Lecture 08: Loop Level Parallelism

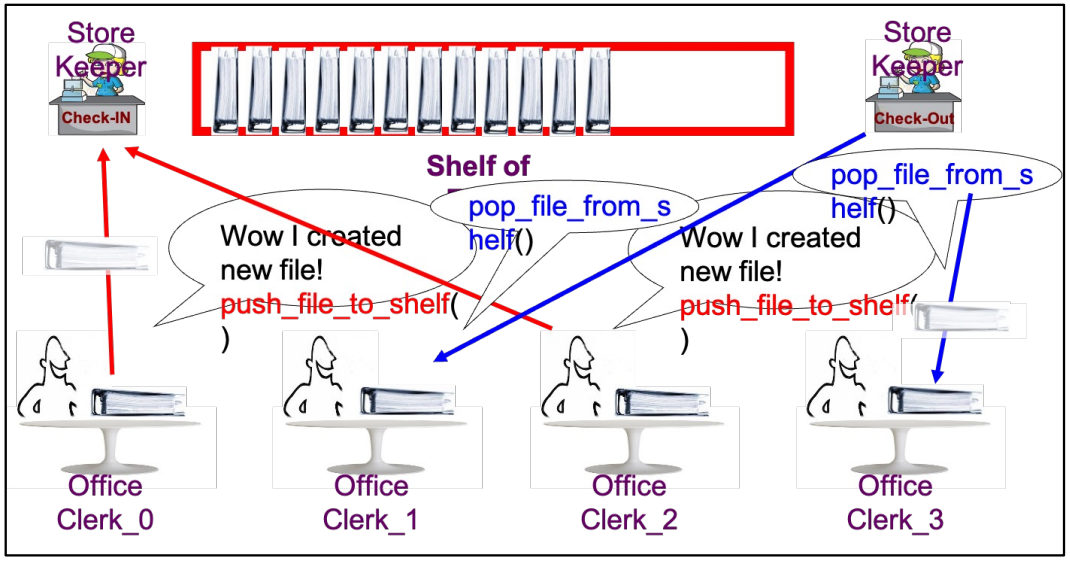
Vivek Kumar

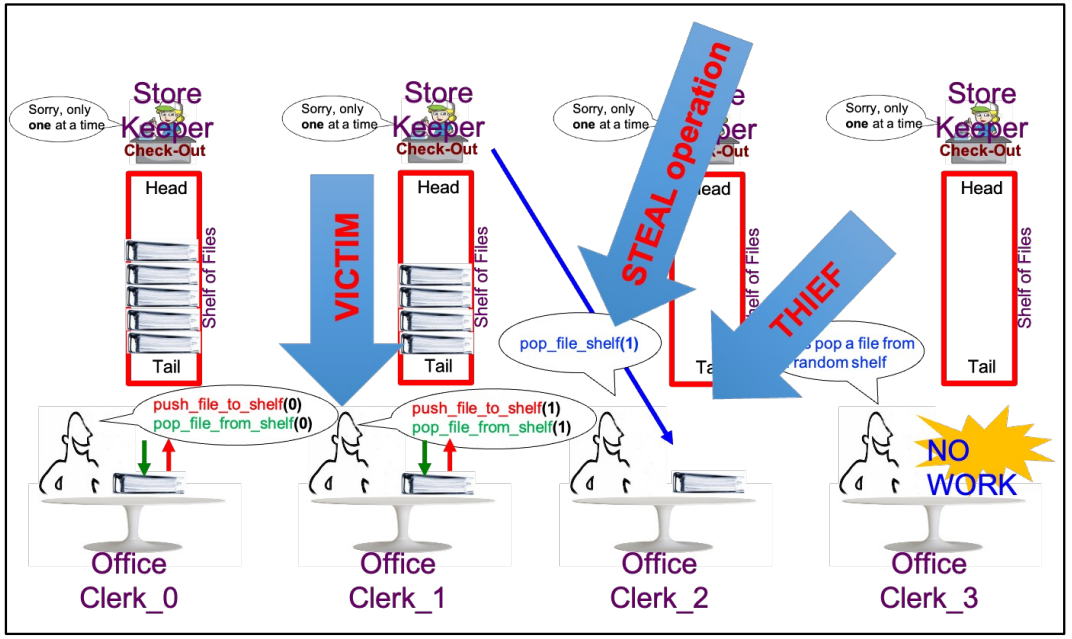
Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture





Today's Lecture

- **Lab-1 Solution**
- Types of work-stealing
- Loop level parallelism

Lab-01: Solution

```
void msort(int M, int N) {
  if (M < N) {
    int mid = M + (N - M) / 2;

    if (N - M > threshold) {
      hclib::finish([=]() {
        hclib::async([=]() {
          msort(M, mid);
        });
        msort(mid + 1, N);
      });
    } else { // sequential version
      msort(M, mid);
      msort(mid + 1, N);
    }
    merge(M, mid, N);
  }
}
```

// +2 Marks
// +2 Marks
// +2 Marks
// -1 Marks

```
int main(int argc, char** argv) {
  if(argc < 3) {
    printf("USAGE: %s <array size>
    exit(1);
  }

  hclib::launch([&]() {
  size = atoi(argv[1]);
  threshold = atoi(argv[2]);
```

Correct working = +2 Marks
Speedup calculated correctly = +2 marks

Today's Lecture

- Lab-1 Solution
- **Types of work-stealing**
- Loop level parallelism

Types of Work-Stealing

- Work-first
  - Cilk
- Help-first
  - Habanero-C library (HClib), Java fork/join

Types of Work-Stealing

With single worker, program execution using work-first policy is similar to serial execution

```
1. finish {
2.     async S1;
3.     //continuation of S1
4.     async S2;
5.     //continuation of S2
6.     S3;
7. }
```

Work-first

```
start_finish();
push_task_to_runtime(Line_3+);
S1;
if(Line_3+_stolen) return;
push_task_to_runtime(Line_5+);
S2;
if(Line_5+_stolen) return;
S3;
end_finish();
```

Help-first

```
start_finish();
push_task_to_runtime(S1);
push_task_to_runtime(S2);
S3;
end_finish();
```

Points to ponder

- What task is getting pushed to deque
  - Continuation in W.F.
  - “async” in H.F.
- When victim becomes a thief
  - When immediate continuation is stolen in W.F.
  - When all asyncs are stolen in H.F.

Parallel Array Sum using async and finish Constructs

Algorithm 2: Two-way Parallel ArraySum

Input: Array of numbers, X.
Output: sum = sum of elements in array X.

```
// Start of Task T1 (main program)
sum1 ← 0; sum2 ← 0;
// Compute sum1 (lower half) and sum2 (upper half) in parallel.
finish{
    async{
        // Task T2
        for i ← 0 to X.length/2 − 1 do
            sum1 ← sum1 + X[i];
    };
    async{
        // Task T3
        for i ← X.length/2 to X.length − 1 do
            sum2 ← sum2 + X[i];
    };
};
// Task T1 waits for Tasks T2 and T3 to complete
// Continuation of Task T1
sum ← sum1 + sum2;
return sum;
```

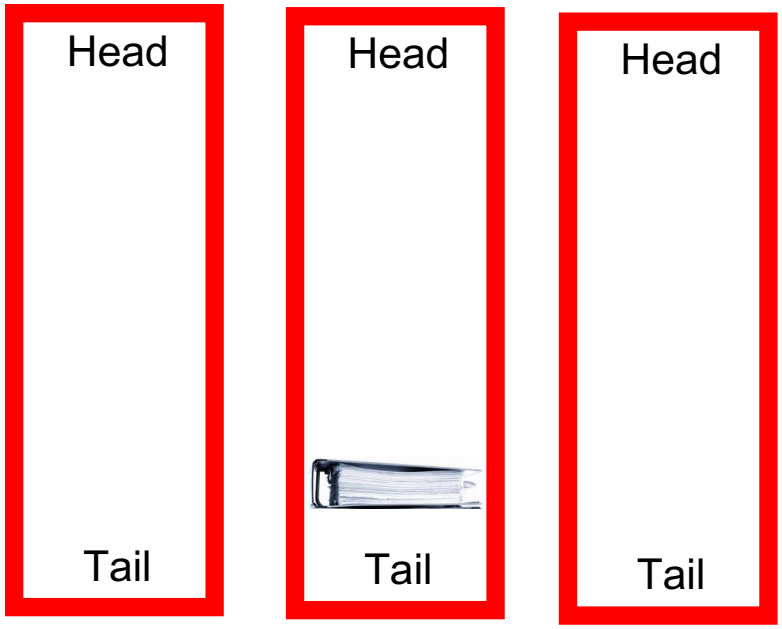
How tasks will be executed in this program over work-first and help-first work-stealing scheduler?

