Managing Loops and Data Synchronization Are Keys to Successful Parallelization

Bryon Moyer, Vector Fabrics, www.vectorfabrics.com

Parallelization of a program is simple in concept but can be more complex in actual implementation. It also depends on the application. Some applications lend themselves to parallelization more than others. Good examples are those that employ a high proportion of individual computational tasks. These can often be found in factory automation applications where a mechanical action is dependant on the recognition of an image. Likewise, CCTV applications can employ fairly complex algorithms to detect and differentiate movements of people and vehicles.

In practice, parallelization always involves loops in one way or another. While it is theoretically possible for an algorithm to involve multiple independent long chains of calculations that don’t involve loops, it’s highly unlikely in real life.

The loops involved may result from algorithmic requirements – for example, the traversing of matrices; they may come from the need to process more data than can be acquired from a single operation, such as reading and processing all of the data in a file; or they can result from a task processor that must repeatedly check whether there is data that needs to be worked, such as a packet processor looking for new packets that have arrived on the network.

It is the repetitive nature of loops that takes what appear to be small tasks and repeats them so many times that they accumulate to a large compute load. It is for this reason that loops are typically of interest when finding ways to parallelize a program to share the load over multiple processing units.

There are two fundamental ways to parallelize data: through fork/join configurations or through loop distribution. Theoretically, using fork/join to parallelize doesn’t require a loop, but, in practice, it only makes sense when done inside of a loop so that the savings accrue over all of the loop iterations.

Fork/Join

With fork/join parallelization, two (or more) tasks are spawned to run in parallel within one iteration of a loop. Such parallelization is most effective if the tasks are

balanced in compute load (Figure 1). If not, then the faster tasks end up waiting for the longest task to complete (Figure 2).

