Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

POSE Reference Manual

POSE software and this manual are entirely the fault of Terry L. Wilmarth. Credit for Charm++ is due to the developers, past and present, at the Parallel Programming Laboratory (`http://charm.cs.uiuc.edu`). Questions or comments on POSE or this manual can be directed to Terry at `wilmarth@cse.uiuc.edu`.

Contents

1 Introduction

POSE (Parallel Object-oriented Simulation Environment) is a tool for parallel discrete event simulation (PDES) based on Charm++. You should have a background in object-oriented programming (preferably C++) and know the basic principles behind discrete event simulation. Familiarity with simple parallel programming in Charm++ is also a plus.

POSE uses the approach of message-driven execution of Charm++, but adds the element of discrete timestamps to control when, in simulated time, a message is executed.

Users may choose synchronization strategies (conservative or several optimistic variants) on a per class basis, depending on the desired behavior of the object. However, POSE is intended to perform best with a special type of *adaptive* synchronization strategy which changes it's behavior on a per object basis. Thus, other synchronization strategies may not be properly maintained. There are two significant versions of the adaptive strategy, *adapt4*, a simple, stable version, and *adept*, the development version.

1.1 Developing a model in POSE

Modeling a system in POSE is similar to how you would model in C++ or any OOP language. Objects are entities that hold data, and have a fixed set of operations that can be performed on them (methods).

Charm++ provides the parallelism we desire, but the model does not differ dramatically from C++. The primary difference is that objects may exist on a set of processors, and so invoking methods on them requires communication via messages. These parallel objects are called *chares*.

POSE adds to Charm++ by putting timestamps on method invocations (events), and executing events in timestamp order to preserve the validity of the global state.

Developing a model in POSE involves identifying the entities we wish to model, determining their interactions with other entities and determining how they change over time.

1.2 PDES in POSE

A simulation in POSE consists of a set of Charm++ chares performing timestamped events in parallel. In POSE, these chares are called *posers*. POSE is designed to work with many such entities per processor. The more a system can be broken down into its parallel components when designing the simulation model, the more potential parallelism in the final application.

A poser class is defined with a synchronization strategy associated with it. We encourage the use of the adaptive strategies, as mentioned earlier. Adaptive strategies are optimistic, and will potentially execute events out of order, but have rollback and cancellation messages as well as checkpointing abilities to deal with this behind the scenes.

Execution is driven by events. An event arrives for a poser and is sorted into a queue by timestamp. The poser has a local time called object virtual time (OVT) which represents its progress through the simulation. When an event arrives with a timestamp $t >$OVT, the OVT is advanced to t. If the event has timestamp $t <$OVT, it may be that other events with greater timestamps were executed. If this is the case, a rollback will occur. If not, the event is simply executed along with the others in the queue.

Time can also pass on a poser within the course of executing an event. An *elapse* function is used to advance the OVT.

POSE maintains a global clock, the global virtual time (GVT), that represents the progress of the entire simulation.

Currently, POSE has no way to directly specify event dependencies, so if they exist, the programmer must handle errors in ordering carefully. POSE provides a delayed error message print and abort function that is only performed if there is no chance of rolling back the dependency error. Another mechanism provided by POSE is a method of tagging events with *sequence numbers*. This allows the user to determine the execution order of events which have the same timestamp.

2 Compiling, Running and Debugging a Sample POSE program

Sample code is available in the Charm++ source distribution. Assuming a netlrts-linux build of Charm++, look in `charm/netlrts-linux/examples/pose`. The SimBenchmark directory contains a synthetic benchmark simulation and is fairly straightforward to understand.

2.1 Compiling

To build a POSE simulation, run `etrans.pl` on each POSE module to get the new source files. etrans.pl is a source to source translator. Given a module name it will translate the `module.h`, `module.ci`, and `module.C` files into `module_sim.h`, `module_sim.ci`, and `module_sim.C` files. The translation operation adds wrapper classes for POSE objects and handles the interface with strategies and other poser options.

To facilitate code organization, the `module.C` file can be broken up into multiple files and those files can appended to the etrans.pl command line after the module name. These additional .C files will be translated and their output appended to the `module_sim.C` file.

