Programming Irregular Applications with OpenMP*

Tim Mattson
Intel Corp.
timothy.g.mattson@ intel.com

Alice Koniges
Berkeley Lab/NERSC
AEKoniges@lbl.gov

Clay Breshears
PAPPS
clay.breshears@gmail.com

Jeremy Kemp
University of Houston
jakemp@uh.edu

1

Preliminaries: Part 1

- Disclosures
  - The views expressed in this tutorial are those of the people delivering the tutorial.
    - We are not speaking for our employers.
    - We are not speaking for the OpenMP ARB
- We take these tutorials VERY seriously:
  - Help us improve … tell us how you would make this tutorial better.

Hands-on Instructions

- We have reserved 200 nodes of Cori and 100 nodes of Edison at NERSC for this tutorial
- Please be sure to send a “THANKS NERSC” email if you appreciate this
- Your training accounts will give you access to our reservation
- Please type your password very carefully, because you will get cut off after 3 tries
- For Cori we have Haswell nodes of Cori Phase 1
  - Cori is a brand new XC40
- Edison is a Cray XC30 with a peak performance of more than 2 petaflops. Edison features the Cray Ariesinterconnect, Intel Xeon processors, 64 GB of memory per node.
- For details, see NERSC Web

Finding the web pages

- For the instructions, please go to the following:
  - www.nersc.gov
  - users
  - software
  - programming-models
  - openmp
  - sc16-openmp-tutorial

- https://www.nersc.gov/users/software/programming-models/openmp/sc16-openmp-tutorial/

Preliminaries: Part 2

- Our plan for the day .. Active learning!
  - We will mix short lectures with short exercises.
  - You will use your laptop to connect to a multiprocessor server.
- Please follow these simple rules
  - Do the exercises that we assign and then change things around and experiment.
    - Embrace active learning!
  - Don't cheat: Do Not look at the solutions before you complete an exercise … even if you get really frustrated.

Outline

- OpenMP overview
- Introducing Explicit Tasks in OpenMP
- Working with Tasks
- Tasks and the conceptual core of OpenMP
- Break
- Working with tasks: the divide and conquer pattern
- Advanced tasking features

OpenMP* overview:

C$OMP FLUSH

#pragma omp critical

C$OMP THREADPRIVATE(/ABC/)

CALL OMP_SET_NUM_THREADS(10)

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

OpenMP: An API for Writing Multithreaded Applications

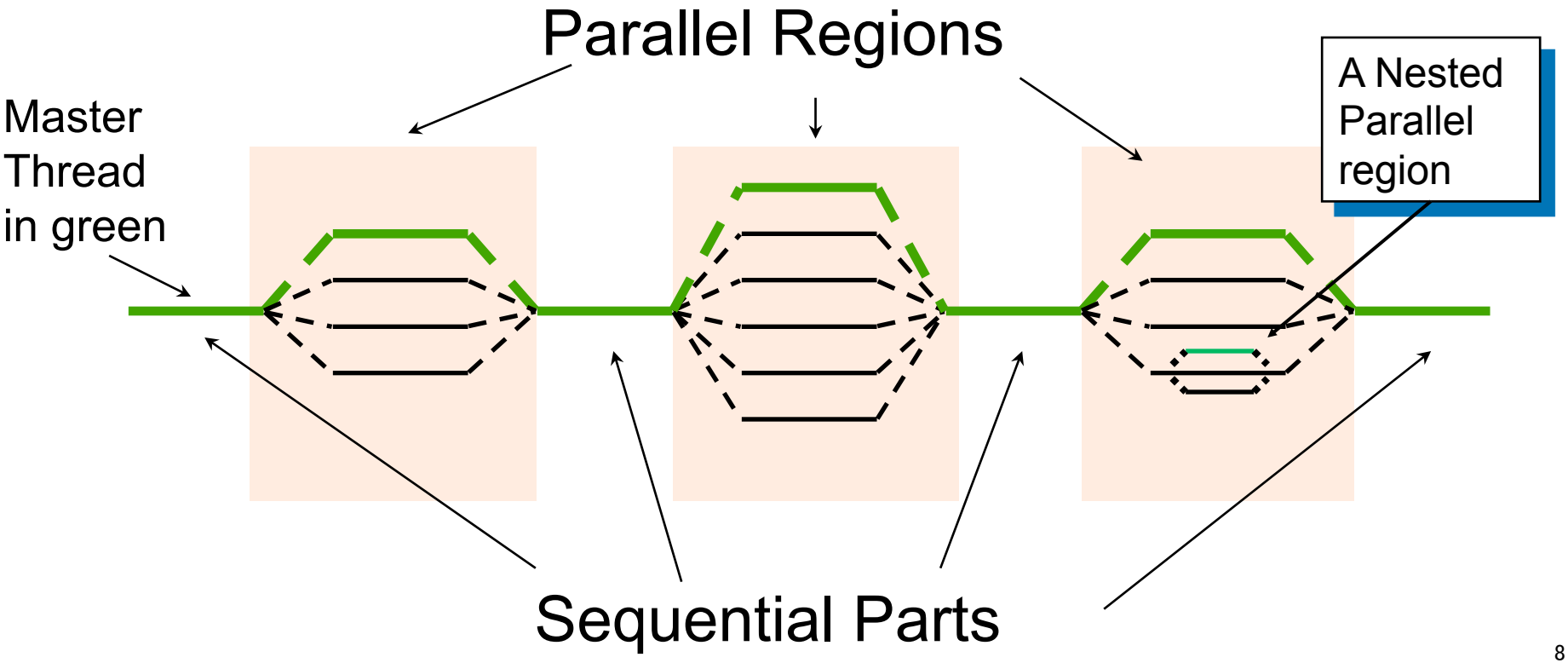
- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

OpenMP Execution Model:

Fork-Join pattern:

- **Master thread** spawns a **team of threads** as needed.
- Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.



Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create a 4 thread Parallel region:

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

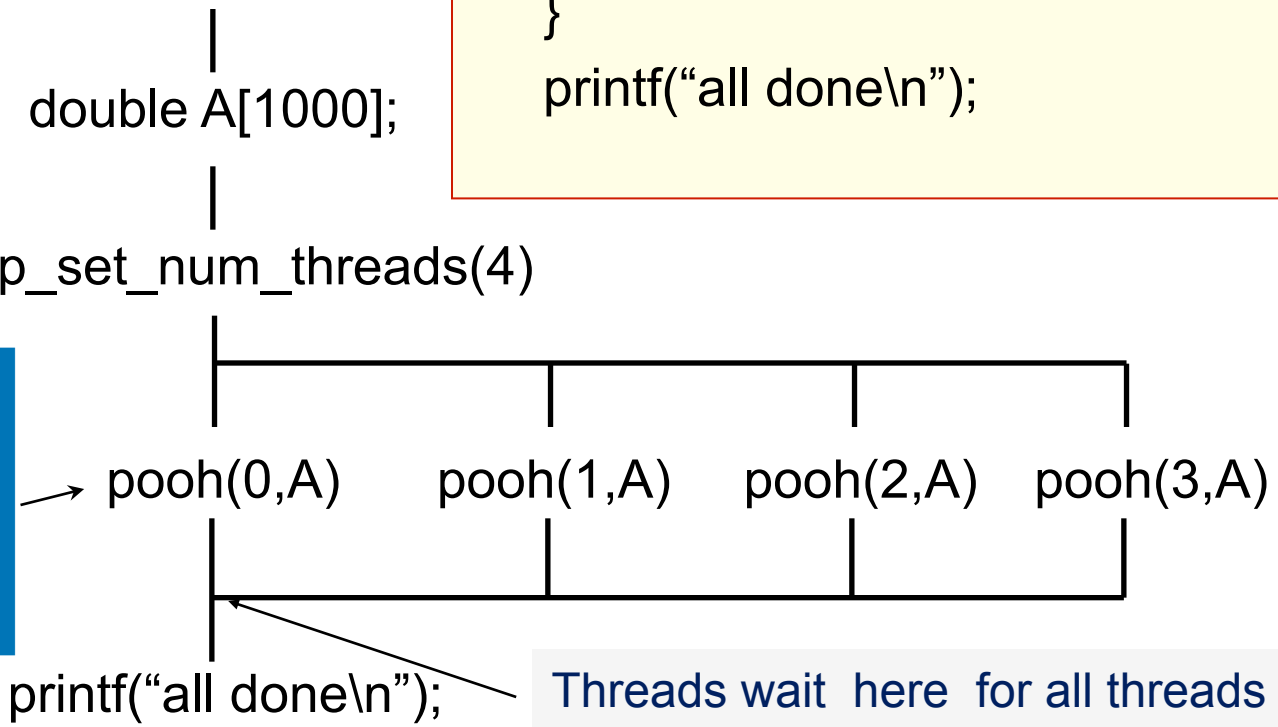
Runtime function returning a thread ID

- Each thread calls pooh(ID,A) for `ID = 0` to `3`

Thread Creation: Parallel Regions

- Each thread executes the same code redundantly.

```
double A[1000];
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

OpenMP: what the compiler does

```
#pragma omp parallel num_threads(4)
{
    foobar ();
}
```

- The OpenMP compiler generates code logically analogous to that on the right of this slide, given an OpenMP pragma such as that on the top-left
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for reach parallel region.
- Only three threads are created because the last parallel section will be invoked from the parent thread.

```
void thunk ()
{
    foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
        &tid[i],0,thunk, 0);
thunk();

for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```

OpenMP data environment - motivation

When operating in parallel – proper sharing, or NOT sharing is essential to correctness and performance.

```
#pragma omp parallel for
{
        for(i=0; i<n; i++){
                tmp= 2.0*a[i];
                a[i] = tmp;
                b[i] = c[i]/tmp;
        }
}
```

```
#pragma omp parallel for private(temp)
{
        for(i=0; i<n; i++){
                tmp= 2.0*a[i];
                a[i] = tmp;
                b[i] = c[i]/tmp;
        }
}
```

By default, all threads share a common address space. Therefore, all threads will be modifying *tmp* simultaneously in the code on the LEFT.

On the RIGHT –**private** clause directs that each thread will have an (uninitialized) private copy.

Initialization is possible with "**firstprivate**" and grabbing the last value is possible with "**lastprivate**." **Reduction**s are important enough to have a special clause, and **default**s can be set (including to "none.")

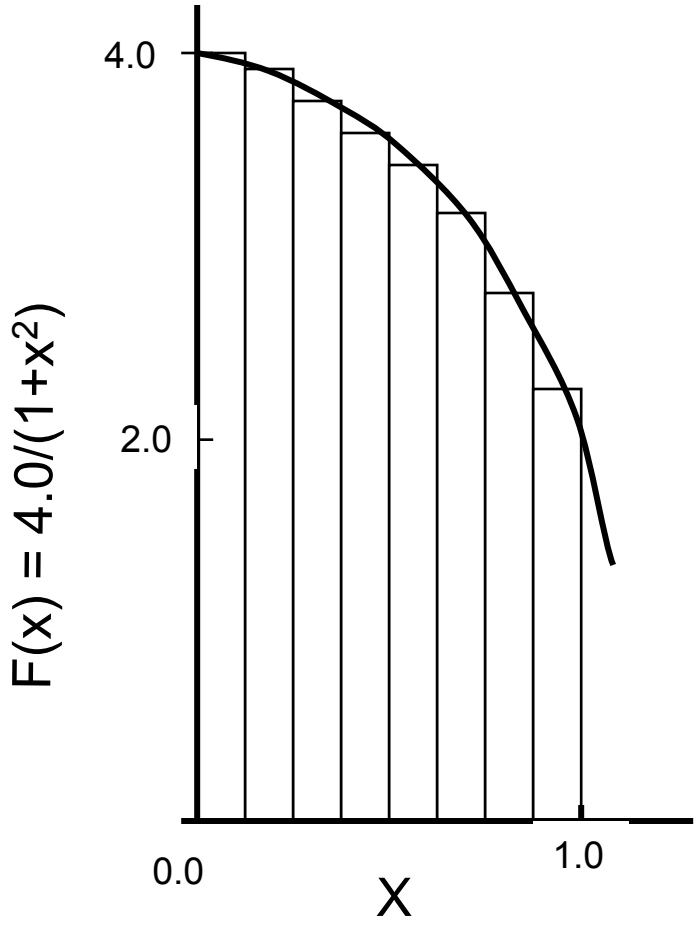
OpenMP data environment - summary

1. Variables are shared by default.
2. Global variables are shared by default.
3. Automatic variables within subroutines called from within a parallel region are private (reside on a stack private to each thread), unless scoped otherwise.
4. Default scoping rule can be changed with default clause.

Data scope attribute clause description
private clause: declares the variables in the list to be private (not shared) to each thread.
firstprivate clause: declares variables in the list to be **private** *plus* the private variables are initialized to the value of the variable when the construct is encountered (”entered”).
lastprivate clause: declares variables in the list to be **private** *plus* the value of from the sequentially last iteration of the associated loops, or the lexically last section construct, is assigned to the original list item(s) after the end of the construct.
shared clause: declares the variables in the list to be shared among all the threads in a team. All threads within a team access the same storage area for shared variables. Synchronization is generally advised if variables are updated.
reduction clause: performs a reduction on the scalar variables that appear in the list, with a specified operator.
default clause: allows the user to affect the data-sharing attribute of the variables appeared in the parallel construct.

A recurring example:

Numerical integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI program

```
static long num_steps = 100000;
double step;
int main ()
{           int i;      double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

See OMP_exercises/pi.c

Example: Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{    int i;            double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      #pragma omp parallel
      {
          double x;
         #pragma omp for reduction(+:sum)
             for (i=0;i< num_steps; i++){
                    x = (i+0.5)*step;
                    sum = sum + 4.0/(1.0+x*x);
             }
       }
            pi = step * sum;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop index is local to a thread by default.

Results*: pi with a loop and a reduction

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{   int i;          double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double x;
      #pragma omp for reduction(+:sum)
          for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
          }
    }
        pi = step * sum;
}
```

threads	PI Loop
1	1.91
2	1.02
3	0.80
4	0.68

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 GHz and 4 Gbyte DDR3 memory at 1.333 GHz.

Exercise: matrix multiplication

- We have provided a simple matrix multiplication program matmult.c
- Build and run this serial program
  - Make matmult
  - ./matmult 100 200 300
- Parallelize the program using OpenMP.

