Computação Paralela

Mest. Int. Engenharia Computacional

Ano letivo 2019/2020

Manuel Barroso, Nuno Lau

Designing Parallel Programs

Acknowledgement: This lecture is based on slides by Florent Nolot,
“Introduction to Parallel Computing”, Université de Reims Champagne-Ardenne

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Types of Synchronization

- **Barrier**
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
  - When the last task reaches the barrier, all tasks are synchronized.
  - What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.

Types of Synchronization

- **Lock / semaphore**
  - Can involve any number of tasks
  - Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
  - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
  - Can be blocking or non-blocking

Types of Synchronization

- **Synchronous communication operations**
  - Involves only those tasks executing a communication operation
  - When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.
  - Discussed previously in the Communications section.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Definitions

- 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.

Examples (1): Loop carried data dependence

```
for (j = START; j<END; j++)
  A(j) = A(j-1) * 2.0500
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.
- If Task 2 has A(J) and task 1 has A(J-1), computing the correct value of A(J) necessitates:
  - Distributed memory architecture - task 2 must obtain the value of A(J-1) from task 1 after task 1 finishes its computation
  - Shared memory architecture - task 2 must read A(J-1) after task 1 updates it

Examples (2): **Loop independent data dependence**

```
task 1          task 2
------          ------
X = 2           X = 4
.               .
.               .
Y = X**2        Y = X**3
```

- As with the previous example, parallelism is inhibited. The value of Y is dependent on:
  - Distributed memory architecture - if or when the value of X is communicated between the tasks.
  - Shared memory architecture - which task last stores the value of X.
- 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?

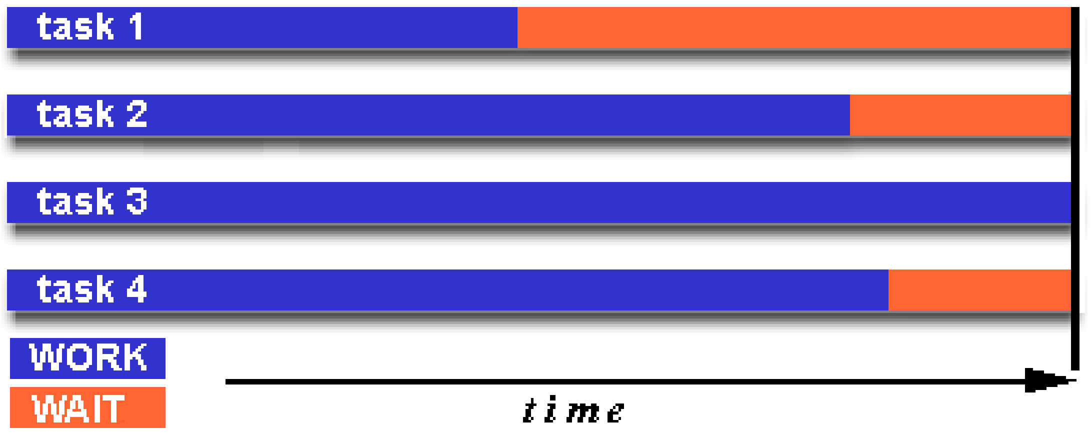
- Distributed memory architectures
  - communicate required data at synchronization points.
- Shared memory architectures
  - synchronize read/write operations between tasks.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Definition

- 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.



How to Achieve Load Balance? (1)

- **Equally partition the work each task receives**
  - 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.
  - If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool to detect any load imbalances. Adjust work accordingly.

How to Achieve Load Balance? (2)

- **Use dynamic work assignment**
  - Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:
    - Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros".
    - Adaptive grid methods - some tasks may need to refine their mesh while others don't.
