Dagger Language: Tutorial

1 Introduction

In order to reduce the complexity of program development, a coordination language called Dagger has been developed on top of Charm system. In Charm, an entry point is executed when there is a message directed to it. If the computation in that entry point is dependent on the computation in another entry point within the same chare, then the programmer must handle this unexpected message by buffering it, and fetching it whenever the entry point becomes eligible for execution. Dagger hides these details from the programmer by providing ready/expect and when-block constructs which will be discussed in the following section.

We will explain this by a simple example. Consider a chare which performs the following computation: it is waiting for three messages *m1, m2* and *m3* at the entry points *e1, e2*, and *e3* where they trigger the sub-computations *c1, c2*, and *c3* respectively. Assume that there exist dependences among these sub-computations. The sub-computations *c1* and *c2* can be executed in any order, but *c3* must be executed after both *c1* and *c2*. Messages may arrive in any order. If the message *m3* arrives before *m1* and *m2*, then it triggers *c3*. However, the computation *c3* can not be executed yet. Therefore, *c3* must check this situation, buffer the message *m3* if necessary, and wait for the completion of other computations. When the messages *m1* and *m2* arrive, *c1* and *c2* are executed (in any order). *c1* and *c2* must contain the code that checks if the message *m3* has already arrived, if so, one of them must trigger the computation *c3*. The Figure 1 illustrates this computation graphically where the circles are the entry points and the rectangles are the sub-computations.

Dagger language provides constructs to express such dependences and hides the details of book-keeping from the user, and still allows message driven execution.

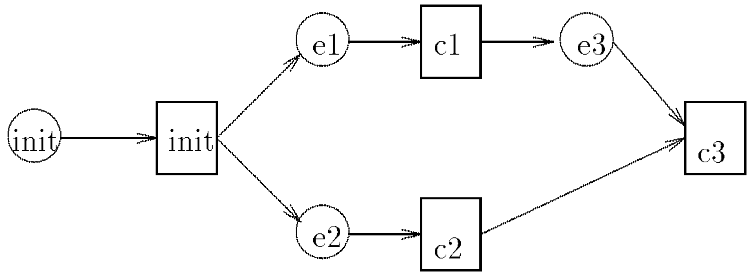

Figure 1: A simple computation

2 Basic Language

The Dagger language augments Charm with a special form of a chare called: a *dag chare*. A dag chare specifies pieces of computations (when-blocks) and dependences among computations and messages. A when-block is guarded by some dependences that must be satisfied before the when-block can be scheduled for execution. These dependences include arrival of messages or completion of other when-blocks.

```
dag chare example {
    local variable declarations
    condition variable declarations
    entry declarations
    when depn_list_1 : {when_body_1}
    ...
    when depn_list_n : {when_body_n}
    private function f1() {C_code_1}
    ....
    private function fm() {C_code_m}
}
```

Figure 2: The Dagger chare template

In the Figure 2, a template for a dag chare is shown. In addition to entries, a dag chare may declare some other data local to that dag in the local variable declaration section. The local variables are shared among when-blocks and private functions of the dag chare. Private functions are regular C functions which may contain Charm or Dagger statements/calls, and they can be called only within the static scope of the dag chare.

As in a chare definition, there are no explicit receive calls in a dag chare. The dag chare declares entry points, and messages are received at these entry points. The entry point declaration, which is in the form:

```
entry entry_name[[var_name]][MATCH] : (message msg_type *msg)
```

defines an entry with the name `entry_name`, and associates a variable with a specified message type with that entry. Messages can be sent to entry points by supplying the `entry_name` in the Charm system calls such as `SendMsg`. The variable `msg` is a pointer to the message received by the entry. The optional fields `[var_name]` and `MATCH` will be explained in the Section 3.

Receiving a message at an entry point is not sufficient to trigger a computation. (In contrast, in Charm, arrival of a message triggers a computation which is associated with that entry point.) The computation must be in a state where it is ready to process the message. A Dagger program tells the Dagger runtime system when it is ready to process a message by using the `expect` statement:

```
expect(entry_name,DAG_NOREF)
```

The second parameter `DAG_NOREF` will be clear in the next Section 3 when we discuss reference numbers. If a message arrives before an `expect` statement has been issued for it, Dagger will buffer the message. The message becomes available only after the `expect` statement is executed.