Outline

- OpenMP overview
- Introducing Explicit Tasks in OpenMP
- Working with Tasks
- Tasks and the conceptual core of OpenMP
- Break
- Working with tasks: the divide and conquer pattern
- Advanced tasking features

Not all programs have simple loops OpenMP can parallelize

- Consider a program to traverse a linked list:

```
p=head;
while (p) {
      processwork(p);
      p = p->next;
}
```

- OpenMP can only parallelize loops in a basic standard form with loop counts known at runtime

Linked lists with parallel loops

```
while (p != NULL) {
     p = p->next;
      count++;
}
p = head;
for(i=0; i<count; i++) {
      parr[i] = p;
      p = p->next;
   }
#pragma omp parallel
{
    #pragma omp for schedule(static,1)
    for(i=0; i<count; i++)
       processwork(parr[i]);
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

	Default schedule	Static,1
One Thread	48 seconds	45 seconds
Two Threads	39 seconds	28 seconds

Results on an Intel dual core 1.83 GHz CPU, Intel IA-32 compiler 10.1 build 2

Linked lists with parallel loops

```
while (p != NULL) {
     p = p->next;
      count++;
}
p = head;
for(i=0; i<count; i++) {
      parr[i] = p;
      p = p->next;
   }
#pragma omp parallel
{
    #pragma omp for schedule(static,1)
    for(i=0; i<count; i++)
       processwork(parr[i]);
}
```

Count number of items in the linked list

There has got to be a better way!!!

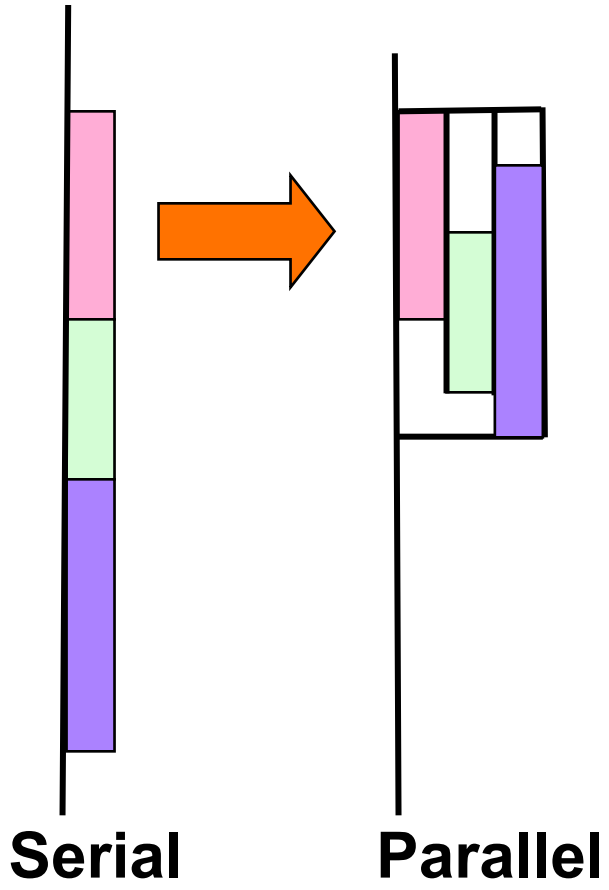
Process nodes in parallel with a for loop

	Default schedule	Static,1
One Thread	48 seconds	45 seconds
Two Threads	39 seconds	28 seconds

Results on an Intel dual core 1.83 GHz CPU, Intel IA-32 compiler 10.1 build 2

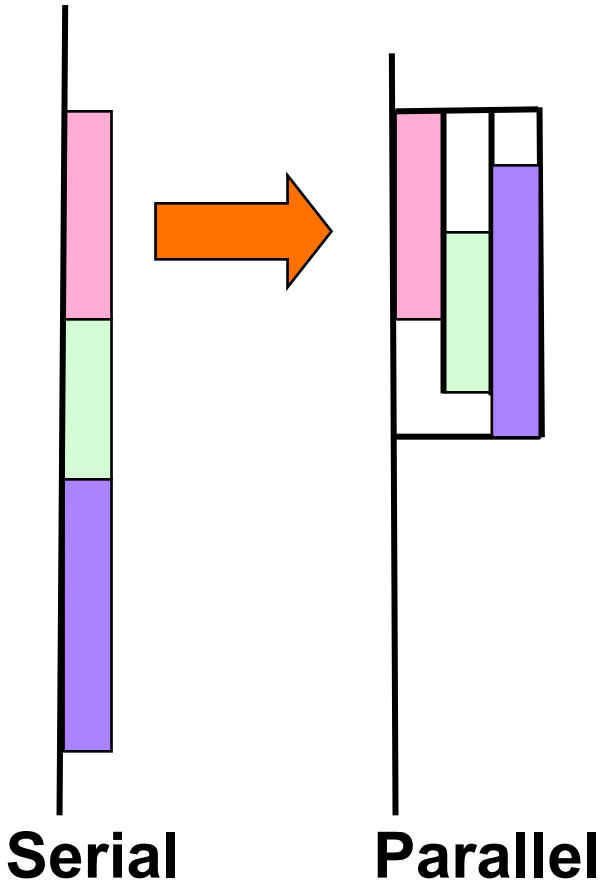
What are tasks?

- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data to compute with
- Threads are assigned to perform the work of each task.
  - The thread that encounters the task construct may execute the task immediately.
  - The threads may defer execution until later



What are tasks?

- The task construct includes a structured block of code
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- Tasks can be nested: i.e. a task may itself generate tasks.



Task Directive

`#pragma omp task` *[clauses]*

structured-block

```
#pragma omp parallel
{
  #pragma omp master
   {
     #pragma omp task
        fred();
     #pragma omp task
        daisy();
     #pragma omp task
        billy();
   }
}
```

Create some threads

Thread 0 packages tasks

Tasks executed by some thread in some order

All tasks complete before this barrier is released

Exercise: Simple tasks

- Write a program using tasks that will “randomly” generate one of two strings:
  - I think race cars are fun
  - I think car races are fun
- Hint: use tasks to print the indeterminate part of the output (i.e. the “race cars” or “car races” part).

- This is called a “Race Condition”. It occurs when the result of a program depends on how the OS schedules the threads.
- NOTE: A “data race” is when threads “race to update a shared variable”. They produce race conditions. Programs containing data races are undefined (in OpenMP but also ANSI standards C++’11 and beyond).

```
#pragma omp parallel
#pragma omp task
#pragma omp master
#pragma omp single
```

Outline

- OpenMP overview
- Introducing Explicit Tasks in OpenMP
- Working with Tasks
- Tasks and the conceptual core of OpenMP
- Break
- Working with tasks: the divide and conquer pattern
- Advanced tasking features

When/where are tasks complete?

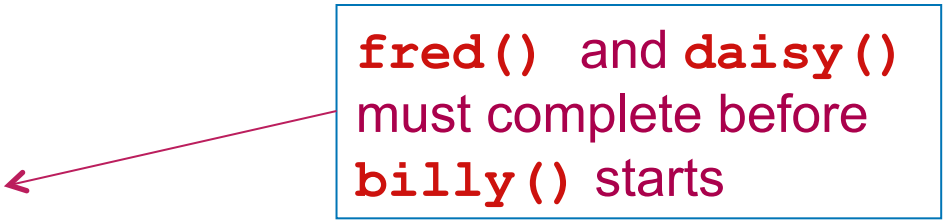
- At thread barriers (explicit or implicit)
  - applies to all tasks generated in the current parallel region up to the barrier

- At taskwait directive
  - i.e. Wait until all tasks defined in the current task have completed.
  - Fortran: **`!$OMP TASKWAIT`**
  - C/C++: **`#pragma omp taskwait`**

  - Note: applies only to tasks generated in the current task, not to “descendants” .
  - The code executed by a thread in a parallel region is considered a task here

Example

```
#pragma omp parallel
{
  #pragma omp master
   {
     #pragma omp task
        fred();
     #pragma omp task
        daisy();
     #pragma taskwait
     #pragma omp task
        billy();
   }
}
```



The task construct (OpenMP 4.5)

#pragma omp task *[clause[[,]clause]...]*
structured-block

Generates an explicit task

where *clause* is one of the following:

if(*[* **task :]***scalar-expression***)**

untied

default(shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

final(scalar-expression)

mergeable

depend(*dependence-type* **:** *list***)**

priority(*priority-value***)**

The evolution of the task construct

OpenMP 3.0 (May'08)

OpenMP 3.1 (Jul'11)

OpenMP 4.0 (Jul'13)

OpenMP 4.5 (Nov'15)

The task construct (OpenMP 4.5)

#pragma omp task *[clause[[,]clause]...]*
structured-block

Generates an explicit task

where *clause* is one of the following:

if(*[* **task :]***scalar-expression***)**

untied

default(shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

final(scalar-expression)

mergeable

depend(*dependence-type* **:** *list***)**

priority(*priority-value***)**

Consider the data environment associated with a task

The evolution of the task construct

OpenMP 3.0

OpenMP 3.1

OpenMP 4.0

OpenMP 4.5

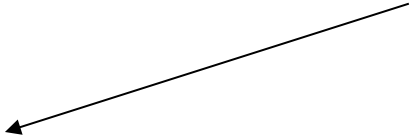
Data scoping with tasks

- Variables can be shared, private or firstprivate with respect to task
- These concepts are a little bit different compared with threads:
  - If a variable is shared on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered
  - If a variable is private on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed
  - If a variable is firstprivate on a construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered

Data scoping defaults

- The behavior you want for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope)
  - Variables that are private when the task construct is encountered are firstprivate by default
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default

```
#pragma omp parallel shared(A) private(B)
{
    ...
#pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```



A is shared
B is firstprivate
C is private

Example: Fibonacci numbers

```
int fib (int n)
{
  int x,y;
  if (n < 2) return n;

  x = fib(n-1);
  y = fib (n-2);
  return (x+y);
}

Int main()
{
  int NW = 5000;
  fib(NW);
}
```

- $F_n = F_{n-1} + F_{n-2}$
- Inefficient $O(n^2)$ recursive implementation!

Parallel Fibonacci

```
int fib (int n)
{   int x,y;
  if (n < 2) return n;

#pragma omp task shared(x)
  x = fib(n-1);
#pragma omp task shared(y)
  y = fib (n-2);
#pragma omp taskwait
  return (x+y);
}

Int main()
{  int NW = 5000;
  #pragma omp parallel
  {
     #pragma omp master
          fib(NW);
  }
}
```

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete (enforced with taskwait)
- `x,y` are local, and so by default they are private to current task
  - must be shared on child tasks so they don't create their own firstprivate copies at this level!

Using tasks

- Getting the data attribute scoping right can be quite tricky
  - default scoping rules different from other constructs
  - as ever, using `default(none)` is a good idea

- Don't use tasks for things already well supported by OpenMP
  - e.g. standard do/for loops
  - the overhead of using tasks is greater

- Don't expect miracles from the runtime
  - best results usually obtained where the user controls the number and granularity of tasks

Exercise: Traversing linked lists

- Consider the program linked.c
  - Traverses a linked list, computing a sequence of Fibonacci numbers at each node
- Parallelize this program using OpenMP tasks

```
#pragma omp parallel
#pragma omp task
#pragma omp taskwait
#pragma omp master
#pragma omp single
```

Linked lists with tasks

```
#pragma omp parallel
{
  #pragma omp single
  {
     p=head;
    while (p) {
      #pragma omp task firstprivate(p)
            processwork(p);
        p = p->next;
    }
  }
}
```

Creates a task with its own copy of “p” initialized to the value of “p” when the task is defined

Parallel linked list traversal

Thread 0:

```
p = listhead ;
while (p) {
< package up task >
   p=next (p) ;
}

while (tasks_to_do){
  < execute task >
}

< barrier >
```

Other threads:

```
while (tasks_to_do) {
< execute task >
}

< barrier >
```

Parallel pointer chasing on multiple lists

```
#pragma omp parallel
{
    #pragma omp for private(p)
    for ( int i =0; i <numlists; i++) {
        p = listheads[i] ;
        while (p ) {
        #pragma omp task firstprivate(p)
            {
              process(p);
            }
        p=next(p);
        }
    }
}
```

All threads package tasks

Outline

- OpenMP overview
- Introducing Explicit Tasks in OpenMP
- Working with Tasks
- Tasks and the conceptual core of OpenMP
- Break
- Working with tasks: the divide and conquer pattern
- Advanced tasking features

The OpenMP specification

- A specification is written for implementers, not users.
- There is a great deal of low level detail in a spec that helps prevent ambiguity even in strange “corner cases” most users never encounter.
  - Users can usually ignore such “corner cases”, implementers cannot.
- The OpenMP specification opens with a jargon-rich discussion that confuses most users and hence is ignored in OpenMP tutorials.
- But as you move from OpenMP-novice to OpenMP-expert, you need to absorb that content.

Execution model jargon

- A program begins execution as a single thread … the **initial thread**.
- Initial thread executes sequentially within an **implicit parallel region** which defines a task region called the **initial task region**
- When a parallel region is encountered:
  - The task region of the thread encountering the parallel construct is suspended.
  - Team of threads is created with the thread encountering the parallel construct becoming the **master** of the new team.
  - Each thread in the team runs an **implicit task** (one per thread, a tied task)
  - When the team of threads complete, the task region associated with the master continues as that thread (and only that thread) proceeds beyond the barrier.

Yuck … why all this complicated jargon? Because now we can define the data environment precisely in terms of task regions and we cleanly cover all the corner cases within a single, task based structure.

Data Environments and Tasks

- The data environment consists of:
  - Variables: a named data storage block the value of which can change as a program runs
  - Internal control variables: Conceptual variables that specify runtime behavior of threads and tasks
  - Thread private variables: a variable replicated with one instance per thread which provides access to a different block of storage per thread.
- A task is … a specific instance of executable code and its data environment

By defining OpenMP constructs in terms of tasks, we only have to define the data environment concepts once … not separately for each type of construct.
This brings a level of consistency that greatly helps people who must implement OpenMP.

Outline

- OpenMP overview
- Introducing Explicit Tasks in OpenMP
- Working with Tasks
- Tasks and the conceptual core of OpenMP
- Break
- Working with tasks: the divide and conquer pattern
- Advanced tasking features

Divide and Conquer Pattern

- Divide and conquer is an important design pattern with two distinct phases
  - Use when a method to divide problem into subproblems and to recombine solutions of subproblems into a global solution is available
- Divide phase:
  - Breaks down problem into two or more sub-problems of the same (or related) type
  - Continue division until these sub-problems become simple enough to be solved directly
- Conquer phase
  - Executes the computations on each of the “indivisible” sub-problems.
  - May also combine solutions of sub-problems until the solution of the original problem is reached.

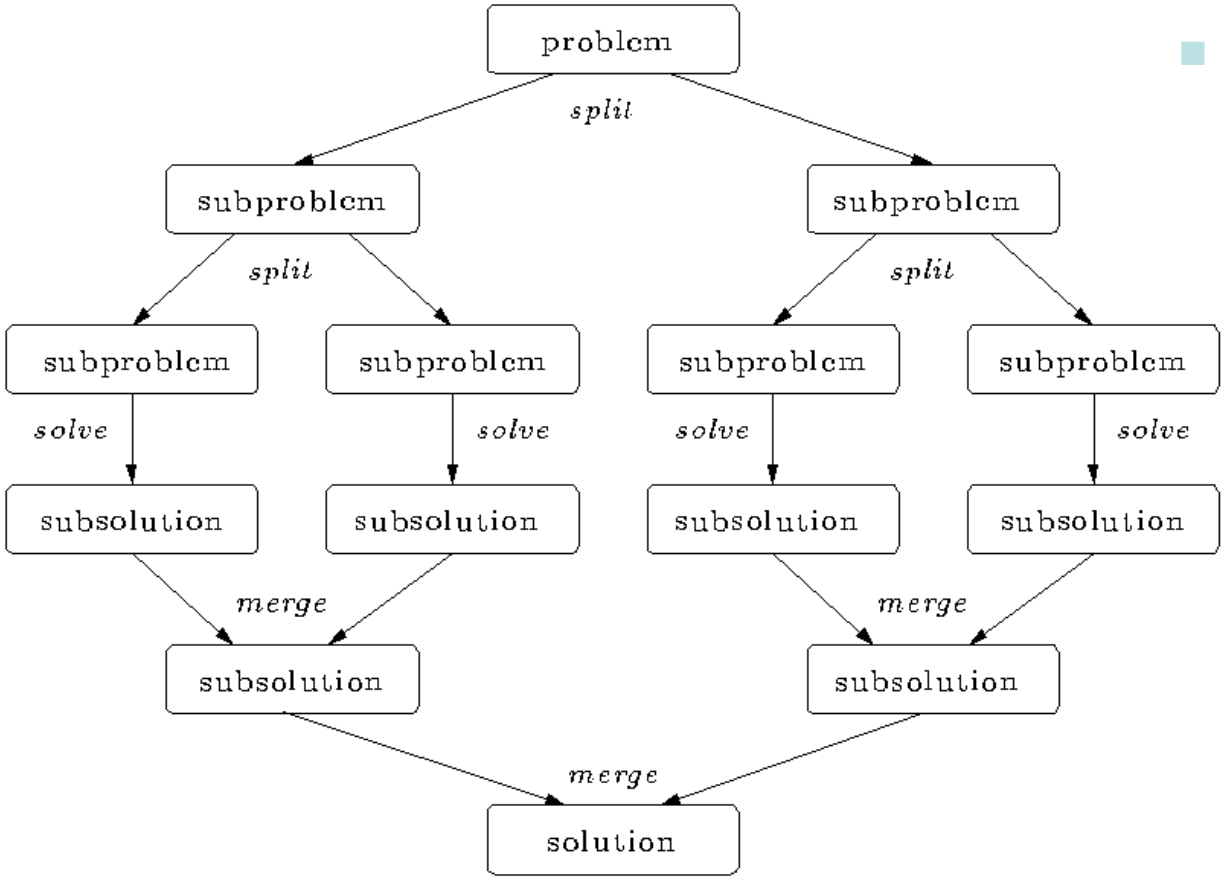
Divide and Conquer Pattern

- Implementation is typically done with recursive algorithms
  - The nature of recursion forms smaller sub-problems that are very much like the larger problem being solved
  - The return from recursive calls can be used to combine partial solutions into an overall solution
- Coding Solution
  - Define a split operation
  - Continue to split the problem until subproblems are small enough to solve directly
  - Recombine solutions to subproblems to solve original global problem

- Note:
  - Computing may occur at any operation phase (split, direct solution, recombination)

Divide and conquer

- Split the problem into smaller sub-problems; continue until the sub-problems can be solve directly



- 3 Options:
  - Do work as you split into sub-problems
  - Do work only at the leaves
  - Do work as you recombine