    - *N*-body simulations - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks.
  - When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a ***scheduler - task pool*** approach. As each task finishes its work, it queues to get a new piece of work.
  - It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

Agenda

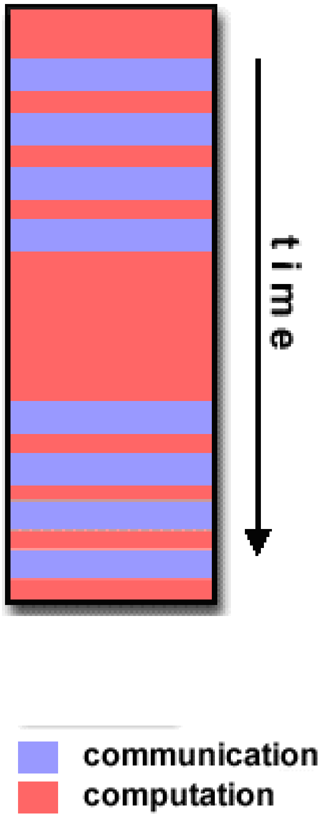
- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Definitions

- Computation / Communication Ratio:
  - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
  - Periods of computation are typically separated from periods of communication by synchronization events.
- Fine grain parallelism
- Coarse grain parallelism

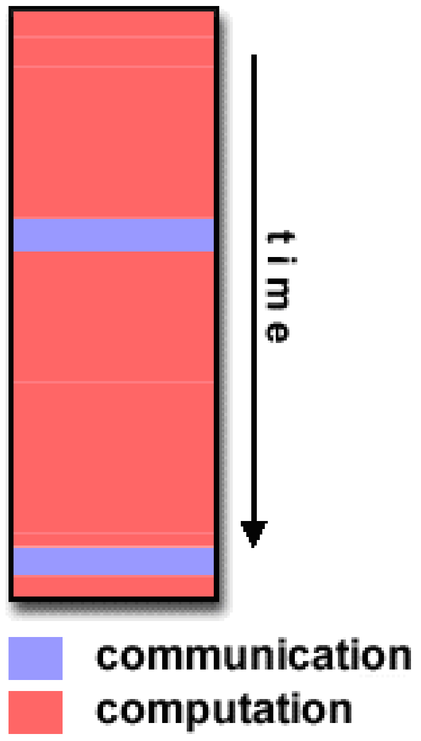
Fine-grain Parallelism

- 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



Which is Best?

- 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.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

The bad News

- I/O operations are generally regarded as inhibitors to parallelism
- Parallel I/O systems are immature or not available for all platforms
- In an environment where all tasks see the same filespace, write operations will result in file overwriting
- Read operations will be affected by the fileserver's ability to handle multiple read requests at the same time
- I/O that must be conducted over the network (NFS, non-local) can cause severe bottlenecks

The good News

- Some parallel file systems are available. For example:
  - GPFS: General Parallel File System for AIX (IBM)
  - Lustre: for Linux clusters (Cluster File Systems, Inc.)
  - PVFS/PVFS2: Parallel Virtual File System for Linux clusters (Clemson/Argonne/Ohio State/others)
  - PanFS: Panasas ActiveScale File System for Linux clusters (Panasas, Inc.)
  - HP SFS: HP StorageWorks Scalable File Share. Lustre based parallel file system (Global File System for Linux) product from HP
- The parallel I/O programming interface specification for MPI has been available since 1996 as part of MPI-2. Vendor and "free" implementations are now commonly available.

Some Options

- If you have access to a parallel file system, investigate using it. If you don't, keep reading...
- Rule #1: Reduce overall I/O as much as possible
- Confine I/O to specific serial portions of the job, and then use parallel communications to distribute data to parallel tasks. For example, Task 1 could read an input file and then communicate required data to other tasks. Likewise, Task 1 could perform write operation after receiving required data from all other tasks.
- For distributed memory systems with shared filespace, perform I/O in local, non-shared filespace. For example, each processor may have /tmp filespace which can used. This is usually much more efficient than performing I/O over the network to one's home directory.
- Create unique filenames for each tasks' input/output file(s)

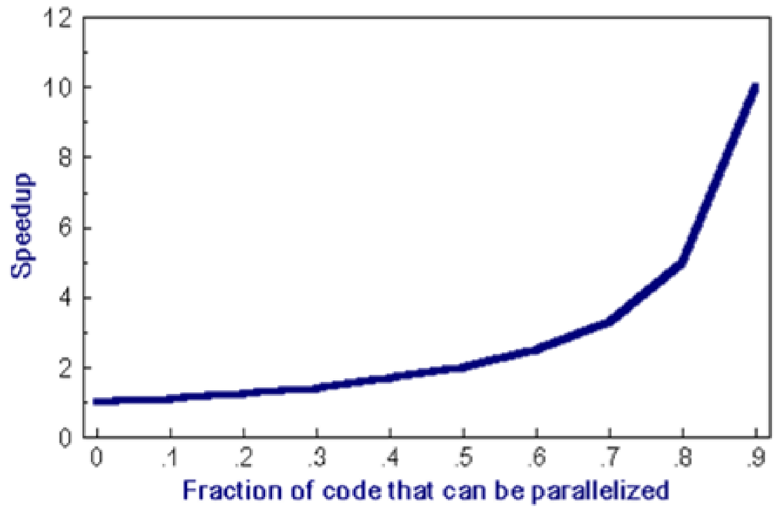
Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Amdahl's Law

- Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

