CS528
Multi-threading and OpenMP

A Sahu

Dept of CSE, IIT Guwahati

Outline

- Thread Safety : 4 Classic cases
- Thread Synchronization: Basic
- **Implicit/Auto Thread Pooling: OpenMP/Cilk**

Shared Variables in Threaded C Programs

- Question: Which variables in a threaded C program are shared variables?
  - Answer not as simple as "global variables are shared" and "stack variables are private"
- Requires answers to the following questions:
  - What is the memory model for threads?
  - How are variables mapped to memory instances?

Parallel Counter: without Lock

```
#define NITERS 100
int cnt = 0; /* shared */
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, count, NULL);
  pthread_create(&tid2, NULL, count, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  if(cnt!=NITERS*2) printf("BOOM! cnt=%d",cnt);
    else printf("OK cnt=%d\n", cnt);
}
```

```
void *count(void *arg) {
for(int i=0;i<NITERS;i++) cnt++;
}
```

```
$./badcnt
BOOM! cnt=196
$./badcnt
BOOM! cnt=184
```

cnt should be 200

What went wrong?!

Thread Safety

- Functions called from a thread must be *thread-safe*
- There are four (non-disjoint) classes of thread-unsafe functions:
  - **Class 1: Failing to protect shared variables : L/UL**
  - Class 2: Relying on persistent state across invocations
  - Class 3: Returning pointer to static variable
  - Class 4: Calling thread-unsafe functions

Class 1: Failing to protect shared variables

- Fix: Use Lock and unlock semaphore operations
- Issue: Synchronization operations will slow down code
- Example: `goodcnt.c`

```
void *count(void *arg) {
for(int i=0;i<NITERS;i++)
      pthread_mutex_lock(&LV);
      cnt++;
      pthread_mutex_unlock(&LV);
} // LV is lock variable
```

Class 2: Relying on persistent state across multiple function invocations

- Random number generator relies on static state
- Fix: Rewrite function so that caller passes in all necessary state, ➔ Maintain Thread Specific State

```
int rand() {
    static uint next = 1;
    next = next*1103515245 + 12345;
    return (uint)(next/65536)% 32768;
}
void srand(uint seed) {
    next = seed;
}
```

Class 3: Returning pointer to `static` variable

- Fixes: 1. Rewrite code so caller passes pointer to `struct`, Issue: Requires changes in caller and callee
- *Lock-and-copy :* Issue: Requires only simple changes in caller (and none in callee), However, caller must free memory

```
struct hostent  *gethostbyname(
                              char* name){
  static struct hostent h;
  <contact DNS and fill in h>
  return &h;
}
```

Class 3: Returning pointer to `static` variable

```
struct hostent *gethostbyname_ts(char *p) {
  struct hostent *q = Malloc(...);
  P(&mutex); /* lock */
  p = gethostbyname(name);
  *q = *p;   /* copy */
  V(&mutex);
  return q;
}
```

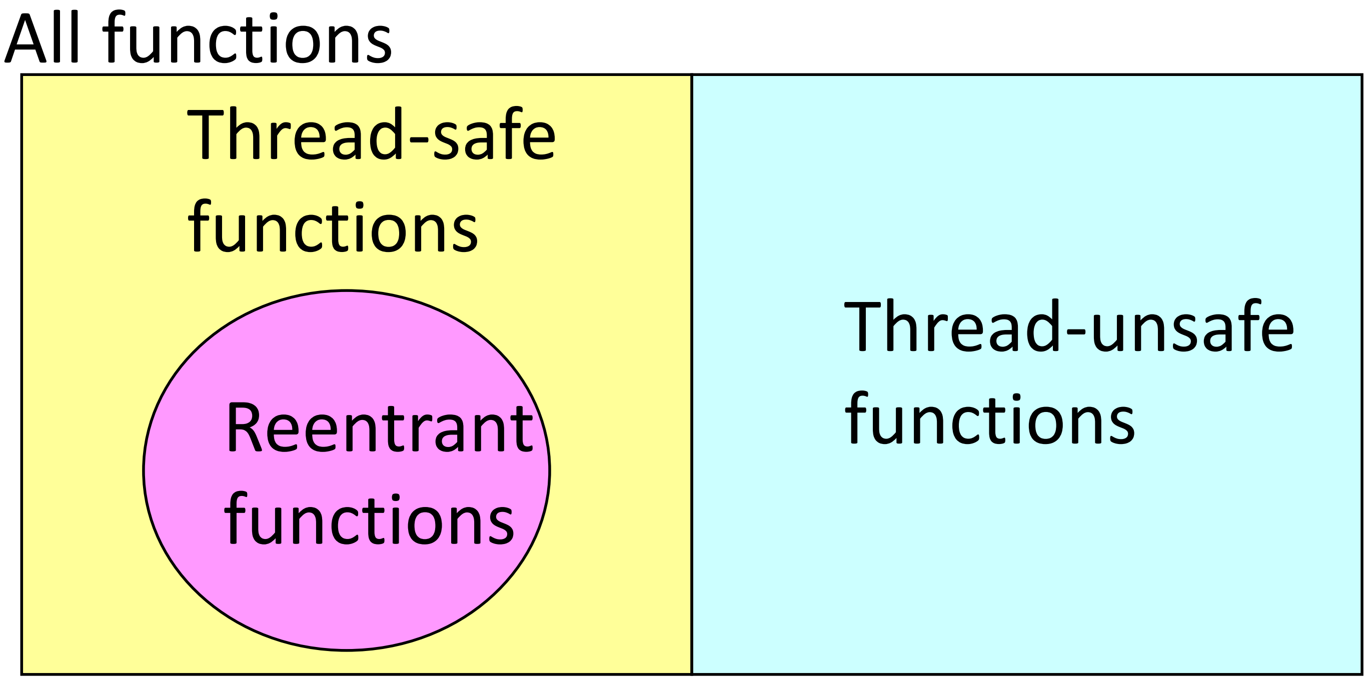
```
hostp = malloc(...));
gethostbyname_r(name, hostp);
```

