COMP 790-033  - Parallel Computing

Lecture 3
Aug 31, 2022

SMM (1)
Shared Memory Multiprocessors
Topics

• Memory systems
  – organization
  – caches and the memory hierarchy
  – influence of the memory hierarchy on algorithms

• Shared memory systems
  – Taxonomy of actual shared memory systems
    • UMA, NUMA, cc-NUMA

• OpenMP shared memory parallel programming
Recall PRAM shared memory system

- **PRAM model**
  - assumes access latency is constant, regardless of the number of processors or the size of memory
  - simultaneous reads permitted under CR model and simultaneous writes permitted under CW model

- **Physically impossible to realize**
  - processors and memory occupy physical space
    - speed of light limitations
      \[ L = \Omega((p + m)^{1/3}) \]
    - CR / CW must be reduced to ER / EW
      - requires \( \Omega(lg p) \) time in general case
Anatomy of a processor ↔ memory system

- Performance parameters of Random Access Memory (RAM)
  - latency $L$
    - elapsed time from presentation of memory address to arrival of data
      - address transit time
      - memory access time $t_{mem}$
      - data transit time
  - bandwidth $W$
    - number of values (e.g. 64 bit words) delivered to processor per unit time
      - simple implementation $W \sim 1/L$
Processor vs. memory performance

- The memory “wall”
  - Processors compute faster than memory delivers data
    - Increasing imbalance \( t_{\text{arith}} \ll t_{\text{mem}} \)
Improving memory system performance

- Decrease latency $L$ to memory
  - speed of light is a limiting factor
    - bring memory closer to processor

- Decrease memory access time by using 2D memory layout
  - access time $\propto s^{1/2}$ (VLSI)

- Use different memory technologies
  - DRAM (Dynamic RAM) 1 transistor per stored bit
    - High density, low power, low cost, but long access time
  - SRAM (Static RAM) 6 transistors per stored bit
    - Short access time, but low density, high power, and high cost.
Improving memory system performance (1)

- **Decrease latency** using cache memory
  - low latency access to frequently used values, high latency for the remaining values

  ![Processor Memory Cache Diagram]

- **Example**
  - 90% of references are to cache with latency $L_1$
  - 10% of references are to memory with latency $L_2$
  - average latency is $0.9L_1 + 0.1L_2$
Improving memory system performance (2)

- Increase bandwidth W
  - multiport (parallel access) memory
    - multiple reads, multiple exclusive writes per memory cycle
      - High cost, very limited scalability

- “blocked” memory
  - memory supplies block of size b containing requested word
    - supports spatial locality in cache access
Improving memory system performance (2)

• Increase bandwidth $W$ (contd)
  - pipeline memory requests
    • requires *independent* memory references
  - interleave memory
    • problem: memory access is limited by $t_{\text{mem}}$
    • use $m$ separate memories (or memory banks)
    • $W \sim \frac{m}{L}$ if references *distribute* over memory banks
Latency hiding

- **Amortize latency using a pipelined interleaved memory system**
  - \( k \) independent references in \( \Omega(L + k \cdot t_{proc}) \) time
    - \( O(L/k) \) amortized (expected) latency per reference

- **Where do we get independent references?**
  - out-of-order execution of independent load/store operations
    - found in most modern performance-oriented processors
    - partial latency hiding: \( k \sim 2 - 10 \) references outstanding
  - vector load/store operations
    - small vector units (AVX512)
      - vector length 2-8 words (Intel Xeon)
      - partial latency hiding
    - high-performance vector units (NEC SX-9, SX-Aurora)
      - vector length \( k = L / t_{proc} \) (128 - 256 words)
      - crossbar network to highly interleaved memory (~ 16,000 banks)
      - full latency hiding: amortized memory access at processor speed
  - multithreaded operation
    - independent execution threads with individual hardware contexts
      - partial latency hiding: 2-way hyperthreading (Intel)
      - full latency hiding: 128-way threading with high-performance memory (Cray MTA)
Implementing the PRAM

• How close can we come to $O(1)$ latency PRAM memory in practice?

