Speculative synchronization in StarSs, a task-based programming model

Rahulkumar Gayatri

Outline

- StarSs (sequential ⟶ parallel)
  - Compiler directives (pragmas)
  - Runtime features (Dependency Graph)
  - Synchronization directives
- Software Transactional Memory (STM) based concurrency control for critical memory updates
- Speculative task generation

- Moose Parallelization (current work)
  - OpenMP and Pthread parallelization
  - Optimizations

StarSs Programming Model

- Implicit parallel programming model for widely used multi-core architectures
- Single parallel application code for SMPs, Cell B.E, GPUs, cluster
- Sequential application with pragmas
  - Pragmas (compiler directives) to annotate a task
  - Runtime schedules the tasks on the available resources
    - Thread pool (Main and Worker threads)
    - Task Dependency Graph (TDG)

Task pragma and its clauses

Application code is divided into fine grained tasks

Task pragma and its clauses

Application code is divided into fine grained tasks

#pragma css task clauses
void task (parameters)

Task pragma and its clauses

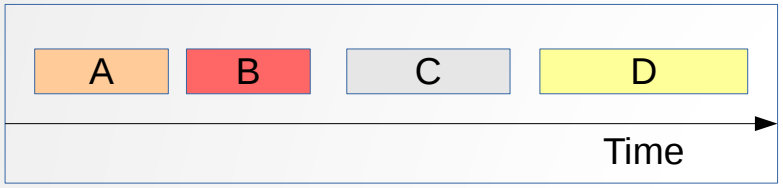
Application code is divided into fine grained tasks

#pragma css task clauses
void task (parameters)

input (list of parameters)
inout (list of parameters)
output (list of parameters)

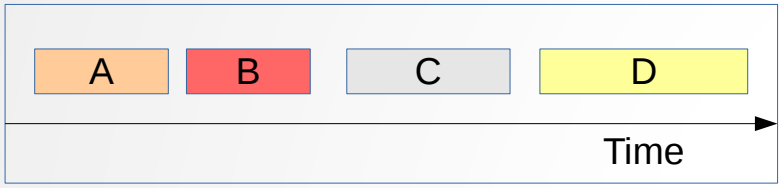
StarSs: annotation of tasks

Sequential code



StarSs: annotation of tasks

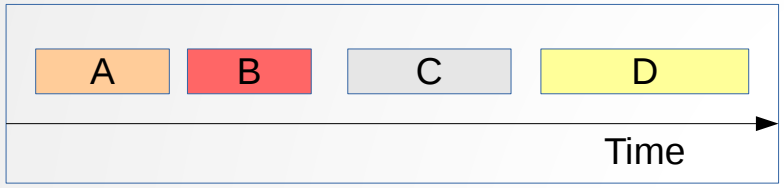
Sequential code



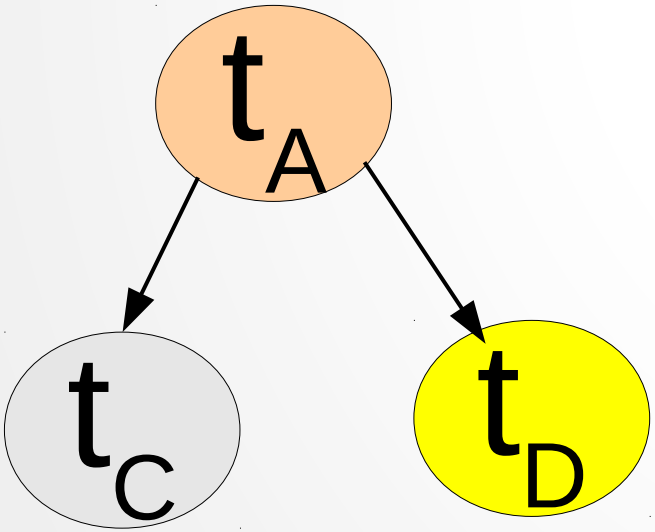

```
#pragma css task input(a) inout(b, c) \
output(d)
void A (a, b, c, d)

void B (e, f)

#pragma css task input(d) inout(b)
void C (d, b)

#pragma css task input(d) inout(c) output(e)
void D (d, c, e)
```

StarSs: TDG

Sequential code



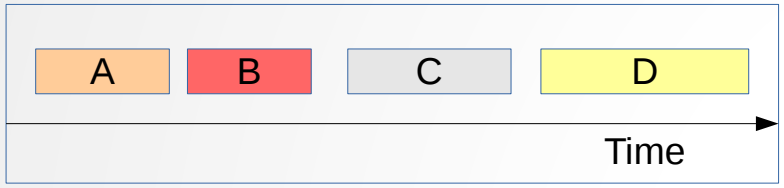
TDG



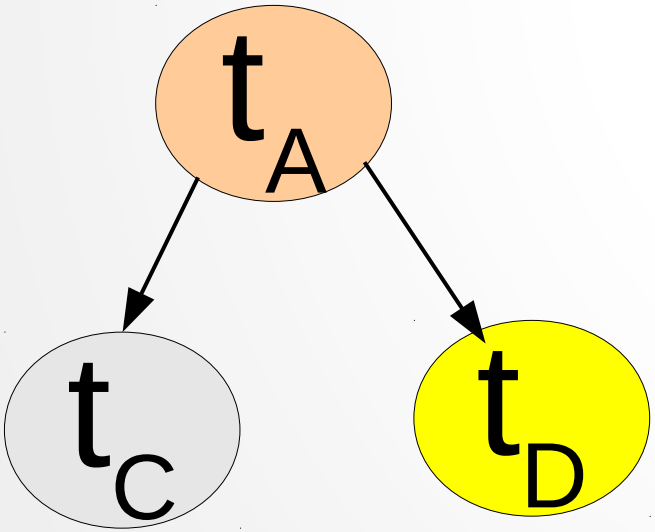

```
#pragma css task input(a) inout(b, c) \
output(d)
void A (a, b, c, d)


void B (e, f)


#pragma css task input(d) inout(b)
void C (d, b)


#pragma css task input(d) inout(c) output(e)
void D (d, c, e)
```

