MODULE 3: PARALLELISM, COMMUNICATION AND COORDINATION
WEEK 3

Learning Outcomes:

After completing this course you are expected to demonstrate the following:

1. Introduce programming constructs for coordinating multiple simultaneous computations. Explain parallelism, communication and coordination and the need for synchronization.

A. Engage

Trivia

IBM Roadrunner

Dr. Don Grice, chief engineer of the Roadrunner project at IBM, shows off the layout for the supercomputer, which has 296 IBM Blade Center H racks and takes up 6,000 square feet.

(source: http://www.computerworld.com)

B. Explore

Video Title**: Communication Patterns – Intro to Parallel Programming**

YouTube Link: https://youtube.com/watch?v=HPxjX5MIGcO15

Module Video Filename: **Week 3 - Communication Patterns – Intro to Parallel Programming**

C. Explain

Parallel Computing is all about many threads solving a problem by working together. Any books on business practices or teamwork will tell you that working together is really all about communication.

In CUDA, this communication takes place through memory. For example, threads may need to read from the same input location, threads may need to write to the same output location. Sometimes, treads may need to exchange partial results.

Conceptually map and transpose are 1 – to – 1. Each input maps to a single unique output. You can think of a gather operation as many to – 1. Many possible inputs can be chosen to compute an output. In this terminology, scatter is – one – to many, so each threads chooses from many possible output destinations.

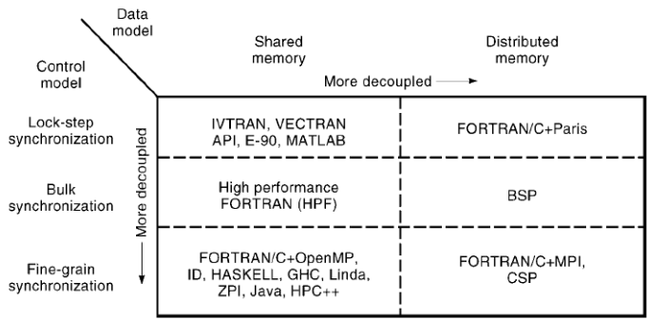
Stencil can be seen as a specialized gather that pulls output from a select few inputs. In a given neighbourhood of the output. So you might turn that to several – to – 1. Reduce could be turned all – to – one. For example, if you are adding up all the numbers in an array, finally scan and sort can be considered all – to – all because all of the input can affect the destination of the resulting output.

D. Elaborate

A **Parallel Programming Language** is a formal notation for expressing algorithms. The meaning of this notation can be defined by appealing to a parallel computational model. Parallel programming languages have more complicated data and control models than sequential programming languages. The data model in sequential languages is that of the random access machine (RAM) model in which there is a single address space of memory locations that can be read and written by the processor. The analog in parallel languages is the shared-memory model in which all memory locations reside in a single address space and are accessible to all the processors (the word processors in this article always refers to the logical processors in the underlying parallel execution model of the language and not to hardware processors). A more decoupled data model is provided by the distributed-memory model in which each processor has its own address space of memory locations inaccessible to other processors.

This form of parallel execution is sometimes called multiple-instruction–multiple-data (MIMD) parallelism. The MIMD model is appropriate for exploiting task parallelism, which arises when autonomous computations (tasks) can execute concurrently, synchronizing only for exclusive access to resources or for coordinating access to data that is being produced and consumed concurrently by different tasks. Table 3.1 classifies the languages discussed in this article according to their control and data models.

Table 3.1 Classification of Parallel Programming Languages



This covers the types of parallelism found in Functional, Lisp and Object-Oriented languages. In particular, it concentrates on the addition of high level parallel constructs to these types of languages.

The three particular programming formalisms were picked because most of the initial ideas seem to have been generated by the designers of the functional languages and most of the current activity seems to be in the Lisp and Objected-Oriented domains. There is also a great deal of activity in the Logic programming area, but this activity is more in the area of executing the existing constructs in parallel as opposed to adding constructs specifically designed to increase parallelism.

