Parallel Programming Patterns

Like all programs, parallel programs contain many **patterns**: useful ways of writing code that are used repeatedly by most developers because they work well in practice. These patterns have been documented by developers over time so that useful ways of organizing and writing good parallel code can be learned by new programmers (and even seasoned veterans).

An organization of parallel patterns

When writing parallel programs, developers use patterns that can be grouped into two main categories:

1. Strategies
2. Concurrent Execution Mechanisms

Strategies

When you set out to write a program, whether it is parallel or not, you should be considering two primary strategic considerations:

1. What *algorithmic strategies* to use
2. Given the algorithmic strategies, what *implementation strategies* to use

In the examples in this document we introduce some well-used patterns for both algorithmic strategies and implementation strategies. Parallel algorithmic strategies primarily have to do with making choices about what tasks can be done concurrently by multiple processing units executing concurrently. Parallel programs often make use of several patterns of implementation strategies. Some of these patterns contribute to the overall structure of the program, and others are concerned with how the data that is being computed by multiple processing units is structured. As you will see, the patternlets introduce more algorithmic strategy patterns and program structure implementation strategy patterns than data structure implementation strategy patterns.

Concurrent Execution Mechanisms

There are a number of parallel code patterns that are closely related to the system or hardware that a program is being written for and the software library used to enable parallelism, or concurrent execution. These *concurrent execution* patterns fall into two major categories:

1. *Process/Thread control* patterns, which dictate how the processing units of parallel execution on the hardware (either a process or a thread, depending on the hardware and software used) are controlled at run time. For the patternlets described in this document, the software libraries that provide system parallelism have these patterns built into them, so they will be hidden from the programmer.
2. *Coordination* patterns, which set up how multiple concurrently running tasks on processing units coordinate to complete the parallel computation desired.

In parallel processing, most software uses one of two major *coordination patterns*:

1. **message passing** between concurrent processes on either single multiprocessor machines or clusters of distributed computers, and
2. **mutual exclusion** between threads executing concurrently on a single shared memory system.

These two types of computation are often realized using two very popular C/C++ libraries:

1. MPI, or Message Passing Interface, for message passing.
2. OpenMP for threaded, shared memory applications.

OpenMP is built on a lower-level POSIX library called Pthreads, which can also be used by itself on shared memory systems.

A third emerging type of parallel implementation involves a *hybrid computation* that uses both of the above patterns together, using a cluster of computers, each of which executes multiple threads. This type of hybrid program often uses MPI and OpenMP together in one program, which runs on multiple computers in a cluster.

Race condition

A **race condition** or **race hazard** is the behavior of an electronics, software, or other system where the output is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when events do not happen in the order the programmer intended.

The term was already in use by 1954, for example in DA Huffman's paper "The synthesis of sequential switching circuits".

Race conditions can occur especially in logic circuits, and in multithreaded or distributed software programs.

