Table of Contents

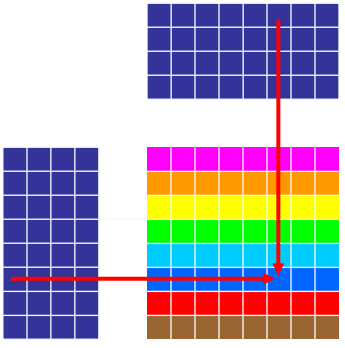
Shared Memory Programming

- Smaller scale parallelism (100's of CPUs/cores)
- Single system image
- Thread-based
- Threads have access to entire shared memory
  - Threads may also have private memory

Shared Memory Programming

- No explicit communication
  - Threads write/read shared data
  - Mutual exclusion used to ensure data consistency
- Explicit Synchronization
  - Ensure correct access order
  - E.g., don't read data until it has been written

Example - Matrix Multiply



```
for(int i=0; i<M; ++i) {
  for(int j=0; j<N; ++j) {
    for(int k=0; k<L; ++k) {
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}
```

One way to parallelize is to compute each row Independently.

Table of Contents

OpenMP

- Allows explicit parallelization of loops
  - Directives for Fortran and C/C++
  - Support for task parallelism

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

- Vendor standard
  - ANSI X3H5 standard in 1994 not adopted
  - OpenMP standard effort started in 1997
  - KAI first to implement new standard

Materials from http://www.llnl.gov/computing/tutorials/openMP/ & ©Intel

The OpenMP Model

Execution Model

- Explicitly parallel
- Single-threaded view
- SPMD
- Implicit data distribution
- Nested parallelism support
- Relaxed consistency within parallel sections

The OpenMP Model

Productivity

- Provides directives for existing languages
- Low level of abstraction
- User level tunability
- Composability supported with nesting of critical sections and parallel loops

Performance

- Load balancing
  - Optional selection of runtime scheduling policy
- Scalable parallelism
  - Parallelism proportional to data size

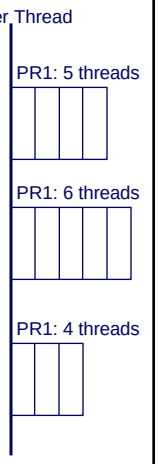
The OpenMP Model

Portability

- Directives allow use of available compilers
  - Application compiles and runs, but no parallelization
- Supports processor virtualization
  - N:1 mapping of logical processes to processors
- Load balancing
  - Optional selection of runtime scheduling policy
- No reliance on system features
  - Can utilize specialized hardware to implement Atomic update

OpenMP Thread Management

- Fork-Join execution model
- User or developer can specify thread count
  - Developer's specification has priority
  - Variable for each parallel region
  - Runtime system has default value
- Runtime system manages threads
  - User/developer specify thread count only
  - Threads "go away" at end of parallel region



OpenMP Thread Management

- Determining number of threads
  - omp_set_num_threads(int) function
  - OMP_NUM_THREADS environment variable
  - Runtime library default
- Threads created only for parallel sections

Creating Parallel Sections

- Parallel for

```
#pragma omp parallel for \
  shared(a,b,c,chunk) \
  private(i) \
  schedule(static,chunk)
for (i=0; i < n; i++)
  c[i] = a[i] + b[i];
```

- Options
  - Scheduling Policy
  - Data Scope Attributes

- Parallel region

```
#pragma omp parallel
{
  // Code to execute
}
```

- Options
  - Data Scope Attributes

Data Scope Attributes

Private	Variables are private to each thread
First Private	Variables are private and initialized with value of original object before parallel region
Last Private	Variables are private and value from last loop iteration or section is copied to original object
Shared	Variables shared by all threads in team
Default	Specifies default scope for all variables in parallel region
Reduction	Reduction performed on variable at end of parallel region
Copy in	Assigns same value to variables declared as thread private

OpenMP Synchronization

- Mutual exclusion by critical sections

```
#pragma omp parallel
{
  // …
  #pragma omp critical
  sum += local_sum
}
```

  - Named critical sections
    - Unnamed sections **treated as one**
  - Critical section is scoped

- Atomic update

```
#pragma omp parallel
{
  // …
  #pragma omp atomic
  sum += local_sum
}
```

  - Specialized critical section
  - May enable fast HW implementation
  - Applies to following statement

OpenMP Synchronization

- Barrier directive
  - Thread waits until all others reach this point
  - Implicit barrier at end of each parallel region

```
#pragma omp parallel
{
  // …
  #pragma omp barrier
  // …
}
```