Class 4: Calling thread-unsafe functions

- Calling one thread-unsafe function makes an entire function thread-unsafe
- Fix: Modify the function so it calls only thread-safe functions

Reentrant Functions

- A function is *reentrant* iff it accesses NO shared variables when called from multiple threads
  - **Reentrant functions are a proper subset of the set of thread-safe functions**
  - NOTE: The fixes to Class 2 and 3 thread-unsafe functions require modifying the function to make it reentrant (only first fix for Class 3 is reentrant)

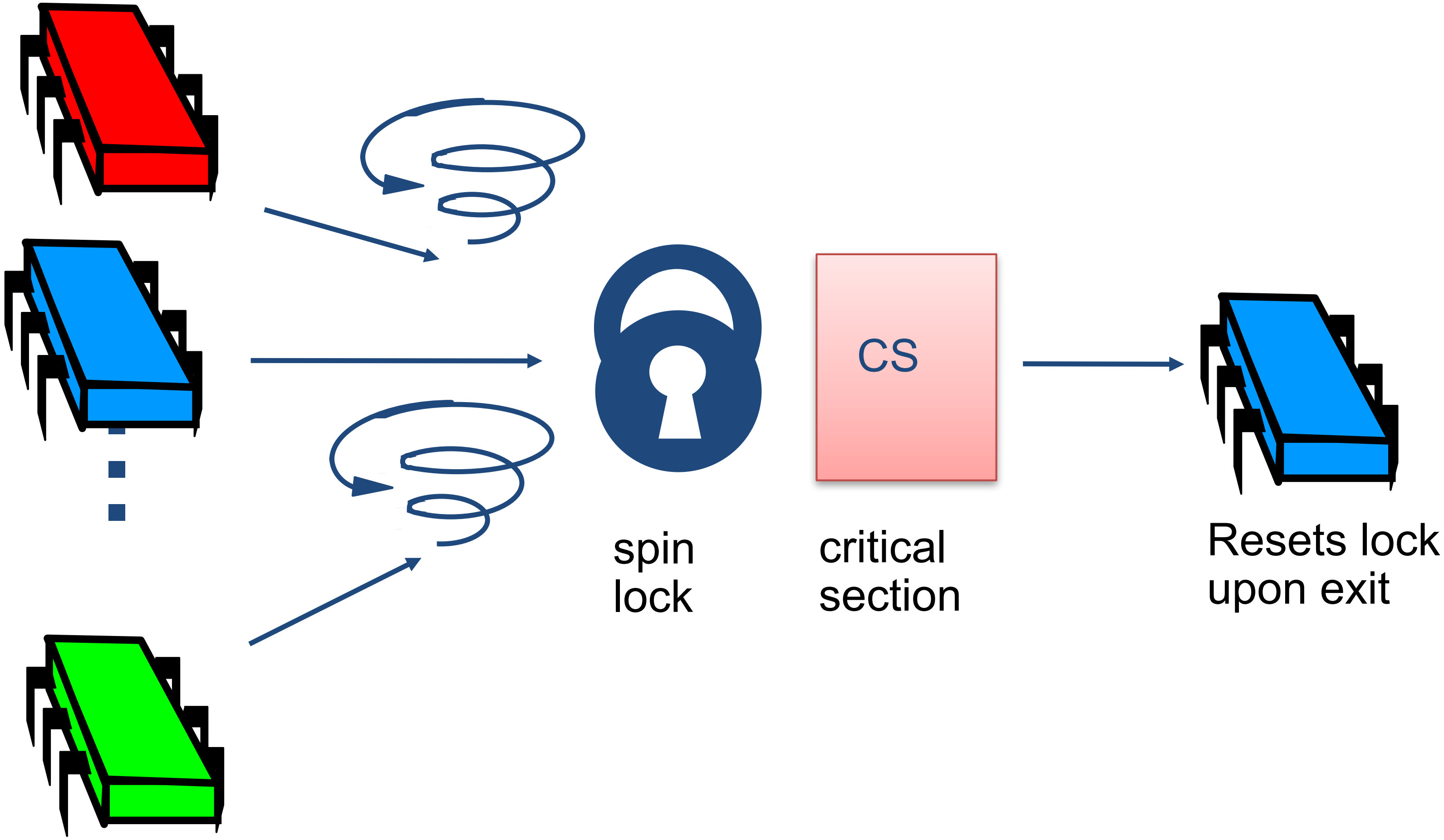


Thread-Safe Library Functions

- Most functions in the Standard C Library (at the back of your K&R text) are thread-safe
  - Examples: **`malloc`**, **`free`**, **`printf`**, **`scanf`**
- All Unix system calls are thread-safe
- Library calls that aren't thread-safe:

Thread-unsafe function	Class	Reentrant version
`asctime`	3	`asctime_r`
`ctime`	3	`ctime_r`
`gethostbyaddr`	3	`gethostbyaddr_r`