Software

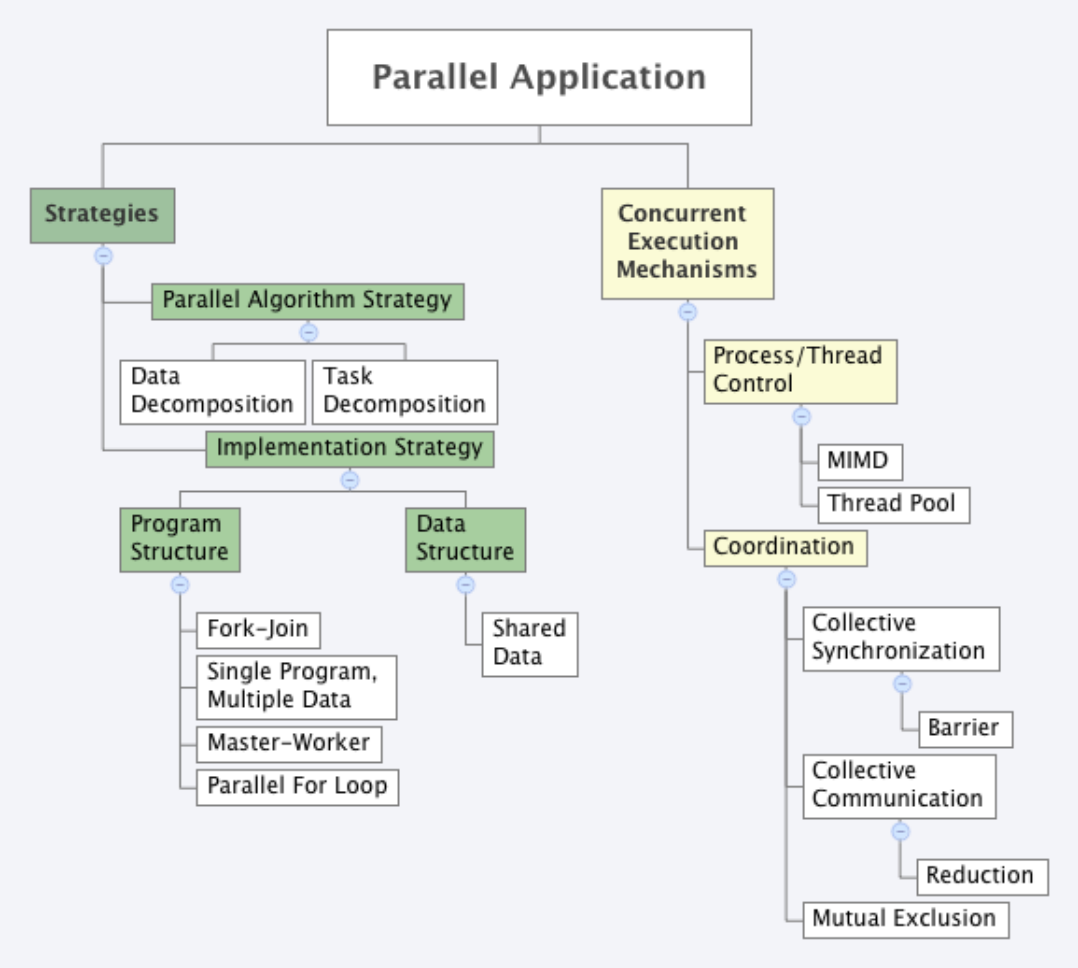
Race conditions arise in software when an application depends on the sequence or timing of processes or threads for it to operate properly. As with electronics, there are critical race conditions that result in invalid execution and bugs. Critical race conditions often happen when the processes or threads depend on some shared state. Operations upon shared states are critical sections that must be mutually exclusive. Failure to obey this rule opens up the possibility of corrupting the shared state.

The memory model defined in the C11 and C++11 standards uses the term "data race" for a race condition caused by potentially concurrent operations on a shared memory location, of which at least one is a write. A C or C++ program containing a data race has undefined behavior

Race conditions have a reputation of being difficult to reproduce and debug, since the end result is nondeterministic and depends on the relative timing between interfering threads. Problems occurring in production systems can therefore disappear when running in debug mode, when additional logging is added, or when attaching a debugger, often referred to as a "Heisenbug". It is therefore better to avoid race conditions by careful software design rather than attempting to fix them afterwards.

