Implementation of Randomized Dining Philosophers

Presented by Xin Zhang

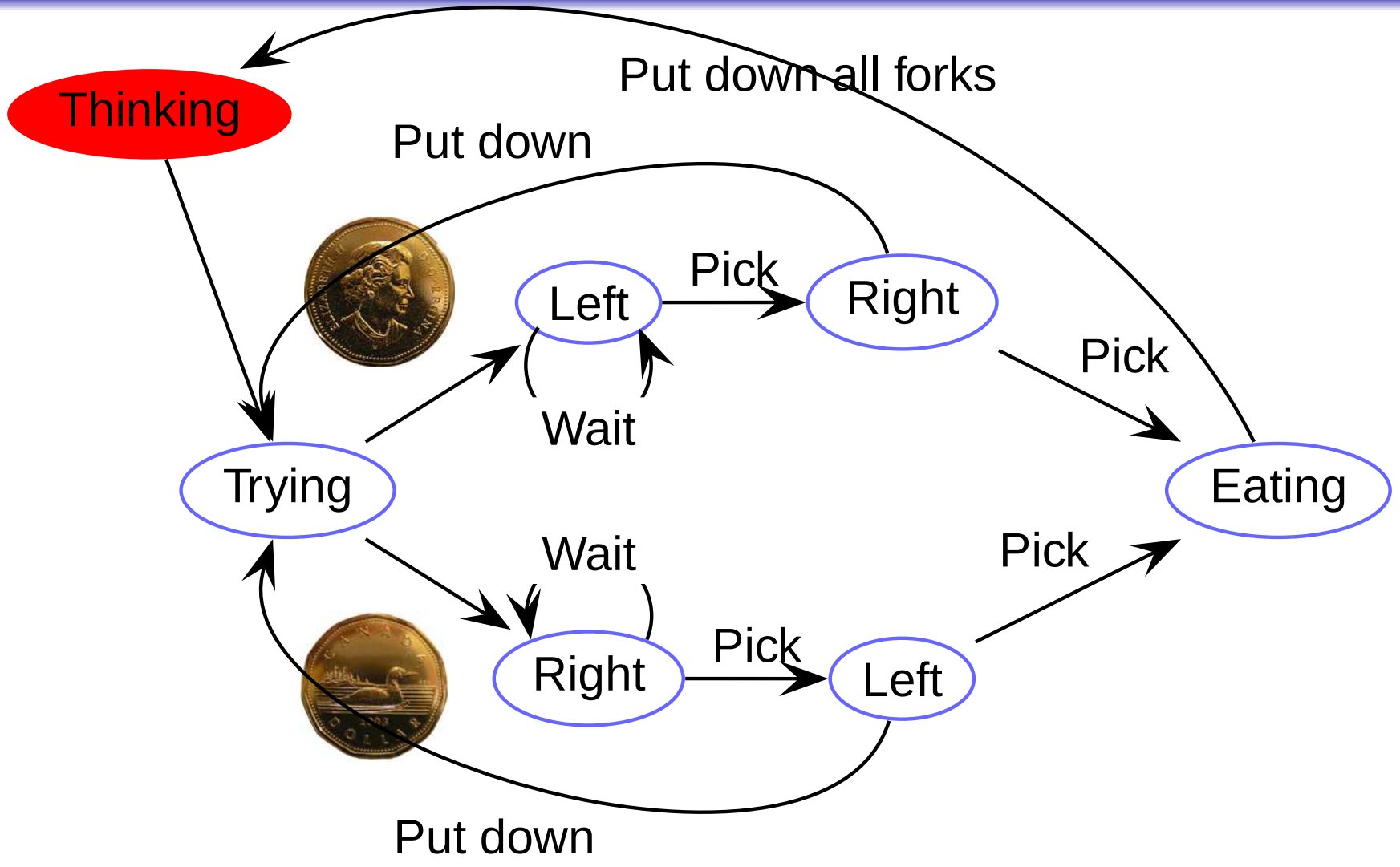
YORK
UNIVERSITÉ
UNIVERSITY

Outline

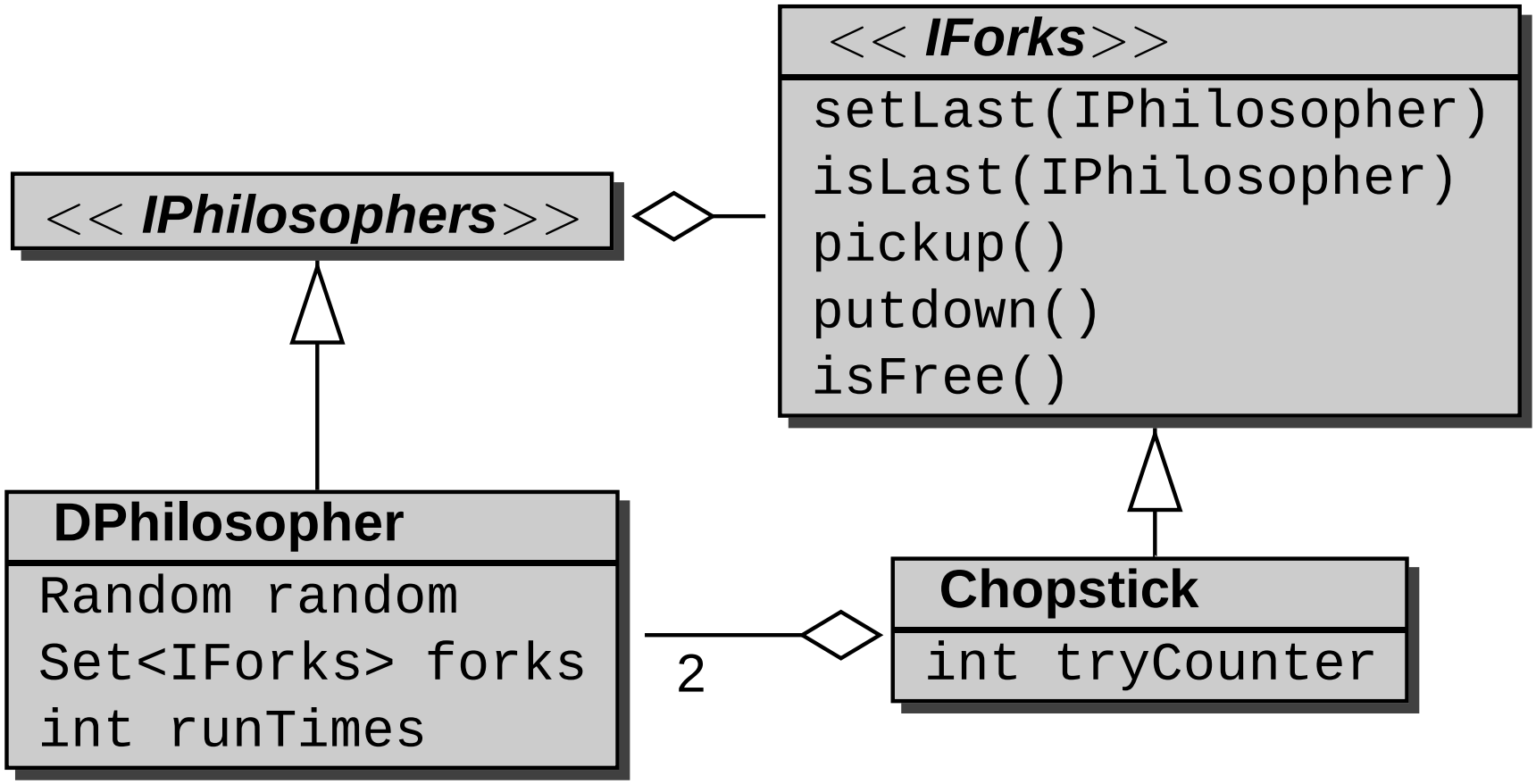
Recall

- We use randomized algorithm to break the initial symmetry
- Use different mechanism to guarantee mutual exclusion and deadlock free
- With extra information, we can achieve liveness property-starvation free

Control Flow Graph



Structure


<< IForks >>
setLast(IPhilosopher)
isLast(IPhilosopher)
pickup()
putdown()
isFree()
<< IPhilosophers >>
DPhilosopher
Random random
Set<IForks> forks
int runTimes
2
Chopstick
int tryCounter

