Parallel Program Design Patterns

Shuhao Zhang

Nanyang Technological University

shuhao.zhang@ntu.edu.sg

June 4, 2024

Objectives

- After understanding of the correctness issues, synchronization techniques, and performance optimization techniques,
- You shall be ready to design your own “parallel program" from scratch.
- And, how? We will see multiple design patterns of parallel program that you may take as reference.

Overview

A parallel programming pattern provides a coordination structure for tasks:

- similar to design patterns from Software Engineering

Implication

- Design a parallel program following a certain pattern makes both the development and analysis easier.
- Some frameworks or libraries provide APIs that force users to follow certain patterns in developing applications. For example, MapReduce (hadoop).

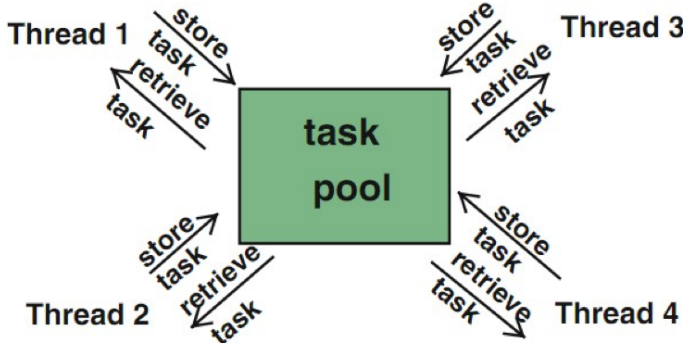
Outline

We will go through the following patterns:

- Task Pool
- Fork-Join
- Parbegin–Parend
- SIMD/SPMD
- Pipeline
- Master-Worker
- Client-Server
- Producer-Consumer

Task (Work) Pools

- A common data structure from which threads can access to retrieve tasks for execution

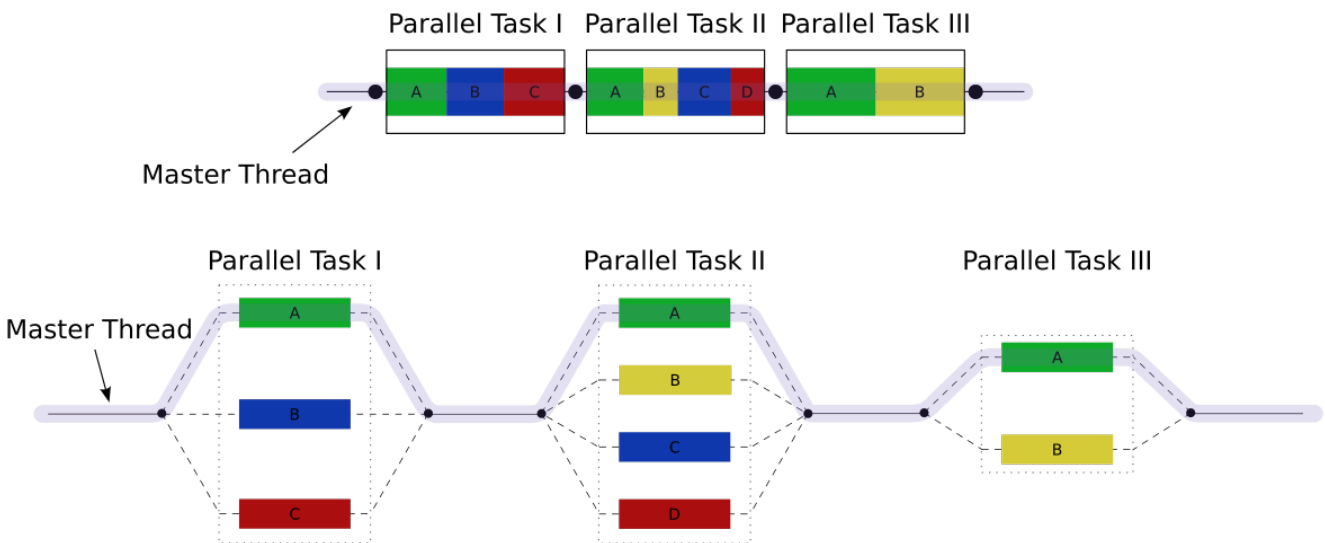


- Number of threads can be tuned dynamically, e.g., *newCachedThreadPool*
- During the processing of a task, a thread can generate new tasks and insert them into the task pool

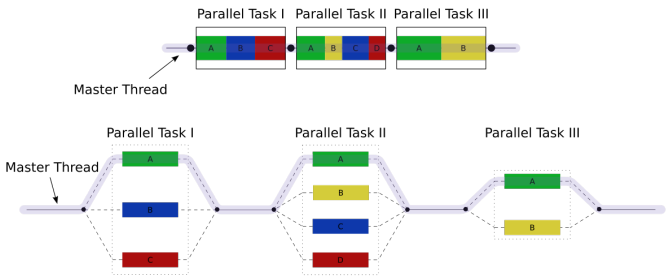
Task (Work) Pools (cont'd)

- Access to the task pool must be synchronized to avoid race conditions
- Execution of a parallel program is completed when
  - Task pool is empty
  - Each thread has terminated the processing of its last task
- Advantages:
  - Useful for adaptive and irregular applications as tasks can be generated dynamically
- Disadvantages:
  - For fine-grained tasks, the overhead of retrieval and insertion of tasks becomes significant. Access to the task pool must be synchronized to avoid race conditions.

Fork-Join


Parallel Task I
Parallel Task II
Parallel Task III
A
B
C
A
B
C
D
A
B
Master Thread
Parallel Task I
Parallel Task II
Parallel Task III
Master Thread
A
B
C
A
B
C
D
A
B

Fork-Join



- Task T creates a number of child tasks *T*1 ,..., *Tm* with a *fork* statement.
  - Child tasks work in parallel and execute a given program part or function
  - Task T can execute the same or a different program part or function
- Task T waits for the termination of T1,..., Tm by a *join* call
- Implementation: Language construct or a library function such as Pthreads, OpenMP and MPI-2

Fork-Join Pseudocode

```
Solve(problem):
if problem is small enough:
    solve problem directly (sequential
        algorithm)
else:
    for part in subdivide(problem)
        fork subtask to solve(part)
    join all subtasks spawned in previous loop
    return combined results
```

Remark

Why a "Join" is required?

