Lecture 09: Recursive Task Parallelism on NUMA Architecture

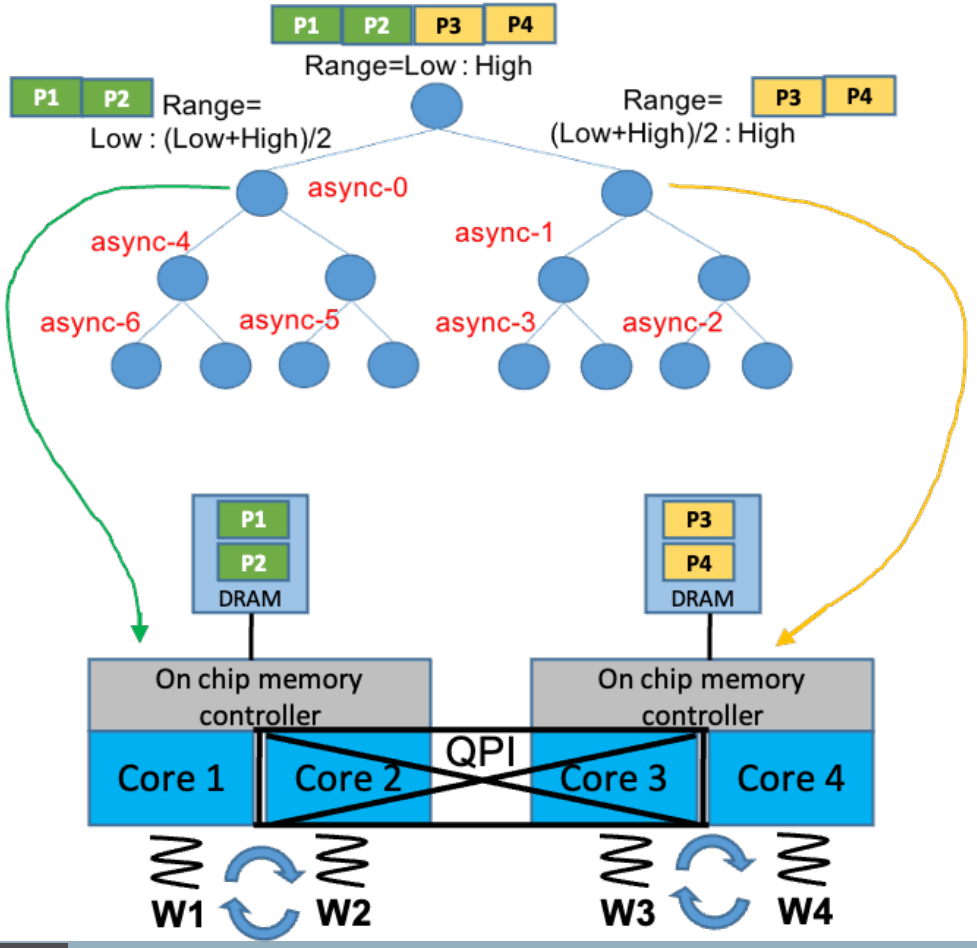
Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture (Recap)

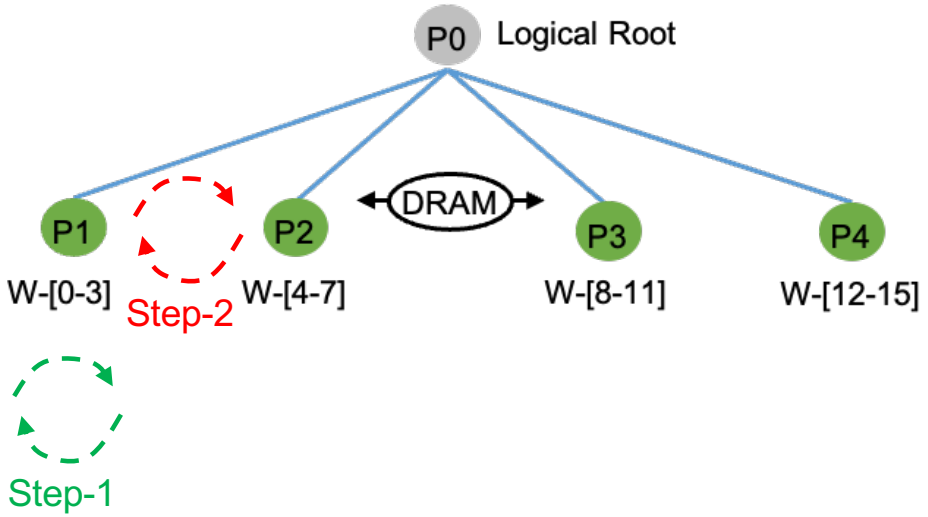


- NUMA aware work-stealing
  - Co-location of an async execution and its data on a NUMA domain can improve the locality
    - Better performance
  - Allocate seed task to each NUMA domain
  - Reduce stealing tasks across NUMA domains

Today's Lecture

- Assigning NUMA affinity to async tasks
- Quiz-2

Hierarchical Work-Stealing: How?



- How to improve the locality on modern NUMA processors?
  - Step-1: Give high preference for local stealing
  - Step-2: Try remote stealing only if local stealing failed
    - But it would hampers the data locality

Hierarchical Work-Stealing: Issues?