Chopstick Object

```
 1  public Semaphore getSemaphore(){
 2      return semaphore;
 3  }
 4
 5  public void pickup(){
 6      state = INUSE;  continousTries = 0;
 7  }
 8  public boolean isLast(IPhilosopher last){
 9      if (last != this.last) return false;
10      else if (continousTries >= MAX_CONTINOUS_TRY)
            return false;
11      else {
12          continousTries ++;
13          return true;
14      }
15  }
```

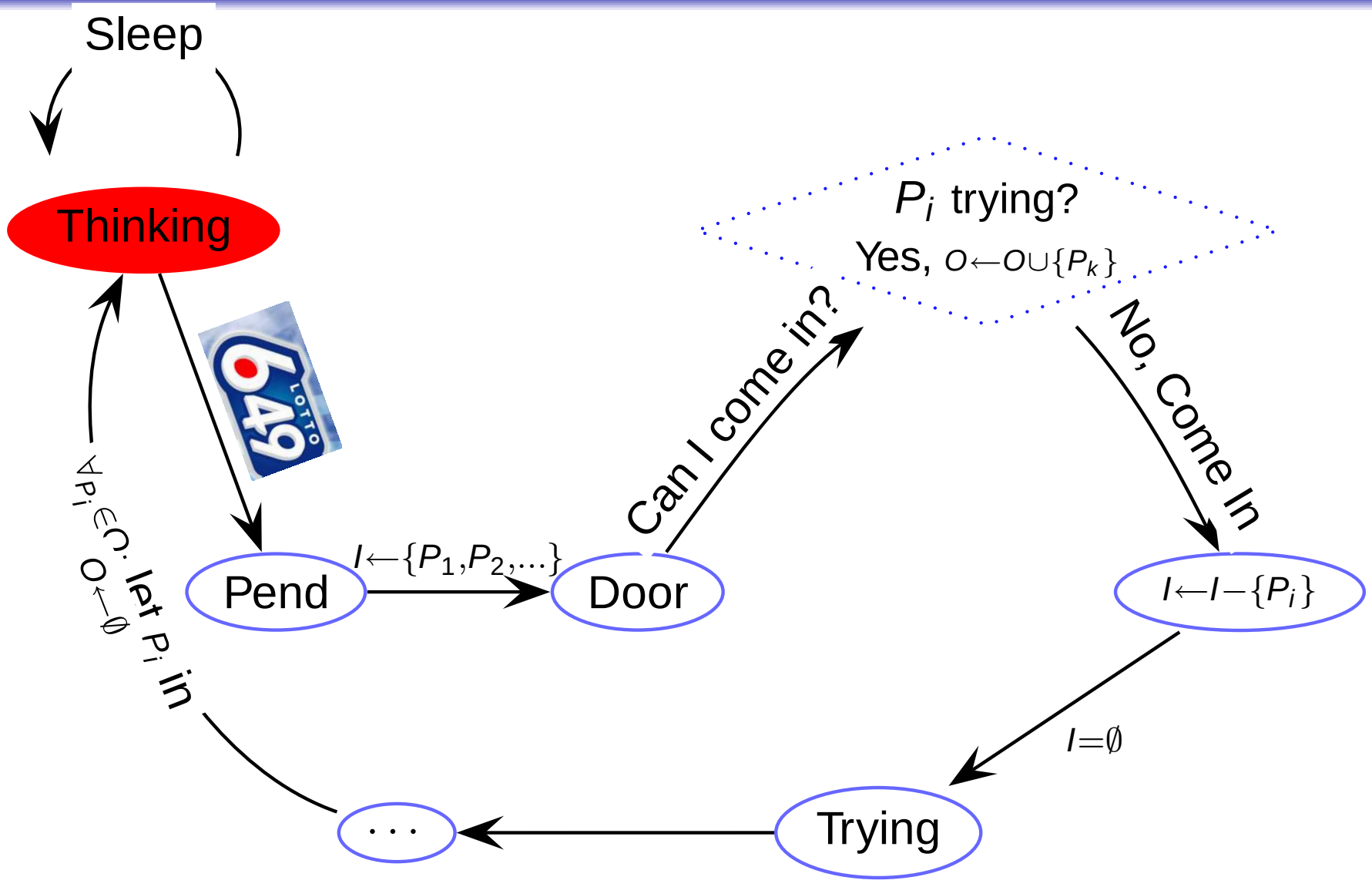
DPhilosopher Object

```
1  while(true){
2      if(random.nextInt(1) == 0){
3          try1 = left; try2 = right; }
4      else {
5          try1 = right; try2 = left; }
6
7      while{retry}{
8          try1.getSemaphore().acquire();
9          if (try1.isFree() && ! try1.isLast(this)){
10             try1.pickup(); retry = false;
11             try1.getSemaphore().release(); }
12         else{
13             try1.getSemaphore().release();}    }
14     try2.getSemaphore().acquire();
15     if (try2.isFree()){
16         try2.pickup(); try2.getSemaphore().release
               (); }
```

