# Towards parallelizing the Gillespie SSA

**Srivastav Ranganathan**
and
**Aparna JS**

**Indian Institute of Technology Bombay**
**Mumbai,**
**India**

# Gillespie Algorithm

- A stochastic simulation approach to study the time evolution of a system of reactions (processes)
- Each reaction occurs at an average rate
- The abundance of various species and their rates decide the propensity of each event to occur
- Many independent trajectories are generated to compute ensemble averaged statistical quantities

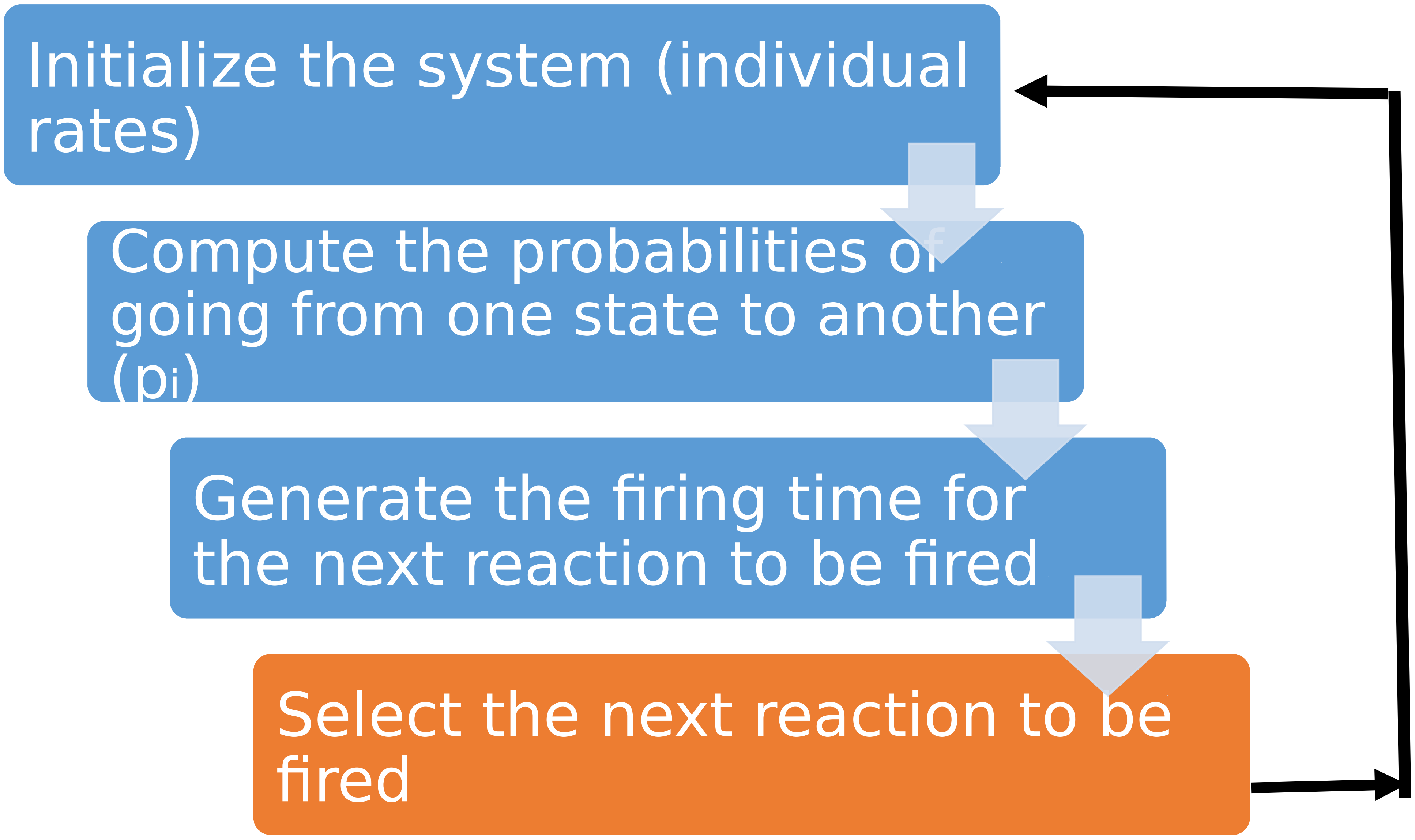
# Where is it used?