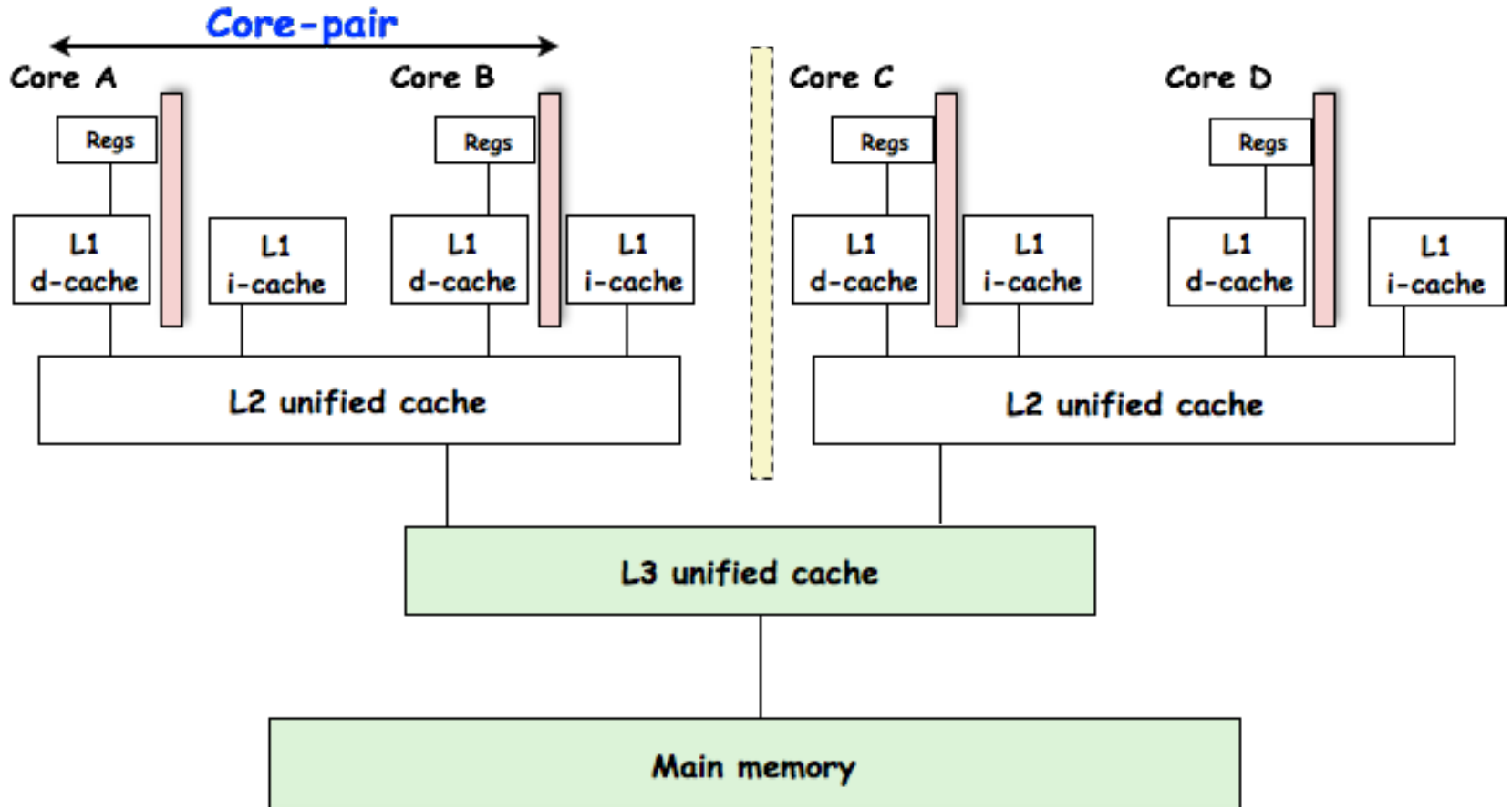
1. Performance
2. Productivity

Hierarchical Work-Stealing: Task Affinity

- The parallel programming constructs that we've studied thus far result in **async** that are assigned to processors *dynamically* by the HClib runtime system
  - Programmer does not worry about task assignment details
- Sometimes, programmer control of task assignment can lead to significant performance advantages due to improved locality
- Motivation for "places"
  - Provide the programmer a mechanism to restrict task execution to a subset of processors for improved locality

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec32-slides-v1.pdf?version=1&modificationDate=1483206145705&api=v2

