Parallel Programming Concepts

Feng Chen
HPC User Services
LSU HPC & LONI
sys-help@loni.org

Louisiana State University
Baton Rouge
May 29, 2017
What we’ll be doing

- Learning to write C/Fortran programs that are explicitly parallel.
  - Parallel Programming Concepts
  - OpenMP
  - OpenACC
  - Understanding Parallel Applications
Outline

- Why parallel programming?
  - Some background
- Parallel hardware and software
  - Parallel computing architecture
  - Parallel programming models
- Performance evaluation
  - Speedup, efficiency
  - Amdahl’s law
  - Gustafson’s law
  - Scalability

- Lab session
  - Compile and run OpenMP/MPI codes
  - Speedup and efficiency
  - Load balancing
Parallel Programming Concepts

Why Parallel Programming?
Applications

Climate Modeling

Protein folding

Energy Research

Data Analysis

Drug Discovery
Some background
Serial hardware and software

Computer runs one program at a time.

Can we have something that just run 100x faster?
The von Neumann Architecture

Central Processing Unit

Control Unit

Arithmetic Logic Unit

Main Memory (code/data) Becomes Larger

Input Device

Output Device
von Neumann bottleneck
The von Neumann Architecture
Cache - Principle of locality

- A CPU cache is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory.

- A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations.

- Accessing one location is followed by an access of a nearby location.
  - **Spatial locality** – accessing a nearby location.
  - **Temporal locality** – accessing in the near future.
Changing Times

- From 1986 - 2002, microprocessors were speeding like a rocket, increasing in performance an average of 50% per year.
- Since then, it’s dropped to about 20% increase per year.

History of Processor Performance

Limitation:
- 2 GHz Consumer
- 4 GHz Server

Source:
http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/
A Little Physics Problem

- Smaller transistors = faster processors.
- Faster processors = increased power consumption.
- Increased power consumption = increased heat.
- Increased heat = unreliable processors.

Solution:
- Move away from single-core systems to multicore processors.
- “core” = central processing unit (CPU)
- Introducing parallelism
  - What if your problem is also not CPU dominant?
The von Neumann Architecture

Around year 2005

CPU

core

cache

CC-interconnect

Cache

Main Memory (code/data)

I/O

I/O
Cache coherence

- Programmers have no control over caches and when they get updated.

A shared memory system with two cores and two caches
An intelligent solution

Instead of designing and building faster microprocessors, put multiple processors on a single integrated circuit.
Now it’s up to the programmers

- Up to now, performance increases have been attributable to increasing density of transistors.
  - But there are inherent problems...

- Adding more processors doesn’t help much if programmers aren’t aware of them...
  - ... or don’t know how to use them.

- Serial programs don’t benefit from this approach (in most cases).
Why we need to write parallel programs

- Running multiple instances of a serial program often isn’t very useful.
  - Exception: Many embarrassingly parallel problems
    - Solution: E.g. GNU Parallel
- Think of running multiple instances of your favorite game.
  - Run VR game twice?
  - What you really want is for it to run faster.
Approaches From Serial To Parallel

- Write translation programs that automatically convert serial programs into parallel programs.
  - This is very difficult to do.
  - Success has been limited.

- **Rewrite serial programs so that they’re parallel.**
  - Some coding constructs can be recognized by an automatic program generator, and converted to a parallel construct.
  - However, it’s likely that the result will be a very inefficient program.
  - Sometimes the best parallel solution is to step back and devise an entirely new algorithm.
How Do We Write Parallel Programs?

- **Task parallelism**
  - Partition various tasks carried out solving the problem among the cores.

- **Data parallelism**
  - Partition the data used in solving the problem among the cores.
  - Each core carries out similar operations on it’s part of the data.
Professor P

15 questions
300 exams
Professor P’s grading assistants
Division Of Work

Data Parallelism

- TA#1: 100 exams
- TA#2: 100 exams
- TA#3: 100 exams

Task Parallelism

- TA#1: Questions 1 - 5
- TA#2: Questions 6 - 10
- TA#3: Questions 11 - 15
Coordination

- **Cores usually need to coordinate their work.**
  - **Communication** – one or more cores send their current partial sums to another core.
  - **Synchronization** – because each core works at its own pace, make sure cores do not get too far ahead of the rest.
  - **Load balancing** – share the work evenly among the cores so that one is not heavily loaded.
Terminology

