure.

For example, consider having a producer module and a consumer module. The producer module is delayed by the operating system, and the consumer calls IEL_tget on the relevant data before the producer calls IEL_tput. So, the tuple server is faced with being asked for data that it does not have. When we started development, the existing tuple server algorithm could not handle this case. The system would become deadlocked and never complete.

We modified the server's algorithm so that it can continue receiving requests from other processes after the above situation arises, giving the producer a chance to put its tuple to the server so the consumer's request can be fulfilled later. It does this by sending a message back to the consumer that the tuple was not found. Since IEL_tget is currently a blocking procedure, the consumer will now wait until it receives another message from the tuple server. In the meantime the tuple server stores the MPI rank of the consumer in a sparse array indexed by the SBC_ID that it requested. Whenever the tuple server receives any tuple via IEL_tput, it checks the sparse array to see if any process is currently waiting on that SBC_ID. If so, it will complete the requests at this time.

3.3 Testing

After developing a working implementation that meets our requirements, we should test the system to make sure all requests are being fulfilled correctly and the tuple server algorithm is thread-safe. The code for our test is located in the Appendix, but what follows is an outline of the algorithm.

3.3.1 A Randomized Stress Test

- Do this ten times, with ITER starting at 0:
  - Send your rank id to the tuple space using your rank ID plus ITER as the input to the hash function
  - Do this until you are done:
    - Based on the number of module processes, pick a rank ID at random
    - Request that ID from the tuple space, using the ID plus ITER as the input to the hash function
    - Repeat until you have received every rank ID in the system, including your own, at which point you are done
  - Increment ITER and repeat

3.3.2 Results of Test

Due to the randomized nature of the test, we should run it many times and then look at the distribution of completion times. We expect this distribution to have no outliers, and any outlier probably signifies a problem. Also, the module needs to check that every time it receives a message through IEL_tget, the tuple is receives is actually the one that it requested.

We ran this test with 16 tuple servers and 256 modules processes on Darter. After 40 trials, the tuple servers collectively fulfill an average of 9.6 million tget/tput requests per trials. Every request is fulfilled correctly. It takes an average of 7.5 seconds to complete, with no obvious outliers.

3.4 Future Development

While the long-term goals of the DIEL in-general are outside the scope of this report, we can name the next steps of tuple-space development specifically:

First, each shared boundary condition represented in the tuple space should have its own dedicated underlying data structure that can act as a queue, stack, generic set, etc. to provide more options for user code.

Also, to match the functionality of most mature tuple space implementations, we should provide both blocking and non-blocking versions of IEL_tget. Our current function is completely blocking: once it requests a tuple from a server, it will wait until it receives it, even if the producing module has not put the tuple to the server yet. This in particular will certainly cause performance issues in many real-world scientific simulations.