Patterns of Parallelism

- Decomposition views
  - Data (static) vs. recursive (dynamic) decomposition
  - Exploratory decomposition vs. speculative decomposition
    - Exploratory - Parallel formulation may perform different amounts of work resulting in super or sub-linear speedup
    - Speculative - Schedule tasks even when they may have dependencies
- Data parallelism: all processors do the same thing on different data.
  - Regular
  - Irregular
- Task parallelism: processors do different tasks or dynamically pick up data to compute on
  - Task queue
  - Pipelines

44

Task Parallelism

- Each process performs a different task.
- Two principal flavors:
  - pipelines
  - task queues
- Program Examples: PIPE (pipeline), TSP (task queue).

45

Pipeline

- Often occurs with image processing applications, where a number of images undergo a sequence of transformations.
- E.g., rendering, clipping, compression, etc.

46

Sequential Program

```
for( i=0; i<num_pic, read(in_pic[i]); i++ ) {
  int_pic_1[i] = trans1( in_pic[i] );
  int_pic_2[i] = trans2( int_pic_1[i]);
  int_pic_3[i] = trans3( int_pic_2[i]);
  out_pic[i] = trans4( int_pic_3[i]);
}
```

47

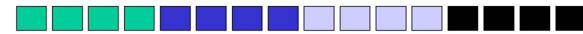
Parallelizing a Pipeline

- For simplicity, assume we have 4 processors (i.e., equal to the number of transformations).
- Furthermore, assume we have a very large number of pictures (>> 4).

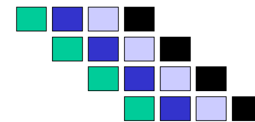
48

Sequential vs. Parallel Execution

- Sequential



- Parallel



(Color -- picture; horizontal line -- processor).

49

Parallelizing a Pipeline (part 1)

Processor 1:

```
for( i=0; i<num_pics, read(in_pic[i]); i++ ) {
  int_pic_1[i] = trans1( in_pic[i] );
  signal(event_1_2[i]);
}
```

50

Parallelizing a Pipeline (part 2)

Processor 2:

```
for( i=0; i<num_pics; i++ ) {
  wait( event_1_2[i] );
  int_pic_2[i] = trans2( int_pic_1[i] );
  signal(event_2_3[i] );
}
```

Same for processor 3

51

Parallelizing a Pipeline (part 3)

Processor 4:

```
for( i=0; i<num_pics; i++ ) {
  wait( event_3_4[i] );
  out_pic[i] = trans4( int_pic_3[i] );
}
```

52

Another Sequential Program

```
for( i=0; i<num_pic, read(in_pic); i++ ) {
  int_pic_1 = trans1( in_pic );
  int_pic_2 = trans2( int_pic_1);
  int_pic_3 = trans3( int_pic_2);
  out_pic = trans4( int_pic_3);
}
```

53

Can we use same parallelization?

Processor 2:

```
for( i=0; i<num_pics; i++ ) {
  wait( event_1_2[i] );
  int_pic_2 = trans1( int_pic_1 );
  signal(event_2_3[i] );
}
```

Same for processor 3

54

Can we use same parallelization?

- No, because of anti-dependence between stages, there is no parallelism
- Another example of privatization
- Costly in terms of memory

55

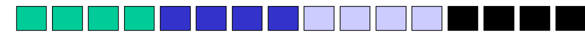
In-between Solution

- Use n>1 buffers between stages.
- Block when buffers are full or empty

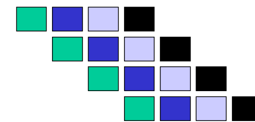
56

Perfect Pipeline

- Sequential



- Parallel



(Color -- picture; horizontal line -- processor).