Parallelism in Functional Programming Languages

The basic concept behind functional languages is quite simple: To allow a language to consist only of expressions and function applications and to disallow all forms of assignment and side effects. Single assignment actually represents no more than a notational convenience for the programmer.

Table 3.2 Functional Languages and their Parallel Constructs

	ForAll	Reduct	Pattern	Lazy	Eager	DistData
1) FP	X	X				
2) SASL			X	X		
3) VAL	X	X				X
4) HOPE	X	X	X	X		
5) SISAL	X	X				
6) SAL	X	X			X	
7) ML	X	X	X			
8) FLucid	X	X				X

To specify the particular order in which the reduction is to be done. The choice is generally between left, right and tree order which may be performed in log as opposed to linear time on a parallel machine with tree connections. Some languages limit the operators that maybe used for reduction to a predefined set of associative operators.

Pattern matching is a method of binding arguments to functions that allows for more parallelism and better compilation than the traditional method of passing arguments and then dispatching on their value within the function using either a case or an if statement For example the function toreverse a list would be written as:

reverse (nil) - *nil*
reverse (a:l) - reverse(l) : [aJ

using pattern matching (: is the cons operator) as opposed to its usual form of:

reverse(l) =
if null(l) then nil
else (reverse (cdr 1) : (aJ)

By delaying the evaluation of the car and cdr pans of the cons until they are actually needed, a great deal of computation may be saved. In addition such "lazy" evaluation allows the definition of infinite data structures since only the pans that are actually accessed will ever be created.
For example:
zeros = 0 : zeros

So, for example, in a language such as SASL where no ordering is specified the binding may be carried out in parallel, whereas this is not the case when some specific ordering dependencies are introduced. Perhaps the most interesting addition to the SASL language is the use of Zermelo-Frankel set abstraction to express list formation. Syntactically, ZF-expression has the form:

(expression; qualifier; ... ;qualifier]

A qualifier is either a Boolean expression or a generator. A generator has the form:

<name><- <list expression>

<name> is a local variable whose scope is that of the expression. So, for example, a map function might be defined as: **map f x - [f a: a<-x]**

Where a takes successive values from the list x. Another example is the definition of a function that generates the permutations of a list which uses both ZF-notation and pattern matching:

```
perms [] = [[]]
perms x = [a:p: a<-x; p<-perms(x--[a])]
```

(The **-- operator** represents list difference) ZF-notation presents interesting possibilities for parallel compilation.

Discussion

From the above language descriptions it should be clear that the basic parallel constructs that have been used in functional languages do not vary greatly from one language to another except in syntax.

Parallelism in Lisp

As with functional languages there are two main approaches to executing Lisp in parallel. One is to use existing code and clever compiling methods to parallelize the execution of the code. This approach is very attractive because it allows the use of already existing code without modification. It relieves the programmer from the significant conceptual overhead of parallelism. This approach, known as the "dusty deck" approach suffers from a simple problem: it is very hard to do.

Parallelism in Object-Oriented Programming Languages

A number of definitions have been given for Object-Oriented programming. Rather than attempting yet another, we set forth the minimal characteristics that an Object-Oriented programming language must exhibit in order to be true to its name. The primary characteristic of an Object-Oriented programming language is that of encapsulation or information hiding. An object provides a framework by which the programmer may keep certain pans of his implementation private, presenting only a certain interface to the user. Yet there are a number of languages that provide such information hiding, but cannot be considered true Object-Oriented programming languages. Examples of such languages are Ada, Modula, and CLU.

Parallelism

The fourth dimension along which Object-Oriented languages vary is the way they deal with concurrent computation. Again, as with the characteristics discussed above, a number of points along this dimension are occupied. There are systems which allow for no parallelism within their model, systems that provide only the most primitive facilities and those in which parallelism is an integral pan of their specification. Smalltalk, for example, provides two classes.

On processes and threads: synchronization and communication In parallel programs

In all our programs, there are some execution phases in which our processes are taking sometime in an I/O operation or in a synchronization operation waiting for the system or another process to finish a task. During these phases, our programs are not progressing on

their work, it is rather the operating system or some other process running on the computer the ones which are taking all the valuable CPU time.