- **Concurrent computing** – a program is one in which multiple tasks can be in progress at any instant.
- **Parallel computing** – a program is one in which multiple tasks cooperate closely to solve a problem
- **Distributed computing** – a program may need to cooperate with other programs to solve a problem.
Concluding Remarks

- The laws of physics have brought us to the doorstep of multicore technology.
- Serial programs typically don’t benefit from multiple cores.
- Automatic parallel program generation from serial program code isn’t the most efficient approach to get high performance from multicore computers.
- Parallel programs are usually very complex and therefore, require sound program techniques and development.
  - Task Parallelism vs Data Parallelism
  - Learning to write parallel programs involves learning how to coordinate the cores.
    - Communication/Load balancing/Synchronization
Parallel Hardware
Parallel Hardware

- A programmer can write code to exploit.
Flynn’s Taxonomy on Computer Architectures

Single Instruction

Multiple Instruction

SISD

SIMD

MISD

MIMD

Single Data

Multiple Data
Flynn’s Taxonomy of Parallelism

<table>
<thead>
<tr>
<th></th>
<th>SISD</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>MISD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIMD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Standard Multi-Core Model

Standard GPU processing model

classic von Neumann: Standard single core CPU

Not commonly found in practice
SIMD

- Parallelism achieved by dividing data among the processors.
- Applies the same instruction to multiple data items.
- Called data parallelism.
SIMD example

Graphics Processing Units (GPU)

- A graphics processing pipeline converts the internal representation into an array of pixels that can be sent to a computer screen.
- Real time graphics application programming interfaces or API’s use points, lines, and triangles to internally represent the surface of an object.
- GPU’s can often optimize performance by using SIMD parallelism.
  - The current generation of GPU’s use SIMD parallelism, although they are not pure SIMD systems.
MIMD

- In computing, MIMD (multiple instruction, multiple data) is a technique employed to achieve parallelism.

- Machines using MIMD have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data.
  - Supports multiple simultaneous instruction streams operating on multiple data streams. Each of the processors in an SMP has its own instruction decoding unit, so they can all carry out different instructions simultaneously (Multiple Instruction).
  - All processors in an SMP address the same memory banks, but there is no architectural requirement that they have to coordinate which elements they are working on within that memory, so processors can simultaneously manipulate different data elements (Multiple Data).
Type Of Parallel **Hardware** Systems

- **Shared-memory**
  - The cores can share access to the computer’s memory.
  - Coordinate the cores by having them examine and update shared memory locations.

- **Distributed-memory**
  - Each computation unit has its own, private memory.
  - The computation units must communicate explicitly by sending messages across a network.
Shared Memory System

- A collection of autonomous processors is connected to a memory system via an interconnection network.
- Each processor can access each memory location.
- The processors usually communicate implicitly by accessing shared data structures.
- Most widely available shared memory systems use one or more multicore processors.
  - (multiple CPU’s or cores on a single chip)
UMA Multicore System

- UMA (Uniform Memory Access) is commonly represented today by Symmetric Multiprocessor (SMP) machines
  - Identical processors
  - Equal access and access times to memory
  - Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

Time to access all the memory locations will be the same for all the cores.
NUMA multicore system

- NUMA (Non-Uniform Memory Access) often made by physically linking two or more SMPs
  - One SMP can directly access memory of another SMP
  - Not all processors have equal access time to all memories
  - Memory access across link is slower
  - If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

[Diagram showing NUMA architecture with nodes and memory connections]
Distributed Memory System

- **Clusters** (most popular)
  - A collection of commodity systems.
  - Connected by a commodity interconnection network.

- **Nodes** of a cluster are individual computations units joined by a communication network.

Also known as hybrid systems
Cluster And A Cluster Rack

The QB2 cluster

Rack
Infiniband Switch
Compute Node
Inside A Dell C8000 Compute Node

- Accelerator 1 (GPU)
- Accelerator 2 (GPU)
- Storage
- Processor
- Processor
- Network Card
- Memory
Conclusion Remarks

- **Flynn’s taxonomy**
  - SIMD - GPU
  - MIMD - Multicore CPU

- **Shared Memory and Distributed Memory hardware systems**
  - Shared memory nodes:
    - NUMA
  - Distributed memory system:
    - Cluster
Parallel Programming Concepts

Parallel Software
Multitasking

- Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources.
- Each CPU executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish.
Process and Thread

- An operating system “process” is basically an instance of program in execution
  - Components of a process:
    - The executable machine language program.
    - Data
    - Process Control Block (PCB)

- Threads are contained within processes:
  - Thread is the smallest execution unit to which processor allocates time.
  - Components of a thread:
    - Program counter, stack, set of registers, A thread id
Difference Between Process/Threads

- **Process:**
  - A process is an instance of a computer program that is being executed. Each process has a complete set of its own variables.

