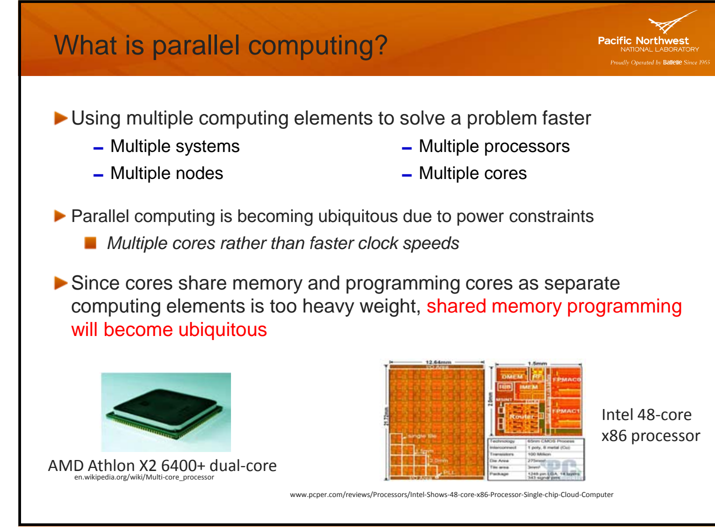
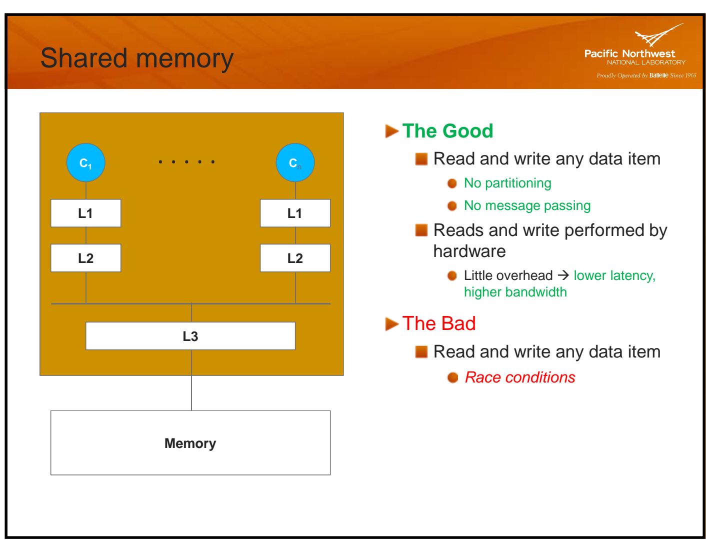
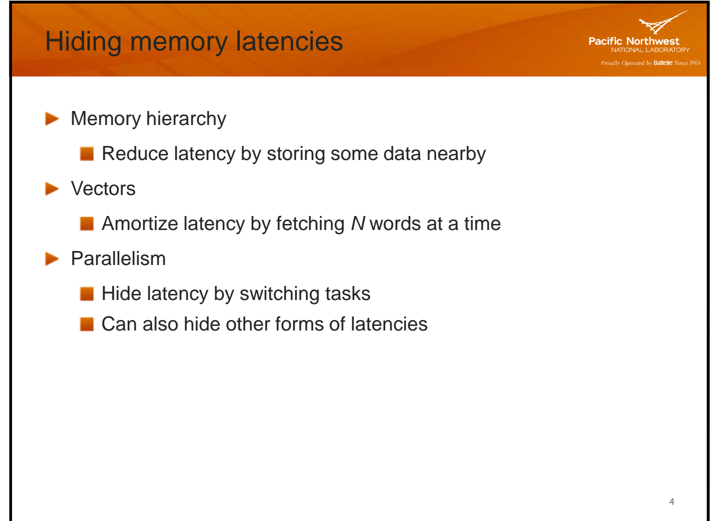
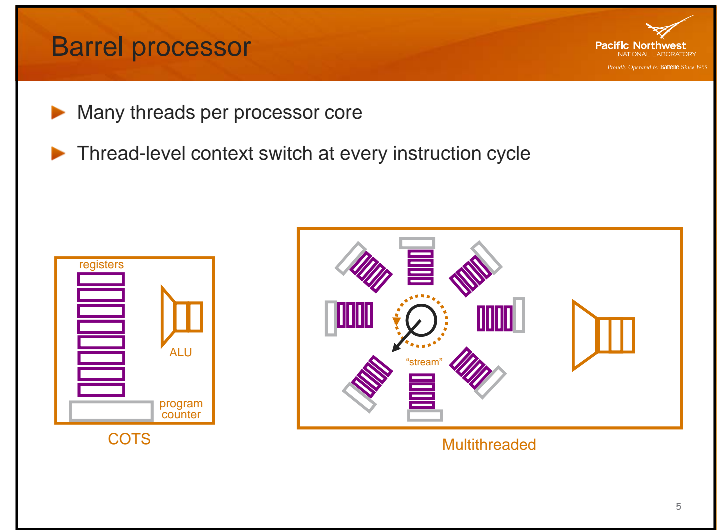
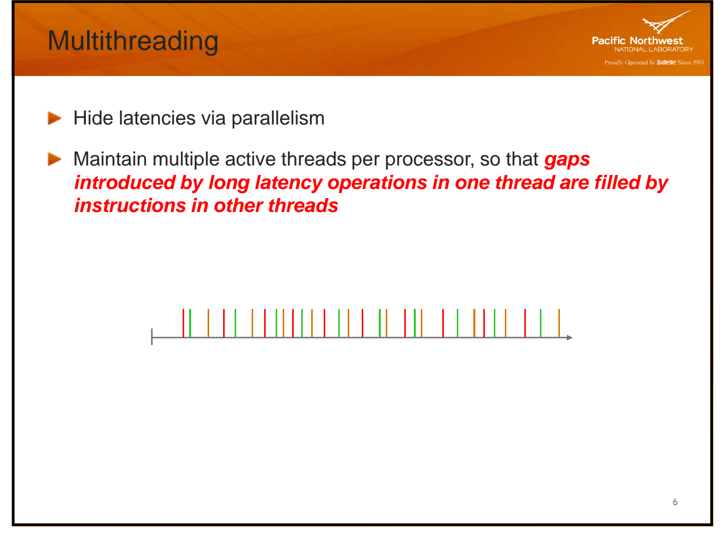
Pacific Northwest
NATIONAL LABORATORY
Programming the Cray XMT
John Feo
Director
Center for Adaptive Supercomputing Software
May 21, 2012
1

