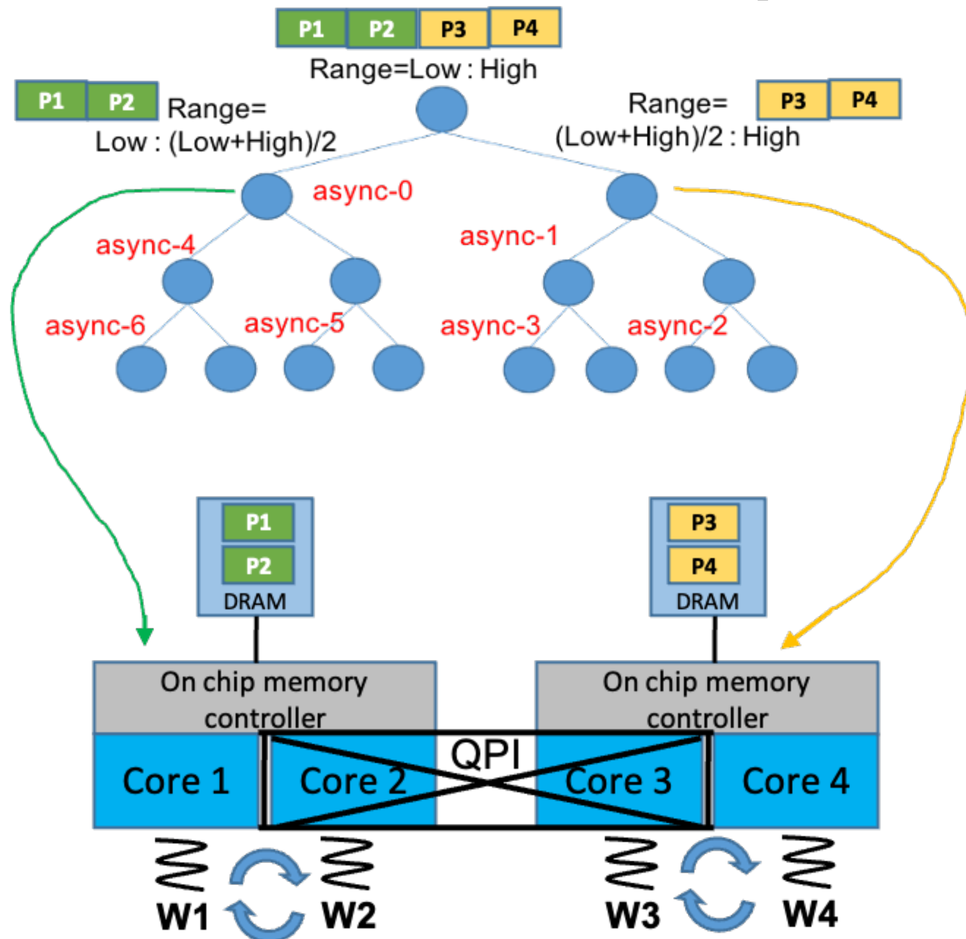
Lecture 12: Hierarchical Place Trees

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture (Recap)



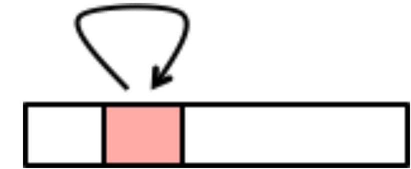
- NUMA aware work-stealing
  - Co-location of an async execution and its data on a NUMA domain can improve the locality
    - Better performance
  - Allocate seed task to each NUMA domain
  - Reduce stealing tasks across NUMA domains

Today's Lecture

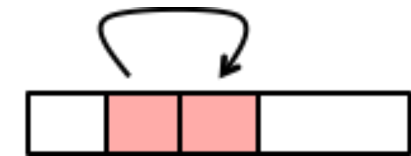
- Assigning NUMA affinity to async tasks
- Quiz-3

Locality

- Principal of Locality
  - Empirical observation: Processors tend to access same set or nearby memory locations repetitively over a short period of time



- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time

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

Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data references
  - Reference array elements in succession (stride-1 reference pattern) — Spatial locality
  - Reference variable sum each iteration — Temporal locality
- Instruction references
  - Reference instructions in sequence — Spatial locality
  - Cycle through loop repeatedly — Temporal locality

