Lecture 12: Recursive Task Parallelism on NUMA Architecture

Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Today's Lecture

- NUMA aware work-stealing in boost::fibers
- Assigning NUMA affinity to async tasks

NUMA Aware Work-Stealing in boost::fibers

```
int main() {
  // Step-1: Gather NUMA topology
  std::vector<boost::fibers::numa::node> topo = boost::fibers::numa::topology();
  // Step-2: Main thread is on Core-0, and rest of the threads pinned to other cores
  auto start_cpuID = *node.logical_cpus.begin();
  std::threads* threads = new std::threads[NUM_CORES];
  int wid=0;
  for(auto & node : topo) {
    for(std::uint32_t cpuID node.logical_cpus) {
      if(start_cpuID != cpuID) { // Exclude master thread
        threads[wid]=std::thread(worker_routine, cpuID, node.id, std::cref(topo));
      }
      wid++;
    }
  }
  // Step-3: Set scheduling policy at fiber manager
  boost::fibers::use_scheduling_algorithm
        <boost::fibers::numa::algo::work_stealing>(start_cpuID, node.id, topo);
  // Rest of the steps same as in default work-stealing (Lecture 05, Slide #7)
}
```

- Main thread
  - Gathers the NUMA topology
  - Pins to core-0
  - Create worker threads on rest of the cores
  - Set NUMA aware work-stealing policy

More info: https://www.boost.org/doc/libs/1_80_0/libs/fiber/doc/html/fiber/numa.html

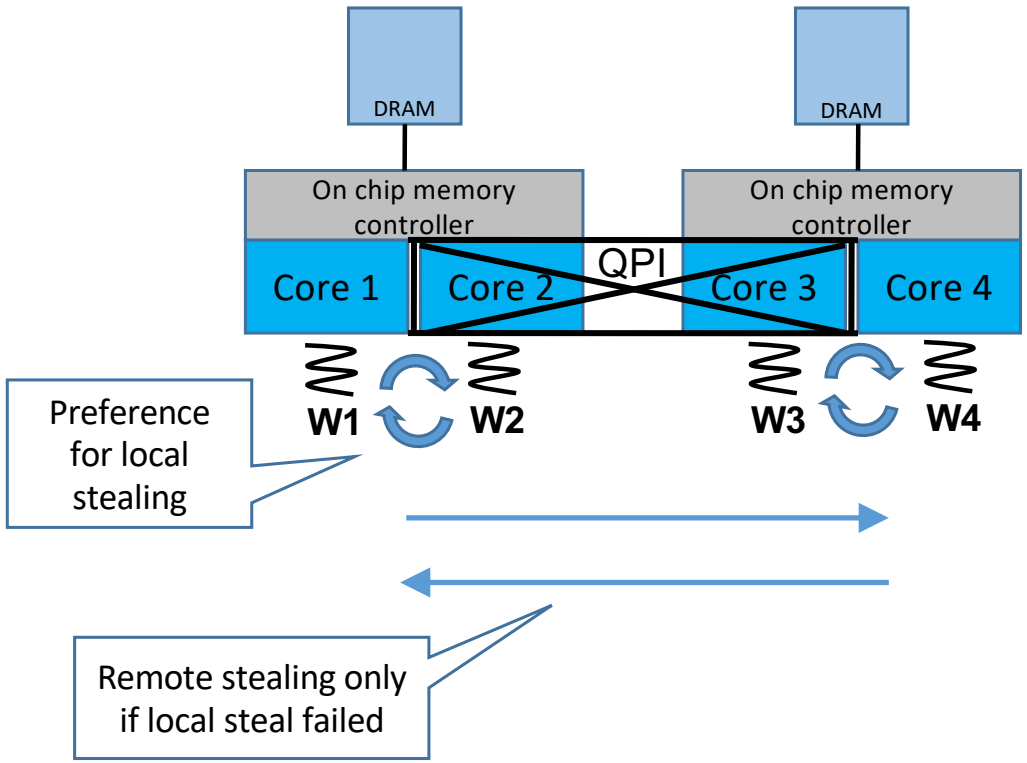
NUMA Aware Work-Stealing in boost::fibers

```
void worker_routine(uint32_t cpuID, uint32_t nodeID,
    std::reference_wrapper<const std::vector<boost::fibers::numa::node>> const &topo) {
  // Step-1: Set scheduling policy at fiber manager
  boost::fibers::use_scheduling_algorithm
        <boost::fibers::numa::algo::work_stealing>(cpuID, nodeID, topo);
  // Rest of the steps same as in default work-stealing (Lecture 05, Slide #8)
}
```

- Worker threads
  - Pins to rest of the cores
  - Set NUMA aware work-stealing policy

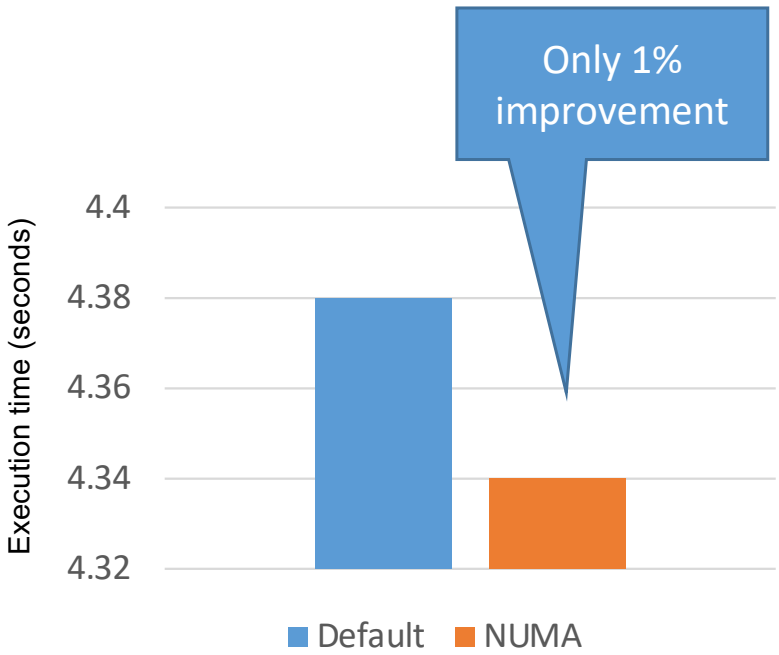
More info: https://www.boost.org/doc/libs/1_80_0/libs/fiber/doc/html/fiber/numa.html

