Performance Comparison of Speculative Taskloop and OpenMP-`for`-Loop Thread-Level Speculation on Hardware Transactional Memory

Juan Salamanca
São Paulo State University (Unesp),Brazil
ISPDC 2022

Agenda

- Background
- Performance Comparison
- Experimental Evaluation
- Conclusions

Agenda

- **Background**
- Performance Comparison
- Experimental Evaluation
- Conclusions

Background

Agenda

- **Background**
  - **DOALL loops**
  - DOACROSS loops
  - *May* DOACROSS loops
- Performance Comparison
- Experimental Evaluation
- Conclusions

DOALL loops

- Example

```
for( i = 0 ; i < N; i += 1){
   A[i]=b*i;
}
```

DOALL loops

- How can we parallelize this loop?

```
for( i = 0 ; i < N; i += 1){
   A[i]=b*i;
}
```

DOALL loops

- How can we parallelize this loop?

```
for( i = 0 ; i < N; i += 1){
   A[i]=b*i;
}
```

Using DOALL techniques: `parallel-for` and `taskloop` in OpenMP

DOALL loops

- parallel-for

```
#pragma omp parallel for schedule(...)
for( i = 0 ; i < N; i += 1){
   A[i]=b*i;
}
```

DOALL loops

- parallel-for

```
#pragma omp parallel for schedule(...)
for( i = 0 ; i < N; i += 1){
   A[i]=b*i;
}
```

schedule could be: static, dynamic, guided, or auto

DOALL loops

- taskloop

```
#pragma omp parallel
#pramga omp [single|master]
#pragma omp taskloop [grainsize(...)|num_tasks(...)]
for( i = 0 ; i < N; i += 1){
   A[i]=b*i;
}
```

DOALL loops

- taskloop

```
#pragma omp parallel
#pramga omp [single|master]
#pragma omp taskloop [grainsize(...)|num_tasks(...)]
for( i = 0 ; i < N; i += 1){
   A[i]=b*i;
}
```

Scheduling at runtime.

Agenda

- **Background**
  - ~~DOALL loops~~
  - **DOACROSS loops**
  - *May* DOACROSS loops
- Performance Comparison
- Experimental Evaluation
- Conclusions

DOACROSS loops

- Example

```
for( i = 0 ; i < N; i += 1){
   A[i]=A[i-1]*i;
   ...
}
```

DOACROSS loops

- Example

```
for( i = 0 ; i < N; i += 1){
   A[i]=A[i-1]*i;
   ...
}
```

DOACROSS loops

- How can we parallelize this loop?

```
for( i = 0 ; i < N; i += 1){
   A[i]=A[i-1]*i;
   ...
}
```

Using DOACROSS techniques: HELIX, DSWP, the `ordered` construct in OpenMP, etc.

DOACROSS loops

- `parallel-for` and `ordered`

```
#pragma omp parallel for ordered(1)
for( i = 0 ; i < N; i += 1){
   #pragma omp ordered depend(sink:i-1)
   A[i]=A[i-1]*i;
   #pragma omp ordered depend(source)
   ...
}
```

DOACROSS loops

- `parallel-for` and `ordered`

```
#pragma omp parallel for ordered(1)
for( i = 0 ; i < N; i += 1){
   #pragma omp ordered depend(sink:i-1)
   A[i]=A[i-1]*i;
   #pragma omp ordered depend(source)
   ...
}
```

It serializes the region between `sink` and `source`

Agenda

- **Background**
  - ~~DOALL loops~~
  - ~~DOACROSS loops~~
  - ***May* DOACROSS loops**
- Performance Comparison
- Experimental Evaluation
- Conclusions

May DOACROSS loops

- Example (`susan_corners`'s 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++;
    }
    …
}
```

May DOACROSS loops

- 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++;
    }
    …
}
```

May DOACROSS loops

- 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++;
    }
    …
}
```

If this condition is **false** for all iterations, the loop is DOALL at runtime.

May DOACROSS loops

- 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++;
    }
    …
}
```

Compilers are conservative, they consider it as DOACROSS.

