Value Speculation

The other kind of speculation is value speculation. In this case, there is a (true) dependency between the result of a computation and its successor:

```
void doWork(int x, int y) {
  int value = longCalculation(x, y);
  return secondLongCalculation(value);
}
```

If the result of `value` is predictable, then we can speculatively execute `secondLongCalculation` based on the predicted value. (Most values in programs are indeed predictable).

```
void doWork(int x, int y) {
    thread_t t1, t2;
    point p(x,y);
    int v1, v2, last_value;
    thread_create(&t1, NULL, &longCalculation, &p);
    thread_create(&t2, NULL, &secondLongCalculation,
                  &last_value);
    thread_join(t1, &v1);
    thread_join(t2, &v2);
    if (v1 == last_value) {
      return v2;
    } else {
      last_value = v1;
      return secondLongCalculation(v1);
    }
}
```

Note that this is somewhat similar to memoization, except with parallelization thrown in. In this case, the original running time is

$$T = T_1 + T_2,$$

while the speculatively parallelized code takes time

$$T_s = \max(T_1, T_2) + S + pT_2,$$

where S and p are the same as above.

Exercise. Do the same computation as for speculative execution.

When can we speculate?

Speculation isn't always safe. We need the following conditions:

- `longCalculation` and `secondLongCalculation` must not call each other.

- `secondLongCalculation` must not depend on any values set or modified by `longCalculation`.

- The return value of `longCalculation` must be deterministic.

As a general warning: Consider the *side effects* of function calls.

Critical Paths

You should be familiar with the concept of a critical path from previous courses; it is the minimum amount of time to complete the task, taking dependencies into account, and without speculating.

Consider the following diagram, which illustrates dependencies between tasks (shown on the arrows). Note that B depends on A, and D depends on B and C, but C does not depend on anything, so it could be done in parallel with everything else. You can also compute expected execution times for different strategies.
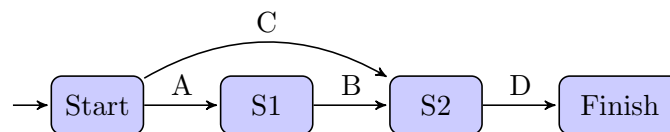
Data and Task Parallelism

There are two broad categories of paralellism: data parallelism and task parallelism. An analogy to data parallelism is hiring a call center to (incompetently) handle large volumes of support calls, *all in the same way*. Assembly lines are an analogy to task parallelism: each worker does a *different* thing.

More precisely, in data parallelism, multiple threads perform the *same* operation on separate data items. For instance, you have a big array and want to double all of the elements. Assign part of the array to each thread. Each thread does the same thing: double array elements.

In task parallelism, multiple threads perform *different* operations on separate data items. So you might have a thread that renders frames and a thread that compresses frames and combines them into a single movie file.

We'll continue by looking at a number of parallelization patterns, examples of how to apply them, and situations where they might apply.

Data Parallelism with SIMD

We'll talk about single-instruction multiple-data (SIMD) later on in this course, but here's a quick look. Each SIMD instruction operates on an entire vector of data. These instructions originated with supercomputers in the 70s. More recently, GPUs; the x86 SSE instructions; the SPARC VIS instructions; and the Power/PowerPC AltiVec instructions all implement SIMD.

Code. Let's look at an application of SIMD instructions.

```
void vadd(double * restrict a, double * restrict b, int count) {
  for (int i = 0; i < count; i++)
    a[i] += b[i];
}
```

Compiling this without SIMD on a 32-bit x86 (`gcc -m32 -march=i386 -S`) might give this:

```
loop:
  fldl  (%edx)
  faddl (%ecx)
  fstpl (%edx)
  addl  8, %edx
  addl  8, %ecx
  addl  1, %esi
  cmp   %eax, %esi
  jle   loop
```

We can instead compile to SIMD instructions (`gcc -m32 -march=prescott -mfpmath=sse`) and get something like this:

```
loop:
  movupd (%edx),%xmm0
  movupd (%ecx),%xmm1
  addpd  %xmm1,%xmm0
  movpd  %xmm0,(%edx)
  addl   16,%edx
  addl   16,%ecx
  addl   2,%esi
  cmp    %eax,%esi
  jle    loop
```

The *packed* operations (`p`) operate on multiple data elements at a time (what kind of parallelism is this?) The implication is that the loop only needs to loop half as many times. Also, the instructions themselves are more efficient, because they're not stack-based x87 instructions.

SIMD is different from the other types of parallelization we're looking at, since there aren't multiple threads working at once. It is complementary to using threads, and good for cases where loops

operate over vectors of data. These loops could also be parallelized; multicore chips can do both, achieving high throughput. SIMD instructions also work well on small data sets, where thread startup cost is too high, while registers are just there to use.