Improving Speculative `taskloop` in Hardware Transactional Memory

Juan Salamanca and Alexandro Baldassin
São Paulo State University (Unesp)

Agenda

- Motivating example
- Background
- Speculative Taskloop (STL)
- Lost-Thread Effect and Solution
- Experimental Evaluation
- Conclusions

Agenda

- **Motivating example**
- Background
- Speculative Taskloop (STL)
- Lost-Thread Effect and Solution
- Experimental Evaluation
- Conclusions

Motivating Example

Motivating example

- susan_corners's loop (`loopV`)

```
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
    }
    …
}
```

Motivating example

- How can we parallelize this loop?

```
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
    }
    …
}
```

Motivating example

- Can we use OpenMP `parallel for` or `taskloop`?

```
#prama omp parallel for shared(n)
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
    }
    …
}
```

Motivating example

- Can we use OpenMP `parallel for` or `taskloop`?

```
#prama omp parallel for shared(n)
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
    }
    …
}
```

No!
The loop is non-DOALL

Motivating example

- Can we use OpenMP `parallel for` or `taskloop`?

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop shared(n)
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
    }
    …
}
```

No!
The loop is non-DOALL

Motivating example

- How about OpenMP `parallel for` and `ordered`?

```
#prama omp parallel for ordered(1) shared(n)
for(i=5 ; i < y_size-5; i++){
    #pragma omp ordered depend (sink:i-1)
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
    }
    …
    #pragma omp ordered depend (source)
}
```

Motivating example

- How about OpenMP `parallel for` and `ordered`?

Correct results, but poor performance

```
#prama omp parallel for ordered(1) shared(n)
for(i=5 ; i < y_size-5; i++){
    #pragma omp ordered depend (sink:i-1)
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
    }
    …
    #pragma omp ordered depend (source)
}
```

Motivating example

- How about OpenMP `parallel for` and `ordered`?

```
#prama omp parallel for ordered(1) shared(n)
for(i=5 ; i < y_size-5; i++){
    #pragma omp ordered depend (sink:i-1)
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
    }
    …
    #pragma omp ordered depend (source)
}
```

Correct results, but poor performance

`ordered` serializes the execution

Motivating example

- How about Speculative `taskloop` (STL)?

```
#pragma omp taskloop tls(S_SIZE) spec_private(n)
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        #pragma tls if_read(n)
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
        #pragma tls if_write(n)
    }
    …
}
```

Motivating example

- How about Speculative `taskloop` (STL)?

Depends on the LLVM OpenMP Runtime

```
#pragma omp taskloop tls(S_SIZE) spec_private(n)
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        #pragma tls if_read(n)
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
        #pragma tls if_write(n)
    }
    …
}
```

Motivating example

- How about Speculative `taskloop` (STL)?

```
#pragma omp taskloop tls(S_SIZE) spec_private(n)
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        #pragma tls if_read(n)
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
        #pragma tls if_write(n)
    }
    …
}
```

Depends on the LLVM OpenMP Runtime

Using `libomp12`: (`nonmonotonic`) Deadlock!

Motivating example

- How about Speculative `taskloop` (STL)?

```
#pragma omp taskloop tls(S_SIZE) spec_private(n)
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        #pragma tls if_read(n)
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
        #pragma tls if_write(n)
    }
    …
}
```

Depends on the LLVM OpenMP Runtime

Using `libomp12` and `monotonic` scheduling: some speed-ups.

Motivating example

- How about Speculative `taskloop` (STL)?

```
#pragma omp taskloop tls(S_SIZE) spec_private(n)
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        #pragma tls if_read(n)
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
        #pragma tls if_write(n)
    }
    …
}
```

Depends on the LLVM OpenMP Runtime

From deadlocks to 1.16× of speed-up

Motivating example

- How about Speculative `taskloop` (STL)?

```
#pragma omp taskloop tls(S_SIZE) spec_private(n)
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    if (x>0 && /*compare x*/){
        #pragma tls if_read(n)
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
        #pragma tls if_write(n)
    }
    …
}
```

Depends on the LLVM OpenMP Runtime

From deadlocks to 1.16× of speed-up

Agenda

- ~~Motivating example~~
- **Background**
  - **Tasks in OpenMP**
  - **`taskloop` construct**
  - **TLS on HTM**
- Speculative Taskloop (STL)
- Lost-Thread Effect and Solution
- Experimental Evaluation
- Conclusions

Background

Tasks in OpenMP

- Task parallelism does not focus on mapping parallelism to threads.

Tasks in OpenMP

- Task parallelism does not focus on mapping parallelism to threads.
- It is oblivious of the physical layout and focuses on exposing more parallelism.

Tasks in OpenMP

- Task parallelism does not focus on mapping parallelism to threads.
- It is oblivious of the physical layout and focuses on exposing more parallelism.
- Added to OpenMP version 3.0.

Tasks in OpenMP

- Task parallelism does not focus on mapping parallelism to threads.
- It is oblivious of the physical layout and focuses on exposing more parallelism.
- Added to OpenMP version 3.0.
- OpenMP 4.5 added a new construct called `taskloop`.

`taskloop` construct

- Parallelizes a loop by dividing its iterations into a number of tasks.

`taskloop` construct

- Parallelizes a loop by dividing its iterations into a number of tasks.
- Similar to `parallel for` (lack of `schedule` clause).

`taskloop` construct

- Parallelizes a loop by dividing its iterations into a number of tasks.
- Similar to `parallel for` (lack of `schedule` clause).
- `taskloop` is restricted to loops DOALL or with reduction patterns.

`taskloop` construct

- Parallelizes a loop by dividing its iterations into a number of tasks.
- Similar to `parallel for` (lack of `schedule` clause).
- `taskloop` is restricted to loops DOALL or with reduction patterns.
- This restriction precludes the parallelization of many *may* DOACROSS loops.

An example (toy1)

```
for(i=0 ; i < N; i++){
    glob=…;
    A[i]=glob*i;
}
```

An example (toy1)

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop ...
for(i=0 ; i < N; i++){
   glob=…;
   A[i]=glob*i;
}
```

An example (toy1)

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop ...
for(i=0 ; i < N; i++){
   glob=…;
   A[i]=glob*i;
}
```

DOALL parallelization

Another example (toy2)

```
for(i=0 ; i < N; i++){
   if (/*cond*/)
      glob++;
   else
      glob=i;
   ...
}
```

Another example (toy2)

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop shared(glob)...
for(i=0 ; i < N; i++){
   if (/*cond*/)
      glob++;
   else
      glob=i;
   ...
}
```

taskloop construct

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop shared(glob)...
for(i=0 ; i < N; i++){
   if (/*cond*/)
      glob++;
   else
      glob=i;
   ...
}
```

Is it correct?

taskloop construct

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop shared(glob)...
for(i=0 ; i < N; i++){
   if (/*cond*/)
      glob++;
   else
      glob=i;
   ...
}
```

No, the loop is non-DOALL

ordered construct

```
for(i=0 ; i < N; i++){
   if (/*cond*/)
      glob++;
   else
      glob=i;
   ...
}
```

may DOACROSS loop

ordered construct

```
#pragma omp parallel for ordered(1) shared(glob)
for(i=0 ; i < N; i++){
   #pragma omp ordered depend (sink:i-1)
   if (/*cond*/)
      glob++;
   else
      glob=i;
   #pragma omp ordered depend (source)
   ...
}
```

Using `ordered`, the parallelization is correct

TLS on HTM

- TLS requires hardware mechanisms that support four primary features: conflict detection, speculative storage, in-order (`monotonic`) commit of transactions, and transaction roll-back.

TLS on HTM

- TLS requires hardware mechanisms that support four primary features: conflict detection, speculative storage, in-order (`monotonic`) commit of transactions, and transaction roll-back.
- HTM implements three out of the four key features required by TLS: conflict detection, speculative storage, and transaction roll-back.

TLS on HTM

- TLS requires hardware mechanisms that support four primary features: conflict detection, speculative storage, in-order (`monotonic`) commit of transactions, and transaction roll-back.
- HTM implements three out of the four key features required by TLS: conflict detection, speculative storage, and transaction roll-back.
- Thus these architectures have the potential to be used to implement it.

TLS on HTM

- TLS requires hardware mechanisms that support four primary features: conflict detection, speculative storage, in-order (`monotonic`) commit of transactions, and transaction roll-back.
- HTM implements three out of the four key features required by TLS: conflict detection, speculative storage, and transaction roll-back.
- Thus these architectures have the potential to be used to implement it.
- STL is based on this approach.