Fork Operation

- Fork creates a child thread
- What does the child do?
- Typically, fork operates by assigning the child thread with some piece of "work"
- Child thread performs the piece of work and then exits by calling join with the parent
- Child work is usually specified by providing the child with a function to call on startup
- Nature of the child work relative to the parent is not specified

Join Operation

- Join informs the parent that the child has finished
- Child thread notifies the parent and then exits
  - Might provide some status back to the parent
- Parent thread waits for the child thread to join
  - Continues after the child thread joins
- Two scenarios
  - 1. Child joins first, then parent joins with no waiting
  - 2. Parent joins first and waits, child joins and parent then continues

Parbegin–Parend

- When an executing thread reaches a "parbegin–parend" construct, a set of threads is created and the statements of the construct are assigned to these threads for execution
- The statements following the parbegin–parend construct are only executed after all these threads have finished their work

Remark

Essentially, a special form of "Fork-Join". It is less powerful as it only applies when program is well structured. However, it is sometimes very convenient to use.

Common Usage

Mostly used by language construct or compiler directives, e.g., OpenMP

Parbegin-Parend Classical Example: Matrix Multiplication

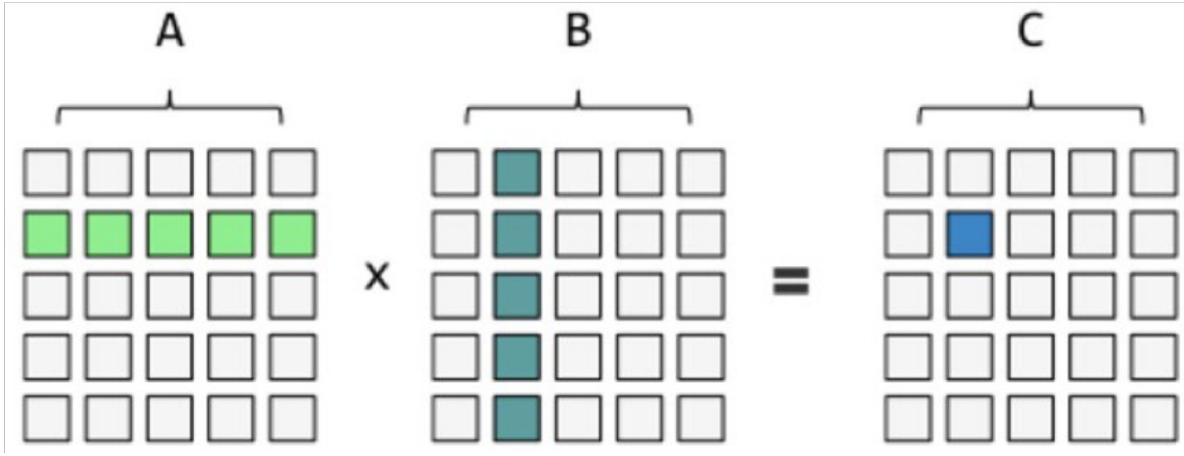


Figure: n by n square matrix multiplication

```
for i ← 0 to n−1
    for j ← 0 to n−1
        c[i, j] ← 0
        for k ← 0 to n−1
            c[i, j] ← c[i, j] + a[i, k] x b[k, j]
```

How to parallelize the program?

Parbegin-Parend Classical Example: Matrix Multiplication

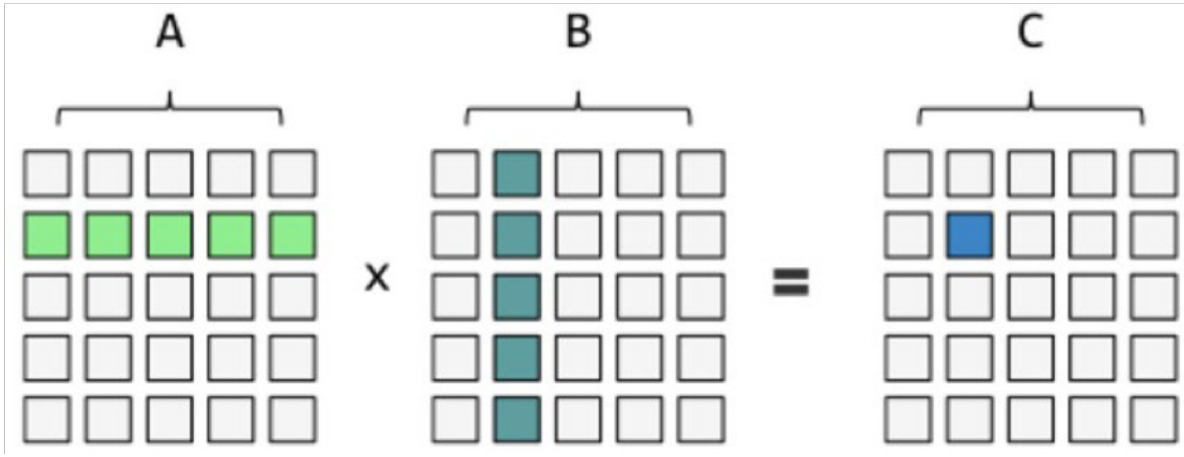


Figure: n by n square matrix multiplication

```
for i ← 0 to n−1
    for j ← 0 to n−1
        c[i, j] ← 0
        for k ← 0 to n−1
            c[i, j] ← c[i, j] + a[i, k] x b[k, j]
```

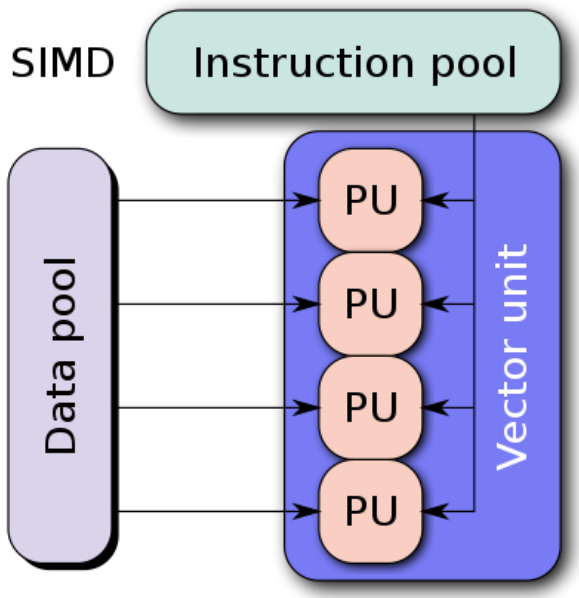
Key idea: Iterations of the for loop can be executed in parallel by a group threads