StarSs: scheduling

Sequential code

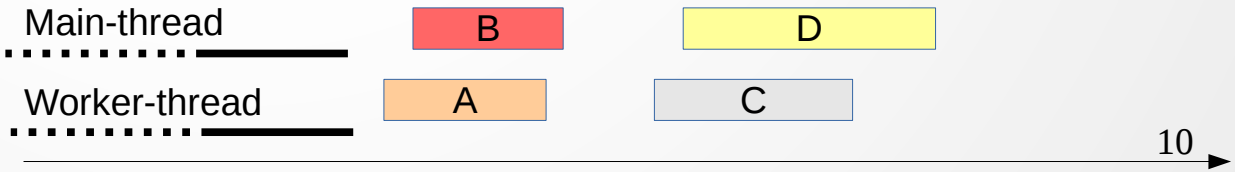


TDG

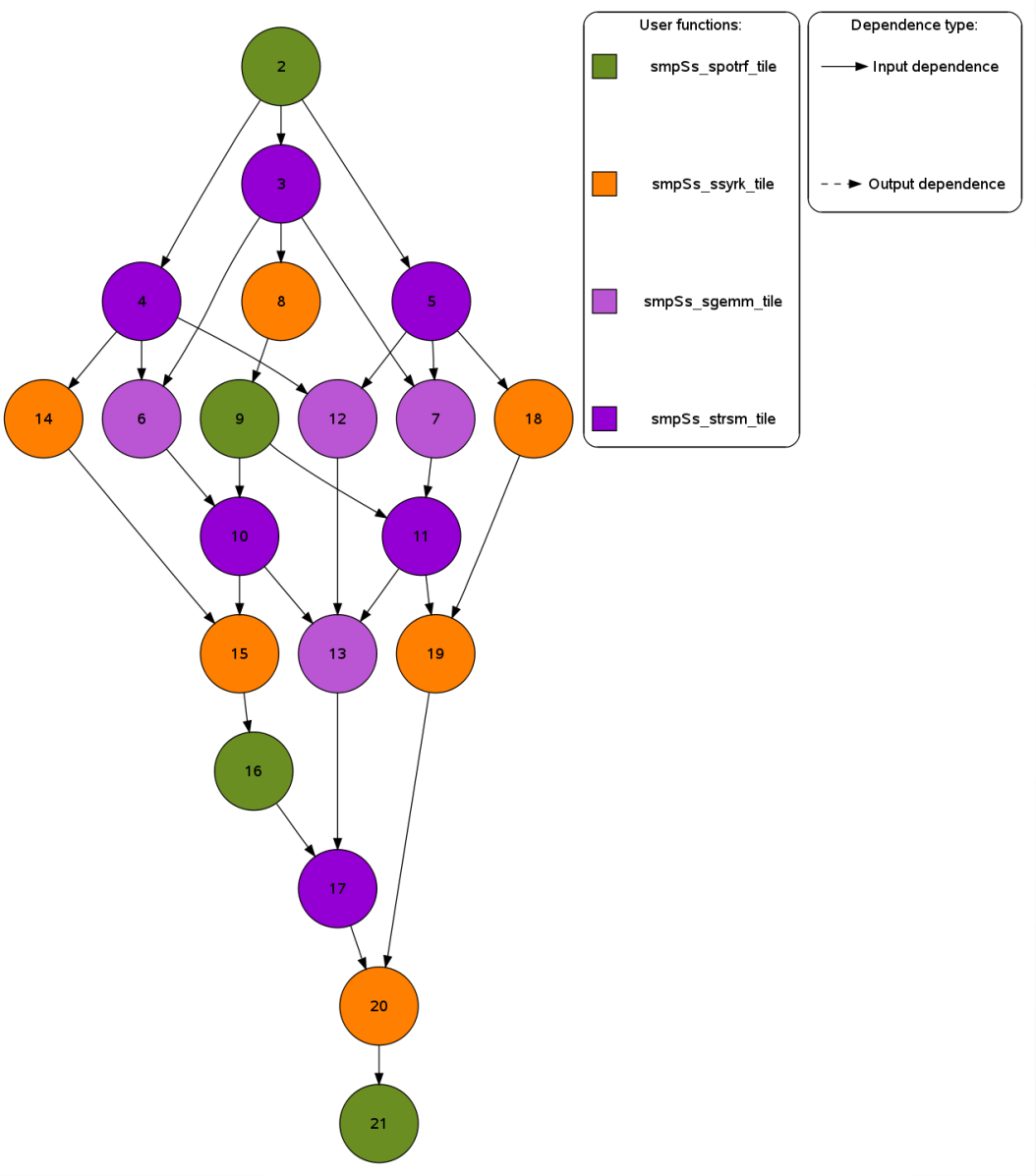



```
#pragma css task input(a) inout(b, c) \
output(d)
void A (a, b, c, d)


void B (e, f)


#pragma css task input(d) inout(b)
void C (d, b)


#pragma css task input(d) inout(c) output(e)
void D (d, c, e)
```

SMPSs scheduling



Cholesky TDG



StarSs: critical updates

- Reduction clause
  - does not create a dependency edge
- Need synchronization to maintain correctness

StarSs: locks

```
# pragma css task input (n, j, a[n]) inout (results)\
                              reduction (results)

void nqueens_task (int n, int j, char* a, int* results)
{
  ...
# pragma css mutex lock (results)
      *results = *results + local_sols;
# pragma css mutex unlock (results)
...
}
```