Agenda

- ~~Motivating example~~
- ~~Background~~
- **Speculative Taskloop (STL)**
- Lost-Thread Effect and Solution
- Experimental Evaluation
- Conclusions

Speculative Taskloop

Speculative Taskloop

- Previous work proposed Speculative Taskloop (STL) to have the best of two worlds: (a) the advantages of task-based parallelism over worksharing constructs; and (b) the effectiveness of TLS to parallelize may DOACROSS loops where OpenMP DOALL or DOACROSS techniques fail.

Speculative Taskloop

- Previous work proposed Speculative Taskloop (STL) to have the best of two worlds: (a) the advantages of task-based parallelism over worksharing constructs; and (b) the effectiveness of TLS to parallelize may DOACROSS loops where OpenMP DOALL or DOACROSS techniques fail.
- Another previous work added clauses (`spec_private` and `spec_reduction`) and directives (`tls if_read`, `tls if_write`, etc.) from Speculative Privatization in OpenMP to Speculative Taskloop, and evaluated the results of this parallelization technique.

Speculative Taskloop

- STL speculates data dependences between tasks generated by a `taskloop` construct in non-DOALL loops and executes multiple tasks of loop iterations to exploit task parallelism and to accelerate code execution.

Speculative Taskloop

- STL speculates data dependences between tasks generated by a `taskloop` construct in non-DOALL loops and executes multiple tasks of loop iterations to exploit task parallelism and to accelerate code execution.
- Parallelization of *may* DOACROSS loops using the TLS algorithm on HTM but reusing the `taskloop` mechanism to create OpenMP explicit tasks and to divide iterations among them.

Speculative Taskloop

- STL speculates data dependences between tasks generated by a `taskloop` construct in non-DOALL loops and executes multiple tasks of loop iterations to exploit task parallelism and to accelerate code execution.
- Parallelization of *may* DOACROSS loops using the TLS algorithm on HTM but reusing the `taskloop` mechanism to create OpenMP explicit tasks and to divide iterations among them.
- Explicit tasks instead of threads as units of speculative parallelization.

Speculative Taskloop

- STL speculates data dependences between tasks generated by a `taskloop` construct in non-DOALL loops and executes multiple tasks of loop iterations to exploit task parallelism and to accelerate code execution.
- Parallelization of *may* DOACROSS loops using the TLS algorithm on HTM but reusing the `taskloop` mechanism to create OpenMP explicit tasks and to divide iterations among them.
- Explicit tasks instead of threads as units of speculative parallelization.
- Believe on the task scheduler to distribute tasks onto cores.

`tls` clause for `taskloop`

- `#pragma omp taskloop` **`tls`**`(size)`

  `for-loop`

`tls` clause for `taskloop`

- `#pragma omp taskloop` **`tls`**`(size)`

  `for-loop`

- In `taskloop` consctructs.

`tls` clause for `taskloop`

- `#pragma omp taskloop` **`tls`**`(size)`

  `for-loop`

- In `taskloop` consctructs.
- Enables programmers to parallelize *may* DOACROSS using STL.

`tls` clause for `taskloop`

- `#pragma omp taskloop` **`tls`**`(size)`

  `for-loop`

- In `taskloop` consctructs.
- Enables programmers to parallelize *may* DOACROSS using STL.
- `size` is the number of iterations assigned to each speculative task.

How STL works

- The `parallel` construct creates a team of threads.

How STL works

- The `parallel` construct creates a team of threads.
- A thread encounters `taskloop` which partitions the iterations of the loop into tasks.

How STL works

- The `parallel` construct creates a team of threads.
- A thread encounters `taskloop` which partitions the iterations of the loop into tasks.
- These tasks are scheduled at runtime to be executed speculatively by the team.

How STL works

- The `parallel` construct creates a team of threads.
- A thread encounters `taskloop` which partitions the iterations of the loop into tasks.
- These tasks are scheduled at runtime to be executed speculatively by the team.
- The order of tasks is not specified in `taskloop`.

How STL works

- The `parallel` construct creates a team of threads.
- A thread encounters `taskloop` which partitions the iterations of the loop into tasks.
- These tasks are scheduled at runtime to be executed speculatively by the team.
- The order of tasks is not specified in `taskloop`.
- STL works well with `monotonic` schedule.

Agenda

- ~~Motivating example~~
- ~~Background~~
- ~~Speculative Taskloop (STL)~~
- **Lost-Thread Effect and Solution**
- Experimental Evaluation
- Conclusions

Lost-Thread Effect and Solution

Lost-Thread Effect

- Tasks executing higher iterations could be scheduled before than lower ones (`nonmonotonic`).

Lost-Thread Effect

- Tasks executing higher iterations could be scheduled before than lower ones (`nonmonotonic`).
- Transactions executed by explicit tasks of higher iterations will abort by order inversion.

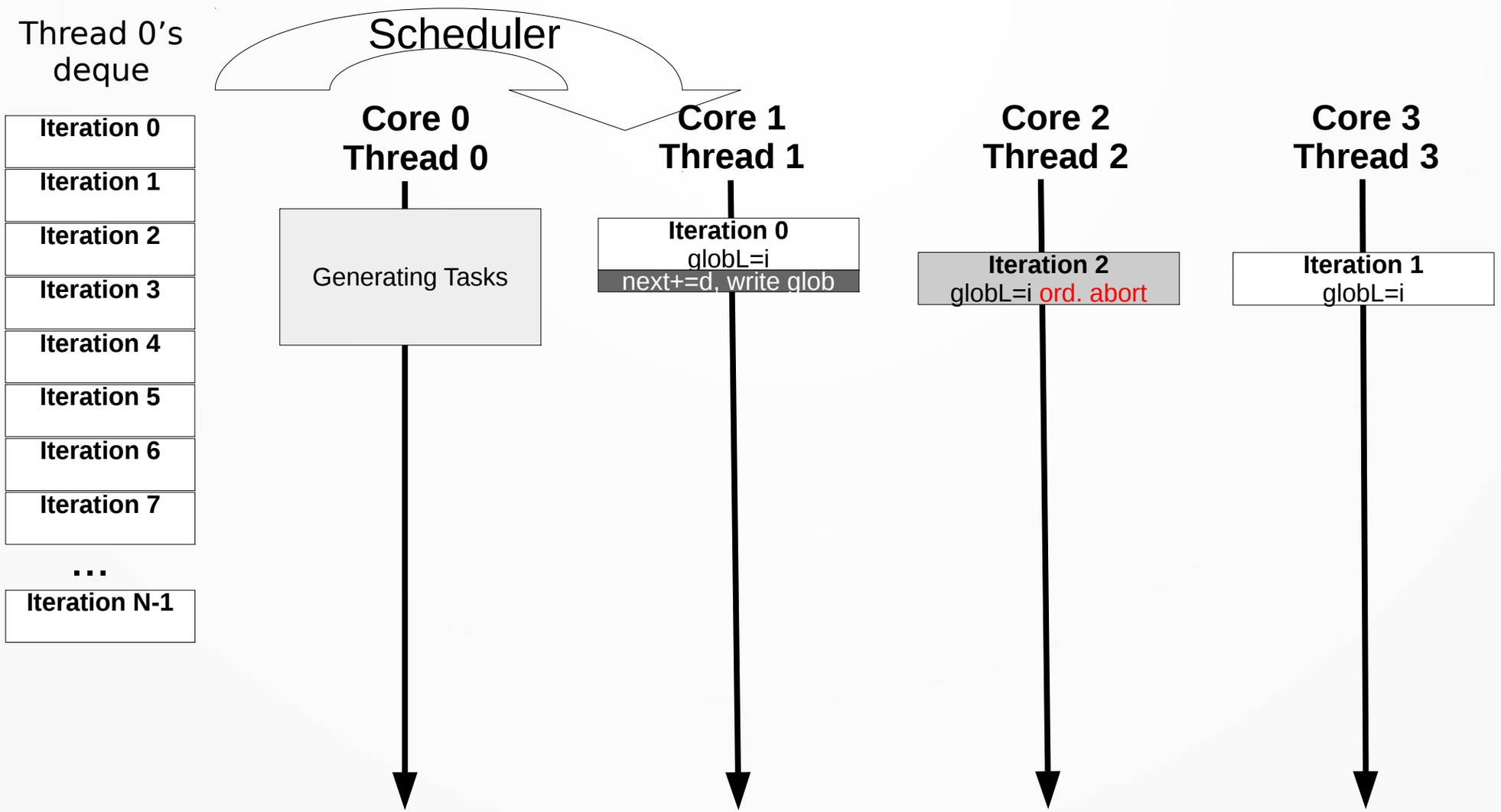
Lost-Thread Effect

- Tasks executing higher iterations could be scheduled before than lower ones (`nonmonotonic`).
- Transactions executed by explicit tasks of higher iterations will abort by order inversion.
- Order inversion is an issue due to the lack of ordered transactions but it is exacerbated in OpenMP tasks due to the Lost-Thread Effect.

