Scaling and Optimizing Stochastic Tuple-Space Communication in the Distributive Interoperable Executive Library

Zaire Ali (Morehouse College)
Jason Coan (Maryville College)
Mentors: Kwai Wong and David White

Abstract

The Distributive Interoperable Executive Library (DIEL) is a multi-component software framework designed to configure and execute an arbitrary series of *intercommunicating* parallel physics solvers.

- Capable of running many existing users' codes on HPC machines such as Darter and Beacon
- Utilizes an inter-process interface defined by the user in a configuration file
- Currently provides direct data exchange using user-defined boundary conditions and a prototype of indirect data exchange via a global tuple space

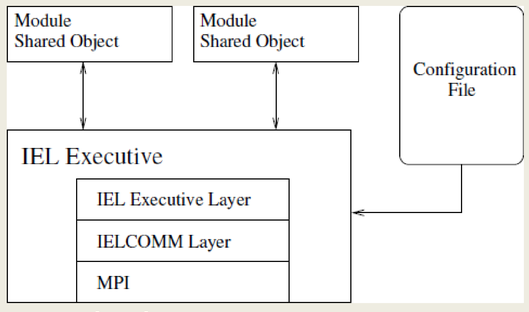
Our task is to improve and extend the tuple space implementation to make it a viable and efficient method of communication. This includes a number of challenges to overcome, both in the code that already exists and the code that we must write, which are discusses here.

The DIEL

The DIEL is composed of the **executive**, **configuration file**, and the **communication library**

Executive

- Written in C using MPI
- Provides a series of functions to add/remove modules and execute simulations using a configuration file
- Preprocesses configuration file and broadcasts communication information to each module
- Loads and executes each module
- Supports simulations with multiple parallel and/or serial physics solvers



Direct Communication

- Direct data exchange among physics modules according to shared boundary conditions specified in the configuration file
- Basic functions:
  - IEL_put(cinfo, handle, data) – non-blocking/blocking send
  - IEL_get(cinfo, handle, data) – blocking receive
- Essentially blocking—a matching send is required for every receive
- **Direct communication has the disadvantage of being synchronous, meaning both the sending and receiving processes must be ready at the same time. If the receiver is not ready, the sender must wait. In a system were performance is a key requirement, asynchronous communication should be made available for when processes are not guaranteed to be in sync with one another.**
- **Also, since even the code we are not specifically tasked to expand is in alpha stage, we sometimes encounter issues with it that need to be resolved before our new asynchronous communication code will work properly.**

Existing Prototype of Tuple Communication

- A dedicated server function with its own tuple space continuously runs in the background (started by the executive)
  - IEL_tput(data, size, tag) send data to the specified server
  - IEL_tget(data, size, tag) receive data from the specified server
- Each committed tuple is associated with a tag, and the server stores and retrieves it according to this tag.
- Memory is dynamically written to a linked list.
- Advantages:
  - **Asynchronous, stochastic communication**
  - Allows for a dedicated process to handle communication and memory management
- **But there are problems with the current implementation:**
  - One server process can only handle one request at a time.
  - The code that existed when we arrived was not thread-safe.
  - The existing executive functions and communication library have certain pitfalls that prevent the starting of multiple tuple servers.

Our Development

We have decided that our improvement of the tuple space should take place in three phases:

1. Allow for multiple concurrent tuple space servers to be started at the same time.
   - Modify the executive to start the number of servers specified in the configuration file. This is done by modifying the calls to **libconfig:**

```
int tupleSize;
if(!config_lookup_int(&cfg, "tuple_space_size", &tupleSize)){
    ERRPRINTF("\tuple_space_size\" option not set\n");
    cleanup_before_err_ret();
    config_destroy(&cfg);
    return -1;
}
exec_info->tuple_size = tupleSize;
```