`gethostbyname`	3	`gethostbyname_r`
`inet_ntoa`	3	(none)
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`

Many threads trying to acquire Mutex LOCK



Synchronization Primitives

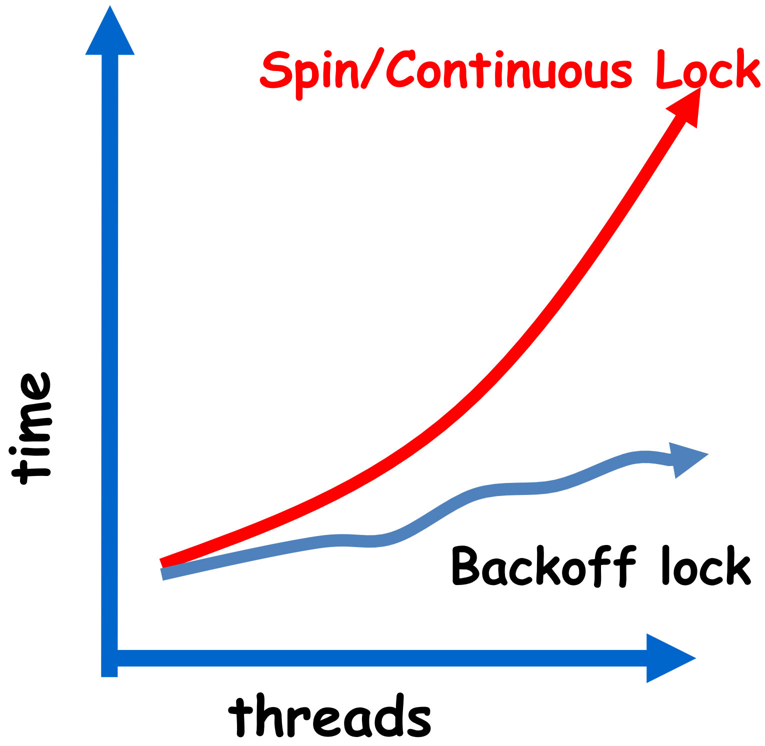
- **int** pthread_mutex_**init**(
  pthread_mutex_t *mutex_lock,
  const pthread_mutexattr_t *lock_attr);
- **int** pthread_mutex_**lock**(
  pthread_mutex_t *mutex_lock);
- **int** pthread_mutex_**unlock**(
  pthread_mutex_t *mutex_lock);
- **int** pthread_mutex_**trylock**(
  pthread_mutex_t *mutex_lock);

Locking Overhead

- Serialization points
  - Minimize the size of critical sections
  - Be careful
- Rather than wait, check if lock is available
  - **pthread_mutex_trylock**
  - If already locked, will return EBUSY
  - Will require restructuring of code
    - **Suspend self by pthread_yeild() Give chance to others**
    - **Suspend self by doing a timed wait..**
      - **{1, 1, 1,..}, {1, 2, 3, 4,..}, {1,2,4,8,..},…**