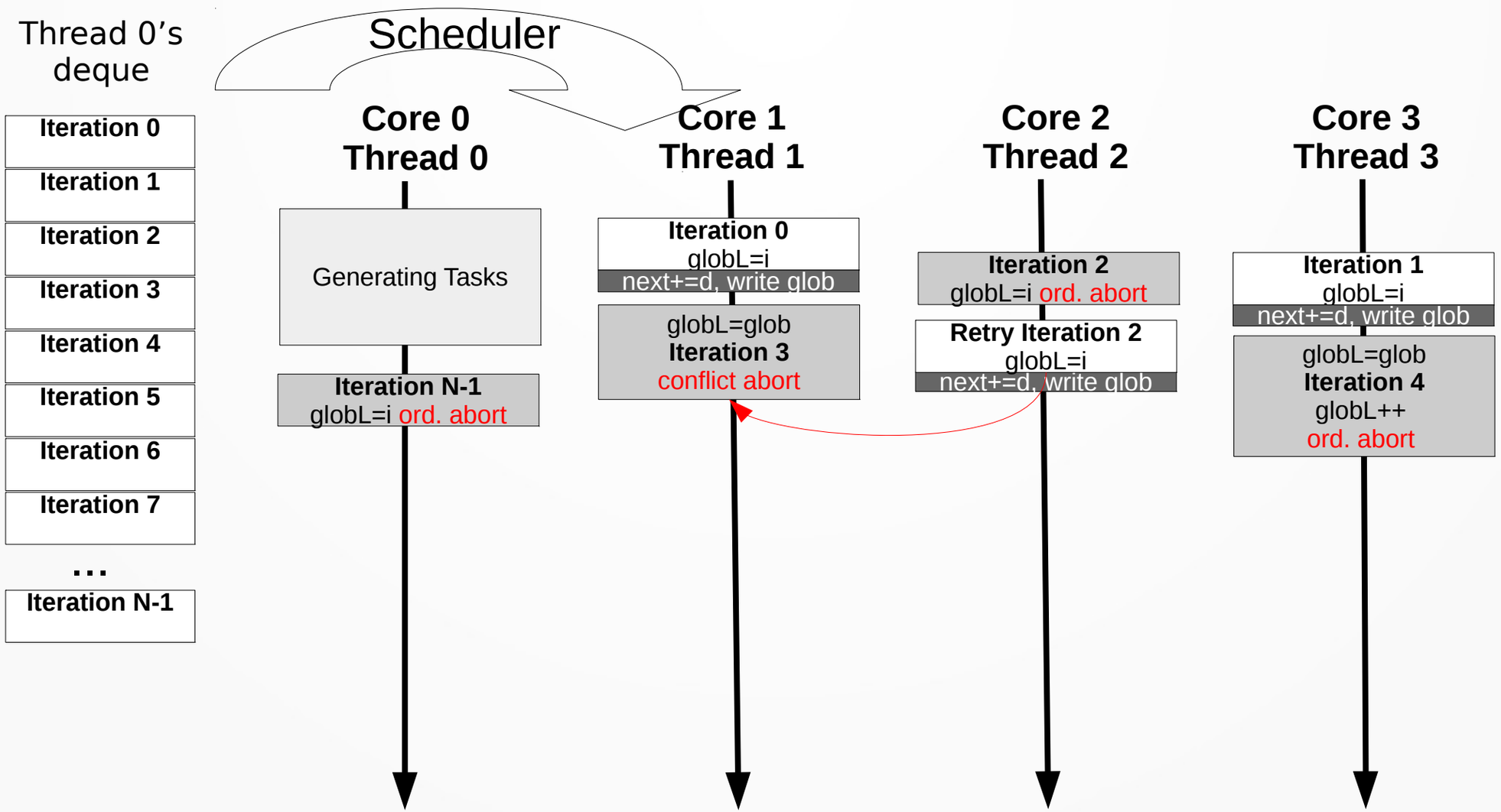
Lost-Thread Effect

- Tasks executing higher iterations could be scheduled before than lower ones (`nonmonotonic`).
- Transactions executed by explicit tasks of higher iterations will abort by order inversion.
- Order inversion is an issue due to the lack of ordered transactions but it is exacerbated in OpenMP tasks due to the Lost-Thread Effect.
- Solutions: (a) Force order of tasks with `priority`?; (b) Implement `monotonic` schedule.

