Lecture 10

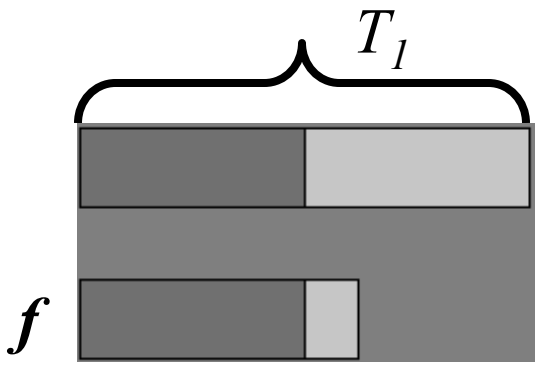
Midterm Review

Announcements

- Testing time – final choice
  - 7pm to 9pm

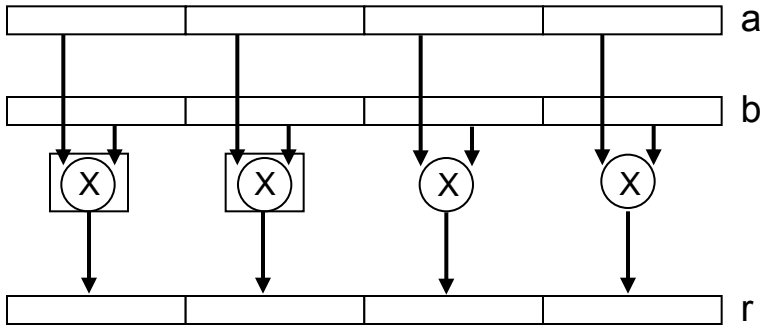
Terms and concepts

- Know the definition and significance of ….
- Parallel speedup and efficiency, super-linear speedup, strong scaling, weak scaling
- Amdahl's law, Gustafson's "law," serial bottlenecks
- Strong and weak scaling
- Granularity
- SSE



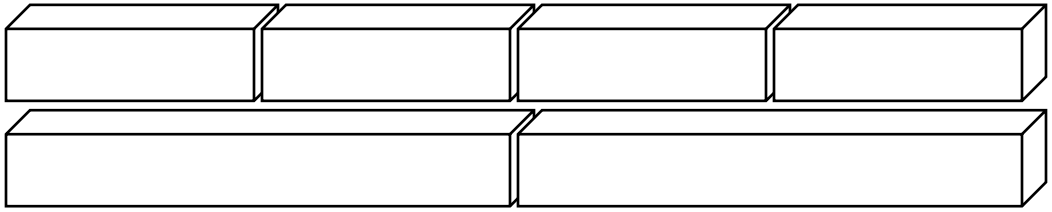
Streaming SIMD Extensions (SSE)

- SSE (SSE4 on Intel Nehalem), Altivec
- Short vectors: 128 bits (256 bits coming)



r[0:3] = a[0:3]*b[0:3]

```
float a[N], b[N], c[N];
        for (int i=0; i<N; i++)
              a[i] = b[  i] *  c[ I ];
```