```
                1
speedup = --------
             1 - P
```



- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup). If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

Amdahl's Law

- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by

```
                              1
speedup    =    ------------
                   P    +   S
                  ---
                   N
```

- where P = parallel fraction, N = number of processors and S = serial fraction

Amdahl's Law

- It soon becomes obvious that there are limits to the scalability of parallelism. For example, at P = .50, .90 and .99 (50%, 90% and 99% of the code is parallelizable)

		speedup	
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

Amdahl's Law

- However, certain problems demonstrate increased performance by increasing the problem size. For example:

2D Grid Calculations	**85 seconds**	**85%**
Serial fraction	**15 seconds**	**15%**

- We can increase the problem size by doubling the grid dimensions and halving the time step. This results in four times the number of grid points and twice the number of time steps. The timings then look like:

2D Grid Calculations	**680 seconds**	**97.84%**
Serial fraction	**15 seconds**	**2.16%**

- Problems that increase the percentage of parallel time with their size are more ***scalable*** than problems with a fixed percentage of parallel time.

Complexity

- In general, parallel applications are much more complex than corresponding serial applications, perhaps an order of magnitude. Not only do you have multiple instruction streams executing at the same time, but you also have data flowing between them.
- The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle:
  - Design
  - Coding
  - Debugging
  - Tuning
  - Maintenance
- Adhering to "good" software development practices is essential when when working with parallel applications - especially if somebody besides you will have to work with the software.

Portability

- Thanks to standardization in several APIs, such as MPI, POSIX threads, HPF and OpenMP, portability issues with parallel programs are not as serious as in years past. However...
- All of the usual portability issues associated with serial programs apply to parallel programs. For example, if you use vendor "enhancements" to Fortran, C or C++, portability will be a problem.
- Even though standards exist for several APIs, implementations will differ in a number of details, sometimes to the point of requiring code modifications in order to effect portability.
- Operating systems can play a key role in code portability issues.
- Hardware architectures are characteristically highly variable and can affect portability.

Resource Requirements

- The primary intent of parallel programming is to decrease execution wall clock time, however in order to accomplish this, more CPU time is required. For example, a parallel code that runs in 1 hour on 8 processors actually uses 8 hours of CPU time.
- The amount of memory required can be greater for parallel codes than serial codes, due to the need to replicate data and for overheads associated with parallel support libraries and subsystems.
- For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation. The overhead costs associated with setting up the parallel environment, task creation, communications and task termination can comprise a significant portion of the total execution time for short runs.

Scalability

- The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more machines is rarely the answer.
- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. Most parallel solutions demonstrate this characteristic at some point.
- Hardware factors play a significant role in scalability. Examples:
  - Memory-cpu bus bandwidth on an SMP machine
  - Communications network bandwidth
  - Amount of memory available on any given machine or set of machines
  - Processor clock speed
- Parallel support libraries and subsystems software can limit scalability independent of your application.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

- As with debugging, monitoring and analysing parallel program execution is significantly more of a challenge than for serial programs.
- A number of parallel tools for execution monitoring and program analysis are available.
- Some are quite useful; some are cross-platform also.
- One starting point: Performance Analysis Tools Tutorial
- Work remains to be done, particularly in the area of scalability.