Quicksort

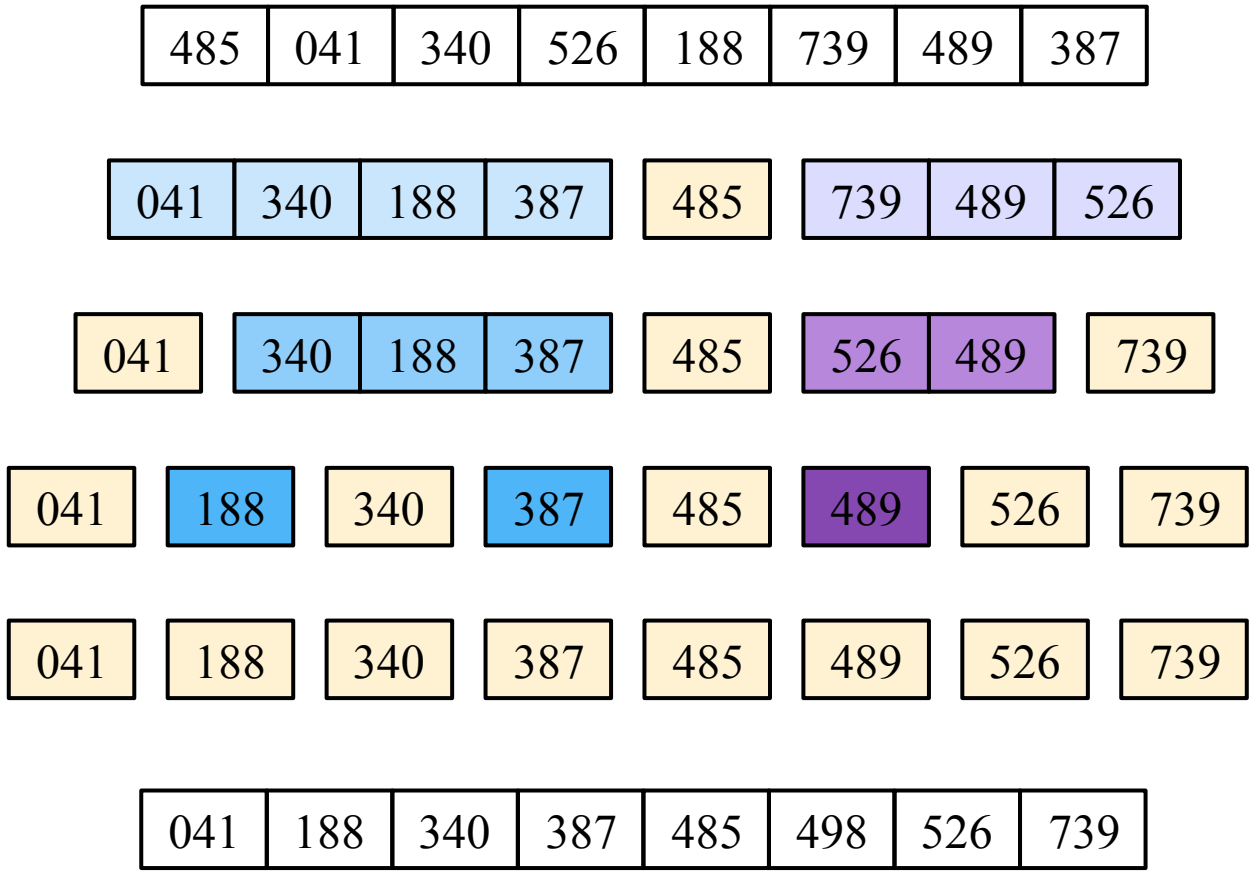
- Serial Quicksort algorithm is implemented recursively
- Given: Unsorted array of elements (each with assoc. key)
- Goal: Sorted array of elements
- Divide phase:
  - In each unsorted sub-array, choose an element from as the “pivot” element
  - Partition the array contents such that the pivot item ends up at the point that divides the array into elements that are less than or equal to the pivot item and elements that are greater than the pivot item
- Conquer Phase:
  - Pivot element has been placed in sorted position after each partition
- Worst case complexity: $O(n^2)$
- Average case complexity: O(n *log* n)

Quicksort Algorithm

```
void QuickSort(int *A, int p, int r)
{
  if (p < r) {
    int q = Partition(A, p, r);
    QuickSort (A, p, q-1);
    QuickSort (A, q+1, r);
  }
}
```

- `Partition()` compares all items against “pivot”
  - Linear search through array (serial)
  - Moves items less than pivot, greater than pivot

Quicksort Illustration



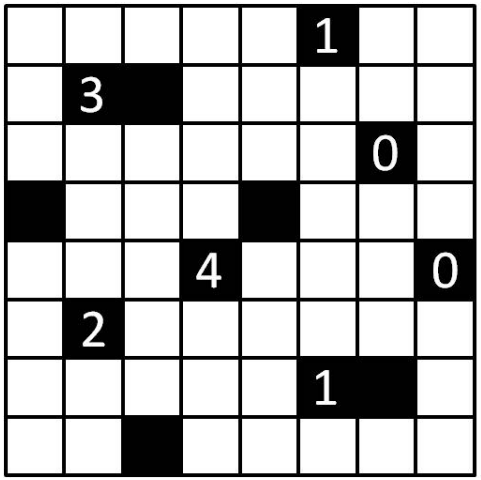
Quicksort Code

```
void QuickSort(int *A, int p, int r)
{
  if (r-p <= 1)
    return; /* List of length one or zero */
  else {
    int q = Partition(A, p, r); /* Find pivot */
#pragma omp task
    QuickSort (A, p, q-1);
#pragma omp task
    QuickSort (A, q+1, r);
  }
}

. . .

#pragma omp parallel
{
#pragma omp single
  QuickSort(A, 0, N-1);
}
```

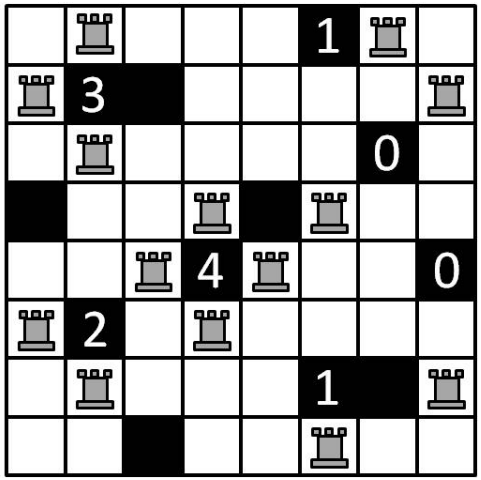
Exercise: Akari

- Japanese logic puzzle from Nikoli
- Goal: Place chess rooks on open squares such that
  - No two rooks attack each other
  - Numbered squares surrounded by specified number of rooks
  - All open squares are “covered” by one or more rooks
  - Black squares block attack of rooks

Initial Configuration



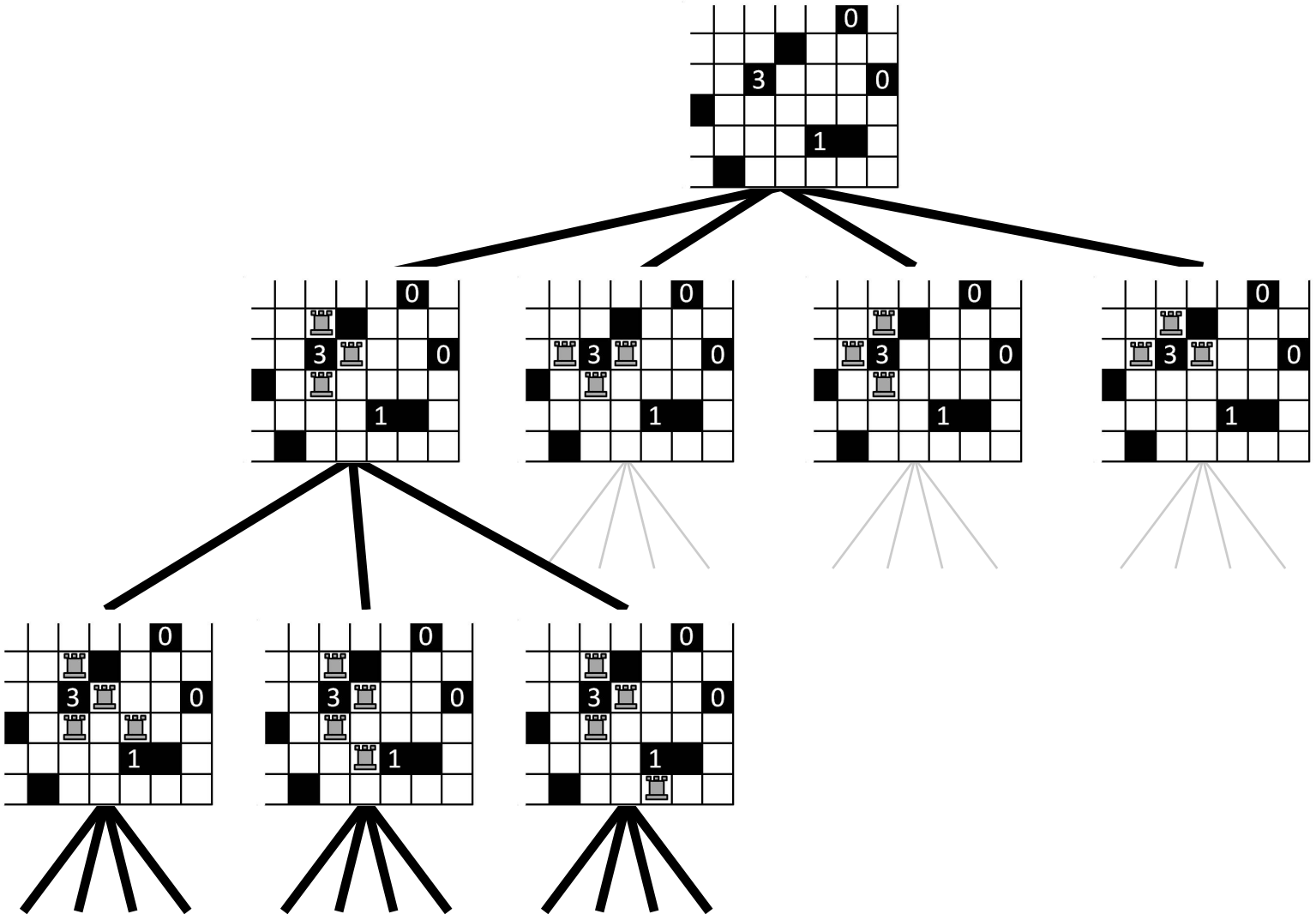
Solution Configuration



Rooks Application

- **Input:** board size and list of number and black squares
- Place rooks around all “4” squares (`placeFour()`)
- Using backtracking:
  - Get next numbered square in list
  - Try all rook combinations around square, via recursive call
    - “3” square => 4 combinations (`placeThree()`)
    - “2” square => 6 combinations (`placeTwo()`)
    - “1” square => 4 combinations (`placeOne()`)
  - If no more numbered squares, compile list of all open squares
  - Using backtracking:
    - Try rook in/out next open square from list
    - Solution reached when no more open squares

Search Tree



Exercise: Rooks with tasks

- Build the serial code with **`make rooks`** command.
- Run the serial code with one of the small test files (rooks15.txt, rooks60.txt).

```
> ./rooks rooks60.txt
```

- Run the serial code with the larger data file and note the execution time:

```
> ./rooks rooks111.txt
```

- Edit the solveboard.c source file. Decide which level of the search should generate tasks for the recursive calls by restoring the pragma lines within one or more functions.
- Rebuild the parallel version of the application.
- Set a number of OpenMP threads and run/time the new executable. How does this compare to the serial code run time?
- If there is time, you can experiment with more or fewer parts of the code to generate tasks for the recursive calls to **`solveBoard()`** or change the number of threads used. How do these runs compare to the serial execution time?

Outline

- OpenMP overview
- Introducing Explicit Tasks in OpenMP
- Working with Tasks
- Tasks and the conceptual core of OpenMP
- Break
- Working with tasks: the divide and conquer pattern
- Advanced tasking features

Task dependencies

!$omp task depend (*type* : *list*)

where *type* is in, out or inout and *list* is a list of variables.

- list may contain subarrays: OpenMP 4.0 includes a syntax for C/C++
- in: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an out or inout clause
- out or inout: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an in, out or inout clause

Task dependencies example

```
#pragma omp task depend (out:a)
  { ... } //writes a
#pragma omp task depend (out:b)
  { ... } //writes b
#pragma omp task depend (in:a,b)
  { ... } //reads a and b
```

- The first two tasks can execute in parallel
- The third task cannot start until the first two are complete

1D Stencil Example

The heat equation:

```
double k = 0.5; // heat transfer coefficient
double dt = 1.; // time step
double dx = 1.; // grid spacing

double heat(double left, double mid, double right)
{
    return mid+(k*dt/dx*dx)*(left-2*mid+right);
}
```

1D Stencil Example

Application of the heat equation to a 1D array

```
void heat_part( int size, double* next,
                double* left,
                double *mid, double *right)
{
    next[0] = heat(left[size-1], mid[0], mid[1]);

    for (int i = 1; i < size-1; ++i)
        next[i] = heat(mid[i-1], mid[i], mid[i+1]);

    next[size-1] = heat(mid[size-2], mid[size-1],
                        right[0]);
}
```

1D Stencil Example

Dividing the work into partitions of the array

```
for (int i = 0; i < np; ++i) {
    heat_part( nx, &next[i*nx],
               &current[idx(i-1, np)*nx],
               &current[i*nx],
               &current[idx(i+1, np)*nx]);
}

//idx does the wrapping here
int idx(int i, int size)
{
    return (i < 0) ? (i + size) % size : i % size;
}
```

1D Stencil Example

Reads and writes need to be done on separate arrays

```
U[0] = malloc(np*nx * sizeof(double));
U[1] = malloc(np*nx * sizeof(double));

double* current = U[0];
double* next    = U[1];
```

1D Stencil Example

Each iteration alternates between arrays

```
for(int t = 0; t < nt; t++) {
    for (int i = 0; i < np; ++i) {
        heat_part( nx, &next[i*nx],
                   &current[idx(i-1, np)*nx],
                   &current[i*nx],
                   &current[idx(i+1, np)*nx]);
    }
    current = U[(t+1) % 2];
    next    = U[ t    % 2];
}
```

1D Stencil Example

Because of the partitioning, one task directive is needed

```
for(int t = 0; t < nt; t++) {
    for (int i = 0; i < np; ++i) {
#pragma omp task untied depend(out: next[i*nx]) \
            depend(in: current[idx(i-1, np)*nx],\
            current[i*nx], current[idx(i+1, np)*nx])
        heat_part( nx, &next[i*nx],
                   &current[idx(i-1, np)*nx],
                   &current[i*nx],
                   &current[idx(i+1, np)*nx]);
    }
    current = U[(t+1) % 2];
    next    = U[ t    % 2];
}
#pragma omp taskwait
```

Outline

- OpenMP overview
  - Brief summary of OpenMP and the server we'll be using for this tutorial.
  - Hands-on: Parallel Loops, matrix multiply.
- Introducing Explicit Tasks in OpenMP
  - Define the task construct
  - Hands-on: The "Racy Car output exercise".
- Working with Tasks
  - Task data environment, default rules plus private, shared and firstprivate
  - Task synchronization: barrier and task wait
  - Hands-on: traversing a linked list
- Tasks and the conceptual core of OpenMP
  - How tasks relate to the conceptual core of OpenMP (implicit parallel regions, implicit tasks, task completion, and other low level details from the task-concepts/execution model section of the spec)
- Break
- Working with tasks: the divide and conquer pattern
  - Divide and conquer design pattern
  - Hands-on: Recursive pi programs
  - Cache oblivious algorithms
  - Hands on: recursive matrix multiply
- Advanced tasking features
  - Task dependencies
  - Hands-on coMD
- Advanced Tasking features
  - Task Groups, task-loops, thread-switching, tied vs. untied tasks, mergable, final

Task definitions

- Task: a specific instance of executable code and its data environment.
- Task region: all the code encountered during the execution of a task.
- When a task construct is encountered by a thread, the generated task may be:
  - Deferred: executed by some thread independently of generating task.
  - Undeferred: completes execution before the generating task continues.
  - Included: Undeferred and executed by the thread that encounters the task construct.
- Tasks once started may suspend, wait, and restart.
  - Tied tasks: if a thread is suspended, the same thread will restart the thread at a later time.
  - Untied tasks: if a task is suspended, any thread in the binding team may restart the thread at a later time.