Drawbacks of lock-based synchronization

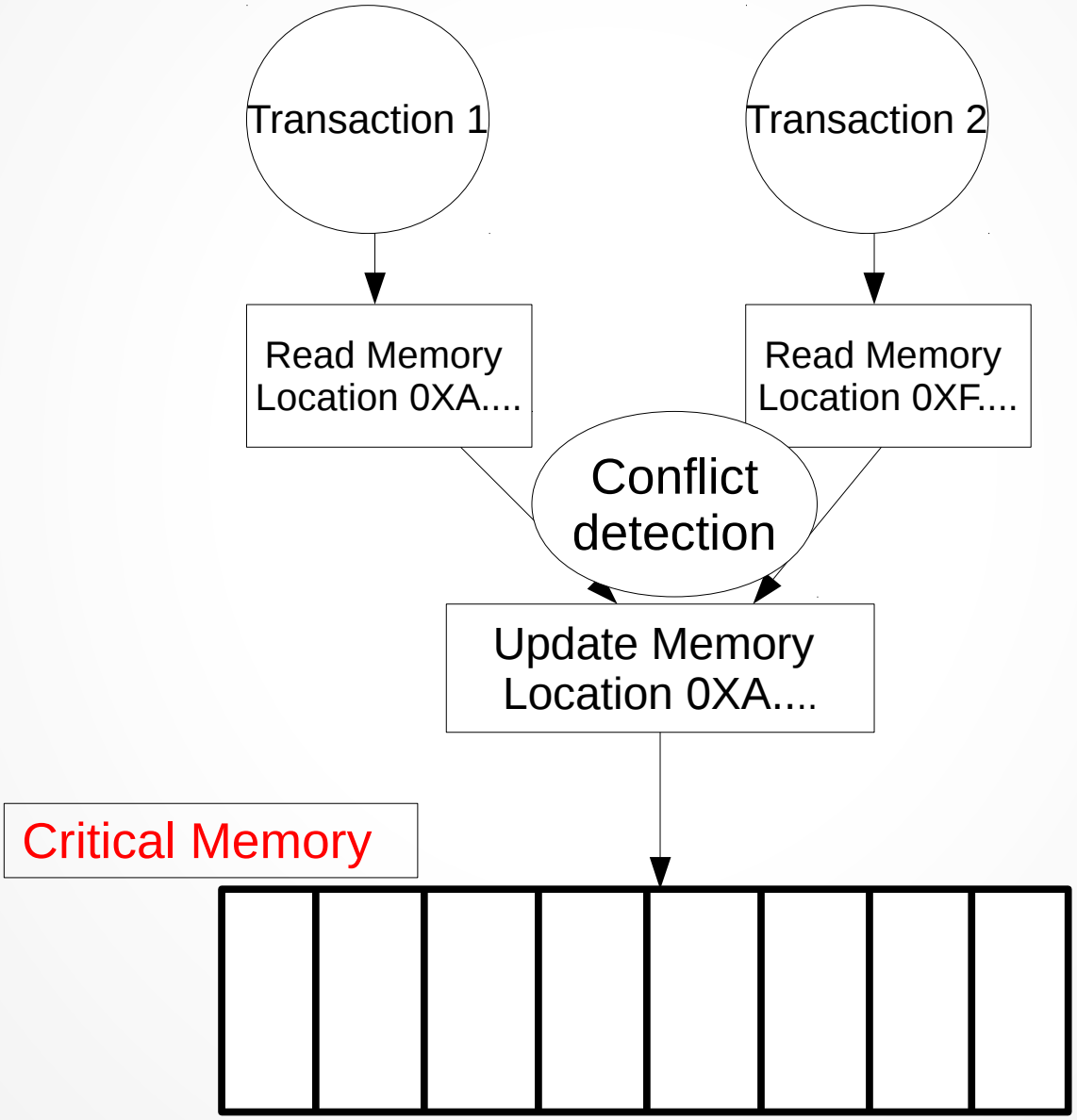
- Deadlock – two concurrent threads wait for each other to release a resource, thus neither progress
- Livelock – state of the threads change with regard to each other but neither progress
- Priority inversion – Lower priority thread blocks the higher priority thread
- Difficult to compose and debug
- Not scalable

Software Transactional Memory (STM)