Memory Hierarchy in a Multicore Processor

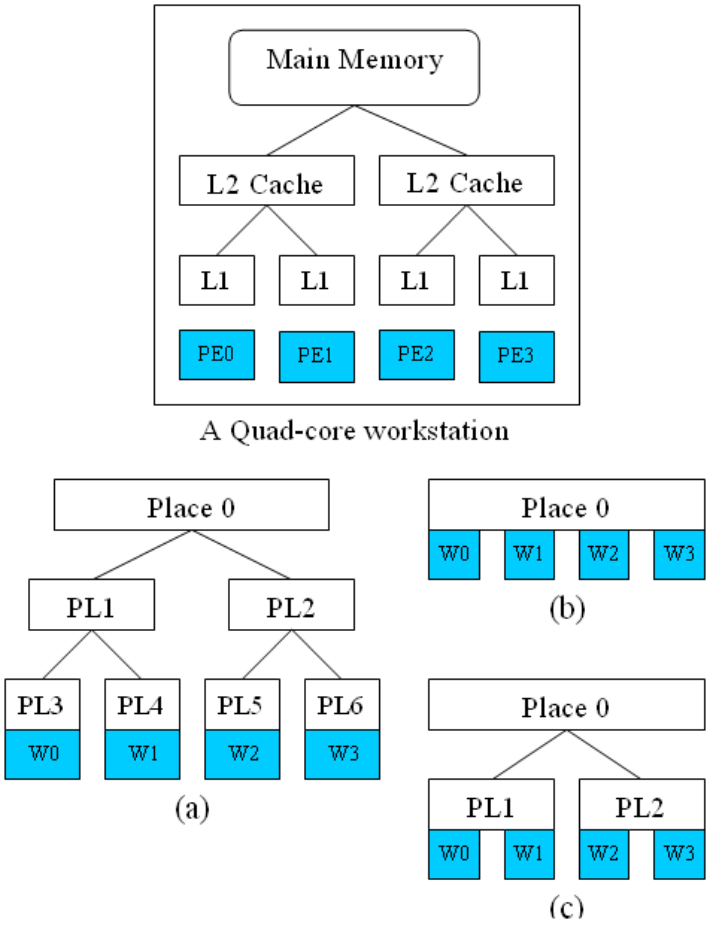


Hierarchical Place Trees

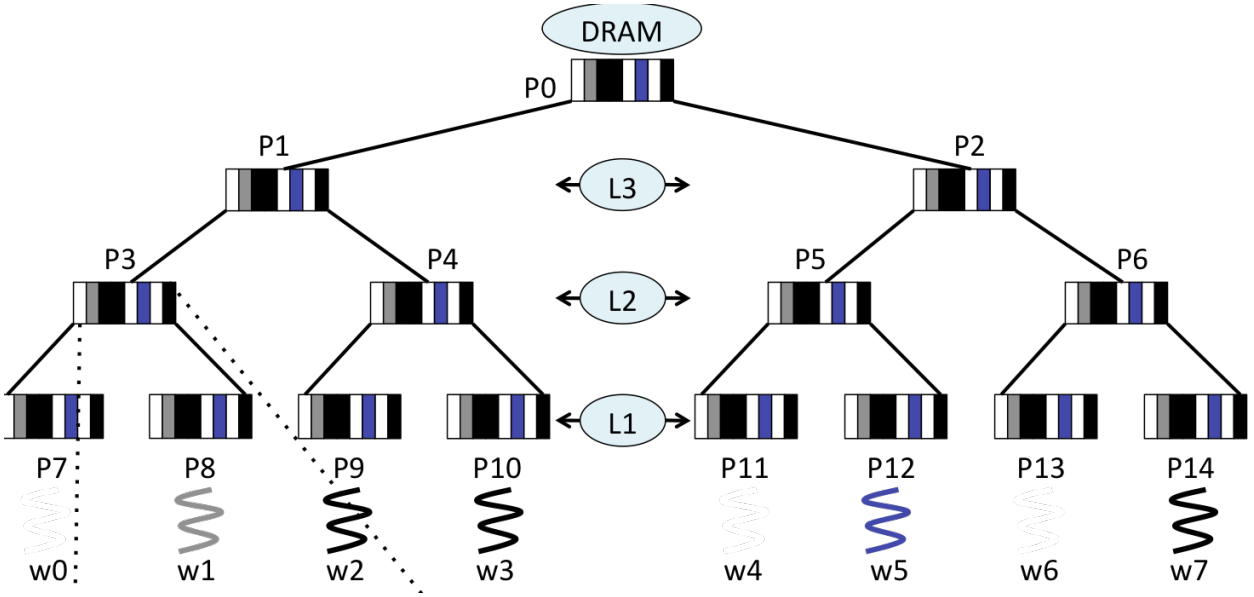
- Abstraction of the memory hierarchy that a program is executed on
  - Provided by the program using some input file
- *HPTs originally introduced by HClib*
- Place denoting affinity group at memory hierarchy level
  - E.g., L1 cache, L2 cache, DRAM
- Leaf places include worker threads
  - E.g., W0, W1, W2, W3
- Workers can push task to any place
  - asyncAtHPT(place*, lambda_function)
    - This is a programming feature provided to the programmer by which he can control the placement of the async tasks in different levels of memory hierarchy