NUMA Aware Work-Stealing in boost::fibers



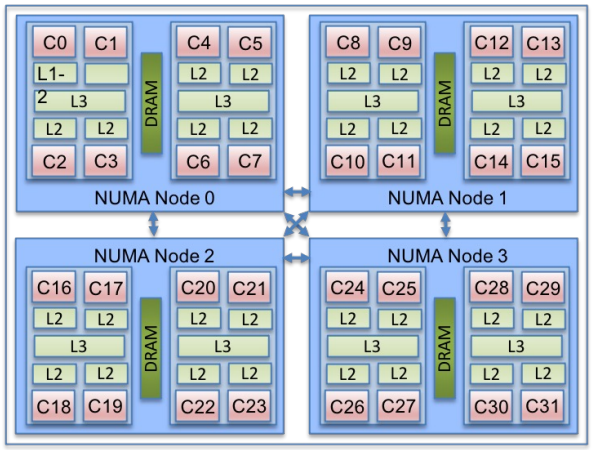
- Stealing algorithm
  - If the local queue of one thread runs out of ready fibers, the thread tries to steal a ready fiber from another thread running within the same NUMA-node
    - Local memory accesses
  - If local steals failed, the thread tries to steal fibers from other NUMA-nodes
    - Remote memory accesses
- Any drawbacks?
  - Does not do anything special about NUMA memory allocation
  - Task and its data could be on two different NUMA domains

NUMA Aware Work-Stealing in boost::fibers



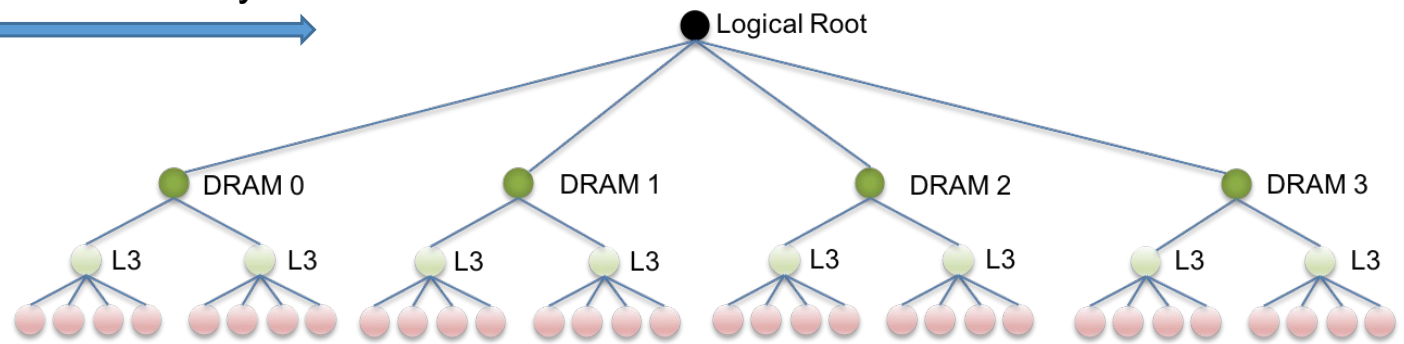
- Evaluation of a recursive task parallel array sum
  - Integer array of total size 1GB
    - Interleaved page allocation on NUMA nodes
  - Launches detached fiber for the first recursive call, and executes the second recursive call sequentially
    - Task creation stops if chunk is <= page size (4KB)
  - AMD EPYC 7551 32-Core processor with hyperthreading enabled
    - Four NUMA domains, each having 16 logical cores