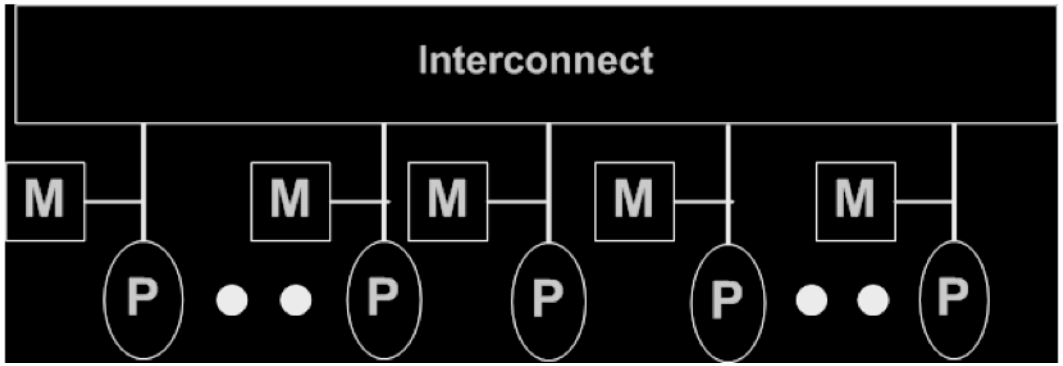


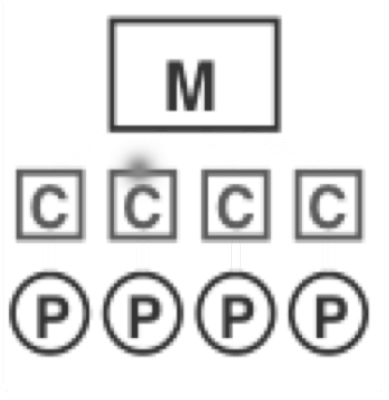
4 floats

2 doubles

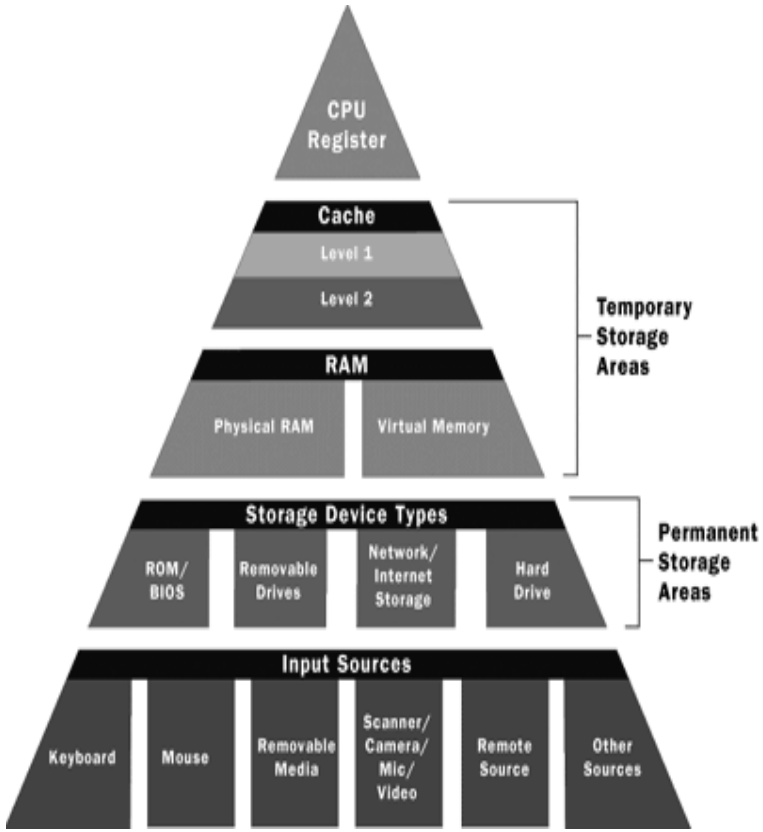
Terms and concepts

- SPMD, MIMD, SIMD
- Multiprocessors and multicomputers
- NUMAs and SMPs
- Processor Memory Gap
- Cache coherence and consistency
- Snooping
- False sharing
- Data dependencies, loop carried dependence
- Critical sections, race conditions



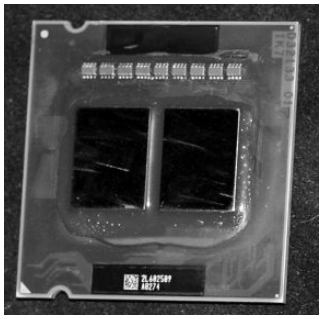


Memory hierarchy pyramid



Intel Clovertown Memory Hierarchy

- Ieng-203
- Intel Xeon X5355 (Intro: 2006)
- Two "Woodcrest" dies on a multichip module



techreport.com/articles.x/10021/2

Line Size = 64B (L1 and L2)

Access latency (clocks)