The `-s` switch can be passed to use the sequential simulator feature of POSE on your simulation, but you must also build a sequential version when you compile (see below).

Once the code has been translated, it is a Charm++ program that can be compiled with `charmc`. Please refer to the CHARM++/CONVERSE Installation and Usage Manual for details on the charmc command. You should build the new source files produced by etrans.pl along with the main program and any other source needed with `charmc`, linking with `-module pose` (or `-module seqpose` for a sequential version) and `-language charm++`. The SimBenchmark example has a Makefile that shows this process.

2.2 Running

To run the program in parallel, a `charmrun` executable was created by `charmc`. The flag `+p` is used to specify a number of processors to run the program on. For example:

```
> ./charmrun pgm +p4
```

This runs the executable `pgm` on 4 processors. For more information on how to use `charmrun` and set up your environment for parallel runs, see the CHARM++/CONVERSE Installation and Usage Manual.

2.3 Debugging

Because POSE is translated to Charm++, debugging is a little more challenging than normal. Multi-processor debugging can be achieved with the `charmrun ++debug` option, and debugging is performed on the `module_sim.C` source files. The user thus has to track down problems in the original POSE source code. A long-term goal of the POSE developers is to eliminate the translation phase and rely on the interface translator of Charm++ to provide similar functionality.

2.4 Sequential Mode

As mentioned above, the same source code can be used to generate a purely sequential POSE executable by using the `-s` flag to `etrans.pl` and linking with `-module seqpose`. This turns off all aspects of synchronization, checkpointing and GVT calculation that are needed for optimistic parallel execution. Thus you should experience better one-processor times for executables built for sequential execution than those built for parallel execution. This is convenient for examining how a program scales in comparison to sequential time. It is also helpful for simulations that are small and fast, or in situations where multiple processors are not available.

3 Programming in POSE

This section details syntax and usage of POSE constructs with code samples.

3.1 POSE Modules

A POSE module is similar to a Charm++ module. It is comprised of an interface file with suffix `.ci`, a header `.h` file, and the implementation in `.C` files. Several posers can be described in one module, and the module can include regular chares as well. The module is translated into Charm++ before the simulation can be compiled. This translation is performed by a Perl script called `etrans.pl` which is included with POSE. It generates files suffixed `_sim.ci`, `_sim.h`, and `_sim.C`.

3.2 Event Message and Poser Interface Description

Messages, be they event messages or otherwise, are described in the `.ci` file exactly the way they are in Charm++. Event messages cannot make use of Charm++'s parameter marshalling, and thus you must declare them in the `.h` file. Charm++ `varsize` event messages are currently not implemented in POSE.

All event messages inherit from a POSE type `eventMsg` which includes data for timestamps and miscellaneous POSE statistics.

message *myMessage*;

Posers are described similar to chares, with a few exceptions. First, the `poser` keyword is used to denote that the class is a POSE simulation object class. Second, event methods are tagged with the keyword `event` in square brackets. Finally, three components are specified which indicate how objects of the poser class are to be simulated. The *sim* component controls the wrapper class and event queue used by the object. The *strat* component controls the synchronization strategy the object should use (*i.e.* adaptive or basic optimistic). The *rep* component specifies the global state representation, which controls how the global state is kept accurate depending on the synchronization strategy being used (*i.e.* checkpointing or no checkpointing). Currently, there is only one wrapper type, `sim`. This 3-tuple syntax is likely to become obsolete, replaced simply by synchronization strategy only. Keeping the global state accurate is largely a function of the synchronization strategy used.

```
poser mySim :  sim strat rep {
   entry mySim(myMessage *);
   entry [event] void myEventMethod(eventMsg *);
   ...
};
```

A typical `.ci` file poser specification might look like this:

```
poser Worker :  sim adapt4 chpt {
   entry Worker(WorkerCreationMsg *);
   entry [event] void doWork(WorkMsg *);
   ...
};
```

Note that the constructors and event methods of a poser must take an event message as parameter. If there is no data (and thereby no message defined) that needs to be passed to the method, then the parameter should be of type `eventMsg *`. This ensures that POSE will be able to timestamp the event.