- In biological systems
- Outcomes of cellular processes are driven by stochasticity at a molecular level
- Deterministic approaches cannot capture the inherent randomness of biological systems

# The algorithm

- Assume a system whose in state A (its configuration)
- ‘M’ independent reactions are possible at any given time
- Each of these ‘i‘ processes could occur with a rate, ri
- The probability of occurrence of each of these processes is thus given by

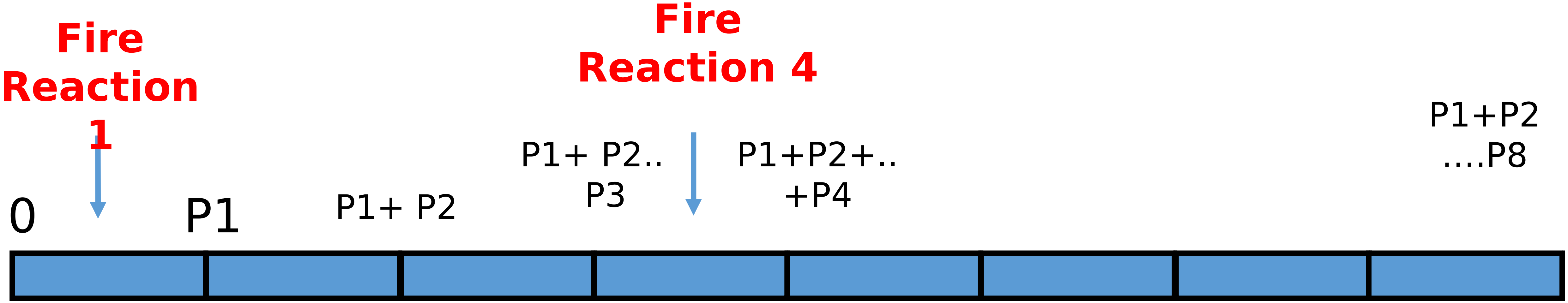