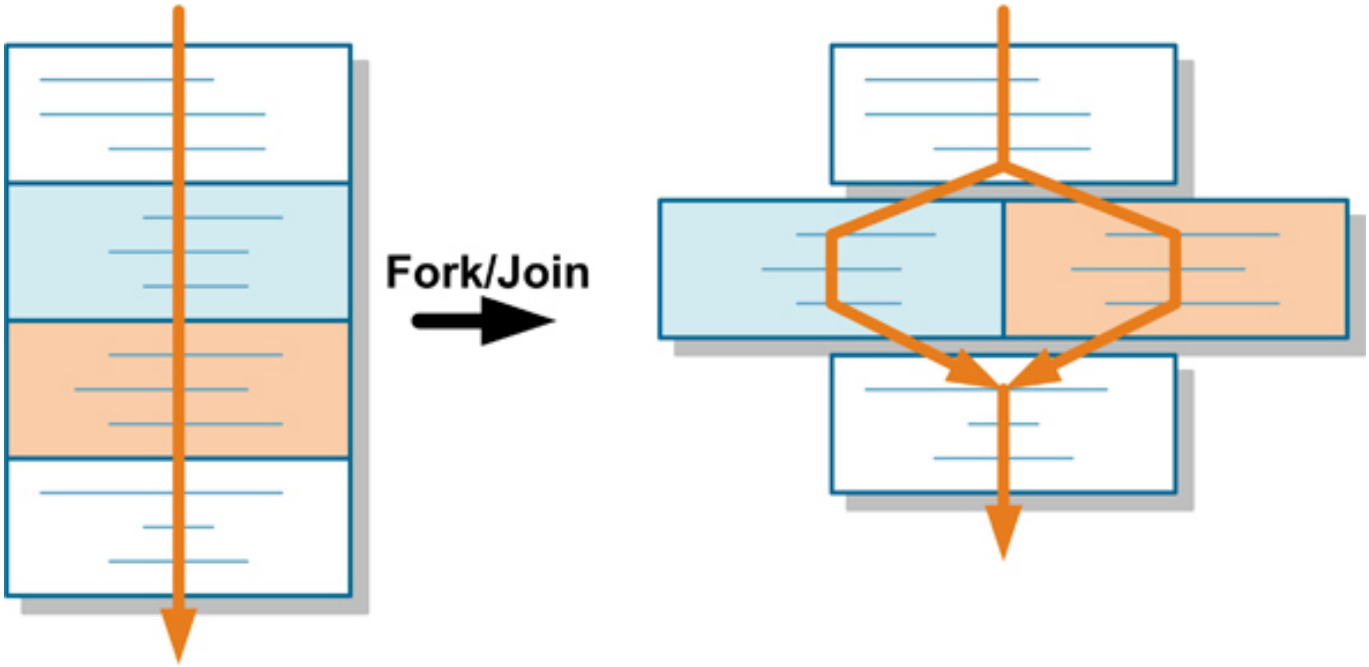

Figure 1. Balanced fork/join parallelization

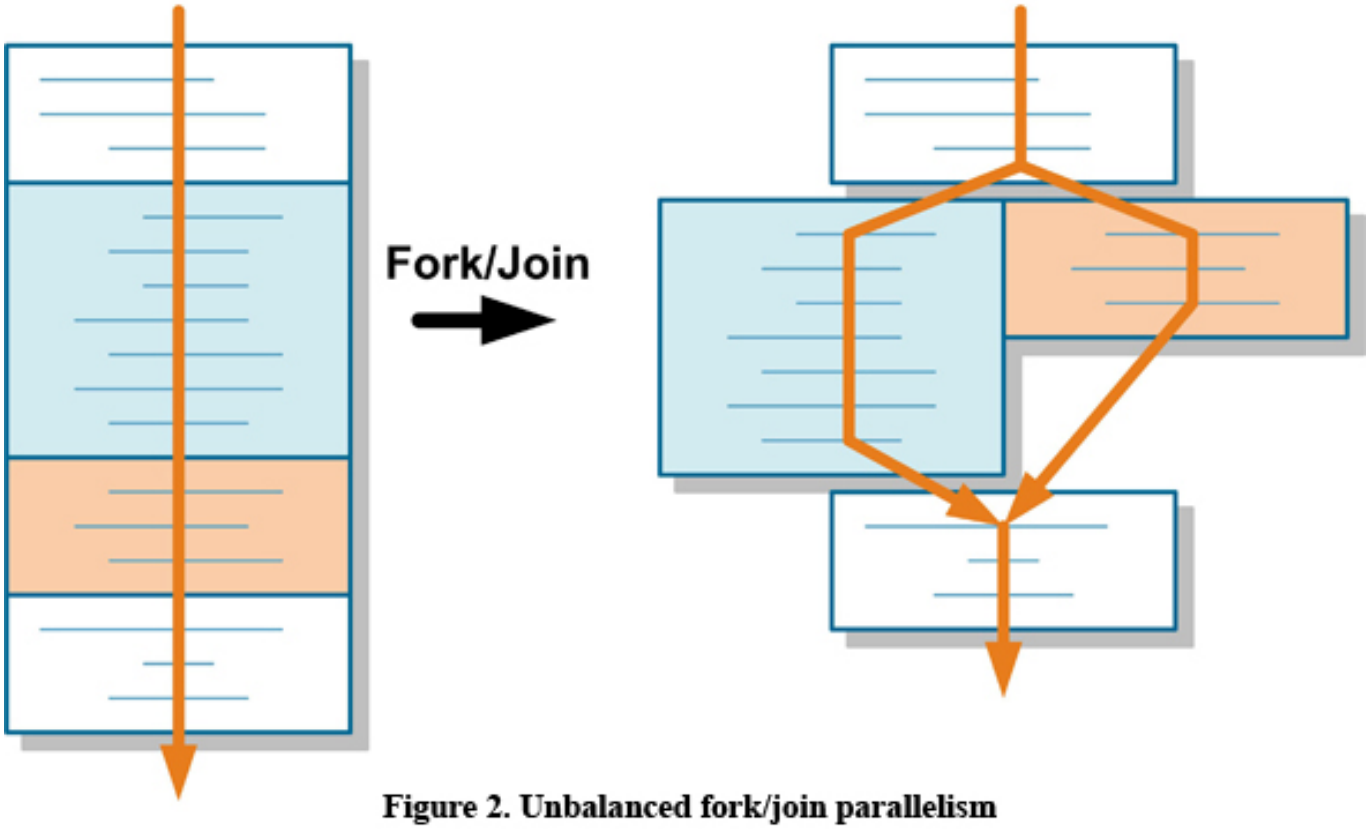

Figure 2. Unbalanced fork/join parallelism

For standard sequential C programs, it is simplest if there is no need to communicate data between the tasks. Unfortunately, this scenario is not common enough to handle most parallelization requirements.

Loop Distribution

When a single iteration of a loop involves a large amount of computing, it is common to split the loop into two (or more) tasks and assigning them to different processors. In the case of two tasks, the second half of the task cannot