  – requires processor to memory network
  • latency $L = $ sum of
    – twice network latency
    – memory cycle time
    – serialization time for CR, CW
  • $L$ increases with $m$, $p$
    – $L$ too large with current technology

  – examples
    – logarithmic depth combining network eliminates memory contention time for CR, CW
      » $\Omega(\lg p)$ latency in network is prohibitive
Implementing PRAM – a compromise

• Using latency hiding with a high-performance memory system
  – implements $p \cdot k$ processor EREW PRAM slowed down by a factor of $k$
    • use $m \geq p \left( \frac{t_{\text{mem}}}{t_{\text{proc}}} \right)$ memory banks to match memory reference rate of $p$ processors
    • total latency $2L$ for $k = L / t_{\text{proc}}$ independent random references at each processor
    • $O(t_{\text{proc}})$ amortized latency per reference at each processor

  – unit latency degrades in the presence of concurrent reads/writes

  ![Diagram showing network connection with multiple memory banks and processors](image)

  – Bottom line: doable but very expensive and only limited scaling in $p$
Memory systems summary

• Memory performance
  – Latency is limited by physics
  – Bandwidth is limited by cost

• Cache memory: low latency access to some values
  – caching frequently used values
    • rewards *temporal locality* of reference
  – caching consecutive values
    • rewards *spatial locality* of reference
  – decrease *average* latency
    • 90 fast references, 10 slow references: effective latency = 0.9L_1 + 0.1L_2

• Parallel memories
  – 100 *independent* references ≈ 100 fast references
  – relatively expensive
  – requires parallel processing
Simple uniprocessor memory hierarchy

- Each component is characterized by
  - capacity
  - block size
  - (associativity)

- Traffic between components is characterized by
  - access latency
  - transfer rate (bandwidth)

- Example:
  - IBM RS6000/320H (ca. 1991)

<table>
<thead>
<tr>
<th>Storage component</th>
<th>Latency (cycles)</th>
<th>Transfer Rate (words [8B] / cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk</td>
<td>1,000,000</td>
<td>0.001</td>
</tr>
<tr>
<td>Main memory</td>
<td>60</td>
<td>0.1</td>
</tr>
<tr>
<td>Cache</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Registers</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Cache operation

- ABC cache parameters
  - associativity
  - block size
  - capacity

- CCC performance model
  - cache misses can be
    - compulsory
    - capacity
    - conflict
Cache operation: read

- Associativity: 256-way
- Block size: 64 bytes (512b)

40-bit address

- Tag
- Index
- blk

Valid
<1>

Tag
<26>

Data
<512>

MUX

Address

Data

1,2,4,8 bytes
Cache basics

• Five basic cache optimizations:
  – Larger block size
    • Reduces compulsory misses
    • Increases capacity and conflict misses, increases miss penalty
  – Larger total cache capacity to reduce miss rate
    • Increases latency, increases power consumption
  – Higher associativity
    • Reduces conflict misses
    • Increases latency, increases power consumption
  – Larger number of cache levels
    • Reduces average memory access time
  – Avoiding address translation in cache indexing
    • reduces latency
The changing memory hierarchy

- **IBM RS6000 320H - 25 MHz (1991)**

<table>
<thead>
<tr>
<th>Storage component</th>
<th>Latency (cycles)</th>
<th>Transfer Rate (words [8B] / cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk</td>
<td>1,000,000</td>
<td>0.001</td>
</tr>
<tr>
<td>Main memory</td>
<td>60</td>
<td>0.1</td>
</tr>
<tr>
<td>Cache</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Registers</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

- **Intel Xeon 61xx [per core @3GHz] (2019)**

<table>
<thead>
<tr>
<th>Storage component</th>
<th>Latency (cycles)</th>
<th>Transfer Rate (words [8B] / cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDD</td>
<td>18,000,000</td>
<td>0.00007</td>
</tr>
<tr>
<td>SSD</td>
<td>300,000</td>
<td>0.02</td>
</tr>
<tr>
<td>Main memory</td>
<td>250</td>
<td>0.2</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>48</td>
<td>0.5</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Registers</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>
Computational Intensity: a key metric limiting performance

- Computational intensity of a problem
  \[ I = \frac{\text{(total \# of arithmetic operations required)}}{\text{(size of input + size of result)}} \text{ in flops} \]
  \[ \text{in 64-bit words} \]

- BLAS - Basic Linear Algebra Subroutines
  - Asymptotic performance limited by computational intensity
    - \( A, B, C \in \mathbb{R}^{n \times n}, \quad x, y \in \mathbb{R}^{n}, \quad a \in \mathbb{R} \)