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

Iterative Averaging

```
//all elements zeroed
float A[SIZE+2], B[SIZE+2];

void iterative_averaging() {
  A[SIZE+1] = B[SIZE+1] = 1;
  for (int iter=0; iter<ITERATIONS; iter++) {
    for (int j=1; j<=SIZE; j++) {
      B[j] = (A[j-1] + A[j+1])/2.0;
    }
    double* temp = B;
    B = A;
    A = temp;
  }
}
```

https://classes.engineering.wustl.edu/cse231/core/index.php/Iterative_Averaging

- For SIZE=9, the array A will eventually converge with values as 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
- Used in ML to improve performance by averaging model parameters
- Used in scientific computing (e.g., heat diffusion simulation)

Spot the Locality

```
//all elements zeroed
float A[SIZE+2], B[SIZE+2];

void iterative_averaging() {
  A[SIZE+1] = B[SIZE+1] = 1;
  for (int iter=0; iter<ITERATIONS; iter++) {
    for (int j=1; j<=SIZE; j++) {
      B[j] = (A[j-1] + A[j+1])/2.0;
    }
    double* temp = B;
    B = A;
    A = temp;
  }
}
```

https://classes.engineering.wustl.edu/cse231/core/index.php/Iterative_Averaging

- Let us only consider the locality in array accesses
- Spatial locality
  - Inner for-loop
    - Reference array elements in succession (stride-1 reference pattern)
- Temporal locality
  - Outer for-loop
    - Accessing each arrays elements repeatedly

The async-finish Version

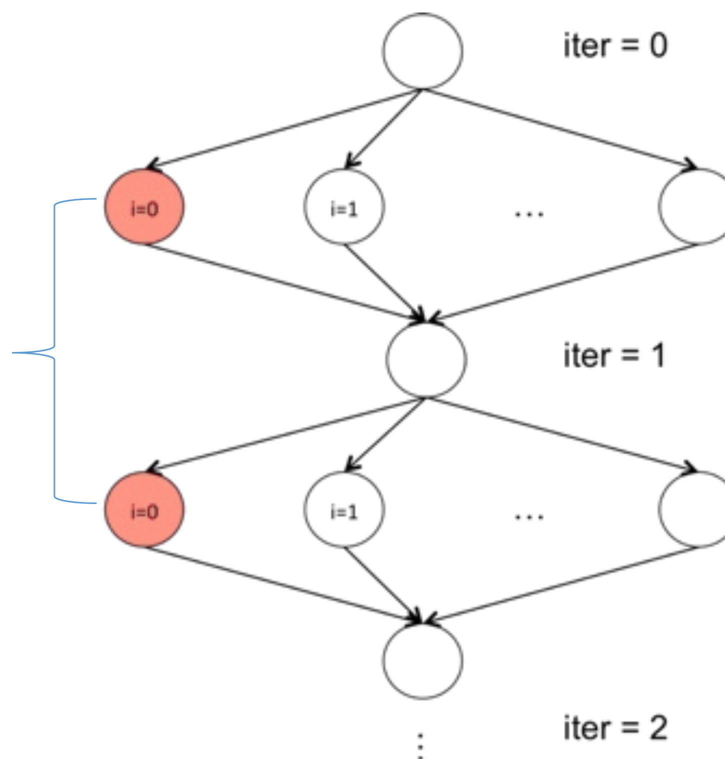
```
//all elements zeroed
float A[SIZE+2], B[SIZE+2];
void iterative_averaging() {
  int chunkSize = SIZE / num_workers();
  A[SIZE+1] = B[SIZE+1] = 1;
  for (int iter=0; iter<ITERATIONS; iter++) {
    finish([=]() {
      for(int chunk=0; chunk<num_workers(); chunk++) {
        async([=]() {
          int start = chunk*chunkSize + 1;
          int end = start + chunkSize – 1;
          for(int j=start; j<=end; j++) {
            B[j] = (A[j-1] + A[j+1])/2.0;
          }
        });
      }
    });
    double* temp = B;
    B = A;
    A = temp;
  }
}
```

Does it provide locality?