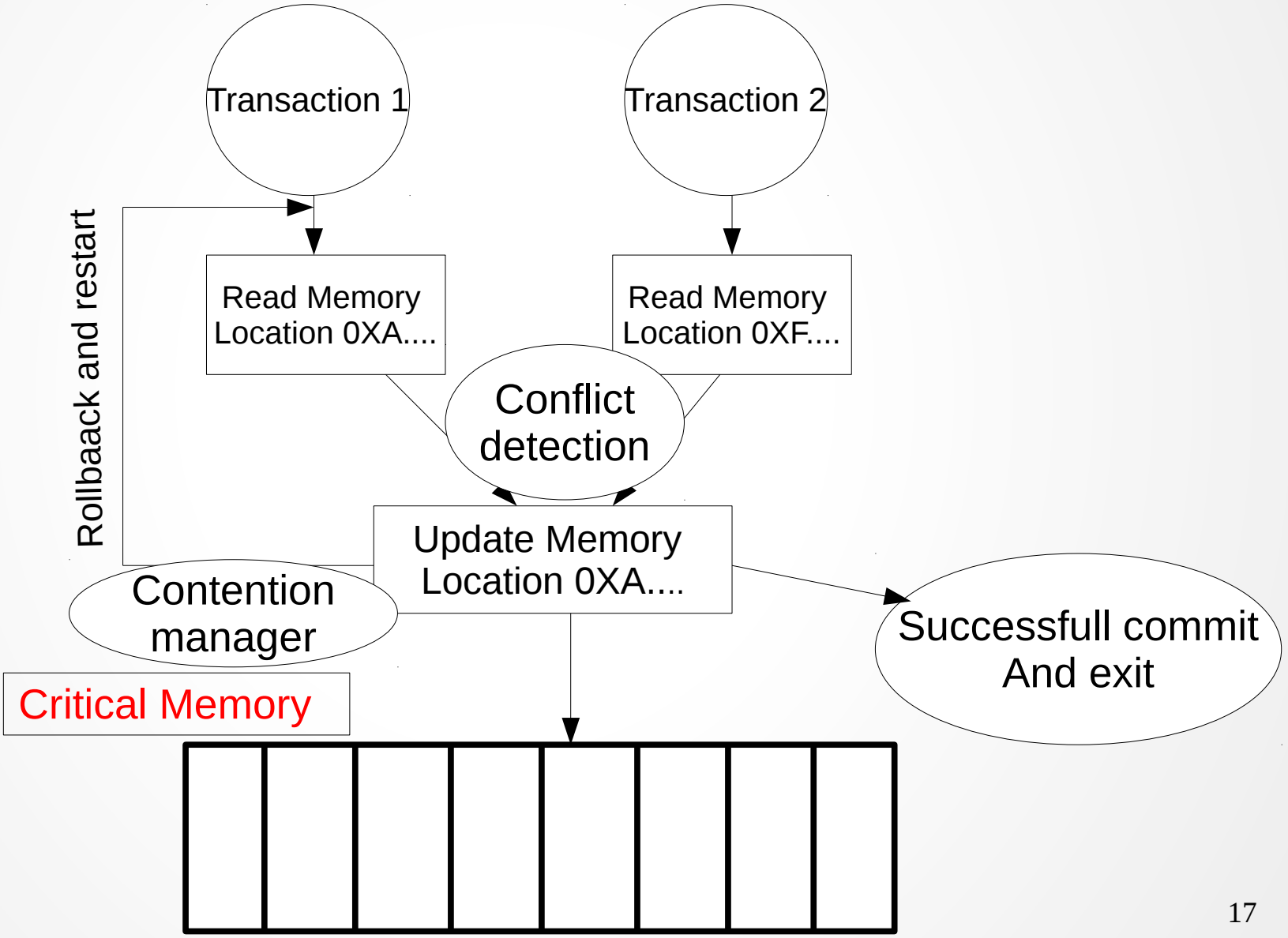
Transaction

Series of read and writes to memory

STM: conflict detection



STM: contention manager



TinySTM

- A light weight STM library
- 3 versions
  - WRITE_BACK_ETL (check conflicts at encounter time)
  - WRITE_BACK_CTL (check conflicts at commit time)
  - WRITE_THROUGH (direcly update shared memory)

TinySTM in StarSs

- Main and worker threads perform the necessary implementations
  - Changes in configuration for the appropriate linking
- Lock pragma starts a transaction and unlock pragma commits the transaction

Compiler vs runtime

- Stack calls in TinySTM library make runtime implementation difficult
- StarSs compiler replaces lock and unlock pragmas with TinySTM transaction calls.

Compiler transformations of lock and unlock

```
Application code

…
#pragma css mutex lock (results)
   *results = *results + local_sols ;
#pragma css mutex unlock (results)
...
```

```
Start the transaction
Save the stack context
Load parameter into the transactional context
```

```
*results = *results + local_sols ;
```

```
Store the updated parameter to memory
Commit the transaction
```

Applications with reductions

- Nqueens
- BFS

→ STM better than Locks

- Specfem3D
- Gmeans

→ Locks better than STM

Locks vs Transactions performance

- Both versions scale similarly
- STM performed better in applications with high lock contention
- More beneficial to update multiple memory locations in a single transaction

STM overhead

- STM is notorious for high software overhead
  - Setting up transaction environment
  - Saving stack information
  - Conflict detection
  - Contention manager
- Hybrid-TM is the way forward