57

Things are often not that perfect

- One stage takes more time than others
- Stages take a variable amount of time
- Extra buffers can provide some cushion against variability

58

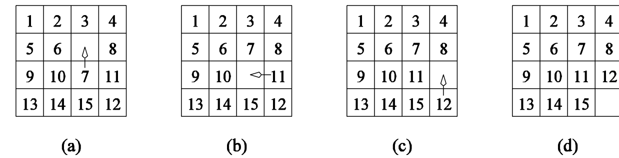
59

Patterns of Parallelism

- Decomposition views
  - Data (static) vs. recursive (dynamic) decomposition
  - Exploratory decomposition vs. speculative decomposition
    - Exploratory - Parallel formulation may perform different amounts of work resulting in super or sub-linear speedup
    - Speculative - Schedule tasks even when they may have dependencies
- Data parallelism: all processors do the same thing on different data.
  - Regular
  - Irregular
- Task parallelism: processors do different tasks or dynamically pick up data to compute on
  - Task queue
  - Pipelines

Exploratory Decomposition

Example: A 15-tile puzzle

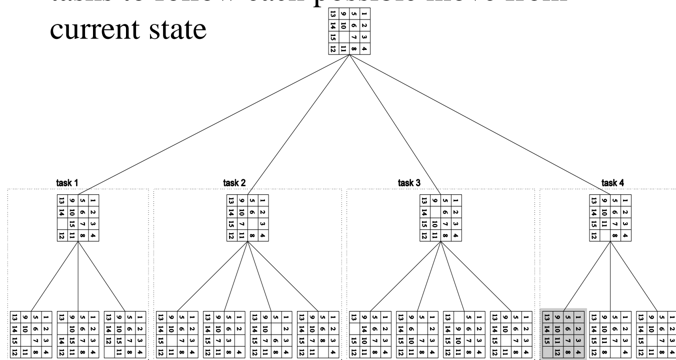


Sequence of 3 moves leads from initial state (a) to final state (d)

In general: explore all possible moves to arrive at solution

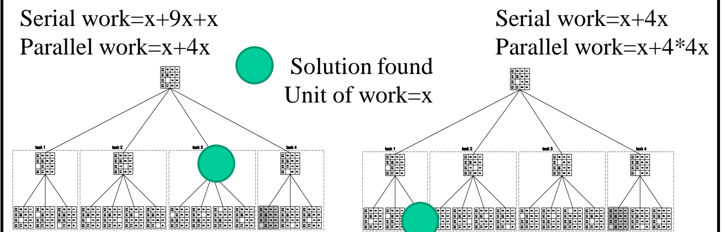
Exploratory Decomposition

Explore state space by creating independent tasks to follow each possible move from current state



Exploratory Decomposition Speedup

- Parallel formulation may perform more or less work depending on when solution is found
  - Superlinear or sublinear speedup



TSP (Traveling Salesman)

- Goal:
  - given a list of cities, a matrix of distances between them, and a starting city,
  - find the shortest tour in which all cities are visited exactly once.
- Example of an NP-hard search problem.
- Algorithm: branch-and-bound.

64

Branching

- Initialization:
  - go from starting city to each of remaining cities
  - put resulting partial path into priority queue, ordered by its current length.
- Further (repeatedly):
  - take head element out of priority queue,
  - expand by each one of remaining cities,
  - put resulting partial path into priority queue.

65

Finding the Solution

- Eventually, a complete path will be found.
- Remember its length as the current shortest path.
- Every time a complete path is found, check if we need to update current best path.
- When priority queue becomes empty, best path is found.

66

Using a Simple Bound

- Once a complete path is found, we have a lower bound on the length of shortest path
- No use in exploring partial path that is already longer than the current lower bound
- Better bounding methods exist …

67

Sequential TSP: Data Structures

- Priority queue of partial paths.
- Current best solution and its length.
- For simplicity, we will ignore bounding.

68

Sequential TSP: Code Outline

```
init_q(); init_best();
while( (p=de_queue()) != NULL ) {
  for each expansion by one city {
        q = add_city(p);
        if( complete(q) ) { update_best(q) };
        else { en_queue(q) };
  }
}
```

69

Parallel TSP: Possibilities

- Have each process do one expansion
- Have each process do expansion of one partial path
- Have each process do expansion of multiple partial paths
- Issue of granularity/performance, not an issue of correctness.
- Assume: process expands one partial path.

70

Parallel TSP: Synchronization

- True dependence between process that puts partial path in queue and the one that takes it out.
- Dependences arise dynamically.
- Required synchronization: need to make process wait if q is empty.