Hierarchical Place Trees

Three different HPTs possible on this quad core processor



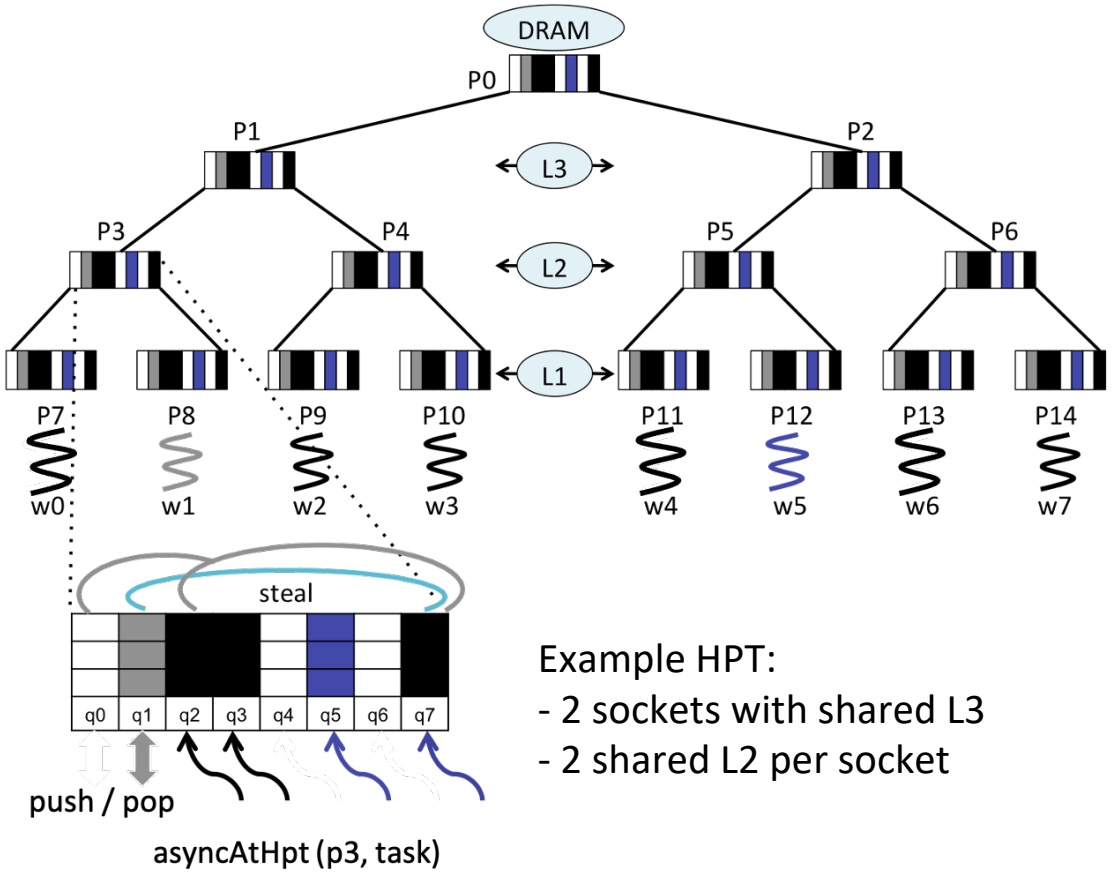
Starting an Async at Non Leaf HPT Node?



asyncAtHpt(P7, S1); asyncAtHpt(P9, S2); asyncAtHpt(P12, S3); asyncAtHpt(P14, S4);
asyncAtHpt(P2, S5);
asyncAtHpt(P4, S6);
asyncAtHpt(P6, S7);
asyncAtHpt(P0, S8);

Picture credit: Runtime Systems for Extreme Scale Platforms, PhD thesis, Sanjay Chatterjee, Rice University, 2013

Work-Stealing in a HPT



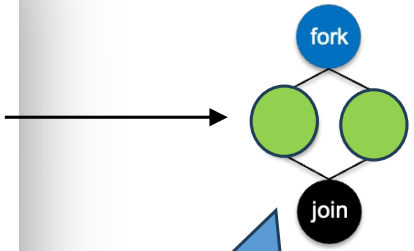
Example HPT:
- 2 sockets with shared L3
- 2 shared L2 per socket

- Round-robin steals instead of random work-stealing
- Workers attach to (own) leaf places
- Each place has one queue per worker
  - Ensures non-synchronized push and pop
- Any worker can push a task at any place
- Pop / steal access permitted to subtree workers
- Workers traverse path from leaf to root
- Tries to pop, then steal, at every place
- After successful pop / steal worker returns to leaf
- Worker threads are bound to cores

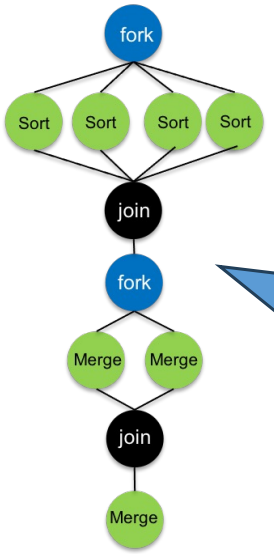
Picture credit: Runtime Systems for Extreme Scale Platforms, PhD thesis, Sanjay Chatterjee, Rice University, 2013

Recursive Task Parallelism on NUMA

```
1.  int *A;
2.  int ArraySum(int low, int high) {
3.    if((high-low)>LIMIT) {
4.      int mid = (high+low)/2, left, right;
5.      finish([&](){
6.        async([&](){ left = ArraySum(low, mid); });
7.        async([&](){ right = ArraySum(mid, high);});
8.      });
9.      return left+right;
10.   } else {
11.     int sum=0;
12.     for(int i=low; i<high; i++) {
13.       sum += A[i];
14.     }
15.     return sum;
16.   }
17. }
```



Regular DAG as the total number of children same at each fork point



How to generate the seed tasks for an irregular DAG, as the total number of children at each fork point isn't the same?

- Parallel recursive array sum generates two tasks inside each finish scope
  - Regular execution DAG
- To improve the performance, each NUMA node should get an equal size computation (seed task) at the start
- How to generate the seed task manually by the user?
  - Easy in case of 2-NUMA nodes, but what to do for 4-NUMA nodes?