Example: Parallel For in OpenMP

```
// Parallelize the matrix multiplication (result = a x b)
// Each thread will work on one iteration of the outer-most loop
// Variables are shared among threads (a, b, result)
// and each thread has its own private copy (i, j, k)

#pragma omp parallel for shared(a, b, result)
                         private (i, j, k)

for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
        for (k = 0; k < size; k++)
            result.element[i][j] += a.element[i][k] *
                                    b.element[k][j];
```

SIMD

- **Single** instructions are executed *synchronously* by the different threads on *multiple* data
- **Implementation:**
  - SSE (Streaming SIMD Extensions) Instruction on intel processor
  - Graphic Processing Units (GPUs)



SPMD

- Same *program* executed on different processors but operate on different data
- All threads have equal rights and different threads work asynchronously with each other
- Different threads may execute different parts of the parallel program because of:
  - Different speeds of the executing processors
  - Control statement in program, e.g., If statement

Figure: Telphone operators

You may also view Mappers/Reducers in MapReduce framework as SPMD.

SIMD/SPMD Application

- SIMD/SPMD model is particularly appropriate for problems with a regular, predictable communication pattern.
  - MATLAB supports SPMD blocks.
  - Often adopted for GPU Programming

Notes

If you are interested for GPU programming, please self-study CUDA.

Concept of Pipeline

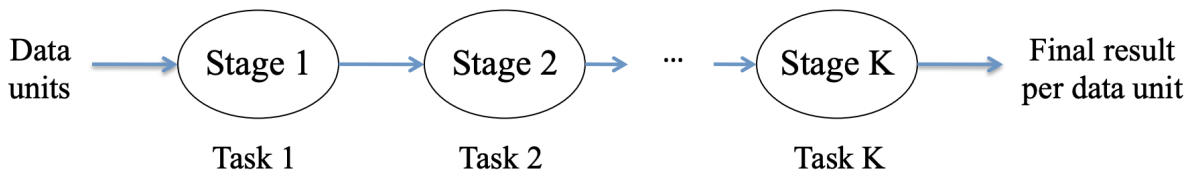
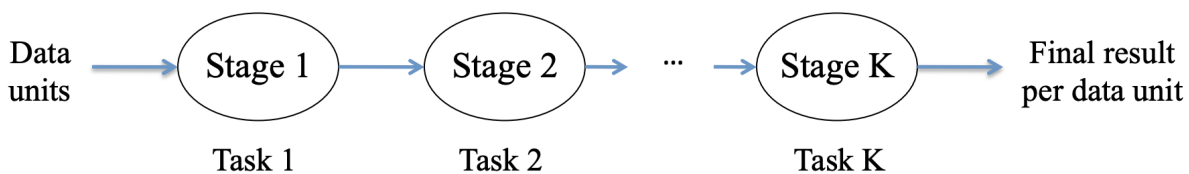


Figure: A pipeline is a linear sequence of stages

- Data flows through the pipeline
  - From Stage 1 to the last stage
  - Each stage performs some task, where inputs are results from the previous stage
  - Data is thought of as being composed of units
  - Each data unit can be processed separately in pipeline

Pipelining

- Data in the application is partitioned into a stream of data elements that flows through the each of the pipeline tasks one after the other to perform different processing steps
  - A form of functional parallelism: Stream parallelism
- **Important Implementation:** Apache Storm, Apache Flink – Stream Processing Engines



Pipelining Illustration

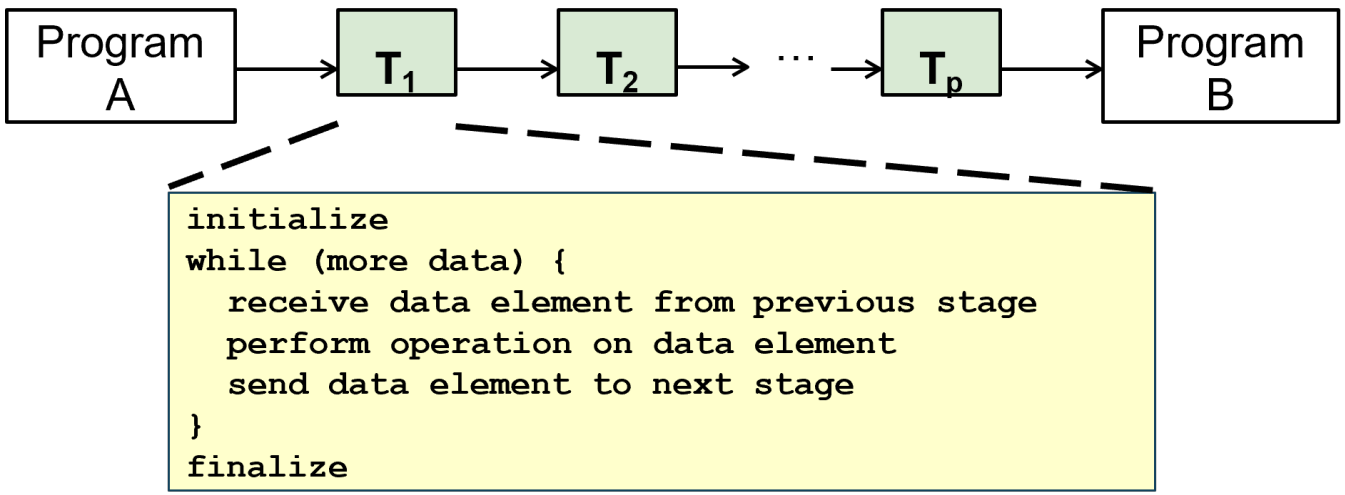


Figure: It also be used to link multiple programs working concurrently.