- **Thread:**
  - A thread of execution results from a fork of a computer program into two or more concurrently running tasks. *In most cases, a thread is contained inside a process.*
  - A thread itself is not a program. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources.

- **The major difference:**
  - Multithreading threads are being executed in one process *sharing common address space* whereas in multi processing different processes have *different address space*. Thus creating multiple processes is costly compare to threads.
A process and threads

A process comprises global variables and threads with local variables and code. Starting a thread is called **forking** and terminating a thread is called **joining**.
The burden is on software

- Hardware and compilers can keep up the pace needed.
- From now on...
  - In shared memory programs:
    - Start a single process and fork threads.
    - Threads carry out tasks.
  - In distributed memory programs:
    - Start multiple processes.
    - Processes carry out tasks.
SPMD – single program multiple data

- A SPMD program consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.
  - Both data and task parallelism

```c
if (I'm thread or process i)
do this;
else
do that;
```
Parallel Programming Models

- **Shared memory programming model**
  - *OpenMP*
  - Pthread
  - **GPU**

- **Distributed memory programming model** (Not covered in this year’s workshop)
  - MPI

- **Hybrid Model** (Not covered in this year’s workshop)
  - Distributed + Shared
    - MPI +
      - OpenMP
      - Pthread
      - GPU
- Distributed Memory Programming is process-based
- Processes running simultaneously communicate by exchanging messages
  - Messages can be 2-sided - both sender and receiver are involved in the process
  - Or they can be 1-sided - only the sender or receiver is involved
- Most message passing programs employ a mixture of data and task parallelism
- Processes cannot access each other’s memory
  - Communication is usually explicit
Message Passing Interface (MPI)

- MPI is a portable library used for writing parallel programs using the message passing model
  - You can expect MPI to be available on any HPC platform you use
- Based on a number of processes running independently in parallel
  - HPC resource provides a command to launch multiple processes simultaneously (e.g. mpirun, aprun)
- There are a number of different implementations but all should support the MPI 2 standard
  - As with different compilers, there will be variations between implantations but all the features specified in the standard should work.
  - Examples: MVAPICH2, OpenMPI, MPICH2
Distributed Memory Prons and Cons

- **Advantages:**
  - Flexible - almost any parallel algorithm imaginable can be implemented
  - Scaling usually only limited by the choice of algorithm
  - Portable - MPI (Message Passing Interface) library is provided on all HPC platforms

- **Disadvantages:**
  - Parallel routines usually become part of the program due to explicit nature of communications
    - Can be a large task to retrofit into existing code
  - May not give optimum performance on shared memory machines
  - Can be difficult to scale to very large number of processes (>100,000) due to overheads
Shard Memory Programming

- Shared memory programming is usually based on threads
  - Although some hardware/software allows processes to be programmed as if they share memory
  - Sometimes known as Symmetric Multi-processing (SMP) although this term is now a little old-fashioned

- Most often used for **Data Parallelism**
  - Each thread operates the same set of instructions on a separate portion of the data

- More difficult to use for **Task Parallelism**
  - Each thread performs a different set of instructions
Shared Memory Concepts

- Threads “communicate” by having access to the same memory space
- Any thread can alter any bit of data
- No explicit communicate between the parallel tasks

\[
\begin{array}{cccc}
  a_0 & a_4 \\
  a_1 & a_5 \\
  a_2 & a_6 \\
  a_3 & a_7 \\
\end{array}
\]
Nondeterminism

```c
printf ( "Thread %d > my_val = %d
", my_rank , my_x ) ;
```

Thread 0 > my_val = 7
Thread 1 > my_val = 19

Thread 1 > my_val = 19
Thread 0 > my_val = 7
Nondeterminism

\[
\begin{align*}
\text{my\_val} &= \text{Compute\_val ( my\_rank )} ; \\
x &=+ \text{ my\_val} ;
\end{align*}
\]