start until the first half has completed, but once the second half starts, the first half can start on its next iteration while the second task finishes the first iteration. Effectively, the first task gets part of the work done and then hands it off to the second task. In the hardware world, this is known as “pipelining.” However, in theory, if the second half were completely independent of the first half, not relying on any data generated in the first half, then it could start immediately, as in Figure 3, with no (or negligible) added latency.

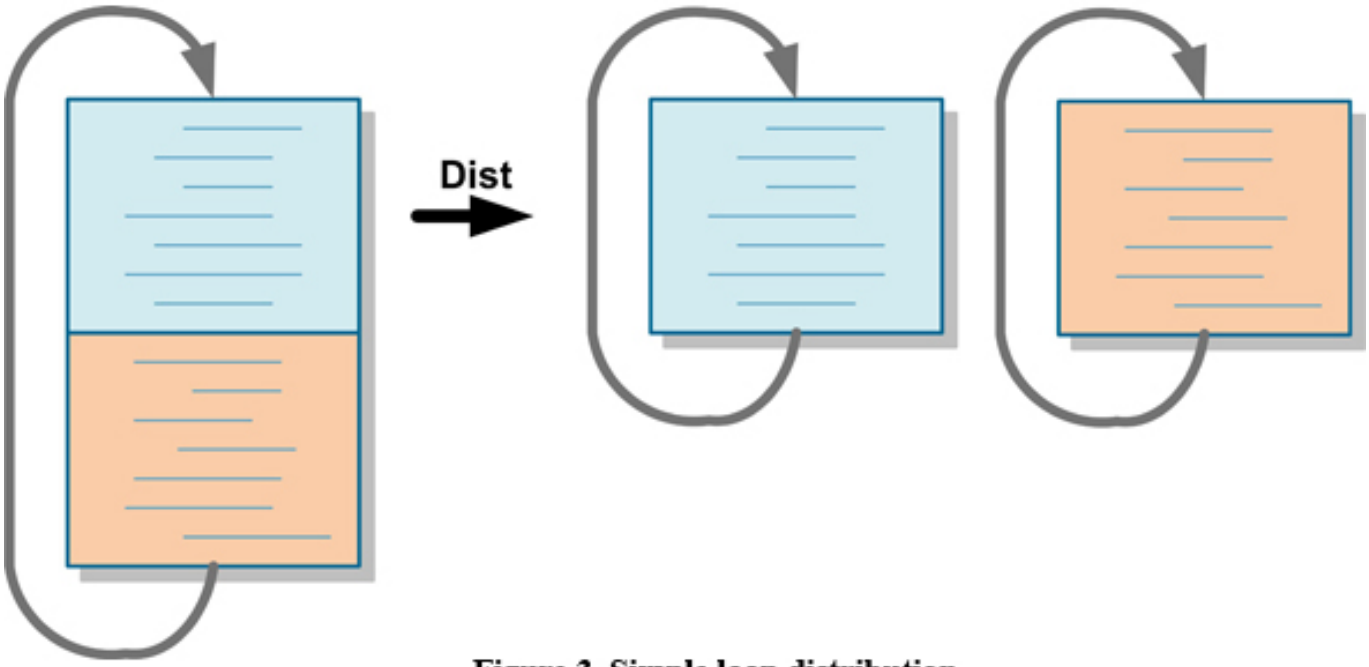

Figure 3. Simple loop distribution.

