An Overview of Parallel Computing

Rahman Tashakkori

and

Darren Greene

January 20, 2007
Department of Computer Science
Appalachian State University

A Consortium to Promote Computational Science and High Performance Computing

Most of the material and figures on this presentation has come from:

Reference - Lawrence Livermore National Lab, Parallel Computing Tutorial
http://www.llnl.gov/computing/tutorials/parallel_comp/
Basics of Parallel Programming

Parallel programming requires some understanding of parallel memory architectures and programming models.

It also requires some skills in designing parallel programs.

It is often deals with converting existing serial programs to their parallelized version.

Why Parallel Programming?

• Save time – do computations faster
• Complex problems - Solve larger problems
• Provide concurrency – Do multiple things concurrently
• Taking advantage of non-local resources - using available compute resources on a wide area network, or even the Internet when limited local compute resources are available.
• Cost savings - using multiple less expensive compute resources instead of paying for time on a supercomputer.
• Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

Note: Cost of memory goes up very fast with the size
Traditional Computation

- Programs are written for *serial* computation
- These programs run on a single Central Processing Unit (CPU) on a single computer.
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.

*Why don’t we build a very very very fast single CPU computer to perform computations faster?*

*Limitations: Transmission speed, hard to make it smaller, and not economical.*

---

Traditional Computation

- Both program and data instructions are stored in the memory
- Program instructions are coded data telling the computer to do something
- Data is the information to be used by the program
- A central processing unit (CPU) gets instructions and/or data from the memory, decodes the instructions, and then *sequentially* executes them.
What is the simplest way of achieving parallelization for the above task?

- Give many of the same computations to many CPU’s
- Give the smaller instructions (t1, t2, …) to different CPUs

Simple Parallel Computation

- Simultaneously use multiple compute resources to solve a computational problem.
- Run using multiple CPUs
- Break the problem into discrete parts that can be solved concurrently on different CPUs
- Further break down the smaller instructions
- Execute instructions from each part simultaneously on different CPUs

Question – Can all problems be parallelized?
Question – Can all problems be solved on a parallel machine?
One Possibility of Parallel Computation

- A single computer with multiple processors;
- An arbitrary number of computers connected by a network;
- A combination of both.

Parallel Computing Compute Resources

One of our 8-node clusters (UNCG Cluster)
Parallel computing is possible or practical when:

- The problem in hand can be broken into discrete pieces that can be solved simultaneously
- The compute resource allows execution of multiple program instructions at any moment in time
- The problem is such that it can be solved in less time with multiple compute resources than with a single compute resource.
Architectures

Parallelization can be done with respect to two independent dimensions: *Instruction* and *Data*.

So we may have:

- **S I S D** - Single Instruction, Single Data
- **S I M D** Single Instruction, Multiple Data
- **M I S D** Multiple Instruction, Single Data
- **M I M D** Multiple Instruction, Multiple Data

---

**SISD**

- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most prevalent form of computer architecture

Examples: most PCs, single CPU workstations and mainframes
**SIMD**

- A type of parallel computer
- Single instruction: All processing units execute the same instruction at any given clock cycle
- Multiple data: Each processing unit can operate on a different data element
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines

Examples: Processor Arrays: Connection Machine CM-2, Maspar MP-1, MP-2, and Vector Pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820
MISD

• A single data stream is fed into multiple processing units.
• Each processing unit operates on the data independently via independent instruction streams.
• Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).

Some conceivable uses might be:
• multiple frequency filters operating on a single signal stream
• multiple cryptography algorithms attempting to crack a single coded message.
MIMD

• Currently, the most common type of parallel computer. Most modern computers fall into this category.
• Multiple Instruction: every processor may be executing a different instruction stream
• Multiple Data: every processor may be working with a different data stream
• Execution can be synchronous or asynchronous, deterministic or non-deterministic
• Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

MIMD

<table>
<thead>
<tr>
<th>prev instruct</th>
<th>load A(1)</th>
<th>load B(1)</th>
<th>C(1)=A(1)*B(1)</th>
<th>store C(1)</th>
<th>next instruct</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prev instruct</th>
<th>call funcD</th>
<th>x=y^2</th>
<th>sum=x^2</th>
<th>call sub1(i,j)</th>
<th>next instruct</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prev instruct</th>
<th>do 10 i=1,N</th>
<th>alpha=w^3</th>
<th>zeta=C(i)</th>
<th>10 continue</th>
<th>next instruct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some Definitions

Task - A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.