<table>
<thead>
<tr>
<th>name</th>
<th>defn</th>
<th>flops</th>
<th>refs</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>( y = ax )</td>
<td>( n )</td>
<td>( 2n )</td>
<td>( 0.5 )</td>
</tr>
<tr>
<td>triad</td>
<td>( y = ax + y )</td>
<td>( 2n )</td>
<td>( 3n )</td>
<td>( 0.67 )</td>
</tr>
<tr>
<td>dot product</td>
<td>( x \cdot y )</td>
<td>( 2n )</td>
<td>( 2n )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>Matrix-vector</td>
<td>( y = y + Ax )</td>
<td>( 2n^2 + n )</td>
<td>( n^2 + 3n )</td>
<td>( \sim 2 )</td>
</tr>
<tr>
<td>rank-1 update</td>
<td>( A = A + xy^T )</td>
<td>( 2n^2 )</td>
<td>( 2n^2 + 2n )</td>
<td>( \sim 1 )</td>
</tr>
<tr>
<td>Matrix product</td>
<td>( C = C + AB )</td>
<td>( 2n^3 )</td>
<td>( 4n^2 )</td>
<td>( n/2 )</td>
</tr>
</tbody>
</table>
Effect of the memory hierarchy on execution time

- \( C^{N \times N} = A^{N \times N} \cdot B^{N \times N} \) naïve implementation

```plaintext
do i = 1,N
  do j = 1,N
    do k = 1,N
      C(i,j) = C(i,j) + A(i,k)*B(k,j)
    end do
  end do
end do
```

- Machine
  - simple L1 cache
    - block size = 16 words
    - capacity = 512 blocks
    - fully associative
  - main memory
    - 4K pages
- Layout of A,B,C in memory
  - Fortran: column-major order
- RAM model suggests \( O(N^3) \) run time
  - actual time follows \( O(N^5) \) growth!

Performance of naïve \( N \times N \) matrix multiply on an IBM RS6000/320 uniprocessor. Time in clock cycles per multiply-add (note \( \log_{10} \) scales). Source: Alpern et al., “The Uniform Memory Hierarchy Model of Computation”, Algorithmica, 1994
Shared memory taxonomy

- **Uniform Memory Access (UMA)**
  - Processors and memory separated by network
  - All memory references cross network
  - Only practical for machines with full latency hiding
    - Parallel vector processors, multi-threaded processors
    - Expensive, rarely available in practice

![Diagram of shared memory multiprocessors](image)
Shared memory taxonomy

- Non-Uniform Memory Access (NUMA)
  - Memory is partitioned across processors
  - References are local or non-local
    - Local references
      - low latency
    - Non-local references
      - high latency
    - non-local : local latency
      - large

- Examples

- Poor performance unless extreme care is taken in data placement
Combining (N)UMA with cache memories

- Processor-local caches
  - Cache all memory references
  - Must reflect changes in value due to other processors in system
  - Cache-misses
    - Usual: compulsory, capacity, and conflict misses
    - New: coherence misses

- Cache-coherent UMA examples
  - Conventional PC-based SMP systems
    - Network is a shared bus
    - Limited scaling ($p \leq 4$)
    - mostly extinct
  - Server-class machines
    - Dual or Quad socket (single card)
    - Intel Xeon or AMD EPYC ($20 \leq p \leq 128$)
    - prevalent

- Cache-coherent NUMA examples
  - scales to larger processor count
    - SGI UltraViolet ($p \sim 1024$)
    - rare
Incorporating *shared* memory in the hierarchy

- **Non-local shared memory**
  - can be viewed as additional level in processor-memory hierarchy

- **Shared-memory parallel programming**
  - extension of memory hierarchy techniques
  - goal:
    - concurrent transfer through parallel levels

<table>
<thead>
<tr>
<th>Storage component</th>
<th>Latency (cycles)</th>
<th>Transfer Rate (words [8B] / cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk</td>
<td>1,000,000</td>
<td>0.001</td>
</tr>
<tr>
<td>Non-local memory</td>
<td>180 - 500</td>
<td>0.1 - 0.01</td>
</tr>
<tr>
<td>Local memory</td>
<td>60</td>
<td>0.1</td>
</tr>
<tr>
<td>Cache</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Registers</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Modern shared-memory server: Intel Xeon series
AMD Infinity

• Speed of light inconveniently slow!
  – miniaturize size of memory and processors