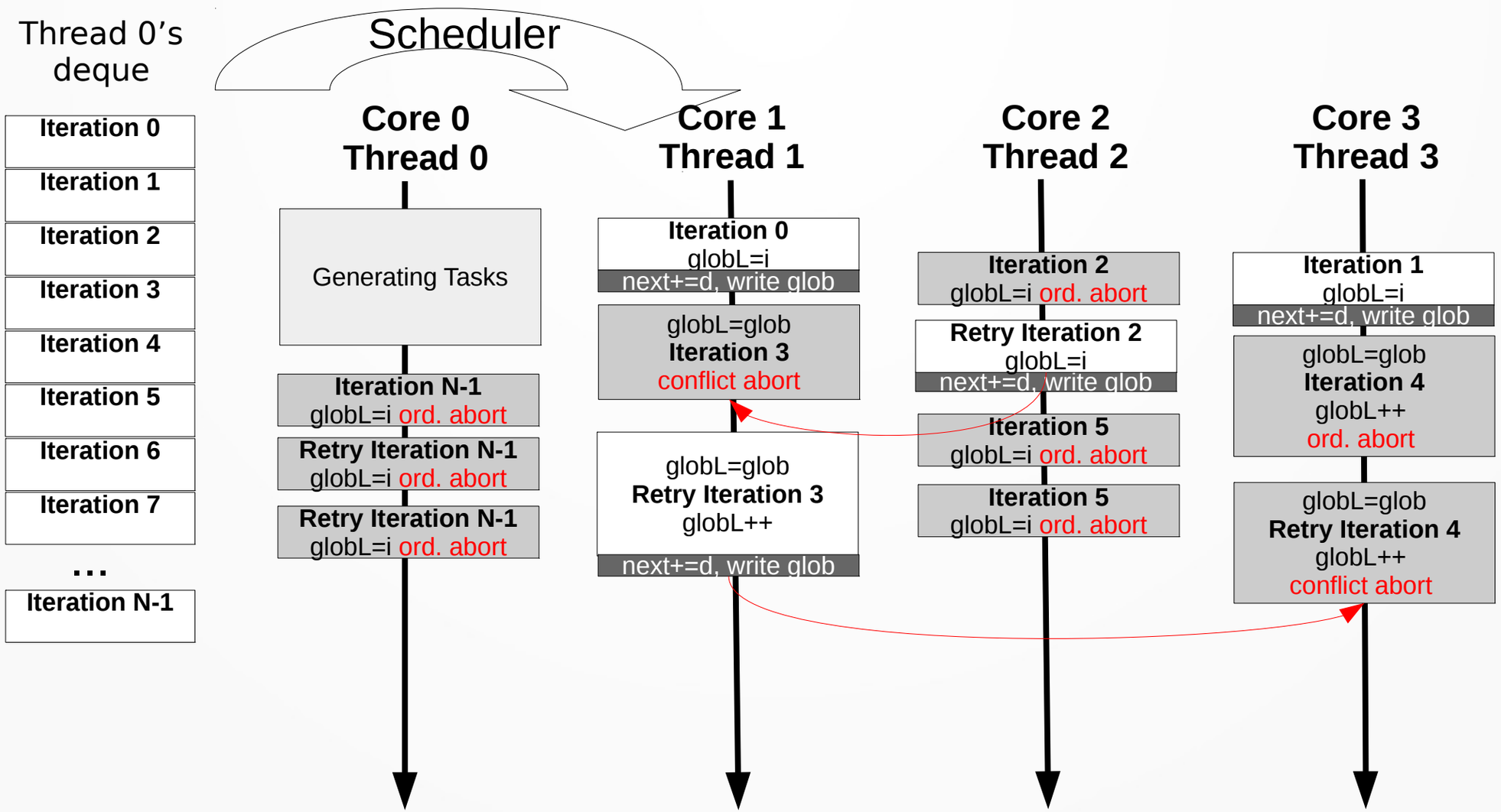
toy2's STL Parallelization



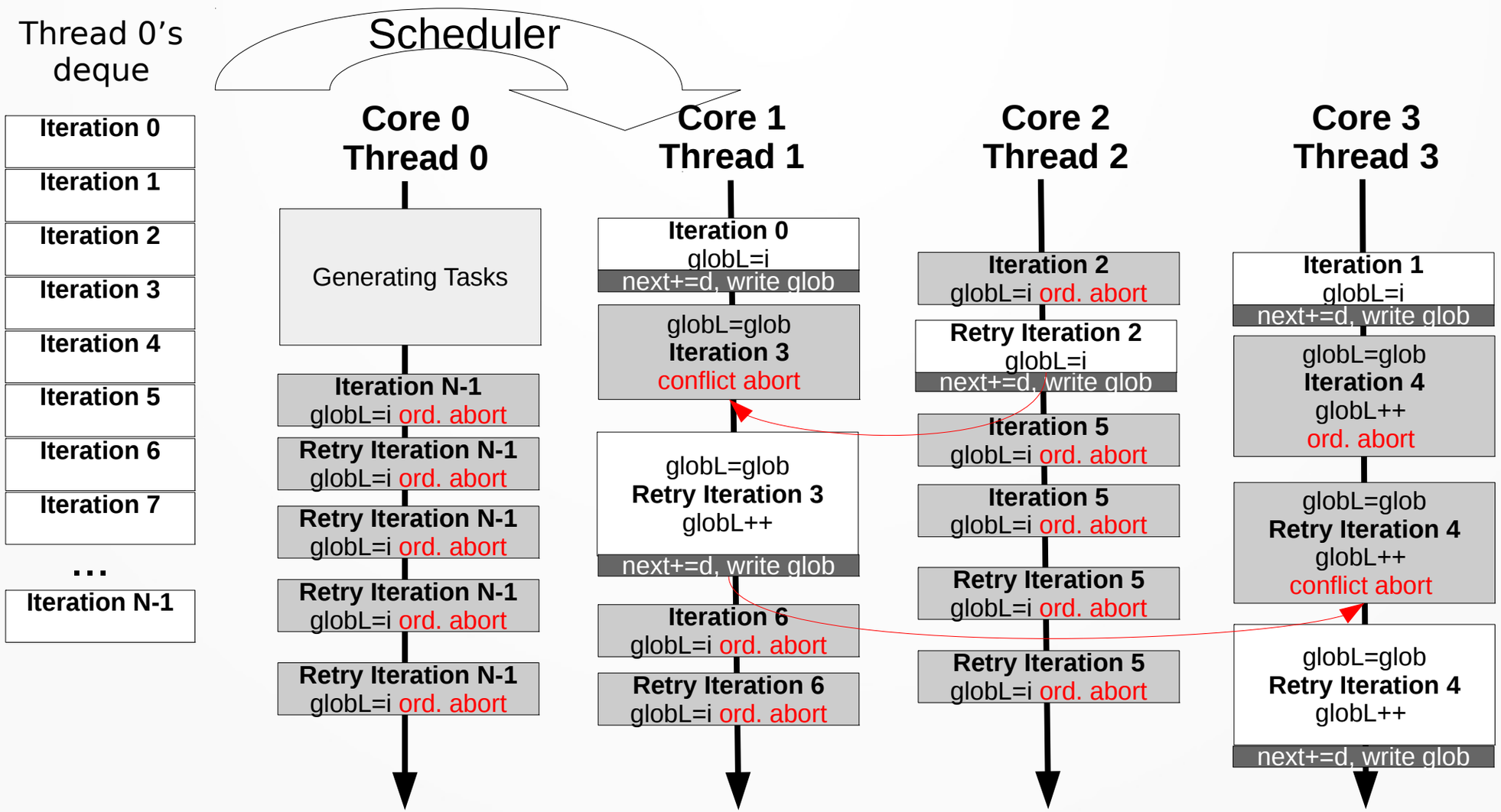
toy2's STL Parallelization



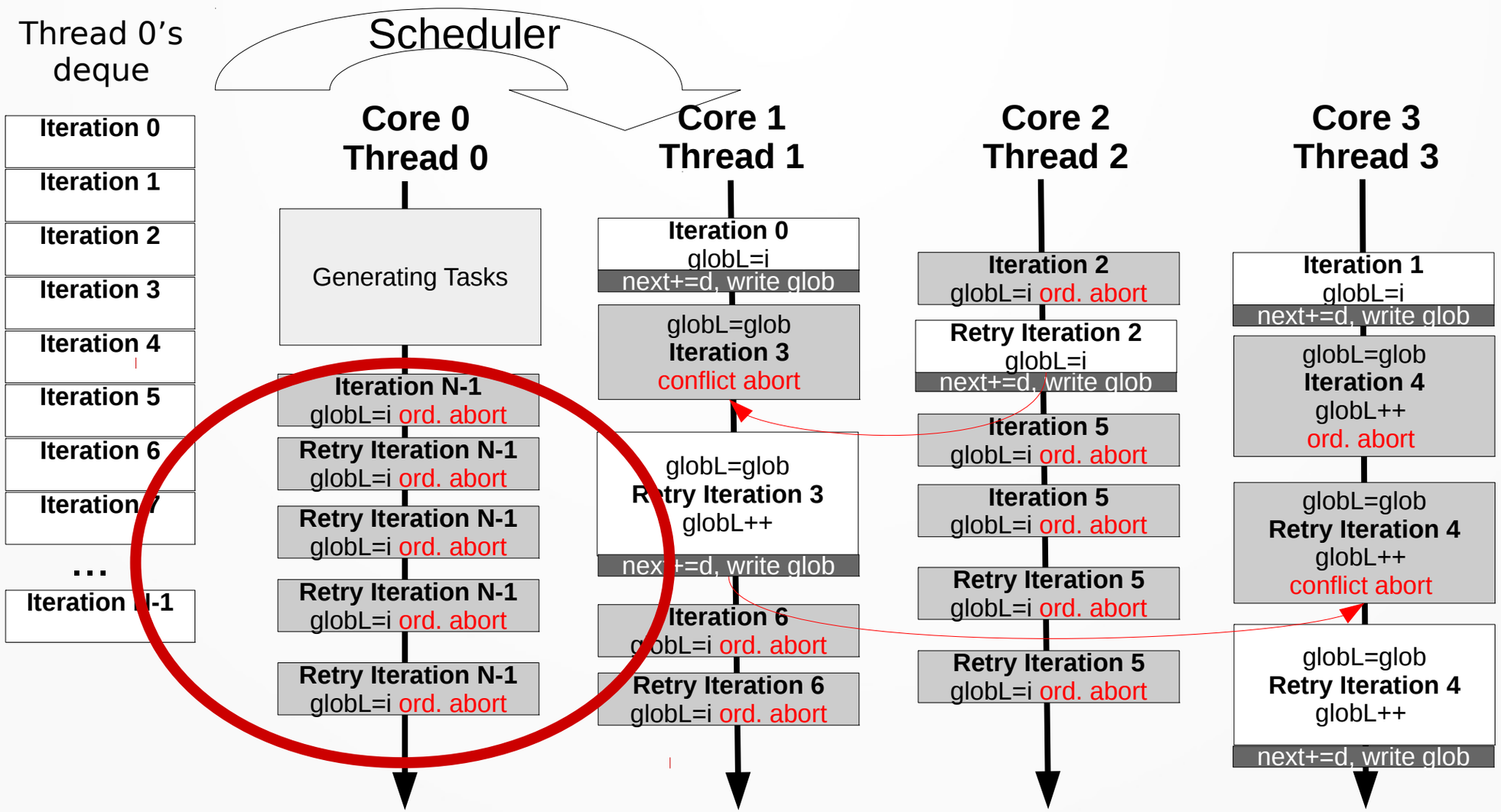
toy2's STL Parallelization



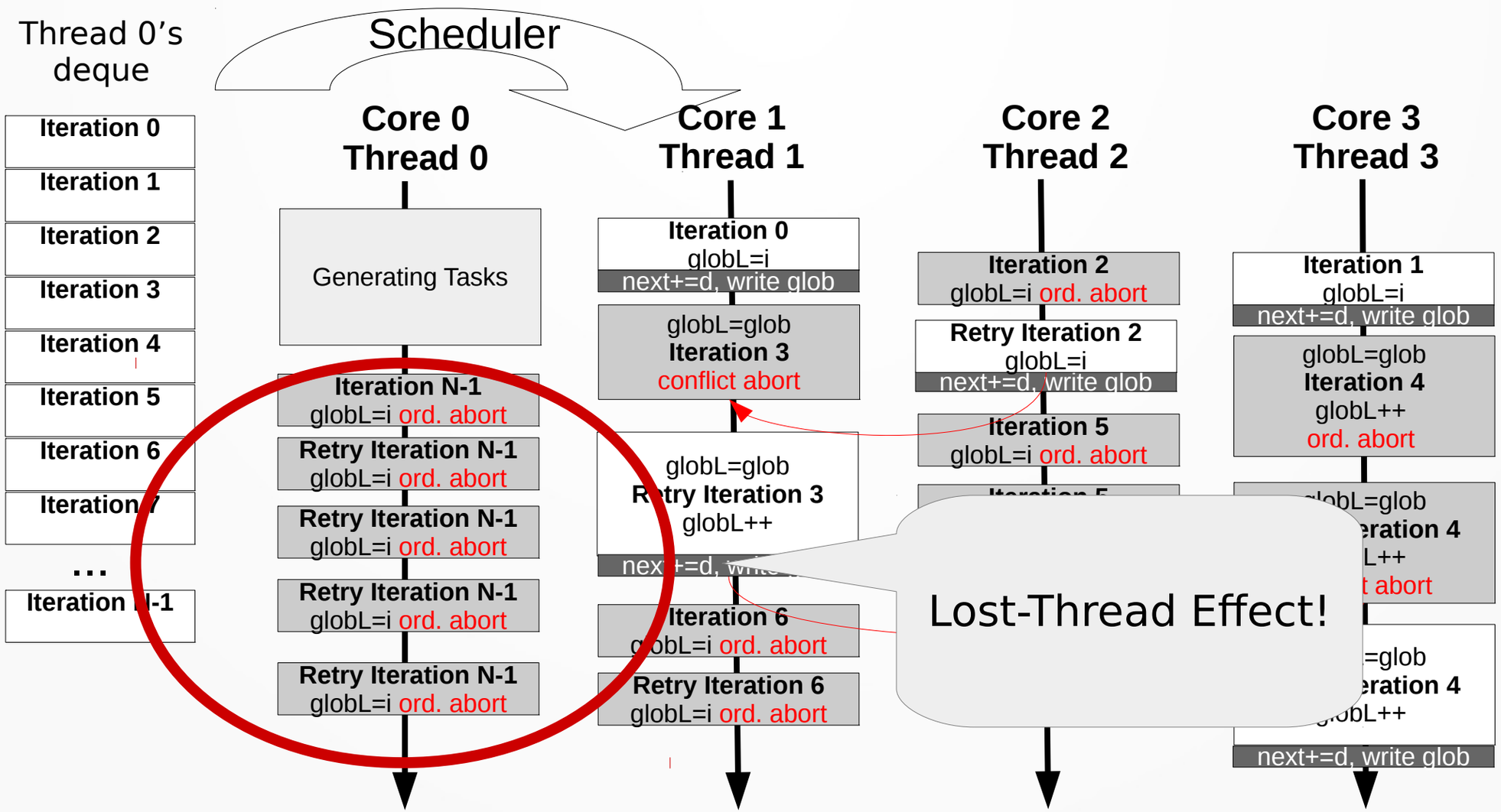
toy2's STL Parallelization



toy2's STL Parallelization



toy2's STL Parallelization



Lost-Thread Effect