Recursive Task Parallelism on NUMA

```
1.  int *A;
2.  int ArraySum(int low, int high) {
3.    if((high-low)>LIMIT) {
4.      int mid = (high+low)/2, left, right;
5.      finish([&](){
6.        async([&](){ left = ArraySum(low, mid); });
7.        async([&](){ right = ArraySum(mid, high);});
8.      });
9.      return left+right;
10.   } else {
11.     int sum=0;
12.     for(int i=low; i<high; i++) {
13.       sum += A[i];
14.     }
15.     return sum;
16.   }
17. }
```

```
1.  int *A;
2.  int ArraySum(int low, int high) {
3.    if((high-low)>LIMIT) {
4.      int mid = (high+low)/2, left, right;
5.      finish([&](){
6.        async_AtHPT(A, left, mid, [&](){ left = ArraySum(low, mid); });
7.        async_AtHPT(A, mid, high,[&](){ right = ArraySum(mid, high);});
8.      });
9.      return left+right;
10.   } else {
11.     int sum=0;
12.     for(int i=low; i<high; i++) {
13.       sum += A[i];
14.     }
15.     return sum;
16.   }
17. }
18. int main() {
19.   A = numa_alloc_blockcyclic<int>(N);
20.   initialize(A, N);
21.   int sum = ArraySum(0, high);
22.   numa_free(A);
23. }
```

Note that it is mandatory to use **asyncAtHPT** for each parallel tasks, i.e. for both left and right chunks

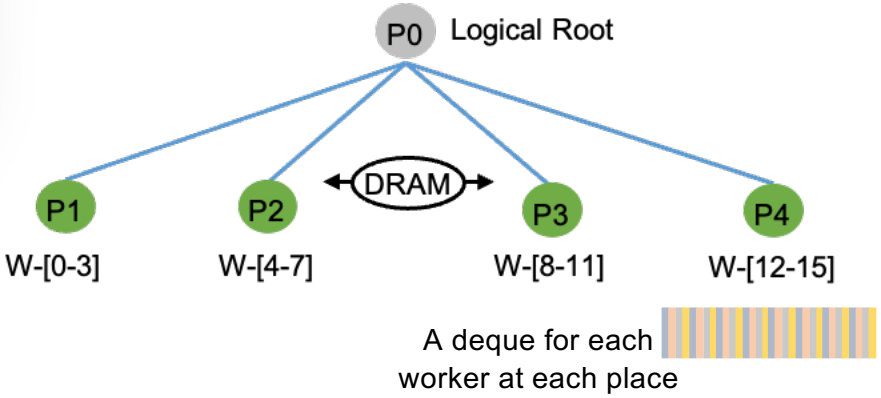
- Programmer allocates the array in block cyclic manner over all NUMA nodes (block size = array_size/num_numa_nodes)
- Programmer assign data-affinity hints to each async tasks. It must supply the starting address of the array being accessed and the range of array access
  - No program modification based on NUMA architecture
  - High productivity (Except for memory allocation/deallocation APIs)

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

Recursive Task Parallelism on NUMA

```
1.  int *A;
2.  int ArraySum(int low, int high) {
3.    if((high-low)>LIMIT) {
4.      int mid = (high+low)/2, left, right;
5.      finish([&](){
6.        async_AtHPT(A, left, mid, [&](){ left = ArraySum(low, mid); });
7.        async_AtHPT(A, mid, high,[&](){ right = ArraySum(mid, high);});
8.      });
9.      return left+right;
10.   } else {
11.     int sum=0;
12.     for(int i=low; i<high; i++) {
13.       sum += A[i];
14.     }
15.     return sum;
16.   }
17. }
18. int main() {
19.   A = numa_alloc_blockcyclic<int>(N);
20.   initialize(A, N);
21.   int sum = ArraySum(0, high);
22.   numa_free(A);
23. }
```

- 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

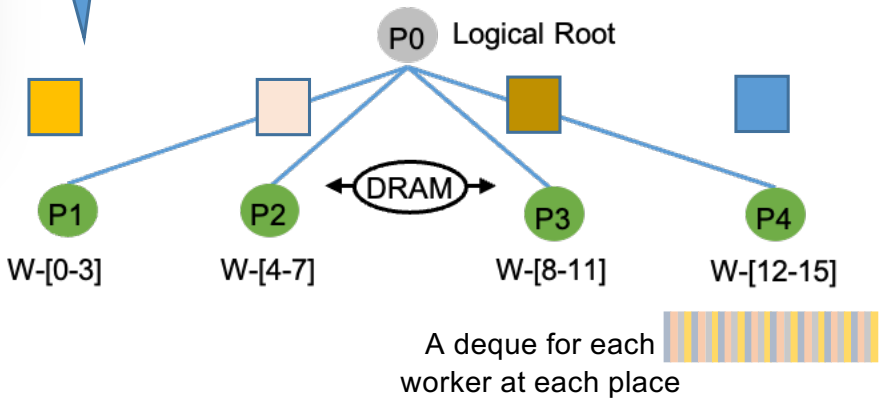


Recursive Task Parallelism on NUMA

```
1.  int *A;
2.  int ArraySum(int low, int high) {
3.    if((high-low)>LIMIT) {
4.      int mid = (high+low)/2, left, right;
5.      finish([&](){
6.        async_AtHPT(A, left, mid, [&](){ left = ArraySum(low, mid); });
7.        async_AtHPT(A, mid, high,[&](){ right = ArraySum(mid, high);});
8.      });
9.      return left+right;
10.   } else {
11.     int sum=0;
12.     for(int i=low; i<high; i++) {
13.       sum += A[i];
14.     }
15.     return sum;
16.   }
17. }
18. int main() {
19.   A = numa_alloc_blockcyclic<int>(N);
20.   initialize(A, N);
21.   int sum = ArraySum(0, high);
22.   numa_free(A);
23. }
```