NUMA Hierarchy of Modern Processors

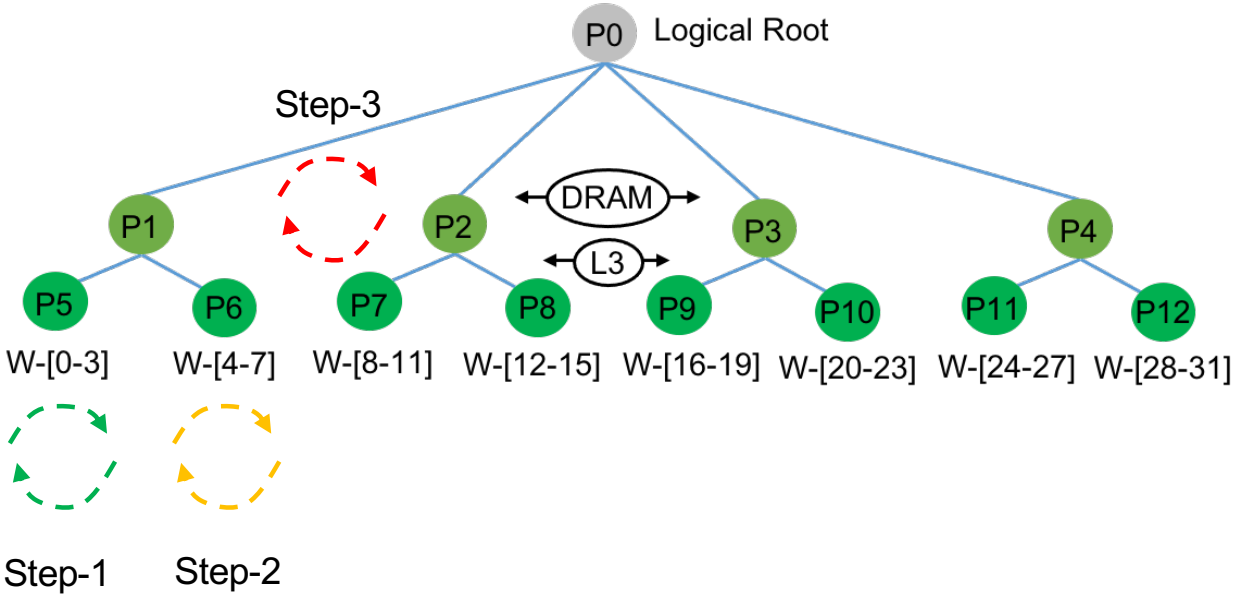


32 Core AMD EPYC Processor

NUMA hierarchy

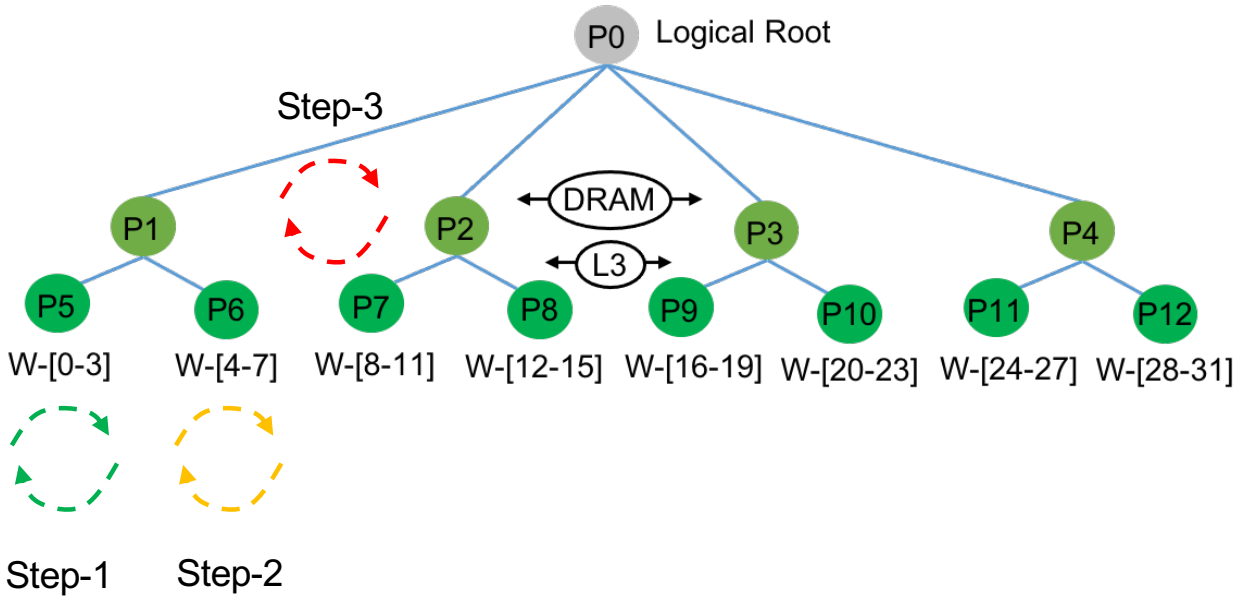


Hierarchical Work-Stealing: How?



- How to improve the locality on modern NUMA processors?
  - Give high preference for local stealing
    - Workers sharing a common L3 can first attempt steal among themselves
    - Try stealing from the sibling L3 (under same DRAM) if a worker failed to steal from victims sharing L3 with the thief
  - Try remote stealing only if local stealing failed

Hierarchical Work-Stealing: Issues?



1. **Performance**
   - Random v/s round robin work-stealing?
   - Who could perform remote steals?
   - How to ensure the locality of memory pages?
     - Page migrations could be performed, but it can lead to overheads
2. **Productivity**
   - Any possibility to improve the locality/performance of any type of recursive application without hurting programmer's productivity?
     - As of now we are considering that the programmer creates the seed task at each NUMA domain

Performance: Random VS. Round Victim Selection

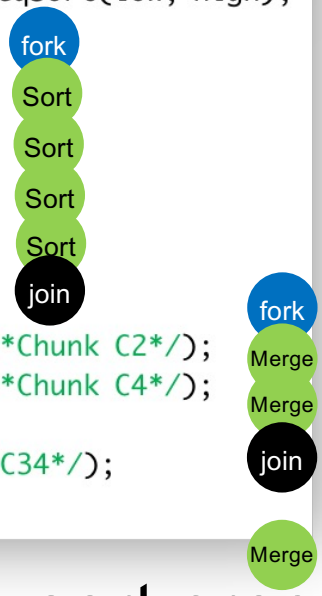
- Random victim selection
  - It ensures quick distribution of work even in highly imbalanced cases
  - May not be optimal in hierarchical work-stealing as remote steals happens only after failed steal attempt from **each and every** local victims
    - Can cause starvation by delaying the remote steals when random victim selection returns the same local victim multiple times
- Using a mix of random & round-robin victim selection during hierarchical stealing
  - Local steal: round robin victim selection inside the local NUMA domain
  - Remote steal: thief chooses a victim from a remote NUMA node based on the distance of the NUMA node
    - Round robin victim selection from a nearby node first, and then from farther node if the steal from nearby node failed
    - If same distance between all NUMA nodes, then first randomly select a NUMA node, and then use round-robin victim selection inside that NUMA node

Performance: Who Could Perform Remote Steals

- Leader based approach
  - A leader could be chosen within each NUMA domain who should be requested for carrying out remote steals
    - Work best in case of a distributed work-stealing (over a cluster) to reduce network congestion
- Free to steal approach
  - Any worker can perform remote steals if it has failed locally
    - Works fine in shared memory based work-stealing

Productivity & Performance: Recursive Programs

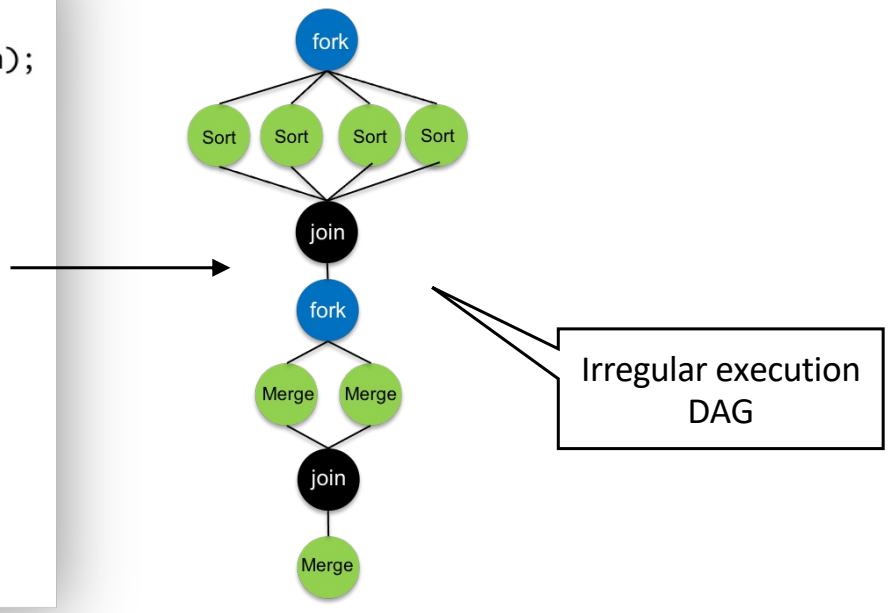
```
1. int *A;
2. void Sort(int low, int high) {
3.   if((high-low)<LIMIT) return SeqSort(low, high);
4.   int Chunks=(high-low)/4;
5.   finish {
6.     async Sort(/*Chunk C1*/);
7.     async Sort(/*Chunk C2*/);
8.     async Sort(/*Chunk C3*/);
9.     async Sort(/*Chunk C4*/);
10.  }
11.  finish {
12.    async Merge(/*Chunk C1*/, /*Chunk C2*/);
13.    async Merge(/*Chunk C3*/, /*Chunk C4*/);
14.  }
15.  Merge(/*Chunk C12*/, /*Chunk C34*/);
16.}
```



- Parallel merge sort program generating **four** recursive tasks for call to sort, and **two** recursive tasks for merging

