



High Performance Computing

**Mr. Rahul Sharma,
Assistant Professor
CSE department, PIT, PU**



Module Topics

Synchronization: Scheduling, Job Allocation, Job Partitioning, Dependency Analysis Mapping Parallel Algorithms onto Parallel Architectures, Thread Basics, The POSIX Thread API, Thread Basics: Creation and Termination, Synchronization Primitives in Pthreads, Controlling Thread and Synchronization Attributes, Thread Cancellation, Composite Synchronization Constructs, Tips for Designing Asynchronous Programs, OpenMP: a Standard for Directive Based Parallel Programming

Parallel Processing Fundamental Design Issues

- Synchronization
- Scheduling
- Job allocation
- Job partitioning
- Dependency analysis
- Mapping parallel algorithm onto Parallel Architecture
- Performance Analysis of Parallel Algorithms

Synchronization

- Parallel program need to manage sequence of work and tasks.
- It is one of factor of program performance.
- It require “serialization” of segment of program.
- Types of Synchronization
 - Barrier
 - Lock/Semaphore
 - Synchronous communication operation

Barrier

- All tasks are involved
- Each task performs its work until it reaches the barrier, after it “blocks”
- When last task reaches the barrier, all tasks are synchronized. Actual work starts from here.
- Often serial section of work must be done.
- In many cases, the tasks are automatically released to continue their work.

Lock/Semaphore

- It involve any number of tasks
- Used to protect access to global data
- Only one task may use lock/semaphore.
- The first task to acquire the lock sets it and can access the protected data.
- Other task wait until task that owns the lock release it.

Synchronous communication operation

- Involve only communicating tasks
- Need of coordination
- Example: before task can perform a send operation, it must first receive an acknowledgement from receiving task that it is ok to send.

Job scheduling

- In parallel system, system selects order of jobs execution to minimize turnaround time.
- Job is mapped to subset of processor.
- Partition of machine: set of processor dedicated to certain job
- User submit job to scheduler, job are queued and are considered for allocation.
- The main aim is to increase processor utilization but because of lack of knowledge regarding future job and execution time, it sometime fails.
- Scheduler allocate compute nodes and other resources to job by matching requirement of job to compute nodes according to data provided by resource manager.

Job scheduling policies

- FCFS:
 - allocate job according to order of their arrival.
 - in addition to no of nodes and no of tasks per node, the job specifies computational processor power, disk storage, memory space, system architecture, high speed network resources.
- Look-ahead optimizing scheduler
- Gang scheduling

Dependency analysis

- Dependence: two computations that places constraints on their execution order
- Dependency analysis: to identify these constraints
- Dependency of 2 type:Control and data dependency

Dependency analysis

- Data dependency:
 - Two memory accesses are involved
 - if data may refer to same memory location and one of the reference is a write.
 - It can either between two distinct program statement or two different dynamic execution of same program statement.
 - Ensure data is produced and consumed in right order

Dependency analysis

- Data dependency of following type:
- FLOW DEPENDENCY:
 - One instruction depends on the final outcome of previous instruction.
 - Also known as write or write-read dependency.
 - Eg. I1: Add R1, R2
I2: Mov R4,R1

Dependency analysis

- ANTI-DEPENDENCY:
 - One instruction depends on data that could be destroyed by another instruction
 - Eg. I1: $A = B + C$
I2: $C = D + E$
 - It occurs between instruction that reads a register and subsequent instruction writes a new value to same location
 - Two instruction have anti-dependency if swapping their order would result in true dependence.

Dependency analysis

- **OUTPUT DEPENDENCY:**
 - Occurs when two instruction both write a result
 - Also known as write write dependency
 - Exists due to limited no of architectural registers.