2nd Phase of PhD

Speculative memory updates to speculative task execution

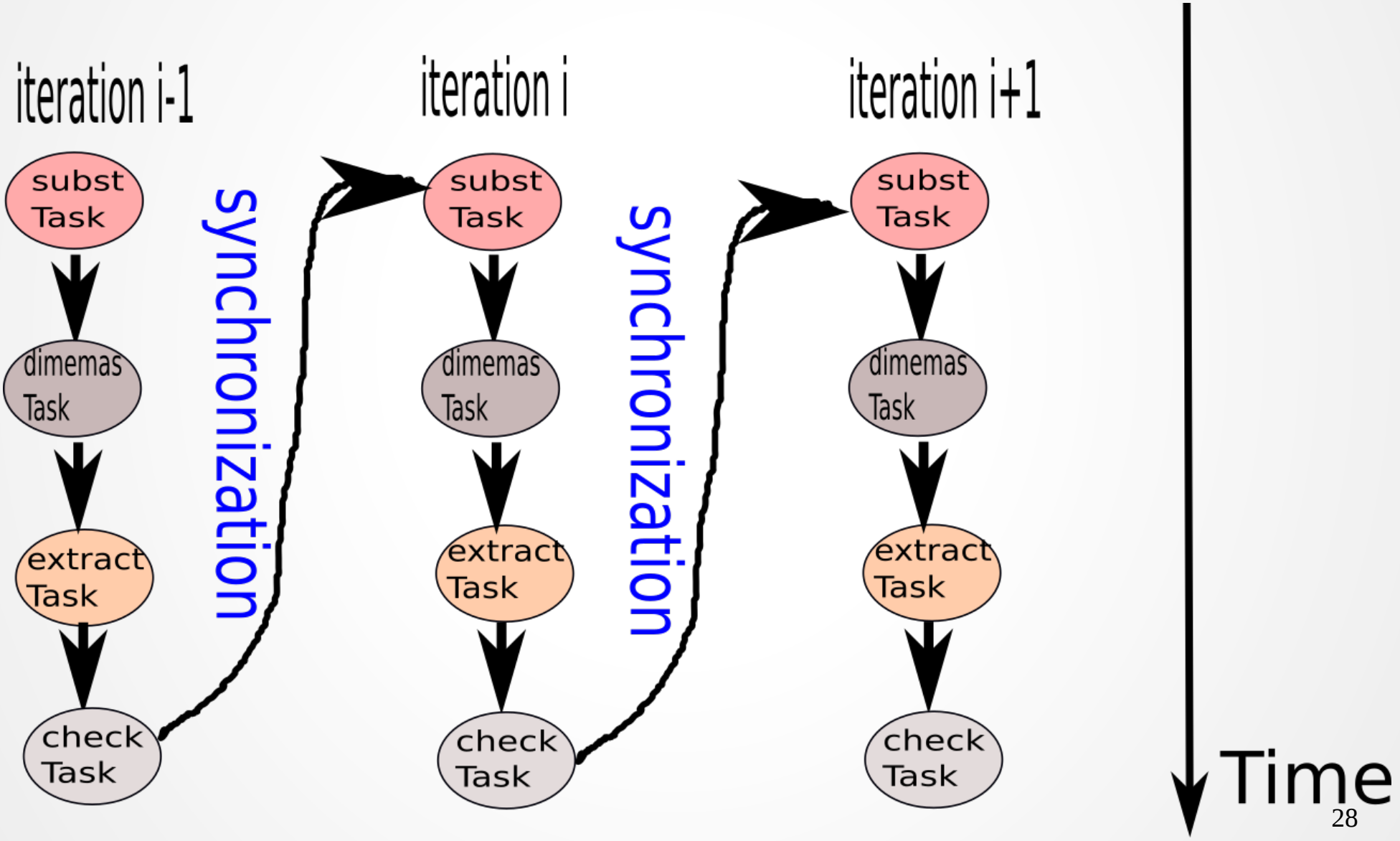
Necessary of task sychronization

- Separation of task generation and execution makes synchronization a necessity
  - #pragma css wait on (addr)
  - #pragma css barrier

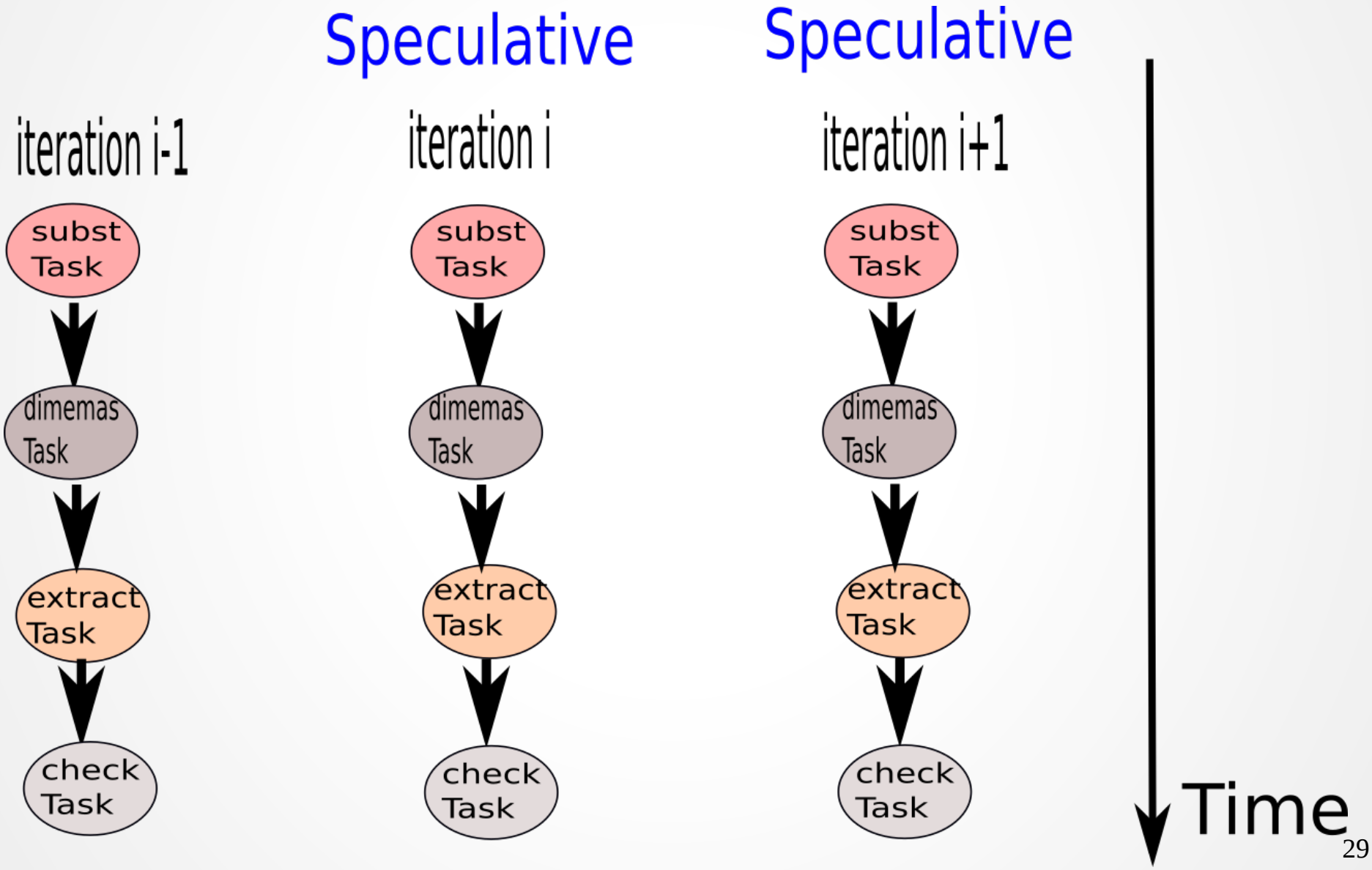
Usage of synchronization

```
while(!goal_achieved)
{
    newBFD = pop_queue();
    subst(refCFG, newBFD, newCFG);
    dimemas(newCFG, trace, dimOUT);
    extract(newBWD, dimOUT, finalOUT);
    check(finalOUT, goal_achieved);
 #pragma css wait on(goal_achieved)
}
```