A dag chare may have a special type of variables, condition variables. A condition variable is declared as follows:

```
CONDVAR cond_var_name
```

The condition variable is used to signal completion of a when-block. In other words, it is used to express the dependences among when-blocks which belong to the same dag chare. A when-block can send a message to an entry which is defined in the same dag chare, however to utilize a shared variable (condition variable) is more efficient. A condition variable is initialized to the *not-ready* state when it is declared. It is set to the *ready* state by the `ready` statement:

```
ready(cond_var_name,DAG_NOREF)
```

Once a condition variable is set, the Dagger may schedule the when-blocks which are waiting for that condition variable to be set.

A when block is a computation which is guarded by a list of entry names and condition names:

`when` $e_1, \ldots, e_n, c_1, \ldots, c_m$ `: {when-body}`

where e_i is an entry name, and c_i is a condition variable. In order to initiate the execution of the when-block, the dependence list of the when-block must be satisfied. The dependence list is satisfied if :

- a message has been received and `expect` statement has been executed for each entry e_i in the dependence list,
- for each condition variable c_i in the list, a `ready` statement has been executed.

The `when-body` is a block of C code possibly including Charm system calls, and `expect` and `ready` Dagger statements. The messages received by the entries e_i's are accessed inside the `when-body` through the message pointers defined in the entry declarations.

2.1 A Simple Example

The *dag chare* for the computation described previously in Section 1 is illustrated in Figure 3. Every dag-chare must have at least one entry point which is named `init`. This entry point is the only one which does not need an `expect`. Therefore, the `CreateChare` call must create an instance of a dag-chare at the `init` entry point (for this example, the CreateChare call would be:

```
CreateChare(example1,example1@init,msg);
```

3 Extended Language

The Dagger Language as defined in section 2 was kept simple for ease of exposition. Supporting full generality of parallel programming acquires two extensions embodied in Dagger which are motivated and described below.

3.1 Reference Numbers

The `expect` statement imposes an order on the execution of messages. This is sufficient for a simple dag computation. However, there are computations where the concurrent phases of a dag exist (in time). An example of that is a dag augmented with a loop where different iterations of the loop may be executed concurrently. Another example is a client-server type of computation. Client processes may send multiple requests concurrently to a server dag. The server dag performs the same computation for different requests concurrently.

In order to handle these cases, messages belonging to different phases of the computation must be distinguished. To accomplish this, we modify the language to include *reference numbers*. Each message may have a reference number (positive integer values). Messages which belong to the same phase of the computation are given the same reference number by the user. Then, Dagger matches the messages with the same reference number to activate a when-block (condition variables may have reference numbers too). In other words, an instance of a when-block is scheduled for execution only if the dependence list is satisfied with the availability of messages and condition variables with matching reference numbers. The `MATCH field` in the entry point declaration specifies that the messages directed to this entry have reference numbers for matching. The `expect` and `ready` statements specify the reference numbers as follows:

```
expect(entry_name,reference_number)
ready(cond_var_name,reference_number)
```

The reference number of messages are accessed and set by the function calls `GetRefNumber(msg)`, and `SetRefNumber(msg,refnum)`. the function call `GetMyRefNumber()` in a when-block returns the reference number of the executing instance of the when-block.

3.1.1 Multiple-Message Entry Points

Another extension to the Dagger deals with entry points which receive multiple messages. This situation arises when the number of messages that a when-block depends on is known only at run time, or differs from processor to processor. The `[var_name]` field in the entry-point declaration associates the variable `var_name`, which must declared as an integer variable in the chare data local declarations, with the entry point. This variable must be initialized in the `init` entry to some positive integer value. Then the entry point expects `var_name` messages (if the `MATCH` is specified, then `var_name` matching ones).

As an example of usage of reference numbers and multiple-message entry points, we will program a simplified iterative linear systems solver. The iterative method is a jacobi-like iterative procedure which is discussed as an example with more details in the "*A Tutorial Introduction to Charm*". It involves a 2-dimensional grid. The grid is decomposed in the two dimensions, and each processor owns a block. Each processor sends its boundary values to its neighbours, receives the boundaries from its neighbours, and updates its grid points using the received boundary values. These steps are repeated a number of times.