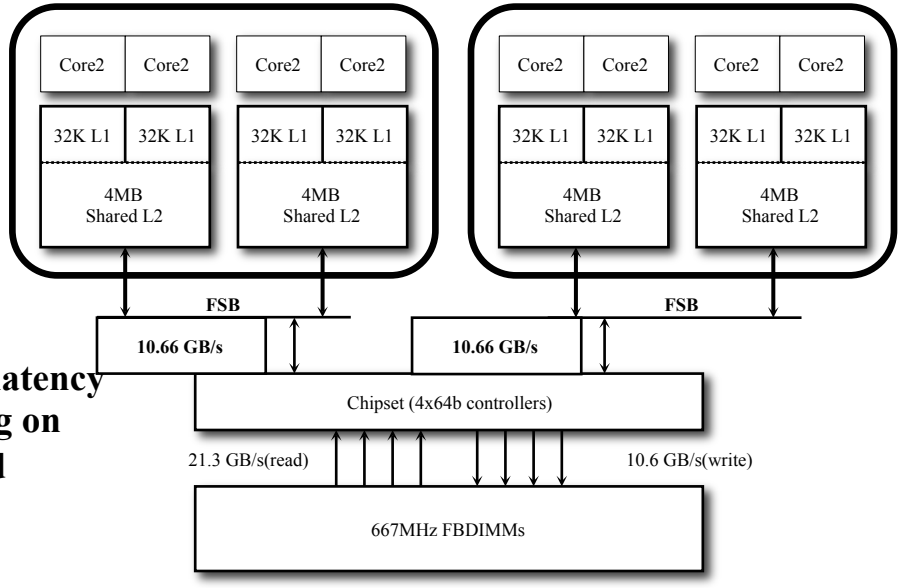
3

14*

Associativity

8

16

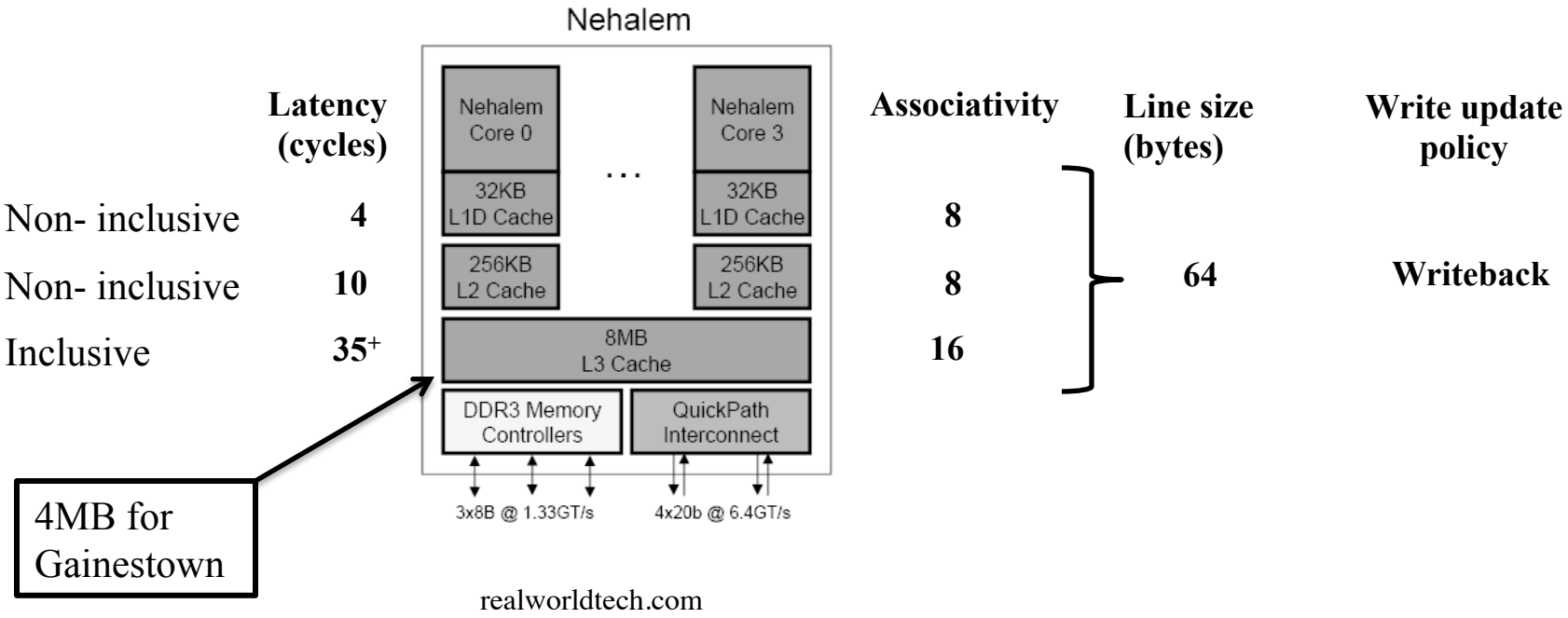


*** Software-visible latency will vary depending on access patterns and other factors**

Sam Williams et al.

Nehalem's Memory Hierarchy

- Source: *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Table 2.7



	Latency (cycles)	Associativity	Line size (bytes)	Write update policy
Non- inclusive	4	8		
Non- inclusive	10	8	64	Writeback
Inclusive	35+	16		

Why do we look at numerically intensive applications?