Dependency analysis

- I/O DEPENDENCE
 - If both I/O statement try to use same file for their operation
- Data flow dependence can not be removed
- Data anti dependence can be removed
- Output dependence can be removed

Dependency analysis

- Control dependency:
 - If control flow of segments cannot be identified before run time
 - An instruction J control dependent on I if the execution of J is controlled by instruction I
- I: if $a < b$;
- J: $a = a + 1$
- Set of branch can also be controlled
- If I
- { B1
- }
- If J
- {B2
- }

Mapping Parallel Algorithms onto Parallel Architecture

- Challenge for parallel computing:
 - Is to map parallel algorithm to parallel platforms
- Mapping requires
 - Analysis of parallel algorithm, definition of logical configuration of platforms and mapping of algorithm to logical platforms
- Issue of mapping arises when
 - No of processes used by algorithm is greater than no of processing unit present.
- Topological variation is when communication structure of algorithm differs from interconnect structure of parallel machine.

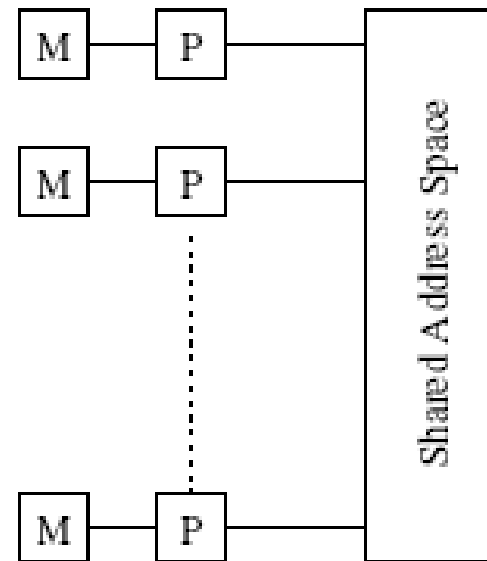
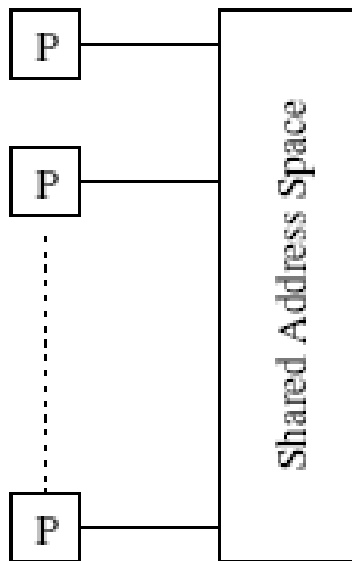
Mapping Parallel Algorithms onto Parallel Architecture

- Cardinality variation occurs when no of processes in the parallel algorithm is greater than no of processing units in parallel machine. But how to map these processes to processing units
- Solution:
 - Parallel algorithm can be represented as graph.
 - assume that bandwidth and length are same for communication path
 - solution to mapping problem is contraction, embedding and multiplexing

Thread Basics

- All memory in the logical machine model of a thread is globally accessible to every thread.
- The stack corresponding to the function call is generally treated as being local to the thread for liveness reasons.
- This implies a logical machine model with both global memory (default) and local memory (stacks).
- It is important to note that such a flat model may result in very poor performance since memory is physically distributed in typical machines.

Thread Basics



- The logical machine model of a thread-based programming paradigm.

Thread Basics

- **Threads provide software portability.**
 - Threaded applications can be developed on serial machines and run on parallel machines without any changes. This ability to migrate programs between diverse architectural platforms is a very significant advantage of threaded APIs.
- **Inherent support for latency hiding.**
 - One of the major overheads in programs (both serial and parallel) is the access latency for memory access, I/O, and communication. By allowing multiple threads to execute on the same processor, threaded APIs enable this latency to be hidden. In effect, while one thread is waiting for a communication operation, other threads can utilize the CPU, thus masking associated overhead.

Thread Basics

- **Scheduling and load balancing.**

–While writing shared address space parallel programs, a programmer must express concurrency in a way that minimizes overheads of remote interaction and idling. While in many structured applications the task of allocating equal work to processors is easily accomplished, in unstructured and dynamic applications (such as game playing and discrete optimization) this task is more difficult. Threaded APIs allow the programmer to specify a large number of concurrent tasks and support system-level dynamic mapping of tasks to processors with a view to minimizing idling overheads. By providing this support at the system level, threaded APIs rid the programmer of the burden of explicit scheduling and load balancing.