Assume N=4096. Total pages required are 4

Block cyclic allocation of 4 pages on 4 NUMA nodes

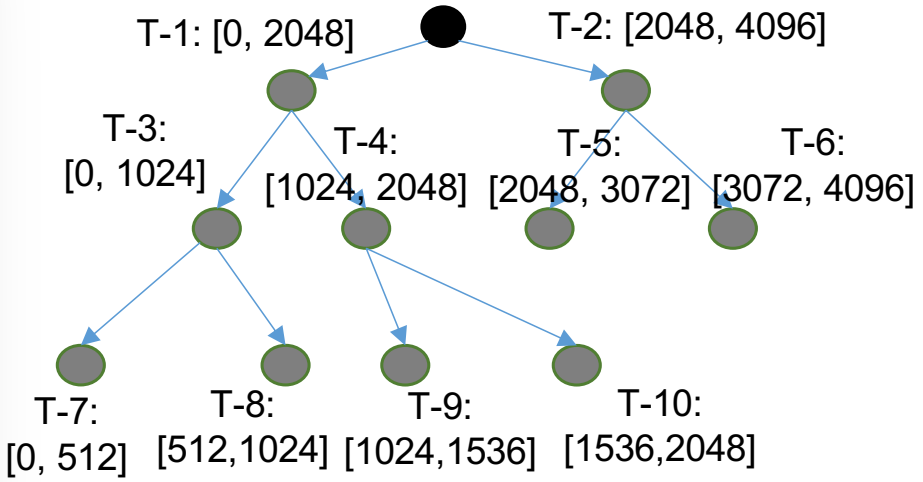


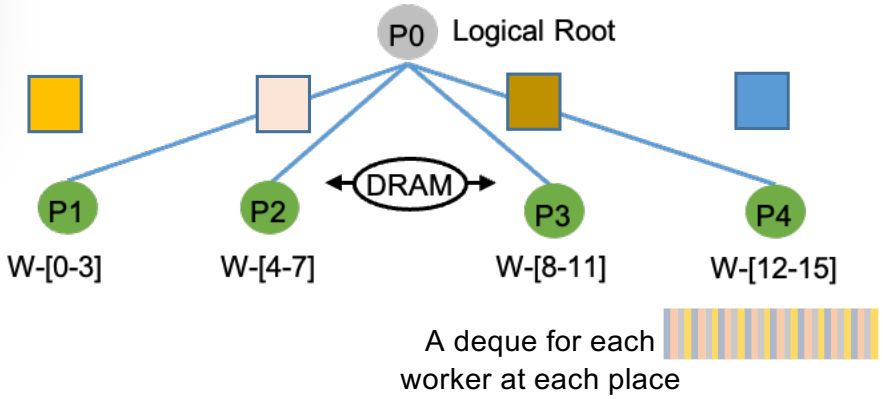
- Block cyclic allocation will ensure equal number (contiguous) of physical pages allocated under each NUMA domain
  - Place P1 will have affinity to memory residing at NUMA node represented by P1
    - Likewise for places P2, P3, and P4

Recursive Task Parallelism on NUMA

```
1.  int *A;
2.  int ArraySum(int low, int high) {
3.    if((high-low)>LIMIT) {
4.      int mid = (high+low)/2, left, right;
5.      finish([&](){
6.        async_AtHPT(A, left, mid, [&](){ left = ArraySum(low, mid); });
7.        async_AtHPT(A, mid, high,[&](){ right = ArraySum(mid, high);});
8.      });
9.      return left+right;
10.   } else {
11.     int sum=0;
12.     for(int i=low; i<high; i++) {
13.       sum += A[i];
14.     }
15.     return sum;
16.   }
17. }
18. int main() {
19.   A = numa_alloc_blockcyclic<int>(N);
20.   initialize(A, N);
21.   int sum = ArraySum(0, high);
22.   numa_free(A);
23. }
```