Categorizing Patterns

There has been a fair amount of work by several researchers who have categorized patterns found in parallel programs. We have shown you simple examples of several of them that are very common when writing OpenMP programs that use shared memory. Now that you have seen them, you can try to imagine the patterns falling into the categories shown on the following diagram:



Parallelism Views

- Where can we find parallelism?
- Program (task) view
  - Statement level
    - Between program statements
    - Which statements can be executed at the same time?
  - Block level / Loop level / Routine level / Process level
    - Larger-grained program statements
- Data view
  - How is data operated on?
  - Where does data reside?
- Resource view

Parallelism, Correctness, and Dependence

- Parallel execution, from any point of view, will be constrained by the sequence of operations needed to be performed for a correct result
- Parallel execution must address control, data, and system dependences
- A *dependency* arises when one operation depends on an earlier operation to complete and produce a result before this later operation can be performed
- We extend this notion of dependency to resources since some operations may depend on certain resources
  - For example, due to where data is located

Executing Two Statements in Parallel

- Want to execute two statements in parallel
- On one processor:

```
Statement 1;
Statement 2;
```

- On two processors:

```
Processor 1:          Processor 2:
  Statement 1;          Statement 2;
```

- Fundamental (*concurrent*) execution assumption
  - Processors execute independent of each other
  - No assumptions made about speed of processor execution

Sequential Consistency in Parallel Execution

- Case 1:

```
Processor 1:        Processor 2:
  statement 1;
                      statement 2;
```

time ↓

- Case 2:

```
Processor 1:        Processor 2:
                      statement 2;
  statement 1;
```

time ↓

- Sequential consistency
  - Statements execution does not interfere with each other
  - Computation results are the same (independent of order)

Independent versus Dependent

- In other words the execution of

```
statement1;
statement2;
```

must be equivalent to

```
statement2;
statement1;
```

- Their order of execution must not matter!
- If true, the statements are *independent* of each other
- Two statements are *dependent* when the order of their execution affects the computation outcome

Examples

- Example 1
  - S1: a=1;
  - S2: b=1;
  - Statements are independent
- Example 2
  - S1: a=1;
  - S2: b=a;
  - Dependent (*true (flow) dependence*)
    - Second is dependent on first
    - Can you remove dependency?
- Example 3
  - S1: a=f(x);
  - S2: a=b;
  - Dependent (*output dependence*)
    - Second is dependent on first
    - Can you remove dependency? How?
- Example 4
  - S1: a=b;
  - S2: b=1;
  - Dependent (*anti-dependence*)
    - First is dependent on second
    - Can you remove dependency? How?

When can two statements execute in parallel?

- Statements S1 and S2 can execute in parallel if and only if there are *no dependences* between S1 and S2
  - True dependences
  - Anti-dependences
  - Output dependences
- Some dependences can be remove by modifying the program
  - Rearranging statements
  - Eliminating statements

How do you compute dependence?

- Data dependence relations can be found by comparing the IN and OUT sets of each node
- The IN and OUT sets of a statement S are defined as:
  - IN(S) : set of memory locations (variables) that may be used in S
  - OUT(S) : set of memory locations (variables) that may be modified by S
- Note that these sets include all memory locations that may be fetched or modified
- As such, the sets can be conservatively large

Loop-Level Parallelism

- Significant parallelism can be identified within loops

```
for (i=0; i<100; i++)
   S1: a[i] = i;
```

```
for (i=0; i<100; i++) {
   S1: a[i] = i;
   S2: b[i] = 2*i;
}
```

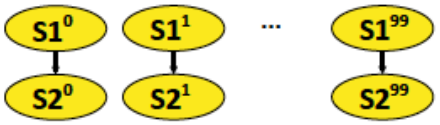
- Dependencies? What about *i*, the loop index?
- *DOALL* loop (a.k.a. *foreach* loop)
  - All iterations are independent of each other
  - All statements be executed in parallel at the same time
    - Is this really true?

Iteration Space

- Unroll loop into separate statements / iterations
- Show dependences between iterations

```
for (i=0; i<100; i++)
    S1: a[i] = i;
```


```
for (i=0; i<100; i++) {
    S1: a[i] = i;
    S2: b[i] = 2*i;
}
```



Key Ideas for Dependency Analysis

- To execute in parallel:
  - Statement order must not matter
  - Statements must not have dependences
- Some dependences can be removed
- Some dependences may not be obvious