TDG


iteration i-1
iteration i
iteration i+1
subst Task
dimemas Task
extract Task
check Task
synchronization
synchronization
Time

Speculative TDG (ideal)



Speculative directive: clauses

```
#pragma css speculate values(finalOUT) wait(goal_achieved)

  while(!goal_achieved)
  {
     newBFD = pop_queue();
     subst(refCFG, newBFD, newCFG);
     dimemas(newCFG, trace, dimOUT);
     extract(newBWD, dimOUT, finalOUT);
     check(finalOUT, goal_achieved);
   #pragma css wait on(goal_achieved)
  }
```

Speculative directive: clauses

```
#pragma css speculate values(finalOUT) wait(goal_achieved)
```

values – protection in case of mis-speculation
wait – determines the progress of the loop

```
while(!goal_achieved)
{
    newBFD = pop_queue();
    subst(refCFG, newBFD, newCFG);
    dimemas(newCFG, trace, dimOUT);
    extract(newBWD, dimOUT, finalOUT);
    check(finalOUT, goal_achieved);
}
```

Success of Speculation

- Valid tasks – Successfull speculation
  - Tasks generated even with the synchronization
- Invalid tasks – Undo their results
  - Loop predicate evaluated to false

Speculative directive: implementation

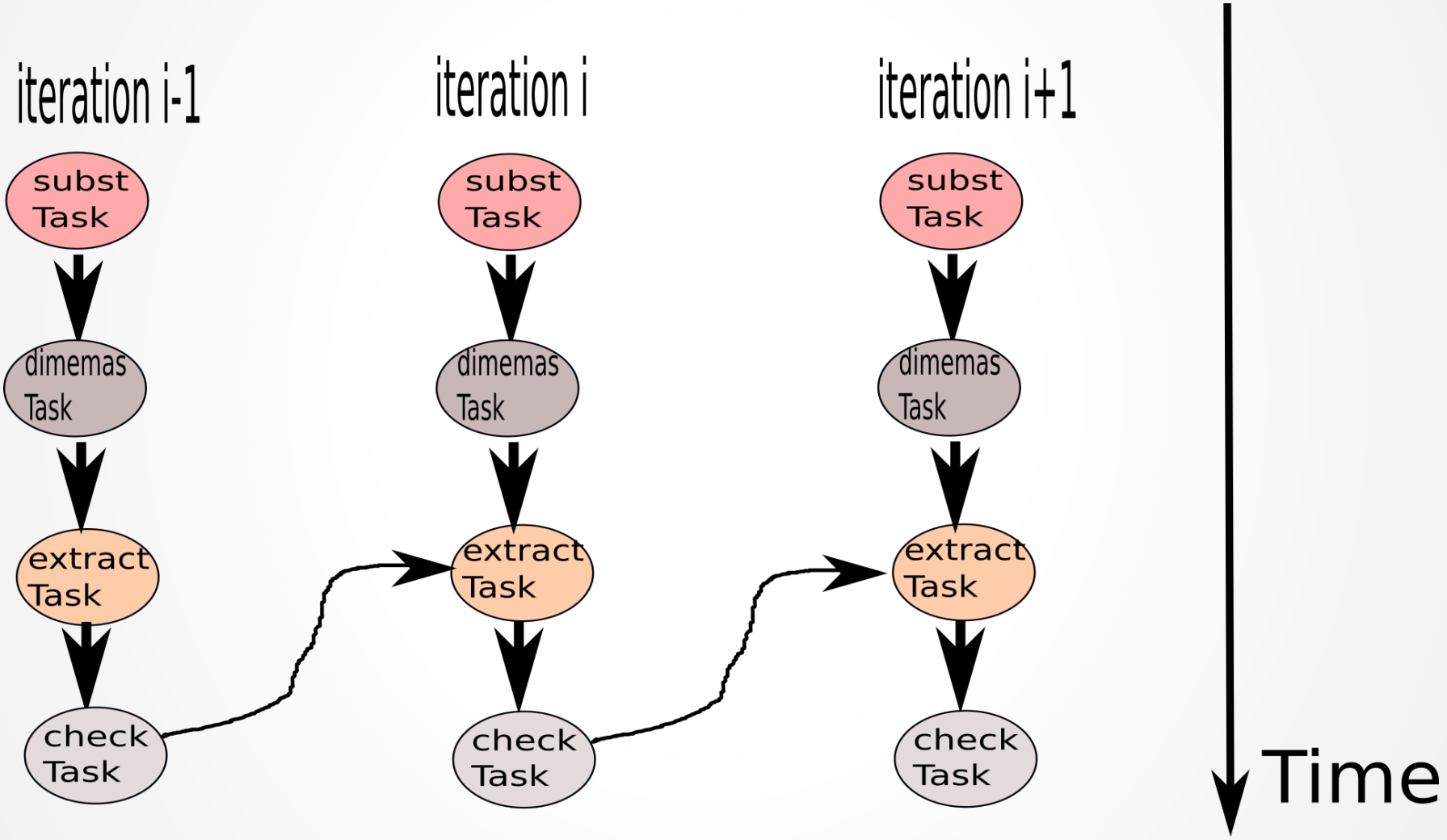
- *Speculative* tasks - transactions
- Parameters from values are loaded into the transactional context
- Store values only if speculation is successful
- Use of a **guard** function to evaluate the success of speculation

