4 Designing Parallel Programs

"Taking a couple of programming courses or programming a home computer does not qualify anyone to produce *safety-critical [*parallel *] software*.

Any engineer [*physicist, mathematician,...*] is not automatically qualified to be a software engineer – an extensive program of study and experience is required.

Safety-critical software engineering requires *training* and experience in addition to that required for noncritical software."

(An Investigation of the Therac-25 Accidents, IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18-41)

4.1 General

Automatic vs. Manual Parallelization

- Traditionally developing parallel programs has been a very manual process. The programmer is responsible for both identifying and implementing parallelism.
- Manually developing parallel codes is a time consuming and error-prone process.
- Automatic tools exist to assist the programmer with converting serial programs into parallel programs. The most common type of tool is a parallelizing compiler.

Automatic vs. Manual Parallelization

- A parallelizing compiler generally works in two different ways:
  - Fully Automatic: The compiler analyzes the source code and identifies opportunities for parallelism (loops etc).
  - Programmer Directed: Using e.g. compiler directives, the programmer explicitly tells the compiler how to parallelize the code.
- Automatic parallellization may not give increase in performance $\Longrightarrow$ limited use.

Before parallelization attempts

1. Choose numerically stable and efficient algorithm. It makes no sense to parallelize unefficient or unreliable code!
2. Identify the program's hotspots.
3. Identify inhibitors to parallelism in the hotspots.
4. Determine whether or not the hotspots can be parallelized.

Examples

- *A parallelizable problem:* Numerical integration of a function $f : \mathbb{R} \to \mathbb{R}$:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i),$$

  where $(w_i, x_i),\ i = 1, ..., n$ are given constants.

- *A non-parallelizable problem:* Calculation of the Fibonacci numbers by $F_{k+2} = F_{k+1} + F_k$. The calculation of the $k + 2$ value uses those of both $k + 1$ and k. These three terms cannot be calculated independently and therefore, not in parallel.

Concentrate on hotspots

- Know where most of the real work is being done.
- The main work in most scientific computing programs is done *in a few places* (linear system solvers, force calculation in MD,...)
- Profilers and performance analysis tools will help here.
- Focus on parallelizing the hotspots and ignore sections of the program that account for little CPU usage.

I/O usually inhibits parallelism

```
do i=1,n
  do j=1,n
    tmp=1.0/(1+(i-j)**2)
    write(6,fmt='(g9.2)',&
         advance='no') tmp
    a(i,j) = tmp
  end do
  write(6,*) ' '
end do
```

```
do i=1,n
  do j=1,n
    a(i,j)=1.0/(1+(i-j)**2)
  end do
end do
do i=1,n
  do j=1,n
    write(6,fmt='(g9.2)',&
     advance='no' ) a(i,j)
  end do
  write(6,*) ' '
end do
```

4.2 Partitioning

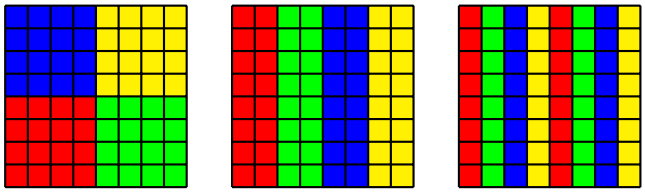
Partition

- 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 of the problem.
- There are two basic ways to partition computational work among parallel tasks: *domain decomposition* and *functional decomposition*.

Partitioning by Domain Decomposition

- The *data* associated with a problem is decomposed. Each parallel task then works on a portion of of the data.
- There are different ways to partition data.

  A matrix can be distributed onto four processors e.g. in the following ways:



Partitioning by Functional Decomposition

- The problem is decomposed according to the *work* that must be done. Each task then performs a portion of the overall work.
- Functional decomposition lends itself well to problems that can be split into different tasks. For example CFD: task1 computes x-veloc., task2 y-veloc, task3 pressure, and task4 temperature.
- Combining these two types of problem decomposition is common and natural.

4.3 Communications and syncronization

When communication is needed?

- Some problems can be decomposed and executed in parallel with virtually no need for tasks to share data.
- These types of problems are called *embarrassingly parallel*. Very little inter-task communication is required.
- Let matrices A, B, and C be decomposed onto different processors in similar ways. Then matrix addition

$$c_{i,j} = a_{i,j} + b_{i,j}, \quad i, j = 1, ..., n$$

  is embarassingly parallel.

When communication is needed?

- Usually, you **do** need communications.
- Most parallel applications are not trivial, and do require tasks to share data with each other.
- Let (square) matrices A, B, and C be decomposed onto different processors in similar ways. The matrix multiplication

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}, \quad i, j = 1, ..., n$$

  requires substantial amount of communication between tasks.

Cost of communications

- Inter-task communication always implies overhead.
- Machine cycles and resources that could be used for computation are instead used to package and transmit data.
- Communications frequently require some type of synchronization between tasks, which can result in tasks waiting instead of doing work.
- Competing communication traffic can saturate the available network bandwidth causing performance problems.