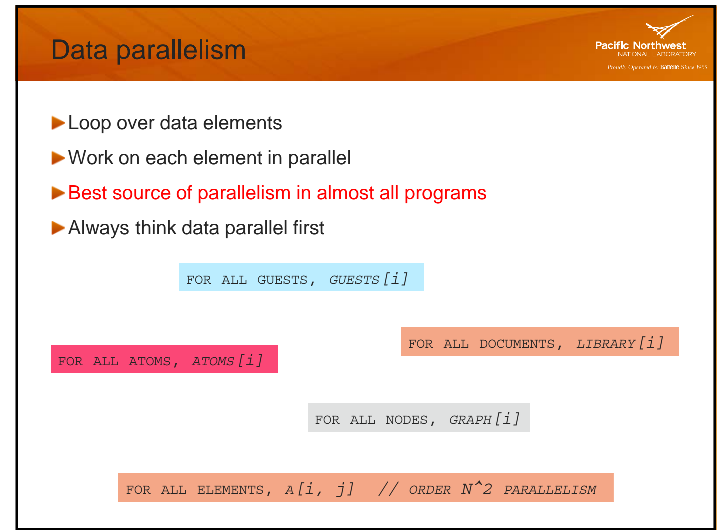

What is parallel computing?
Pacific Northwest
NATIONAL LABORATORY
Using multiple computing elements to solve a problem faster
– Multiple systems
– Multiple processors
– Multiple nodes
– Multiple cores
Parallel computing is becoming ubiquitous due to power constraints
Multiple cores rather than faster clock speeds
Since cores share memory and programming cores as separate computing elements is too heavy weight, shared memory programming will become ubiquitous
AMD Athlon X2 6400+ dual-core
en.wikipedia.org/wiki/Multi-core_processor
Intel 48-core x86 processor
www.pcper.com/reviews/Processors/Intel-Shows-48-core-x86-Processor-Single-chip-Cloud-Computer

