# CS528
# OpenMP and Cilk

A Sahu

Dept of CSE, IIT Guwahati

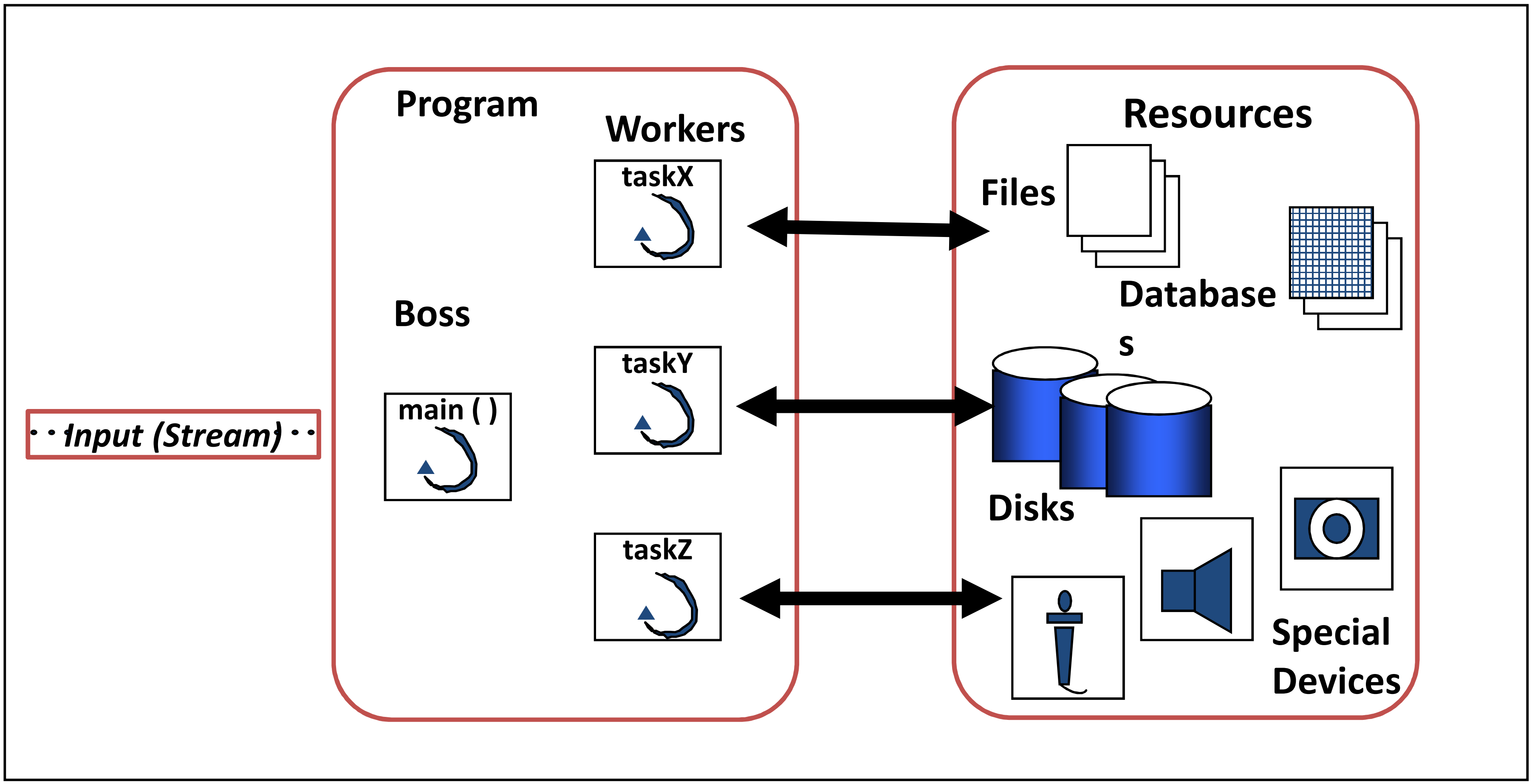
# Outline

- Thread Programming Model
  - Boss/Worker, Peers, Pipeline
- Thread Pooling: Example : Manual
- **Implicit/Auto Thread Pooling: OpenMP/Cilk**
- **OpenMP**
- **Cilk**