- **Ease of programming and widespread use.**

–Due to the aforementioned advantages, threaded programs are significantly easier to write than corresponding programs using message passing APIs. Achieving identical levels of performance for the two programs may require additional effort, however. With widespread acceptance of the POSIX thread API, development tools for POSIX threads are more widely available and stable. These issues are important from the program development and software engineering aspects.

The POSIX Thread API

- Commonly referred to as Pthreads, POSIX has emerged as the standard threads API, supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.

Thread Basics: Creation and Termination

- Pthreads provides two basic functions for specifying concurrency in a program:

```
#include <pthread.h>
```

```
int pthread_create (  
    pthread_t *thread_handle, const pthread_attr_t *attribute,  
    void * (*thread_function)(void *),  
    void *arg);
```

```
int pthread_join (  
    pthread_t thread,  
    void **ptr);
```

- The function pthread_create invokes function thread_function as a thread

Thread Basics: Creation and Termination

```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
....
main() {
    ...
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    pthread_attr_init (&attr);
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    ...
}
```

Thread Basics: Creation and Termination

```
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x = (double) (rand_r(&seed)) / (double) ((2<<14)-1);
        rand_no_y = (double) (rand_r(&seed)) / (double) ((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Synchronization Primitives in Pthreads

- When multiple threads attempt to manipulate the same data item, the results can often be incoherent if proper care is not taken to synchronize them.
- Consider:
 /* each thread tries to update variable best_cost as follows */
 if (my_cost < best_cost)
 best_cost = my_cost;
- Assume that there are two threads, the initial value of best_cost is 100, and the values of my_cost are 50 and 75 at threads t1 and t2.
- Depending on the schedule of the threads, the value of best_cost could be 50 or 75!
- The value 75 does not correspond to any serialization of the threads.

Controlling Thread and Synchronization Attributes

- The Pthreads API allows a programmer to change the default attributes of entities using *attributes objects*.
- An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties.
- Once these properties are set, the attributes object can be passed to the method initializing the entity.
- Enhances modularity, readability, and ease of modification.

Attributes Objects for Threads

- Use `pthread_attr_init` to create an attributes object.
- Individual properties associated with the attributes object can be changed using the following functions:
 - `pthread_attr_setdetachstate,`
 - `pthread_attr_setguardsize_np,`
 - `pthread_attr_setstacksize,`
 - `pthread_attr_setinheritsched,`
 - `pthread_attr_setschedpolicy,` and
 - `pthread_attr_setschedparam`

Attributes Objects for Mutexes

- Initialize the attributes object using function:
`pthread_mutexattr_init.`
- The function `pthread_mutexattr_settype_np` can be used for setting the type of mutex specified by the mutex attributes object.
`pthread_mutexattr_settype_np (`
`pthread_mutexattr_t *attr,`
`int type);`
- Here, type specifies the type of the mutex and can take one of:
 - `PTHREAD_MUTEX_NORMAL_NP`
 - `PTHREAD_MUTEX_RECURSIVE_NP`
 - `PTHREAD_MUTEX_ERRORCHECK_NP`

Composite Synchronization Constructs

- By design, Pthreads provide support for a basic set of operations.
- Higher level constructs can be built using basic synchronization constructs.
- We discuss two such constructs - read-write locks and barriers.

Tips for Designing Asynchronous Programs

- Never rely on scheduling assumptions when exchanging data.
- Never rely on liveness of data resulting from assumptions on scheduling.
- Do not rely on scheduling as a means of synchronization.
- Where possible, define and use group synchronizations and data replication.

OpenMP: a Standard for Directive Based Parallel Programming

- OpenMP is a directive-based API that can be used with FORTRAN, C, and C++ for programming shared address space machines.
- OpenMP directives provide support for concurrency, synchronization, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization.

OpenMP Programming Model

- OpenMP directives in C and C++ are based on the `#pragma` compiler directives.
- A directive consists of a directive name followed by clauses.
`#pragma omp directive [clause list]`
- OpenMP programs execute serially until they encounter the `parallel` directive, which creates a group of threads.
`#pragma omp parallel [clause list]`
`/* structured block */`
- The main thread that encounters the `parallel` directive becomes the *master* of this group of threads and is assigned the thread id 0 within the group.

OpenMP Programming Model

- The clause list is used to specify conditional parallelization, number of threads, and data handling.
 - Conditional Parallelization:** The clause `if` (scalar expression) determines whether the parallel construct results in creation of threads.
 - Degree of Concurrency:** The clause `num_threads`(integer expression) specifies the number of threads that are created.
 - Data Handling:** The clause `private` (variable list) indicates variables local to each thread. The clause `firstprivate` (variable list) is similar to the `private`, except values of variables are initialized to corresponding values before the parallel directive. The clause `shared` (variable list) indicates that variables are shared across all the threads.

OpenMP Programming Model