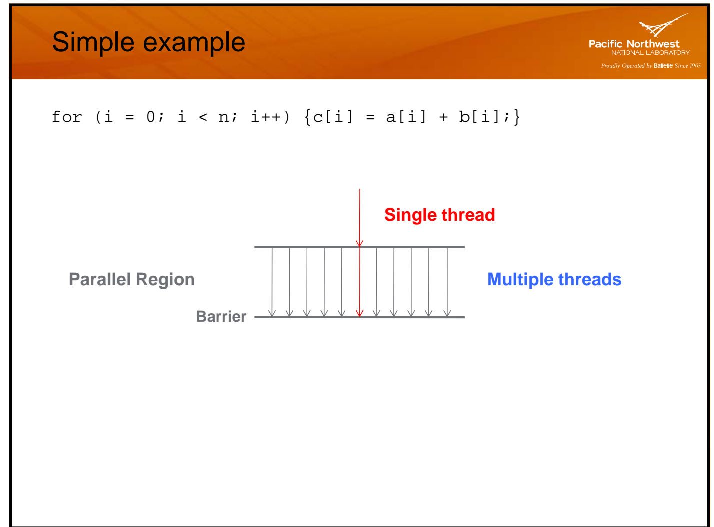

Shared memory
Pacific Northwest
NATIONAL LABORATORY
L1
L1
L2
L2
L3
Memory
The Good
Read and write any data item
No partitioning
No message passing
Reads and write performed by hardware
Little overhead → lower latency, higher bandwidth
The Bad
Read and write any data item
Race conditions

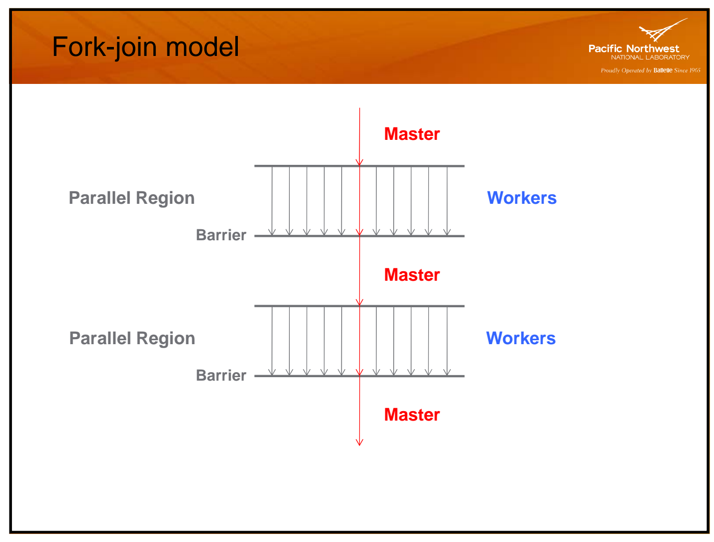

Hiding memory latencies
Pacific Northwest
NATIONAL LABORATORY
Memory hierarchy
Reduce latency by storing some data nearby
Vectors
Amortize latency by fetching N words at a time
Parallelism
Hide latency by switching tasks
Can also hide other forms of latencies
4

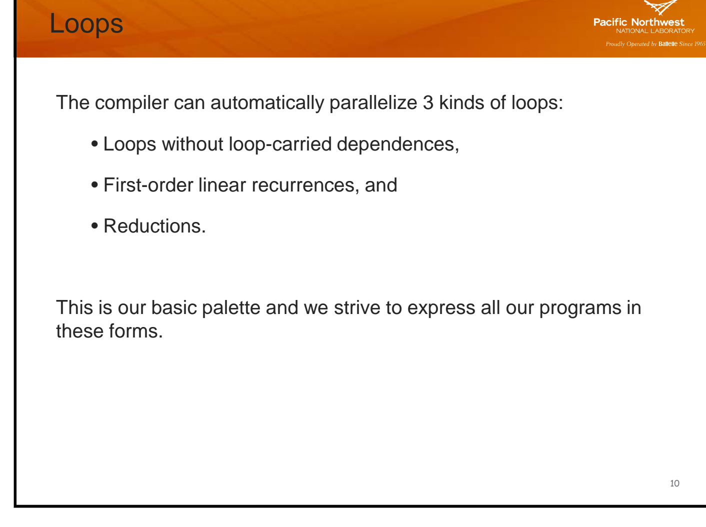

Barrel processor
Pacific Northwest NATIONAL LABORATORY
Many threads per processor core
Thread-level context switch at every instruction cycle
registers
ALU
program counter
COTS
"stream"
Multithreaded
5