Productivity & Performance: Recursive Programs

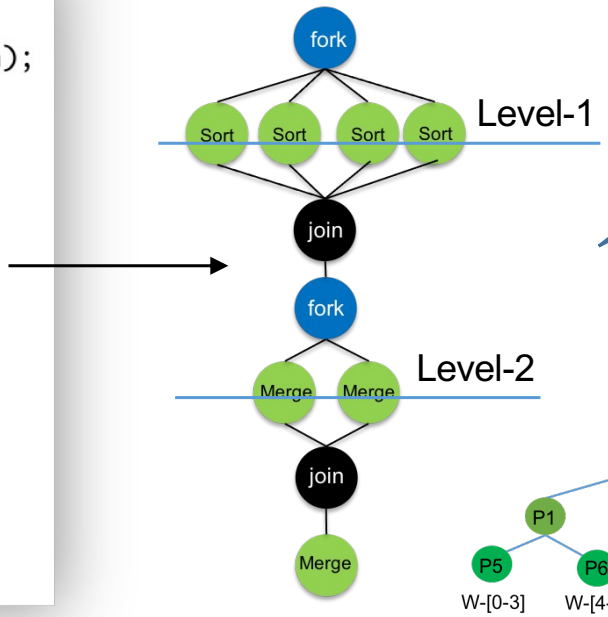
```
1. int *A;
2. void Sort(int low, int high) {
3.   if((high-low)<LIMIT) return SeqSort(low, high);
4.   int Chunks=(high-low)/4;
5.   finish {
6.     async Sort(/*Chunk C1*/);
7.     async Sort(/*Chunk C2*/);
8.     async Sort(/*Chunk C3*/);
9.     async Sort(/*Chunk C4*/);
10.  }
11.  finish {
12.    async Merge(/*Chunk C1*/, /*Chunk C2*/);
13.    async Merge(/*Chunk C3*/, /*Chunk C4*/);
14.  }
15.  Merge(/*Chunk C12*/, /*Chunk C34*/);
16.}
```



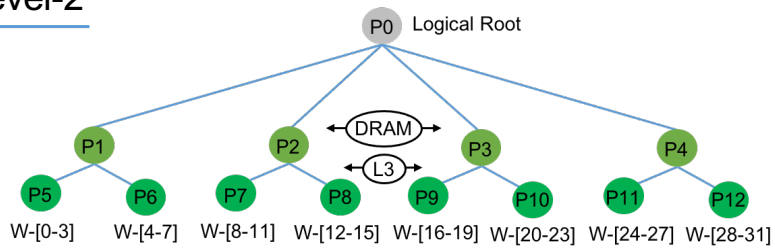
- This program generates an irregular execution DAG as total number of children at each fork point isn't the same

Productivity & Performance: Recursive Programs

```
1. int *A;
2. void Sort(int low, int high) {
3.   if((high-low)<LIMIT) return SeqSort(low, high);
4.   int Chunks=(high-low)/4;
5.   finish {
6.     async Sort(/*Chunk C1*/);
7.     async Sort(/*Chunk C2*/);
8.     async Sort(/*Chunk C3*/);
9.     async Sort(/*Chunk C4*/);
10.  }
11.  finish {
12.    async Merge(/*Chunk C1*/, /*Chunk C2*/);
13.    async Merge(/*Chunk C3*/, /*Chunk C4*/);
14.  }
15.  Merge(/*Chunk C12*/, /*Chunk C34*/);
16.}
```



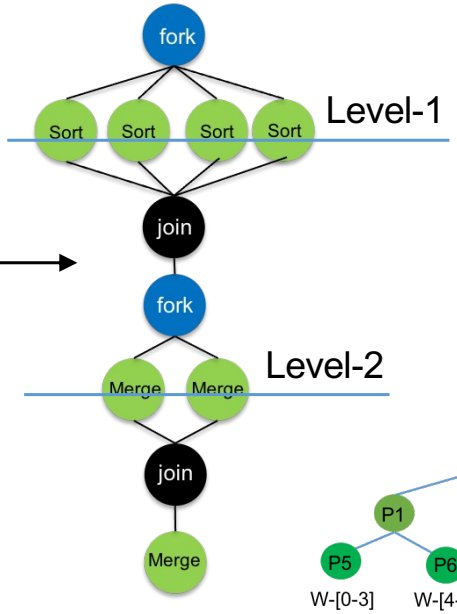
How to initiate the computation for good performance?

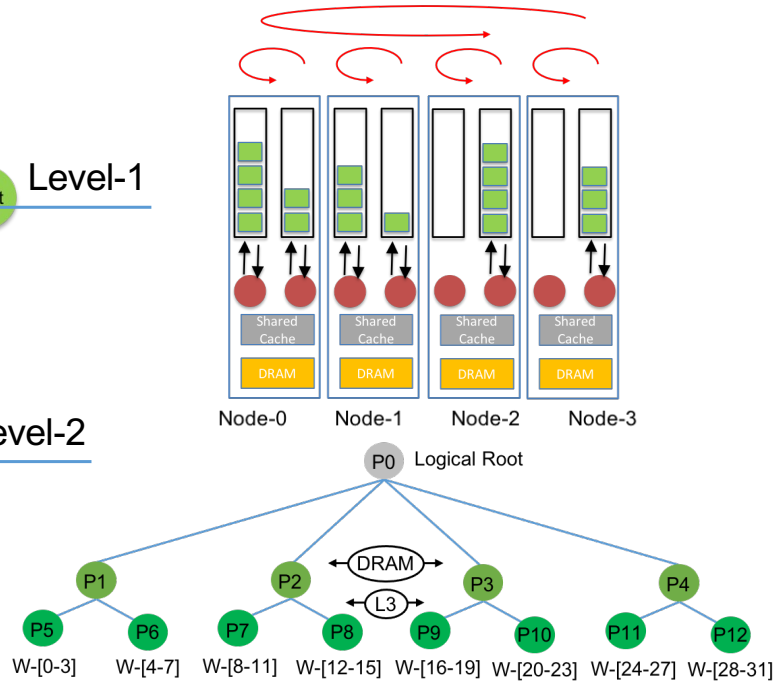


- To improve the performance, each node should get an equal size computation (seed task) to begin with
  - Four seed tasks at top-level for recursive call to sort (Level-1)
  - Two seed tasks at top-level for recursive call to merge (Level-2)

Productivity & Performance: Recursive Programs

```
1. int *A;
2. void Sort(int low, int high) {
3.   if((high-low)<LIMIT) return SeqSort(low, high);
4.   int Chunks=(high-low)/4;
5.   finish {
6.     async Sort(/*Chunk C1*/);
7.     async Sort(/*Chunk C2*/);
8.     async Sort(/*Chunk C3*/);
9.     async Sort(/*Chunk C4*/);
10.  }
11.  finish {
12.    async Merge(/*Chunk C1*/, /*Chunk C2*/);
13.    async Merge(/*Chunk C3*/, /*Chunk C4*/);
14.  }
15.  Merge(/*Chunk C12*/, /*Chunk C34*/);
16.}
```