# Thread Programming models

- The boss/worker model
- The peer model
- A thread pipeline

# The boss/worker model



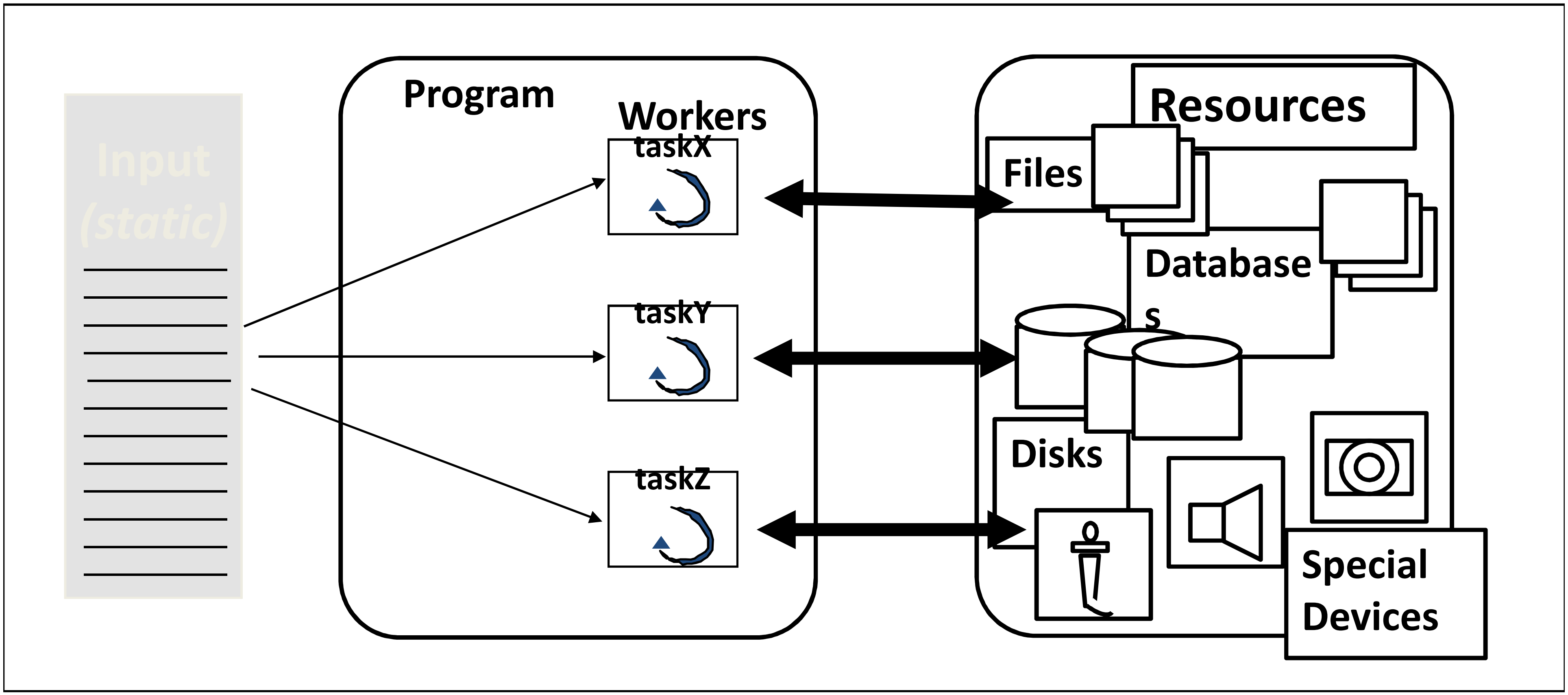
# Example

```
main(){ /* the boss */
   do(1) {
      get a request;
      switch( request )
      case X: pthread_create(....,taskX);
      case Y: pthread_create(....,taskY);…
   }
}
taskX(){/* worker */
   perform the task, sync if accessing shared
   resources
}
taskY(){/* worker */
   perform the task, sync if accessing shared
   resources
```

# Example

- Above runtime overhead of creating thread
- Can be solved by **thread pool**
  - the boss thread creates all worker thread at program Initialization
  - and each worker thread suspends itself immediately for a wakeup call from boss