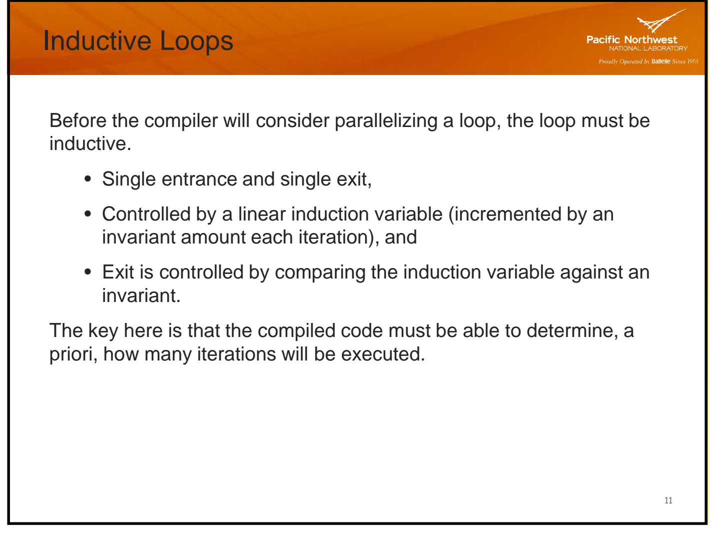

Multithreading
Pacific Northwest NATIONAL LABORATORY
Hide latencies via parallelism
Maintain multiple active threads per processor, so that gaps introduced by long latency operations in one thread are filled by instructions in other threads
6


Data parallelism
Pacific Northwest NATIONAL LABORATORY
Loop over data elements
Work on each element in parallel
Best source of parallelism in almost all programs
Always think data parallel first
FOR ALL GUESTS, GUESTS[i]
FOR ALL DOCUMENTS, LIBRARY[i]
FOR ALL ATOMS, ATOMS[i]
FOR ALL NODES, GRAPH[i]
FOR ALL ELEMENTS, A[i, j] // ORDER N^2 PARALLELISM


Simple example
Pacific Northwest NATIONAL LABORATORY
for (i = 0; i < n; i++) {c[i] = a[i] + b[i];}
Single thread
Parallel Region
Multiple threads
Barrier







Examples of for loops

Pacific Northwest NATIONAL LABORATORY

```
for (i = 0; i < n; i++) b[i] = (a[i] + a[i + 1])) / 2;

for (i = 0; i < n; i*=2) b[i] = (a[i] + a[i + 1])) / 2;

for (i = 0; i < n; i++)
  if (c[i] == 0] break; else d[i] = 1.0 / c[i];

for (i = 0; i < n; i++)
  for (j = 0; j < A[i]; j++)
    B[i,j] = C[i] + D[j];
```

Which loops are legal ??

Matrix multiplication

```
void matmult(int n, int m, double **a, double **b, double **c) {

  for (int i = 0; i < n; i ++) {
  for (int j = 0; j < n; j ++) {

    double sum = 0.0;
    for (int k = 0; k < m; k++) sum += a[i][k] * b[k][j];

    c[i][j] = sum;

} } }
```