<table>
<thead>
<tr>
<th>Time</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Finish assignment to my_val</td>
<td>In call to Compute_val</td>
</tr>
<tr>
<td>1</td>
<td>Load x = 0 into register</td>
<td>Finish assignment to my_val</td>
</tr>
<tr>
<td>2</td>
<td>Load my_val = 7 into register</td>
<td>Load x = 0 into register</td>
</tr>
<tr>
<td>3</td>
<td>Add my_val = 7 to x</td>
<td>Load my_val = 19 into register</td>
</tr>
<tr>
<td>4</td>
<td>Store x = 7</td>
<td>Add my_val to x</td>
</tr>
<tr>
<td>5</td>
<td>Start other work</td>
<td>Store x = 19</td>
</tr>
</tbody>
</table>
Nondeterminism

- Race condition
- Critical section
- Mutually exclusive
- Mutual exclusion lock (mutex, or simply lock)

```c
my_val = Compute_val ( my_rank ) ;
Lock(&add_my_val_lock ) ;
x += my_val ;
Unlock(&add_my_val_lock ) ;
```
Shared Memory Programming Models

- **Pthread**
  - POSIX Threads

- **OpenMP**
  - (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming

- **Threading Building Blocks**
  - A C++ template library developed by Intel for parallel programming on multi-core processors.

- **Cilk**
  - Cilk, Cilk++ and Cilk Plus are general-purpose programming languages designed for multithreaded parallel computing.

- **OpenCL**
  - OpenCL is an open standard that can be used to program CPUs, GPUs, and other devices from different vendors.

- **CUDA**
  - CUDA is a parallel computing platform and application programming interface (API) model created by NVidia.
Shared Memory Prons and Cons

- **Advantages:**
  - Conceptually simple
  - Usually minor modifications to existing code
  - Often very portable to different architectures

- **Disadvantages**
  - Difficult to implement task-based parallelism, lack of flexibility
  - Often does not scale well
  - Requires a large amount of inherent data parallelism (e.g. large arrays) to be effective
  - Can be surprisingly difficult to get performance
Parallel Programming Concepts

Performance
Take Timing

- In order to determine the run time of a code, we usually need to include calls to a timer function in our source code.
- We are more interested in wall clock time, not CPU time
  - The program may be “active”-for example, waiting for a message-even when the CPU is idle.
- To take parallel times:
  - synchronize the processes/threads before starting the timer,
  - after stopping the timer, find the maximum elapsed time among all the processes/threads.
- Because of system variability, we usually need to run a program several times with a given data set,
  - Usually take the minimum time from the multiple runs.
- To reduce variability and improve overall run-times
  - Usually run no more than one thread per core.
Typical Timing Functions

- **Unix, Linux System Call**
  ```c
  #include <sys/time.h>
  int gettimeofday(struct timeval *tv, struct timezone *tz);
  ```

- **OpenMP**
  ```c
  double omp_get_wtime(void);
  ```

- **MPI**
  ```c
  double MPI_Wtime(void)
  ```

- **Typical code segments:**
  ```c
  double start_time = get_current_time_function();
  // barrier function if needed
  // code segments to be timed
  double time_elapsed = get_current_time_function() - start_time;
  ```

- **In Bash:**
  ```bash
  TIC=$(date +%s.%N);
  # your code to be timed ...
  TOC=$(date +%s.%N);
  DURATION=$(echo "$TOC - $TIC" | bc -l)
  ```
Timing Example

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>

int main(char *argc, char **argv) {
    const long N = 1000000000;
    double sum=0.0;
    long i;
    // openmp directives
    double start_time = omp_get_wtime();
    #pragma omp parallel for reduction(+:sum)
    for (i=0; i<N; i++)
        sum += i*2.0+i/2.0; // doing some floating point operations
    double time = omp_get_wtime() - start_time;
    printf("time= %4.3e\n", time);
    printf("sum= %4.3e\n", sum);
    return 0;
}
Speedup

- Number of cores = \( p \)
- Serial run-time = \( T_{\text{serial}} \)
- Parallel run-time = \( T_{\text{parallel}} \)
- Speedup of a parallel program

\[
S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}
\]

- Linear speedup:

\[
T_{\text{parallel}} = \frac{T_{\text{serial}}}{p}
\]

Not the time from running a parallel code using one thread/process
Efficiency Of A Parallel Program

- Definition of efficiency:

\[
E = \frac{S}{p} = \left( \frac{T_{\text{serial}}}{T_{\text{parallel}}} \right) = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}
\]
Speedups And Efficiencies Of A Parallel Program - Results

