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. The DIEL is capable of running many existing users’ codes on high performance computing machines such as Darter and Beacon while utilizing an inter-process interface defined by the user in a configuration file. In addition, it 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.

**Abstract**

The DIEL consists of the “Execute” and a communication library. 

- **Direct communication** – wrappers for MPI_Send() and MPI_Recv() that enforce shared boundary conditions.
- **Indirect communication** – global “tuple space” used to store data until it is needed.

**The DIEL**

- 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 where performance is a key requirement, asynchronous communication should be made available for when processes are not guaranteed to be in sync with one another.

**Our Development**

We decided that our improvement of the DIEL and tuple space should take place in several phases:

1. We expanded the DIEL to accept all C, Fortran, and JAVA based code. Developed scripts that accept serial C, Fortran, and JAVA code as input and produce DIEL-infused C code. Developed scripts that execute DIEL module code multiple times across processors simultaneously.

2. We allowed for multiple concurrent tuple space servers to be started at the same time.

- Converted the tuple space into a DIEL module, like any other.
- Modified the executable to start the number of servers specified in the configuration file. This is done by modifying the calls to ibconf: include tupleSize; if(strcmp_tuple_size("tuple_space_size", tupleSize)) { EVPF ntohs("tuple_space_size" option not set
); cleanup before exit()
; config_Destroy(iefs)
; return -6;
}

**Existing Prototype of Tuple Space Communication**

- A dedicated server function with its own tuple space continuously runs in the background (started by the executable).
- 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 execution and communication library have certain pitfalls that prevent the starting of multiple tuple servers.

**Testing**

- We wrote a randomized stress test designed to create many challenging situations for the tuple server algorithm to see how well it handles them.
- Due to the randomized nature of the test, we should run it many times and then look at the distribution of completion times.
- 16 tuple servers, 256 module processes on Darter.
- After 40 trials, the tuple servers collectively fulfill an average of 9.6 million tuple/put requests per trial.
- It takes an average of 7.5 seconds to complete one trial.
- There is little variation in the time it takes to complete the test, and copious safety checks ensure us that every request is being fulfilled correctly.

**Future Goals**

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

1. The hash function needs to be improved to provide an index on the server as well as the server’s rank.
2. 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.
3. To match functionality of most mature tuple space implementations, we should provide both blocking and non-blocking versions of IEL_tuple. 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.

**Contact Information**

Zaire Ali, zaireal493@yahoo.com
Jason Coan, jason.coan@maryvillecollege.edu
Kwai Wong, kwong@utk.edu

**Abstract**

The DIEL is a multi-component software framework designed to configure and execute an arbitrary series of intercommunicating parallel physics solvers. The DIEL is capable of running many existing users’ codes on high performance computing machines such as Darter and Beacon while utilizing an inter-process interface defined by the user in a configuration file. In addition, it 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.

**The DIEL**

- Consists of the “Execute” and a communication library.
- Direct communication – wrappers for MPI_Send() and MPI_Recv() that enforce shared boundary conditions.
- Indirect communication – global “tuple space” used to store data until it is needed.

**Our Development**

We decided that our improvement of the DIEL and tuple space should take place in several phases:

1. We expanded the DIEL to accept all C, Fortran, and JAVA based code.

   - Developed scripts that accept serial C, Fortran, and JAVA code as input and produce DIEL-infused C code.
   - Developed scripts that execute DIEL module code multiple times across processors simultaneously.

2. We allowed for multiple concurrent tuple space servers to be started at the same time.

   - Converted the tuple space into a DIEL module, like any other.
   - Modified the executable to start the number of servers specified in the configuration file. This is done by modifying the calls to ibconf:

     ```c
     include tupleSize;
     if(strcmp_tuple_size("tuple_space_size", tupleSize)) { EVPF ntohs("tuple_space_size" option not set
); cleanup before exit()
; config_Destroy(iefs)
; return -6;
}
```

**Testing**

- We wrote a randomized stress test designed to create many challenging situations for the tuple server algorithm to see how well it handles them.
- Due to the randomized nature of the test, we should run it many times and then look at the distribution of completion times.
- 16 tuple servers, 256 module processes on Darter.
- After 40 trials, the tuple servers collectively fulfill an average of 9.6 million tuple/put requests per trial.
- It takes an average of 7.5 seconds to complete one trial.
- There is little variation in the time it takes to complete the test, and copious safety checks ensure us that every request is being fulfilled correctly.

**Future Goals**

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

1. The hash function needs to be improved to provide an index on the server as well as the server’s rank.
2. 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.
3. To match functionality of most mature tuple space implementations, we should provide both blocking and non-blocking versions of IEL_tuple. 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.

**Contact Information**

Zaire Ali, zaireal493@yahoo.com
Jason Coan, jason.coan@maryvillecollege.edu
Kwai Wong, kwong@utk.edu