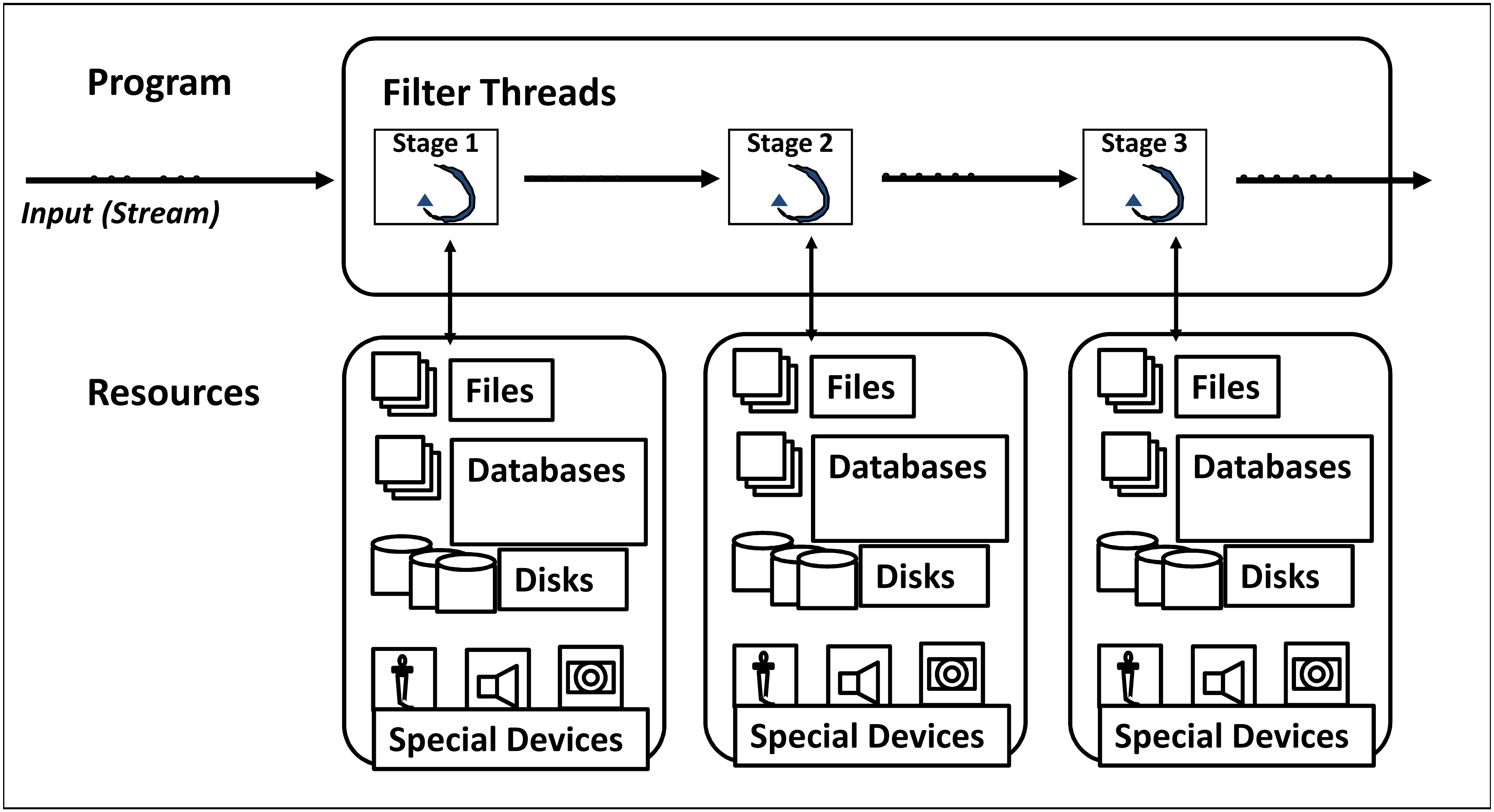
# The peer model



# Example

```
main(){
  do(1){
  thread_create(....,thread1...task1);
  thread_create(....,thread2...task2);
     ....
  signal all workers to start
  wait for all workers to finish
  do any cleanup
  }
}
```

# A thread pipeline



# Example

```
main() {
  pthread_create(....,stage1);
  pthread_create(....,stage2);
  ....
  wait for all pipeline threads to finish
  do any cleanup
}
stage1() {
  get next input for the program
  do stage 1 processing of the input
  pass result to next thread in pipeline
}
```

# Example

```
stage2(){
  get input from previous thread in pipeline
  do stage 2 processing of the input
  pass result to next thread in pipeline
}
stageN()
{
  get input from previous thread in pipeline
  do stage N processing of the input
  pass result to program output.
}
```