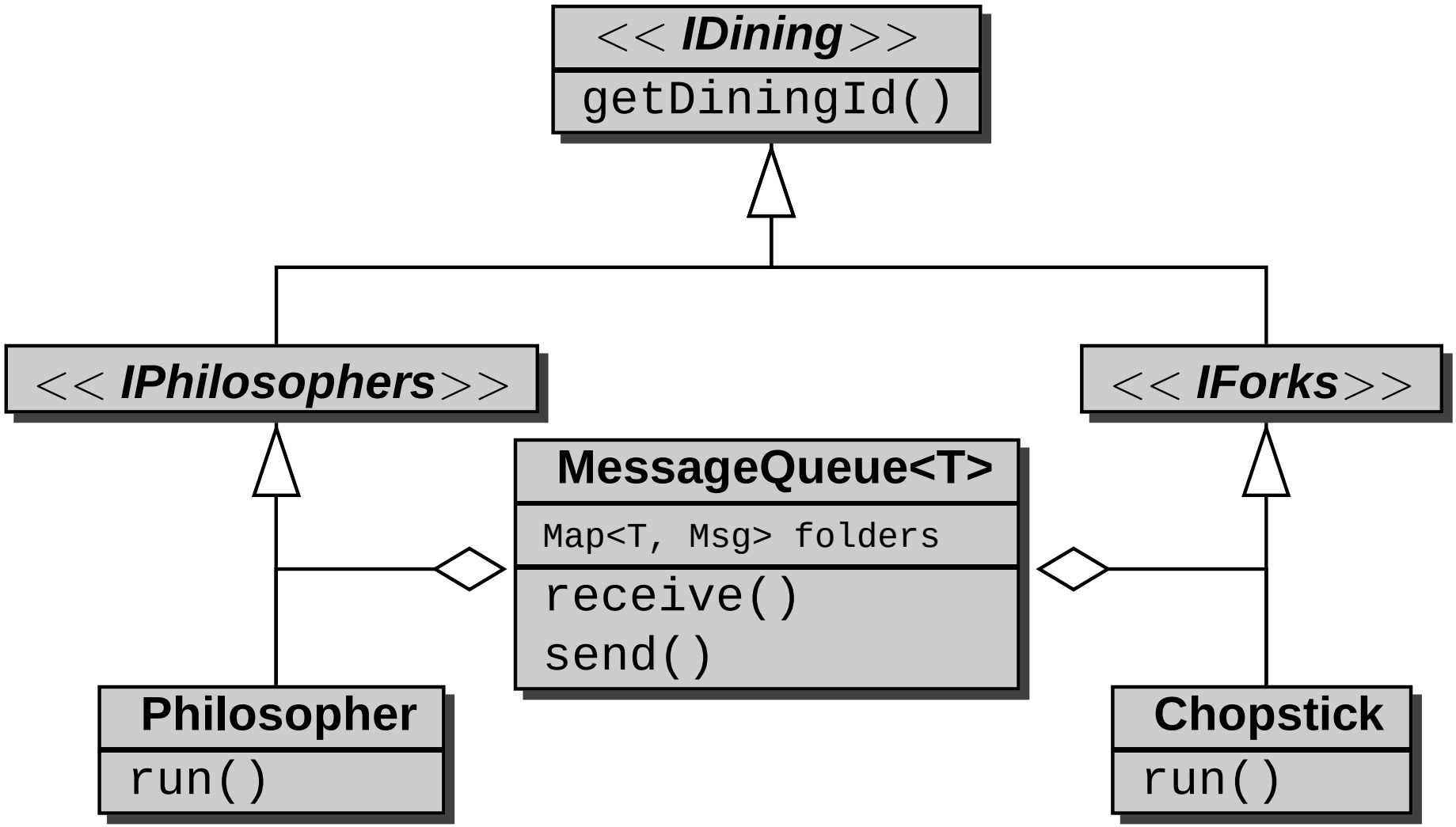
DPhilosopher Object-cont

```
16
17          else{  try2.getSemaphore().release();
18              try1.getSemaphore().acquire();  try1.putdown
                    ();
19              try1.getSemaphore().release();  continue;
20          }
21          //eating
22          try1.getSemaphore().acquire();
23          try1.putdown();try1.setLast(this);
24          try1.getSemaphore().release();
25          try2.getSemaphore().acquire();
26          try2.putdown();  try2.setLast(this);
27          try2.getSemaphore().release();
28      }
```