$$p_i = \frac{ri}{\sum ri}$$




Most expensive of all these steps

# Selecting the event to be fired

- Draw a uniform random number **ran1**

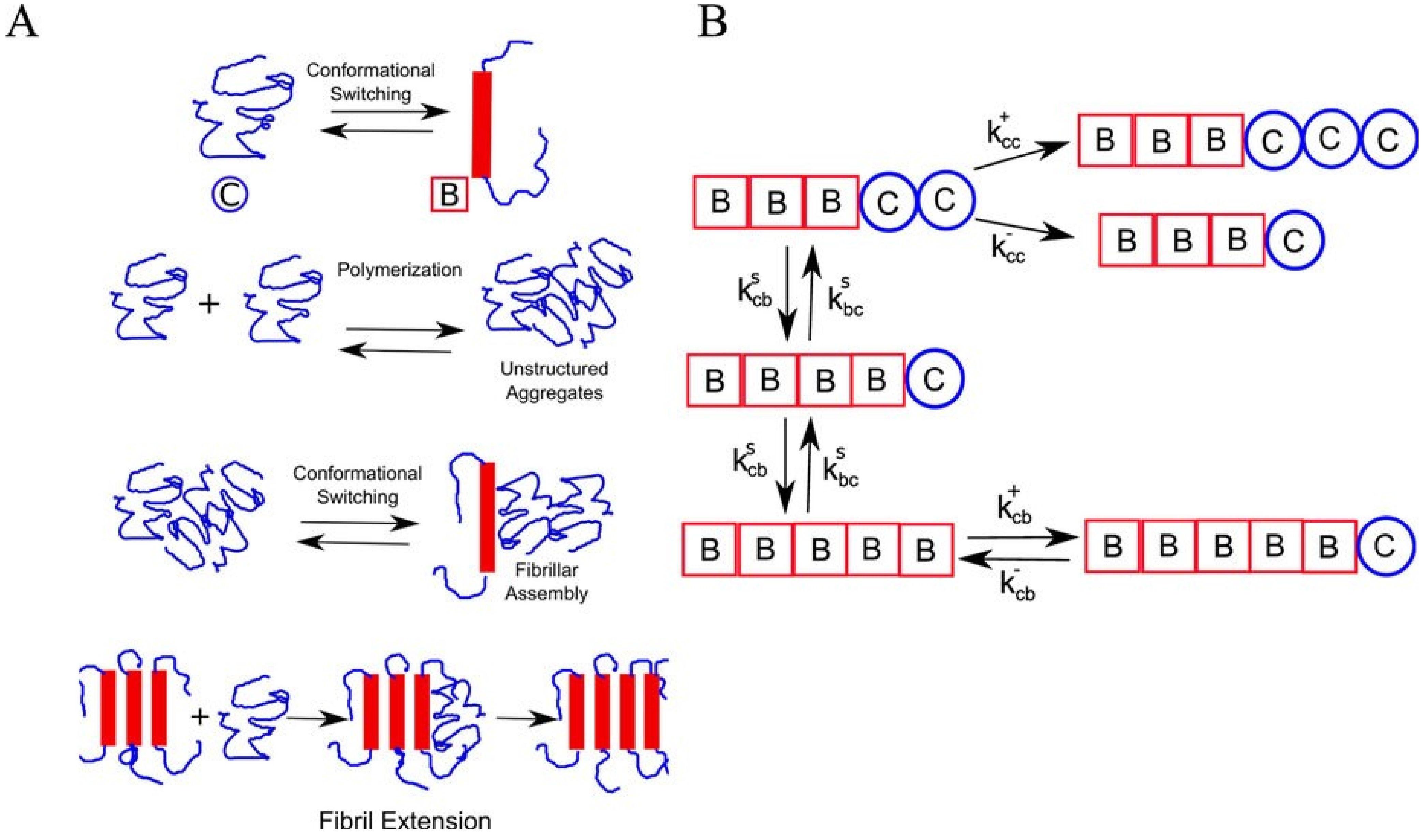



- Update the system configuration based on the fired reaction (abundance, rates etc)

- Update the time based on the exponential distribution of wait time between events (

$$dt = \frac{1}{\sum ri} ln \frac{1}{ran2}$$

This search for the next event to be fired is really expensive if there is a large reaction space!