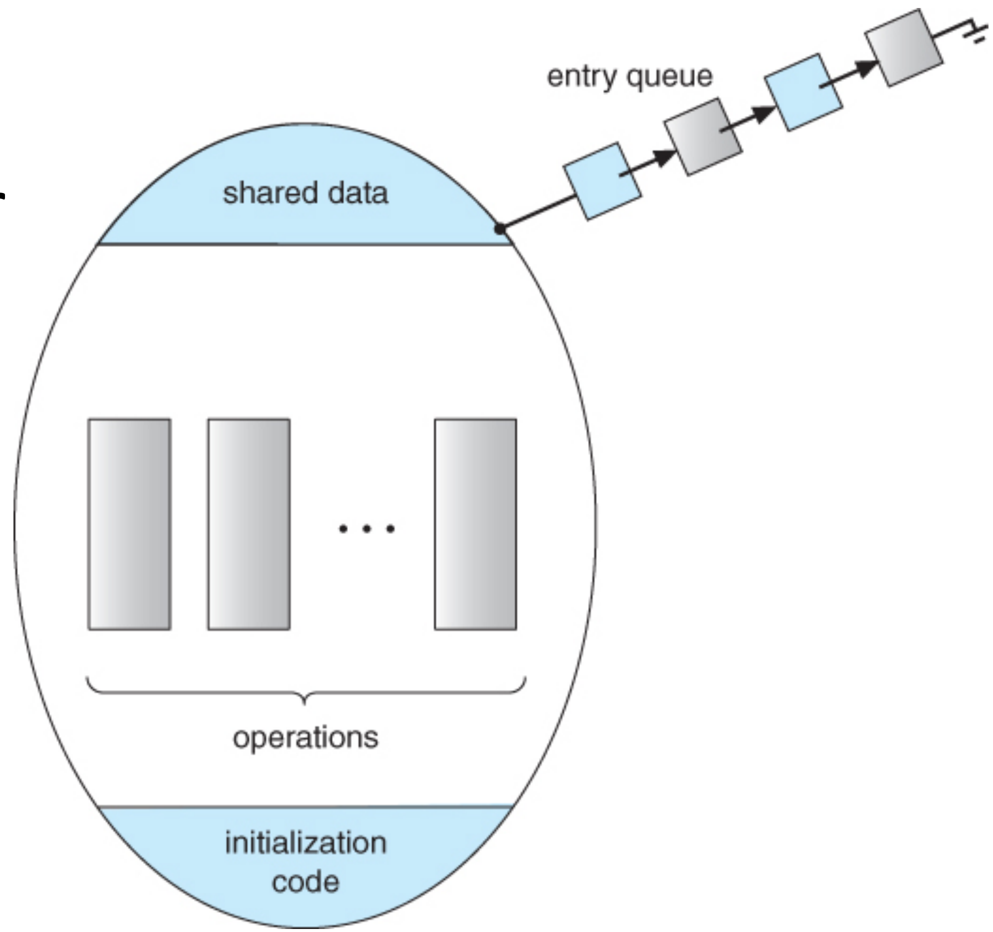
Performance of Locking

- **Spinning/busy wait waste time**
- Recall MAC Protocol
  - Non Persistence CSMA protocol
  - Wait random time if medium if busy, then send
- Spin lock with exponential back-off reduces contention
  - Wait k amount of time for 1^{st} attempt
  - Wait $k*c^i$ amount of time for i^{th} attempt



Performance of Locking: Monitor

- Spinning/busy wait waste time
- There should be methods for
  - Queuing the shared accessor
  - Calling the next guy when one guy finishes his work
- Best example is : Barber shop
  - Book using Phone, get call before some time of your turn



Locking in Data Structure

- Concurrent Data Structure
  - CDS-List, CSE-Stack, CDS-Queue
  - CDS-PQ, CDS-Hash, CSD-Heap
- Given a Bank with 10^6 account
  - It is not worth to lock whole DB of bank
  - The bank DB may be maintained by List/Hash
  - Suppose A transfer money to B, It is preferred to lock A and Lock B to do the transaction (A.bal-X) and (B.bal+X)
  - Fine grain locking
- Book: Wait free and lock free DS
  - C++ in action : Mileswky
  - Art of multiprocessor programming, by Shavit & Herlihy: **Godel prize winner**

Synch. Primitives

- $\boxed{\text{pthread\_mutex\_init, lock, unlock, trylock}}$
- pthread_cond_wait, signal, broadcast, init, destroy

Condition Variables for Synchronization

- Condition variable allows a thread
  - To block itself until specified data reaches a predefined state.
- Condition variable
  - Associated with a predicate (P)
  - When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition.
- Single condition variable
  - May be associated with more than one predicate
  - Ex: P= X OR Y AND (Z OR K)

Synchronization Hierarchy ☺ ☺ ☺

- One == (used by)== > other
- **LL+SC** ==> **TAS/CAS/FAI/XCHG**==>**Lock/Unlock**
  - All **TAS/CAS/GAS/FAI/XCHG** do the same work
- Lock/Unlock == > Mutex //Mutex use L/UL
- Mutex == > Semaphore // Semaphore uses Mutex
  - Wait() and Signal()
- Semaphore == > Monitor //Monitor uses Semaphore
  - Many wait/Many Signal, Processes in Queue
  - Monitor : Another Abstract Type
    - *which use semaphore, mutex, conditions*