Control Flow



Structure


<< IDining >>
getDiningId()
<< IPhilosophers >>
<< IForks >>
MessageQueue<T>
Map<T, Msg> folders
receive()
send()
Philosopher
run()
Chopstick
run()

MessageQueue Object

```
1  public void send(IMessage<T> message) {
2      List<IMessage<T>> list = null;
3      T from = message.getRecipient();
4      synchronized(mFolders){
5          if ( !mFolders.containsKey(from) ){
6              list = Collections.synchronizedList(new
                   LinkedList<IMessage<T>>());
7              mFolders.put(from, list); }
8          else {
9              list = mFolders.get(from); }  }
10     list.add(message);
11 }
12 public IMessage<T> receive(T recipient) {
       synchronized(mFolders){...}}
13 public IMessage<T> receiveBlocked(T recipient)
       {...}
```

Chopstick Object

```
1  state = FREE;
2  while(true){
3       IMessage<IDining> msg = mQueue.receiveBlocked(
            this);
4       if (type == TYPE_REGISTER){...}
5       else if (type == TYPE_REQUEST){...}
6       else if(type == TYPE_FAIL || type ==
            TYPE_RELEASE){...}
7       else if(type == TYPE_REMOVE){...}
8  }
```

Philosopher Object

```
1  if (state == STATE_THINKING){
2    state = STATE_PENDING; in = new HashSet<IDining
       >();
3  for (IDining p : contenders){
4    if (p != this) { //send p request to enter door }
5  else if(state == STATE_TRYING && isTimeOut()){
6    if(random.nextInt(contenders.size()) != 0){ //
       lottery
7      waitTime = (waitTime * 2) % MAX_WAIT_TIME }
8    else {
9      for(IForks fork :  forks){//send pickup to
         forks}
10     state = STATE_HOPEFUL; grantedForkSet.clear();
         } }
11 else if (state == STATE_PENDING && in.isEmpty()){
12   state = STATE_TRYING;}
13 }
```