```
int a, b;
main() {
    [ // serial segment
      #pragma omp parallel num_threads (8) private (a) shared (b)
      { [ // parallel segment
        ]
      [ // rest of serial segment
        ]
    }
```

Sample OpenMP program

Code inserted by the OpenMP compiler

```
int a, b;
main() {
    [ // serial segment
      for (i = 0; i < 8; i++)
        pthread_create (....., internal_thread_fn_name, ...);
      for (i = 0; i < 8; i++)
        pthread_join (.....);
    [ // rest of serial segment
      ]

    void *internal_thread_fn_name (void *packaged_argument) {
        int a;

    [ // parallel segment
      ]
    }
```

Corresponding Pthreads translation

OpenMP Programming Model

- A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

```
#pragma omp parallel if (is_parallel== 1) num_threads(8) \  
    private (a) shared (b) firstprivate(c) {  
    /* structured block */  
}
```

- If the value of the variable `is_parallel` equals one, eight threads are created.
- Each of these threads gets private copies of variables `a` and `c`, and shares a single value of variable `b`.
- The value of each copy of `c` is initialized to the value of `c` before the parallel directive.
- The default state of a variable is specified by the clause `default (shared)` or `default (none)`.

Reduction Clause in OpenMP

- The reduction clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.
- The usage of the reduction clause is reduction (operator: variable list).
- The variables in the list are implicitly specified as being private to threads.
- The operator can be one of +, *, -, &, |, ^, &&, and ||.

```
#pragma omp parallel reduction(+: sum) num_threads(8) {  
/* compute local sums here */  
}
```

```
/*sum here contains sum of all local instances of sums */
```

OpenMP Programming: Example

```
/* *****  
An OpenMP version of a threaded program to compute PI.  
***** */  
#pragma omp parallel default(private) shared (npoints) \  
    reduction(+: sum) num_threads(8)  
{  
    num_threads = omp_get_num_threads();  
    sample_points_per_thread = npoints / num_threads;  
    sum = 0;  
    for (i = 0; i < sample_points_per_thread; i++) {  
        rand_no_x = (double)(rand_r(&seed))/((double)((2<<14)-1));  
        rand_no_y = (double)(rand_r(&seed))/((double)((2<<14)-1));  
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +  
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)  
            sum ++;  
    }  
}
```

Specifying Concurrent Tasks in OpenMP

- The parallel directive can be used in conjunction with other directives to specify concurrency across iterations and tasks.
- OpenMP provides two directives - for and sections - to specify concurrent iterations and tasks.
- The for directive is used to split parallel iteration spaces across threads. The general form of a for directive is as follows:

```
#pragma omp for [clause list]
/* for loop */
```
- The clauses that can be used in this context are: private, firstprivate, lastprivate, reduction, schedule, nowait, and ordered.

Specifying Concurrent Tasks in OpenMP

```
#pragma omp parallel default(private) shared (npoints) \  
    reduction(+: sum) num_threads(8)  
{  
    sum = 0;  
    #pragma omp for  
    for (i = 0; i < npoints; i++) {  
        rand_no_x = (double)(rand_r(&seed))/(double)((2<<14)-1);  
        rand_no_y = (double)(rand_r(&seed))/(double)((2<<14)-1);  
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +  
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)  
            sum ++;  
    }  
}
```

Parul[®]
University

NAAC
GRADE **A++**



<https://paruluniversity.ac.in/>