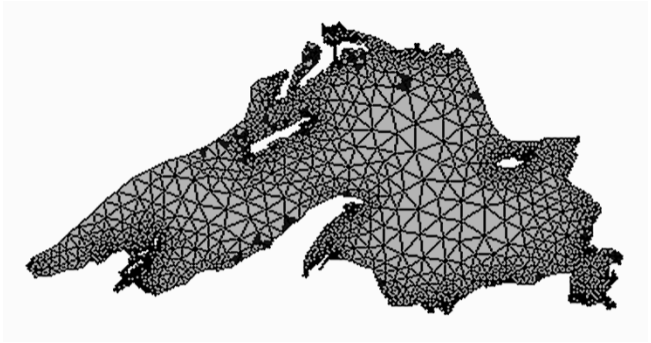
- Highly repetitive computations are prime candidates for parallel implementation
- Tight loop nests are the "low hanging fruit"
- Many real world applications
  - Data Mining
  - Image processing
  - Simulations – financial modeling, weather, biomedical
- We can classify applications according to Patterns of *communication* and computation that persist over time and across implementations

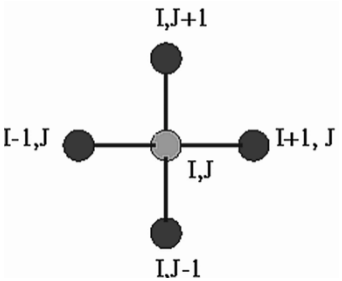
  Phillip Colella's 7 Dwarfs

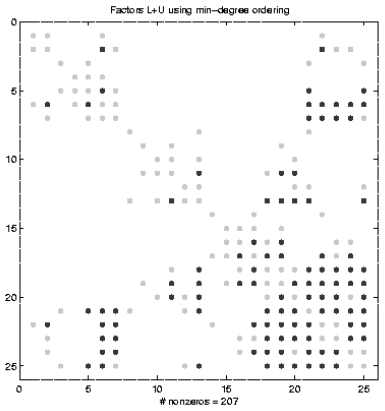
Courtesy of Randy Bank

Classifying the application domains

- Patterns of *communication* and computation that persist over time and across implementations
- Structured grids
  - Lab #2
- Dense linear algebra:
  - Matrix multiply, Vector-Mtx Mpy Gaussian elimination
- N-body methods
- Sparse linear algebra
  - In a sparse matrix, we take advantage of knowledge about the locations of non-zeros, improving some aspect of performance
- Unstructured Grids
- Spectral methods (FFT)
- Monte Carlo







Courtesy of Randy Bank

Application-specific knowledge is important

- There currently exists no tool that can turn a serial program into an efficient parallel program

  … for all applications … all of the time… on all hardware

- The more we know about the application…

  … specific problem … math/physics ... initial data …

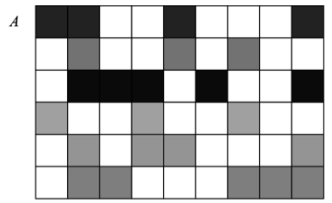
  … context for analyzing the output…

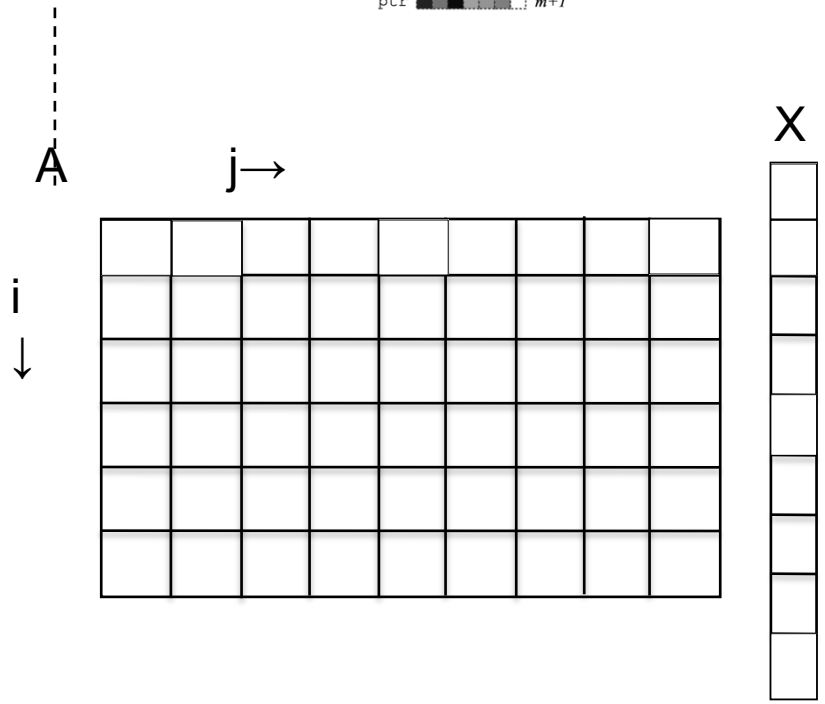
  … the more we can improve productivity

- Issues
  - Data motion and locality
  - Load balancing
  - Serial sections

Sparse matrix vector multiply

```
// y[i] += A[i,j] × x[j]
#pragma parallel for schedule
                (dynamic,chunk)
for i = 0 : N-1        // rows
  i0= ptr[i]
  i1 = ptr[i+1] – 1
  for j = i0 : i1      // cols
    y[ ind[j]] +=
         val[ j ] *x[ ind[j] ]
  end j
end i
```