• Single card server
  – 7 nm process technology
  – 64 – 256 cores total,
  – 4 TB memory
Shared-memory programming models

- **Work-Time programming model**
  - sequential programming language + `forall`
    - PRAM execution
      - synchronous
      - scheduling implicit (via Brent’s theorem)
    - W-T cost model (work and steps)

- **Loop-level parallel programming model**
  - sequential programming language + directives to mark `for` loop as “forall”
    - shared-memory multiprocessor execution
      - asynchronous execution of loop iterations by multiple threads *in a single address space*
        - must avoid dependence on synchronous execution model
      - scheduling of work across threads is controlled via directives
        - implemented by the compiler and run-time systems
    - cost model depends on underlying shared memory architecture
      - can be difficult to quantify
      - but some general principles apply
OpenMP parallel programming model

- OpenMP shared-memory parallel programming model
  - loop-level parallel programming

- Characterizing performance
  - performance measurement of a simple program
  - how to monitor and present program performance
  - general barriers to performance in parallel computation
Example shared-memory machine

Phaedra

- 10 compute cores per socket, total 20 cores
- Single shared physical address space
- 64 GB memory per socket, 128 GB total shared memory
- Cache-coherence protocol for performance
OpenMP

• OpenMP
  – parallelization directives for mainstream performance-oriented sequential programming languages
    • C/C++ , Fortran (88, 90/95)
  – directives are written as comments in the program text
    • ignored by non-OpenMP compilers
    • honored by OpenMP-compliant compilers in “OpenMP” mode
  – directives specify
    • parallel execution
      – create multiple threads, generally each thread runs on a separate core in a CC-NUMA machine
    • partitioning of variables
      – a variable is either shared between threads OR each thread maintains a private copy
    • work scheduling in loops
      – partitioning of loop iterations across threads

• C/C++ binding of OpenMP
  – form of directives
    • #pragma omp . . .
OpenMP parallel execution of loops

```c
...
printf("Start.\n");
for (i = 1; i < N-1; i++) {
    b[i] = (a[i-1] + a[i] + a[i+1]) / 3;
}
printf("Done.\n");
...
```

- Can different iterations of this loop be executed simultaneously?
  - for different values of $i$, the body of the loop can be executed simultaneously

- Suppose we have $n$ iterations and $p$ threads?
  - we have to *partition* the iteration space across the threads
OpenMP directives to control partitioning

```c
... printf(“Start.\n”);
#pragma omp parallel for shared(a,b) private(i)
for (i = 1; i < N-1; i++) {
    b[i] = (a[i-1] + a[i] + a[i+1]) / 3;
}
printf(“Done.\n”);
...```

- The `parallel` directive indicates the next *statement* should be executed by all threads.
- The `for` directive indicates the work in the loop body should be partitioned across threads.
- The `shared` directive indicates that arrays `a` and `b` are shared by all threads.
- The `private` directive indicates `i` has a separate instance in each thread.
- The last two directives would be inferred by the OpenMP compiler.
OpenMP components

• Directives
  – specify parallel vs sequential regions
  – specify shared vs private variables in parallel regions
  – specify work sharing: distribution of loop iterations over threads
  – specify synchronization and serialization of threads

• Run-time library
  – obtain parallel processing resources
  – control dynamic aspects of work sharing

• Environment variables
  – external to program
  – specification of resources available for a particular execution
    • enables a single compiled program to run using differing numbers of processors
C/OpenMP concepts: parallel region

Fork-join model
- master thread forks a team of threads on entry to block
  - variables in scope within the block are
    - shared among all threads
      » if declared outside of the parallel region
      » if explicitly declared shared in the directive
    - private to (replicated in) each thread
      » if declared within the parallel region
      » if explicitly declared private in the directive
      » if variable is a loop index variable in a loop within the region
  - the team of threads has dynamic lifetime to end of block
    - statements are executed by all threads
  - the end of block is a barrier synchronization that joins all threads
    - only master thread proceeds thereafter

#pragma omp parallel shared(..) private(..)
  <single entry, single exit block>
C/OpenMP concepts: work sharing