Code available on github:
https://github.com/vivkumar/cse502/blob/master/hclib/test/lec10/

Analyzing Locality Iterative Averaging

Locality benefits will be reached if all instances of chunk 0 execute on the same core and reuse data from the same cache



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

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

Task Affinity

- 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 (HPTs) as logical representation of the underlying NUMA topology

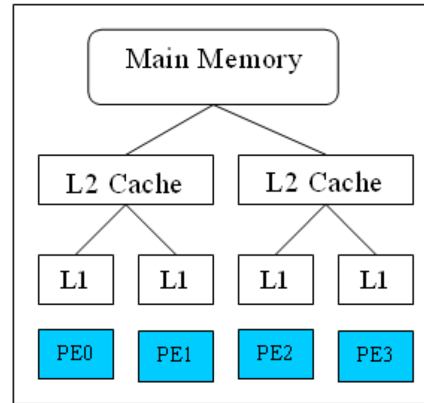
Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec32-slides-v1.pdf?version=1&modificationDate=1483206145705&api=v2

Hierarchical Place Trees

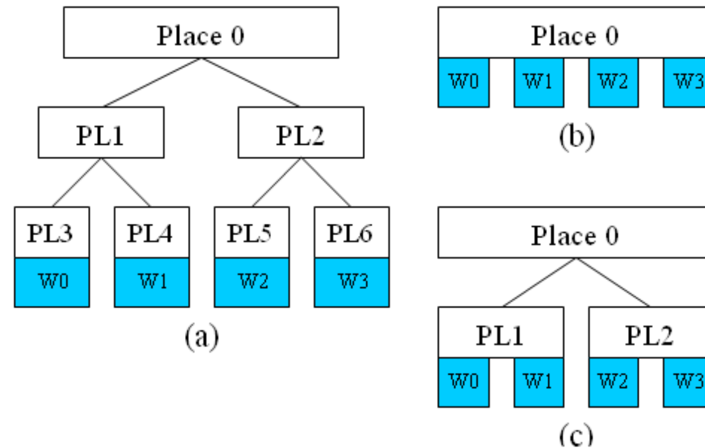
- Abstraction of the memory hierarchy that a program is executed on
  - Provided by the program using some input file
  - *HPTs originally introduced by HClib*
- Place denoting affinity group at memory hierarchy level
  - E.g., L1 cache, L2 cache, DRAM
- Leaf places include worker threads
  - E.g., W0, 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



A Quad-core workstation




```
<?xml version="1.0"?>
<!DOCTYPE HPT SYSTEM "hpt.dtd">
<HPT version="0.1" info="HPT test">
    <place num="1" type="mem">
        <place num="2" type="cache">
            <worker num=”2"/>
        </place>
    </place>
</HPT>
```

A sample XML input file to inform the HClib runtime about the underlying HPT (c)

Places in HClib

Some basic APIs in HClib for HPTs

place_t* get_current_place() //place at which current task is executing

int get_num_places(place_type_t type) // total number of places
// (runtime constant)

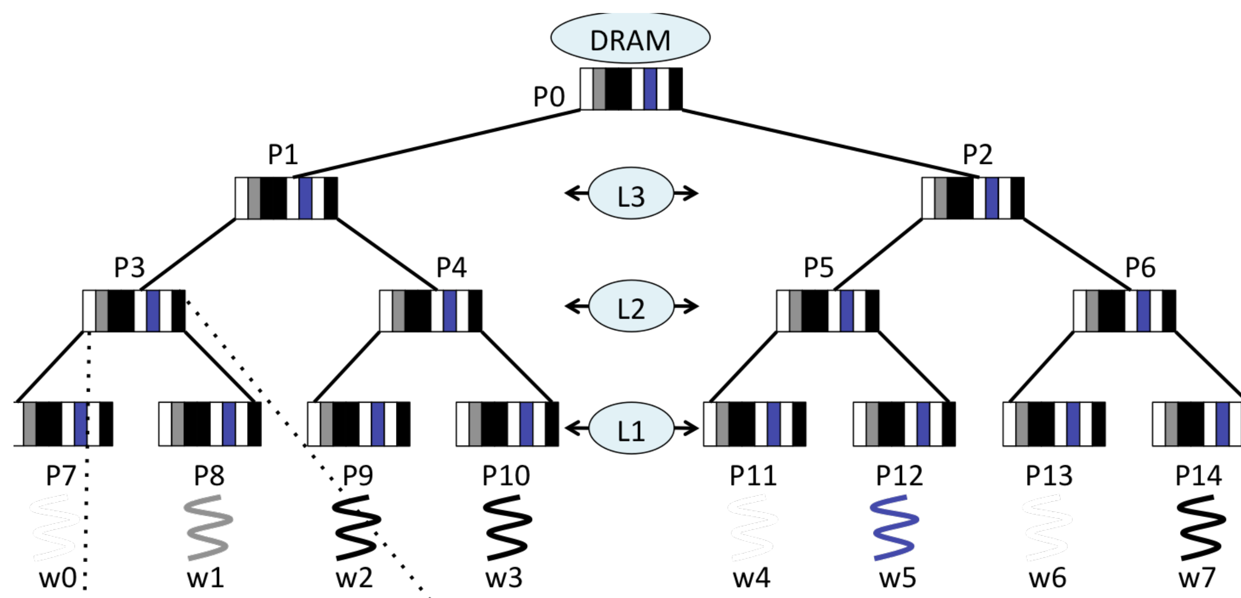
- type = CACHE_PLACE or MEM_PLACE (accelerator places coming up)