The task construct (OpenMP 4.5)

#pragma omp task *[clause[[,]clause]...]*
structured-block

Generates an explicit task

where *clause* is one of the following:

if(*[* **task :]***scalar-expression***)**

untied

default(shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

final(scalar-expression)

mergeable

depend(*dependence-type* **:** *list***)**

priority(*priority-value***)**

The evolution of the task construct

OpenMP 3.0

OpenMP 3.1

OpenMP 4.0

OpenMP 4.5

The task construct: the newer/rarely used clauses

untied	The created task, if suspended, can be executed by a different thread
final(scalar-expression)	If the scalar-expression is true, generated tasks are undeferred and execute immediately by the encountering thread.
mergeable	The task is mergable if it is undeferred and included (i.e. uses the parent tasks data environment).
priority(*priority-value*)	Gives a hint to the compiler to schedule tasks with a larger priority value (>0) before tasks with a lower value.

OpenMP 3.0 **OpenMP 3.1** **OpenMP 4.0** **OpenMP 4.5**

Waiting for tasks to complete

```
#pragma omp taskwait
```

OpenMP 3.0

Causes current task region to suspend and wait for completion of all the child tasks created before the taskwait to complete

- A standalone directive
- Defines a task scheduling point

```
#pragma omp taskgroup
        structured-block
```

OpenMP 4.0

A thread encounters the taskgroup construct. It executes the code in the structured block.
That thread suspends and waits at the end of the taskgroup region until all child tasks and any of their descendant tasks are complete.

Task switching

- Consider the following example ... Where the program may generate so many tasks that the internal data structures managing tasks overflow.

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Solution … Task switching; Threads can switch to other tasks at certain points called ***thread scheduling*** points.
- With Task switching, a thread can
  - Execute an already generated task … to “drain the task pool”
  - Execute the encountered task immediately (instead of deferring task execution for later)

Explicit task scheduling

#pragma omp taskyield

OpenMP 3.1

Tells the OpenMP runtime that the current task can be suspended in favor of execution of a different task

- A standalone directive
- Defines an explicit task scheduling point

A function that only one task at a time can execute (mutual exclusion)

```
#include <omp.h>
void something_useful ( void );
void mutual_excl_op( void );
void foo ( omp_lock_t * lock, int n )
{   for (int  i = 0; i < n; i++ )
    #pragma omp task
    {    something_useful();
         while ( !omp_test_lock(lock) ) {
                #pragma omp taskyield
         }
         mutual_excl_op();
         omp_unset_lock(lock);
    }
}
```

Grab a lock if you can, return if you can't

Tell the runtime it can suspend current task and schedule another

Release the lock that protected mutual_excl_op()

Task scheduling Points

- Task switching can only occur at Task Scheduling points.
- Task scheduling points happen …
  - After generation of an explicit task
  - After completion of a task region
  - In a taskyield region
  - In a taskwait region
  - At the end of a taskgropup or barrier
  - In and around regions associated with target constructs (not discussed here).
- At a task scheduling point, *any of* the following can happen for any tasks bound to the current team
  - Begin execution of a tied or untied task
  - Resume any suspended task (tied or untied)

Task Scheduling Details

- An included task is executed immediately after generation of the task
- Scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a barrier region.
  - If this set is empty, any new tied task may be scheduled.
  - Otherwise, a new tied task may be scheduled only if it is a descendent task of every task in the set.
- A dependent task shall not be scheduled until its task dependences are fulfilled.
- When an explicit task is generated by a construct containing an if clause for which the expression evaluated to false, and the previous constraints are already met, the task is executed immediately after generation of the task.

Task Execution around task scheduling points

- Think of a task as a set of “task regions” between task scheduling points
- Each “task region” executes uninterrupted from start to end in the order they are encountered.
- A correct program must behave correctly and consistently with all conceivable scheduling sequences that are compatible with the rules above.
  - If multiple “task regions” between scheduling points modify values in threadprivate storage, a data race is produced and the state of threadprivate storage is not defined.
  - Lock acquire and release in different task regions may break program-order lock protocols and deadlock.

The taskloop construct (OpenMP 4.5)

#pragma omp taskloop *[clause[[,]clause]...]*
structured-block

where *clause* is one of the following:

if([taskloop :]scalar-expr)

shared(list)

private(list)

firstprivate(list)

lastprivate(list)

default(shared | none)

grainsize(grain-size)

num_tasks(num-tasks)

collapse(n)

final(scalar-expr)

priority(priority-value)

untied

mergeable

nogroup

- The structured block contains loops in the standard form
- Loop iterations are turned into tasks that execute within a taskgroup (unless the nogroup clause is present)
- Grainsize specifies the number of iterations per task
- Num_tasks stipulates the number of tasks to create (unless there are too few loop iterations)

Conclusion

- OpenMP was created to handle loop-level programs and basic multi-threading programs with the Single Program Multiple Data (SPMD) pattern.
- With OpenMP 3.0, the task construct was added to support irregular programs:
  - While loops or loops whose iteration limits are not known at compiler time.
  - Recursive algorithms
  - divide and conquer problems.
- The task construct has expanded over the years with new features to support irregular problems with tasks in each new release of OpenMP

Programming Irregular Applications with OpenMP*

Thank You for attending

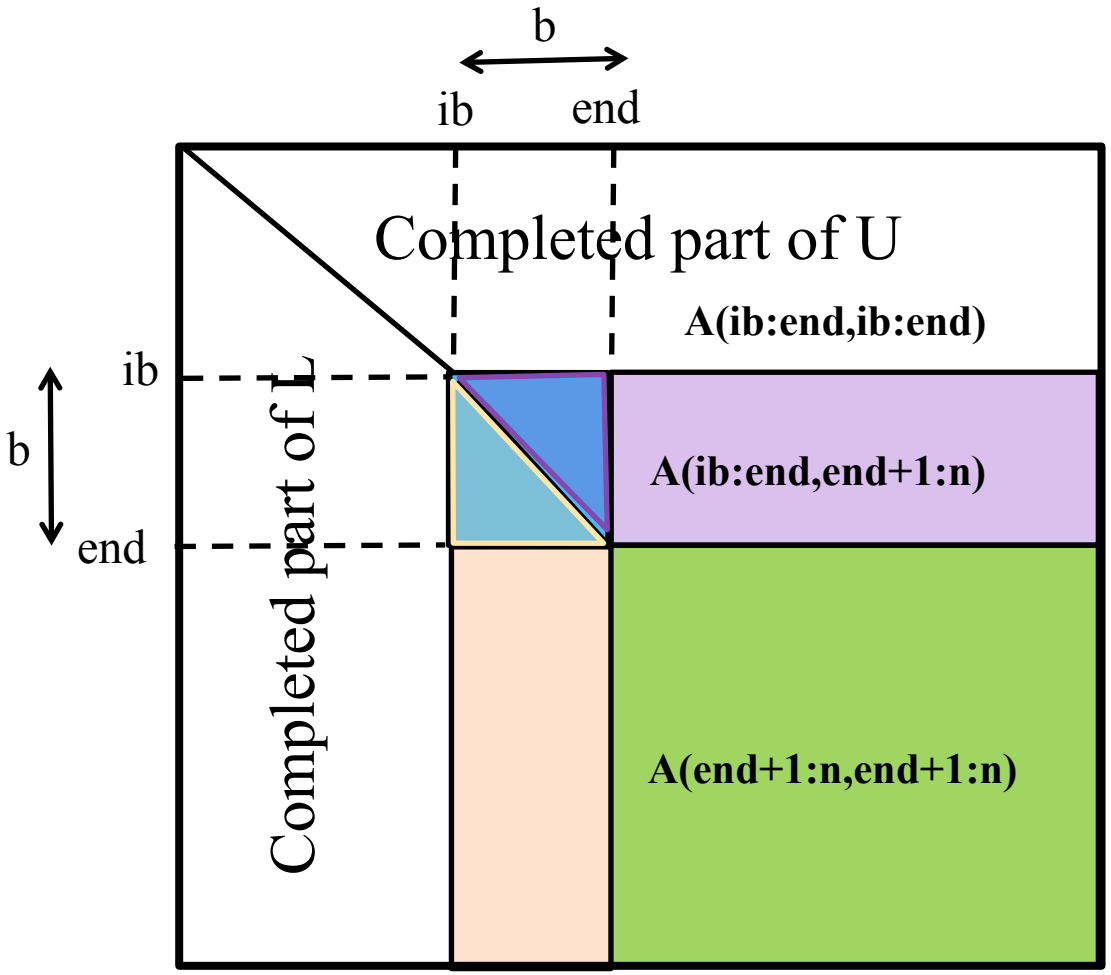
Please fill out your tutorial evaluations

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

78

Backup slides to keep around … important raw material

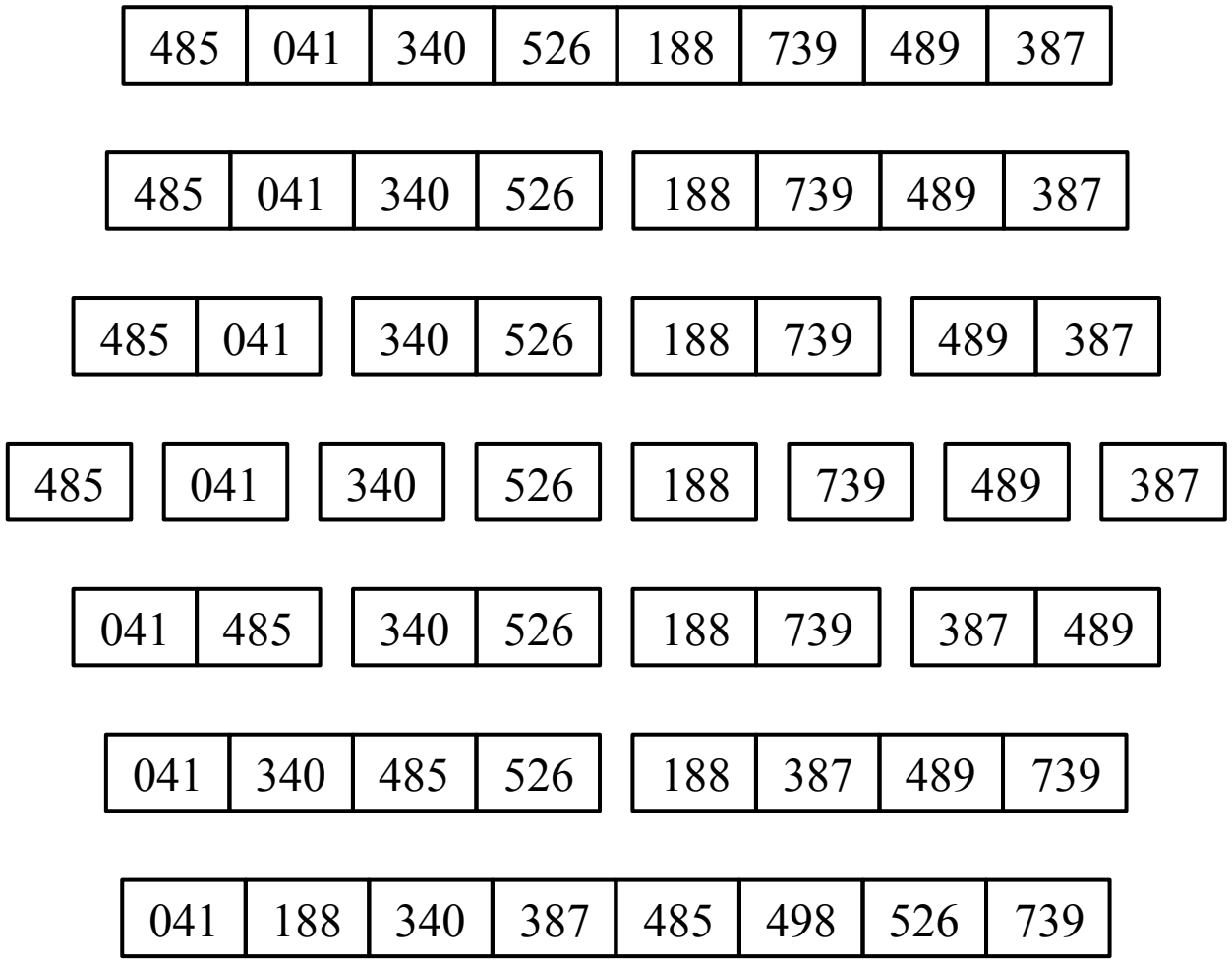
BLAS 3 based Gaussian Elimination
A = LU



Mergesort

- Prototypical Divide and Conquer via recursive algorithm
- Given: Unsorted array of elements (each with assoc. key)
- Goal: Sorted array of elements
- Divide Phase:
  - Split the unsorted array into two unsorted sub-arrays
  - Continue splitting each sub-array until a single element is reached
    - At this point, each sub-array contains a sorted list
- Conquer phase:
  - Take two adjacent sorted sub-arrays and uses merge to create a larger sorted sub-array
    - The recursive solution provides a means to automatically retrace the divide computations in the reverse order

Mergesort Illustration



Parallel Mergesort Questions

- Very regular pattern of array division
- In parallel:
  - Where do merge results get stored?
    - Serial Merge deposits elements from sorted lists A and B into third array C (size equal to A+B)
    - Alternating scheme via parameters? (ala recursive Towers of Hanoi)
  - If tasks used, how do they coordinate to not overwrite other task results?
    - Allocate new storage for each merge result?
    - In-place merge algorithm?

Program: OpenMP tasks (divide and conquer pattern)

```
 include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK  10000000
double pi_comp(int Nstart,int Nfinish,double step)
{   int i,iblk;
   double x, sum = 0.0,sum1, sum2;
   if (Nfinish-Nstart < MIN_BLK){
     for (i=Nstart;i< Nfinish; i++){
       x = (i+0.5)*step;
       sum = sum + 4.0/(1.0+x*x);
     }
   }
   else{
     iblk = Nfinish-Nstart;
     #pragma omp task shared(sum1)
         sum1 = pi_comp(Nstart,        Nfinish-iblk/2,step);
     #pragma omp task shared(sum2)
          sum2 = pi_comp(Nfinish-iblk/2, Nfinish,       step);
     #pragma omp taskwait
       sum = sum1 + sum2;
   }return sum;
}
```

```
int main ()
{
  int i;
  double step, pi, sum;
   step = 1.0/(double) num_steps;
   #pragma omp parallel
   {
      #pragma omp single
         sum =
            pi_comp(0,num_steps,step);
    }
     pi = step * sum;
}
```

Results*: pi with tasks

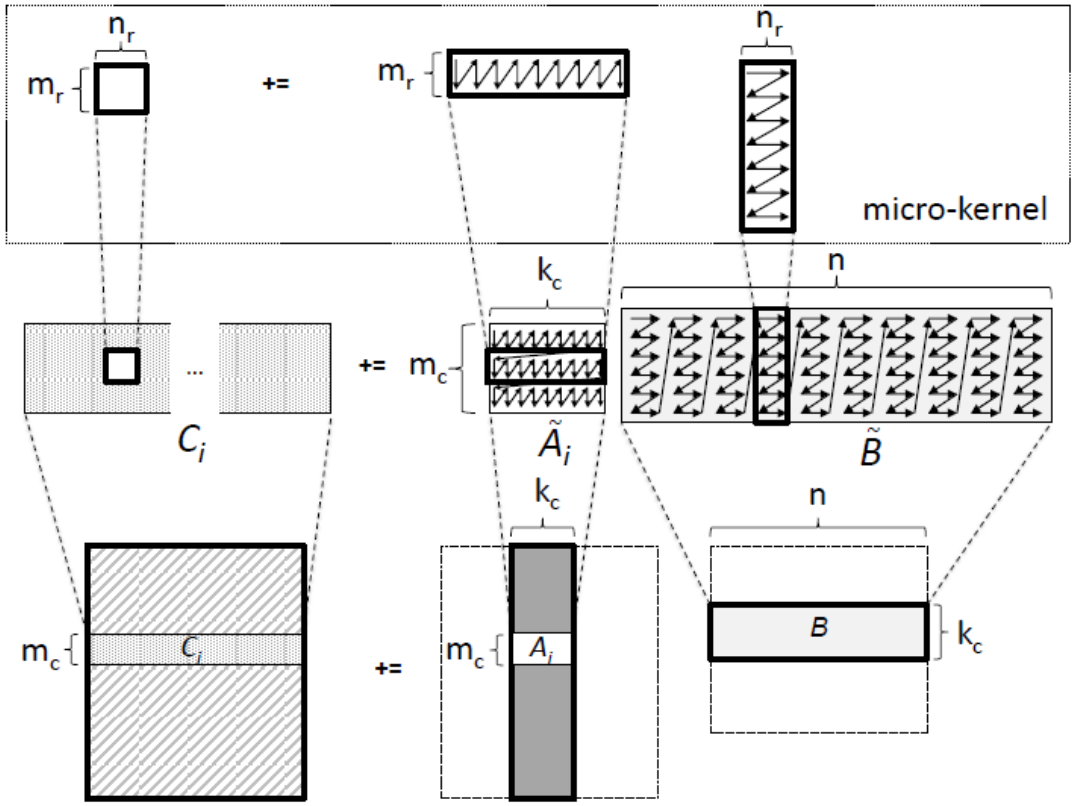
threads	1st SPMD	SPMD critical	PI Loop	Pi tasks
1	1.86	1.87	1.91	1.87
2	1.03	1.00	1.02	1.00
3	1.08	0.68	0.80	0.76
4	0.97	0.53	0.68	0.52

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Cache Oblivious algorithms

- Linear algebra libraries are expected to deliver peak performance.
- Library developers go to great lengths to divide their problem into blocks that fit into the caches on a particular system
- This works great, but (1) it requires skills few programmers have, and (2) code may need to change in response to small chances in memory hierarchy.