Lock Variable in Mutithreaded App

- Lock Variable (LV): In single core machine looks fine
- **Multithreaded App: Run on Multicore**
- Private Caches: available on Multiple Cores
- **Can the Lock variable be cached?**
  - Lock variable will be at private cache different cores
  - Who will maintain the consistency?
  - Working of Lock variable require: Atomic Transaction
  - Ans: Cache coherence protocol responsible to maintain state of lock variable

C++ Thread:atomic

```
atomic_uint AtomicCount;
void DoCount(){
   int j, timesperthrd;
   timesperthrd=(TIMES/NUM_THREADS);
   for(j=0;j<timesperthrd;j++) AtomicCount++;
}

main(){
  thread T[N_THRDS]; int i;
  for(i=0;i<N_THRDS;i++) T[i]=thread(DoCount);
  for(i=0;i<N_THRDS;i++) T[i].join();
}
```

Improved

```
atomic_uint AtomicCount;
void DoCount(){
  int j, timesperthrd,localcount=0;
  timesperthrd=(TIMES/NUM_THREADS);
  for(j=0;j<timesperthrd;j++)localcount++;
           AtomicCount+=localcount;
}
main(){
     thread T[N_THRDS]; int i;
  for(i=0;i<N_THRDS;i++) T[i]=thread(DoCount);
  for(i=0;i<N_THRDS;i++) T[i].join();
}
```

OpenMP

OpenMP

- Compiler directive: Automatic parallelization
- Auto generate thread and get synchronized

```
#include <openmp.h>
main(){
#pragma omp parallel
#pragma omp for schedule(static)
{
   for (int i=0; i<N; i++) {
       a[i]=b[i]+c[i];
   }
}
}
```

```
$ gcc –fopenmp  test.c
$ export OMP_NUM_THREADS=4
$./a.out
```

OpenMP: Parallelism

Sequential code

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

OpenMP: Parallelism

(Semi) manual parallel

```
#pragma omp parallel
{
 int id =omp_get_thread_num();
 int Nthr=omp_get_num_threads();
 int istart = id*N/Nthr
 int iend= (id+1)*N/Nthr;
 for (int i=istart;i<iend;i++){
       a[i]=b[i]+c[i];
   }
}
```

OpenMP: Parallelism

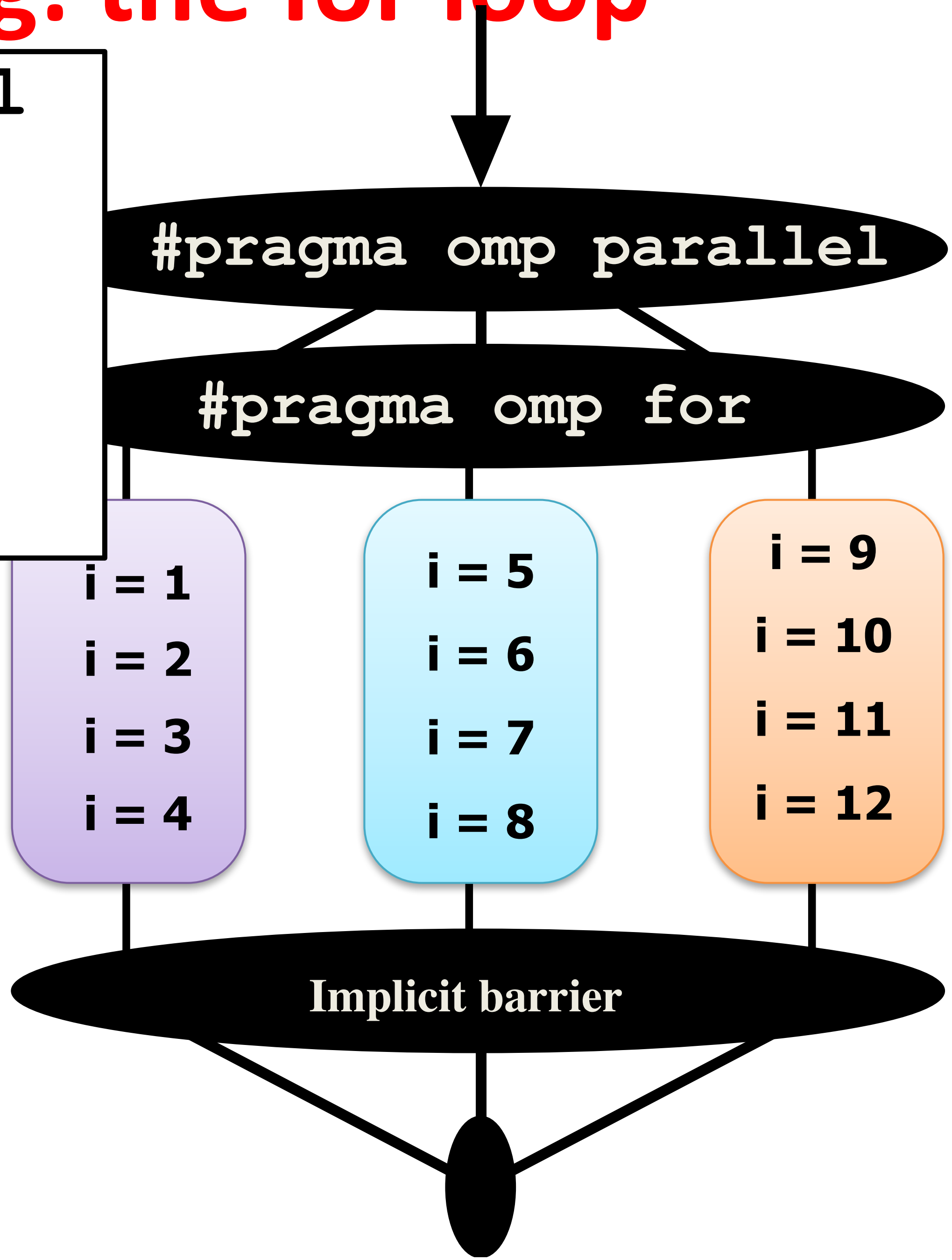
Auto parallel for loop

```
#pragma omp parallel
#pragma omp for schedule(static)
{
  for (int i=0; i<N; i++){
    a[i]=b[i]+c[i];
  }
}
```