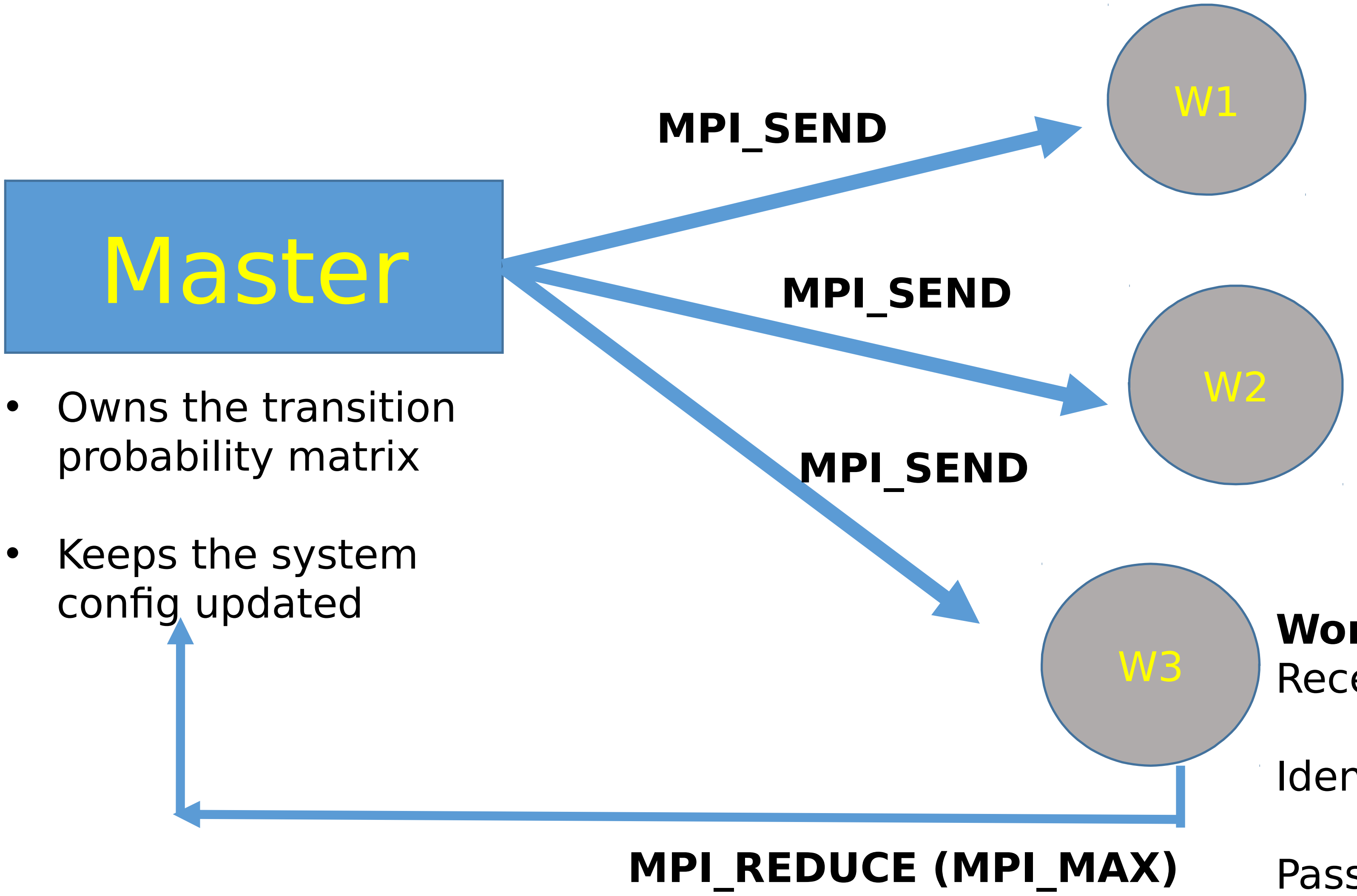


A
Conformational Switching
C
B
Polymerization
Unstructured Aggregates
Conformational Switching
Fibrillar Assembly
Fibril Extension
B
$k^{+}_{cc}$
$k^{-}_{cc}$
$k^{s}_{cb}$
$k^{s}_{bc}$
$k^{+}_{cb}$
$k^{-}_{cb}$
B B B C C C
B B B C C
B B B C
B B B B C
B B B B B
B B B B B C

```c
for(n=0;n<fibnum;n++)
   {
   if(n==0)
   {
     if ((ranevent < P_sum[4]) && (ranevent > P_init))
           {
   if ((P_init < ranevent) && (ranevent < P_sum[n*num_events+0]))
        {
      fibsize[n] = fibsize[n] + 1;
      N_A = N_A - 1 ;
      rccount[n] = rccount[n] + 1;
      break;
        }

    else if ((P_sum[n*num_events+1] > ranevent) && (ranevent > P_sum[n*num_events+0]))
        {
       fibsize[n] = fibsize[n] - 1 ;
       N_A = N_A + 1 ;
       rccount[n] = rccount[n] - 1;
       break;
        }

    else if ((P_sum[n*num_events+2] > (ranevent)) && (ranevent > P_sum[n*num_events+1]))
        {
       fibsize[n] = fibsize[n] - 1 ;
       betacount[n] =  betacount[n] - 1;
       N_A = N_A + 1 ;
       break;
        }

    else if ((P_sum[n*num_events+3] > (ranevent)) && (ranevent > P_sum[n*num_events+2]))
        {
       betacount[n] = betacount[n] + 1;
       rccount[n] = rccount[n] - 1;
       break;
```