Blocked Matrix Multiply with BLIS

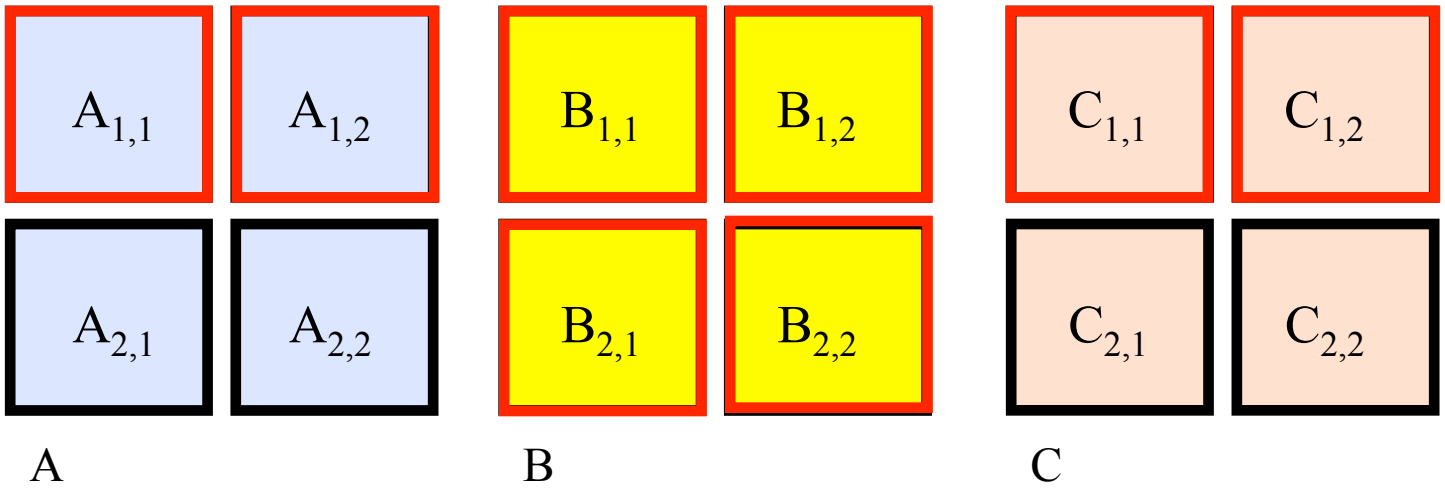


- An Alternative approach: Cache Oblivious algorithms. Use divide and conquer to naturally decompose a problem into small subproblems that fit a memory hierarchy. No explicit cache blocking required!!

Source: Field G. Van Zee and Robert van de Geijn, BLIS: A framework for Rapidly Instantiating BLAS functionality, submitted to ACM TOMS, 2013.

Cache Oblivious matrix multiplication using a recursive algorithm

- Quarter each input matrix and output matrix
- Treat each submatrix as a single element and multiply
- 8 submatrix multiplications, 4 additions



$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

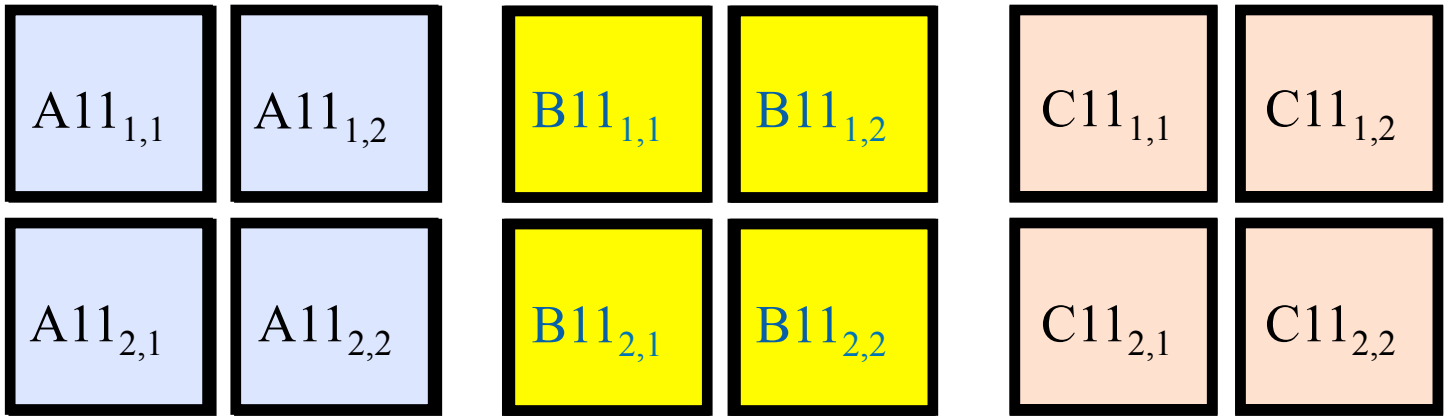
$$C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$

$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

$$C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

Recursive matrix multiplication
How to multiply submatrices?

- Use the same routine that is computing the full matrix multiplication
  - Quarter each input submatrix and output submatrix
  - Treat each sub-submatrix as a single element and multiply



$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

$$C11_{1,1} = A11_{1,1} \cdot B11_{1,1} + A11_{1,2} \cdot B11_{2,1} + A12_{1,1} \cdot B21_{1,1} + A12_{1,2} \cdot B21_{2,1}$$

Recursive matrix multiplication
Recursively multiply submatrices

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \qquad C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$

$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \qquad C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

- Need range of indices to define each submatrix to be used

```
void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl,
                double **A, double **B, double **C)
{// Dimensions: A[mf..ml][pf..pl]  B[pf..pl][nf..nl]  C[mf..ml][nf..nl]

// C11 += A11*B11
 matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A,B,C);
// C11 += A12*B21
 matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A,B,C);
   . . .
}
```

- Also need stopping criteria for recursion

Exercise: Parallel recursive matrix multiply

- Source code implementing this algorithm is provided in the file matmul_recur.c
- Parallelize this program using OpenMP tasks

```
#pragma omp parallel
#pragma omp task
#pragma omp taskwait
#pragma omp master
#pragma omp single
double omp_get_wtime()
int omp_get_thread_num();
int omp_get_num_threads();
```

Recursive matrix multiplication

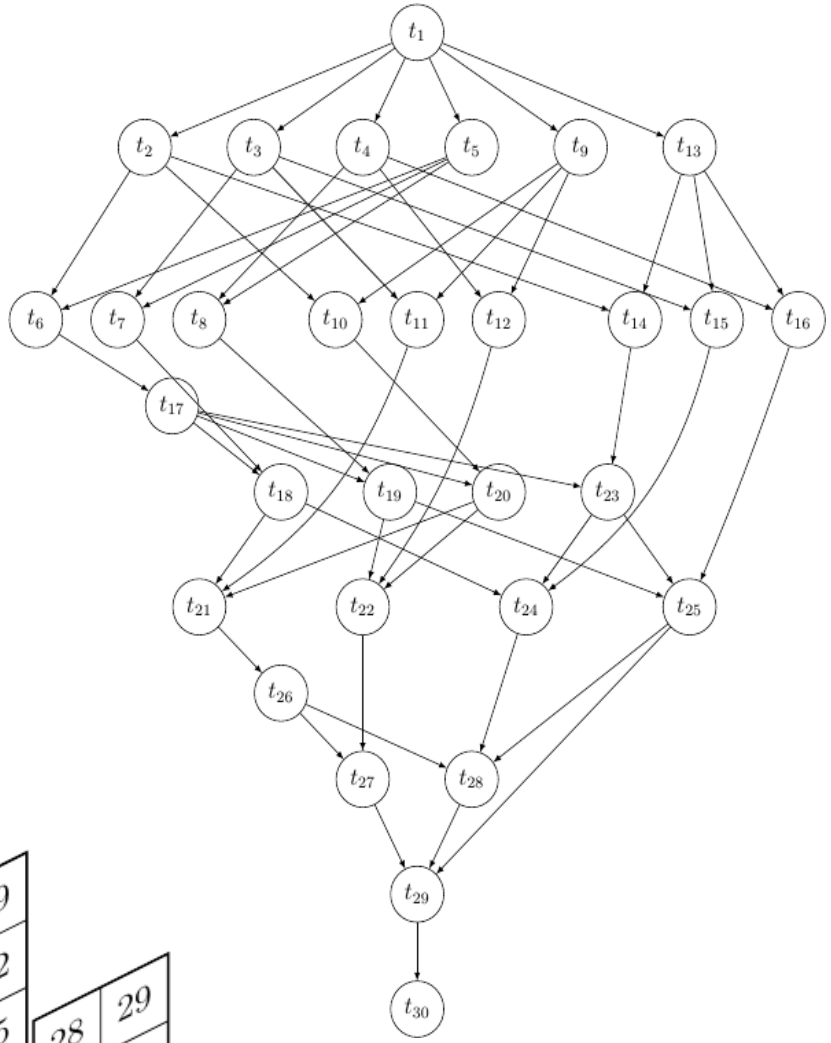
- Could be executed in parallel as 4 tasks
  - Each task executes the two calls for the same output submatrix of C
- However, the same number of multiplication operations needed

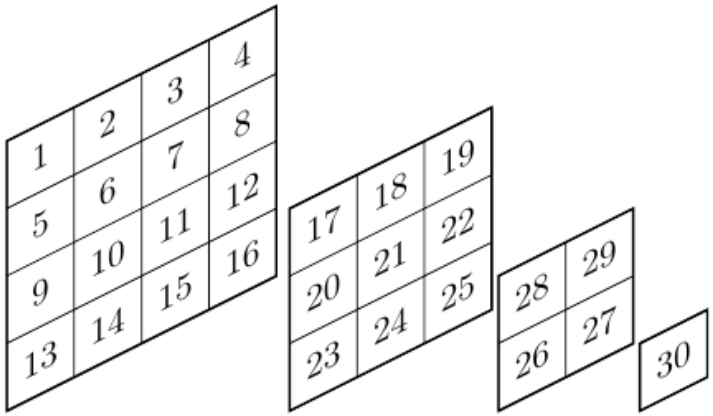
```
#define THRESHOLD 32768   // product size below which simple matmult code is called

void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl,
                double **A, double **B, double **C)

// Dimensions: A[mf..ml][pf..pl]    B[pf..pl][nf..nl]   C[mf..ml][nf..nl]

{
   if ((ml-mf)*(nl-nf)*(pl-pf) < THRESHOLD)
      matmult (mf, ml, nf, nl, pf, pl, A, B, C);
   else
   {
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C11 += A11*B11
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C11 += A12*B21
}
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C12 += A11*B12
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C12 += A12*B22
}
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
     matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C21 += A21*B11
     matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C21 += A22*B21
}
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
     matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C22 += A21*B12
     matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C22 += A22*B22
}
#pragma omp taskwait

   }
}
```

Extra: LU Example

LU Decomposition

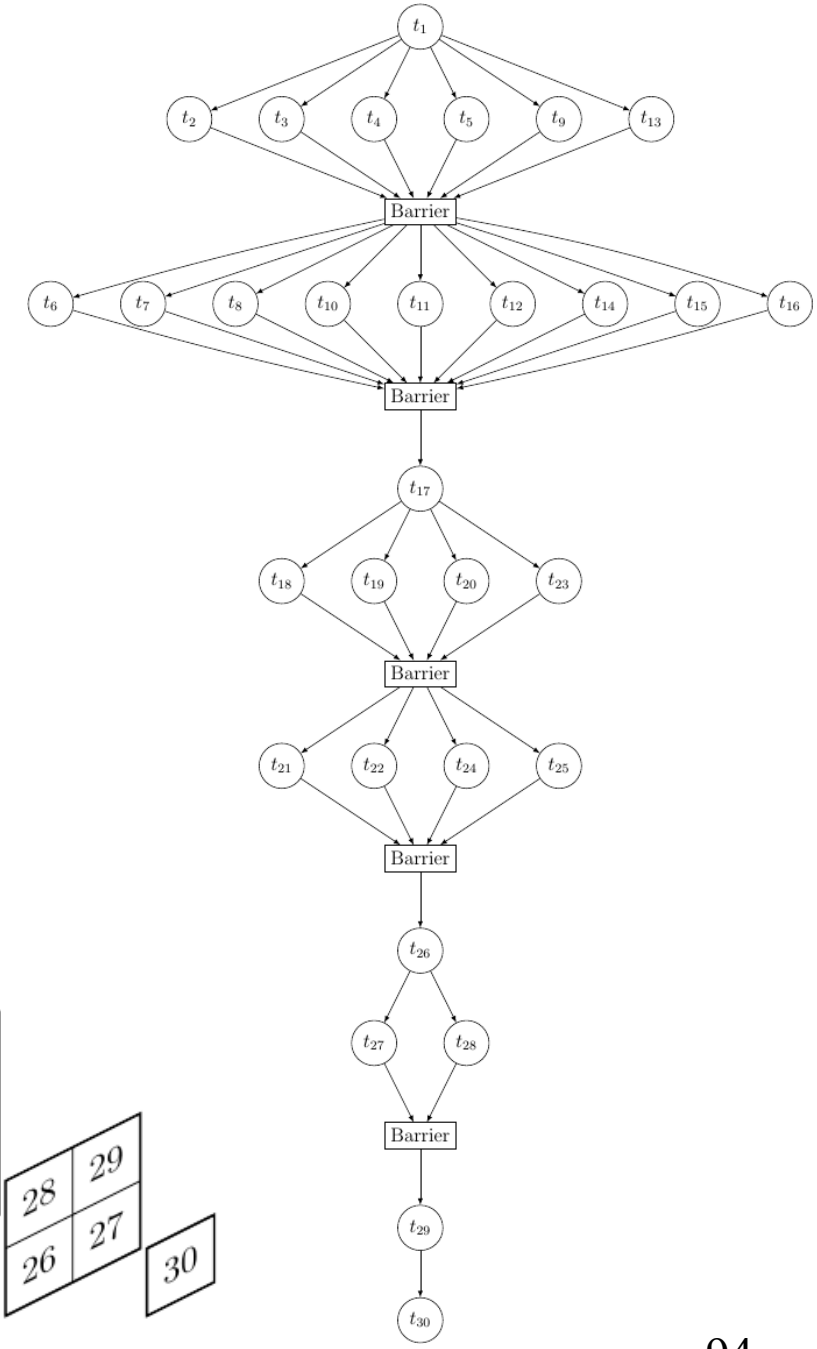
- The matrix is divided into NxN blocks, and a task operates on one block.
- Each iteration the working matrix gets one block smaller in each dimension, resulting in a task graph resembling the one to the right.