- Example Parallel Program

<table>
<thead>
<tr>
<th>p</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>2.74</td>
<td>1.38</td>
<td>0.70</td>
<td>0.37</td>
<td>0.21</td>
</tr>
<tr>
<td>S</td>
<td>1.00</td>
<td>1.99</td>
<td>3.93</td>
<td>7.50</td>
<td>12.91</td>
</tr>
<tr>
<td>E=S/p</td>
<td>1.00</td>
<td>0.99</td>
<td>0.98</td>
<td>0.94</td>
<td>0.81</td>
</tr>
</tbody>
</table>
# Speedups And Efficiencies On Different Problem Sizes

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=1.0E+09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>2.74</td>
<td>1.38</td>
<td>0.70</td>
<td>0.37</td>
<td>0.21</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1.00</td>
<td>1.99</td>
<td>3.93</td>
<td>7.50</td>
<td>12.91</td>
<td></td>
</tr>
<tr>
<td>E=S/p</td>
<td>1.00</td>
<td>0.99</td>
<td>0.98</td>
<td>0.94</td>
<td>0.81</td>
<td></td>
</tr>
<tr>
<td><strong>Double</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=2.0E+09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>5.47</td>
<td>2.74</td>
<td>1.39</td>
<td>0.72</td>
<td>0.39</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1.00</td>
<td>1.99</td>
<td>3.94</td>
<td>7.61</td>
<td>13.99</td>
<td></td>
</tr>
<tr>
<td>E=S/p</td>
<td>1.00</td>
<td>1.00</td>
<td>0.98</td>
<td>0.95</td>
<td>0.87</td>
<td></td>
</tr>
<tr>
<td><strong>Half</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=5.0E+08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>1.37</td>
<td>0.69</td>
<td>0.36</td>
<td>0.19</td>
<td>0.11</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1.00</td>
<td>1.97</td>
<td>3.84</td>
<td>7.10</td>
<td>11.93</td>
<td></td>
</tr>
<tr>
<td>E=S/p</td>
<td>1.00</td>
<td>0.99</td>
<td>0.96</td>
<td>0.89</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td><strong>Quarter</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=2.5E+08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0.69</td>
<td>0.35</td>
<td>0.19</td>
<td>0.10</td>
<td>0.07</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1.00</td>
<td>1.97</td>
<td>3.71</td>
<td>6.78</td>
<td>9.16</td>
<td></td>
</tr>
<tr>
<td>E=S/p</td>
<td>1.00</td>
<td>0.99</td>
<td>0.93</td>
<td>0.85</td>
<td>0.57</td>
<td></td>
</tr>
</tbody>
</table>
Speedup vs Threads for Different Problem Size
(Original Size N=10^8)
Efficiency

Efficiency vs Threads for Different Problem Size

- Original Size
- Double Size
- Half Size
- Quarter Size

Speedup vs Threads for Different Problem Size
Effect of Overhead

- Many parallel programs are developed by dividing the work of the serial program among the processes/threads and adding in the necessary “parallel overhead” such as mutual exclusion or communication.

- $T_{\text{overhead}}$ denotes the parallel overhead:

$$T_{\text{parallel}} = \frac{T_{\text{serial}}}{p} + T_{\text{overhead}}$$

- To increase Speedup and Efficiency with increasing problem size, what kind of condition should be satisfied for $T_{\text{serial}}$ and $T_{\text{overhead}}$?
  - See Exercise 2b)
Scalability of Parallel Systems

- **Amadahl’s Law**
  - The size of the sequential fraction $f$ determines how much speedup is achievable

\[
S \leq \frac{T_{\text{serial}}}{f \cdot T_{\text{serial}} + (1-f) \cdot \frac{T_{\text{serial}}}{p}} = \frac{p}{1 - f (1 - p)} = \frac{1}{f + \frac{(1-f)}{p}}
\]

- One processor

- $p$ processors
Effect of Amdahl’s Law

- Plot of Speedup vs Number of processors at different $f$ (parallel portion)
Amdahl’s Law

- Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.