Source: https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s16-lec1-slides.pdf?version=1&modificationDate=1452732285045&api=v2

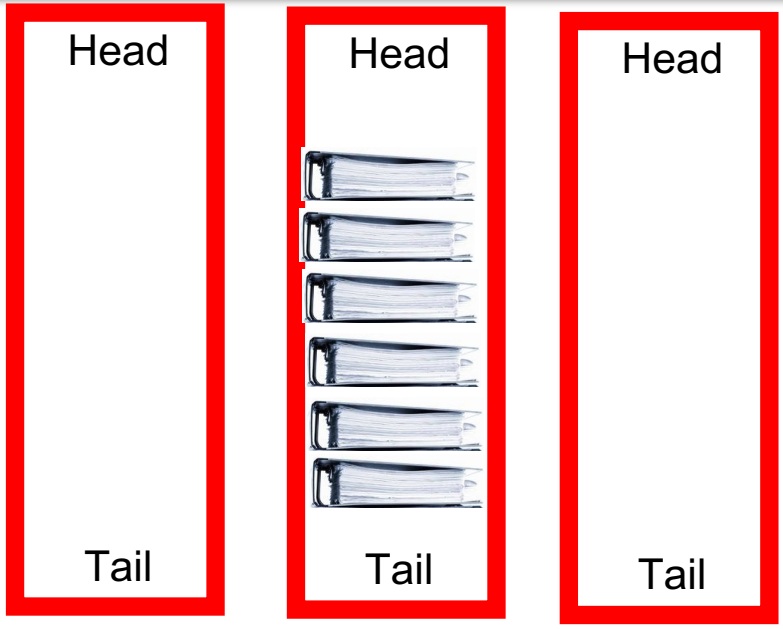
Types of Work-Stealing

```
finish {
   for(int i=0; i<N; i++) {
      async S; // S does not spawn any async
   }
}
```

Does it affect steal operations?



Work-first: at any given time there will be just one task available for stealing. New task will be generated only after the first one is stolen, leading to serialized steals. This will become scalability bottleneck with large number of workers



Help-first: plenty of tasks available for stealing as all the tasks are created upfront.

Types of Work-Stealing

- Does it affect context switches?
  - Work-first
    - Every steal will trigger a context switch of the victim
  - Help-first
    - Every task is executed after a context switch

Today's Lecture

- Types of work-stealing
- **Loop level parallelism**

Loop Level Parallelism

```
void foo() {
  for (uint64_t i=0; i<SIZE; i++) {
    S(i); // can execute in parallel for all i
  }
}
```

- Case-1: S(i) is doing the same amount of computation in each iteration
  - Static decomposition is applied and each worker receives equal sized chunk of the for-loop
    - Pros:
      - Perfect load balancing
        - total tasks = total workers (assuming SIZE%num_workers==0)
    - Cons
      - Programmer has to modify the sequential code for avoiding tasking overheads and for achieving perfect load balancing
        - Hampers productivity as no serial elision

Loop Level Parallelism

```
void foo() {
  for (uint64_t i=0; i<SIZE; i++) {
    S(i); // can execute in parallel for all i
  }
}
```

- Case-2: S(i) is **NOT** doing the same amount of computation in each iteration
  - Static decomposition still possible but the programmer has to divide total iterations into "small chunks" (tiling)
    - Pros:
      - Perfect load balancing is still possible if total_chunks >> total_workers
    - Cons
      - Programmer has to modify the sequential code for avoiding tasking overheads and for achieving perfect load balancing
        - Hampers productivity as no serial elision
      - If S(i) does not create any async then there is a single producer producing large number of tasks and there are multiple consumer
        - How does it affects work-first and help-first scheduling ?