Parallel Task - A task that can be executed by multiple processors safely (yields correct results).

Serial Execution - Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, virtually all parallel tasks will have sections of a parallel program that must be executed serially.

Parallel Execution - Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.
Some Definitions

**Shared Memory** - From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

Non-shared model - Flow of file data from server to client.

![Non-shared model diagram]

**Importance of Shared Memory in Client Server Model**

In a non-shared model:

- The client reads the data from the IPC channel, normally requiring the data to be copied from the kernel to the process.

- Finally, the data is copied from the client’s buffer, the second argument to the write function, to the output file.

A total of four copies of the data are normally required. Additionally, these four copies are done between the kernel and a process, often an expensive copy (more expensive than copying data within the kernel, or copying data within a single process). The figure shown on the previous page depicts this movement of the data between the client and server, through the kernel.
Some Definitions

Distributed Memory - In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

Copying file data from server to client using shared memory.

• The server gets access to a shared memory object using (say) a semaphore.
• The server reads from the input file into the shared memory object. The second argument to the read, the address of the data buffer, points into the shared memory object.
• When the read is complete, the server notifies the client, using a semaphore.
• The client writes the data from the shared memory object to the output file.

As the figure shows the data is copied only twice—from the input file into shared memory and from shared memory to the output file. The dashed box encloses the client and the shared memory object, and another dashed box enclosing the server and the shared memory object, to reinforce that the shared memory object appears in the address space of both the client and the server.
Communications - Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.

Synchronization - The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point. Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.

Demo both.

Some Definitions

Granularity - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

- **Coarse:** relatively large amounts of computational work are done between communication events
- **Fine:** relatively small amounts of computational work are done between communication events. Demo.

Speedup - Speedup of a code which has been parallelized, defined as: wall clock of serial execution – wall clock of parallel execution

Parallel Overhead - The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:

1) Task start-up time   2) Synchronizations   3) Data communications Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.   4) Task termination time
**Some Definitions**

**Massively Parallel** - Refers to the hardware that comprises a given parallel system - having many processors. The meaning of many keeps increasing, but currently we seem to be talking about 6 digits.

**Scalability** - Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:

- Hardware - particularly memory-cpu bandwidths and network communications
- Application algorithm
- Parallel overhead related
- Characteristics of your specific application and coding

---

**Common Parallel Computer Memory Architectures**

**Shared Memory Architecture**

**Distributed Memory Architecture**
Parallel Programming Models

Parallel programming models exist as an abstraction above hardware and memory architectures. There are several parallel programming models in common use:

- Shared Memory
- Threads
- Message Passing
- Data Parallel
- Hybrid
**Message Passing Model**

- A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well as across an arbitrary number of machines.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

![Diagram of message passing](image)

**Implementation of Message Passing Model**

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are imbedded in source code. The programmer is responsible for determining all parallelism.

- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.

- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.
Implementation of Message Passing Model

- Part 1 of the Message Passing Interface (MPI) was released in 1994. Part 2 (MPI-2) was released in 1996. Both MPI specifications are available on the web at:


- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI. A few offer a full implementation of MPI-2.

- For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons.

Data Parallel Model

- Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
- A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
- Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".
On shared memory architectures, all tasks may have access to the data structure through global memory. On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task.

**Implementation of Data Parallelization**

- Programming with the data parallel model is usually accomplished by writing a program with data parallel constructs.

- The constructs can be calls to a data parallel subroutine library or, compiler directives recognized by a data parallel compiler.

  - **Fortran 90 and 95 (F90, F95)**
  - **High Performance Fortran (HPF)**
Understanding Parallel Programming

Whether the intention of your work is to parallelize an existing program or to start a new parallel program, you have to make sure that the problem has a solution that is parallelizable.

Non-parallelizable

Calculation of the Fibonacci series (1, 1, 2, 3, 5, 8, 13, 21, ...) using the formula:

\[ F(k + 2) = F(k + 1) + F(k) \]

Why this is not parallelizable?
Identify the program's **hotspots**:
- Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
- Profilers and performance analysis tools can help here.
- **Focus** on parallelizing the hotspots and **ignore** those sections of the program that account for little CPU usage.