Pipeline Model: Example[1]

- Consider 3 data units and 4 tasks (stages)

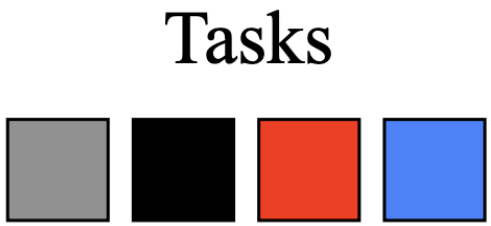
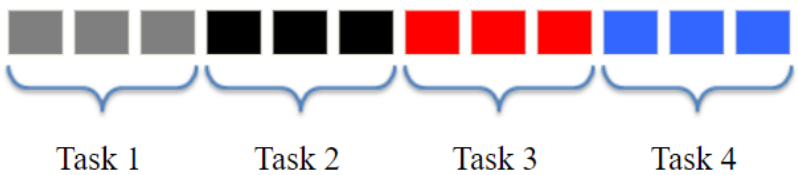


Figure: Assuming 4 tasks (stages)

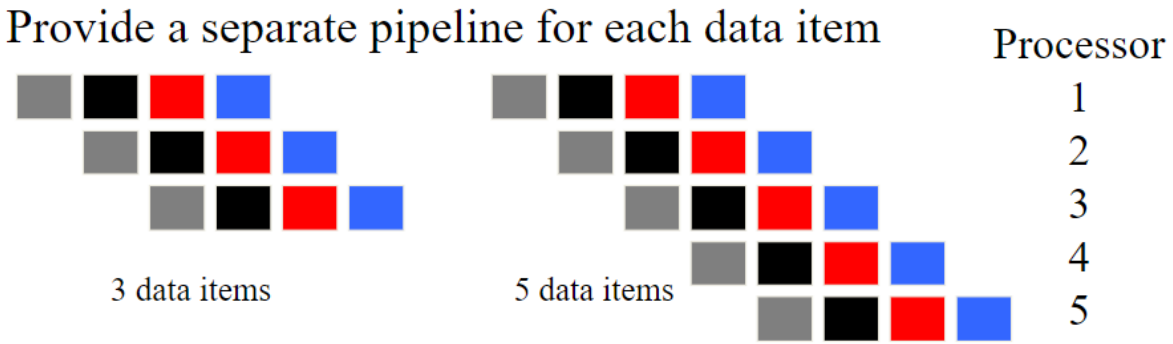
- Sequential pipeline execution (no parallel execution)



[1] credit:university of oregon, https://ipcc.cs.uoregon.edu/lectures/lecture-10-pipeline.pdf

Pipeline Model: Example

- We can provide a separate pipeline for each data item



- What do you notice as we increase the number of data items?

Pipeline Model: Example

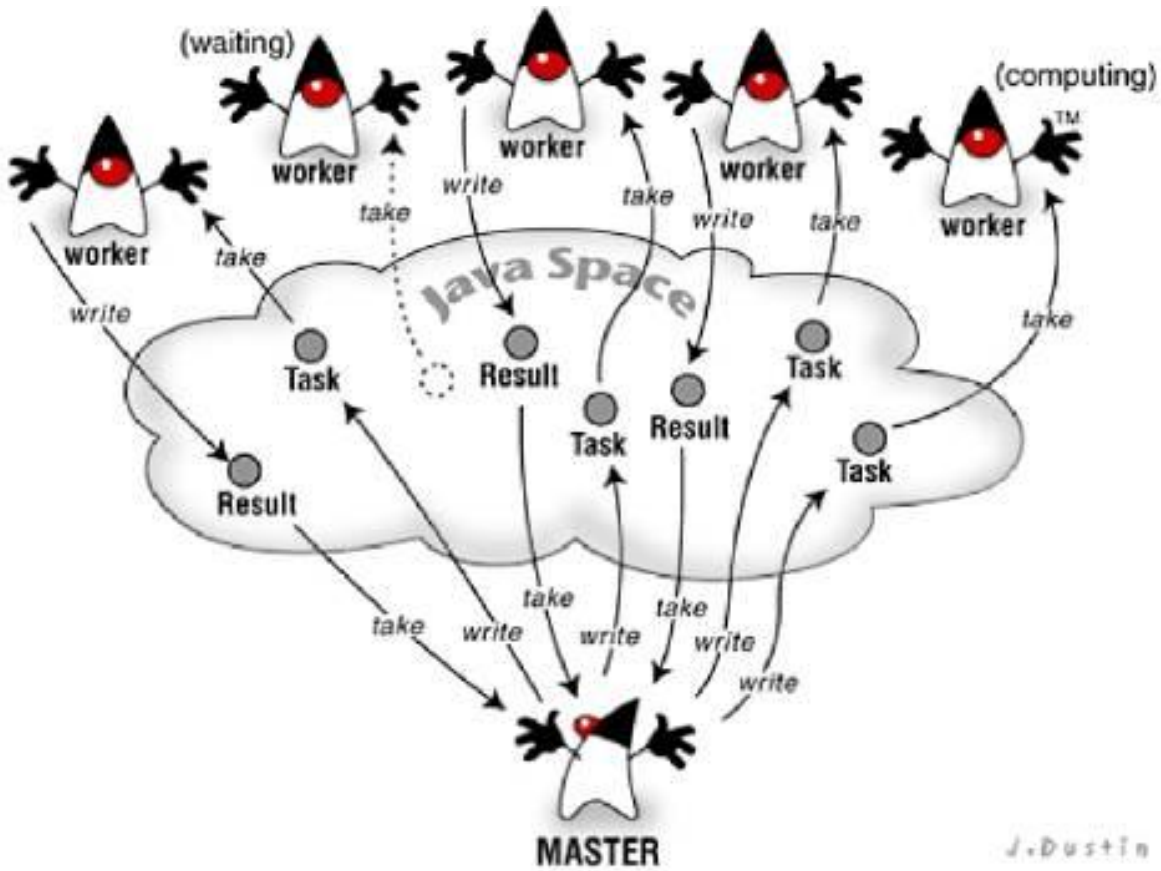
- The number of data items determine the maximum parallelism.

- Two parallel approach here:
  - processor executes the entire pipeline
  - processor assigned to a single task
- which one is better?

Master-Worker Pattern

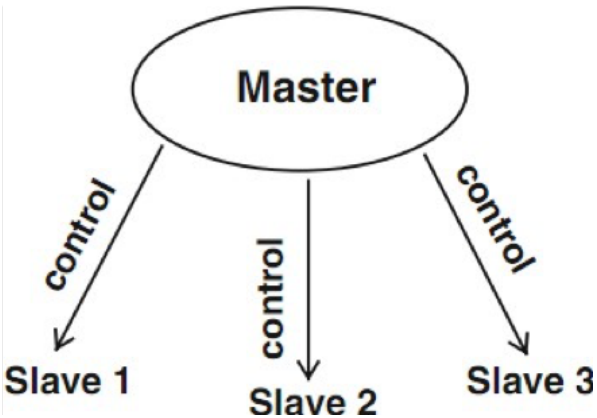
- A single program (master) controls the execution of the program
  - Master executes the main function
  - Assigns work to slave threads to perform computations
- Master task:
  - Generally responsible for coordination and perform initializations, timings, and output operations
- Worker task:
  - Wait for instruction from master task

Master-Worker Pattern



Master–Slave (or Master–Worker)

- A single program (master) controls the execution of the program
  - Master executes the main function
  - Assigns work to slave threads to perform the actual computations



- Master task:
  - Generally responsible for *coordination* and perform *initializations*, timings, and output operations.
- Slave task:
  - Wait for instruction from master task
- **Important Implementation:** Google MapReduce, Hadoop, Spark system.