Assume LIMIT=512

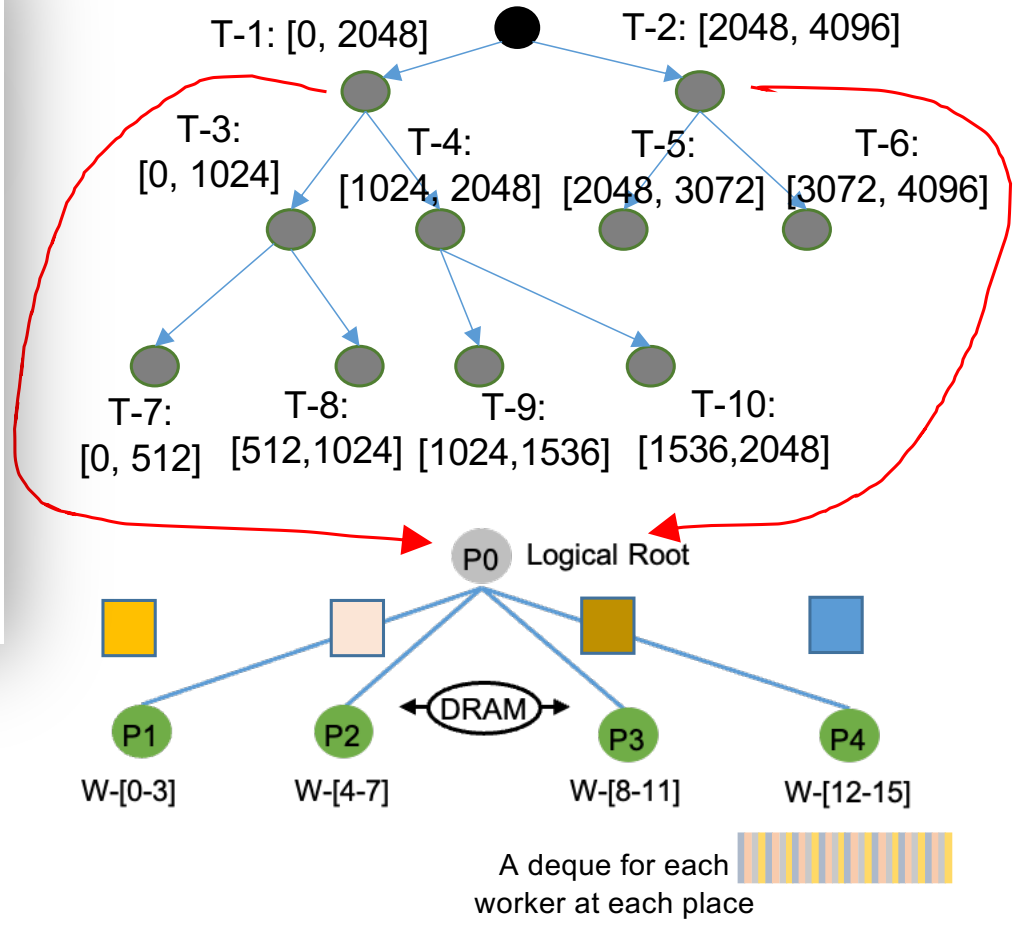




- Any worker executing an asyncAtHPT has to dynamically decide the place in the HPT where it should push this task (affinity of task)
- It calculates the palce affinity of a task using the range of memory being accessed inside this task

Recursive Task Parallelism on NUMA

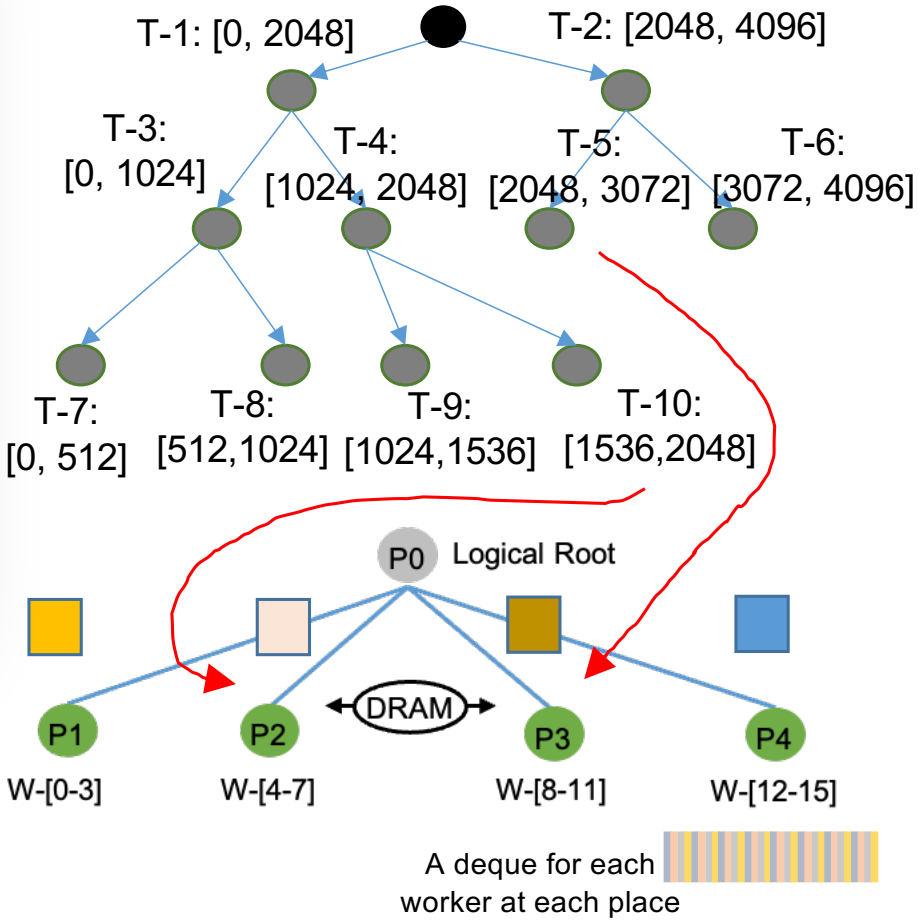
```
1.  int *A;
2.  int ArraySum(int low, int high) {
3.    if((high-low)>LIMIT) {
4.      int mid = (high+low)/2, left, right;
5.      finish([&](){
6.        async_AtHPT(A, left, mid, [&](){ left = ArraySum(low, mid); });
7.        async_AtHPT(A, mid, high,[&](){ right = ArraySum(mid, high);});
8.      });
9.      return left+right;
10.   } else {
11.     int sum=0;
12.     for(int i=low; i<high; i++) {
13.       sum += A[i];
14.     }
15.     return sum;
16.   }
17. }
18. int main() {
19.   A = numa_alloc_blockcyclic<int>(N);
20.   initialize(A, N);
21.   int sum = ArraySum(0, high);
22.   numa_free(A);
23. }
```