- This is pretty daunting. Should we give up and go home?
Gustafson's Law

- Addresses the shortcomings of Amdahl's law, which is based on the assumption of a fixed problem size.
  - Apply $p$ processors to a task that has serial fraction $f$, scaling the task to take the same amount of time as before, the speedup is:

$$S = f + (1 - f) \cdot p = p - f \cdot (p - 1)$$
Effect of Gustafson’s Law

- Plot of Speedup vs Number of processors at different f (serial portion)
Gustafson’s Law - Analogy

Commuting Time:

Biological Brains: v.s.
Amdahl vs Gustafson

- Amdahl’s law based on fixed problem size.
- Gustafson’s law based on fixed run time.
Strong and Weak Scaling

- If we can find a corresponding rate of increase in the problem size so that the program always has efficiency $E$, then the program is **scalable**.

  - **Strong Scaling**: Keeping the problem size fixed and pushing in more workers or processors
    - Goal: Minimize time to solution for a given problem

  - **Weak Scaling**: Keeping the work per worker fixed and adding more workers/processors (the overall problem size increases)
    - Goal: solve the larger problems
Scalability Example

- Serial execution: \( T_{\text{serial}} = n \)

- Parallel execution: \( T_{\text{parallel}} = \frac{n}{p} + 1 \)

- Efficiency: \( E = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}} = \frac{n}{p\left(\frac{n}{p} + 1\right)} = \frac{n}{n + p} \)

- To see if the program is scalable, we increase the number of processors by a factor of \( k \), and we want to find the factor \( x \) that we need to increase the problem size by so that \( E \) is unchanged:

\[
E' = \frac{xn}{xn + kp} = E = \frac{n}{n + p}
\]

- Solve the above equation, we have:

\( x = k \)

- Thus the program is scalable (Efficiency will be unchanged)
Parallel Programming Concepts

Designing Parallel Programs
Design a Parallel Program

- **How to design a parallel program**
  - **Partitioning**
    - Divide the work among the processes/threads so that
      - Each process/thread gets roughly the same amount of work;
      - Communication is minimized.
  - **Communication**
    - Arrange for the processes/threads to synchronize.
  - **Synchronization**
    - Arrange for communication among processes/threads
Partitioning

- There are two basic ways to partition computational work among parallel tasks:

  **domain decomposition**

  ![Diagram](diagram1.png)

  - Problem
  - Data Set
  - Task 0
  - Task 1
  - Task 2
  - Task 3

  **1D**
  - Block
  - Cyclic
  - Block, *
  - *, Block
  - Block, Block

  **2D**
  - Block, *
  - *, Block
  - Block, Block
  - Cyclic, *
  - *, Cyclic
  - Cyclic, Cyclic

  **functional decomposition**

  ![Diagram](diagram2.png)

  - Problem Instruction Set
  - Task 0
  - Task 1
  - Task 2
  - Task 3

  ![Diagram](diagram3.png)

  - Atmospheric Model
  - Hydrology Model
  - Ocean Model
  - Land/Surface Model
Load Balancing

- **Load balancing**
  - Distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time.
  - Important to parallel programs for performance reasons.

- **How to Achieve Load Balance:**
  - Equally partition the work each task receives, e.g.:
    - For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
    - For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
  - Use dynamic work assignment
    - It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.
Communications

- Factors to Consider:
  - Cost of communications
  - Latency vs. Bandwidth
    - latency is the time it takes to send a minimal (0 byte) message from point A to point B.
    - bandwidth is the amount of data that can be communicated per unit of time.
  - Synchronous vs. asynchronous communications
  - Scope of communications
    - Point-to-point
    - Collective
  - Efficiency of communications
  - Overhead and Complexity

- Overall, try to minimize the communication!
Synchronization

- Managing the sequence of work and the tasks performing it is a critical design consideration for most parallel programs.
  - Can be a significant factor in program performance (or lack of it)
  - Often requires "serialization" of segments of the program.

- Types of Synchronization:
  - Barrier
  - Lock/semaphore
  - Synchronous communication operations
Communication and Synchronization

- In shared-memory programs, we often implicitly communicate among the threads by synchronizing them.
- In distributed-memory programs, we often implicitly synchronize the processes by communicating among them.
Parallel Programming Concepts

Questions?
Parallel Programming Concepts

Lab and Exercise Session
Quiz

- Quiz link
  - https://goo.gl/forms/j8B1HXtrrNoCE0vx1
Exercise 0 - Log onto Cluster