# Thread Pooling

# Thread Pool

- Initialized all threads/worker to wait for the task
- Thread pool : assign work to thread
- Thread goes to wait/sleep mode until they have don’t have task
- Thread pool maintain a queue of task
- If any task comes, it will awake one thread if some thread is in sleep/wait mode

# Thread Pool

```
class ThreadPool    {
public: ThreadPool(); ~ThreadPool();
    ThreadPool( size_t threads );
    void initializeWithThreads( size_t threads );
    void schedule( const function<void()>& );
   //waits until threads have processed all tasks of Q.
    void wait() const;
private:
    vector<thread>                _workers;
    queue<function<void()>>       _taskQueue;
    atomic_uint                   _taskCount;
    mutex                         _mutex;
    condition_variable            _condition;
    atomic_bool                   _stop;
};
```

# Thread Pool

```
ThreadPool::ThreadPool()
    : _workers(), _taskQueue(), _taskCount( 0u
  ),_mutex(),  _condition(), _stop( false ) {}

ThreadPool::ThreadPool( size_t threads
  ):ThreadPool() {
    initializeWithThreads( threads );
}
ThreadPool::~ThreadPool() {
    _stop = true;
    _condition.notify_all();
    for ( thread& w: _workers ) w.join();
}
```

# Thread Pool

```
void ThreadPool::InitWithThreads(size_t threads) {
 for ( size_t i = 0; i < threads; i++ ) {
  _workers.emplace_back([this]() -> void {
   while (true) { function<void()> task;
   { unique_lock<mutex> lock( _mutex );
   condition.wait(lock,_taskQueue.empty()||_stop);
     if (_stop && _taskQueue.empty()) return;
     task = move( _taskQueue.front() );
      _taskQueue.pop();
   }    //release lock
    task(); _taskCount--; //atomic DEC
   }  //while
   });
 }}  //for & TP
```

# Thread Pool

```
void ThreadPool::schedule( const
  function<void()>& task ){
    unique_lock<mutex> lock( _mutex );
    _taskQueue.push( task );
    _taskCount++;
    _condition.notify_one();
}
void ThreadPool::wait() const {
    while ( _taskCount != 0u ) {
    this_thread::sleep_for(
  chrono::microseconds(1) );
    }
}
```

# Thread Pool

```
void w1(){cout<<"1\n";}
void w2(){cout<<"2\n";}
void w3(){cout<<"3\n";}
int main(){
     ThreadPool TP(4); //with 4 threads
     for(int i=0;i<100;i++){int x=i%4;
       switch (x){
          case 0: TP.schedule(w1);
          case 1: TP.schedule(w2);
          case 2: TP.schedule(w3);
       }
     }
  return 0;
}
```