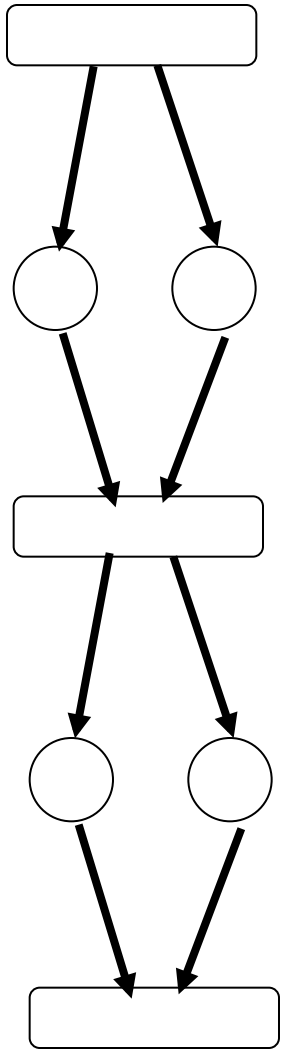


Implementation techniques

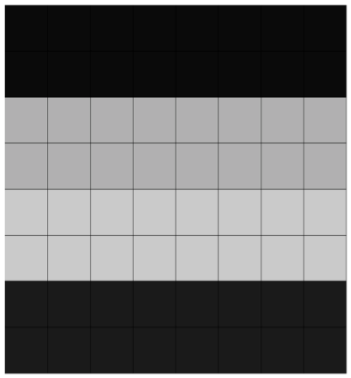
- pthreads
- OpenMP
- Mutexes and barriers
- Processor geometry
- Layouts: BLOCK, Cyclic, Block Cyclic
- Elimination data dependencies
- Red-Black/ Odd-even

OpenMP llLow control

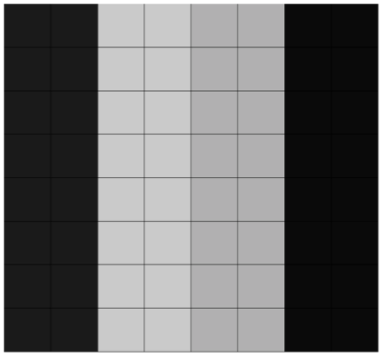
```
printf("Start\n");
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf("Finish\n");
```



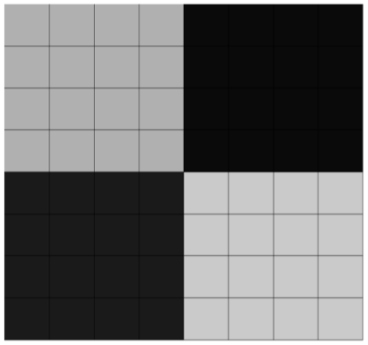
Layouts



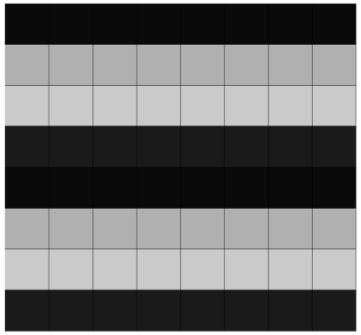
[Block, *]



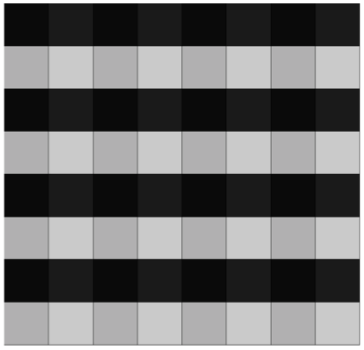
[*, Block]



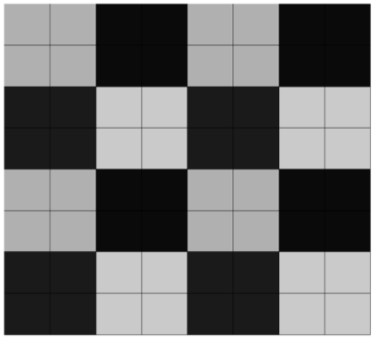
[Block, Block]



[Cyclic, *]



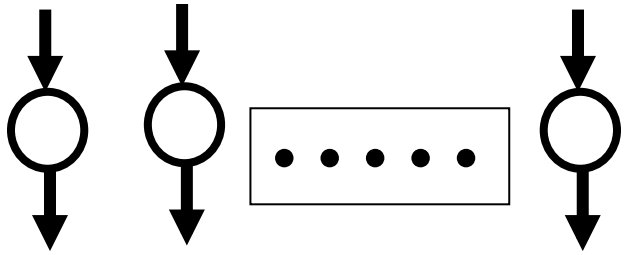
[Cyclic, Cyclic]