C ▾    Tab Width: 8 ▾    Ln 1, Col

# Our attempt (Scheme 1, One-One communications)




- Owns the transition probability matrix
- Keeps the system config updated

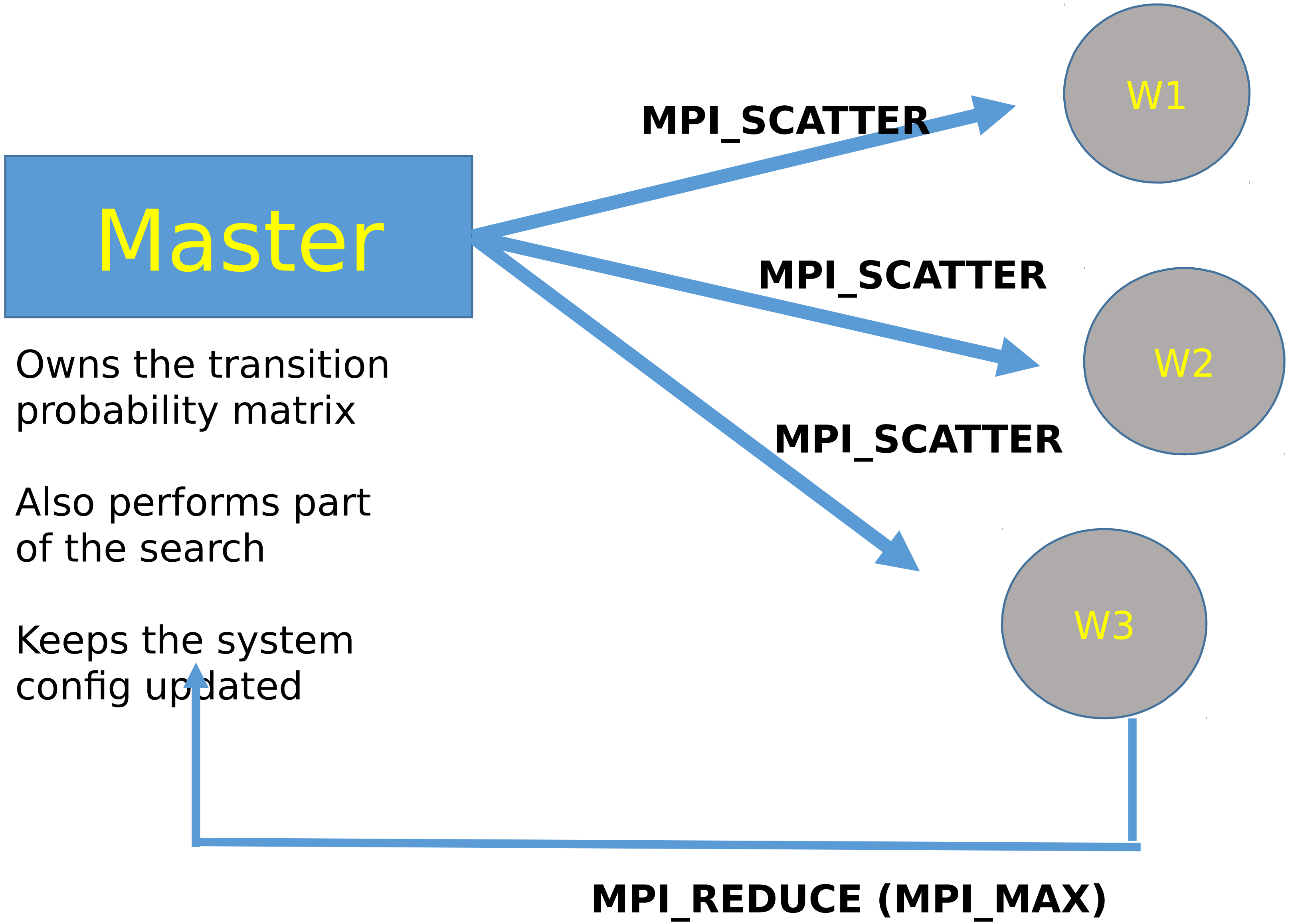
**Worker Nodes:**
Receive blocks of the search space