- The causes of Lost-Thread Effect are:
  - The generating thread removes tasks from its deque’s tail (previous example).

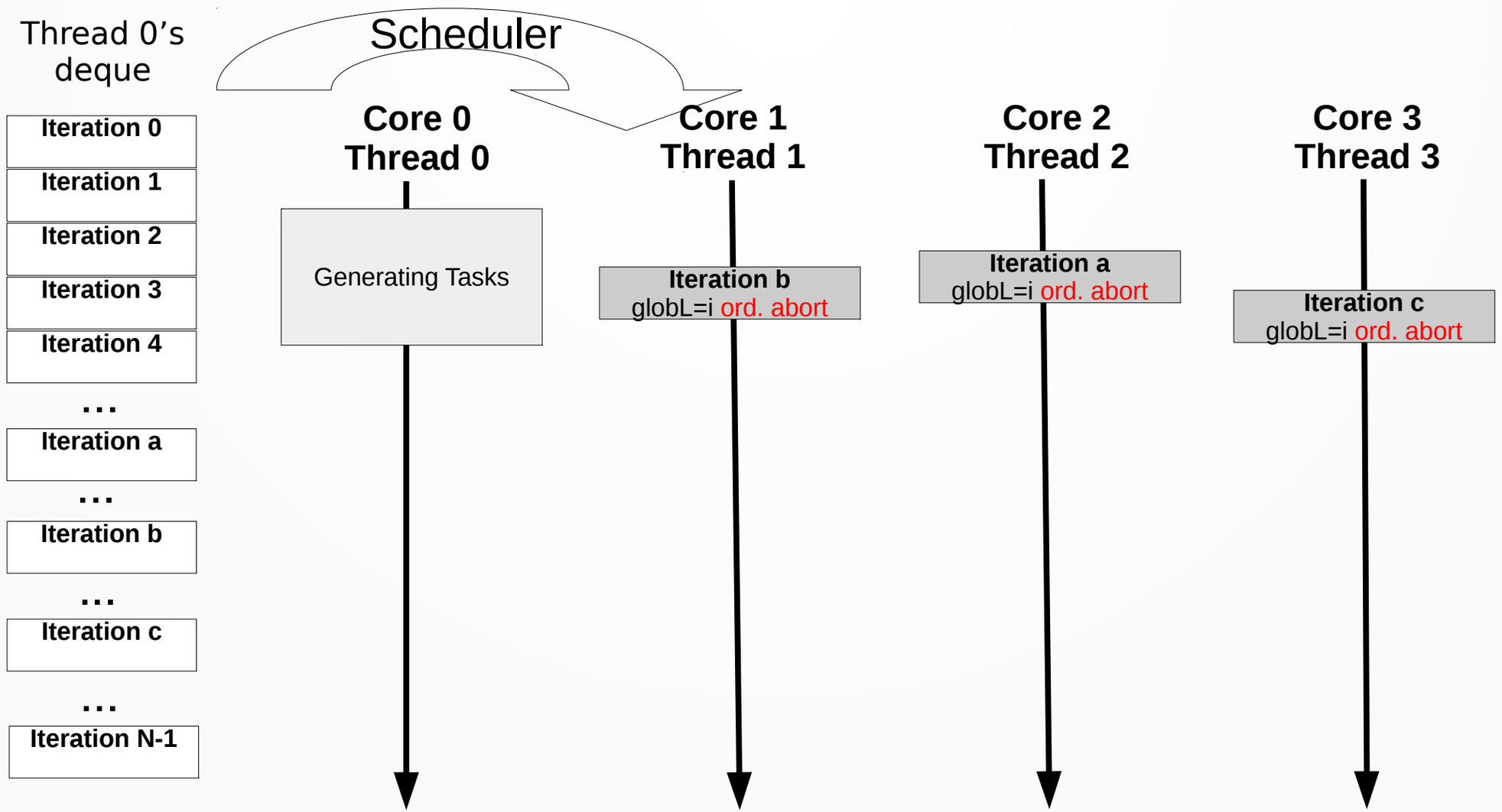
Lost-Thread Effect

- The causes of Lost-Thread Effect are:
  - The generating thread removes tasks from its deque's tail (previous example).
  - The immediate execution of tasks by the generating thread when its deque is full.