- Seed task at level-1 generates equal load on all four NUMA nodes as all four NUMA node receives a seed task
- Unequal load in level-2 as only two top-level tasks are available for four NUMA nodes
  - Load imbalance as nodes without a seed task would starve, and would directly start stealing from remote NUMA nodes
  - Bad locality as task and its data may not be on the same NUMA node as tasks are being migrated across the nodes during the merge phase (Level-2)
- Low productivity as the programmer has to create the seed tasks for each NUMA node at each levels (only once at the start)

Productivity & Performance: Recursive Programs

```
1. int *A;
2. void Sort(int low, int high) {
3.   if((high-low)<LIMIT) return SeqSort(low, high);
4.   int Chunks=(high-low)/4;
5.   finish {
6.     async Sort(/*Chunk C1*/);
7.     async Sort(/*Chunk C2*/);
8.     async Sort(/*Chunk C3*/);
9.     async Sort(/*Chunk C4*/);
10.  }
11.  finish {
12.    async Merge(/*Chunk C1*/, /*Chunk C2*/);
13.    async Merge(/*Chunk C3*/, /*Chunk C4*/);
14.  }
15.  Merge(/*Chunk C12*/, /*Chunk C34*/);
16.}
```

async-finish for UMA

```
1. int *A;
2. void Sort(int low, int high) {
3.   if((high-low)<LIMIT) return SeqSort(low, high);
4.   int Chunks=(high-low)/4;
5.   finish {
6.     async_hinted (A, C1_start, C1_end) Sort(/*Chunk C1*/);
7.     async_hinted (A, C2_start, C2_end) Sort(/*Chunk C2*/);
8.     async_hinted (A, C3_start, C3_end) Sort(/*Chunk C3*/);
9.     async_hinted (A, C4_start, C4_end) Sort(/*Chunk C4*/);
10.  }
11.  finish {
12.    async_hinted (A, C1_start, C2_end) Merge(/*Chunk C1*/, /*Chunk C2*/);
13.    async_hinted (A, C3_start, C4_end) Merge(/*Chunk C3*/, /*Chunk C4*/);
14.  }
15.  Merge(/*Chunk C12*/, /*Chunk C34*/);
16.}
17.void kernel() {
18.  A = numa_alloc_blockcyclic<int>(N);
19.  initialize();
20.  Sort(0, N);
21.  numa_free(A)
22.}
```

async_hinted-finish for NUMA

- Programmer allocates the array in block cyclic manner over all NUMA nodes (block size = array_size/num_numa_nodes)
- Programmer assign data-affinity hints to each async tasks
  - No program modification based on NUMA architecture
    - Same program can run on any number of NUMA nodes as the parallel runtime will dynamically create tasks on a node that contains the range of memory addresses on which that task is going to operate
  - High productivity
    - Except for two NUMA memory allocation/deallocation APIs

- Let us try to understand the implementation of the async_hinted using a simple parallel recursive array sum having that generates two tasks in each recursion
  - Note that it is mandatory to use async_hinted for each parallel tasks, i.e., for both left and right chunk

Productivity & Performance: Recursive Programs

```
int array_sum(int low, int high) {
  if(high - low > 512) {
    int mid = (low + high)/2;
    future<int> left = async_hinted(array, low, mid, [=]() {
                           return array_sum(low, mid);
                       });
    future<int> right = async_hinted(array, mid, high, [=]() {
                           return array_sum(mid, high);
                        });
    return left.get() + right.get();
  } else {
    int sum = 0;
    for(int i=low; i<high; i++) {
       sum += array[i];
    }
    return sum;
  }
}

int main() {
  int* array = numa_alloc_blocksize(4096); // 4 memory pages
  array_sum(0, 4096);
}
```

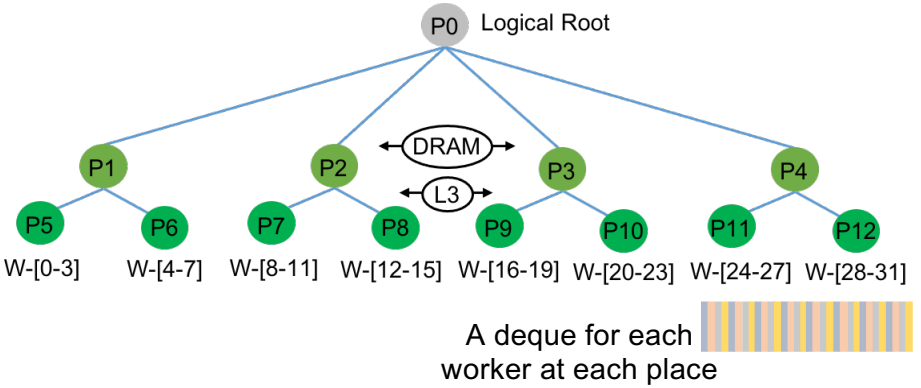
- Recursive array sum
  - Each recursion must be an async_hinted task
  - Task must supply the starting address of the array being accessed and the range of array access

Productivity & Performance: Recursive Programs

```
int array_sum(int low, int high) {
  if(high - low > 512) {
    int mid = (low + high)/2;
    future<int> left = async_hinted(array, low, mid, [=]() {
                           return array_sum(low, mid);
                       });
    future<int> right = async_hinted(array, mid, high, [=]() {
                           return array_sum(mid, high);
                        });
    return left.get() + right.get();
  } else {
    int sum = 0;
    for(int i=low; i<high; i++) {
       sum += array[i];
    }
    return sum;
  }
}

int main() {
  int* array = numa_alloc_blocksize(4096); // 4 memory pages
  array_sum(0, 4096);
}
```

- Hierarchical place tree (HPT)
  - A tree data structure where each node (called as place) represents a level in the NUMA memory hierarchy
  - Each node has “N” number of deques, where “N” is total size of thread pool
    - Allows each worker to push/pop task at any place in the HPT without asynchronously