71

Parallel TSP: First Cut (part 1)

```
process i:
    while( (p=de_queue()) != NULL ) {
      for each expansion by one city {
            q = add_city(p);
            if complete(q) { update_best(q) };
            else en_queue(q);
      }
    }
```

72

Parallel TSP: First cut (part 2)

- In de_queue: wait if q is empty
- In en_queue: signal that q is no longer empty

73

Parallel TSP

```
process i:
    while( (p=de_queue()) != NULL ) {
      for each expansion by one city {
            q = add_city(p);
            if complete(q) { update_best(q) };
            else en_queue(q);
      }
    }
```

74

Parallel TSP: More synchronization

- All processes operate, potentially at the same time, on q and best.
- This must not be allowed to happen.
- Critical section: only one process can execute in critical section at once.

75

Parallel TSP: Critical Sections

- All shared data must be protected by critical section.
- Update_best must be protected by a critical section.
- En_queue and de_queue must be protected by the same critical section.

76

Parallel TSP

```
process i:
    while( (p=de_queue()) != NULL ) {
      for each expansion by one city {
            q = add_city(p);
            if complete(q) { update_best(q) };
            else en_queue(q);
      }
    }
```

77

Termination condition

- How do we know when we are done?
- All processes are waiting inside de_queue.
- Count the number of waiting processes before waiting.
- If equal to total number of processes, we are done.

78

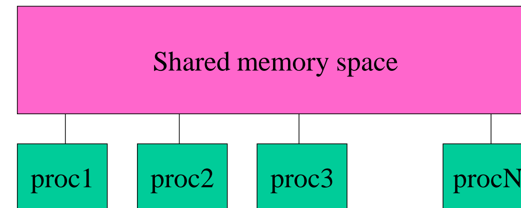
79

Programming Models

- Standard models of parallelism
  - shared memory (Pthreads)
  - message passing (MPI)
  - data parallel (Fortran 90 and HPF)
  - shared memory + data parallel (OpenMP)
  - Remote procedure call
  - Global address space (UPC)

80

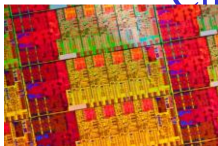
Shared Memory



81

The Performance Transparency Challenge

Modern multicore systems…



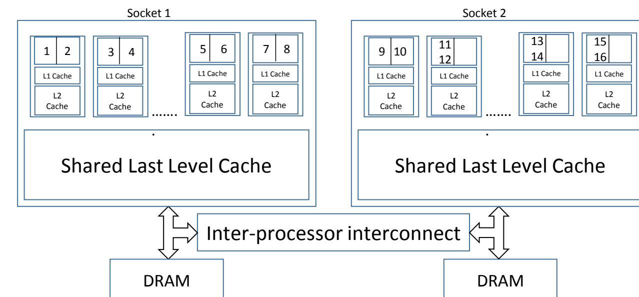
Source: http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested/5

10s to 100s of hardware contexts

- Shared hardware resource access
  - Functional units, caches, on- and off-chip interconnects, memory
- Shared software resource access
  - E.g., locks or shared data
- Non-uniform access latencies

82

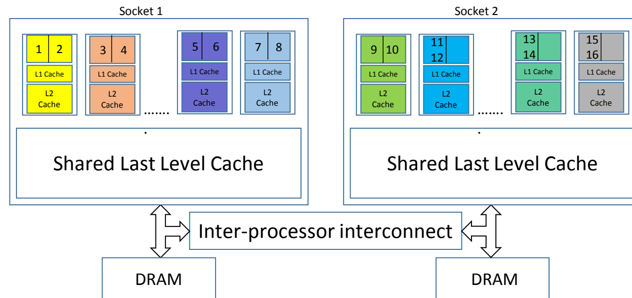
A Data-Centric View of a Modern Multicore