- Open a terminal and log onto SuperMike2, download the git repository, compile the examples:

```bash
# log onto SuperMike2, hpctrn?? is your training account
[user@locallaptop ]$ ssh -X hpctrn??@mike.hpc.lsu.edu
# submit an interactive job to shelob queue
[hpctrn01@mike2]$ qsub -l nodes=1:ppn=16,walltime=8:00:00 -q shelob -A hpc_trn17mike2
qsub: waiting for job 658202.mike3 to start
...job welcome message...
[hpctrn01@shelob001]$ git clone https://github.com/lsuhpchelp/loniworkshop2017.git
[hpctrn01@shelob001 ~]$ git clone https://github.com/lsuhpchelp/loniworkshop2017.git
Initialized empty Git repository in /home/hpctrn01/loniworkshop2017/.git/
...
Unpacking objects: 100% (47/47), done.
[hpctrn01@shelob001 ~]$ cd loniworkshop2017/day1morning/exercise/
# build all executables
[hpctrn01@shelob001 exercise]$ make
[hpctrn01@shelob001 exercise]$ vi ~/.soft
# Add +Python-2.7.3-gcc-4.4.6 to ~/.soft and then resoft, for plotting purpose
[hpctrn01@shelob001 exercise]$ resoft
[hpctrn01@shelob001 exercise]$ which python
/usr/local/packages/Python/2.7.3/gcc-4.4.6/bin/python
```
Exercise 1

Compile and run OpenMP, MPI programs

- There is an example OpenMP and MPI programs provided for you, compile the two programs using the intel (recommended) or gcc compiler

  a) Observe the nondeterminism of the thread/process output

  ```bash
  # compile and run the hello world openmp code
  $ icc -openmp hello_openmp.c -o hello_openmp.out
  $ ./hello_openmp.out
  # compile and run the hello world mpi code
  $ mpicc hello_mpi.c -o hello_mpi.out
  $ mpirun -np 16 ./hello_mpi.out
  
  b) Observe the running multiple processes/threads using the “top” utility (with the H option for openmp codes)
  
  # run the mpi laplacian solver
  $ ./run_lp_mpi.sh
  # open another terminal and ssh to the compute node, then type “top”
  # run the openmp laplacian solver
  $ ./run_lp_omp.sh
  # open another terminal and go to the compute node, then type “top -H”
  ```
Exercise 1

- b) Expected results from the “top”
  - For the MPI version, you should see **16 processes**:

```
    top - 22:18:12 up 53 days, 7:46, 1 user, load average: 1.39, 1.84, 1.49
    Tasks: 830 total, 17 running, 813 sleeping, 0 stopped, 0 zombie
    Cpu(s): 99.9%us, 0.1%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
    Mem: 65908644k total, 36938372k used, 28970272k free, 154016k buffers
    Swap: 134217724k total, 10584k used, 134207140k free, 30115232k cached

    PID  USER     PR  NI  VIRT  RES  SHR  %CPU %MEM    TIME+  COMMAND
    ---  -------  --  --  -----  ---  ---  -----  -----  -------  
    68185 hpctrn01  20   0  219m  18m  6588 R 100.1  0.0  0:05.28  lp_mpi.out
    68187 hpctrn01  20   0  219m  18m  6546 R 100.1  0.0  0:05.28  lp_mpi.out
    68180 hpctrn01  20   0  348m  21m  6480 R  99.8  0.0  0:05.27  lp_mpi.out
    68181 hpctrn01  20   0  219m  18m  6508 R  99.8  0.0  0:05.27  lp_mpi.out
    68182 hpctrn01  20   0  219m  18m  6512 R  99.8  0.0  0:05.27  lp_mpi.out
    68183 hpctrn01  20   0  219m  18m  6524 R  99.8  0.0  0:05.27  lp_mpi.out
    68184 hpctrn01  20   0  219m  18m  6508 R  99.8  0.0  0:05.27  lp_mpi.out
    68186 hpctrn01  20   0  219m  18m  6508 R  99.8  0.0  0:05.26  lp_mpi.out
```