Check compilation

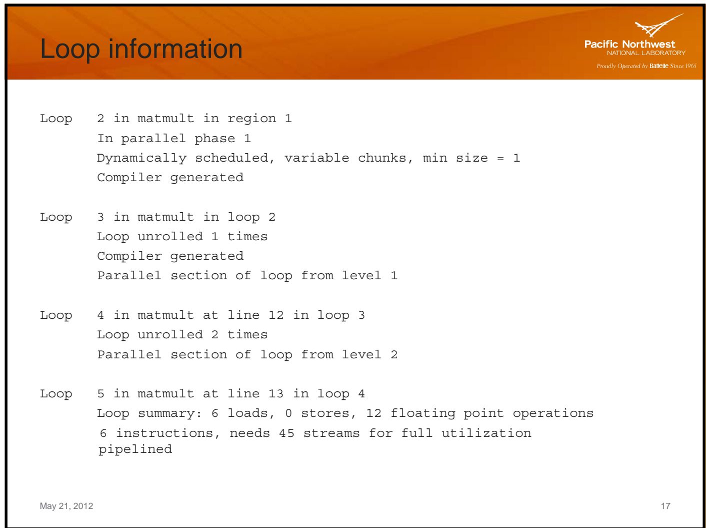
```
           |    for (int i = 0; i < n; i ++) {
           |    for (int j = 0; j < n; j ++) {
   3 --    |        double sum = 0.0;
   5 --P:$ |        for (int k = 0; k < m; k++) sum += a[i][k] * b[k][j];
   3 --    +
** reduction moved out of 1 loop
   3 SS    |        c[i][j] = sum;
           | }  } }
```

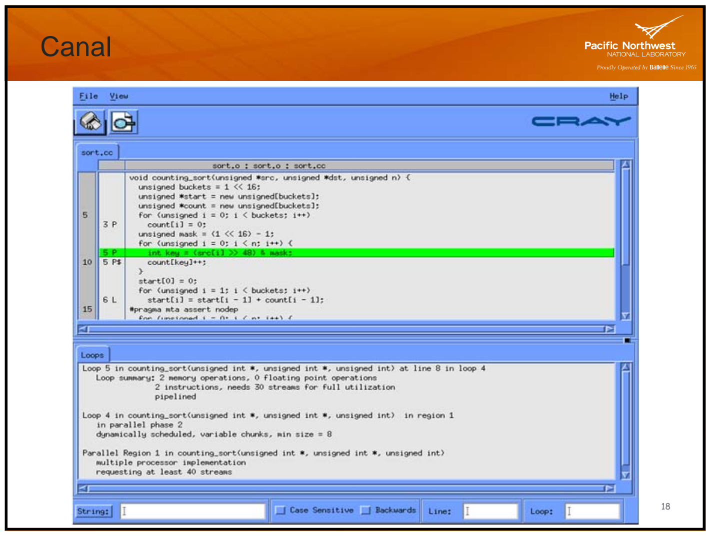

Need to use pragmas

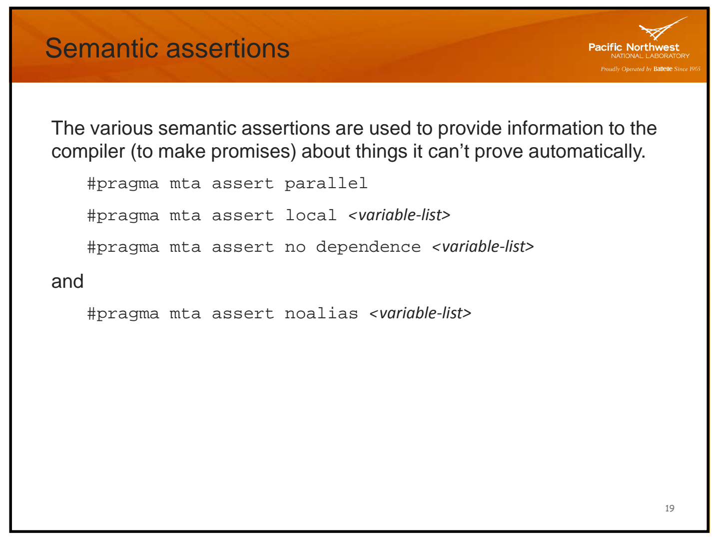
```
void matmult(int n, int m, double **a, double **b, double **c) {

  #pragma mta assert no dependence
  for (int i = 0; i < n; i ++) {

  #pragma mta assert no dependence
  for (int j = 0; j < n; j ++) {

    double sum = 0.0;
    for (int k = 0; k < m; k++) sum += a[i][k] * b[k][j];

    c[i][j] = sum;

} } }
```