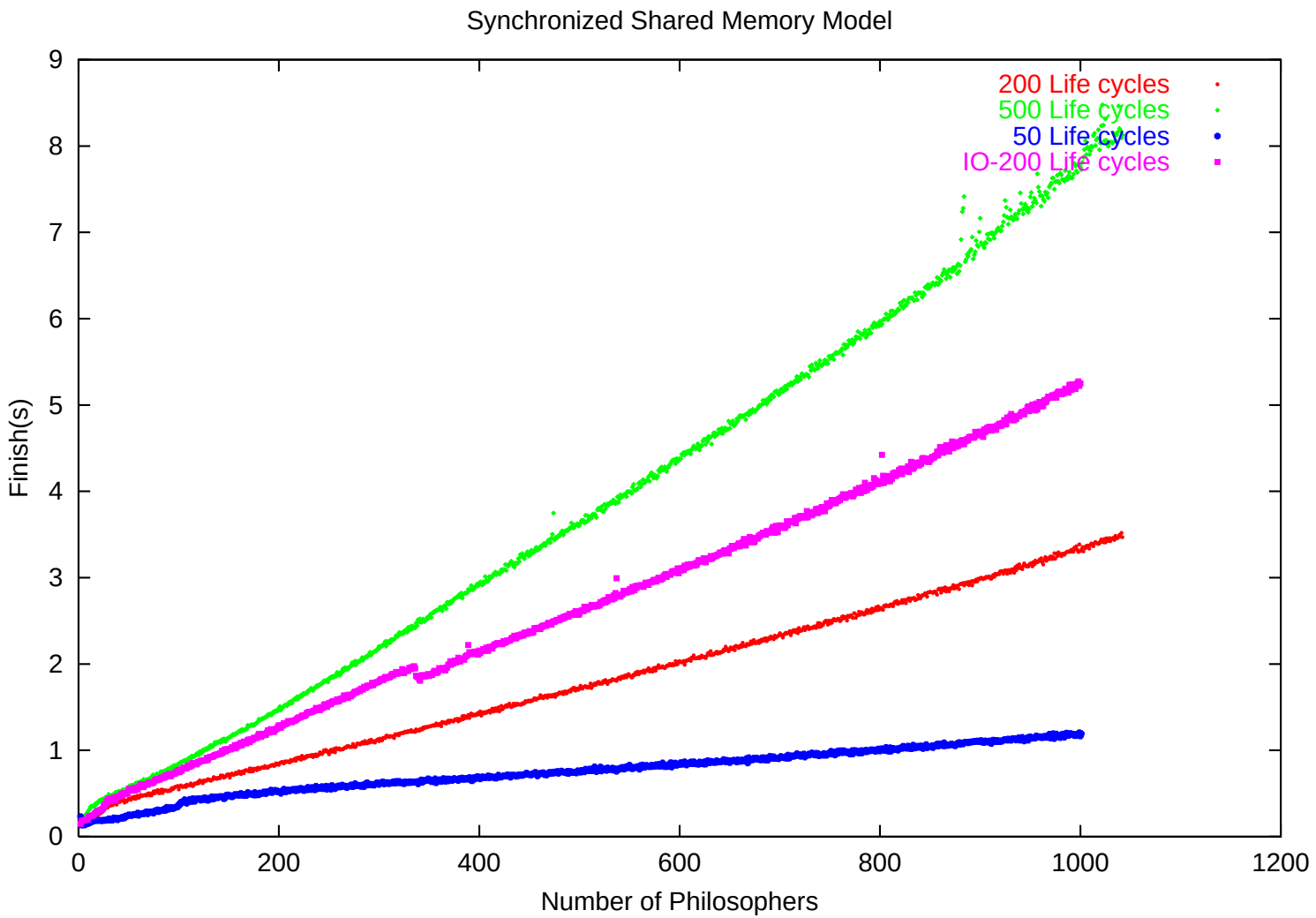
Philosopher Object

```
13  IMessage<IDining> msgIn = mQueue.receive(this);
14  IDining process = msgIn.getSender(); int type =
        msgIn.getType();
15  if (type == TYPE_UPDATE){...    }
16  else if (type == TYPE_ENTER){    //request from
        other philosophers to enter
17    if (state == STATE_TRYING ){ out.add(process); }
18    else{//send back to allow in} }
19  else if(type == TYPE_OUTSIDE && state ==
        STATE_PENDING){ //get permission to enter
20            in.remove(process);    }
21  else if(type == TYPE_REMOVE){...    }
```