Master Thread

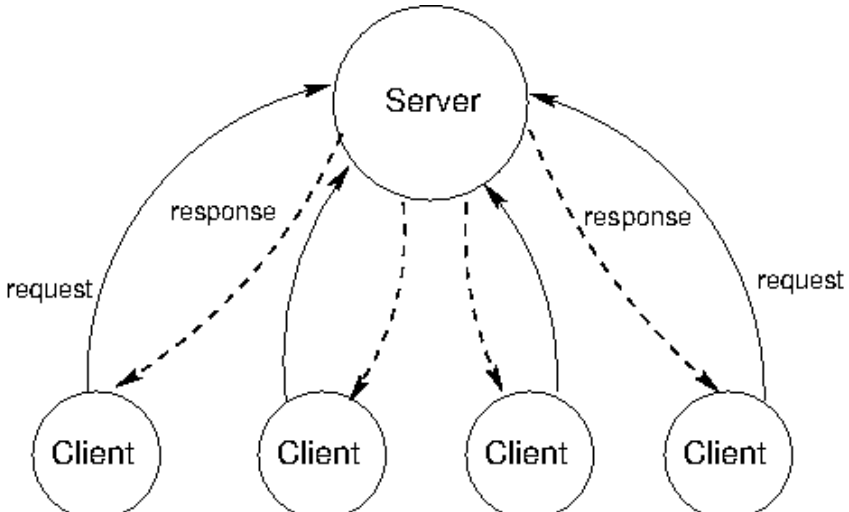
```
public class Master {
    private int slaveCount = 10;
    private Workload workload = new Workload();
    private Slave[] slaves = new Slave[slaveCount];
    public void run() {
      for(int i = 0; i < slaveCount; i++) {
         slaves[i] = new Slave(workload);  // create slaves
      }
      for(int i = 0; i < slaveCount; i++) {
         slaves[i].start(); // start slaves
      }
      for(int i = 0; i < slaveCount; i++) {
         try {
            slaves[i].join();// wait for slaves to stop
         } catch(InterruptedException ie) {
               System.err.println(ie.getMessage());
         } finally {
            System.out.println(slaves[i].getName() + " has stopped");
         }
      }
      System.out.println("The master will now stop ... ");
   }
}
```

Slave Thread

```
class Slave extends Thread {
        private Workload workload;
        private boolean done = false;
        public Slave(Workload workload) {
                this.workload = workload;
        }
        protected boolean work() {
        //perform some work that reduces the workload.
        }
    public void run() {
      while(!done) {
        done = work();
            // be cooperative:
        try{
            Thread.sleep(1000);
                }// sleep for 1 sec.
        catch(Exception e) {}
      }
    }
}
```

Client-Server

- Server compute requests from multiple client tasks concurrently
  - Can use multiple threads to compute a single request
- A task can generate requests to other tasks (client role) and process requests from other tasks (server role)
- Useful in heterogeneous systems such as cloud and grid computing

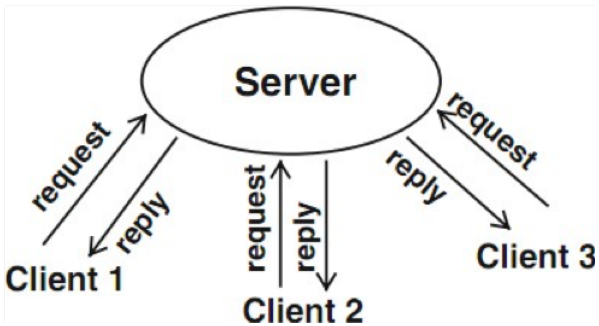


Client-Server

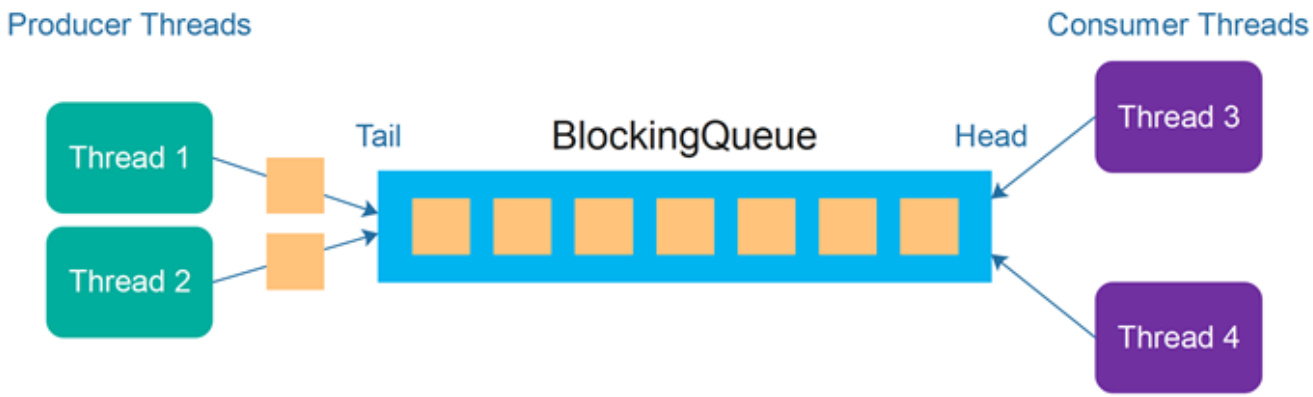
- MPMD (multiple program multiple data) model
- Server compute requests from multiple client tasks concurrently
  - Can use multiple threads to compute a single request?
- A task can generate requests to other tasks (client role) and process requests from other tasks (server role)

- Useful in heterogeneous systems such as cloud and grid computing

Reverse to master-worker: requests are sent from clients (workers).