3.3 Declaring Event Messages and Posers

Currently, event messages are declared with no reference to what they might inherit from (unlike in Charm++). The translator takes care of this. In addition, they must define `operator=`.

```
class myMessage {
   public:
```

```
   int someData;
   myMessage& operator=(const myMessage& obj) {
      eventMsg::operator=(obj);
      someData = obj.someData;
      return *this;
   }
};
```

Similarly, posers do not refer to a base class when they are declared. Posers are required to have a void constructor declared that simply initializes the data to sensible values. A destructor must be provided as well. In addition, a `pup` and `operator=` must be provided. The `pup` method should call the `pup` method of the global state representation class being used.

```
class mySim {
   int anInt; float aFloat; char aString[20];
public:
   mySim();
   mySim(myMessage *m);
   ~mySim();
   void pup(PUP::er &p);
   mySim& operator=(const mySim& obj);
   void myEventMethod(eventMsg *m);
   void myEventMethod_anti(eventMsg *m);
   void myEventMethod_commit(eventMsg *m);
   ...
};
```

Further, for each event method, a commit method should be declared, and if the synchronization strategy being used is optimistic or involves any sort of rollback, an anti-method should also be provided. The syntax of these declarations is shown above. Their usage and implementation will be described next.

3.4 Implementing Posers

The void constructor for a poser should be defined however the user sees fit. It could be given an empty body and should still work for POSE. Poser entry constructors (those described in the `.ci` file) should follow the template below:

```
mySim::mySim(myMessage *m)
{
   // initializations from m
   ...
   delete m;
   ...
};
```

Note that while the incoming message m may be deleted here in the constructor, event messages received on event methods should **not** be deleted. The PDES fossil collection will take care of those.

An event method should have the following form:

```
void mySim::myEventMethod(eventMsg *m) {
   // body of method
};
```

Again, m is never deleted in the body of the event. A side effect of optimistic synchronization and rollback is that we would like the effects of event execution to be dependent only upon the state encapsulated in the corresponding poser. Thus, accessing arbitrary states outside of the simulation, such as by calling `rand`, is forbidden. We are planning to fix this problem by adding a `POSE_rand()` operation which will generate a random number the first time the event is executed, and will checkpoint the number for use in subsequent re-executions should a rollback occur.

3.5 Creation of Poser Objects

Posers are created within a module using the following syntax:

```
int hdl = 13; // handle should be unique
myMessage *m = new myMessage;
m->someData = 34;
POSE_create(mySim(m), hdl, 0);
```

This creates a `mySim` object that comes into existence at simulation time zero, and can be referred to by the handle 13.

Creating a poser from outside the module (*i.e.* from `main`) is somewhat more complex:

```
int hdl = 13;
myMessage *m = new myMessage;
m->someData = 34;
m->Timestamp(0);
(*(CProxy_mySim *) & POSE_Objects)[hdl].insert(m);
```

This is similar to what the module code ultimately gets translated to and should be replaced by a macro with similar syntax soon.

3.6 Event Method Invocations

Event method invocations vary significantly from entry method invocations in Charm++, and various forms should be used depending on where the event method is being invoked. In addition, event messages sent to an event method should be allocated specifically for an event invocation, and cannot be recycled or deleted.

There are three ways to send events within a POSE module. The first and most commonly used way involves specifying and offset in simulation time from the current time. The syntax follows:

```
aMsg = new eventMsg;
POSE_invoke(myEventMethod(aMsg), mySim, hdl, 0);
```

Here, we've created an `eventMsg` and sent it to `myEventMethod`, an event entry point on `mySim`. `mySim` was created at handle `hdl`, and we want the event to take place now, i.e. at the current simulation time, so the offset is zero.