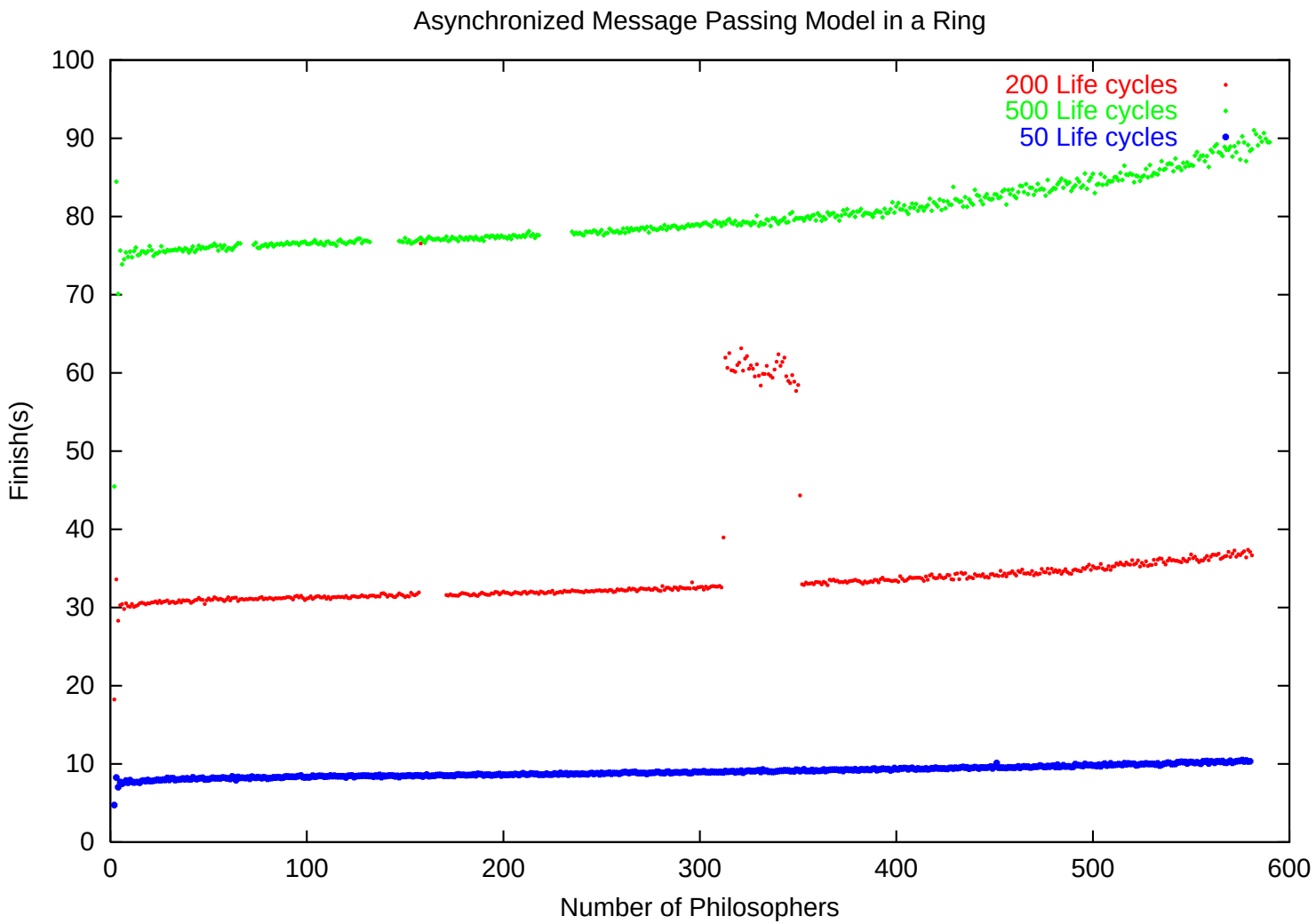
Philosopher Object

```
21  else if(type == TYPE_GRANT && state ==
        STATE_HOPEFUL){
22    grantedForkSet.add(process);
23    if (grantedForkSet.size() == forks.size()){
24      state = STATE_EATING;
25      //release fork by sending message
26      for (IDining outP: out){ //send outP allowance
            message} }
27  else if(type == TYPE_REJECT && state ==
        STATE_HOPEFUL){
28      for (IDining fork : grantedForkSet){//send fork
            failed message to release}
29      state = STATE_TRYING;
30      waitTime = (waitTime * 2) % MAX_WAIT_TIME;}
31  else if(type == TYPE_GRANT && state !=
        STATE_HOPEFUL){//release  }
```