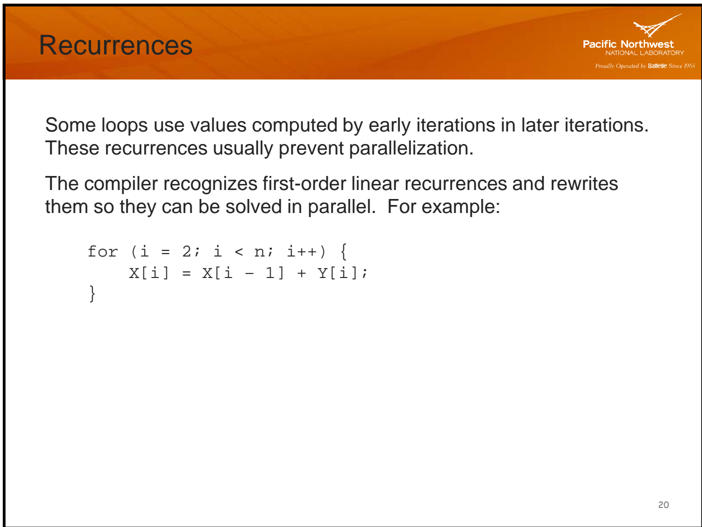
What you want to see

```
           | #pragma mta assert no dependence
           |    for (int i = 0; i < n; i ++) {
           | #pragma mta assert no dependence
           |    for (int j = 0; j < n; j ++) {
   4 PP    |        double sum = 0.0;
   5 PP-   |        for (int k = 0; k < m; k++) sum += a[i][k] * b[k][j];
   4 PP    +
   4 PP    |        c[i][j] = sum;
           | }  } }
```


Loop information
Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965
Loop 2 in matmult in region 1
In parallel phase 1
Dynamically scheduled, variable chunks, min size = 1
Compiler generated
Loop 3 in matmult in loop 2
Loop unrolled 1 times
Compiler generated
Parallel section of loop from level 1
Loop 4 in matmult at line 12 in loop 3
Loop unrolled 2 times
Parallel section of loop from level 2
Loop 5 in matmult at line 13 in loop 4
Loop summary: 6 loads, 0 stores, 12 floating point operations
6 instructions, needs 45 streams for full utilization
pipelined
May 21, 2012
17


Canal
Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965
File View Help
CRAY
sort.cc
sort.o : sort.o : sort.cc
void counting_sort(unsigned *src, unsigned *dst, unsigned n) {
unsigned buckets = 1 << 16;
unsigned *start = new unsigned[buckets];
unsigned *count = new unsigned[buckets];
5 for (unsigned i = 0; i < buckets; i++)
3 P count[i] = 0;
unsigned mask = (1 << 16) - 1;
for (unsigned i = 0; i < n; i++) {
5 P int key = (src[i] >> 48) & mask;
10 5 P$ count[key]++;
}
start[0] = 0;
for (unsigned i = 1; i < buckets; i++)
6 L start[i] = start[i - 1] + count[i - 1];
15 #pragma mta assert nodep
Loops
Loop 5 in counting_sort(unsigned int *, unsigned int *, unsigned int) at line 8 in loop 4
Loop summary: 2 memory operations, 0 floating point operations
2 instructions, needs 30 streams for full utilization
pipelined
Loop 4 in counting_sort(unsigned int *, unsigned int *, unsigned int) in region 1
in parallel phase 2
dynamically scheduled, variable chunks, min size = 8
Parallel Region 1 in counting_sort(unsigned int *, unsigned int *, unsigned int)
multiple processor implementation
requesting at least 40 streams
String: Case Sensitive Backwards Line: Loop:
18


Semantic assertions
Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965
The various semantic assertions are used to provide information to the compiler (to make promises) about things it can't prove automatically.
#pragma mta assert parallel
#pragma mta assert local <variable-list>
#pragma mta assert no dependence <variable-list>
and
#pragma mta assert noalias <variable-list>
19


Recurrences
Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965
Some loops use values computed by early iterations in later iterations. These recurrences usually prevent parallelization.
The compiler recognizes first-order linear recurrences and rewrites them so they can be solved in parallel. For example:
for (i = 2; i < n; i++) {
X[i] = X[i - 1] + Y[i];
}
20