# Implicit Threading
# Example: OpenMP, TBB and Cilk

# 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];
   }
}
```

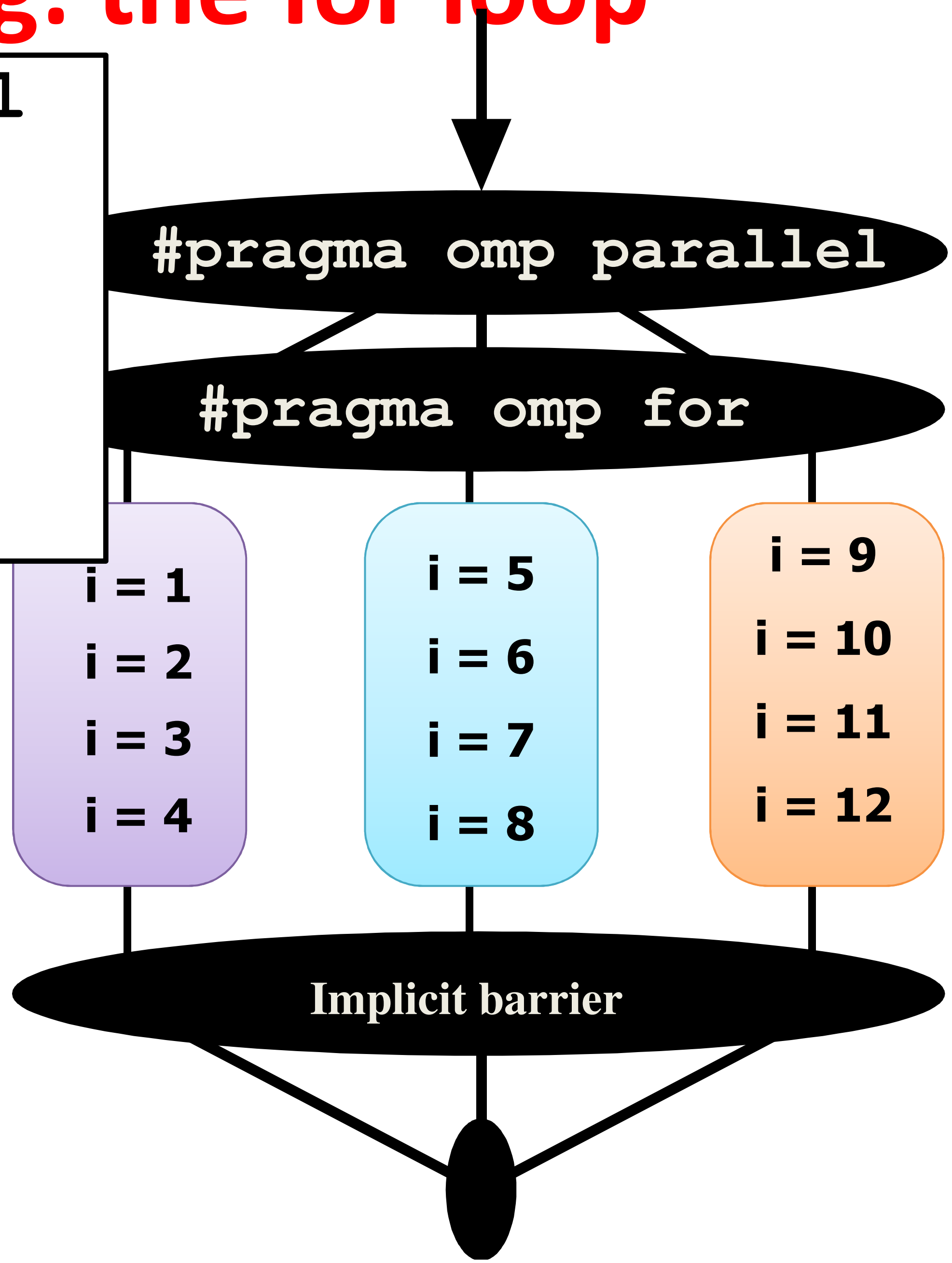
# OpenMP: Parallelism

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

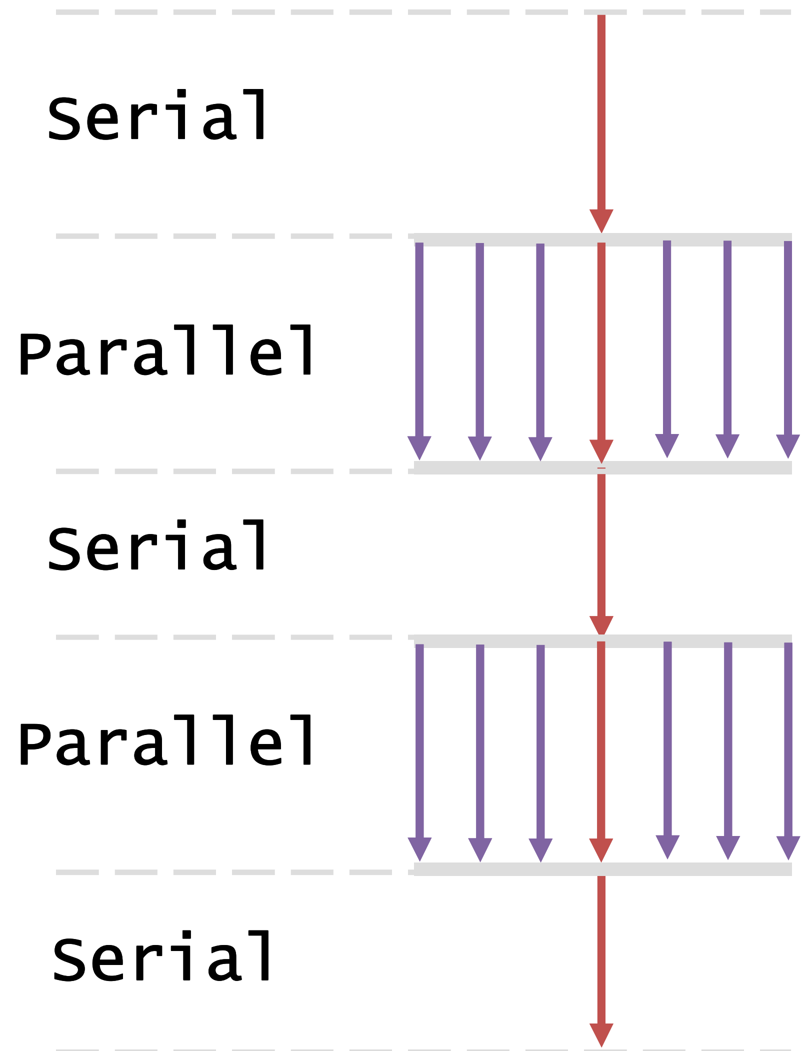
```
printf(“begin\n”);
N = 1000;