place_t* get_places(places_array, place_type_t type) // array of all
// places of "type"

asyncAtHpt(place_t*, S) // Creates new task to
// execute statement S at place P

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

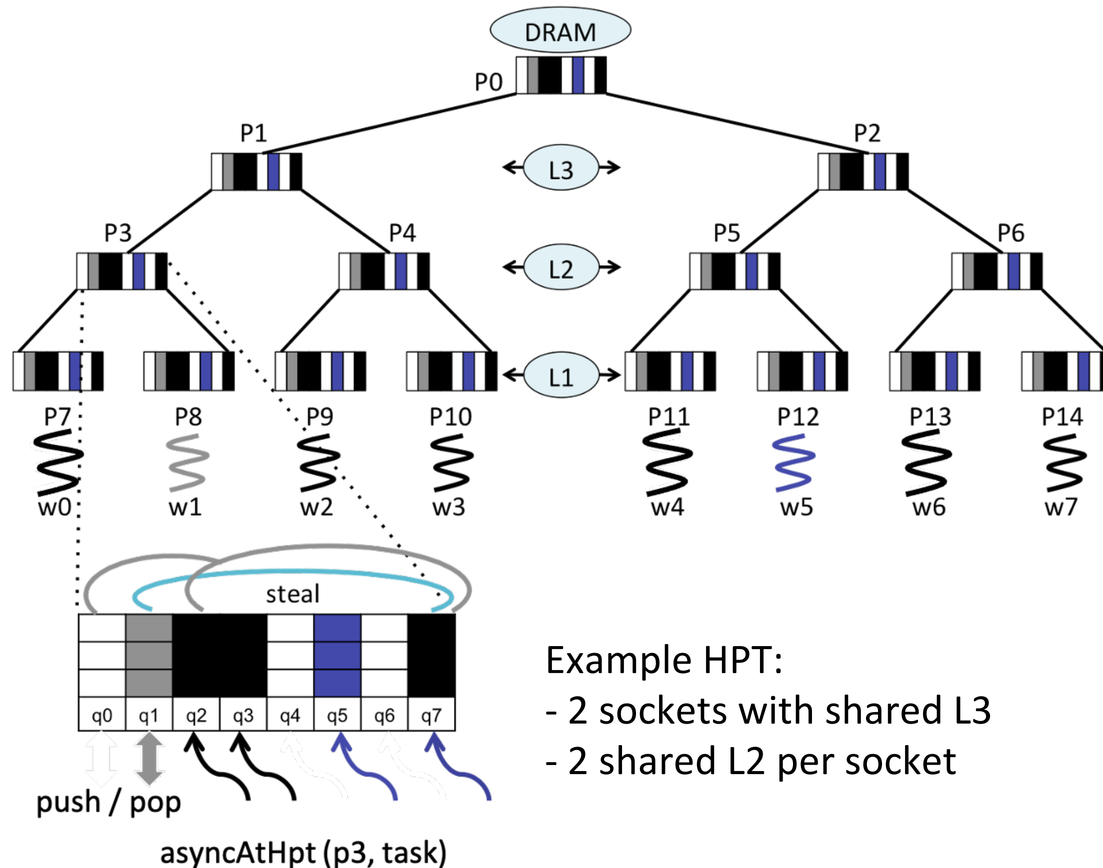
Starting an Async at a Place in HPT



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

Iterative Averaging with HPTs in HClib

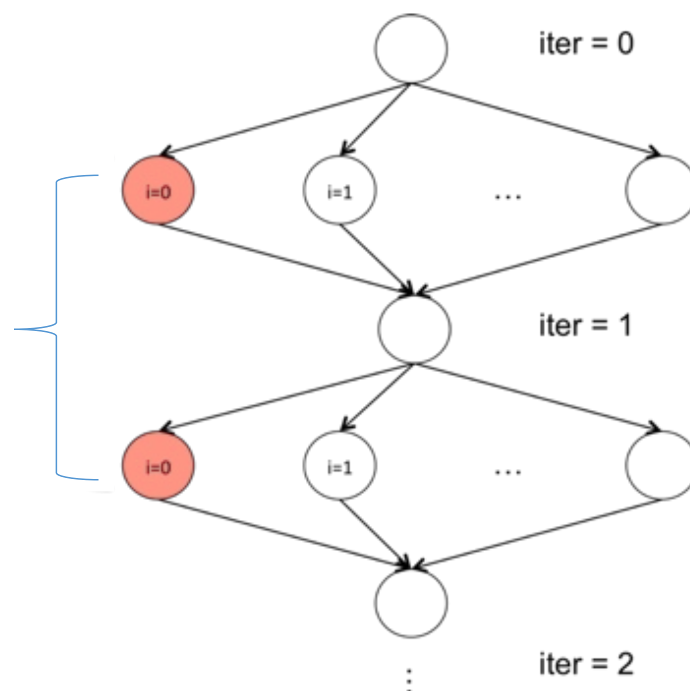
```
//all elements zeroed
float A[SIZE+2], B[SIZE+2];
void iterative_averaging() {
  int chunkSize = SIZE / num_workers();
  A[SIZE+1] = B[SIZE+1] = 1;

  for (int iter=0; iter<ITERATIONS; iter++) {
    finish([=]() {
      for(int chunk=0; chunk<num_workers(); chunk++) {
        asyncAtHpt(places[chunk], [=]() {
          int start = chunk*chunkSize + 1;
          int end = start + chunkSize - 1;
          for(int j=start; j<=end; j++) {