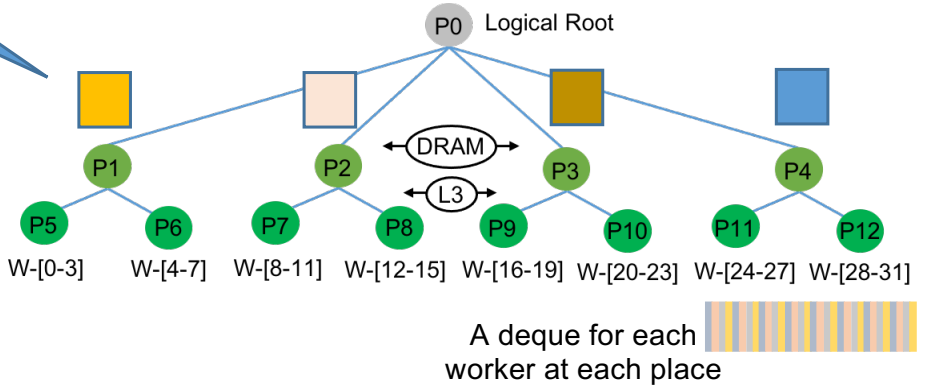
Productivity & Performance: Recursive Programs

```
int array_sum(int low, int high) {
  if(high - low > 512) {
    int mid = (low + high)/2;
    future<int> left = async_hinted(array, low, mid, [=]() {
                           return array_sum(low, mid);
                       });
    future<int> right = async_hinted(array, mid, high, [=]() {
                           return array_sum(mid, high);
                        });
    return left.get() + right.get();
  } else {
    int sum = 0;
    for(int i=low; i<high; i++) {
       sum += array[i];
    }
    return sum;
  }
}

int main() {
  int* array = numa_alloc_blocksize(4096); // 4 memory pages
  array_sum(0, 4096);
}
```

Block cyclic allocation of 4 memory pages on 4 NUMA nodes

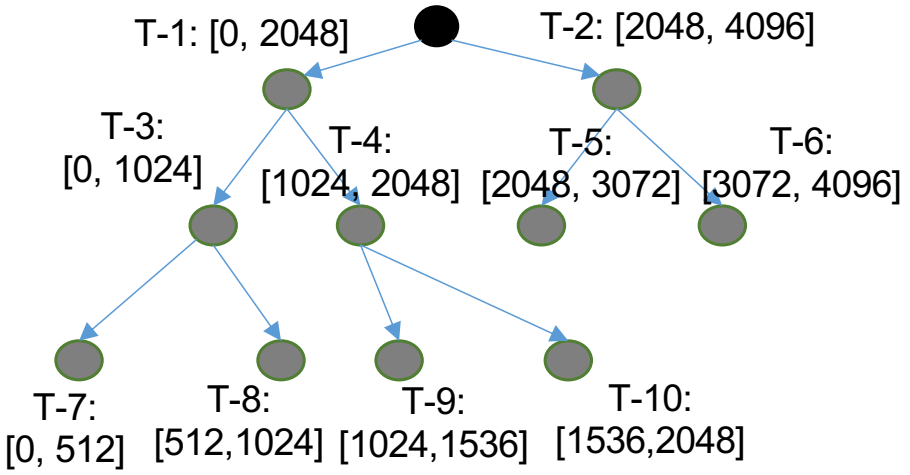
- Block cyclic allocation will ensure equal number (contiguous) of physical pages allocated under each NUMA domain
  - Places P1 and other places in its subtree (P5 & P6) will have affinity to memory residing at NUMA node represented by P1
    - Likewise for places P2, P3, and P4



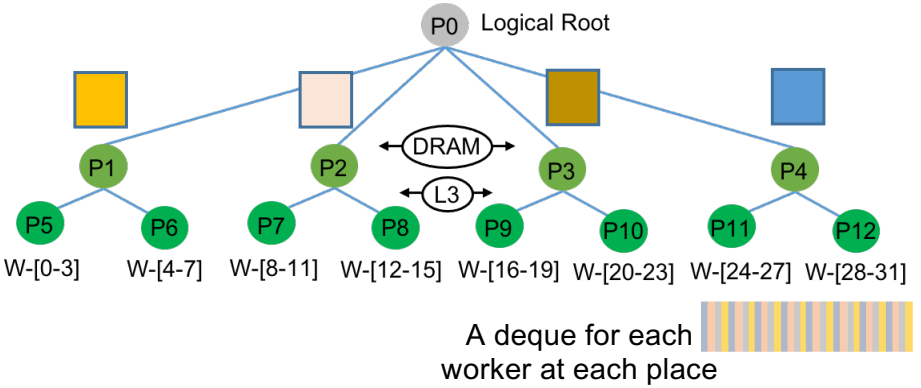
Productivity & Performance: Recursive Programs

```
int array_sum(int low, int high) {
  if(high - low > 512) {
    int mid = (low + high)/2;
    future<int> left = async_hinted(array, low, mid, [=]() {
                           return array_sum(low, mid);
                       });
    future<int> right = async_hinted(array, mid, high, [=]() {
                           return array_sum(mid, high);
                        });
    return left.get() + right.get();
  } else {
    int sum = 0;
    for(int i=low; i<high; i++) {
       sum += array[i];
    }
    return sum;
  }
}

int main() {
  int* array = numa_alloc_blocksize(4096); // 4 memory pages
  array_sum(0, 4096);
}
```



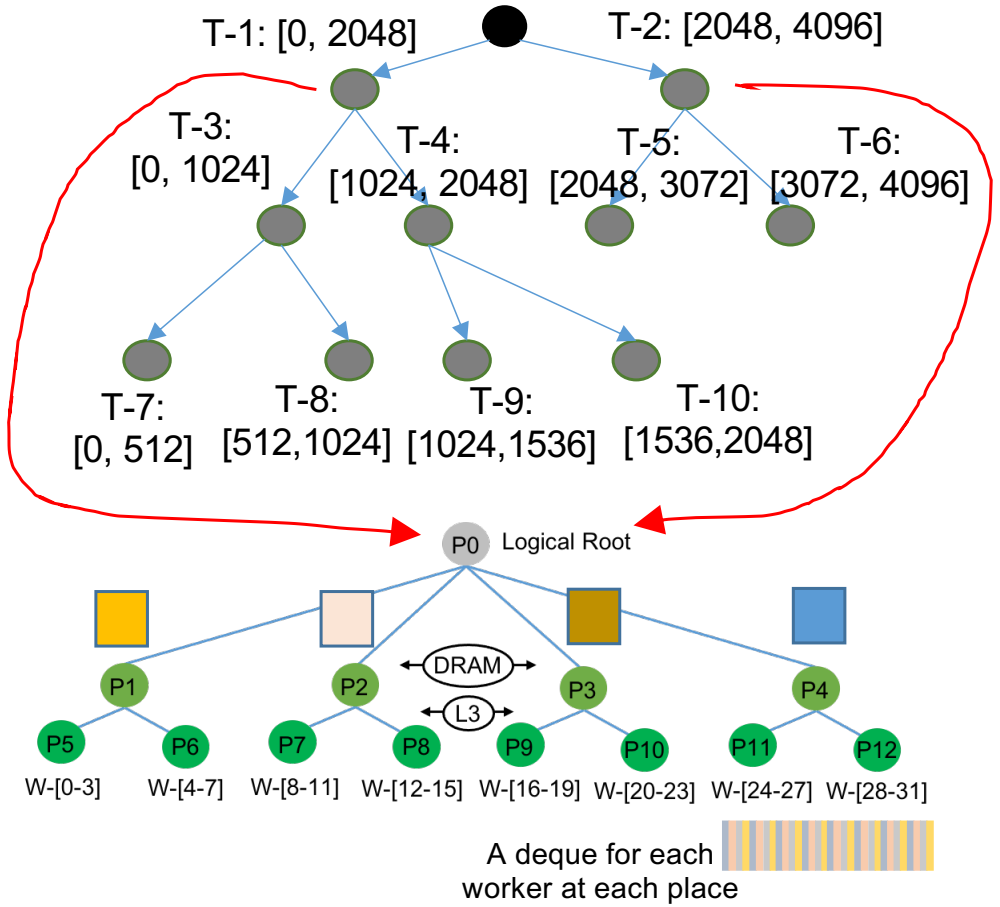
- Any worker executing an async_hinted has to dynamically decide the place in the HPT where it should push this task (affinity of task)
- It calculates the palce affinity of a task using the range of memory being accessed inside this task