#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 executes the critical section**

# Reduction Clause

Shared variable

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

# OpenMP Schedule

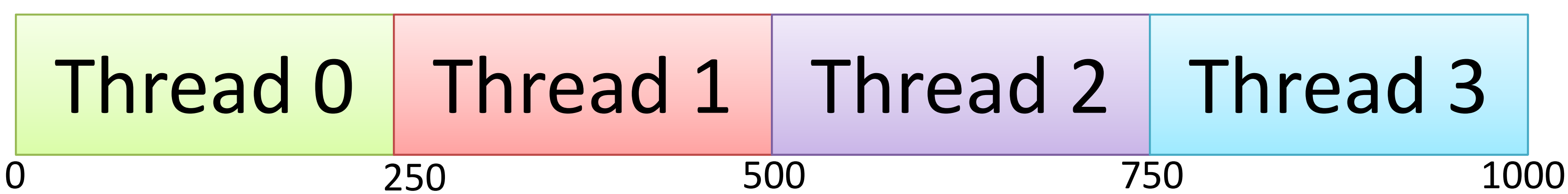
- Can help OpenMP decide how to handle parallelism

    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

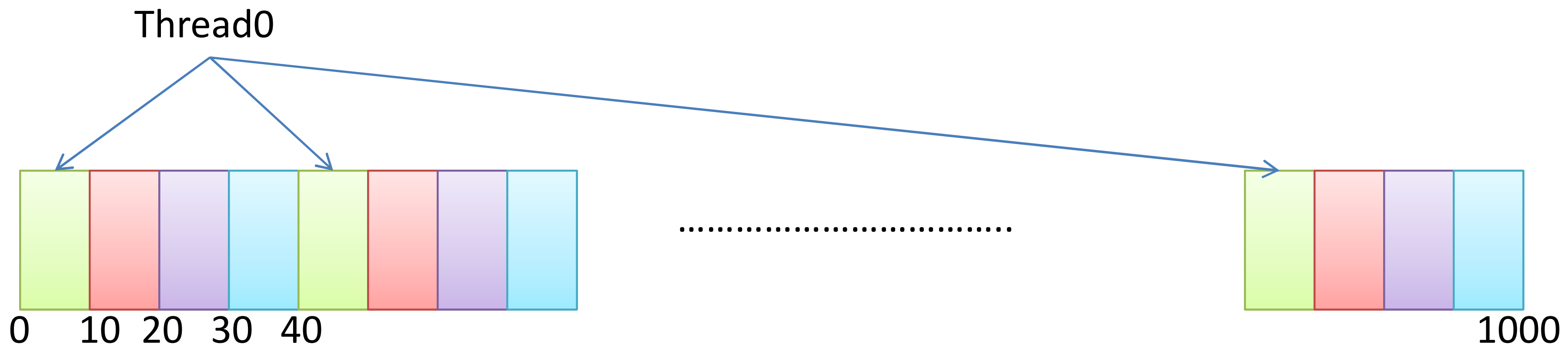
- Although the OpenMP standard does not specify how a loop should be partitioned
- Most compilers split the loop in N/p (N #iterations, p #threads) chunks by default.
- This is called a static schedule (with chunk size N/p)
  - *For example, suppose we have a loop with 1000 iterations and 4 omp threads.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