```plaintext
#pragma omp for schedule(…)
for (<var> = <lb>; <var> <op> <ub>; <incr-expr>)
  <loop body>
```

- **Work sharing**
  - only has meaning inside a parallel region

  - the *iteration space* is distributed among the threads
    - several different scheduling strategies available

  - the loop construct must follow some restrictions
    - `<var>` has a signed integer type
    - `<lb>`, `<ub>`, `<incr-expr>` must be loop invariant
    - `<op>`, `<incr-expr>` restricted to simple relational and arithmetic operations

  - implicit barrier at completion of loop
#include <stdio.h>
#include <omp.h>
#define N 50000000
#define NITER 100

double a[N], b[N];
main()
{
double t1, t2, td;
int i, t, max_threads, niter;

max_threads = omp_get_max_threads();
printf("Initializing:  N = %d, max # threads = %d\n", N, max_threads);

/*
 * initialize arrays
 */
for (i = 0; i < N; i++) {
    a[i] = 0.0;
    b[i] = 0.0;
}
a[0] = b[0] = 1.0;
/* 
* time iterations
*/
t1 = omp_get_wtime();
for (t = 0; t < NITER; t = t + 2){
    
    #pragma omp parallel for private(i)
    for (i = 1; i < N-1; i++)
        b[i] = (a[i-1] + a[i] + a[i+1]) / 3.0;

    #pragma omp parallel for private(i)
    for (i = 1; i < N-1; i++)
        a[i] = (b[i-1] + b[i] + b[i+1]) / 3.0;
}

t2 = omp_get_wtime();
td = t2 - t1;
printf("Time per element = %6.1f ns\n", td * 1E9 / (NITER * N));
}
/*
  * time iterations
  */
t1 = omp_get_wtime();

#pragma omp parallel private(i,t)
for (t = 0; t < NITER; t = t + 2){
    #pragma omp for
    for (i = 1; i < N-1; i++)
       b[i] = (a[i-1] + a[i] + a[i+1]) / 3.0;

    #pragma omp for
    for (i = 1; i < N-1; i++)
       a[i] = (b[i-1] + b[i] + b[i+1]) / 3.0;
}

t2 = omp_get_wtime();

td = t2 - t1;
printf("Time per element = %6.1f ns\n", td * 1E9 / (NITER * N));
Complete program (V3 – page and cache affinity)

```c
#include <stdio.h>
#include <omp.h>
define N 50000000
#define NITER 100

double a[N], b[N];

main ()
{
    double t1, t2, td;
    int i, t, max_threads, niter;

    max_threads = omp_get_max_threads();
    printf("Initializing:  N = %d, max # threads = %d\n", N, max_threads);

    #pragma omp parallel private(i, t)
    { // start parallel region

        /*
         * initialize arrays
        */
        #pragma omp for
        for (i = 1; i < N; i++){
            a[i] = 0.0;
            b[i] = 0.0;
        }