The second way to send an event is reserved for use by non-poser objects within the module. It should not be used by posers. This version allows you to specify an absolute simulation time at which the event happens (as opposed to an offset to the current time). Since non-poser objects are not a part of the simulation, they do not have a current time, or OVT, by which to specify an offset. The syntax is nearly identical to that above, only the last parameter is an absolute time.

```
aMsg = new eventMsg;
POSE_invoke_at(myEventMethod(aMsg), mySim, hdl, 56);
```

Posers should not use this approach because of the risk of specifying an absolute time that is earlier than the current time on the object sending the event.

Using this method, event methods can be injected into the system from outside any module, but this is not recommended.

The third approach is useful when an object send events to itself. It is simply a slightly shorter syntax for the same thing as `POSE_invoke`:

```
aMsg = new eventMsg;
POSE_local_invoke(myEventMethod(aMsg), offset);
```

3.7 Elapsing Simulation Time

We've seen in the previous section how it is possible to advance simulation time by generating events with non-zero offsets of current time. When such events are received on an object, if the object is behind, it advances its local simulation time (object virtual time or OVT) to the timestamp of the event.

It is also possible to elapse time on an object while the object is executing an event. This is accomplished thus:

```
elapse(42);
```

The example above would simulate the passage of forty-two time units by adding as much to the object's current OVT.

3.8 Interacting with a POSE Module and the POSE System

POSE modules consist of <*modname*>`.ci`, <*modname*>`.h` and <*modname*>`.C` files that are translated via `etrans.pl` into <*modname*>`_sim.ci`, <*modname*>`_sim.h` and <*modname*>`_sim.C` files. To interface these with a main program module, say *Pgm* in files `pgm.ci`, `pgm.h` and `pgm.C`, the `pgm.ci` file must declare the POSE module as extern in the `mainmodule Pgm` block. For example:

```
mainmodule Pgm {
   extern module <modname>;
   readonly CkChareID mainhandle;

   mainchare main {
      entry main();
   };
};
```

The `pgm.C` file should include `pose.h` and <*modname*>`_sim.h` along with its own headers, declarations and whatever else it needs.

Somewhere in the `main` function, `POSE_init()` should be called. This initializes all of POSE's internal data structures. The parameters to `POSE_init()` specify a termination method. POSE programs can be terminated in two ways: with inactivity detection or with an end time. Inactivity detection terminates after a few iterations of the GVT if no events are being executed and virtual time is not advancing. When an end time is specified, and the GVT passes it, the simulation exits. If no parameters are provided to `POSE_init()`, then the simulation will use inactivity detection. If a time is provided as the parameter, this time will be used as the end time.

Now POSE is ready for posers. All posers can be created at this point, each with a unique handle. The programmer is responsible for choosing and keeping track of the handles created for posers. Once all posers are created, the simulation can be started:

```
POSE_start();
```

4 Configuring POSE

POSE can be configured in two different ways. Fundamental behaviors are controlled by altering values in the `pose_config.h` file in the POSE installation, and rebuilding POSE. Many of these configuration options can (and should) be controlled by command line options. These will be designated here by an asterisk (∗). See section 4.1 for the command line options.

- **POSE_STATS_ON** ∗
  ◦ Turn on timing and statistics gathering for internal POSE operations. Produces a small slowdown in program.

- **POSE_DOP_ON** ∗
  ◦ Turn on timing and statistics gathering for degree of parallelism calculations. Generates log files that can be loaded by ploticus scripts to produce graphs plotting active entities over time. Slows down program dramatically.

- **POSE_COMM_ON**
  ◦ Turn on streaming communication optimization for small message packing.

- **COMM_TIMEOUT**
  ◦ Used by streaming communication library. Time to wait (in ?) before sending buffered messages.

- **COMM_MAXMSG**
  ◦ Used by streaming communication library. Number of messages to buffer before packing and sending as one.

- **LB_ON** ∗
  ◦ Turn on POSE load balancing.

- **STORE_RATE** ∗
  ◦ Default checkpointing rate: 1 for every `STORE_RATE` events.

- **SPEC_WINDOW** ∗
  ◦ Speculative window size: this is how far (in virtual time units) ahead of GVT posers are allowed to go.

- **MIN_LEASH** ∗ and **MAX_LEASH** ∗
  ◦ Bounds on the speculative window, these are adjusted by adaptive synchronization strategies.