# Static Schedule with chunk

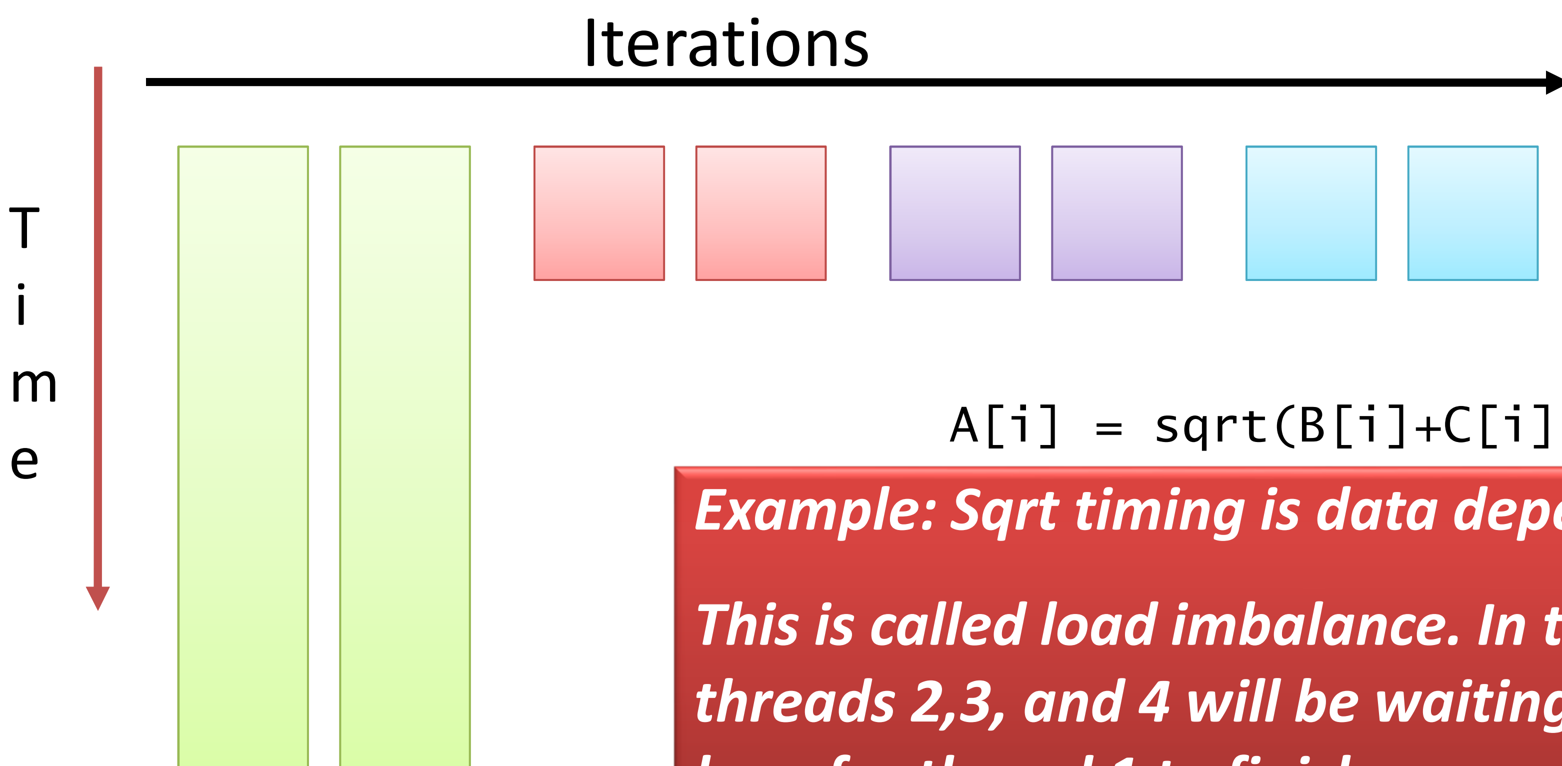
- A loop with 1000 iterations and 4 omp threads. Static Schedule with Chunk 10

```
#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/

# CS528
# Cilk

**Slides are adopted from**
http://supertech.csail.mit.edu/cilk/
Charles E. Leiserson

A Sahu
Dept of CSE, IIT Guwahati

# Cilk

- Developed by **Leiserson at CSAIL, MIT**
  - **Chapter 27, Multithreaded Algorithm, Introduction to Algorithm, Coreman, Leiserson and Rivest**
- Initiated a startup: Cilk Plus
  - Added Cilk_for Keyword, Cilk Reduction features
  - Acquired by Intel, Intel uses Cilk Scheduler
- Addition of 6 keywords to standard C
  - Easy to install in linux system
  - With gcc and pthread

# Cilk

- In 2008, ACM SIGPLAN awarded **Best influential paper of Decade**
  - **The Implementation of the Cilk-5 Multithreaded Language**, PLDI 1998
- PLDI 2008 Best paper Award
  - Reducers and Other Cilk++ Hyperobjects , PLDI 2008

# Cilk : Biggest principle

- Programmer should be responsible for
  - Exposing the parallelism,
  - Identifying elements that can safely be executed in parallel
- Work of run-time environment (scheduler) to
  - Decide during execution how to actually divide the work between processors
- Work Stealing Scheduler
  - Proved to be good scheduler
  - Now also in GCC, Intel CC, **Intel acquire Cilk++**

# Fibonacci