The code for this computation is shown in Figure 4. For the sake of simplicity, only the parallel control part of the code is presented. Before explaining the code, a couple of points needs to be explained. First, the number of neighbours of a processor is dependent on the position of the processor. If it is a corner processor it has 2 neighbours, if it is on the edge than 3, otherwise 4 neighbours. Secondly, the messages belonging two consecutive iterations may come in reverse order. That is, in i^{th} iteration, a processor may receive a boundary message belonging to $i + 1^{th}$ iteration. Assume the following scenario: Processor 1 and 2 are neighbours. Processor 1 and 2 send their boundaries belonging to i^{th} iteration. Processor 2 receives all the messages it needs, performs the local computation, and sends the boundaries for the $i + 1^{th}$ iteration. During this time, the i^{th} iteration message might be in the network still. Therefore, there may exist messages belonging to different iterations directed to a processor, and they may arrive in reverse order. The processor, hence, has to handle the wrong order of messages. We will use multiple-message entry point to solve the first complication, and use iteration number as a reference number in the messages to handle the second one.

The entry declaration:

```
entry boundary[num_ofneighbours] MATCH  : (message MSG *m);
```

associates the variable `num_of_neighbours`, with the entry `boundary`. This variable is initialized to the number of neighbours i n the `init` entry. The entry point `boundary` now expects `num_of_neighbours` messages stored in an array of message pointers accessed by the paramater `m` in order to trigger a when-block `when boundary:{...}`. Note that, this when-block is executed after all the messages arrives with matching reference numbers. It is possible to access each of these messages as they arrive with a special when-block construct:

```
when boundary[ANY] : { ... }
```

The Dagger guarantees that the `when boundary : {..}` is executed after all `when boundary[ANY] : {..` have been completed.

Now, it can explained how the program works. In the `init` entry, the multiple-message entry is initialized. The first boundary messages are sent, boundaries for the first iteration are expected. As the boundaries arrive, they are copied to the computation grid. After all are received, the local computation is done, and the boundaries for next iteration are sent. Now, the processor is ready to process neighbouring boundaries belonging to the new iteration, therefore it issues an `expect`. Notice that, the new boundaries may already arrive, but they would be buffered by the Dagger until the `expect` call.

```
dag chare example1 {

    int  data;

    CONDVAR c;
    entry   init : (message INIT     *m );
    entry   e1   : (message MSGTYPE1 *m1);
    entry   e2   : (message MSGTYPE2 *m2);
    entry   e3   : (message MSGTYPE2 *m3);

    when init : {
         /* code for initialization */
         /* the message  is accessed through the message name */
         data = m->data;

         expect(e1,DAG_NOREF);
         expect(e2,DAG_NOREF);
    }

    when e1 : {
         /* computation c1 */
         c1(m1);
         expect(e3,DAG_NOREF);
    }

    when e2 : {
         /* computation c2 */
         c2(m2);
         ready(c,DAG_NOREF);
    }

    when e3 , c: {
         /* computation c3 */
         c3(m3);
    }
}
```

Figure 3: A simple dag chare example

```
dag BranchOffice Jacobi {
   GRID  A;
   int   iteration_no;
   int   num_of_neighbours;
   entry boundary[num_of_neighbours] MATCH : (message BOUNDARY *m);

   when init : {
      num_of_neighbours = myneighbours_count(MyPeNum());
      iteration_no      = 1;
      expect(boundary,iteration_no);
      send_my_boundaries(MyPeNum(),iteration_no,A);
   }

   when boundary[ANY] : {
      copy_msg_to_array(m,A);
   }

   when boundary : {
      iterate();
      iteration_no++;
      if (iteration_no < ITERATION_LIMIT) {
         send_my_boundaries(MyPeNum(),iteration_no,A);
         expect(boundary,iteration_no);
      }
   }
}

   send_my_boundaries(mypenum,iteration_no,A)
   int mypenum, iteration_no;
   GRID A;
   {
       for (each neighbour of mine) {
          msg = (BOUNDARY *) CkAllocateMsg(BOUNDARY);
          .* set the reference number */
          SetRefNumber(msg,iteration_no);
          copy_array_to_msg(A,m);
          SendMsg(boundary,msg,neighbour);
       }
   }
```

Figure 4: Jacobi-like relaxation example