- **LEASH_FLEX** ∗
  ◦ Granularity of flexibility when speculative window is shrunk or expanded.

- **MAX_POOL_SIZE**
  ◦ Memory used by event messages is recycled. This controls how many messages of a particular size will be kept on hand.

- **MAX_RECYCLABLE**
  ◦ This is the largest size of message that will be recycled.

- **LB_SKIP** ∗
  ◦ This controls the frequency of load balance invocation. 1 in every `LB_SKIP` executions of the GVT algorithm will invoke load balancing.

- **LB_THRESHOLD** ∗
  ◦ What the heck does this number mean? I can't remember. I'll have to look through the code... later. Meanwhile, I think this indicates some sort of threshold a single processor has to cross before we even bother with analyzing the load.

- **LB_DIFF** ∗
  ◦ Once the load has been analyzed, we compute the difference between the max and min PE loads. Only if this difference exceeds `LB_DIFF` do we bother migrating posers.

Several of the above flags and constants will be eliminated as the adaptive strategy is expanded. What remains will eventually become run-time options.

4.1 POSE Command Line Options

Command line options are handled like Charm++ command line parameters. For namespace purity all POSE command line options have a _pose suffix. They can be inspected by appending a -h to an execution of a POSE program. Command line options override any defaults set in the `pose_config.h` file

- **+stats_pose**
  ◦ Turn on timing and statistics gathering for internal POSE operations. Produces a small slowdown in program.

- **+dop_pose**
  ◦ Turn on timing and statistics gathering for degree of parallelism calculations. Generates log files that can be loaded by ploticus scripts to produce graphs plotting active entities over time. Slows down program dramatically.

- **+lb_on_pose**
  ◦ Turn on POSE load balancing.

- **+store_rate_pose N**
  ◦ Default checkpointing rate: 1 for every `STORE_RATE` events.

- **+spec_window_pose N**
  ◦ Speculative window size: this is how far (in virtual time units) ahead of GVT posers are allowed to go.

- **+min_leash_pose N** and **+min_leash_pose N**
  ◦ Bounds on the speculative window, these are adjusted by adaptive synchronization strategies.

- **+leash_flex_pose N**
  ◦ Granularity of flexibility when speculative window is shrunk or expanded.

- **+lb_skip_pose N**
  ◦ This controls the frequency of load balance invocation. 1 in every `LB_SKIP` executions of the GVT algorithm will invoke load balancing.

- **+lb_threshold_pose N**
  ◦ Minimum threshold for load balancing, default is 4000

- **+lb_diff_pose N**
  ◦ Once the load has been analyzed, we compute the difference between the max and min PE loads. Only if this difference exceeds `LB_DIFF` do we bother migrating posers.

- **+checkpoint_gvt_pose N**
  ◦ Checkpoint to disk approximately every N GVT ticks (N is an integer). The default is 0, which indicates no checkpointing.

- **+checkpoint_time_pose N**
  ◦ Checkpoint to disk every N seconds (N is an integer). The default is 0, which indicates no checkpointing. If both this parameter and +checkpoint_gvt_pose are greater than 0, a warning will be given, the value of this parameter will be set to 0, and POSE will checkpoint based on GVT ticks.

As a technical point, pose command line parsing is done inside the `POSE_init()` call. Therefore, the most consistent behavior for interleaving pose command line options with user application options will be achieved by calling `POSE_init()` before handling user application command line arguments.

5 Communication Optimizations

6 Load Balancing

7 Glossary of POSE-specific Terms

- `void` **POSE_init**`()`
  ◦ Initializes various items in POSE; creates the load balancer if load balancing is turned on; initializes the statistics gathering facility if statistics are turned on.
  ◦ Must be called in user's main program prior to creation of any simulation objects or reference to any other POSE construct.

- `void` **POSE_start**`()`
  ◦ Sets busy wait to default if none specified; starts quiescence detection; starts simulation timer.
  ◦ Must be called in user's main program when simulation should start.