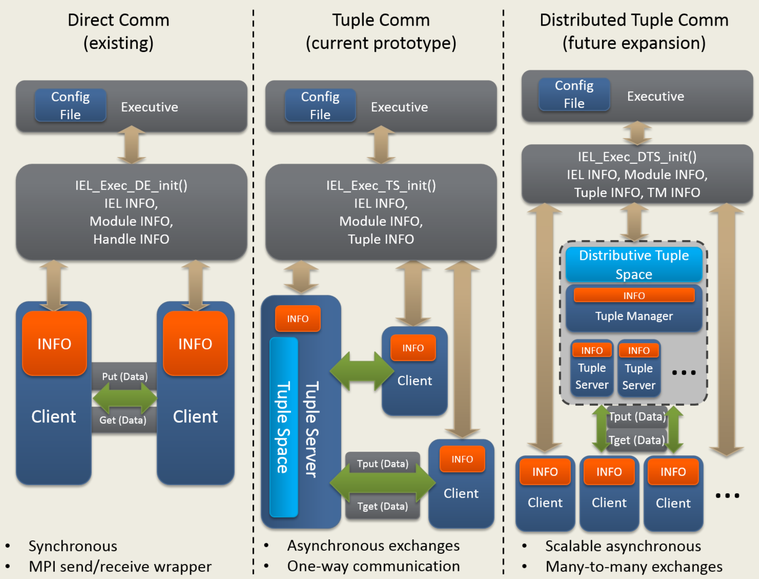
   - Modify the IEL_tput and IEL_tget functions to be able to specify a specific server to which to send/receive the data. For example, our new IEL_tput function looks like this:

```
int IEL_tput(size_t size, int tag, int serverRank, void * data)
{
    int rv;
    …
    …
    //blocking send of handshake containing size of data to be sent
    rv = MPI_Send(&size, 1, MPI_UNSIGNED, serverRank, TUPLE_PUT,
                                                MPI_COMM_WORLD);

    if(rv != MPI_SUCCESS) {
        ERRPRINTF("error in handshake, SEND1\n");
        return IEL_SEND_ERROR;
    }
    else
        DBGPRINTF("tput sent handshake with buffer-size=%d\n",
                                                (int)size);

    //blocking send of tuple with tag corresponding to committed tuple's
    //tag
    rv = MPI_Send(data, size, MPI_BYTE, serverRank, tag, MPI_COMM_WORLD);
    …
    …
    return IEL_SUCCESS;
}
```

The DIEL: Past, Present, and Future



- Synchronous
- MPI send/receive wrapper

- Asynchronous exchanges
- One-way communication

- Scalable asynchronous
- Many-to-many exchanges

Our Development (cont.)

2. Re-implement the associativity used in the direct communication functions as the rule of associativity for the tuple space.
   - The tuple servers should use the same shared boundary condition data from the configuration file to store/receive it's data, as opposed to an arbitrary tag. This means the tuple servers need to receive the same component information that the executive previously broadcasted to the modules
   - In the code to the left, we see that IEL_tput is using an arbitrary tag to identify the tuple to the server. In phase 2 we want that tag to be determined by the DIEL , not the user, according to the configuration file.
3. Create a master Tuple Space Manager that forms one abstract tuple space from all of the individual tuple space servers.
   - Should be the process to enforce the rules of associativity.
   - It should inform modules of which tuple server to send a specific tuple to based on the shared boundary condition that it represents.
   - It is desirable for one process to be in charge of this in order to avoid race conditions and incoherency.
   - The potential problem here is that the Manager may end up having too much to do, and therefore become a bottleneck for the entire system. On the other hand, if some of this responsibility is delegated back to the tuple servers themselves, it may involve too many messages being passed around. Since relatively large latencies are associated with passing messages, the number of messages passed should be minimized.
   - We will therefore need to test out different implementations and collect metrics to determine which performs best in different circumstances.

Contact Information

Zaire Ali, zaireali493@yahoo.com
Jason Coan, jason.coan@my.maryvillecollege.edu
Kwai Wong, kwong@utk.edu