```
int fib (int n) {
if (n<2) return (n);
  else {
    int x,y;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
  }
}
```

*C elision*

*Cilk code*

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

Cilk is a ***faithful*** extension of C. A Cilk program's ***serial elision*** is always a legal implementation of Cilk semantics. Cilk provides *no* new data types.

# Basic Cilk Keywords

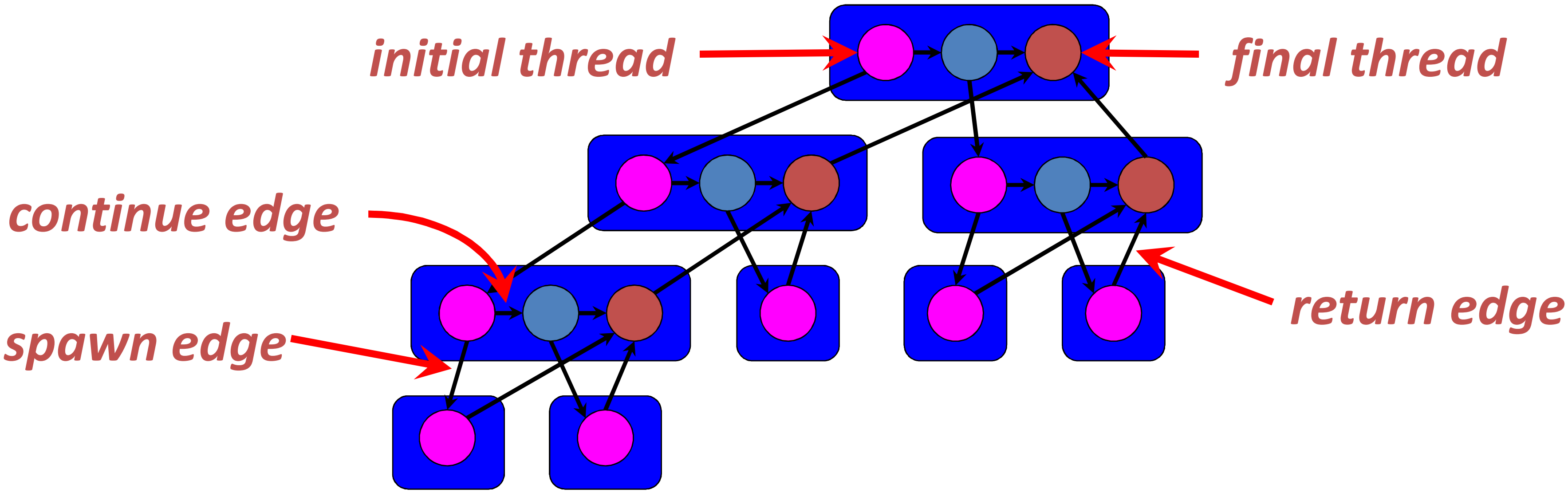
```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

Identifies a function as a ***Cilk procedure***, capable of being spawned in parallel.

The named ***child*** Cilk procedure can execute in parallel with the ***parent*** caller.

Control cannot pass this point until all spawned children have returned.

# Multithreaded Computation




- The dag $G = (V, E)$ represents a parallel instruction stream.
- Each vertex $v$ 2 $V$ represents a ***(Cilk) thread***: a maximal sequence of instructions not containing parallel control (**`spawn`**, **`sync`**, **`return`**).
- Every edge $e$ 2 $E$ is either a ***spawn*** edge, a ***return*** edge, or a ***continue*** edge.

# Fib: Cilk++ Version

```
int fib(int n) {
    if (n < 2) return n;
    int x=cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}
```