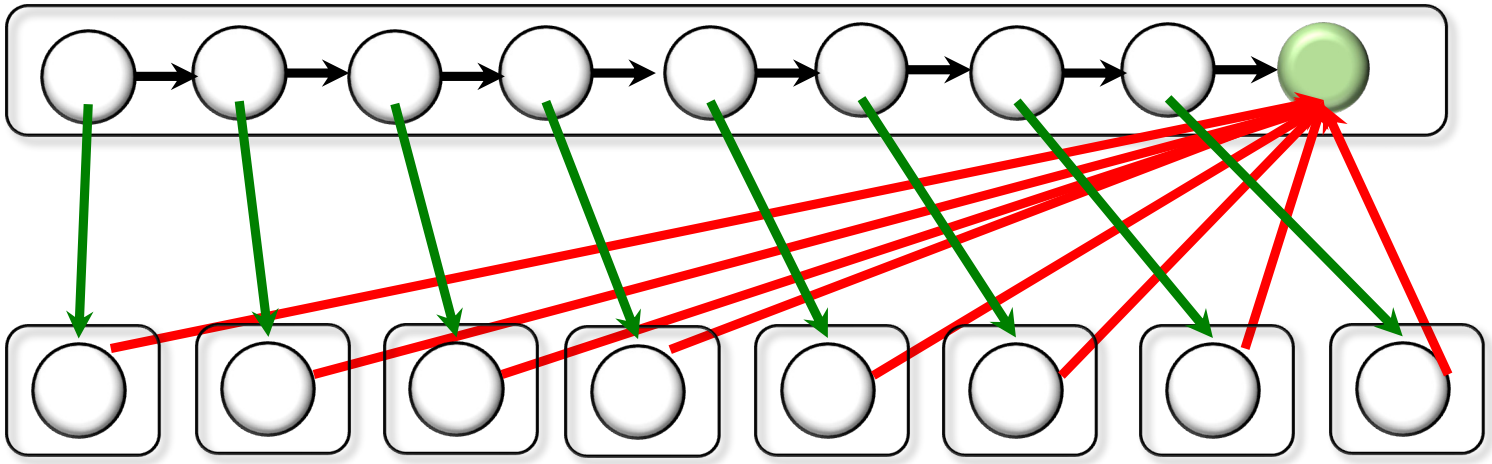
"forasync" – Construct for Harnessing Loop Level Parallelism in HClib

```
void foo() {
  loop_domain_t loop = {0, SIZE, 1, tile_size};
  finish([&]() {
    forasync1D (&loop, [=](int i) {
      S(i); // can execute in parallel for all i
    }, FORASYNC_MODE_RECURSIVE);
  });
}
```

- loop_domain_t loop = {lowBound, highBound, loopStride, tileSize};
  - forasync1D(loop_domain_t* loop, lambda_function, mode);
- loop_domain_t loop[2] = { {lowBound0, highBound0, loopStride0, tileSize0}, {…} };
  - forasync2D(loop_domain_t* loop, lambda_function, mode);
- loop_domain_t loop[3] = { {lowBound0, highBound0, loopStride0, tileSize0}, {…}, {…} };
  - forasync3d(loop_domain_t* loop, lambda_function, mode);
- mode
  - FORASYNC_MODE_RECURSIVE: recursively partition total iteration space until "tileSize" is reached
  - FORASYNC_MODE_FLAT: chunk iterations into blocks of length "tileSize"

FORASYNC_MODE_FLAT

```
void foo() {
  loop_domain_t loop = {0, 8, 1, 1};
  finish([&]() {
    forasync1D (&loop, [=](int i) {
      S(i); // can execute in parallel for all i
    }, FORASYNC_MODE_FLAT);
  });
}
```



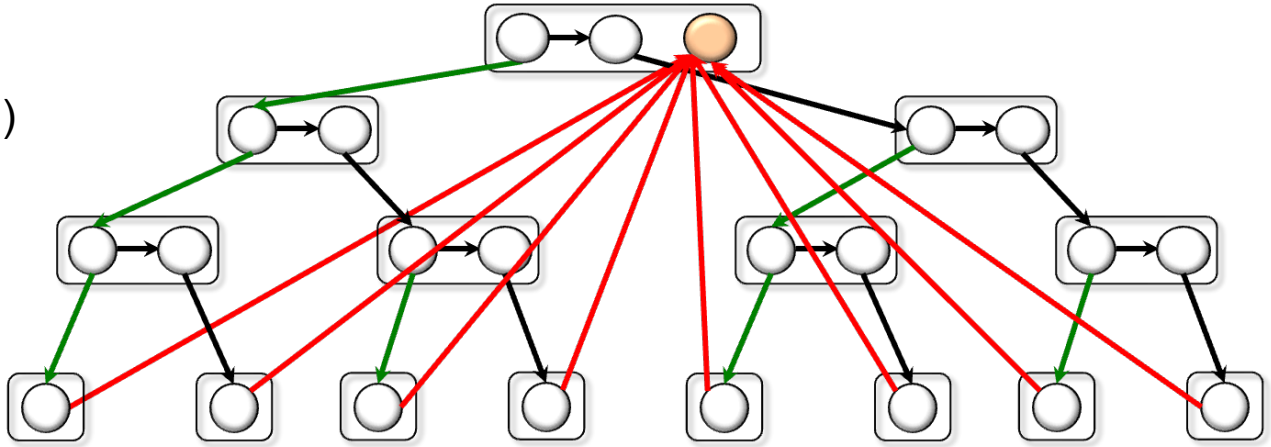
Work = O(n)
CPL = O(n)

Modification of source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec02_parallelism.pdf

FORASYNC_MODE_RECURSIVE (Divide-and-Conquer)