Productivity & Performance: Recursive Programs

```
int array_sum(int low, int high) {
  if(high - low > 512) {
    int mid = (low + high)/2;
    future<int> left = async_hinted(array, low, mid, [=]() {
                          return array_sum(low, mid);
                       });
    future<int> right = async_hinted(array, mid, high, [=]() {
                          return array_sum(mid, high);
                        });
    return left.get() + right.get();
  } else {
    int sum = 0;
    for(int i=low; i<high; i++) {
       sum += array[i];
    }
    return sum;
  }
}

int main() {
  int* array = numa_alloc_blocksize(4096); // 4 memory pages
  array_sum(0, 4096);
}
```

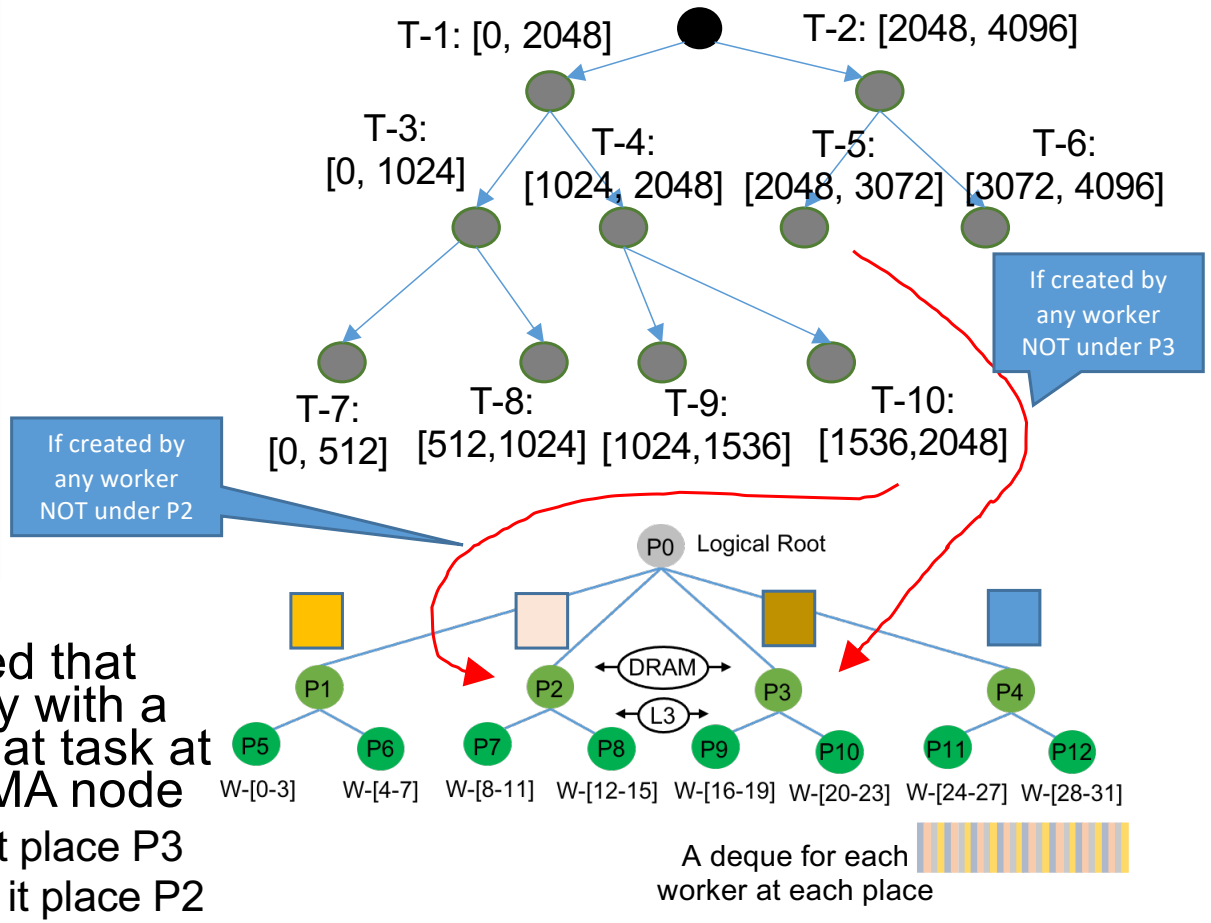


- Any task whose memory access span across multiple NUMA domains (P1, P2, P3, and P4), should be pushed at Place P0

Productivity & Performance: Recursive Programs

```
int array_sum(int low, int high) {
  if(high - low > 512) {
    int mid = (low + high)/2;
    future<int> left = async_hinted(array, low, mid, [=]() {
                           return array_sum(low, mid);
                       });
    future<int> right = async_hinted(array, mid, high, [=]() {
                           return array_sum(mid, high);
                        });
    return left.get() + right.get();
  } else {
    int sum = 0;
    for(int i=low; i<high; i++) {
       sum += array[i];
    }
    return sum;
  }
}

int main() {
  int* array = numa_alloc_blocksize(4096); // 4 memory pages
  array_sum(0, 4096);
}
```