83

Performance Transparency Challenge: Resource Sharing/Contention

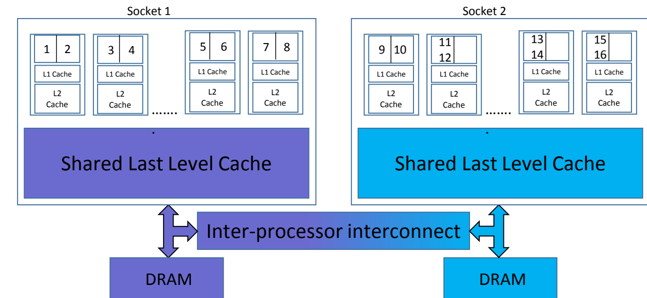
Problem: Simultaneous multi-threading



84

Performance Transparency Challenge: Resource Sharing/Contention

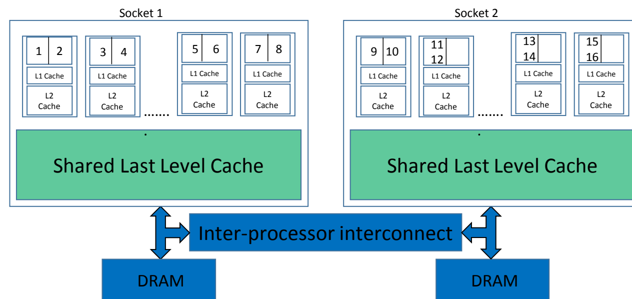
Problem: Intra-socket resource sharing



85

Performance Transparency Challenge: Resource Sharing/Contention

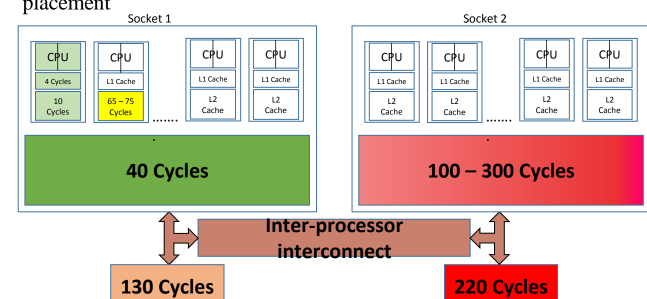
Problem: Inter-socket resource sharing



86

Performance Transparency Challenge: Non-Uniform Access Latencies

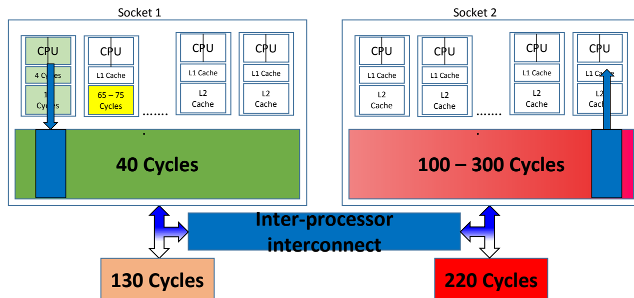
Problem: Communication costs a function of thread/data placement



Ref: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

87

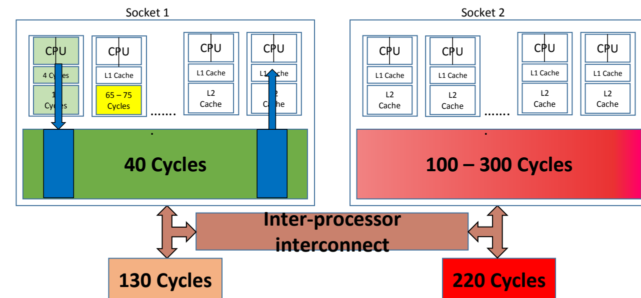
Impact of Thread Placement on Data Sharing Costs



Ref: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

88

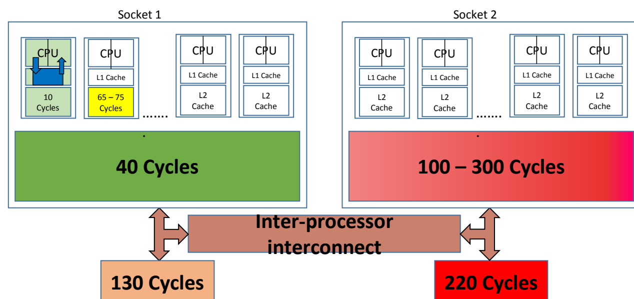
Impact of Thread Placement on Data Sharing Costs



Ref: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

89

Impact of Thread Placement on Data Sharing Costs



Ref: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

90

Acknowledgements

Slides reflect content from Willy Zwaenepoel and from Grama/Gupta/Karypis/Kumar that accompany their corresponding course/textbooks and have been adapted to suit the content of this course

155