Producer-Consumer Illustration

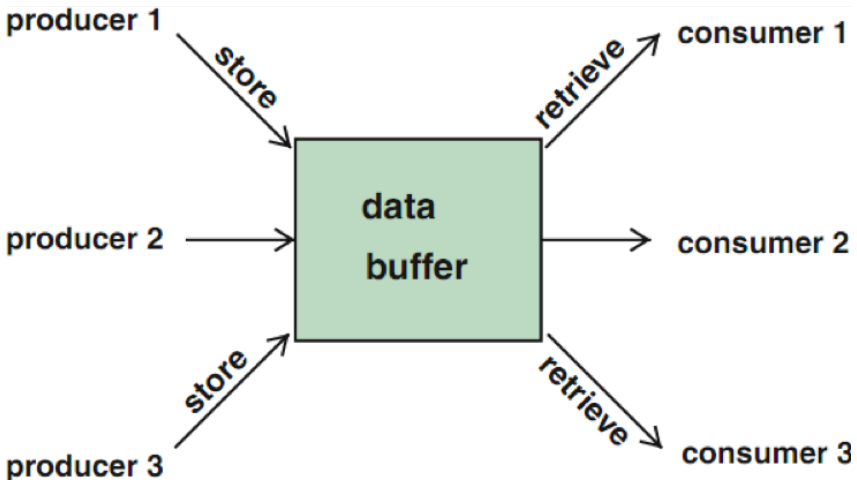


Producer-Consumer Pattern

- Producer-Consumer patterns are very common in programs;
- Usually some kind of buffering is involved between P and C;
- The buffer can be implemented as a **blocking queue**.

Producer–Consumer

- Producer threads produce data which are used as input by consumer threads



- Synchronization has to be used to ensure a correct coordination between producer and consumer threads

Producer–Consumer: Shared Buffers

```
void produce() {
    synchronized (buffer) {
        while (buffer is full)
            buffer.wait();
        Store an item to buffer;
        if (buffer was empty)
            buffer.notify();
    }
}
```

```
void consume() {
    synchronized (buffer) {
        while (buffer is empty)
            buffer.wait();
        Retrieve an item from buffer;
        if (buffer was full)
            buffer.notify();
    }
}
```

Block Queue

- Blocking queues are a powerful resource management tool for building reliable applications: they make your program more robust to overload by throttling activities that threaten to produce more work than can handled.
- Blocking queues are typically used in implementing the producer-consumer pattern

Executor Framework

- Java provides a fairly easy mechanism to write parallel program with task pool parallel pattern.
- Called - *Executor Framework*.

Executor Framework

- Instead of manually handle the assignment & orchestration & mapping.
  - Partition is still designed by developers.
- A easier way is to rely on the executor framework
- Once you define the tasks (Runable), you can let the executor framework to schedule the tasks to run in parallel and the JVM will take care of the rest – a short-cut
- A nice application of the task pool parallel programming pattern

Executor Framework

- The executor framework offers flexible thread pool management
- Executor provides a standard means of decoupling task submission from task execution.
  - The *Runnable* is the task itself.
  - The method *execute* defines how it is executed.

Example Task Pool Implementation: Java Thread Pool Executor

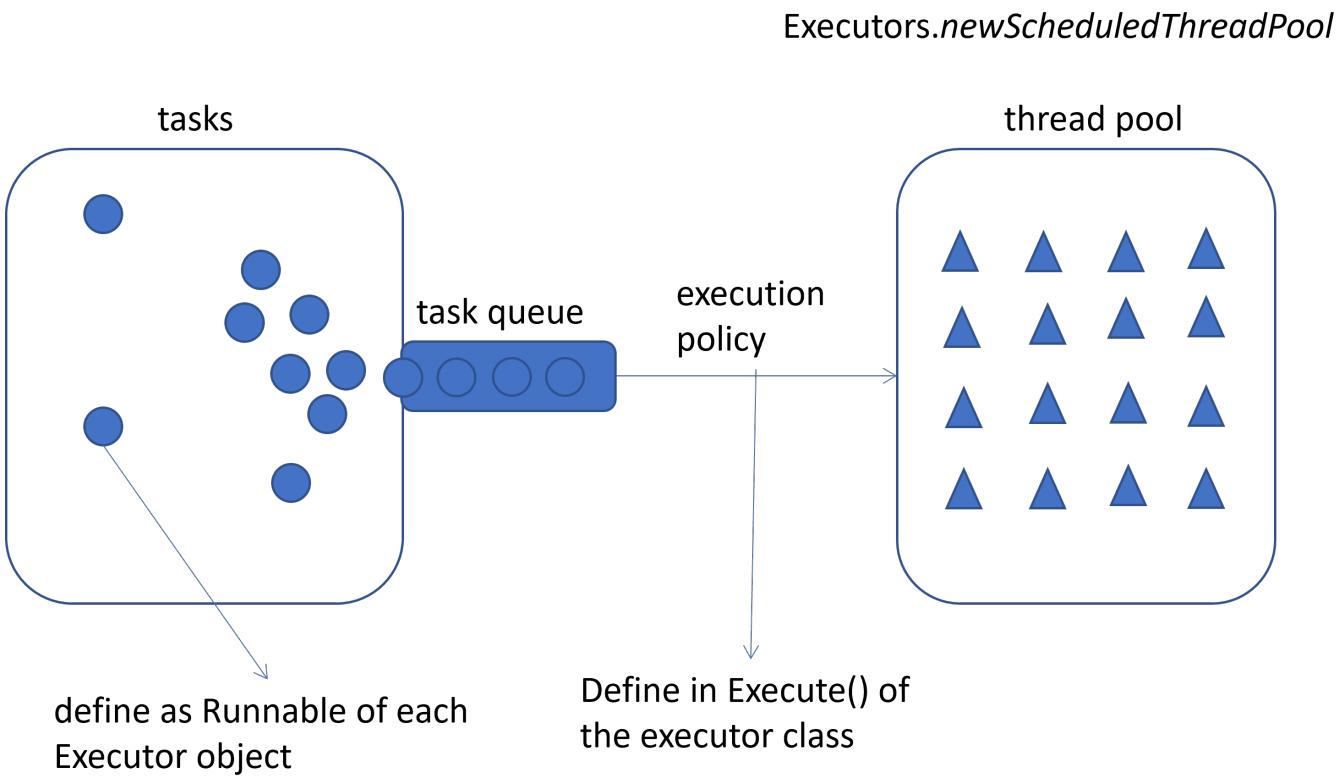
```
class ThreadPoolExample {

    public static void main(String[] args) {
        ExecutorService executor =
                    Executors.newFixedThreadPool(5);

        for (int i = 0; i < 10; i++) {
            Runnable Task = new Task( ..... );
            executor.execute( Task );
        }
        ......
    }
}
```