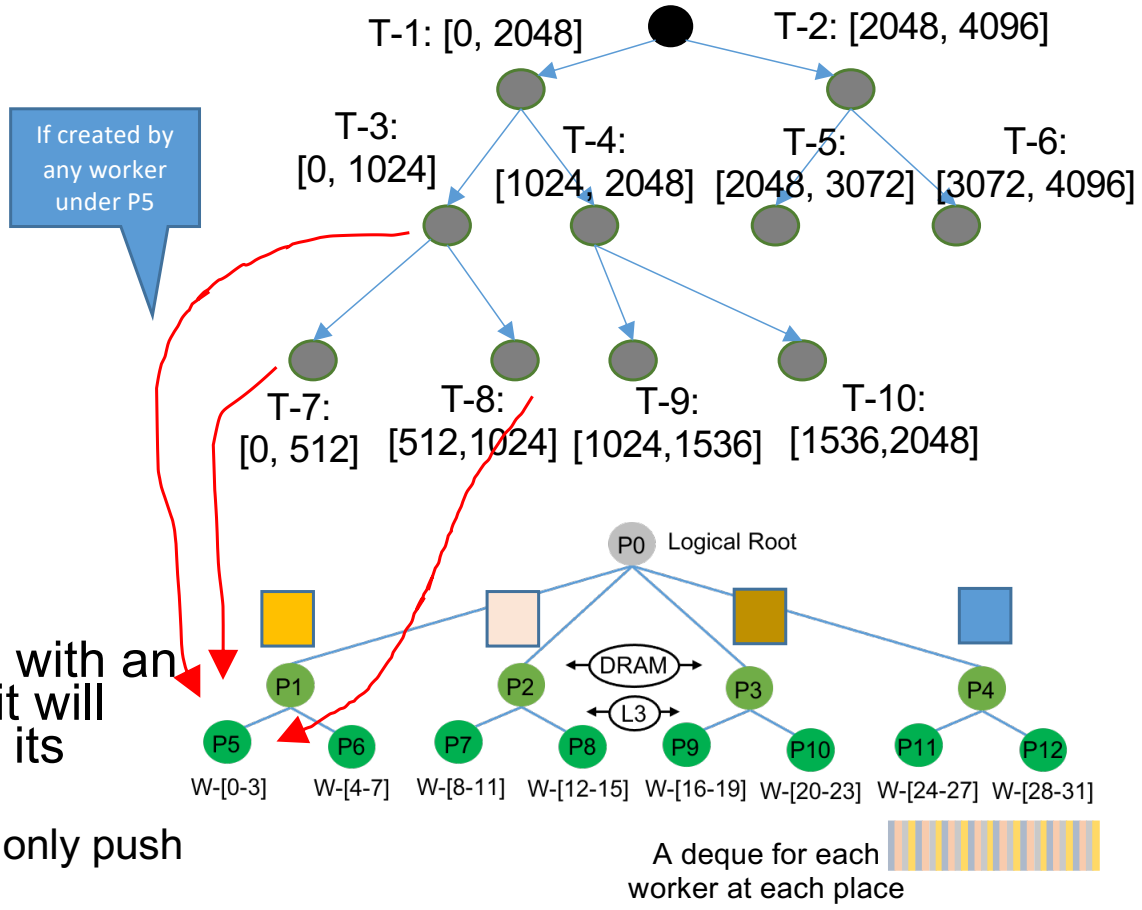


- If a worker executes an async_hinted that access memory range having affinity with a remote place, then it should push that task at the DRAM place of that remote NUMA node
  - If W0 create task T5, it would push it at place P3
  - If W31 creates task T10, it would push it place P2

Productivity & Performance: Recursive Programs

```
int array_sum(int low, int high) {
  if(high - low > 512) {
    int mid = (low + high)/2;
    future<int> left = async_hinted(array, low, mid, [=]() {
                           return array_sum(low, mid);
                       });
    future<int> right = async_hinted(array, mid, high, [=]() {
                           return array_sum(mid, high);
                        });
    return left.get() + right.get();
  } else {
    int sum = 0;
    for(int i=low; i<high; i++) {
       sum += array[i];
    }
    return sum;
  }
}

int main() {
  int* array = numa_alloc_blocksize(4096); // 4 memory pages
  array_sum(0, 4096);
}
```

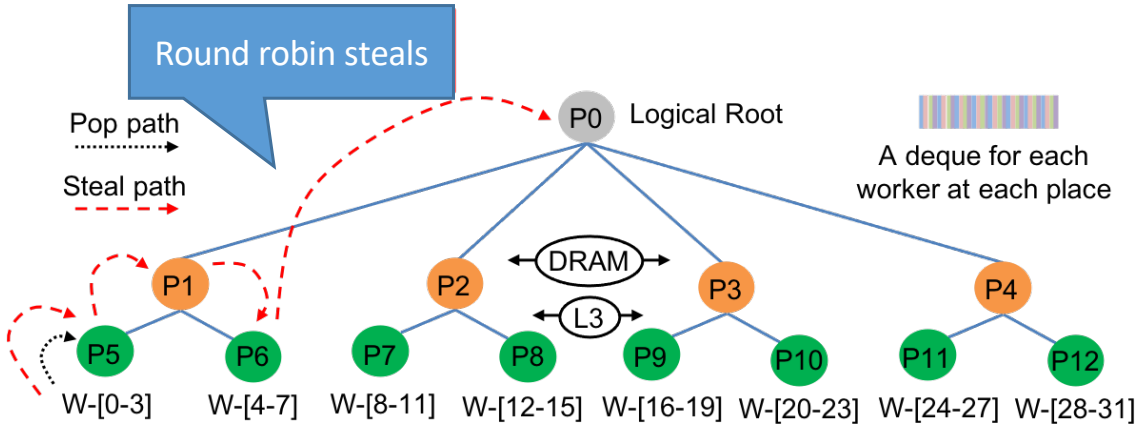


- Any worker creating an async_hinted with an affinity to its local DRAM place, then it will push this task ONLY into its deque at its cache place
  - If W0 create tasks T3, T7, and T8, it will only push them to its cache place P5

Productivity & Performance: Recursive Programs

```
int array_sum(int low, int high) {
  if(high - low > 512) {
    int mid = (low + high)/2;
    future<int> left = async_hinted(array, low, mid, [=]() {
                           return array_sum(low, mid);
                       });
    future<int> right = async_hinted(array, mid, high, [=]() {
                           return array_sum(mid, high);
                        });
    return left.get() + right.get();
  } else {
    int sum = 0;
    for(int i=low; i<high; i++) {
       sum += array[i];
    }
    return sum;
  }
}

int main() {
  int* array = numa_alloc_blocksize(4096); // 4 memory pages
  array_sum(0, 4096);
}
```



- Any worker can **pop** only from its leaf place (cache)
  - Recall, there will not be any tasks in its own deque at parent DRAM place as only remote workers can push task at its DRAM place
- Hierarchical **steals** within a NUMA domain and then from logical root
  - W0 at place P5 steal from all deques at places P5, P1, P6, and P0, respectively until successful
    - Strict locality without worker starvation

Reading Materials

- Hierarchical work-stealing
  - https://hal.inria.fr/inria-00429624v2/document
- async_hinted on NUMA architectures
  - https://vivkumar.github.io/papers/hipc2020.pdf

Next Lecture on 12/10 (L #13)

- No lectures on 01/10 (tomorrow) and 07/10
- Trace and replay of task parallel programs
- Quiz-2
  - Syllabus: Lectures 06-12