Work-sharing: the for loop

```
#pragma omp parallel
#pragma omp for
{
  for(i=1;i<13;i++)
    c[i]=a[i]+b[i];
}
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct



OpenMP Fork-and-Join model

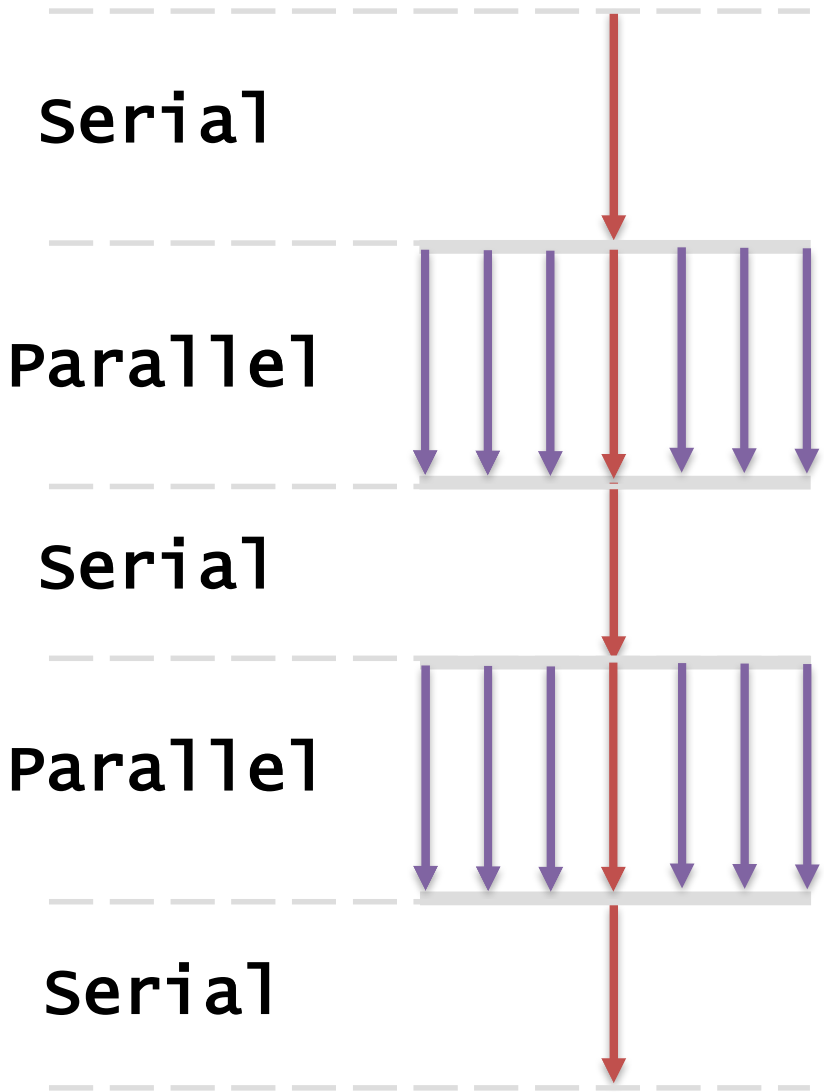
```
printf("begin\n");
N = 1000;