        #pragma omp master
        a[0] = b[0] = 1.0;
    }
}
```
/*
 * time iterations
 */
#pragma omp master
    t1 = omp_getwtime();

    for (t = 0; t < NITER; t = t + 2){

        #pragma omp for
        for (i = 1; i < N-1; i++)
            b[i] = (a[i-1] + a[i] + a[i+1]) / 3.0;

        #pragma omp for
        for (i = 1; i < N-1; i++)
            a[i] = (b[i-1] + b[i] + b[i+1]) / 3.0;
    }
}  // end parallel region

    t2 = omp_get_wtime();
    td = t2 - t1;
    printf("Time per element = %6.1f ns\n", td * 1E9 / (NITER * N));
}
Effect of caches

- Time to update one element in *sequential execution*
  - $b[i] = (a[i-1] + a[i] + a[i+1]) / 3.0$;
  - depends on where the elements are found
  - registers, L1 cache, L2 cache, main memory

![Graph showing the effect of caches on time per element](image-url)
How to present scaling of parallel programs?

• Independent variables
  – either
    • number of processors $p$
    • problem size $n$

• Dependent variable (choose)
  – Time (secs)
  – Rate (opns/sec)
  – Speedup $S = T_1 / T_p$
  – Efficiency $E = T_1 / pT_p$

• Horizontal axis
  – independent variable (n or p)

• Vertical axis
  – Dependent variable (e.g. time per element)
  – May show multiple curves (e.g. different values of n)
Time

- Shortest time is our true goal
  - But hard to judge improvements because values get very small at large $p$
Execution rate (MFLOP / second)

- Shows work per time
  - easier to judge scaling
  - highest detail at large $n, p$
  - how to measure MFLOPS?

Parallel performance
**Speedup**

- Speedup of run time relative to single processor \((t_1 / t_p)\)
  - How to define \(t_1\)?
    - run time of parallel algorithm at \(p = 1\)?
    - run time of best serial algorithm?
  - Superlinear speedup?
OpenMP: scheduling loop iterations

- Scheduling a loop with n iterations using p threads
  - The unit of scheduling is a chunk of k iterations
  - $T_i$ means iteration(s) executed by thread $i$
- **schedule**(static, k)
  - Chunks mapped to threads in
    at entry to loop
  - default $k = n/p$
- **schedule**(dynamic, k)
  - chunks handed out consecutively
    to ready threads
  - default $k = 1$
- **schedule**(guided, k)
  - size $d$ chunk handed to
    first available thread
  - $d$ decreases exponentially
    from $n/p$ down to $k$:
    $d_{i+1} = (1-1/p)d_i$ where $d_0=n/p$
  - default $k = 1$
Varying scheduling strategy: diffusion problem

Speedup by schedule type
\( (n = 10,000,000) \)
Causes of poor parallel performance

Possible suspects:

- Low computational intensity
  - Performance limited by memory performance

- Poor cache behavior
  - access pattern has poor locality
  - access pattern is poorly matched to CC-NUMA

- Sequential overhead
  - Amdahl’s law
    - fraction f serial work limits speedup to 1/f

- Load imbalance
  - Unequal distribution of work, or
  - Unequal thread progress on equal work
    - busy machine, uncooperative OS
    - CC-NUMA issues

- Bad luck
  - Insufficient sampling - show timing variation on plots!
Cache-related mysteries

Execution rate

The graph shows the execution rate in MFLOP/second as a function of the number of processors, for two different values of the variable $n$: $n = 10,000,000$ (red squares) and $n = 1,000,000$ (blue diamonds). The execution rate increases as the number of processors increases, but the rate of increase is higher for $n = 10,000,000$ compared to $n = 1,000,000$. This suggests that there are cache-related mysteries that affect the performance of the processors.
Cache-related mysteries: speedup

Parallel speedup
(single parallel region)

speedup

number of processors

n = 1,000,000
n = 10,000,000
OpenMP on CC-NUMA

- Performance guidelines
  - shared data structures
    - use cache-line spatial locality
      - linear access patterns (read and write)
      - structs with components grouped by access
  - don’t mix reads and writes to same data on different processors
    - use phased updates
  - avoid false sharing
    - unrelated values sharing a cache line updated by multiple threads
  - make sure data structures are physically distributed across memory
    - by parallel initialization
      » artifact of page placement policy under e.g. Linux
    - by explicit placement directives and page allocation policies
OpenMP on CC-(N)UMA

• Other guidelines
  – Enlarge parallel region
    • to retain processor – data affinity
    • to avoid overhead of repeated entry to parallel region in an inner loop
  – Use appropriate work distribution schedule
    • static, else
    • guided, else
    • dynamic with large chunksize
    • runtime-specified schedule involves relatively small overhead
  – Don’t use too many processors
    • OS scheduling of threads behaves erratically when machine is oversubscribed
    • be aware of dynamic thread adjustment (OMP_DYNAMIC)
Reductions and critical statements

• a reduction loop does not have independent iterations

```c
for (i = 0; i < n; i++) {
    sum = sum + a[i];
}
```

• the loop may be parallelized by inserting a critical section
  – the critical directive serializes a single statement or block

```c
#pragma omp parallel for
for (i = 0; i < n; i++) {
    #pragma omp critical
    sum = sum + a[i];
}
```
  – but this is a poor strategy!

• a reduction loop can be identified using a reduction directive

```c
#pragma omp parallel for reduction(+: sum)
for (i = 0; i < n; i++) {
    sum = sum + a[i];
}
```
Implementation of reduction directive

• A better implementation of the reduction loop
  
  ```
  sum = 0;
  #pragma omp parallel
  {
    int i, local_sum = 0;
    #pragma omp for
    for (i=0; i<n; i++) {
      local_sum = local_sum + a[i];
    }
    #pragma omp critical
    sum = sum + local_sum
  }
  ```

  reduces number of critical operations from \( n \) to \( p \)

• other reduction strategies
  – serialization: master thread sequentially combines local_sum values
  – tree-based reduction
  – hybrid strategy

  OpenMP compiler should generate code that selects optimal strategy at run time