Guard function

Guard function

```
int guard (void *speculate_params[1])
{
  return (!*(speculate_params[0]));
}
speculate_params[0] = &goal_achieved ;
```

Speculative TDG



Types of Speculative tasks

- Data-dependent speculative tasks
  - Cross-iteration dependencies
  - Tasks from iteration *i* consume values from tasks in *i-1*
  - Overlap task generation with task execution
- Control dependent speculative tasks
  - Tasks from iteration *i* are independent of tasks from *i-1*
  - Parallel execution of tasks

Applications for testing

- Gauss – Seidel
- Jacobi
- Red-black
- Kmeans

Observations

- Non-speculative versions perform better
  - Both versions scale similarly
- Better performance with increasing number of threads
- TinySTM overhead cannot be overcome with the additional parallelism obtained

TinySTM overhead

- Less number of bigger transactions is beneficial
  - Bigger tasks more beneficial than smaller tasks
  - Longer rollbacks versus higher number of TinySTM calls
- Increase task granularity
  - Instead of increasing the number of tasks in the TDG, we reduce them to compensate for the overhead

Data-version based speculative task execution

- TinySTM load and store a major overhead
- Redundant conflict detection mechanisms
- Data-versions to maintain correctness

Data-version based speculative task execution

- TinySTM load and store a major overhead
- Redundant conflict detection mechanisms
- Data-versions to maintain correctness

Overhead

1) 1 copy for successful speculation
2) 2 copies for mis-speculation
3) Evaluation of the guard function

Observations

- On average a 20% performance improvement over non-speculative application
  - Significant since the improvement is over an already parallelized code
- The benefits of speculation are better observed with increasing number of threads
- With increasing problem sizes more number of threads are needed for performance benefits

Optimizations

- Update original memory locations inside speculative tasks
- Limit on the number of iterations to be speculated upon
  - speculation_tasks, an environment variable

Current Work

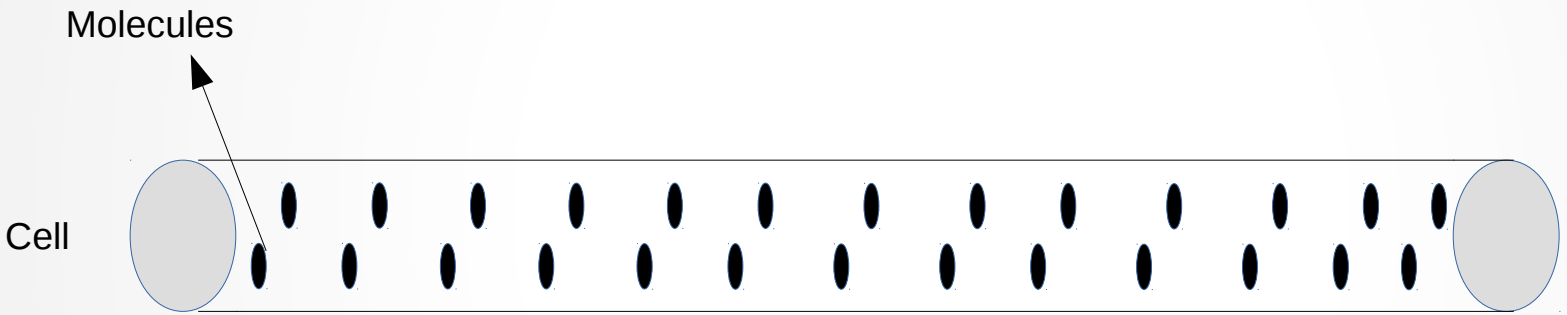
Parallelization of Moose

Current Work

MOOSE, a Multiscale Object-Oriented Simulation Environment.

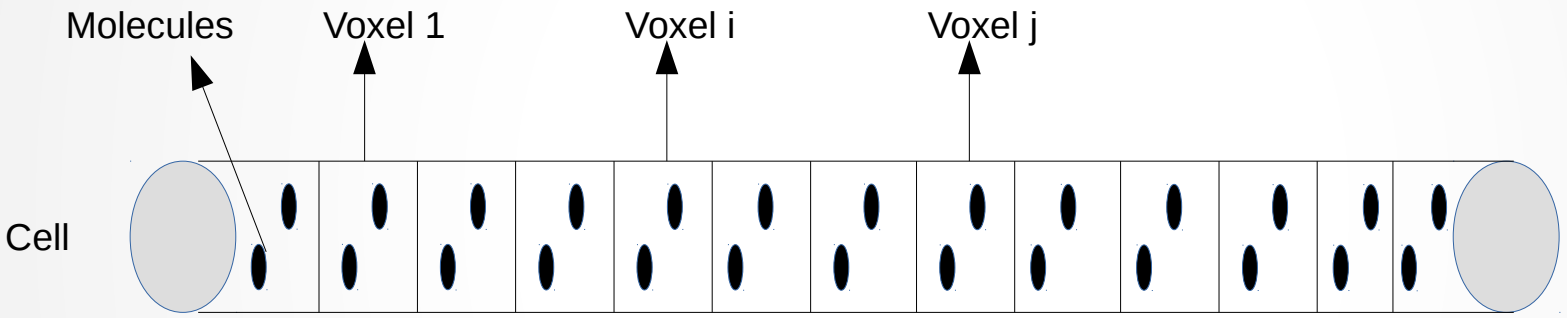
Cell/Neuron division

MOOSE, a Multiscale Object-Oriented Simulation Environment.