Reductions

Pacific Northwest NATIONAL LABORATORY

A particularly simple form of recurrence is the reduction.

```
double s = 0.0;
for (i = 1; i < n; i++) {
  s = s + X[i] * Y[i];
}

double Max = X[0];
for (i = 1; i < n; i++) {
  if (X[i] > Max) Max = X[i];
}

int    ndx = 0;
double Max = X[0];
for (i = 1; i < n; i++) {
  if (X[i] > Max) {Max = X[i]; ndx = i;}
}
```

21

Example of atomic

Pacific Northwest NATIONAL LABORATORY

```
for (i = 0; i < nBonds; i++) {
  int headNode = Bond[i][0];
  int tailNode = Bond[i][1];

  double X_Force = … x force calculation;
  double Y_Force = … y force calculation;
  double Z_Force = … z force calculation;

  Force[headNode][0] += X_Force;
  Force[headNode][1] += Y_Force;
  Force[headNode][2] += Z_Force;

  Force[tailNode][0] += X_Force;
  Force[tailNode][1] += Y_Force;
  Force[tailNode][2] += Z_Force;
}  /* End of parallel regions -- */
```

Synchronized memory operations

Pacific Northwest NATIONAL LABORATORY

- Every memory has a full-and-empty bit
- *normal* loads and stores disregard the full-empty bit
- *sync loads* wait for the full-empty bit to be full, then set it full (multiple readers) or empty (single reader)
  - readff(&addr)   readfe(&addr)
- *sync stores* wait for the full-empty bit to empty, then set it full
  - writeef(&addr, value)
- *sync* and *normal* memory operations take the "same time"

Waiting consumes no processor cycles

23

Keeping the machine busy

Pacific Northwest NATIONAL LABORATORY

To help keep the machine busy, we like to exploit multiple levels of parallelism. Consider a matrix-vector product:

```
for (i = 1; i < m; i++) {
for (j = 1; j < n; j++) {
  Y[i] = Y[i] + X[j] * A[i][j];
} }
```

If we (or the compiler) run the outer loop in parallel, we'll do well when *m* is large, but poorly when *m* is small.

Similarly, if we run the inner loop in parallel, we'll do well only when *n* is large.

24

Nested parallelism

The compiler handles such cases by collapsing the loops, yielding code that looks something like this:

```
for (ij = 1; ij < m*n; ij++) {
    i = ij / n;
    j = ij % n;
    Y[i] = Y[i] + X[j] * A[i][j];
}
```

By using this approach, we get good parallelism whenever *m* * *n* is large.

25

Recursion

Recursion is a parallel programming method that can quickly saturate the machine.

XMT programming model supports *futures*

Easy and cost effective; however, each spawn must have enough work to justify the overhead.

Terminate recursion at low levels with insufficient work

26

The `future` construct

```
future x$(i) {
  …
  …
  …
}
```

- the statement purges `x$` (sets empty)
- arguments are passed by value
- enqueued and executed asynchronously in ~FIFO order
- the return value is stored to `x$`

27

Tree search (simple)

```
struct Tree {
  Tree *llink;
  Tree *rlink;
  int data;
};

int search_tree(Tree *root, int target) {
  future int left$, right$;
  if (root == 0) return 0;

  future left$(root, target)
      { return search_tree(root->llink, target); }
  future right$(root, target)
      { return search_tree(root->rlink, target); }

  int sum = (root->data == target);
  return sum + left$ + right$;
}
```

28

Tree search (better)

```
struct Tree {
  Tree *llink;
  Tree *rlink;
  int data;
};

int search_tree(Tree *root, int target) {
  future int left$;
  if (root == 0) return 0;

  future left$(root, target)
      { return search_tree(root->llink, target); }

  int right = search_tree(root->rlink, target);

  int sum = (root->data == target);
  return sum + left$ + right$;
}
```

29

Tree search (best)

```
struct Tree {
  Tree *llink;
  Tree *rlink;
  int data;
};

int search_tree(Tree *root, int target) {
  future int left$;
  if (root == 0) return 0;

  future left$(root, target)
      { return search_tree(root->llink, target); }

  int right = search_tree(root->rlink, target);

  int sum = (root->data == target);
  return sum + touch(left$) + right$;
}
```

30