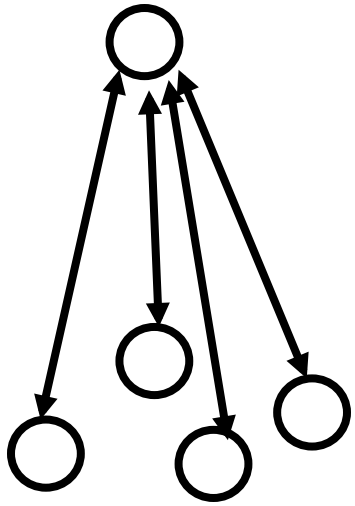


[Cyclic(2), Cyclic(2)]

Work sharing paradigms

- Task parallelism
- Data parallel
- Self scheduling





Multithreading in perspective

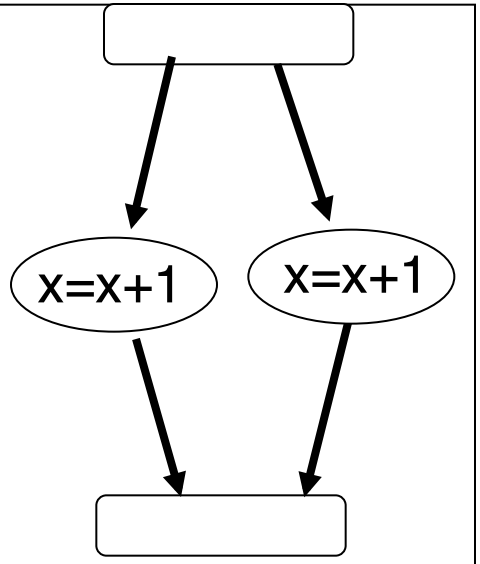
- Benefits
  - Harness parallelism to improve performance
  - Ability to multitask to realize concurrency, e.g. display
- Pitfalls
  - Program complexity
    - Partitioning, synchronization, complicated parallel control flow
    - Data dependencies
    - Shared vs. local state (globals like errno)
    - Thread-safe code, re-entrant functions, protect the use of globals
  - New aspects of performance
    - Complicated locality models
  - New aspects of debugging
    - Race conditions
    - Deadlock

Under the hood of a race condition

- Consider this statement, assume x == 0

  x=x+1;

- Generated code
  - r1 ← (x)
  - r1 ← r1 + #1
  - r1 → (x)



- Possible interleaving with two threads

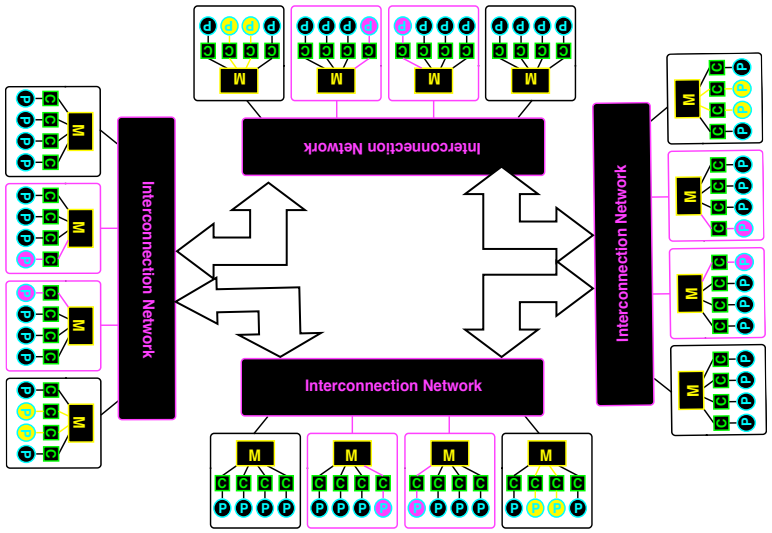
P1	P2	
r1 ← x		*r1(P1) gets 0*
	r1 ← x	*r2(P2) also gets 0*
r1 ← r1+ #1		*r1(P1) set to 1*
	r1 ← r1+#1	*r1(P1) set to 1*
x ← r1		*P1 writes its R1*
	x ← r1	*P2 writes its R1*

Threaded program design and implementation

- Code organization and re-use
- Parallelizing serial code (code reorganization and restructuring)
- How to design with parallelism in mind
- Performance

What will multicore processors look like?