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

Recursive Task Parallelism on NUMA

```
1.  int *A;
2.  int ArraySum(int low, int high) {
3.    if((high-low)>LIMIT) {
4.      int mid = (high+low)/2, left, right;
5.      finish([&](){
6.        async_AtHPT(A, left, mid, [&](){ left = ArraySum(low, mid); });
7.        async_AtHPT(A, mid, high,[&](){ right = ArraySum(mid, high);});
8.      });
9.      return left+right;
10.   } else {
11.     int sum=0;
12.     for(int i=low; i<high; i++) {
13.       sum += A[i];
14.     }
15.     return sum;
16.   }
17. }
18. int main() {
19.   A = numa_alloc_blockcyclic<int>(N);
20.   initialize(A, N);
21.   int sum = ArraySum(0, high);
22.   numa_free(A);
23. }
```

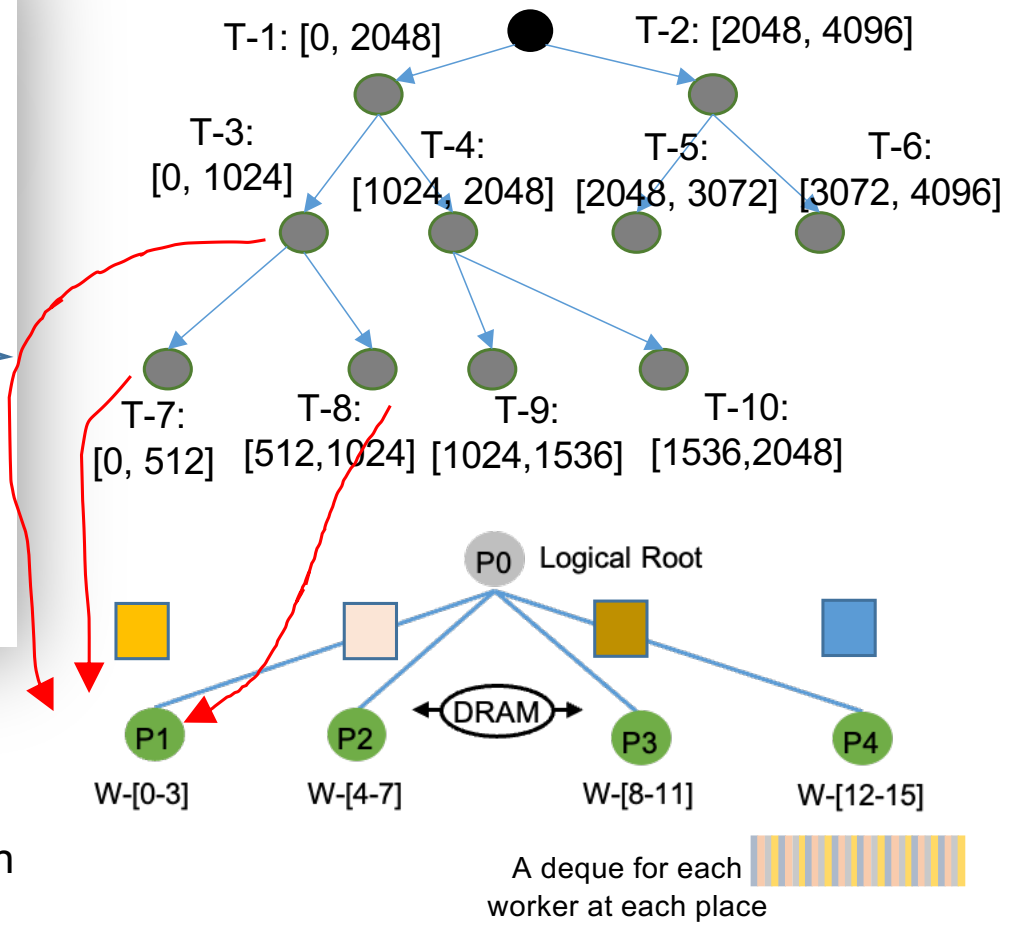


- If a worker executes an asyncAtHPT that access memory range having affinity with a remote place, then it should push that task at the DRAM place of that remote NUMA node
  - If W0 create task T5, it would push it at place P3
  - If W0 creates task T10, it would push it place P2

Recursive Task Parallelism on NUMA

```
1.  int *A;
2.  int ArraySum(int low, int high) {
3.    if((high-low)>LIMIT) {
4.      int mid = (high+low)/2, left, right;
5.      finish([&](){
6.        async_AtHPT(A, left, mid, [&](){ left = ArraySum(low, mid); });
7.        async_AtHPT(A, mid, high,[&](){ right = ArraySum(mid, high);});
8.      });
9.      return left+right;
10.   } else {
11.     int sum=0;
12.     for(int i=low; i<high; i++) {
13.       sum += A[i];
14.     }
15.     return sum;
16.   }
17. }
18. int main() {
19.   A = numa_alloc_blockcyclic<int>(N);
20.   initialize(A, N);
21.   int sum = ArraySum(0, high);
22.   numa_free(A);
23. }
```

If created by any worker under P5



- Any worker creating an asyncAtHPT with an affinity to its local DRAM place, then it will push this task ONLY into its deque at its leaf place
  - If W0 create tasks T3, T7, and T8, it will only push them to its leaf place P1

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 (#10)

- Trace and replay of task parallel programs