5 threads

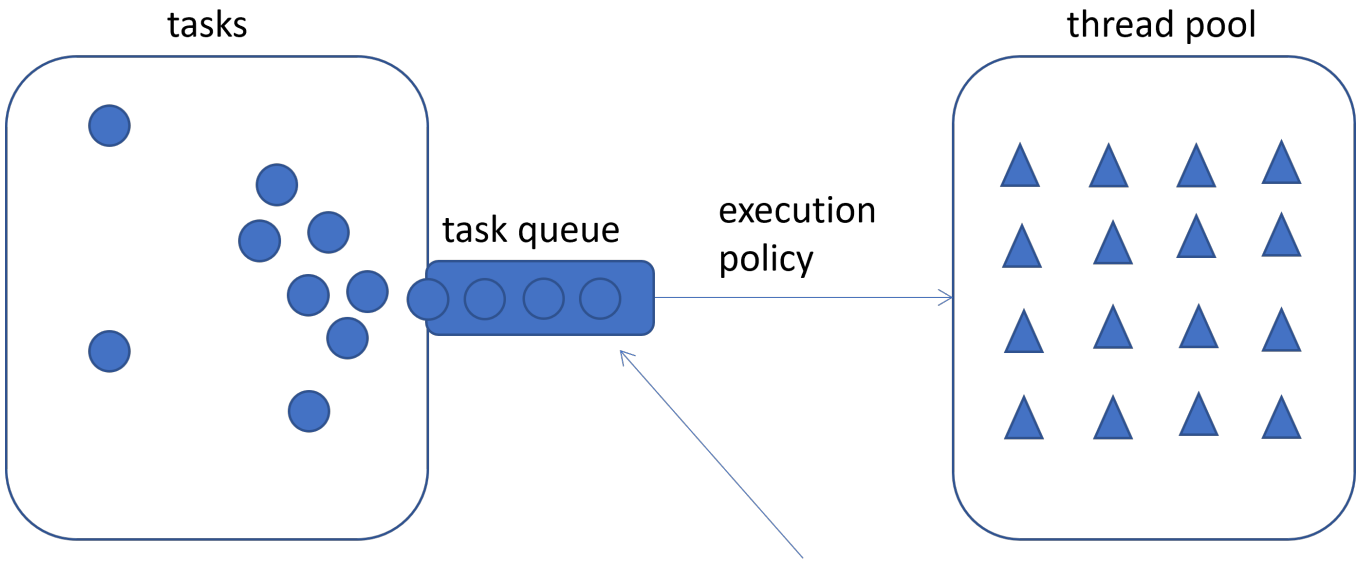
10 tasks added to the pool.

- The executor will assign task to the 5 threads:
  - After a thread finishes its task, another task from the pool will be assigned

How Executor Framework Works



How Executor Framework Works



The queue may grow unbounded!
When a bounded task queue fills up, the saturation policy comes into play.
The default: throw RejectedExecutionException

Advantage of Applying Executor Framework

- Reusing an existing thread; reduce thread creation and teardown costs.
- No latency associated with thread creation; improves responsiveness.

Tuneable

By properly tuning the size of the thread pool, you can have enough threads to keep the processors busy while not having so many that your application runs out of memory or thrashes due to competition among threads for resources

When to Use Executor Framework?

Executor Framework work best when tasks are homogeneous and independent.

- Dependency between tasks in the pool creates constraints on the execution policy which might result in problems (deadlock, liveness hazard, etc.)
  - It is still developers' responsibility to ensure program correctness.
- Long-running tasks may impair the responsiveness of the service managed by the Executor.
  - Especially when task queue is bounded.
- Reusing threads create channels for communication between tasks – risky to use them.
  - To be sure, you may enforce "violate" for example.

Executor Lifecycle

Shut down an Executor through ExecutorService

```
public interface ExecutorService extends Executor {
        void shutdown();
        List<Runnable> shutdownNow();
        boolean isShutdown();
        boolean isTerminated();
        boolean awaitTermination(long timeout,
            TimeUnit unit)
                throws InterruptedException;
        //additional convenience methods for task
            submission
}
```

shutdown() vs shutdownNow()

- shutdown()
  - will just tell the executor service that it can't accept new tasks, but the already submitted tasks continue to run
- shutdownNow()
  - will do the same AND will try to cancel the already submitted tasks by interrupting the relevant threads. Note that if your tasks ignore the interruption, shutdownNow() will behave exactly the same way as shutdown().

Different Built-in Executor Frameworks

- newFixedThreadPool
  - Fixed-size thread pool; creates threads as tasks are submitted, up to the maximum pool size and then attempts to keep the pool size constant
- newCachedThreadPool
  - Boundless, but the pool shrinks and grows when demand dictates so
- newSingleThreadExecutor
  - A single worker thread to process tasks, sequentially according to the order imposed by the task queue
- newScheduledThreadPool
  - A fixed-size thread pool that supports delayed and periodic task execution.

Tuning the Thread Pools

- The ideal size for a thread pool depends on the types of tasks and the deployment system
  - If it is too big, resource saturate
  - If it is too small, throughput suffers
- Heuristics
  - For compute intensive tasks, N+1 threads for a N-processor system
  - For tasks including I/O or other blocking operations, you want a larger pool

Optimal CPU Utilization

- Given these definitions:
  - N = number of CPUs (e.g., 4)
  - U = target CPU utilization (e.g., 1)
  - W/C = ratio of wait time to compute time (e.g., 0)
- The optimal pool size is:
  - M = N * U * (1 + W/C) (e.g., 4*1*1)
- The number of CPUs can be obtained by:
  - Runtime.getRuntime().availableProcessors()

More Than CPUs

- Other resources that can contribute to sizing constraints are memory, file handles, socket handles, database connections, etc.
  - Add up how much of those resources each task requires and divide that into the total quantity available.
- Alternatively, the size of the thread pool can be tuned by running the application using different pool sizes and observing the level of CPU and other resource utilization.