- Still and open book
- Exposed hierarchies - cache coherence limited
- Locality domains



In class exercises

Questions

1. Time constrained scaling
2. Tree Summation
3. Synchronization
4. Iteration to thread mapping
5. Printing the letters of the alphabet
6. Removing data dependencies
7. Dependence analysis
8. Performance

1. Time constrained scaling

- Sum N numbers on P processors
- Let N >> P
- Determine the largest problem that can be solved in time $T=10^4$ time units on 512 processors
- Let time to perform one addition = 1 time unit
- Let β = time to add a value inside a critical section

Performance model

- Local additions: N/P - 1
- Reduction: β (lg P)
- Since N >> P

  T(N,P) ~ (N/P) + β (lg P)
- Determine the largest problem that can be solved in time T= 10^4 time units on P=512 processors, β = 1000 time units
- Constraint: T(512,N) $\leq$ 10^4

  $\Rightarrow$ (N/512) + 1000 (lg 512)

  = (N/512) + 1000*(9) $\leq$ 10^4

  $\Rightarrow$ N $\leq$ 5×10^5 (approximately)

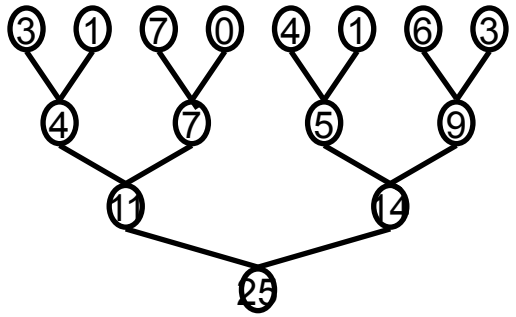
2. Tree Summation

- Input: an array x[], length N >> P
- Output: Sum of the elements of x[]
- Goal: Compute the sum in lg P time

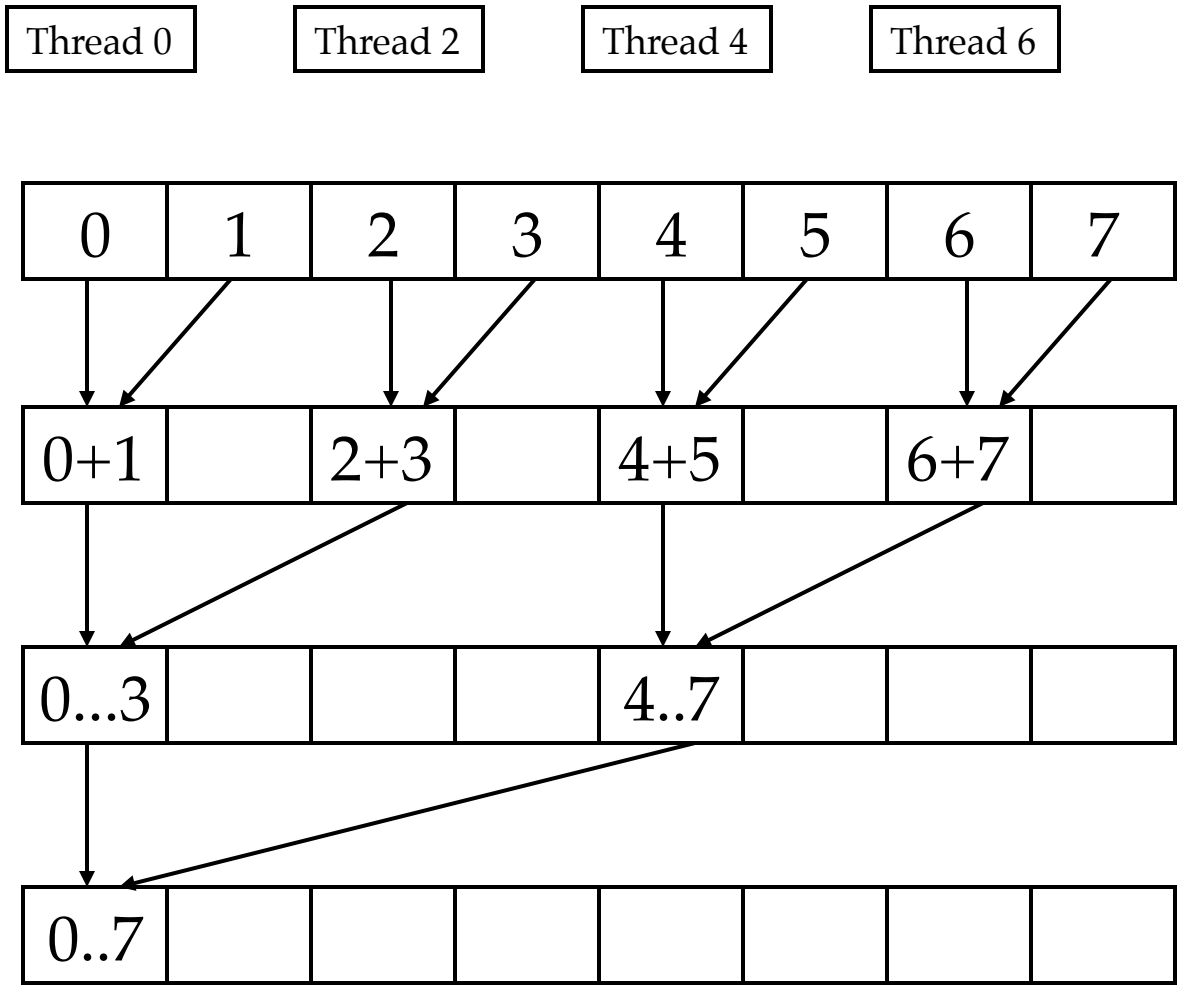
```
sum = 0;
for i=0 to N-1
    sum += x[i]
```