May DOACROSS loops

- Using `parallel-for` and `ordered`

```
#pragma omp parallel for ordered(2)
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    #pragma omp ordered depend(sink:i-1)
    if (x>0 && /*compare x*/){
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
    }
    #pragma omp ordered depend(source)
}
```

Compilers are conservative, they consider it as DOACROSS.

May DOACROSS loops

- Using `parallel-for` and `ordered`

```
#pragma omp parallel for ordered(2)
for(i=5 ; i < y_size-5; i++){
    …
    x=r[i][j];
    #pragma omp ordered depend(sink:i-1)
    if (x>0 && /*compare x*/){
        corner_list[n].info=0;
        corner_list[n].x=j;
        …
        n++;
    }
    #pragma omp ordered depend(source)
}
```

Poor performance!
Slowdowns!

May DOACROSS loops

- Can we improve this?

```
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++;
    }
    …
}
```

May DOACROSS loops

- Can we improve this?

```
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++;
    }
    …
}
```

Yes, using **Thread-Level Speculation (TLS)**

May DOACROSS loops

- Using TLS in two flavors: (a)FOR-TLS[TPDS18]; (b) Speculative Taskloop(STL)[IWOMP19&21,HPCS20].

```
#pragma omp [parallel for | 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)
    }
}
```

May DOACROSS loops

- Using TLS in two flavors: (a)FOR-TLS[TPDS18]; (b) Speculative Taskloop(STL)[IWOMP19&21,HPCS20].

```
#pragma omp [parallel for | 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)
    }
}
```

Speed-ups of **1.2x** with FOR-TLS and of **1.33x** with STL using 4 cores

Agenda

- ~~Background~~
- **Performance Comparison**
- Experimental Evaluation
- Conclusions

Performance Comparison

Performance Comparison

- Without Speculation?

Performance Comparison

- Without Speculation?
  - Scheduling of DOALL loop parallelization can be:

Performance Comparison

- Without Speculation?
  - Scheduling of DOALL loop parallelization can be:
    - **Static** (in `parallel-for`) which favors **balanced** loops (regular loops, with static dependencies, uniform data distribution, etc.)

Performance Comparison

- Without Speculation?
  - Scheduling of DOALL loop parallelization can be:
    - **Static** (in `parallel-for`) which favors **balanced** loops (regular loops, with static dependencies, uniform data distribution, etc.)
    - **Dynamic** (in `parallel-for` and `taskloop`) which favors loops that have load **imbalance** (irregular, with dynamic dependencies, non-uniform distribution, with many conditionals, etc.)

Performance Comparison

- With Speculation?

Performance Comparison

- With Speculation?
  - Scheduling of *may* DOACROSS loop parallelization can be:
    - **Static** (in FOR-TLS) which would favor **balanced** loops
    - **Dynamic** (in STL) which would favor loops that have load **imbalance.**

Performance Comparison

- With Speculation?
  - Scheduling of *may* DOACROSS loop parallelization can be:
    - **Static** (in FOR-TLS) which would favor **balanced** loops
    - **Dynamic** (in STL) which would favor loops that have load **imbalance.**
  - We study loop features to find out why a loop is better using FOR-TLS or STL.

Performance Comparison

- With Speculation?
  - Scheduling of *may* DOACROSS loop parallelization can be:
    - **Static** (in FOR-TLS) which would favor **balanced** loops
    - **Dynamic** (in STL) which would favor loops that have load **imbalance.**
  - We study loop features to find out why a loop is better using FOR-TLS or STL.
  - The loop features are: *Tloop*, the regularity of the loop, the function calls that exist inside `if` statements, the transaction duration, the binomial transaction duration and loop regularity, %*lc*, the average iteration size, and S_SIZE.