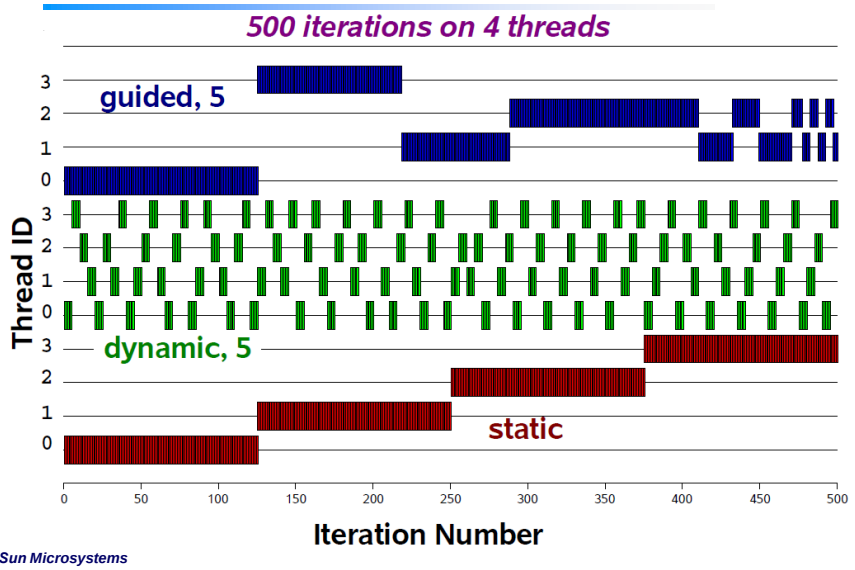
OpenMP Scheduling

- Load balancing handled by runtime scheduler
- Scheduling policy can be set for each parallel loop

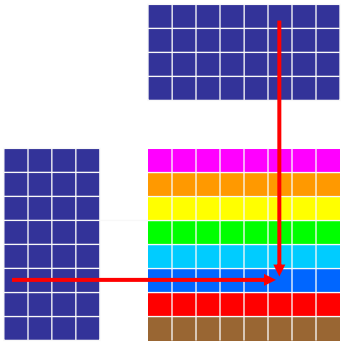
Scheduling Policies

Static	Create blocks of size *chunk* and assign to threads before loop begins execution. Default chunk creates equally-sized blocks.
Dynamic	Create blocks of size *chunk* and assign to threads during loop execution. Threads request a new block when finished processing a block. Default chunk is 1.
Guided	Block size is proportional to number of unassigned iterations divided by number of threads. Minimum block size can be set.
Runtime	No block size specified. Runtime system determines iteration assignment during loop execution.

OpenMP Scheduling



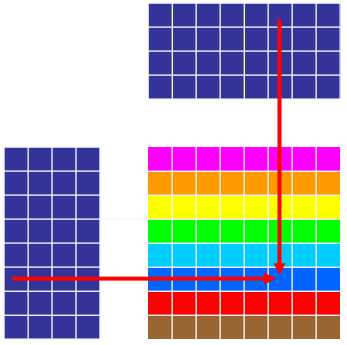
OpenMP Matrix Multiply



```
#pragma omp parallel for
for(int i=0; i<M; ++i) {
  for(int j=0; j<N; ++j) {
    for(int k=0; k<L; ++k) {
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}
```

OpenMP Matrix Multiply

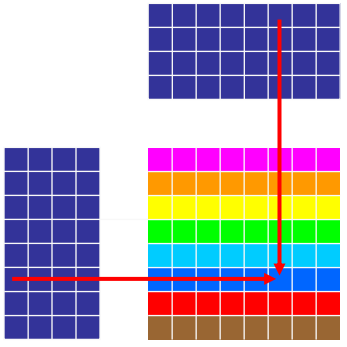
- Parallelizing two loops
  - Uses nested parallelism support
  - Each element of result matrix computed independently



```
#pragma omp parallel for
for(int i=0; i<M; ++i) {
  #pragma omp parallel for
  for(int j=0; j<N; ++j) {
     for(int k=0; k<L; ++k) {
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}
```