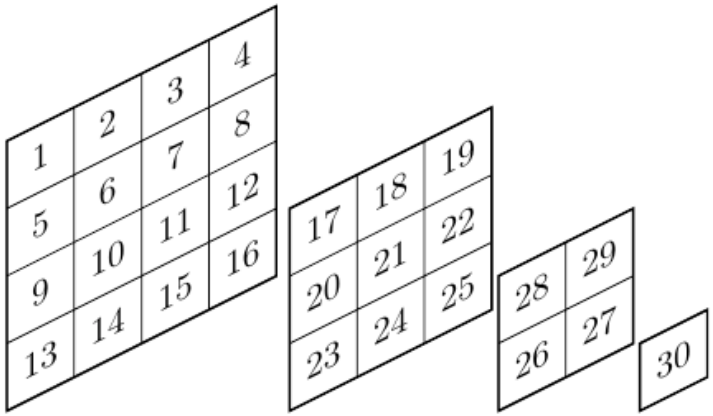




LU Decomposition

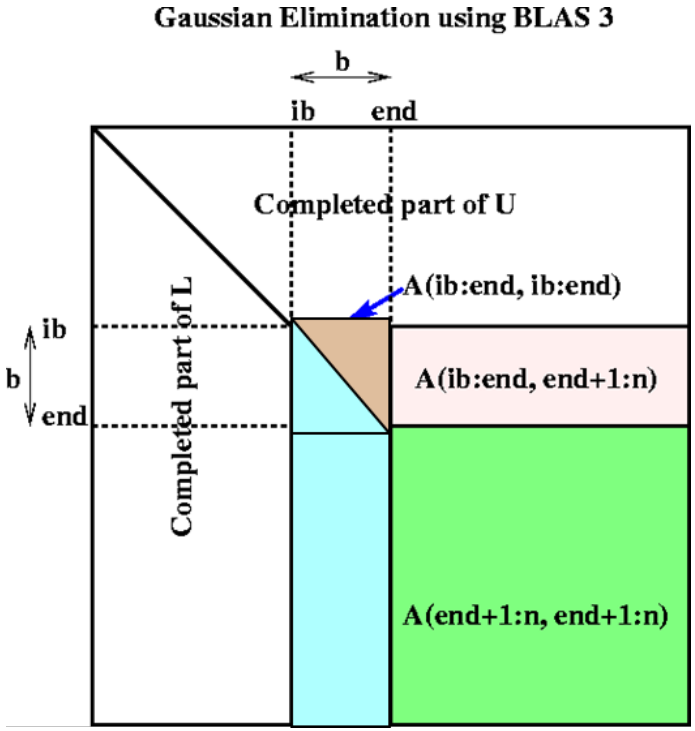
For Comparison, the tasking version without dependencies resembles a fork join programming model, similar to a worksharing version.

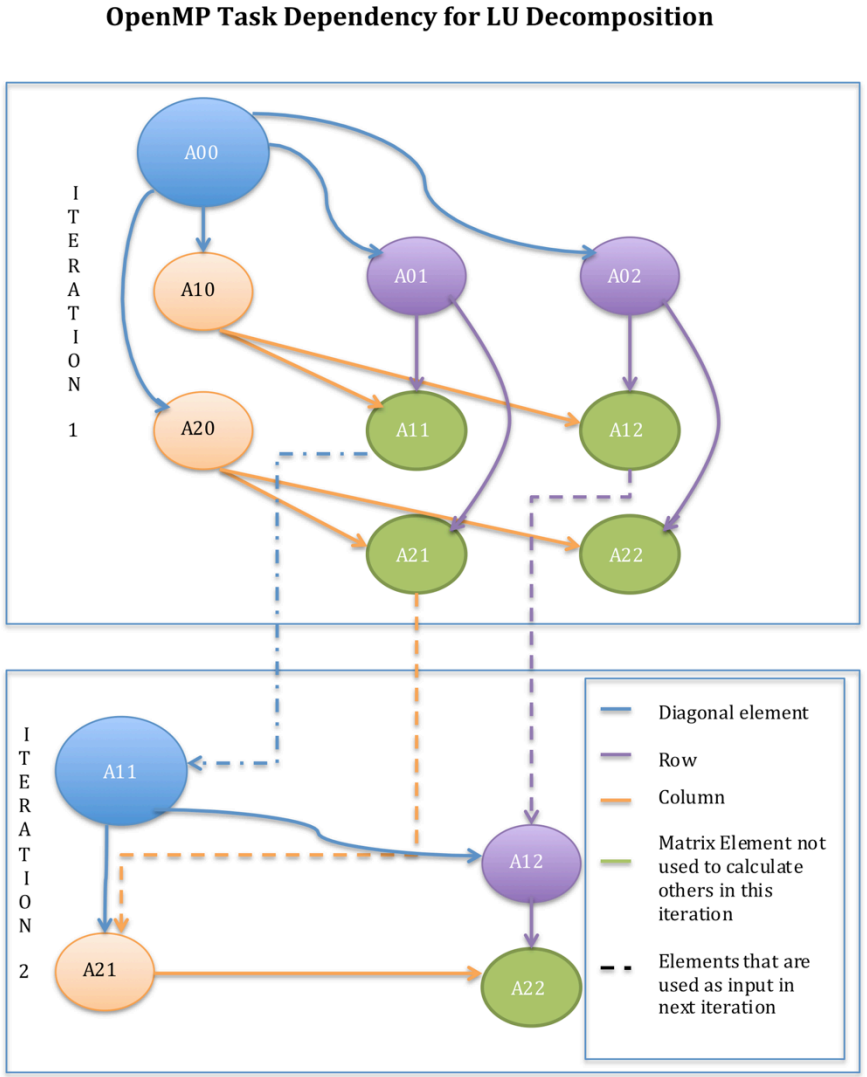




LU Decomposition

There are 4 different operations (diag, row, col, and inner), and the the dependencies between these operations are shown in the graph to the right.





Source: CS164 lecture Fall 2014, Kurt Keutzer of UC Berkeley

LU Decomposition

```
void diag_op(const block &B) {
    for(int i = 0; i < B.width; i++)
        for(int j = i+1; j < B.width; j++) {
            B.start[j*B.stride+i] /= B.start[i*B.stride+i];
            for(int k = i+1; k < B.width; k++)
                B.start[j*B.stride+k] -= B.start[j*B.stride+i] * B.start[i*B.stride+k];
}}
void col_op(const block &B1, const block &B2) {
    for(int i=0; i < B2.width; i++)
        for(int j=0; j < B1.height; j++) {
            B1.start[j*B1.stride+i] /= B2.start[i*B2.stride+i];
            for(int k = i+1; k < B2.width; k++)
                B1.start[j*B1.stride+k] += -B1.start[j*B1.stride+i] * B2.start[i*B2.stride+k];
}}
void row_op(const block &B1, const block &B2) {
    for(int i=0; i < B2.width; i++)
        for(int j=i+1; j < B2.width; j++)
            for(int k=0; k < B1.width; k++)
                B1.start[j*B1.stride+k] += -B2.start[j*B2.stride+i] * B1.start[i*B1.stride+k];
}
void inner_op(const block &B1, const block &B2, const block &B3) {
    for(int i=0; i < B3.width; i++)
        for(int j=0; j < B1.height; j++)
            for(int k=0; k < B2.width; k++)
                B1.start[j*B1.stride+k] += -B3.start[j*B3.stride+i] * B2.start[i*B2.stride+k];
}
```

LU Decomposition

- Each operation is put into a function, and the core logic (without tasks) is shown below

```
void LU(int num_blocks) {
    for(int i=0; i<num_blocks; i++) {
        diag_op( block_list[i][i] );
        for(int j=i+1; j<num_blocks; j++){
            row_op( block_list[i][j], block_list[i][i] );
            col_op( block_list[j][i], block_list[i][i] );
        }
        for(int j=i+1; j<num_blocks; j++) {
            for(int k=i+1; k<num_blocks; k++) {
                inner_op( block_list[j][k], block_list[i][k],
                                            block_list[j][i] );
            }
        }
    }
}
```

LU Decomposition

- Now to add the directives that create tasks and establish dependencies.

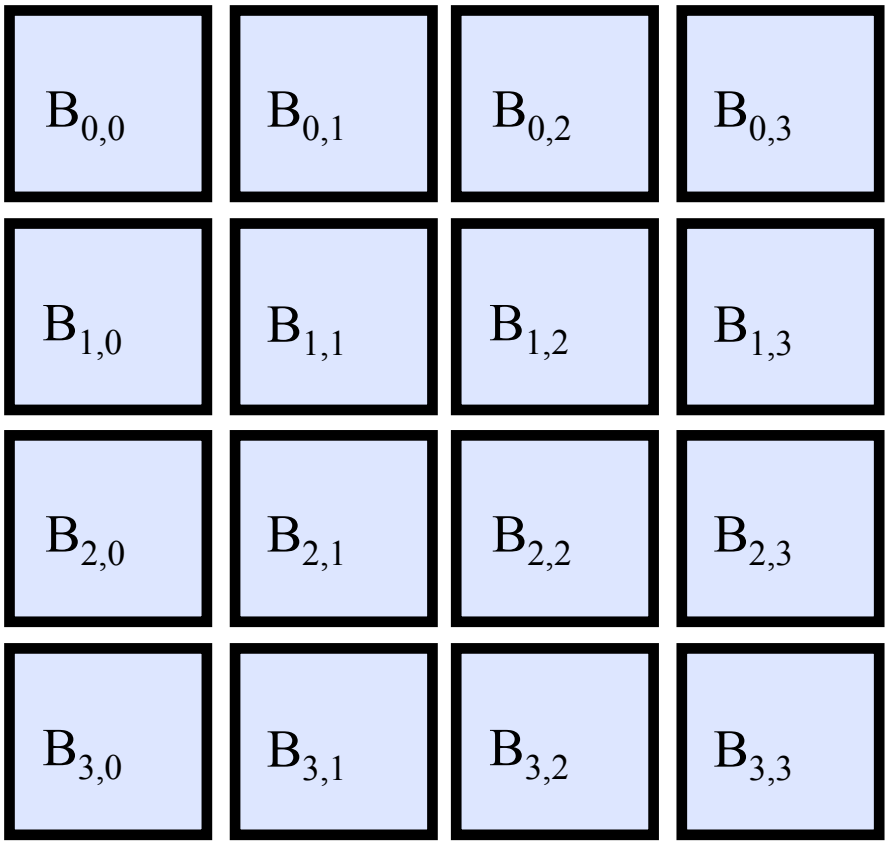
```
for(int i=0; i<num_blocks; i++) {
#pragma omp task depend(inout: block_list[i][i])
    diag_op( block_list[i][i] );
    for(int j=i+1; j<num_blocks; j++) {
#pragma omp task depend(in   : block_list[i][i]) \
                 depend(inout: block_list[i][j])
        row_op( block_list[i][j], block_list[i][i] );
#pragma omp task depend(in   : block_list[i][i]) \
                 depend(inout: block_list[j][i])
        col_op( block_list[j][i], block_list[i][i] );
    }
    for(int j=i+1; j<num_blocks; j++) {
        for(int k=i+1; k<num_blocks; k++) {
#pragma omp task depend(   in: block_list[i][k], block_list[j][i]) \
                 depend(inout: block_list[j][k])
            inner_op( block_list[j][k], block_list[i][k],
                                        block_list[j][i] );
        }
    }
}
#pragma omp taskwait
```

LU Decomposition
Recursive Cache Oblivious Algorithm

- This approach forces the amount of work per task and the blocking size for the targeted cache to be the same.
- This becomes an issue on larger matrix sizes, and on architectures with smaller caches. Either the number of tasks gets very large and increases overhead, or the tasks don’t take advantage of Cache.
- A cache oblivious algorithm provides a way to control the number of tasks while still optimizing for one or more levels of cache within each task.

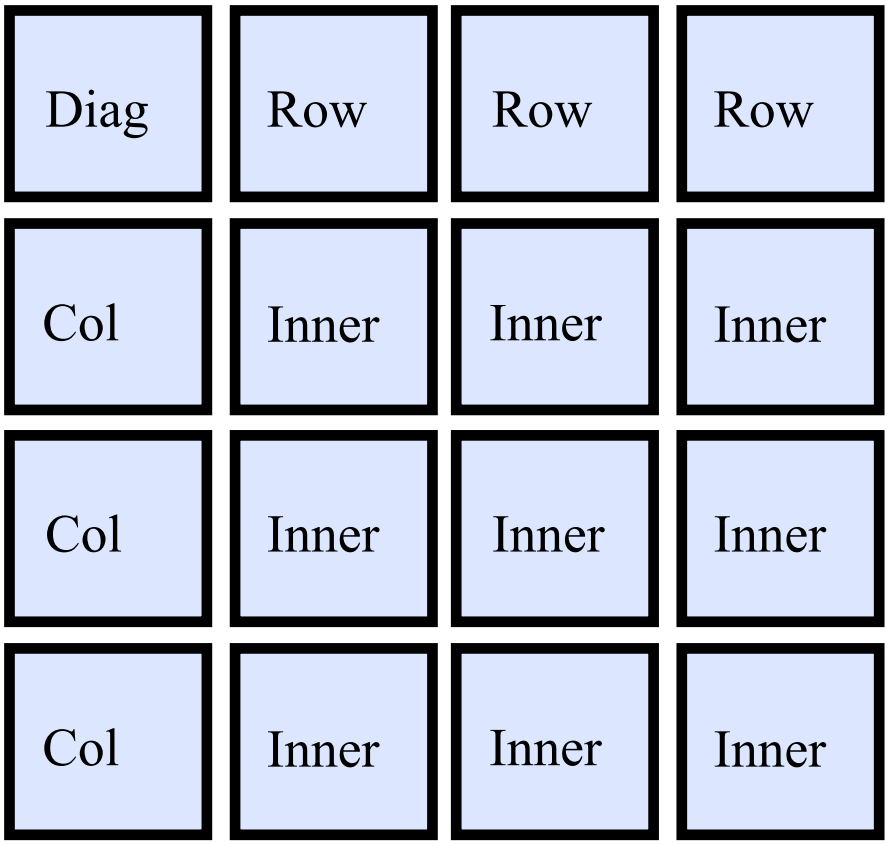
LU Decomposition
Recursive Cache Oblivious Algorithm

- To start with an example, take a matrix divided into 4x4 blocks



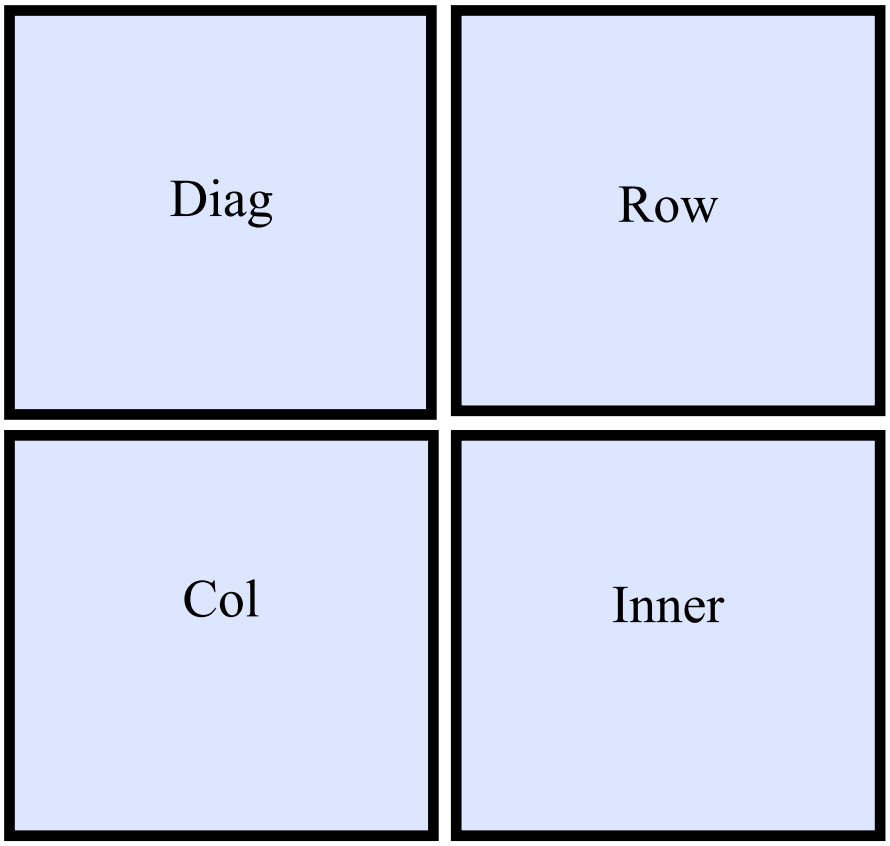
LU Decomposition
Recursive Cache Oblivious Algorithm

- The first version would go through the first iteration and create tasks for these blocks, then move on to the next iteration.