Latency vs. Bandwidth

- Latency is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
- Bandwidth is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec.
- The time needed communicate L megabytes long message from A to B is thus

$$t = t_{lat} + L/B$$

- Sending many small messages can cause latency to dominate communication overheads:

```
call send( nrows, 1, dest, message_type )
call send( ncols, 1, dest, message_type )
call send( offset, 1, dest, message_type )
```

- It is more efficient to pack small messages into a larger message:

```
mdata(1:3) = (/ nrows, ncols, offset /)
call send( mdata, 3, dest, message_type )
```

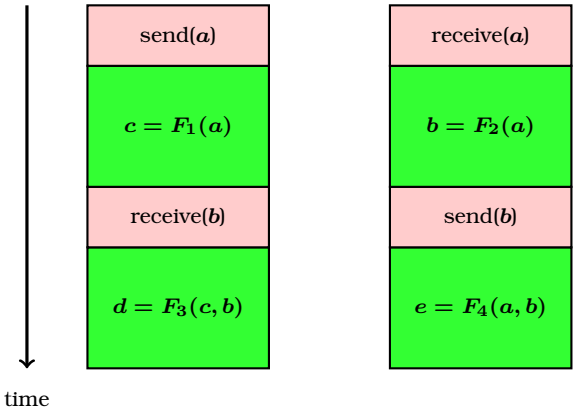
Synchronous vs. asynchronous communications

- Synchronous communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
- Synchronous communications are often referred to as *blocking communications* since other work must wait until the communications have completed.

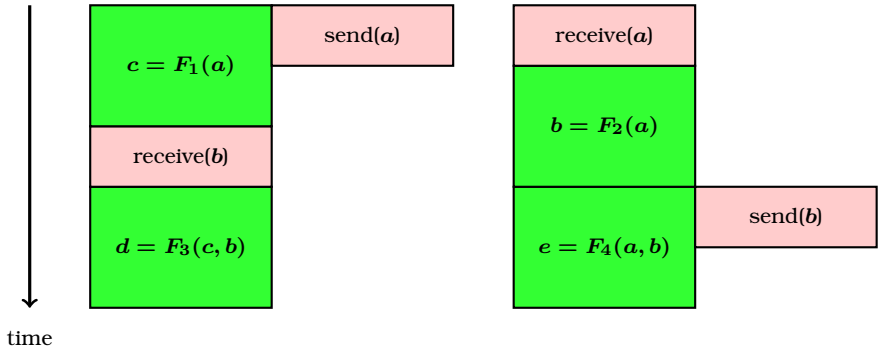
Synchronous vs. asynchronous communications

- Asynchronous communications allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
- Asynchronous communications are often referred to as *non-blocking communications* since other work can be done while the communications are taking place.
- Interleaving computation with communication is the benefit for using asynchronous communications.

Example: blocking communication



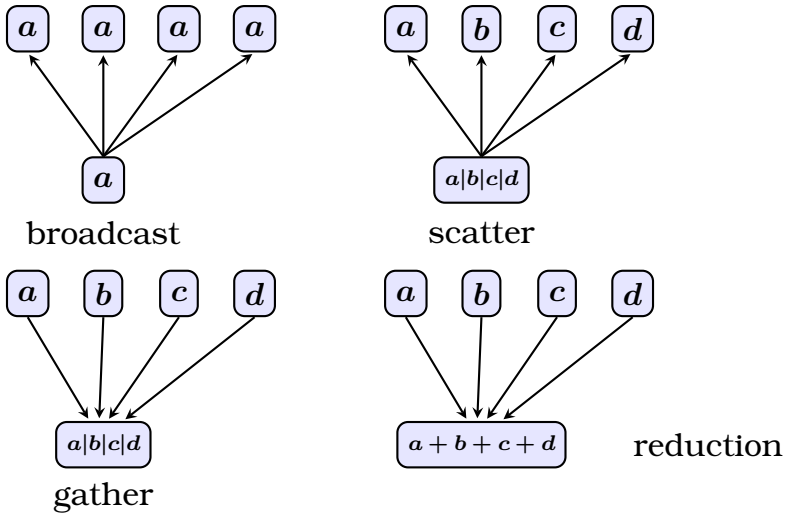
Example: non-blocking communication



Scope of communications

- Knowing which tasks must communicate with each other is critical during the design stage of a parallel code.
- *Point-to-point* involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver.
- *Collective* involves data sharing between *all* tasks (or between several tasks being members of a specified group).
- Point-to-point and collective communication may be implemented synchronously or asynchronously.

Typical collective communications



Synchronization by a barrier

- Usually 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.

Syncronisation by a 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.
- 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.

Synchronization by 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.
- Before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.

4.4 Data Dependencies

Data dependency – definition

- 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.

Example: Loop carried data dependence

```
DO j = my_start, my_end
   a(j) = a(j-1) * 2.0
END DO
```

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.

How to Handle Data Dependencies?

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

4.5 Load balancing and granularity

Load balancing

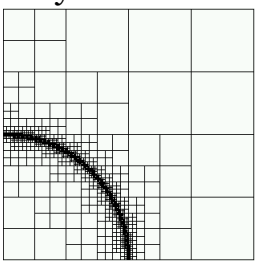
- Load balancing refers to the practice of distributing work among tasks so that all tasks are kept busy all of the time.
- Load balancing can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons.
- Example: if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.

Equally partition the work each task receives

- For array 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, use performance analysis tools to detect any load imbalances. Adjust work accordingly.

Use dynamic work assignment

- Certain classes of problems result in load imbalances even if data is evenly distributed among tasks. Example: Adaptive mesh methods – some tasks may need to refine their mesh while others don't.



- 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.

Granularity

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

- Relatively small amounts of computation are done between communication events.
- Low computation to communication ratio.
- Makes load balancing easier.
- Implies high communication overhead and less opportunity for performance enhancement.
- To fine granularity $\Longrightarrow$ 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.

Fine vs. coarse-grain parallelism – which is best?

- Depends on the algorithm and the hardware environment!
- Usually the overhead associated with communications and synchronization is high
  $\Longrightarrow$ advantageous to have coarse granularity.
- Fine-grain parallelism can help to reduce overheads due to load imbalance.