Identify **bottlenecks** in the program
- Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
- It possible, restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas.
Understanding Parallel Programming

• Identify inhibitors to parallelism. One common class of inhibitor is *data dependence*, as demonstrated by the Fibonacci sequence above.

• Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.

Partitioning for Parallel Programming

• One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning.

• There are two basic ways to partition computational work among parallel tasks: *Domain Decomposition* (Dealing with Data) and *Functional Decomposition* (Dealing with the problem).
Partitioning for Parallel Programming

**Domain Decomposition**
In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.

There are different ways to partition data by block or by cycle. Example: images in image processing.

**Functional Decomposition**
In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.

Important things to consider
There are a number of important factors to consider when designing your program's inter-task communications:

- Cost of communications
- Latency vs. Bandwidth
- Visibility of communications
- Synchronous vs. asynchronous communications
- Scope of communications
- Efficiency of communications
- Overhead and Complexity
Synchronization

There are a number of important factors to consider when designing your program's inter-task communications:

- Barrier
- Lock / semaphore
- Synchronous communication operations

Designing Parallel Programs

We have to pay close attention to the following.

- A *dependence* exists between program statements when the order of statement execution affects the results of the program.

- A *data dependence* results from multiple use of the same location(s) in storage by different tasks.

- Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.
**Designing Parallel Programs**

- **Loop carried data dependence**

  ```
  DO 500 J = MYSTART,MYEND
  A(J) = A(J-1) * 2.0
  500 CONTINUE
  ```

  The value of A(J-1) must be computed before the value of A(J), therefore A(J) exhibits a data dependency on A(J-1). Parallelism is inhibited.

- **Loop independent data dependence**

<table>
<thead>
<tr>
<th>task 1</th>
<th>task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>X = 2</td>
<td>X = 4</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
</tr>
<tr>
<td>Y = X**2</td>
<td>Y = X**3</td>
</tr>
</tbody>
</table>

  Parallelism is inhibited. Why?

---

Although all data dependencies are important to identify when designing parallel programs, loop carried dependencies are particularly important since loops are possibly the most common target of parallelization efforts.
**How to Handle Data Dependencies:**

- Distributed memory architectures - communicate required data at synchronization points.
- Shared memory architectures - synchronize read/write operations between tasks.

**Load Balancing:**

- Load balancing refers to the practice of distributing work among tasks so that all tasks are kept busy all of the time. It can be considered a minimization of task idle time.

- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.
Designing Parallel Programs

Load Balancing:

• Equally partition the work each task receives
• Use dynamic work assignment

Designing Parallel Programs

Granularity:

• Computation / Communication Ratio
• Fine-grain Parallelism
• Coarse-grain Parallelism
• Best practices …
**Fine-grain Parallelism:**

- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.

**Coarse-grain Parallelism**

- Relatively large amounts of computational work are done between communication/synchronization events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently
Best granularity Practices

- The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
- In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
- Fine-grain parallelism can help reduce overheads due to load imbalance.

Limits and Costs of Parallel Programming

Amdahl’s Law: Potential program speedup is defined by the fraction of code (P) that can be parallelized:

\[ speedup = \frac{1}{1-P} \]

Modified version includes the number of processors as:

\[ speedup = \frac{1}{ \frac{P}{N} + S } \]

where \( P \) = parallel fraction, \( N \) = number of processors and \( S \) = serial fraction
Measure of Complexity of a Program

The costs of complexity are measured in programmer’s time in virtually every aspect of the software development cycle:

- Design
- Coding
- Debugging
- Tuning
- Maintenance

Example

Improving the intensity of an image by multiplying all pixels with a constant larger than one.

```java
for (int i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        f(i, j) = conts * f(i, j);
```

The computation for each pixel is independent of others. This computation must be computationally intensive to make the parallelization worth the efforts.
One possible parallelization:

```cpp
for (int i = start1; i < end1; i++)
    for (j = 0; j < M; j++)
        f(i, j) = conts * f(i, j);
}
}
for (int i = start2; i < end2; i++)
    for (j = 0; j < M; j++)
        f(i, j) = conts * f(i, j);
}
}
```

Some Quick and Valuable Resources:

- Dr. Li’s lecture material for Parallel Computing course (next section)
- The Lawrence Livermore Parallel Computing Tutorial
- Barry Wilkinson Parallel Computing Book