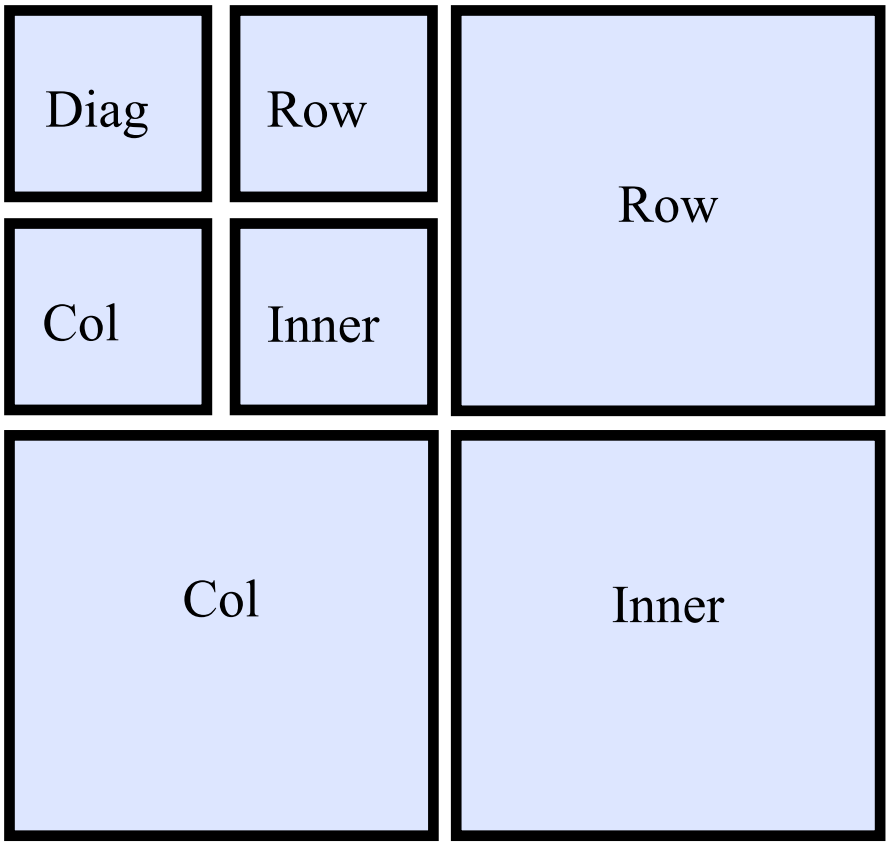
LU Decomposition
Recursive Cache Oblivious Algorithm

- The recursive version starts by calling Diag to divide the whole matrix into quadrants.
- Each of these quadrants is processed, and then Diag is called again on the output of Inner, which handles the second half of iterations.



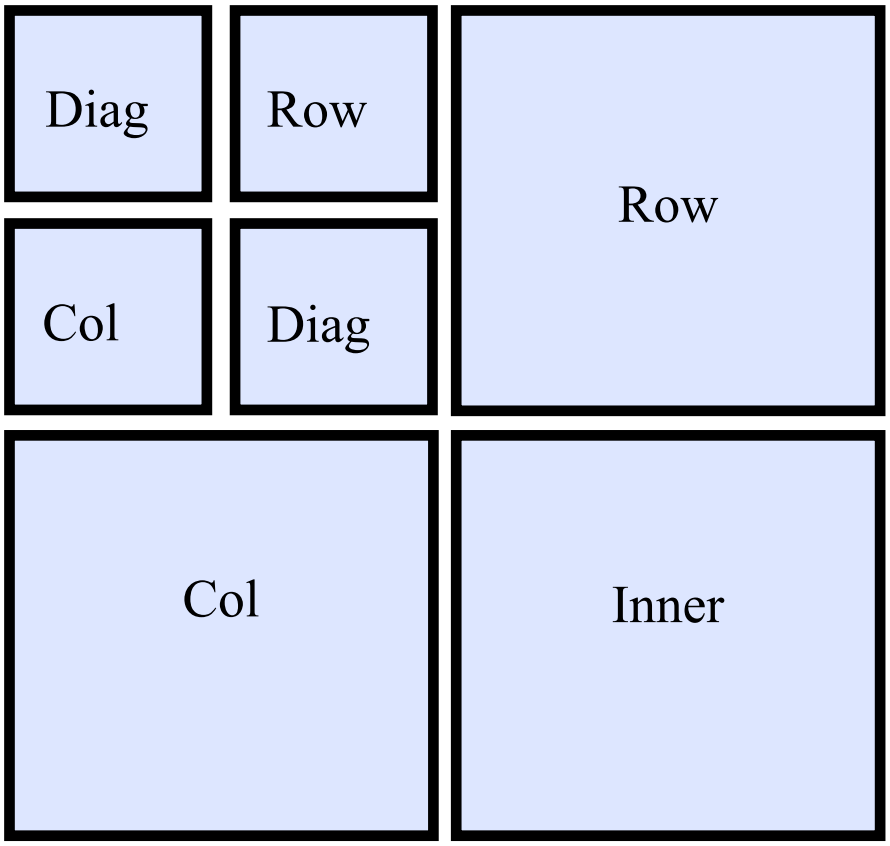
LU Decomposition
Recursive Cache Oblivious Algorithm

- Within diag, the blocks are processed as shown.



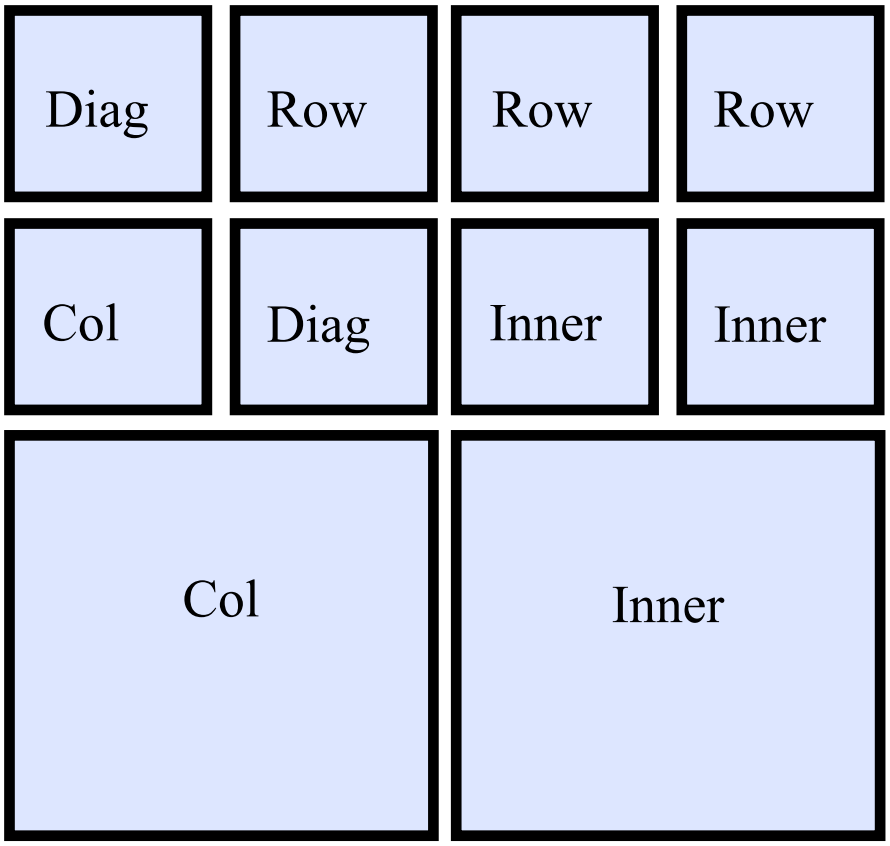
LU Decomposition
Recursive Cache Oblivious Algorithm

- Then, like mentioned earlier, diag is called again to handle the next iteration.



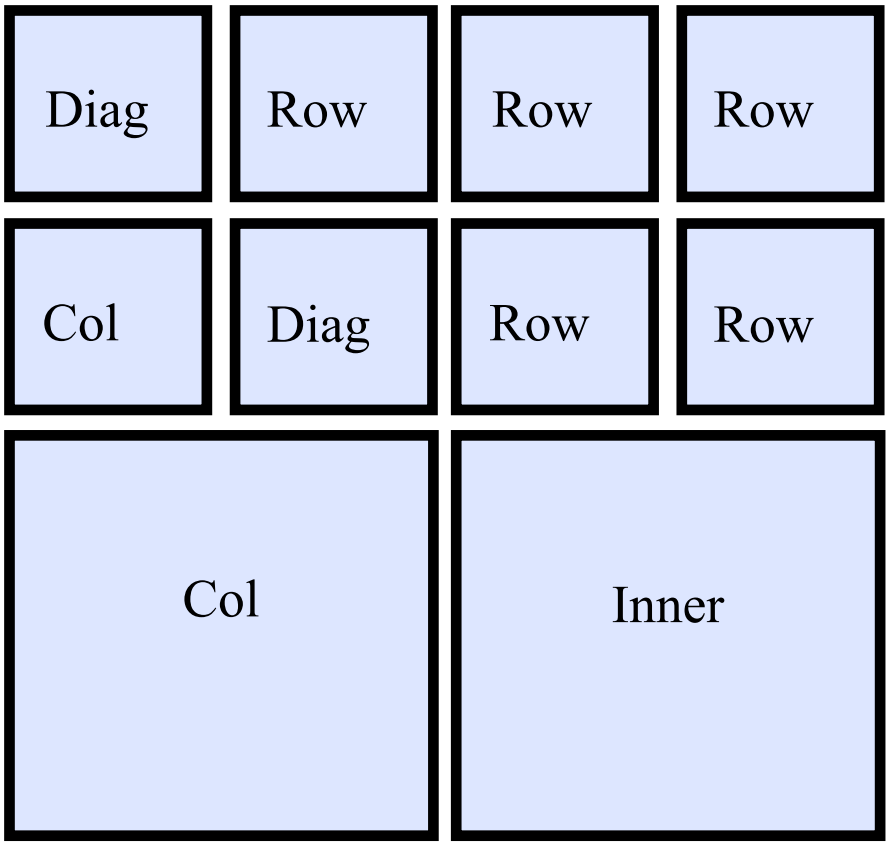
LU Decomposition
Recursive Cache Oblivious Algorithm

- Similarly, row and inner are called for the first iteration.



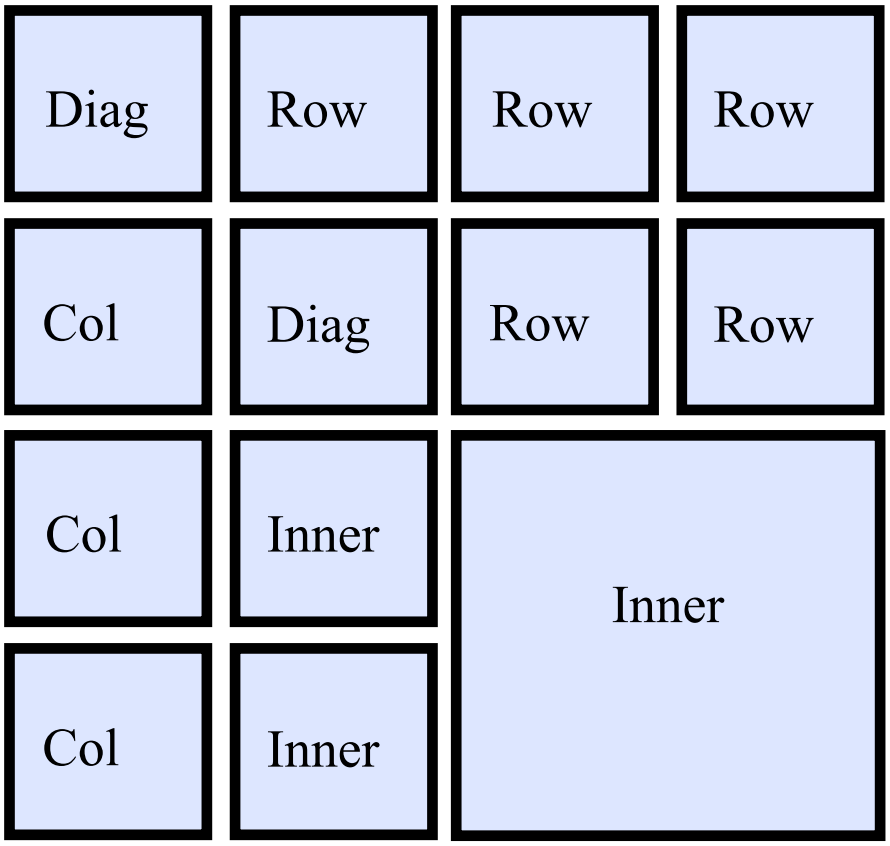
LU Decomposition
Recursive Cache Oblivious Algorithm

- Then row is called again for the second iteration.



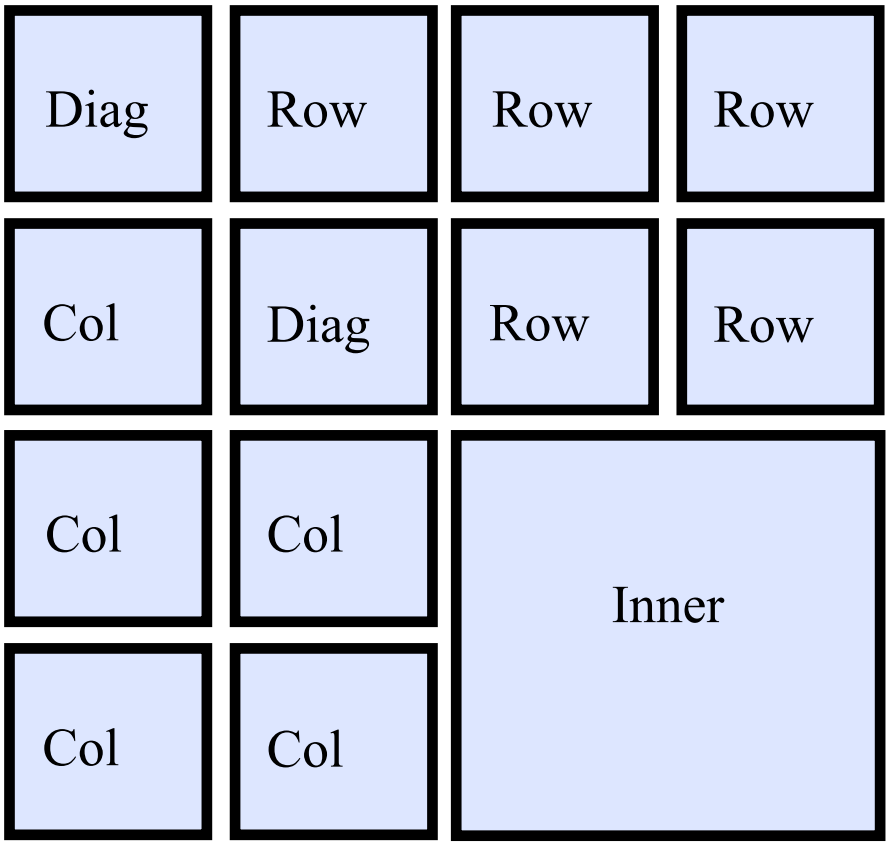
LU Decomposition
Recursive Cache Oblivious Algorithm

- Once the row quadrant is finished, the col quadrant is similarly processed.



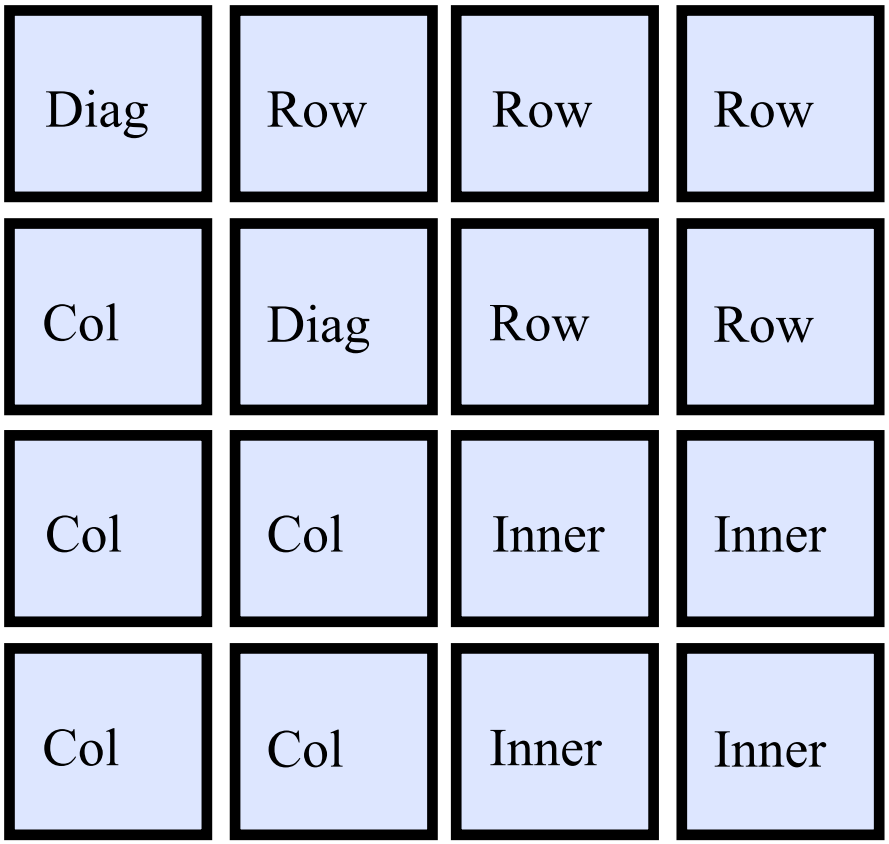
LU Decomposition
Recursive Cache Oblivious Algorithm

- And again, col is processed for the second iteration.