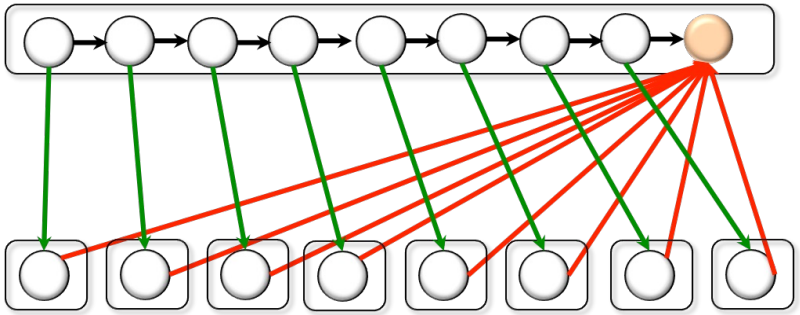
```
void foo() {
  loop_domain_t loop = {0, 8, 1, 1};
  finish([&]() {
    forasync1D (&loop, [=](int i) {
      S(i); // can execute in parallel for all i
    }, FORASYNC_MODE_FLAT);
  });
}
```

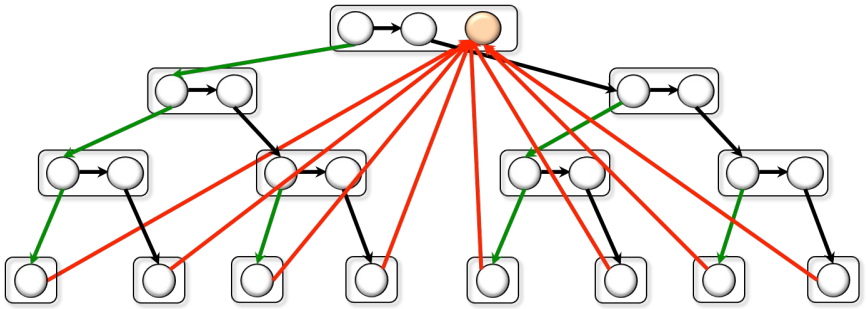
Work = O(n)
CPL = O(log n)



Modification of source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec02_parallelism.pdf

Question





- In a possible scenario, HClib (help-first scheduling) is initialized with a single worker. This worker is having a deque of size **N**. You have to launch a forasync1D computation with loop_domain_t = {0, **2N**, 1, 1}. Will there will be a deque overflow?
  - Yes if FORASYNC_MODE_FLAT
  - No if FORASYNC_MODE_RECURSIVE

Parallelizing Matrix Multiplication

```
for (uint64_t i=0; i<N; i++) {
  for (uint64_t j=0; j<N; j++) {
    C[i][j] = 0;
  }
}

for (uint64_t i=0; i<N; i++) {
  for (uint64_t j=0; j<N; j++) {
    for (uint64_t k=0; k<N; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

- Sequential matrix multiplication (NxN)
  - How to parallelize using forasync?

Observations on finish-for-async Version of Parallel Matrix Multiplication

- finish and async are general constructs, and are not specific to loops
- Loops in sequential version of matrix multiplication are "perfectly nested"
  - e.g., no intervening statement between "for(i = ...)" and "for(j = ...)"
- The ordering of loops nested between finish and async is arbitrary
  - They are parallel loops and their iterations can be executed in any order

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec11-slides-v1.key.pdf?version=1&modificationDate=1483206144935&api=v2

Parallelizing Matrix Multiplication

```
loop_domain_t loop[2] = { {0, N, 1, tile}, {0, N, 1, tile} };


  forasync2D (loop, [=] (int i, int j) {
      C[i][j] = 0;
  }, FORASYNC_MODE_RECURSIVE);

                                  Data Race !!

  forasync2D (loop, [=] (int i, int j) {
      for (uint64_t k=0; k<N; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
  }, FORASYNC_MODE_RECURSIVE);
```

- Parallel matrix multiplication (NxN)
  - forasync2D

Parallelizing Matrix Multiplication

```
loop_domain_t loop[2] = { {0, N, 1, tile}, {0, N, 1, tile} };

finish {[&]() {
  forasync2D (loop, [=] (int i, int j) {
      C[i][j] = 0;
  }, FORASYNC_MODE_RECURSIVE);
});

finish {[&]() {
  forasync2D (loop, [=] (int i, int j) {
      for (uint64_t k=0; k<N; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
  }, FORASYNC_MODE_RECURSIVE);
});
```

- Parallel matrix multiplication (NxN)
  - forasync2D
    - High productivity achieved !

Code available on github:
https://github.com/vivkumar/cse502/blob/master/hclib/test/lec11/

Next Lecture

- Mutual Exclusion in Async-Finish Programs
- Quiz-2
  - Syllabus
    - Lectures 5-8