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

M = 500;

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

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



AutoMutex: Critical Construct

```
sum = 0;
#pragma omp parallel private (lsum)
{
    lsum = 0;
    #pragma omp for
    for (i=0; i<N; i++) {
      lsum = lsum + A[i];
    }
    #pragma omp critical
    { sum += lsum; }
}
```

Threads wait their turn;
only one thread at a time

Reduction Clause

Shared variabl

```
sum = 0;
#pragma omp parallel for reduction (+:sum)
{
  for (i=0; i<N; i++) {
    sum = sum + A[i];
  }
}
```

OpenMP Schedule

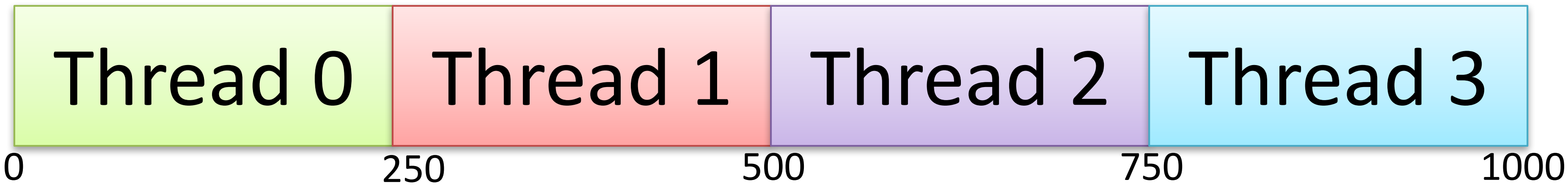
- Can help OpenMP decide how to handle parallelism

  schedule(type [,chunk])

- **Schedule Types**
  - **Static** – Iterations divided into size chunk, if specified, and statically assigned to threads
  - **Dynamic** – Iterations divided into size chunk, if specified, and dynamically scheduled among threads

Static Schedule

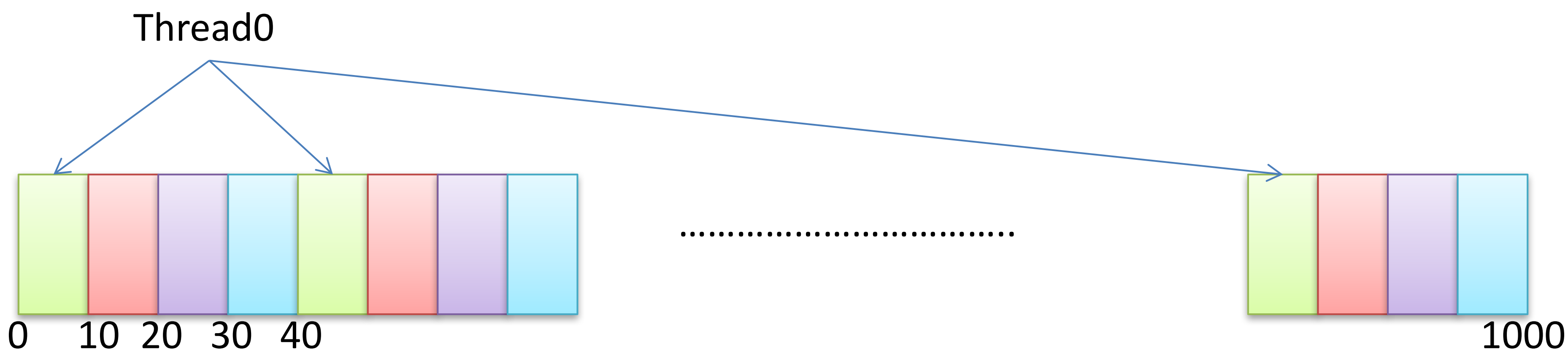
- Although the OpenMP standard does not specify how a loop should be partitioned
- Most compilers split the loop in N/p (N #iterations, p #threads) chunks by default.
- This is called a static schedule (with chunk size N/p)
  - *For example, suppose we have a loop with 1000 iterations and 4 omp threads. The loop is partitioned as follows:*