- Assume P is a power of 2, K = lg P
- Starter code

```
for m = 0 to K-1 {

}
```



Visualizing the Summation



3. Synchronization

List all possible outputs that result from running the following

```
#pragma omp parallel for shared(j,k)
for ( int i=0, j=0, k=0; i< 5; i++ )
    j = j + 10;
    k = j + 10;
}
cout << “k = “ << k << endl;
```

4. Iteration to thread mapping

```
#pragma omp parallel shared(N,iters) private(i)
#pragma omp for
for (i = 0; i < N; i++)
        iters[i] = omp_get_thread_num();
```

$HW/Examples/Assign

N = 9, # of openMP threads = 3

0 0 0 1 1 1 2 2 2

N = 16, # of openMP threads = 4, schedule(static,2)

0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3

N=9: 0 0 1 1 2 2 0 0 1

N = 16, # of openMP threads = 4, schedule(dyanmic,2)

3 3 0 0 1 1 2 2 3 3 3 3 3 3 3 3

2 2 3 3 0 0 1 1 2 2 2 2 2 2 2 2

5. Printing the letters of the alphabet

```
#pragma omp parallel private(i){
    int LettersPerThread = 26 / omp_get_num_threads();
    int ThisThread = omp_get_thread_num();
    int Start= 'a'+ThisThread*LettersPerThread;
    int End = 'a'+(ThisThreadNum+1)*LettersPerThread;
     for (i=Start; i<End; i++)
        printf ("%c", i);
}
```

6. Removing data dependencies

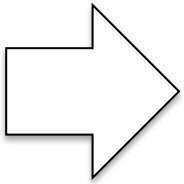
- B initially: 0 1 2 3 4 5 6 7
- B on 1 thread: 0 9 10 11 12 21 22 23
- B on 2 threads : 0 17 18 19 12 13 14 15
- How can we split into 2 loops so that each loop parallelizes, the result it correct?

```
for i  = 1 to N
    B[i] += B[N-i];
```

Splitting a loop

- For iterations i=N/2+1 to N, B[200-i] reference newly computed data
- All others reference "old" data
- B initially: 0 1 2 3 4 5 6 7
- Correct result: 0 9 10 11 12 21 22 23

```
for i = 1 to N
    B[i] += B[N-i];
```



```
for i = 1 to N/2
    B[i] += B[N-i];
for i = N/2+1 to N
    B[i] += B[N-i];
```

7. Loop Dependence Analysis

- Can we run parallelize these with openmp?

1.
```
for i = 1 to N
    A[i] = A[i] + B[i-1];
```

2.
```
for i = 1 to N
    A[i+1] = A[i] + 1;
```

3.
```
A[0] = 0;
for i = 1 to N-1
    A[i] = A[i-1] + i;
```

4.
```
for i = 1 to N{
    A[i] = B[i];
    C[i] = A[i] + B[i];
    E[i] = C[i+1];
}
```

Nested loop dependence analysis

- Can we parallelize the inner loops as shown?

LOOP #1

```
for j = 0 to n-1
    for i = 0 to n-1
        A[i, j+1] = A[i, j];
```

LOOP #2

```
for j = 0 to n-1
    for i = 0 to n-1
        A[i, j+1] = A[i, j];
```

8. Performance

- You observe the following running times for a parallel program running a fixed workload N
- Assume that the only losses are due to serial sections
- What is the speedup and efficiency on 8 processors?
- What will the running time be on 4 processors?
- What is the maximum possible speedup on an infinite number of processors?
- What fraction of the total running time on 1 processor corresponds to the serial section?
- What fraction of the total running time on 2 processors corresponds to the serial section?

NT	Time
1	10000
2	6000
8	3000