Candidates for Parallelism

Some ways to achieve parallelism.

- **Data Parallelism:** Divide things so that the same task can be run on different data in parallel.
  - Important for there to be no dependencies (eg: one copy of the task can’t alter data being worked on by another copy of the task).

- **Task Parallelism:** Divide things do that different tasks can be run on the same data or on different data in parallel.
  - Consider things that don’t alter the data at all, like finding the average of a list versus the median of a list.

- **Pools of work:** Different tasks and/or different data, set up to be ready for the next available processing element to do some work.
  - The finer the granularity the better the potential for efficient use of parallelism.
**Some terminology for issues (mostly from JCIP)**

Throughput: the rate at which a set of concurrent tasks is completed.

Responsiveness: the delay between a request for and completion of some action (also called latency).

Scalability: the improvement in throughput (or lack thereof) as more resources (usually CPUs) are made available.

Graceful Degradation: how a system should act if the system becomes overloaded, rather than simply falling over under heavy load.

---

**Go for the low-hanging fruit first...**

There are a few “quick and easy” places where you might be able to take advantage of the naturally-available parallelism in an algorithm.

- Recursion: Whenever you have a divide & conquer algorithm that is naturally presented via independent recursive calls, you have a target for concurrency.

- Loops: If each iteration of a loop is independent of the others, you have a target for concurrency.
Parallelism Candidate: Recursive Call⇒Task

One programming approach that lends itself very well to concurrent tasks is divide and conquer recursion. If there are multiple recursive calls from a branch point, and their work is independent of each other, each of those could be turned into a new task to be executed concurrently.

If you have an giant pool of processing elements, this might work well. However, if you have a limited number (such as 4 cores on a personal computer) the time that you spend creating and queuing tasks or that threads spend causing each other to be swapped out of the CPU might significantly eat into, or even outweigh, your savings from the concurrency.

Recursion: Too much or too little of a good thing?

If you have set a limited number of threads based on the number of processors and dependencies between the levels of recursion, you risk deadlock. Imagine MergeSort where a task is set to create and submit two new MergeSort tasks to the pool, wait for them to complete, and then Merge their results. With a fixed size thread pool, the later tasks will not have threads in which to run and the earlier tasks won’t give up their threads until the later tasks complete.

We could write our code so that once we are running out of threads, we stop putting recursive calls into new tasks and rather run them “normally” in the thread that has them. This would avoid the previous problem but if we write our code so that once we are running out of threads, we stop putting recursive calls into new tasks and rather run them “normally” in the thread that has them, as those tasks do complete, we no longer have a way to take advantage of the newly-available threads (and processors).
**Parallelism Candidate: Sequential Threshold**

With many optimization problems there will be a threshold beyond which it’s better to do things the “simple” way rather than the more “complex” way. For example, it turns out that if your list is smaller than $8$ elements, you are statistically better off using `InsertionSort` rather than `QuickSort`.

The same is true with concurrent programming; you will reach a point where the cost of the sequential solution is less than the cost of the concurrent solution due to the overhead associated with it.

These can be (and are) found either through proofs or through empirical explorations, and they will often vary based on the specific architecture of a machine.

Our `Quicksort` could end up creating $N/2$ different tasks!

---

**Parallelism Candidate: Iteration≡Task**

With independent loops, there are two general approaches to consider (the first is used by the second).

Every iteration of the loop could be spawned off as its own task.

```java
for (int i=0; i<size; i++) {
    process element i
}
```

could become

```java
for (int i=0; i<size; i++) {
    submit new parallel task {process element i}
}
```

after which you would wait for them to all finish.

However, with this approach the processing of each element needs to be substantial and should have little or no synchronization required. This approach would not have helped us much with parallel max finding since the tasks would all bunch up waiting for the shared ‘max’ field.
Parallelism Candidate: Unrolling

Generally speaking, we could **unroll** the loops, and rewrite

```java
for (int i=0; i<size; i++) {
    process element i
}
```

as

```java
for (int i=0; i<CPUs; i++) {
    process element i*(size/CPUs) \rightarrow (i+1)*(size/CPUs) - 1
}
```

or possibly as

```java
for (int i=0; i<size/CHUNK; i++) {
    process element i*CHUNK \rightarrow ((i+1)*CHUNK) - 1
}
```

which would be more likely to make concurrent use of the processors relative to the overhead of spawning off new tasks (depending on the system’s architecture and tweaking of the size of CHUNK).

We would have to special-case the last chunk if the size wasn’t evenly divisible, but we would still get $\text{CPUs} - 1$ processors being well-utilized.

---

Parallelism Candidate: Servers (I)

Many approaches... You could have each **service** handled by a server machine running in its own thread:

```java
class SingleThreadWebServer { //JCIP Listing 6.1
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }
}
```

It works, but it would only handle one request at a time. For a busy web server, this would not be desirable or practical.

Even with only a single processor, a web request might use the network connection, the hard drive(s), and the processor (any of which could block, but which also provides opportunities for parallelism).
Parallelism Candidate: Servers (II)

Many approaches... You could have each request handled by a server machine running in its own thread:

class ThreadPerTaskWebServer { //JCIP Listing 6.2
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }
}

It works, would have good responsiveness, could give better throughput if there are multiple processors or blocking on IO, but it would not scale well or tend to exhibit graceful degradation. Thread creation and termination is costly in terms of time and resources. This could easy consume all available stack space. For a busy web server, this would not be desirable or practical.