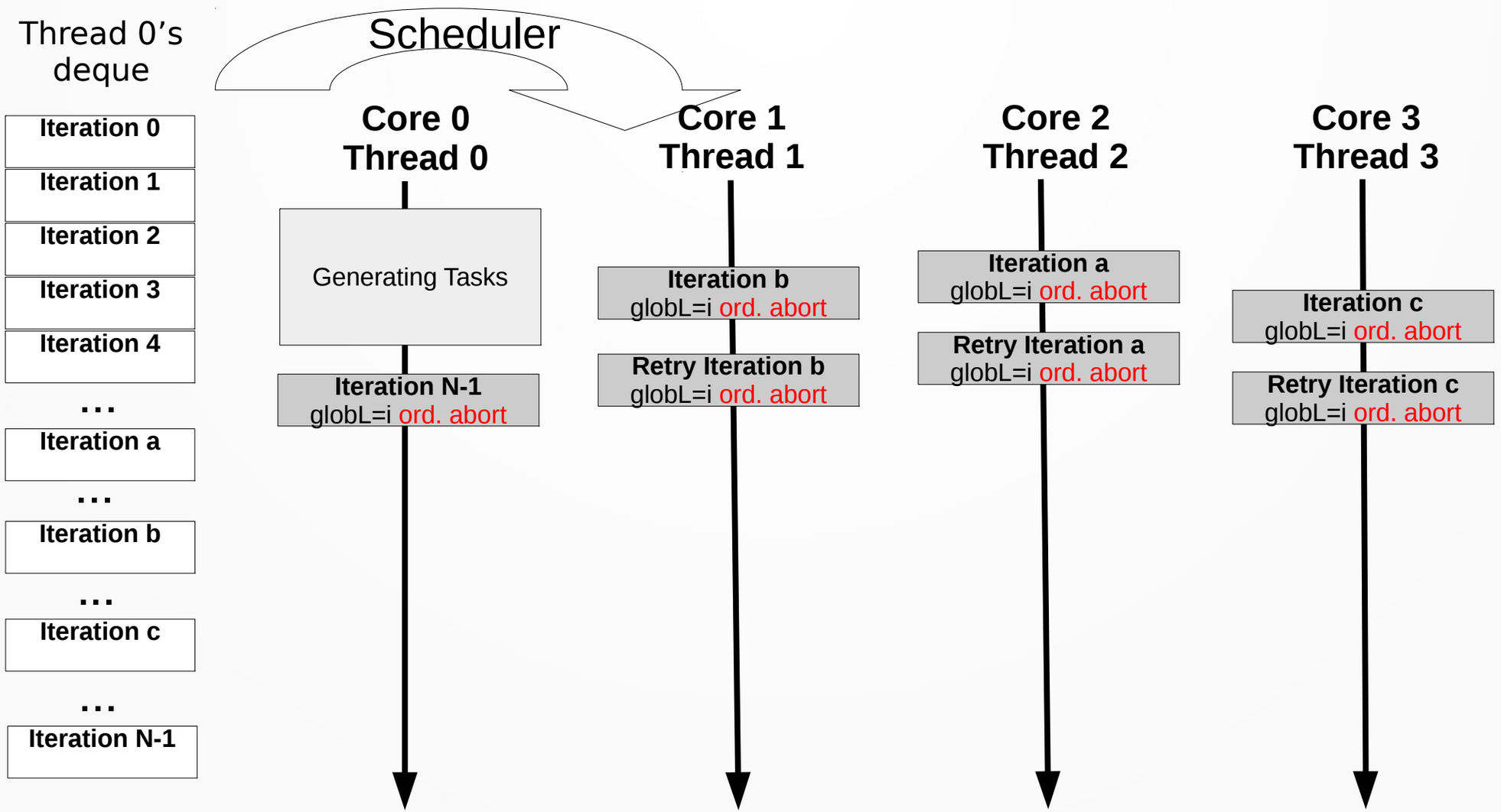
Lost-Thread Effect

- The causes of Lost-Thread Effect are:
  - The generating thread removes tasks from its deque's tail (previous example).
  - The immediate execution of tasks by the generating thread when its deque is full.
  - The recursive partition of iterations to generate tasks in a `nonmonotonic` fashion (`libomp12`).

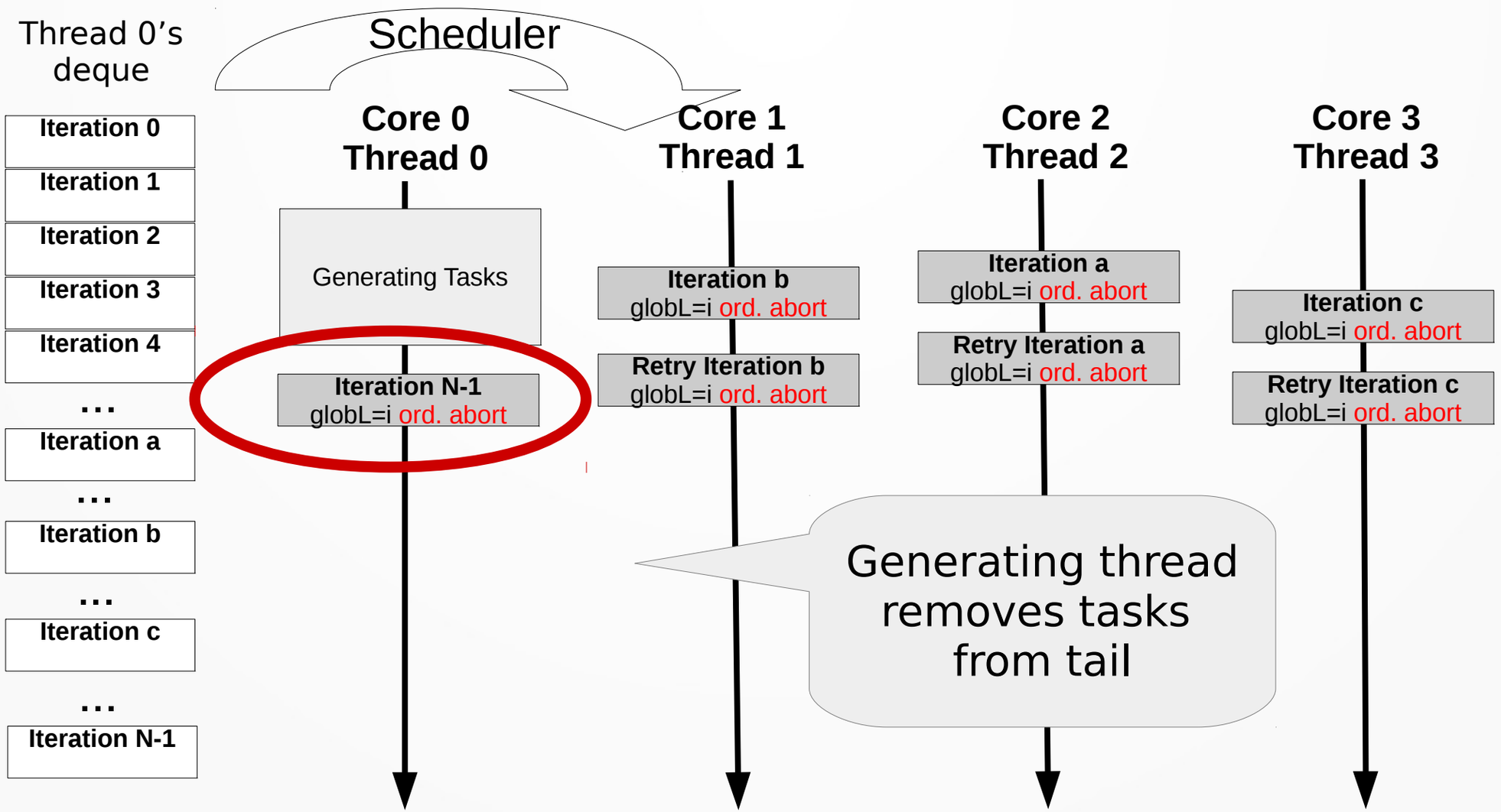
Recursive Partition (nonmonotonic)



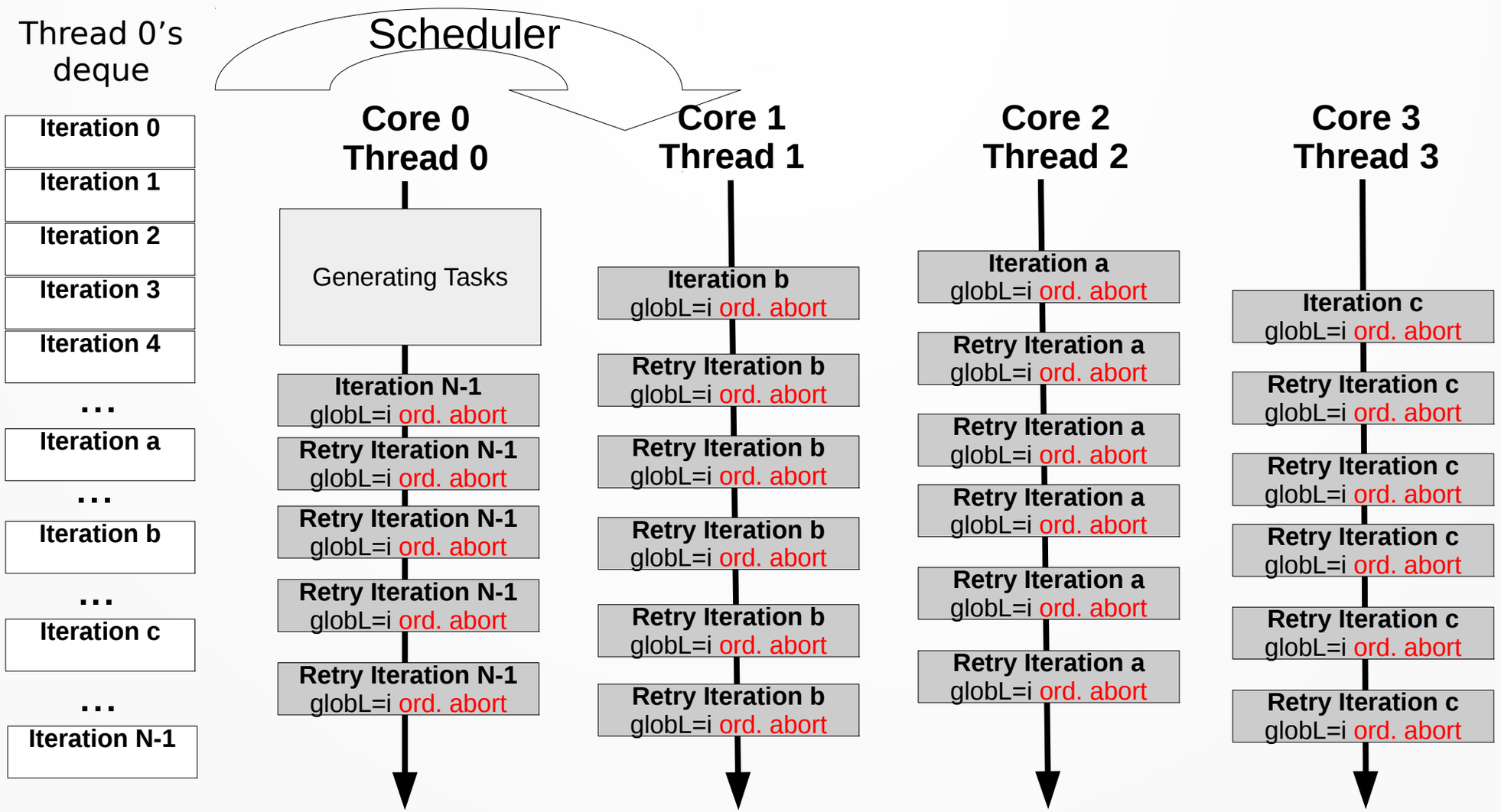
Recursive Partition (nonmonotonic)