OpenMP Matrix Multiply

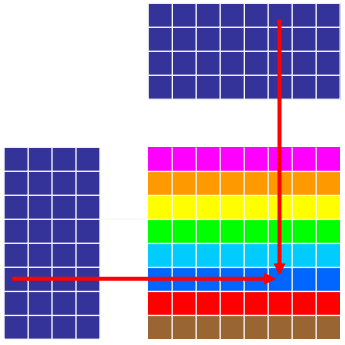
- Parallelizing inner loop
  - Inner loop parallelized instead of outer loop
    - Minimizes work in each parallel loop – for illustration purposes only
  - Multiple threads contribute to each element in result matrix
  - Critical section ensures only one thread updates at a time



```
for(int i=0; i<M; ++i) {
  for(int j=0; j<N; ++j) {
    #pragma omp parallel for
    for(int k=0; k<L; ++k) {
      #pragma omp critical
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}
```

OpenMP Matrix Multiply

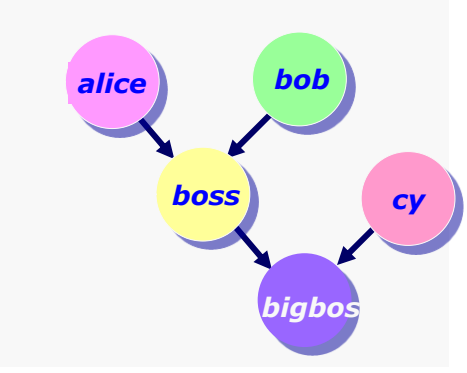
- Use dynamic scheduling of iterations



```
#pragma omp parallel for \
schedule(dynamic)
for(int i=0; i<M; ++i) {
  for(int j=0; j<N; ++j) {
    for(int k=0; k<L; ++k) {
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}
```

OpenMP 3.0 – Task Decomposition

```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n",
  bigboss(s,c));
```



alice,bob, and cy can be computed in parallel

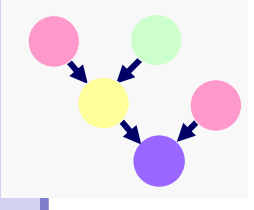
omp sections

- **`#pragma omp sections`**
- Must be inside a parallel region
- Precedes a code block containing of *N* blocks of code that may be executed concurrently by *N* threads
- Encompasses each omp section

- **`#pragma omp section`**
- Precedes each block of code within the encompassing block described above
- May be omitted for first parallel section after the parallel sections pragma
- Enclosed program segments are distributed for parallel execution among available threads

Functional Level Parallelism w sections

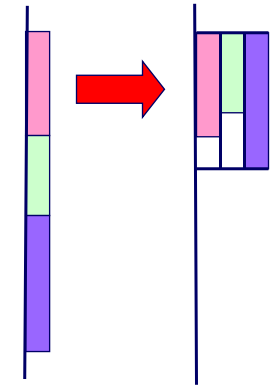
```
#pragma omp parallel sections
{
#pragma omp section  /* Optional */
   a = alice();
#pragma omp section
   b = bob();
#pragma omp section
    c = cy();
}

s = boss(a, b);
printf ("%6.2f\n",
         bigboss(s,c));
```



Advantage of Parallel Sections

- Independent sections of code can execute concurrently – reduce execution time

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```

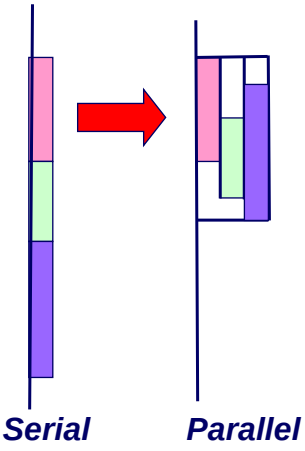


New Addition to OpenMP 3.0

- Tasks – Main change for OpenMP 3.0
- Allows parallelization of irregular problems
  - Unbounded loops
  - Recursive algorithms
  - Producer/consumer

What are tasks?

- Tasks are independent units of work
- Threads are assigned to perform the work of each task
  - Tasks may be deferred
- Tasks may be executed immediately
- The runtime system decides which of the above
  - Tasks are composed of:
    - **Code** to execute
    - **Data** environment
    - **Internal control variables** (ICV)