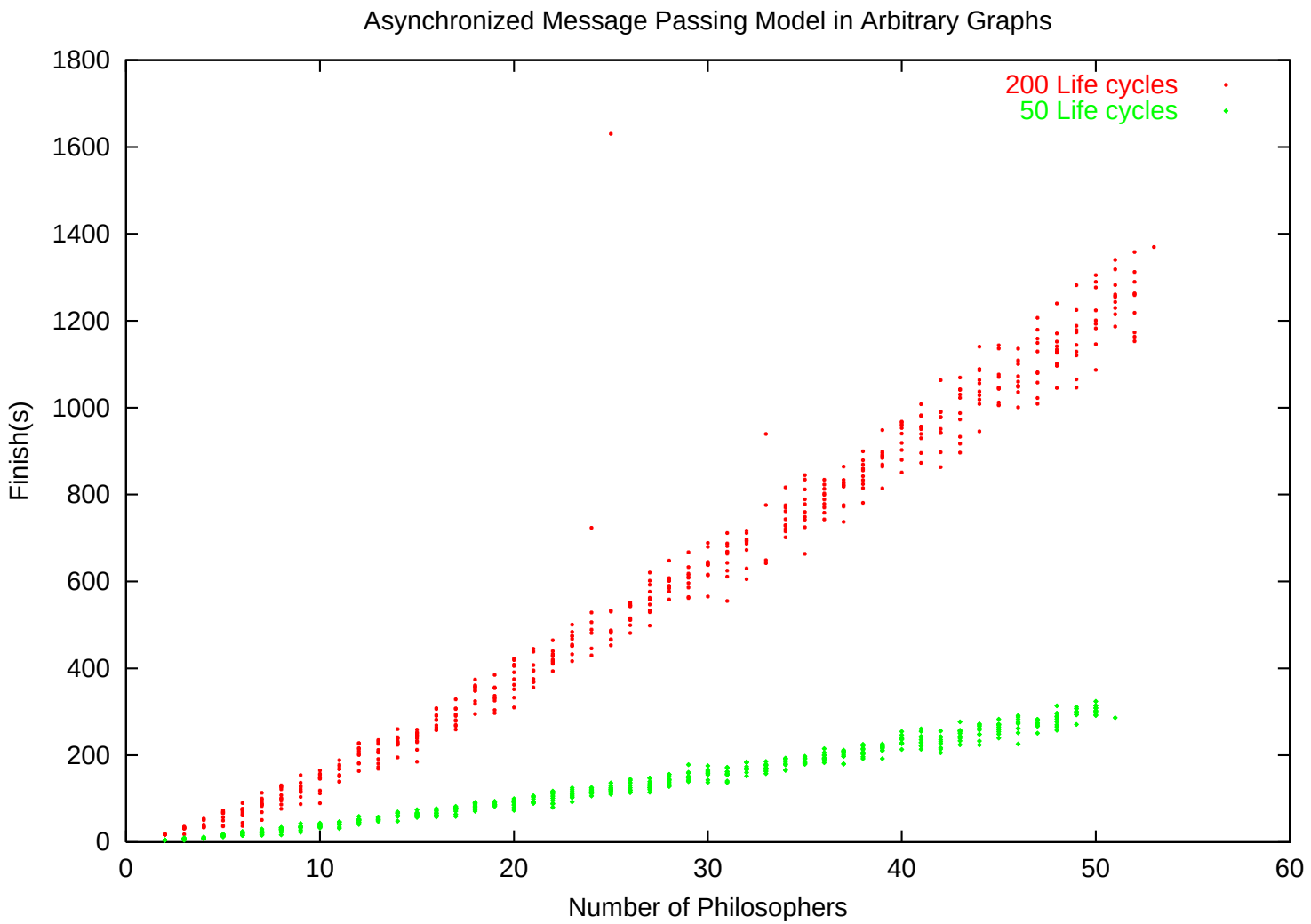
Result for Shared Memory Algorithm



Result for Message Passing Algorithm in the Ring



Result for Message Passing Algorithm in the arbitary graph



Other Thoughts

- Both achieve linear or approximately linear time and space complexity regarding the number of thread
- Messageing passing algorithm is much slower and bigger varaince since it is asynchronized, and has more states
- Degree of contending affects the complexity of running time dramatically
- However messageing passing increases the flexibility
- It seems liveness property achieved, still need to verify

Future Work

- Expand synchronized shared memory model to arbitrary graphs
- Test different settings to find the real impact of graph and degree contending for asynchronized message passing model
- Try different mutual exclusion mechanism to increase performance in asynchronized message passing model