Cell/Neuron division

MOOSE, a Multiscale Object-Oriented Simulation Environment.



- A cell is subdivided into Voxels
- Independent computations in each voxel at a given timestep

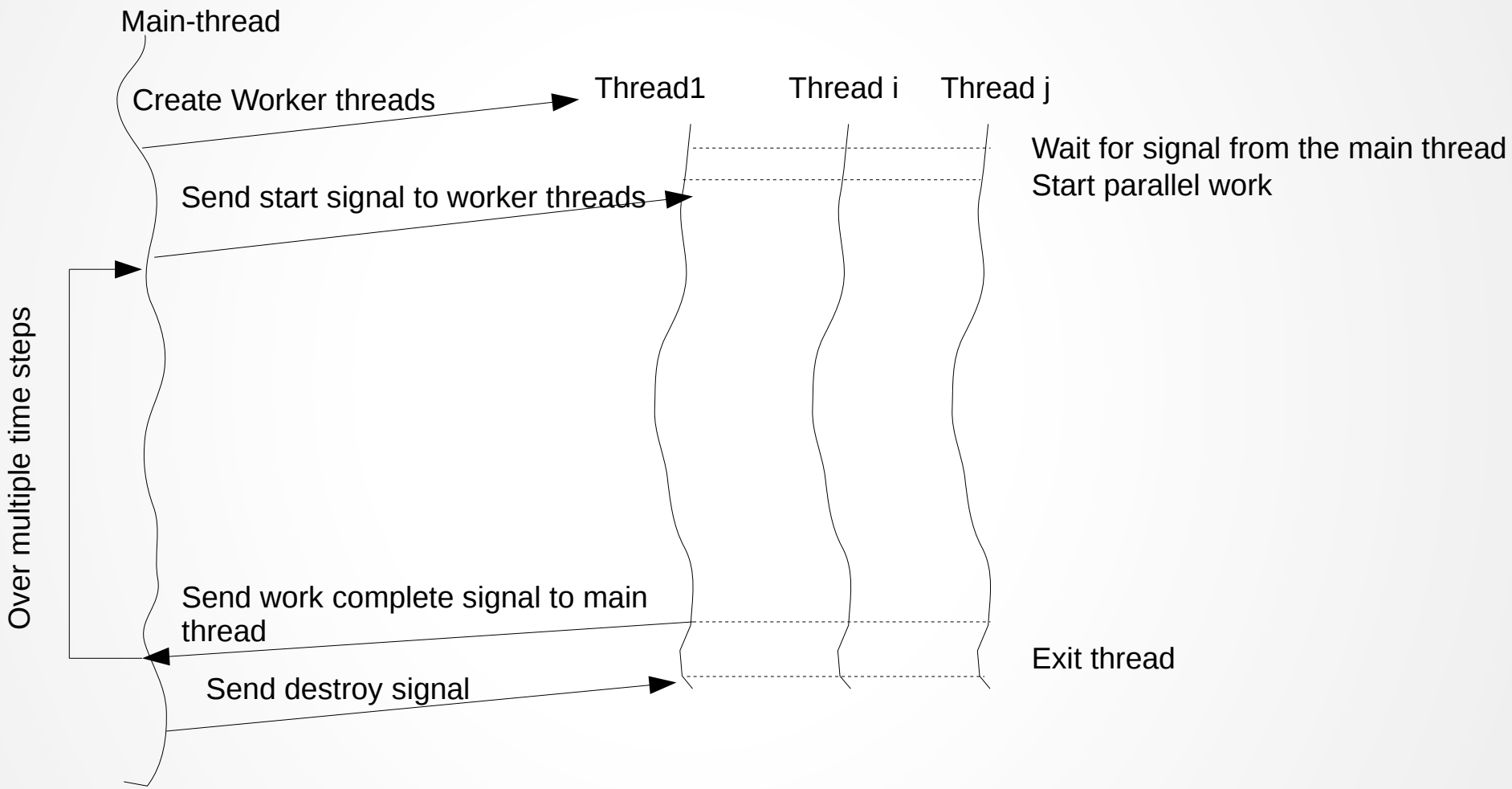
Single cell simulation

- Chemical interactions on the molecules
- Kinetic solver
  - Large volume (deterministic behavior)
  - Simulates the behavior of the molecule using linear transformation (rungekutta method of order 5)
- Stochastic solver
  - Small volume
  - Simulation is performed over individual molecules
- Diffusion solver
  - Calculate the diffusion gradient

Programming Models used

- OpenMP
  - Parallel-for and task-based paralleization
  - Ease of use
- Pthreads
  - More flexibility
  - Optimized on thread creation-destruction by the use of semaphores

Optimization of pthreads using semaphores



Optimizations

- Memory alignment
- Reduce the number of OpenMP parallel pragmas
- Remove hyperthreading

Results of Parallelization on 8 threads

- Kinetic solver
  - OpenMP – 5X
  - Pthreads – 4.5X
- Stochastic Solver
  - OpenMP – 5.5X
  - Pthreads – 4.8X
- Diffusion solver
  - OpenMP – 3.8X
  - Pthreads – 3.7X

Conclusion

- PhD Thesis
  - Compiler and runtime development
  - Speculative synchronization (using STM)
- Cell simulation
  - Paralleization of solvers for chemical interactions

THANK YOU!!