Performance Comparison

- Example (`susan_smoothing`'s loop)

```
for(j=mask_size;j<x_size-mask_size;j++){ //loopE
   area = 0;
   total = 0;
   ...
   centre = in[i*x_size+j];
   ...// calulating area and total
   tmp = area-10000;
   if (tmp==0) *out++=median(in,i,j,x_size);
   else *out++=((total-(centre*10000))/tmp);
}
```

Performance Comparison

- Example (`susan_smoothing`'s loop)

```
for(j=mask_size;j<x_size-mask_size;j++){ //loopE
    area = 0;
    total = 0;
    ...
    centre = in[i*x_size+j];
    ...// calulating area and total
    tmp = area-10000;
    if (tmp==0) *out++=median(in,i,j,x_size);
    else *out++=((total-(centre*10000))/tmp);
}
```

Performance Comparison

- Example (`susan_smoothing`'s loop)

```
for(j=mask_size;j<x_size-mask_size;j++){ //loopE
    area = 0;
    total = 0;
    ...
    centre = in[i*x_size+j];
    ...// calulating area and total
    tmp = area-10000;
    if (tmp==0) *out++=median(in,i,j,x_size
    else *out++=((total-(centre*10000))/tmp);
}
```

Irregular!

Performance Comparison

- Example (`susan_smoothing`'s loop)

```
for(j=mask_size;j<x_size-mask_size;j++){ //loopE
    area = 0;
    total = 0;
    ...
    centre = in[i*x_size+j];
    ...// calulating area and total
    tmp = area-10000;
    if (tmp==0) *out++=median(in,i,j,x_size);
    else *out++=((total-(centre*10000))/tmp);
}
```

So, will it have better performance with STL?

Agenda

- ~~Background~~
- ~~Performance Comparison~~
- **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, FOR-TLS 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, FOR-TLS and `ordered` parallelizations of *may* DOACROSS loops.
- We used the LLVM `libomp12` OpenMP Runtime.

Experimental Evaluation

- The performance assessment in this work reports speed-ups and abort/commit ratios (transaction outcome) for the STL, FOR-TLS and `ordered` parallelizations of *may* DOACROSS loops.
- We used the LLVM `libomp12` OpenMP Runtime.
- We performed the experimental evaluation on the OpenMP Runtime Library using the modified version to allow monotonic scheduling.

Setup and Environment

- Quadcore Intel Skylake (TSX-NI)

Setup and Environment

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

Setup and Environment

- Quadcore Intel Skylake (TSX-NI)
- 9 loops from cBench and 1 loop from SPEC
- 5 benchmarks: `susan_corners` (image recognition package, recognizes corners of MRI of the brain), `susan_edges` (recognizes edges of MRI), `susan_smoothing` (smooths an image), `bitcount` (tests the bit manipulation abilities of a processor), and `429.mcf` (single-depot vehicle scheduling)

Setup and Environment

- Quadcore Intel Skylake (TSX-NI)
- 9 loops from cBench and 1 loop from SPEC
- 5 benchmarks: `susan_corners`, `susan_edges`, `susan_smoothing`, `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)
- 9 loops from cBench and 1 loop from SPEC
- 5 benchmarks: `susan_corners`, `susan_edges`, `susan_smoothing`, `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	Benchmark	Location	Invocations
A	automotive_bitcount	bitcnts.c,65	560
B	automotive_susan_c	susan.c,1458	344080
C	automotive_susan_e	susan.c,1118	165308
D	automotive_susan_e	susan.c,1057	166056
E	automotive_susan_s	susan.c,725	22050
H	automotive_susan_e	susan.c,1117	374
I	automotive_susan_e	susan.c,1056	374
J	automotive_susan_s	susan.c,723	49
V	automotive_susan_c	susan.c,1614	782
mcf	429.mcf	pbeampp.c,165	21854886

Results

It is necessary to study some characteristics:

- *Tloop*, because if it is very short, the overhead of using transactions, code transformations, and the runtime scheduler (only for STL) will exceed the parallelization gain.