Static Schedule with chunk

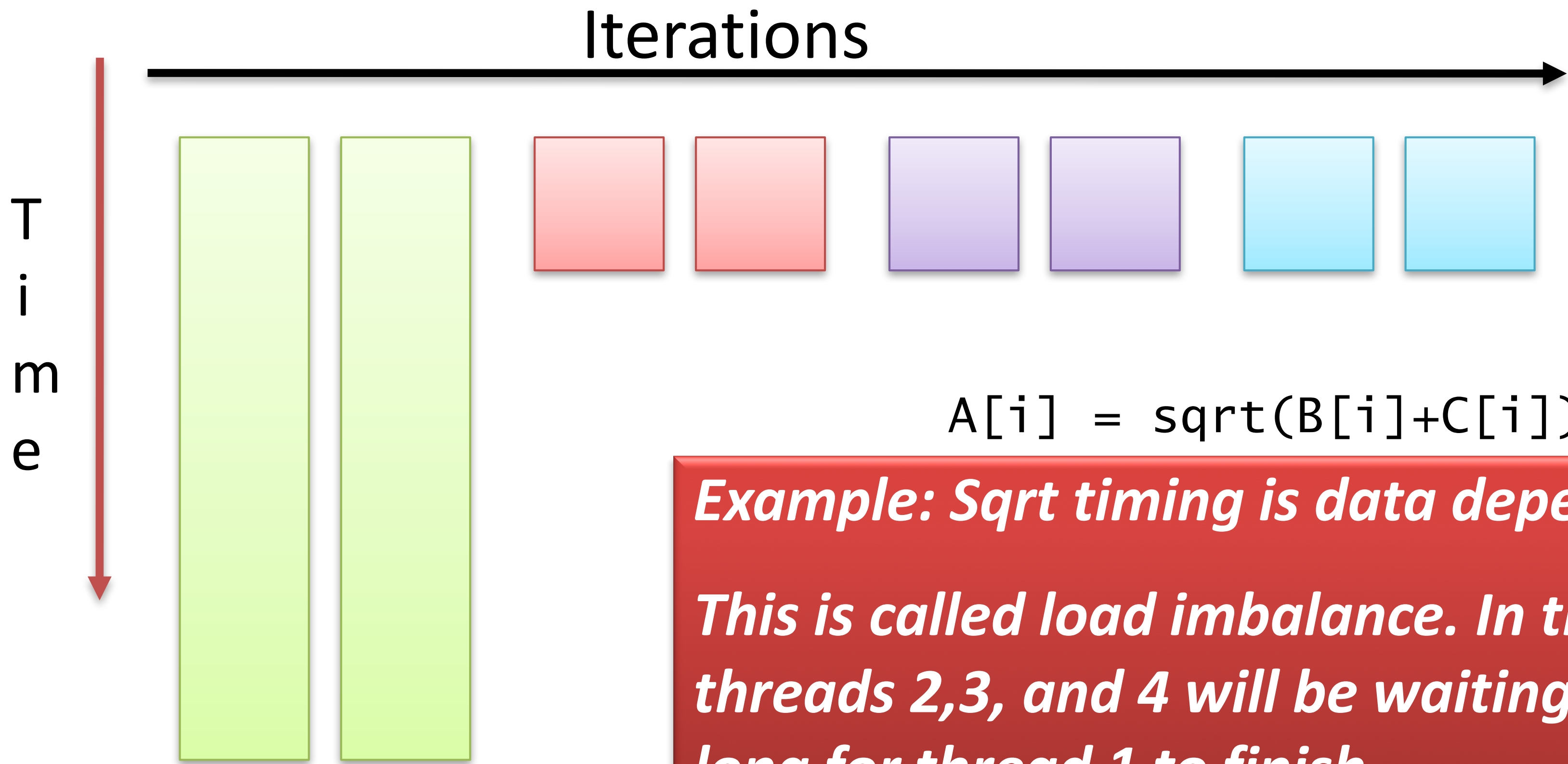
- A loop with 1000 iterations and 4 omp threads. Static Schedule with Chunk 10

```
#pragma omp parallel for schedule (static, 10)
{
for (i=0; i<1000; i++)
    A[i] = B[i] + C[i];
}
```



Issues with Static schedule

- With static scheduling the number of iterations is evenly distributed among all openmp threads (i.e. Every thread will be assigned similar number of iterations).
- This is not always the best way to partition. Why is This?




```
A[i] = sqrt(B[i]+C[i]);
```

Example: Sqrt timing is data dependent…

This is called load imbalance. In this case threads 2,3, and 4 will be waiting very long for thread 1 to finish

Dynamic Schedule

- With a dynamic schedule new chunks are assigned to threads when they come available.
- SCHEDULE(DYNAMIC,n)
  - Loop iterations are divided into pieces of size chunk. When a thread finishes one chunk, it is dynamically assigned another.

Dynamic Schedule

- SCHEDULE(GUIDED,n)
  - Similar to DYNAMIC but chunk size is relative to number of iterations left.
- Although Dynamic scheduling might be the prefered choice to prevent load inbalance
  - In some situations, there is a significant overhead involved compared to static scheduling.

More Examples on OpenMP

- http://users.abo.fi/mats/PP2012/examples/OpenMP/
- https://people.sc.fsu.edu/~jburkardt/c_src/mxm_openmp/mxm_openmp.html