Simple Task Example

```
#pragma omp parallel
// assume 8 threads
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

A pool of 8 threads is created here

One thread gets to execute the while loop

The single "while loop" thread creates a task for each instance of processwork()

Task Construct – Explicit Task View

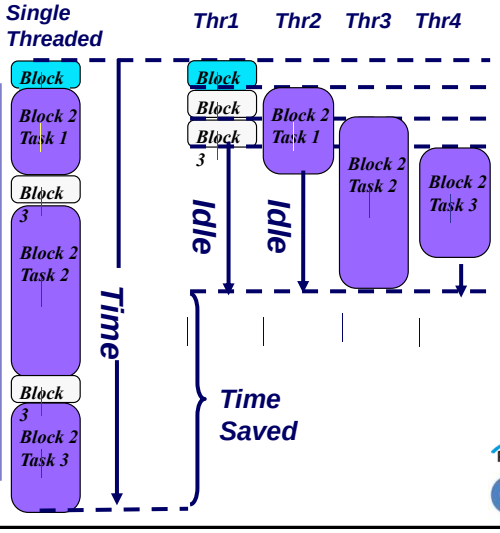
- A team of threads is created at the omp parallel construct
- A single thread is chosen to execute the while loop – lets call this thread "L"
- Thread L operates the while loop, creates tasks, and fetches next pointers
- Each time L crosses the omp task construct it generates a new task and has a thread assigned to it
- Each task runs in its own thread
- All tasks complete at the barrier at the end of the parallel region's single construct

```
#pragma omp parallel
{
   #pragma omp single
   {  // block 1
      node * p = head;
      while (p) {  //block 2
      #pragma omp task
        private(p)
         process(p);
      p = p->next; //block 3
      }
   }
}
```

Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
{
   #pragma omp single
   {  // block 1
      node * p = head;
      while (p) {
//block 2
      #pragma omp task
         process(p);
//block 3
      p = p->next; }
   }
}
```



Linked Lists using Tasks

- **Objective:** Modify the linked list pointer chasing code to implement tasks to parallelize the application

```
while(p != NULL){
  do_work(p->data);
  p = p->next;
}
```

When are tasks guaranteed to be complete?

- ***Tasks are guaranteed to be complete:***
- ***At thread or task barriers***
  - ***At the directive:*** `#pragma omp barrier`
  - ***At the directive:*** `#pragma omp taskwait`

Task Completion Example

```
#pragma omp parallel
{
   #pragma omp task
   foo();
   #pragma omp barrier
   #pragma omp single
   {
      #pragma omp task
      bar();
   }
}
```

Multiple foo tasks created here – one for each thread

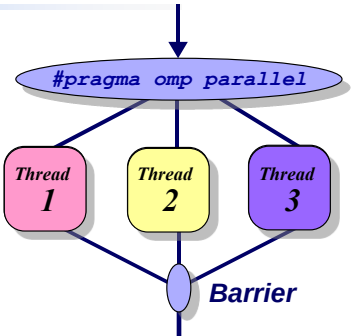
All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

Parallel Construct – Implicit Task View

- Tasks are created in OpenMP even without an explicit task directive.
- Lets look at how tasks are created implicitly for the code snippet below
  - Thread encountering parallel construct packages-up a set of *implicit* tasks
  - Team of threads is created
  - Each thread in team is assigned to one of the tasks (and *tied* to it)
  - Barrier holds original master thread until all implicit tasks are finished




```
#pragma omp parallel
   {
      int mydata;
      code…
   }
```

Task Construct

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

***where** clause can be one of:*

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none  )
```

Tied & Untied Tasks

- Tied Tasks:
  - A tied task gets a thread assigned to it at its first execution and the same thread services the task for its lifetime
  - A thread executing a tied task, can be suspended, and sent of to execute some other task, but eventually, the same thread will return to resume execution of its original tied task
  - Tasks are tied unless explicitly declared untied
- Untied Tasks:
  - An united task has no long term association with any given thread. Any thread not otherwise occupied is free to execute an untied task. The thread assigned to execute an untied task may only change at a "task scheduling point".
  - An untied task is created by appending “untied” to the task clause
  - Example: #pragma omp task untied

Task switching

- **task switching** The act of a *thread* switching from the execution of one *task* to another *task*
- The purpose of task switching is distribute threads among the unassigned tasks in the team to avoid piling up long queues of unassigned tasks
- Task switching, for tied tasks, can only occur at task scheduling points located within the following constructs:
  - Encountered **task** *constructs*
  - Encountered **taskwait** *constructs*
  - Encountered **barrier** *directives*
  - Implicit **barrier** *regions*
  - At the end of the *tied task region*
- Untied tasks have implementation dependent scheduling points

Task switching example

- ***The thread executing the "for loop" , AKA the generating task, generates many tasks in a short time so...***
- ***The SINGLE generating task will have to suspend for a while when "task pool" fills up***
  - ***Task switching is invoked to start draining the "pool"***
  - ***When the "pool" is sufficiently drained – then the single task can begin generating more tasks again***

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

Table of Contents

Pthreads

- Specification part of larger IEEE POSIX standard
  - POSIX is the **P**ortable **O**perating **S**ystem **I**nterface
  - Standard C API for threading libraries
    - IBM provides Fortran API
  - Introduced in 1995
- Explicit threading of application
  - User calls functions to create/destroy threads

Materials from http://www.llnl.gov/computing/tutorials/pthreads/

The Pthreads Model

- Execution Model
  - Explicit parallelism
  - Explicit synchronization
- Productivity
  - Not a primary objective
  - Library for existing language
  - Low level of abstraction
  - Uses opaque objects – prevents expansion

The Pthreads Model

- Performance
  - No attempts to manage latency
  - Load balancing left to OS
  - Developer responsible for creating high degree of parallelism by spawning threads
- Portability
  - Library widely available

Pthreads Thread Management

- User creates/terminates threads
- Thread creation
  - `pthread_create`
  - Accepts a single argument (void *)
- Thread termination
  - `pthread_exit`
  - Called from within terminating thread

Pthreads Synchronization

Mutual Exclusion Variables (mutexes)

- Mutexes must be initialized before use
- Attribute object can be initialized to enable error checking

