of interaction among the workers. Manufacturing pipelines are different still, because they generally have strict ordering constraints with the separate stages often being performed sequentially; the parallelism comes from having many instances of the product in the pipeline at once. And juggling is an instance of event-driven parallelism, where an event—a falling ball—causes the execution of operations—catching, throwing—in response to the event. Such familiar forms of parallelism will also arise in our consideration of parallel computation.

Parallelism in Computer Programs

The main motivation for executing program instructions in parallel is to complete a computation faster. But most programs today are incapable of much improvement through parallelism, because they were written assuming that the instructions would be executed in order, one at a time, that is, *sequentially*. The semantics of most programming languages embed sequential execution, and the resulting programs typically rely so heavily on this property for their correctness that it is rare to find significant opportunities for parallel execution. To be sure, there are some opportunities, as when the expression `(a+b)*(c+d)` must be evaluated; assuming these are simple variables, the subexpressions `(a+b)` and `(c+d)` are independent of each other, so they can be computed simultaneously. Such opportunities are an example of Instruction Level Parallelism (ILP).

Indeed, one reason that we have continued to write sequential programs is because computer architects have been so successful at exploiting parallelism. They have used the steady improvements in silicon technology to add several kinds of parallelism, including ILP, into sequential processor design. First, architects provide separate wires and caches for instructions and data. The separation allows instruction and data memory references to execute in parallel without interfering. Second, instruction execution is pipelined, fetching and decoding future instructions while the current instruction is being executed and while the results of past instructions are still being written to memory. Furthermore, the processors issue (initiate) more than one instruction at a time, they prefetch instructions and data, they speculatively perform operations in parallel even if they cannot be sure that they will be needed, and they use highly parallel circuits to perform basic arithmetic operations. In short, modern processors are highly parallel systems.

The key point for programmers is that all of this parallelism has been transparently available to sequential programs. We call this *hidden parallelism*. Such parallelism, together with increasing clock speeds, has allowed each succeeding generation of processor chip to execute programs faster, while preserving the illusion of sequential execution. But the prospects for finding new opportunities to apply parallelism while preserving sequential semantics are becoming limited. More seriously, existing techniques for exploiting ILP have largely reached the point of diminishing returns, in terms of both power consumption and performance. So, given current