- For the OpenMP version, you should see **16 threads (using top –H)**:

```
    top - 22:10:09 up 53 days, 7:38, 1 user, load average: 8.55, 2.34, 0.85
    Tasks: 783 total, 17 running, 766 sleeping, 0 stopped, 0 zombie
    Cpu(s): 99.3%us, 0.7%sy, 0.0%ni, 0.1%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
    Mem: 65908644k total, 36829108k used, 20079536k free, 154016k buffers
    Swap: 134217724k total, 10584k used, 134207140k free, 30103032k cached

    PID  USER     PR  NI  VIRT  RES  SHR  %CPU %MEM    TIME+  COMMAND
    ---  -------  --  --  -----  ---  ---  -----  -----  -------  
    67757 hpctrn01  20   0  1230m 142m 1120 R 100.2  0.2  0:42.79  lp_omp.out
    67747 hpctrn01  20   0  1230m 142m 1120 R  99.9  0.2  0:42.91  lp_omp.out
    67751 hpctrn01  20   0  1230m 142m 1120 R  99.9  0.2  0:42.77  lp_omp.out
    67752 hpctrn01  20   0  1230m 142m 1120 R  99.9  0.2  0:42.73  lp_omp.out
    67756 hpctrn01  20   0  1230m 142m 1120 R  99.9  0.2  0:42.75  lp_omp.out
    67758 hpctrn01  20   0  1230m 142m 1120 R  99.9  0.2  0:42.84  lp_omp.out
    67749 hpctrn01  20   0  1230m 142m 1120 R  99.6  0.2  0:42.87  lp_omp.out
```
Exercise 2 Performance

a) Suppose the run-time of a serial program is given by $T_{serial} = n^2$, where the units of the run-time are in microseconds. Suppose that a parallelization of this program has run-time

$$T_{parallel} = n^2 + \log_2(p)$$

Write a program that plots the speedups and efficiencies of this program for various values of $n$ and $p$. Run your program with $n = 10, 20, 40, 80, 160, 320$, and $p = 1, 2, 4, 8, 16, 32, 64, 128$.

You can use the python template script provided for you (ex2a.py), complete the portion (in FIXME) that calculates the Speedup $S$ and Efficiency $E$, or you can use your most comfortable plotting tool), based on the plot:

- What happens to the speedups and efficiencies as $p$ is increased and $n$ is held fixed?
- What happens when $p$ is fixed and $n$ is increased?
Exercise 2 Performance

b) Suppose that $T_{\text{serial}}$ and $T_{\text{overhead}}$ are both functions of the problem size $n$:

$$T_{\text{parallel}} = T_{\text{serial}} / p + T_{\text{overhead}}$$

Also suppose that we fix $p$ and increase the problem size $n$.

To increase Speedup and Efficiency with increasing problem size, what condition should be satisfied for $T_{\text{serial}}$ and $T_{\text{overhead}}$?
Exercise 3 - Scalability

- Compile and run the OpenMP version of the floating point sum operation (*floatoper_omp_par.c*). Perform strong scaling and weak scaling test using this code:

  - For strong scaling, the Efficiency $E$:

    $$ E_{\text{strong}} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}} $$

  - For weak scaling, the Efficiency $E$:

    $$ E_{\text{weak}} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} $$

  - Try to change the problem size and see the difference in efficiency
  - Hint: You can use the bash scripts provided for you, and try to change the variable *BASE*(the problem size N in the for loop) in *perf_omp_strong.sh* and *perf_omp_weak.sh*, observe the Efficiency v.s. Number of processors curve at different problem size N:

```
[fchen14@mike388 solution]$ ./perf_omp_strong.sh
[fchen14@mike388 solution]$ ./perf_omp_weak.sh
[fchen14@mike388 solution]$ ./ex3_perf_omp.py
```
Exercise 4 Load Balancing

- An MPI code need to process \( n \) data elements (indexed from 0 to \( n-1 \)) using \( p \) processes (indexed from 0 to \( p-1 \)), in order to achieve load balancing, each core (process) should be assigned roughly the same number of elements of computations. (*loadbalance_mpi.c*)
  
  - Try to complete the code (in *FIXME*) so that each process (identified by \( \text{pid} \)) prints the start and end index of the elements, for the below example the output should be similar to:

To compile the program:

[fchen14@mike275 exercise]$ mpicc loadbalance_mpi.c
# start 3 processes to process 13 elements
[fchen14@mike275 exercise]$ mpirun -np 3 ./a.out 13

<table>
<thead>
<tr>
<th>pid:0, my_start=0, my_size=5, my_end=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid:1, my_start=5, my_size=4, my_end=8</td>
</tr>
<tr>
<td>pid:2, my_start=9, my_size=4, my_end=12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>start_index</th>
<th>end_index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

elements \( n=13 \)

processes \( p=3 \)

\( p0 \) \( p1 \) \( p2 \)