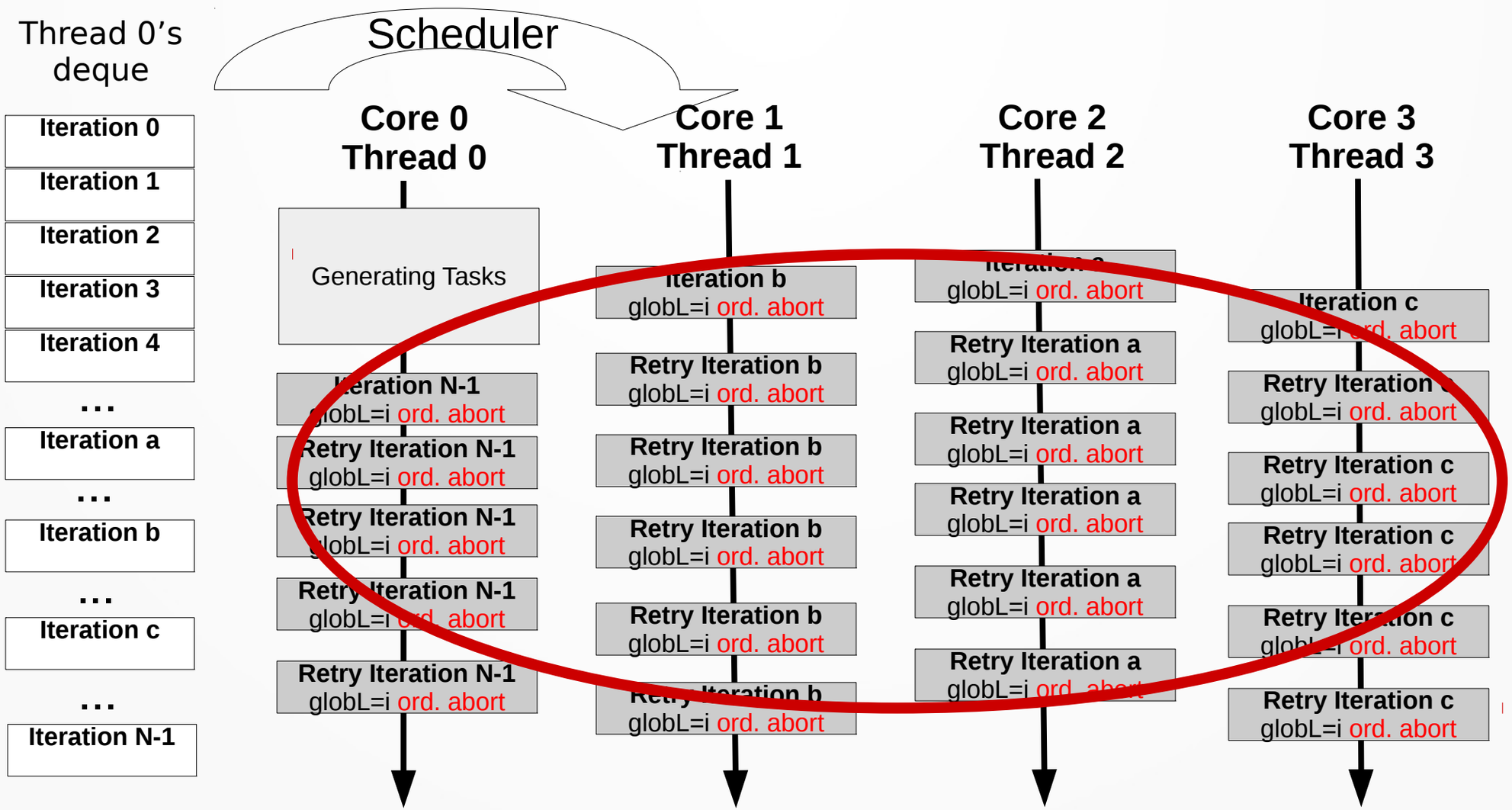
Recursive Partition (nonmonotonic)



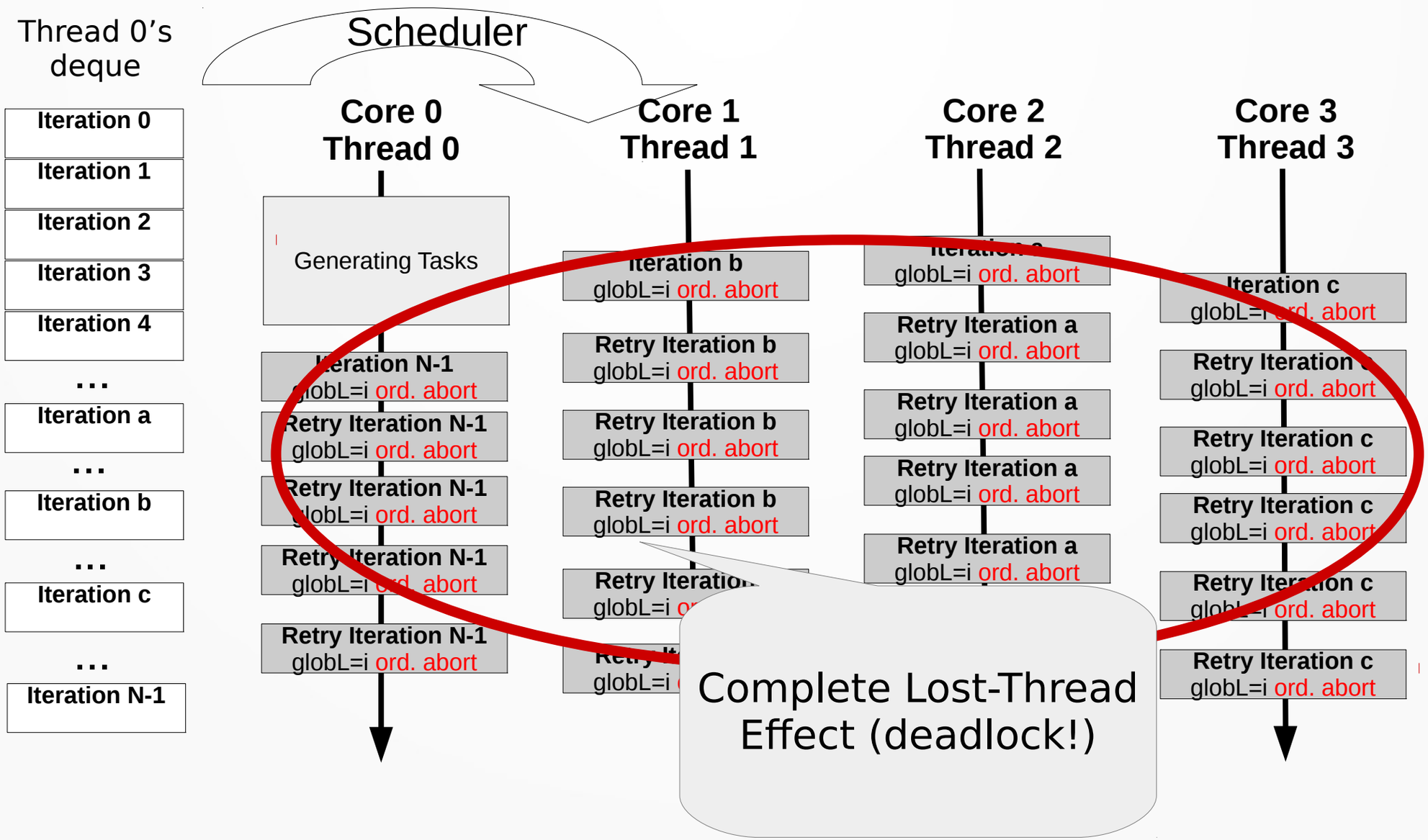
Recursive Partition (nonmonotonic)



Recursive Partition (nonmonotonic)



Recursive Partition (nonmonotonic)



Solution

- Modify the LLVM OpenMP Runtime Library to implement `monotonic` schedule for `taskloop`:

Solution

- Modify the LLVM OpenMP Runtime Library to implement `monotonic` schedule for `taskloop`:
  - Force a linear partition of iterations.