Identify the event to be fired

Pass the event info into a buffer, if a hit is received

```
    }
      }

MPI_Isend(P_send ,num_events*(fibnum/(comm_size-1))+1, MPI_DOUBLE, n_prc, n_prc, MPI_COMM_WORLD,&request);

n_prc++;
}
  }
ranevent = ran2(&id);
work_send[0]=0;
work_send[1]=0;

   if(my_rank != 0)
   {
    MPI_Recv(P_rec ,num_events*(fibnum/(comm_size-1))+1 , MPI_DOUBLE, 0, my_rank, MPI_COMM_WORLD,MPI_STATUS_IGNORE);

   if((ranevent>P_rec[0])&&(ranevent<P_rec[fibnum*num_events/(comm_size-1)]))
   {
      for(k=0;k<(fibnum/(comm_size-1));k++)
    {
       for(j=0;j<num_events;j++)
     {
         if ((P_rec[k*num_events+j] < ranevent) && (ranevent < P_rec[k*num_events+j+1]))
        {
       event_id=k*num_events+j;
       work_send[0]=event_id+1;
       work_send[1]=my_rank*((fibnum/(comm_size-1)))-((fibnum/(comm_size-1))-k);
       break;
        }
      }
    }
   }
  }
MPI_Barrier( MPI_COMM_WORLD);
MPI_Reduce(work_send,work_rec,2,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
```

# Our attempt (Scheme 2, Collective communication)




- Owns the transition probability matrix
- Also performs part of the search
- Keeps the system config updated

**Worker Nodes:**
Receive blocks of the search space

Identify the event to be fired

Pass the event info into a buffer, if a hit is received

```
MPI_Scatter(P_sum,num_events*(fibnum/(comm_size)),MPI_DOUBLE,P_rec,num_events*(fibnum/(comm_size)),MPI_DOUBLE,0,MPI_COMM_WORLD);
MPI_Scatter(Last_sum,1,MPI_DOUBLE,&Last_rec,1,MPI_DOUBLE,0,MPI_COMM_WORLD);

ranevent = ran2(&id);
work_send[0]=0;
work_send[1]=0;

   if(ranevent>Last_rec && ranevent<P_rec[((fibnum/comm_size)*num_events)-1])
 {
   for(k=0;k<(fibnum/(comm_size));k++)
     {
       for(j=0;(j<num_events);j++)
         {

     if(k*num_events+j == 0)
       {
         if((ranevent>Last_rec) && (ranevent < P_rec[0]))
           {
             work_send[0]=0;
             work_send[1]=my_rank*(fibnum/comm_size)+0;
           }
        }

   else if((P_rec[k*num_events+j-1] < ranevent) && (ranevent < P_rec[k*num_events+j]))
    {
       event_id=k*num_events+j;
       work_send[0]=event_id;
       work_send[1]=my_rank*(fibnum/comm_size)+k;
       break;
               }
             }
           }
         }

MPI_Barrier(MPI_COMM_WORLD);
MPI_Reduce(work_send,work_rec,2,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
```

# What worked

- Our naïve serial code was optimized to minimize cache misses (a speedup of 1.5 times)
- The MPI code did give us correct results (compared with the serial code and analytical results!)
- Exposed us to a new way of thinking

# What did`nt?

- MPI codes show a speedup from 1 to 3 cores but scale poorly

- Performance slows down at 5 processes or more

- Probably due to huge communication overhead in our code

- Possibly revisit the whole algorithm or use a more parallel-friendly algorithm!

# Thank You!