FYI: Design Our Own Executor

Decoupling submission from execution is that it lets you specify the execution policy for a given class of tasks.

- In what thread will tasks be executed?
- In what order should tasks be executed (FIFO)?
- How many tasks may execute concurrently?
- How many tasks may be queued pending execution?
- If a task has to be rejected because the system is overloaded, which task should be selected and how the application be notified?
- What actions should be taken before or after executing a task?

FYI: `https://www.geeksforgeeks.org/customthreadpoolexecutor-in-java-executor-framework/`

OpenMP

- OpenMP is a simple, powerful way to write shared memory programs.
- Start with sequential code and parallelize it using #pragma omp compiler directives.
- Incremental parallelization – make existing program parallel bit by bit.
- Initially, a single master thread exists.
- Parallel regions (sections of code) can be executed by a team of threads.
- Compiler takes care of creating and coordinating threads.
- Available for C / C++ and Fortran. Documentation at `http://openmp.org/wp/openmp-specifications/`

Parallel Regions

- The parallel directive forks a team of threads, each of which executes the following region, enclosed in {...}.
- Threads join at the end of the parallel region, and execution resumes with the single master thread.
- Number of threads can be set by:
  - `num_threads` clause after the parallel directive.
  - `omp_set_num_threads()` library routine.
  - Environment variable `OMP_NUM_THREADS`.
- Recommendation is one thread per processor/core.
- Threads can do the work in the region in parallel.
- Threads can do different things based on thread ID.
- Threads can share work using `for`, `sections`, `task`, etc. directives.
- Parallel regions can be nested.

Work Sharing

- Share some work inside a parallel region among threads.
- For example, `for` construct inside a parallel region partitions iterations of the loop among the threads.

```
#pragma omp for
for(i=0; i<n; i++) {do_stuff(i);}
```

- The way in which iterations are assigned to threads can be specified by an additional `schedule` clause.
- Does not start a new team of threads - that is done by an enclosing parallel construct.
- Implicit barrier at the end of the construct unless a `nowait` clause is included.

```
#pragma omp for nowait
for(i=0; i<n; i++) {do_stuff(i);}
```

Schedule Clause

Used for assigning iterations of parallel `for` to threads.

- `schedule(static[,chunk])`
  - Each thread gets a chunk of iterations of size "chunk" – by default chunks are approximately equal.
  - Chunks assigned in round-robin order.
- `schedule(dynamic[,chunk])`
  - Each time a thread finishes its iterations, it grabs "chunks" more iterations, until all have been executed – default is 1.
  - Dynamic scheduling has some overhead, but can result in better load balancing if iterations are not all equal sized.

Schedule Clause

- `schedule(guided[,chunk])`
  - Each thread dynamically grabs iterations where the size starts large and shrinks down to "chunk".
  - Dynamic load balancing with less overhead.
- `schedule(runtime)`
  - Schedule type and chunk size taken from the `OMP_SCHEDULE` environment variable.

Combined Parallel for

- If a parallel directive is followed by a single `for` directive, they can be combined.

```
#pragma omp parallel for schedule(static)
for (i=0; i<n; i++) {
    a[i] = a[i] + b[i];
}
```

- Several restrictions on the structure of the loop:
  - Number of iterations `n` must not change.
  - Loop increment must be fixed.
  - Must not exit loop prematurely (with `break`, `goto`, `throw`).
- Purpose of restrictions is so the amount of work in the loop can be determined at the start.

Other Work Sharing Constructs

- Sections construct
  - Each thread assigned some sections of work.
  - An implicit barrier at the end of the sections block. Can be turned off using `nowait`.
  - Each thread is responsible for some (possibly 0) sections.

```
#pragma omp parallel {
#pragma omp sections {
    #pragma omp section
    structured-block
    #pragma omp section
    structured-block
}
}
```

Other Work Sharing Constructs

- Single construct
  - Structured block is executed by one thread of the parallel region only.
  - Barrier implied unless `nowait` is used.

```
#pragma omp single
structured-block
```

- Master construct
  - Structured block is executed by the master thread only (no implicit barrier).

```
#pragma omp master
structured-block
```

Synchronization Constructs

- Critical sections
  - Only one thread can execute the associated structured block at a time.
  - Name can be used to identify the critical section. Critical sections with no name default to the same.

```
#pragma omp critical [name]
structured-block
```

- Atomic operations
  - Only one thread can execute the associated expression-statement at a time.
  - Only works for simple statements such as x++, max, test&set, etc.
  - More efficient than a critical section, done in hardware.

```
#pragma omp atomic
expression-statement
```

Synchronization Constructs

- Barriers
  - All threads must reach the barrier before any can proceed.
  - Make sure all threads always hit the barrier. Otherwise some threads can block.

```
#pragma omp barrier
```

- Ordered statements
  - Used in `for` and `parallel for` constructs to cause the structured block to be executed in strict loop order.

```
#pragma omp ordered
structured-block
```

- Flushing values
  - `flush(variable-list)` writes listed variables to memory to ensure memory consistency (see manual).

```
#pragma omp flush(variable-list)
```

Data Environment

- OpenMP has a shared memory programming model.
- Most variables are shared by default.
- Some variables are private by default:
  - Loop index of `for` / `parallel for` construct.
  - Stack variables (e.g. function argument or local variable) created during execution of a parallel region.
- Variable status can be changed using the following clauses in parallel regions and worksharing constructs, except `shared` which only applies to parallel regions.
  - `shared(variable-list)`
  - `private(variable-list)`

Data Environment

- Reduction combines values from threads.
  - `reduction(op : variable-list)`
  - Variables in the list must be shared in the enclosing parallel region.
  - Each thread initially makes a local copy of each list variable and updates it.
  - Local copies are reduced into a single global copy at the end of the construct.
  - More efficient than using a critical section.

```
#pragma omp parallel for
        reduction (+ : x)
for (i=0; i<n; i++) {
    x = x + a[i];
}
#pragma omp parallel for
for (i=0; i<n; i++) {
    #pragma omp critical
    {
        x = x + a[i];
    }
}
```

Runtime Execution

- Runtime functions
  - Locks: `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`.
  - Runtime environment routines: `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_get_thread_num()`, `omp_num_procs()`.
- Environment variables
  - Behavior of `omp for schedule(RUNTIME)`.
  - `OMP_SCHEDULE` schedule[, chunk_size].
  - Set the default number of threads to use: `OMP_NUM_THREADS` n.