            B[j] = (A[j-1] + A[j+1])/2.0;
          }
        });
      }
    });
    double* temp = B;
    B = A;
    A = temp;
  }
}
```

```
int numPlace = get_num_places(place_type_t:: CACHE_PLACE);
place_t** places = new *place_t[numPlace];
get_places(place, place_type_t::CACHE_PLACE);
```

Code available on github:
https://github.com/vivkumar/cse502/blob/master/hclib/test/lec10/

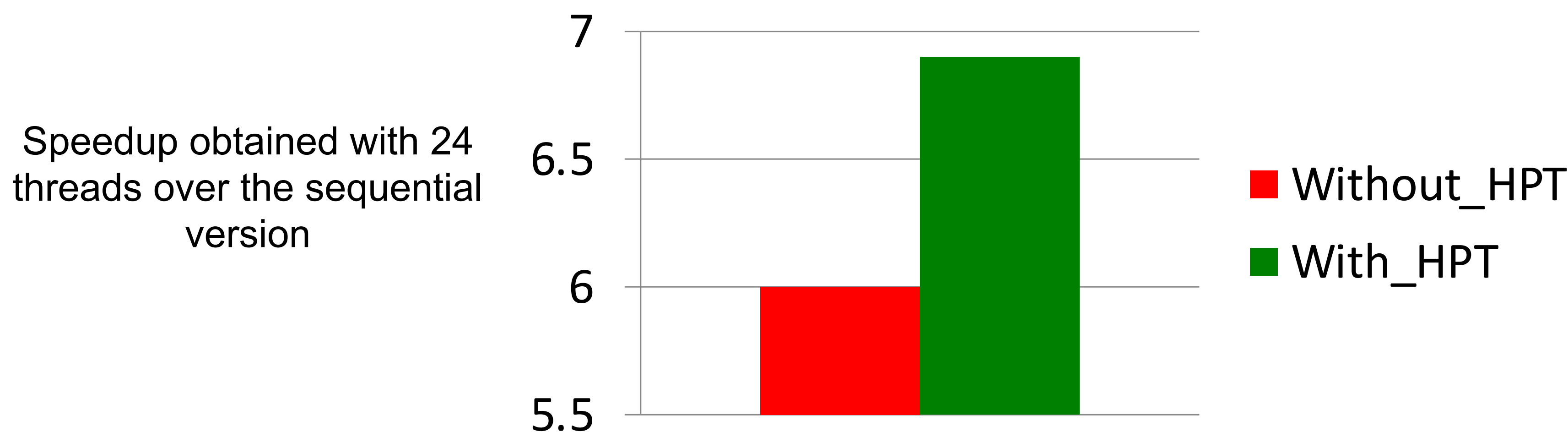
Analyzing Locality of Iterative Averaging with Places

Locality benefit is achieved as all instances of chunk 0 execute on the same core and reuse data from the same cache



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

Performance Analysis of 1D Iterative Averaging with/without HPT



Dual socket 6 core Intel E5-2667 processor with hyperthreading
Array size 3MB and total iterations=100

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

- Sequential overheads in work-stealing