```
pthread_mutex_t mutexsum;
void *dot_product(void *arg) {
  …
  pthread_mutex_lock (&mutexsum);
  sum += mysum;
  pthread_mutex_unlock (&mutexsum);
  …
}
int main() {
  pthread_mutex_init(&mutexsum, NULL);
  …
  pthread_mutex_destroy(&mutexsum);
}
```

Pthreads Synchronization

Condition Variables

- Allows threads to synchronize based on value of data
- Threads avoid continuous polling to check condition
- Always used in conjunction with a mutex
  - Waiting thread(s) obtain mutex then wait
    - pthread_cond_wait() function unlocks mutex
    - mutex locked for thread when it is awakened by signal
  - Signaling thread obtains lock then issues signal
    - pthread_cond_signal() releases mutex

Condition Variable Example

Two threads update a counter
Third thread waits until counter reaches a threshold

```
pthread_mutex_t mtx;
pthread_cond_t cv;

int main() {
…
pthread_mutex_init(&mtx, NULL);
pthread_cond_init (&cv, NULL);
…
pthread_create(&threads[0], &attr,
               inc_count, (void *)&thread_ids[0]);
pthread_create(&threads[1], &attr,
               inc_count, (void *)&thread_ids[1]);
pthread_create(&threads[2], &attr,
               watch_count, (void *)&thread_ids[2]);
…
}
```

Condition Variable Example

Incrementing Threads

```
void *inc_count(void *idp) {
…
for (i=0; i<TCOUNT; ++i) {
  pthread_mutex_lock(&mtx);
  ++count;
  if (count == LIMIT)
    pthread_cond_signal(&cv);
  pthread_mutex_unlock(&mtx);
  …
}
…
}
```

Waiting Thread

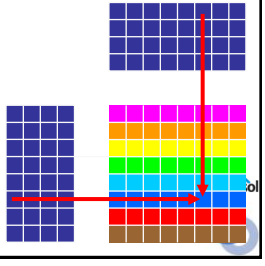
```
void *watch_count(void *idp) {
…
pthread_mutex_lock(&mtx);
while (count < COUNT_LIMIT) {
  pthread_cond_wait(&cv, &mtx);
}
pthread_mutex_unlock(&mtx);
…
}
```

pthread_cond_broadcast() used if multiple threads waiting on signal

Pthreads Matrix Multiply

```
int tids[M];
pthread_t threads[M];
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(
  &attr,
  PTHREAD_CREATE_JOINABLE);

for (i=0; i<M; ++i) {
  tids[i] = i;
  pthread_create(&threads[i],
   &attr, work, (void *) &tids[i]);
}

for (i=0; i<M; ++i) {
  pthread_join(threads[i], NULL);
}
```

```
void* work(void* tid) {
  for(int j=0; j<N; ++j) {
    for(int k=0; k<L; ++k) {
      C[tid][j] +=
        A[tid][k]*B[k][j];
    }
  }
  pthread_exit(NULL);
}
```



References

OpenMP

- http://www.openmp.org
- http://www.llnl.gov/computing/tutorials/openMP

Pthreads

- http://www.llnl.gov/computing/tutorials/pthreads
- "Pthreads Programming". B. Nichols et al. O'Reilly and Associates.
- "Programming With POSIX Threads". D. Butenhof. Addison Wesley