Results

It is necessary to study some characteristics:

- *Tloop*, because if it is very short, the overhead of using transactions, code transformations, and the runtime scheduler (only for STL) will exceed the parallelization gain.
- The regularity of the loop in terms of the number of `if` statements inside the loop body, if a loop is irregular (more `if` statements) it will have more aborts due to order inversion.

Results

It is necessary to study some characteristics:

- *Tloop*, because if it is very short, the overhead of using transactions, code transformations, and the runtime scheduler (only for STL) will exceed the parallelization gain.
- The regularity of the loop in terms of the number of `if` statements inside the loop body, if a loop is irregular (more `if` statements) it will have more aborts due to order inversion.
- The different function calls that exist inside the `if` statements in the loop body that make the execution time of each iteration significantly variable (irregular).

Results

- The duration of the transaction, since a very short transaction will produce a large number of aborts due to order inversion while a very long transaction will produce aborts due to the OS quantum reached. Apart from this, a very short duration will imply that the loop becomes practically regular.

Results

- The duration of the transaction, since a very short transaction will produce a large number of aborts due to order inversion while a very long transaction will produce aborts due to the OS quantum reached. Apart from this, a very short duration will imply that the loop becomes practically regular.
- The binomial transaction duration and loop regularity, since in the case of loop irregularity, STL, through load balancing, will always contribute to reducing aborts due to order inversion, but in the event that the loop is regular, the effectiveness of STL will depend on the duration of the transaction.

Results

- The duration of the transaction, since a very short transaction will produce a large number of aborts due to order inversion while a very long transaction will produce aborts due to the OS quantum reached. Apart from this, a very short duration will imply that the loop becomes practically regular.
- The binomial transaction duration and loop regularity, since in the case of loop irregularity, STL, through load balancing, will always contribute to reducing aborts due to order inversion, but in the event that the loop is regular, the effectiveness of STL will depend on the duration of the transaction.
- If it has a very short duration and is regular, STL will only cause an overhead since the iterations are already balanced and the runtime scheduler of tasks can delay the completion of the transactions and generate more aborts due to order inversion than not using tasks. While if the duration is not short and the loop is regular, the described overhead can be neglected and STL has no harmful or beneficial effect on aborts due to order inversion.

Results

- The %*lc* of each loop for a given input can alter the regularity of the loop tremendously if tied to `if` conditions.

Results

- The %*lc* of each loop for a given input can alter the regularity of the loop tremendously if tied to `if` conditions.
- The iteration size of the loop and the `S_SIZE` used in the parallelization to extend the transaction duration (short duration causes aborts due to order inversion) because if the capacity of the loop is exhaustive, the `S_SIZE` used to decrease aborts due to order inversion can generate aborts due to capacity overflow.

Agenda

- ~~Background~~
- ~~Performance Comparison~~
- ~~Experimental Evaluation~~
- **Conclusions**

Conclusions

Conclusions

- This paper compares two speculative loop parallelization techniques and shows that it is necessary to study some characteristics of loops to be parallelized.

Conclusions

- This paper compares two speculative loop parallelization techniques and shows that it is necessary to study some characteristics of loops to be parallelized.
- Generally, FOR-TLS performs much better on regular loops and STL on irregular loops; however, some cases, such as the problem with spec_private of arrays using tasks in loops from the susan_e benchmark, can alter the expected STL performance.

Conclusions

- This paper compares two speculative loop parallelization techniques and shows that it is necessary to study some characteristics of loops to be parallelized.
- Generally, FOR-TLS performs much better on regular loops and STL on irregular loops; however, some cases, such as the problem with spec_private of arrays using tasks in loops from the susan_e benchmark, can alter the expected STL performance.
- Moreover, the whole-program performance of each benchmark when using FOR-TLS deteriorates too much with respect to the performance of the loop, which does not happen in STL since it uses an efficient way of creating the team of threads.

Conclusions

- Finally, OpenMP `ordered` is not a technique that is well suited for loops that are *may* DOACROSS, as it serializes the iterations and has a synchronization overhead not counterbalanced by parallel segments, which are difficult to recognize and demarcate.

Thanks!