Processes and threads

In traditional operating systems, the concept of process is described as the instance of an executing program. In these, a process can only contain a running program where there is a single execution flow, that is, the program will be executing a single instruction at anytime.

Communication and synchronization between processes or threads

In any operating system, processes compete for accessing shared resources or they co-operate within a single application to communicate information to each other. Both situations are managed by the operating system by using synchronization mechanisms that allow exclusive access to shared resources and communication elements in a coordinated way. In multi threading operating systems, the address space within any single process is shared between its threads, and therefore communication is performed by means of data structures within this shared memory space.

Barriers

Barriers are synchronization elements used in parallel programming to synchronize a finiteset of n processes between them. A barrier will stop the first processes arriving to thesynchronization primitive until all the n processes have executed the call. The genericsynchronization primitive is executed as follows:

structbarrier_t bar; int n; ... barrier (bar, n) ...

Where bar is the name of the barrier that we are applying and n is the number of processes that want to synchronize with it. The necessary condition continues the execution of a process executing the primitive barrier is that n processes have called this function.

Synchronization with message passing primitives

In message passing, it is important to understand how the implicit synchronization behaves when two processes are communicating. The primitive receive will block the process if the communication channel is empty of messages, and otherwise the first message will be returned and the process will continue its execution.

Unlimited capacity: this is the ideal communication channel. The sending process will never, under any condition, be blocked.

Limited capacity: the sender-process will be blocked only when the communication channel is full.

Null capacity: this possibility makes sense only in direct communication. As no message can be stored, the two processes have to synchronize to each other every time they want to communicate: the first of them that is ready for sending or receiving is blocked until the other is ready for the opposite operation.

The producer-consumer scheme using message passing

```
producer() {
int element;
char message[50];
while (1) {
produce_element(&element);
compose_message (&message, element);
send(MBX, message);
}
}
consumer() {
int element;
char message[50];
while (1) {
receive(MBX, &message);
decompose_message (message, &element);
consume_element(element);
}
}
```

Two Parallel Programming Models Concepts

1. SPMD
2. OpenMP

Deriving a Synchronization-Free Affine Partitioning

1. Setting up the space partition constraints (keep iterations involved in a dependence on the same processor)
2. Solve the sparse partition constraints (linear algebra)
3. Eliminate empty iterations (Fourier-Motzkin)
4. Eliminate tests from inner loop (more Fourier-Motzkin)
5. Using the above to derive primitive affine transformations

SPMD

1. Single program multiple data program should check what processor it is running on and execute some subset of the iterations based on that

```
MPI_Init(&Argc,&Argv);!
// p is the processor id!
MPI_Comm_rank(MPI_COMM_WORLD,&p);!
```

OpenMP

1. shared memory, thread-based parallelism
2. pragmas indicate that a loop is fully parallel

```
#pragma omp for!
for (i=0; i<N; i++) {!
}!
```

E. Evaluate

ASSESSMENT:

Instruction: Answer the questions below using the Answer Sheet (AS) provided in this module.

CONTENT FOR ASSESSMENT:
(2-points each)

1. It is all about many threads solving a problem by working together. Any books on business practices or teamwork will tell you that working together is really all about communication.
2. A formal notation for expressing algorithms. The meaning of this notation can be defined by appealing to a parallel computational model.
3. A local variable whose scope is that of the expression.
4. Synchronization elements used in parallel programming to synchronize a finite set of n processes between them.
5. The sender-process will be blocked only when the communication channel is full.

References:

1. ***PARALLEL AND VECTOR PROGRAMMING LANGUAGES****, KESHAV PINGALI Cornell University Ithaca, New York*

2. ***A Survey of Parallel Programming Constructs,*** *Michael van Biema, Columbia University, Dept. of Computer Science, New York, N.Y. 10027, Tel: (212)2802736, MICHAEL@CS.COLUMBIA.EDU*

Facilitated By:		
Name	:	
MS Teams Account (email)	:	
Smart Phone Number	:	