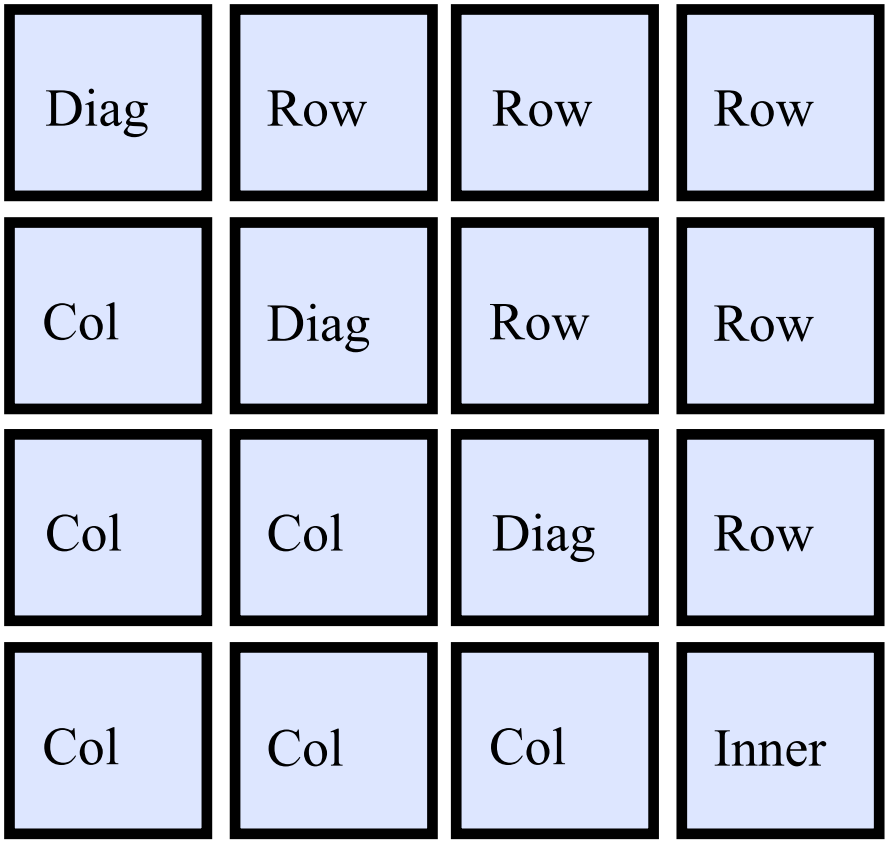
LU Decomposition
Recursive Cache Oblivious Algorithm

- Each of the blocks in inner is processed using row and column 0 for the first iteration. Then processed again using row and column 1 for the second iteration.



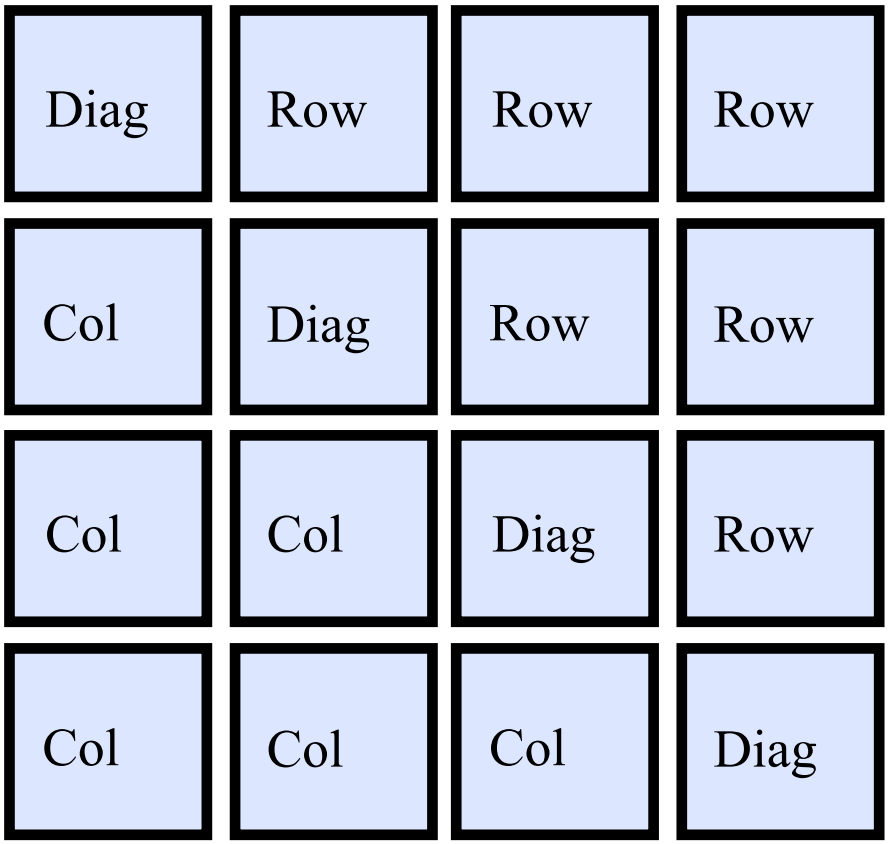
LU Decomposition
Recursive Cache Oblivious Algorithm

- Now the Inner quadrant is done and ready to be passed to diag, and perform what would be the third iteration.



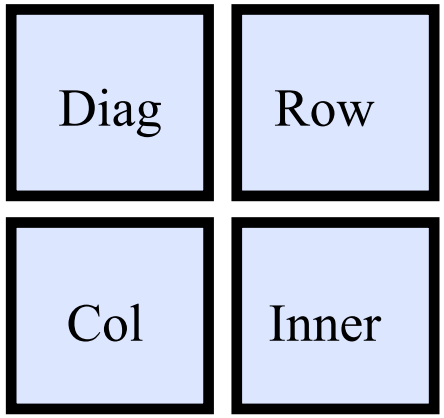
LU Decomposition
Recursive Cache Oblivious Algorithm

- And the final step is diag on the last block, for the fourth iteration.



LU Decomposition
Recursive Cache Oblivious Algorithm

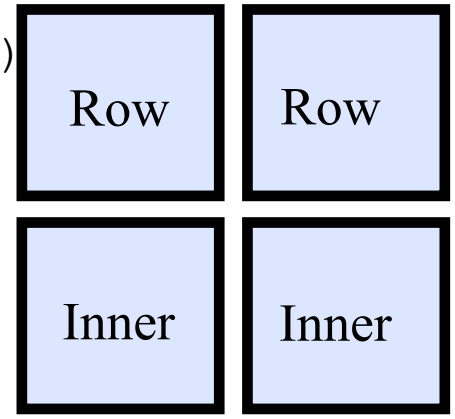
And now code for the serial version




```
void rec_diag(int iter, int mat_size) {
    int half = mat_size/2;
    if(mat_size == 1) {
        diag_op(block_list[iter][iter]);
    } else {
        rec_diag (iter, half);
        rec_row  (iter, iter+half, half);
        rec_col  (iter, iter+half, half);
        rec_inner(iter, iter+half, iter+half, half);
        rec_diag (iter+half, half);
    }
}
```

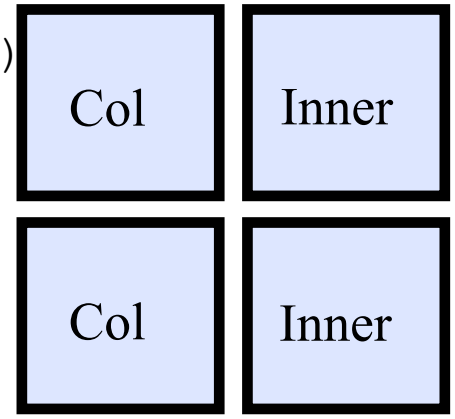
LU Decomposition
Recursive Cache Oblivious Algorithm

```
void rec_row(int iter, int i, int mat_size)
    int half= mat_size/2;
    if(mat_size == 1) {
        row_op(block_list[iter][i],
               block_list[iter][iter]);
    } else {
        //left side
        rec_row  ( iter, i, half);
        rec_inner( iter, iter+half, i, half);
        rec_row  ( iter+half, i, half);
        //right side
        rec_row  ( iter, i+half, half);
        rec_inner( iter, iter+half, i+half, half);
        rec_row  ( iter+half, i+half, half);
    }
}
```

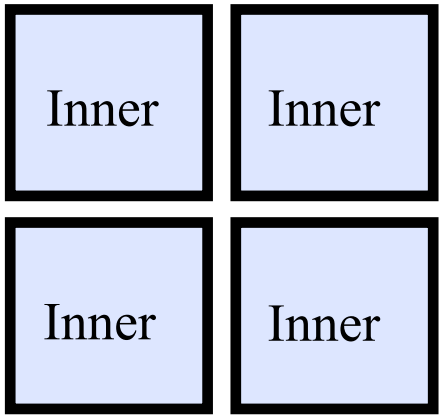


LU Decomposition
Recursive Cache Oblivious Algorithm

```
void rec_col(int iter, int i, int mat_size)
    int half= mat_size/2;
    if(mat_size == 1) {
        col_op(block_list[i][iter],
               block_list[iter][iter]);
    } else {
        //top half
        rec_col   ( iter, i, half);
        rec_inner( iter, i, iter+half, half);
        rec_col   ( iter+half, i, half);
        //bottom half
        rec_col   ( iter, i+half, half);
        rec_inner( iter, i+half, iter+half, half);
        rec_col   ( iter+half, i+half, half);
    }
}
```



LU Decomposition
Recursive Cache Oblivious Algorithm




```
void rec_inner(int iter,
               int i, int j, int mat_size) {
    int half = mat_size/2;
    int offset_i = i+half;
    int offset_j = j+half;
if(mat_size == 1){
        inner_op(block_list[i][j],
                 block_list[iter][j],
                 block_list[i][iter]);
    } else {
        rec_inner( iter,        i,        j, half);
        rec_inner( iter,        i, offset_j, half);
        rec_inner( iter, offset_i,        j, half);
        rec_inner( iter, offset_i, offset_j, half);

        rec_inner( iter+half,        i,        j, half);
        rec_inner( iter+half,        i, offset_j, half);
        rec_inner( iter+half, offset_i,        j, half);
        rec_inner( iter+half, offset_i, offset_j, half);
    }
}
```

LU Decomposition
Recursive Cache Oblivious Algorithm

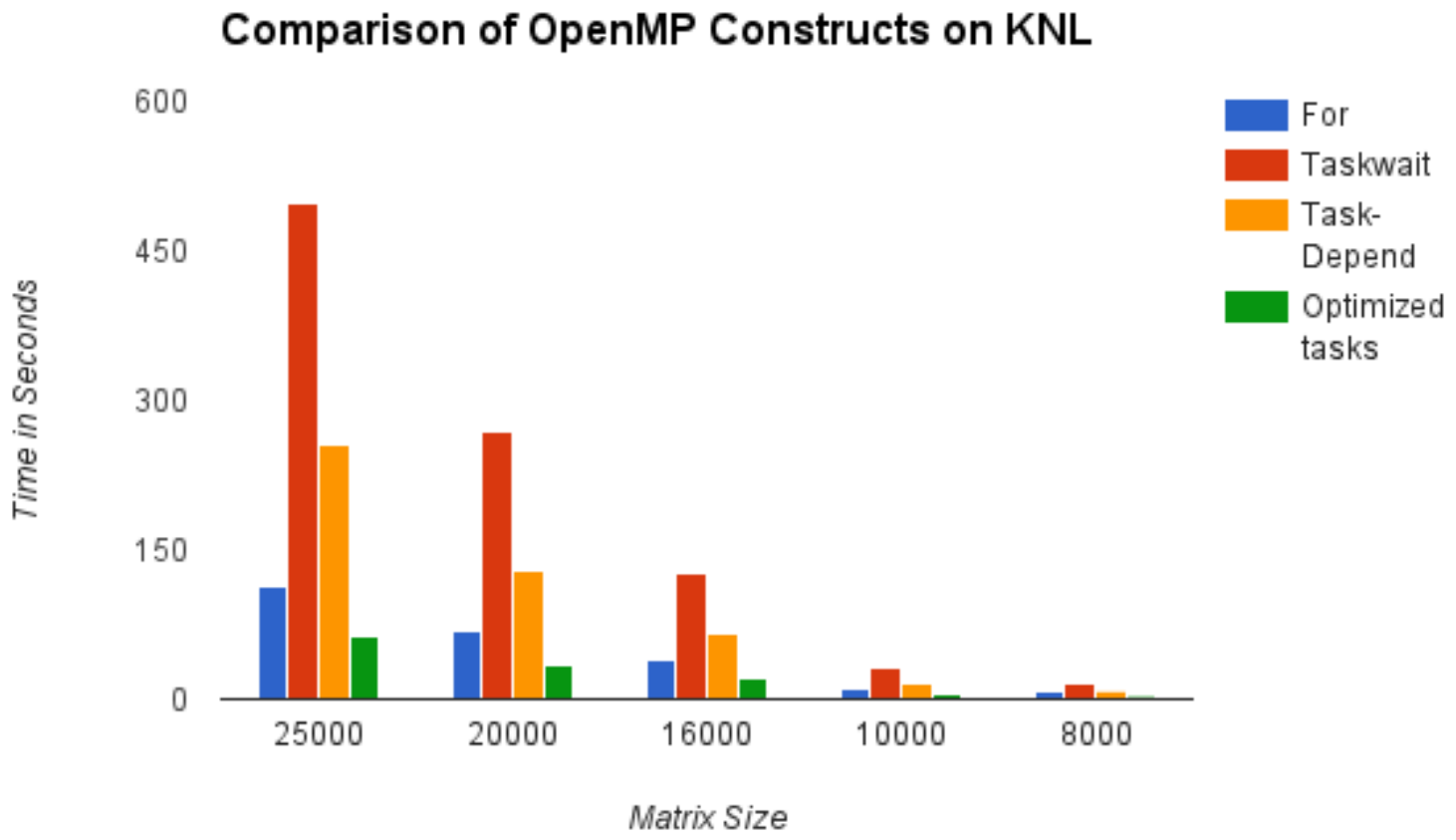
- Adding only tasking directives with depend the clause to this serial version would result in the program creating the same tasks as the previous version.
- In order to get the locality benefits of the cache oblivious algorithm, a cutoff is needed.

LU Decomposition
Recursive Cache Oblivious Algorithm

```
void rec_diag(int iter, int mat_size) {
    int half = mat_size/2;
    if(half == nesting_size_cutoff) {
#pragma omp task depend( inout: block_list[iter][iter])
        rec_diag (iter, half);
#pragma omp task depend( in: block_list[iter][iter]) \
                 depend( inout: block_list[iter][iter+half])
        rec_row  (iter, iter+half, half);
#pragma omp task depend( in: block_list[iter][iter]) \
                 depend( inout: block_list[iter+half][iter])
        rec_col  (iter, iter+half, half);
#pragma omp task depend( in: block_list[iter][iter+half], block_list[iter
+half][iter]) \
                 depend( inout: block_list[iter+half][iter+half])
        rec_inner(iter, iter+half, iter+half, half);
#pragma omp task depend( inout: block_list[iter+half][iter+half])
        rec_diag (iter+half, half);
    } else if(mat_size == 1) {
        diag_op(block_list[iter][iter]);
    } else {
        rec_diag (iter, half);
        rec_row  (iter, iter+half, half);
        rec_col  (iter, iter+half, half);
        rec_inner(iter, iter+half, iter+half, half);
        rec_diag (iter+half, half);
    }
}
```

LU Decomposition
Recursive Cache Oblivious Algorithm

The recursive cache version improves performance substantially.



LU Decomposition
Recursive Cache Oblivious Algorithm

The recursive cache version improves performance substantially.