- `void` **POSE_registerCallBack**`(CkCallback cb)`
  ◦ Registers callback function with POSE – when program ends or quiesces, function is called.
  ◦ CkCallback is created with the index of the callback function and a proxy to the object that function is to be called on. For example, to register the function `wrapUp` in the main module as a callback:

```
CProxy_main M(mainhandle);
POSE_registerCallBack(CkCallback(CkIndex_main::wrapUp(), M));
```

- `void` **POSE_stop**`()`
  ◦ Commits remaining events; prints final time and statistics (if on); calls callback function.
  ◦ Called internally when quiescence is detected or program reaches `POSE_endtime`.

- `void` **POSE_exit**`()`
  ◦ Similar to `CkExit()`.

- `void` **POSE_set_busy_wait**`(int n)`
  ◦ Used to control granularity of events; when calling `POSE_busy_wait`, program busywaits for time to compute $fib(n)$.

- `void` **POSE_busy_wait**`()`
  ◦ Busywait for time to compute $fib(n)$ where n is either 1 or set by `POSE_set_busy_wait`.

- **POSE_useET(t)**
  ◦ Set program to terminate when global virtual time (GVT) reaches t.

- **POSE_useID()**
  ◦ Set program to terminate when no events are available in the simulation.

- `void` **POSE_create**(*constructorName*`(eventMsg *m), int handle, int atTime)`
  ◦ Creates a poser object given its constructor, an event message m of the appropriate type, any integer as the handle (by which the object will be referred from then on), and a time (in simulation timesteps) at which it should be created.
  ◦ The handle can be thought of as a chare array element index in Charm++.

- `void` **POSE_invoke_at**(*methodName*`(eventMsg *m),` *className*`, int handle, int atTime)`
  ◦ Send a *methodName* event with message m to an object of type *className* designated by handle *handle* at time specified by $atTime$.
  ◦ This can be used by non-poser objects in the POSE module to inject events into the system being simulated. It should not be used by a poser object to generate an event.

- `void` **POSE_invoke**(*methodName*`(eventMsg *m),` *className*`, int handle, int timeOffset)`
  ◦ Send a *methodName* event with message m to an object of type *className* designated by handle *handle* at current OVT + $timeOffset$.
  ◦ This is used by poser objects to send events from one poser to another.

- `void` **POSE_local_invoke**(*methodName*`(eventMsg *m), int timeOffset)`
  ◦ Send a *methodName* event with message m to this object at current OVT + $timeOffset$.
  ◦ This is used by poser objects to send events to themselves.

- `void` **CommitPrintf**`(char *s, args...)`
  ◦ Buffered print statement; prints when event is committed (i.e. will not be rolled back).
  ◦ Currently, must be called on the wrapper class (parent) to work properly, but a fix for this is in the works.

- `void` **CommitError**`(char *s, args...)`
  ◦ Buffered error statement; prints and aborts program when event is committed.
  ◦ Currently, must be called on the wrapper class (parent) to work properly, but a fix for this is in the works.

- `void` **elapse**`(int n)`
  ◦ Elapse n simulation time units.

- **poser**
  ◦ Keyword (used in place of chare) to denote a poser object in the `.ci` file of a POSE module.

- **event**
  ◦ Keyword used in square brackets in the `.ci` file of a POSE module to denote that the entry method is an event method.

- **eventMsg**
  ◦ Base class for all event messages; provides timestamp, priority and many other properties.

- **sim**
  ◦ Base class of all wrapper classes.

- **strat**
  ◦ Base class of all strategy classes.

- **con**
  - Simple conservative strategy class.
- **opt, opt2, opt3, spec, adapt, adapt2**
  - Optimistic strategy classes.
- **rep**
  - Base class for all representation classes.
- **chpt**
  - Simple checkpointing representation class.
- **OVT()**
  - Returns the object virtual time (OVT) of the poser in which it is called
- `void` *MySim*`::`**terminus**`()`
  - When simulation has terminated and program is about to exit, this method is called on all posers. Implemented as an empty method in the base `rep` class, the programmer may choose to override this with whatever actions may need to be performed per object at the end of the simulation.