Solution

- Modify the LLVM OpenMP Runtime Library to implement `monotonic` schedule for `taskloop`:
  - Force a linear partition of iterations.
  - Prevent a task from being executed immediately if the deque is full by reallocating it with an increased size.
  - Increase the initial deque size.

Solution

- Modify the LLVM OpenMP Runtime Library to implement `monotonic` schedule for `taskloop`:
  - Force a linear partition of iterations.
  - Prevent a task from being executed immediately if the deque is full by reallocating it with an increased size.
  - Increase the initial deque size.
  - Ensure that the generating thread also removes its own tasks from its deque’s head.

Agenda

- ~~Motivating example~~
- ~~Background~~
- ~~Speculative Taskloop (STL)~~
- ~~Lost-Thread Effect and Solution~~
- **Experimental Evaluation**
- Conclusions

Experimental Evaluation

Experimental Evaluation

- The performance assessment in this work reports speed-ups and abort/commit ratios (transaction outcome) for the STL (Speculative Taskloop) and `ordered` parallelizations of *may* DOACROSS loops.

Experimental Evaluation

- The performance assessment in this work reports speed-ups and abort/commit ratios (transaction outcome) for the STL (Speculative Taskloop) and `ordered` parallelizations of *may* DOACROSS loops.
- We used two versions of the LLVM OpenMP Runtime: `libomp2016` and `libomp12`.

Experimental Evaluation

- The performance assessment in this work reports speed-ups and abort/commit ratios (transaction outcome) for the STL (Speculative Taskloop) and `ordered` parallelizations of *may* DOACROSS loops.
- We used two versions of the LLVM OpenMP Runtime: `libomp2016` and `libomp12`.
- Both versions in two flavors: `monotonic` (`stl-lx-mon`) and `nonmonotonic` (`stl-lx-nm`).

Setup and Environment

- Quadcore Intel Skylake (TSX-NI)

Setup and Environment

- Quadcore Intel Skylake (TSX-NI)
- 5 loops from cBench and 1 loop from SPEC

Setup and Environment

- Quadcore Intel Skylake (TSX-NI)
- 5 loops from cBench and 1 loop from SPEC
- 5 benchmarks: `susan_c`, `susan_e`, `susan_s`, `bitcount`, and `429.mcf`

Setup and Environment

- Quadcore Intel Skylake (TSX-NI)
- 5 loops from cBench and 1 loop from SPEC
- 5 benchmarks: `susan_c`, `susan_e`, `susan_s`, `bitcount`, and `429.mcf`
- Baseline: serial execution of the same benchmark program compiled at the same optimization level

Setup and Environment

- Quadcore Intel Skylake (TSX-NI)
- 5 loops from cBench and 1 loop from SPEC
- 5 benchmarks: `susan_c`, `susan_e`, `susan_s`, `bitcount`, and `429.mcf`
- Baseline: serial execution of the same benchmark program compiled at the same optimization level
- Default input for each benchmark and reference input for `mcf`

Setup and Environment

Loop ID	Loop Information			
	Benchmark	Location	%*Cov*	Invocations
A	automotive_bitcount	bitcnts.c,65	100%	560
E	automotive_susan_s	susan.c,725	100%	22050
H	automotive_susan_e	susan.c,1117	18%	374
I	automotive_susan_e	susan.c,1056	56%	374
V	automotive_susan_c	susan.c,1614	7%	782
mcf	429.mcf	pbeampp.c,165	40%	21854886

Results

`stl-lx-mon`

- The performance of the STL parallelization of the benchmarks improves with the implementation of `monotonic` scheduling in the two OpenMP runtime libraries.

Results

`stl-lx-mon`

- The performance of the STL parallelization of the benchmarks improves with the implementation of `monotonic` scheduling in the two OpenMP runtime libraries.
- This is more noticeable with two threads since the Lost-Thread Effect causes a large number of aborts due to order inversion (almost 100% of the transactions started with 2 threads).

Results

`stl-lx-mon`

- The performance of the STL parallelization of the benchmarks improves with the implementation of `monotonic` scheduling in the two OpenMP runtime libraries.
- This is more noticeable with two threads since the Lost-Thread Effect causes a large number of aborts due to order inversion (almost 100% of the transactions started with 2 threads).
- It is possible to achieve speed-ups using STL and the modified libomp12 (`stl-l12-mon`) in loops that did not even finish (deadlock) with `nonmonotonic` scheduling (`stl-l12-nm`).

Results

`stl-lx-mon`

- The performance of the STL parallelization of the benchmarks improves with the implementation of `monotonic` scheduling in the two OpenMP runtime libraries.
- This is more noticeable with two threads since the Lost-Thread Effect causes a large number of aborts due to order inversion (almost 100% of the transactions started with 2 threads).
- It is possible to achieve speed-ups using STL and the modified libomp12 (`stl-l12-mon`) in loops that did not even finish (deadlock) with `nonmonotonic` scheduling (`stl-l12-nm`).
- Reduction of order-inversion aborts with respect to `nonmonotonic`.

Results

`stl-l12-mm`

- A complete Lost-Thread Effect is generated causing no thread to progress due to the recursive partition of the iteration space.

Results

stl-l12-mm

- A complete Lost-Thread Effect is generated causing no thread to progress due to the recursive partition of the iteration space.
- However, `loopE` and `mcf` complete their execution because the number of tasks generated is less than `num_task_min` invoking to the `kmp_taskloop_linear` function.

Results

stl-l12-mm

- A complete Lost-Thread Effect is generated causing no thread to progress due to the recursive partition of the iteration space.
- However, `loopE` and `mcf` complete their execution because the number of tasks generated is less than `num_task_min` invoking to the `kmp_taskloop_linear` function.

stl-lx-mm

- The `mcf` hottest loop is a particular case because, with the `S_SIZE` used, only four tasks are generated. The Lost-Thread Effect does not occur because the other three threads will execute the three first tasks and will arrive quickly when the generating thread is executing the fourth task.

Results

- Only for this particular reason, the `nonmonotonic` version of this loop offers better speed-ups with four threads than the `monotonic` version.

Results

- Only for this particular reason, the `nonmonotonic` version of this loop offers better speed-ups with four threads than the `monotonic` version.

  `ordered`

- For the parallelization with `ordered`, `schedule (auto)` clause is used, otherwise the performance is even worse (`loopA` had slowdowns up to 0.01× using 4 threads).

Results

- Only for this particular reason, the `nonmonotonic` version of this loop offers better speed-ups with four threads than the `monotonic` version.

  `ordered`

- For the parallelization with `ordered`, `schedule (auto)` clause is used, otherwise the performance is even worse (`loopA` had slowdowns up to 0.01× using 4 threads).
- In the evaluated loops, the performance is poor because either it is not possible to distinguish parallel/serial components or the work of the parallel components is not significant.

Agenda

- ~~Motivating example~~
- ~~Background~~
- ~~Speculative Taskloop (STL)~~
- ~~Lost-Thread Effect and Solution~~
- ~~Experimental Evaluation~~
- **Conclusions**

Conclusions

Conclusions

- This paper confirms our claim in previous work about the performance detriment of the Lost-Thread Effect and shows that the implementation of `monotonic` scheduling improves the performance of Speculative taskloop.

Conclusions

- This paper confirms our claim in previous work about the performance detriment of the Lost-Thread Effect and shows that the implementation of `monotonic` scheduling improves the performance of Speculative taskloop.
- We present an evaluation with two different versions of the OpenMP runtime, both optimized for STL, that reveals that, for certain loops, slowdowns or infinite slowdowns (deadlocks), using the original OpenMP runtime, can be transformed in speed-ups by applying `monotonic` scheduling.

Thanks!