Loop distribution requires attention to where the loop body is split. If data is produced before the split and consumed after the split, then that data must be communicated – or synchronized – between tasks after the loop is distributed. The second part of the loop can’t start until all the necessary data is available, adding some latency. This is shown in Figure 4, with the “P” triangles indicating data production and the “C” triangles indicating data consumption.

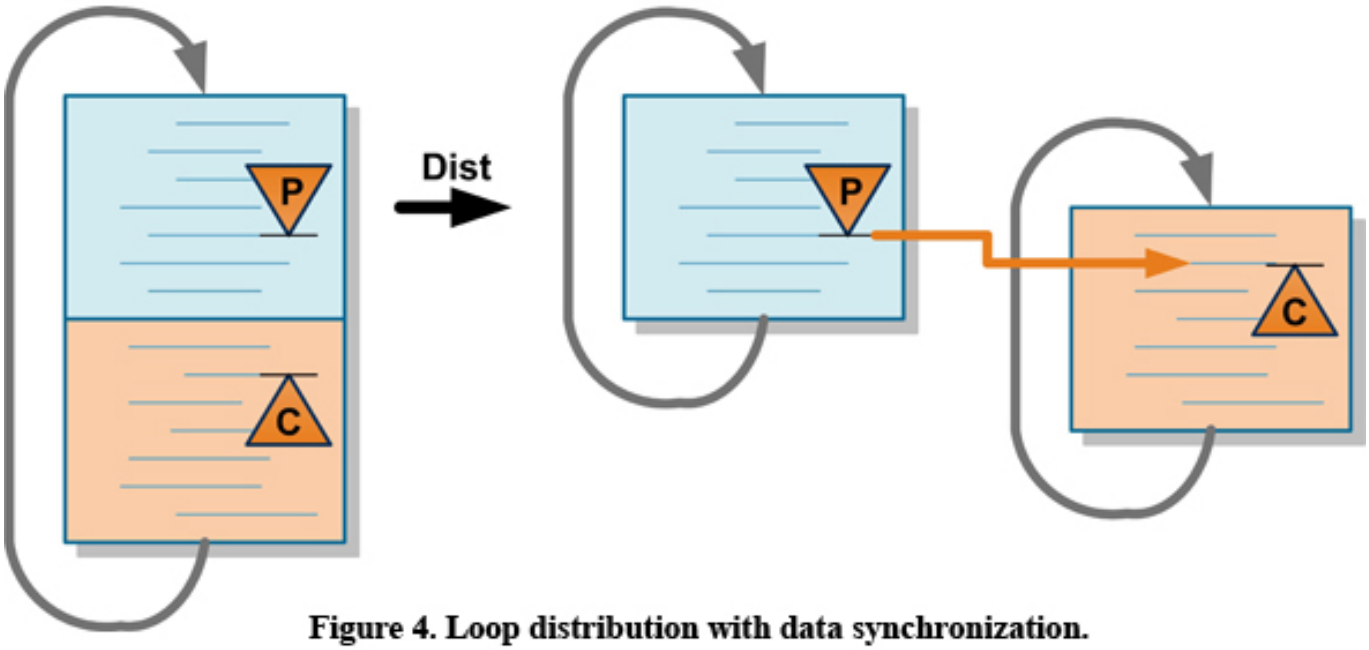

Figure 4. Loop distribution with data synchronization.

Loops can be completely parallelized by a combination of unrolling and distribution. Loop unrolling refers to taking the loop body and explicitly duplicating it within the loop, reducing the number of loop iterations accordingly. For example, if a loop has 16 iterations and you unroll four times, then the loop body now has four explicit copies of the original loop body, meaning the resulting loop only executes four times, as shown in Figure 5.

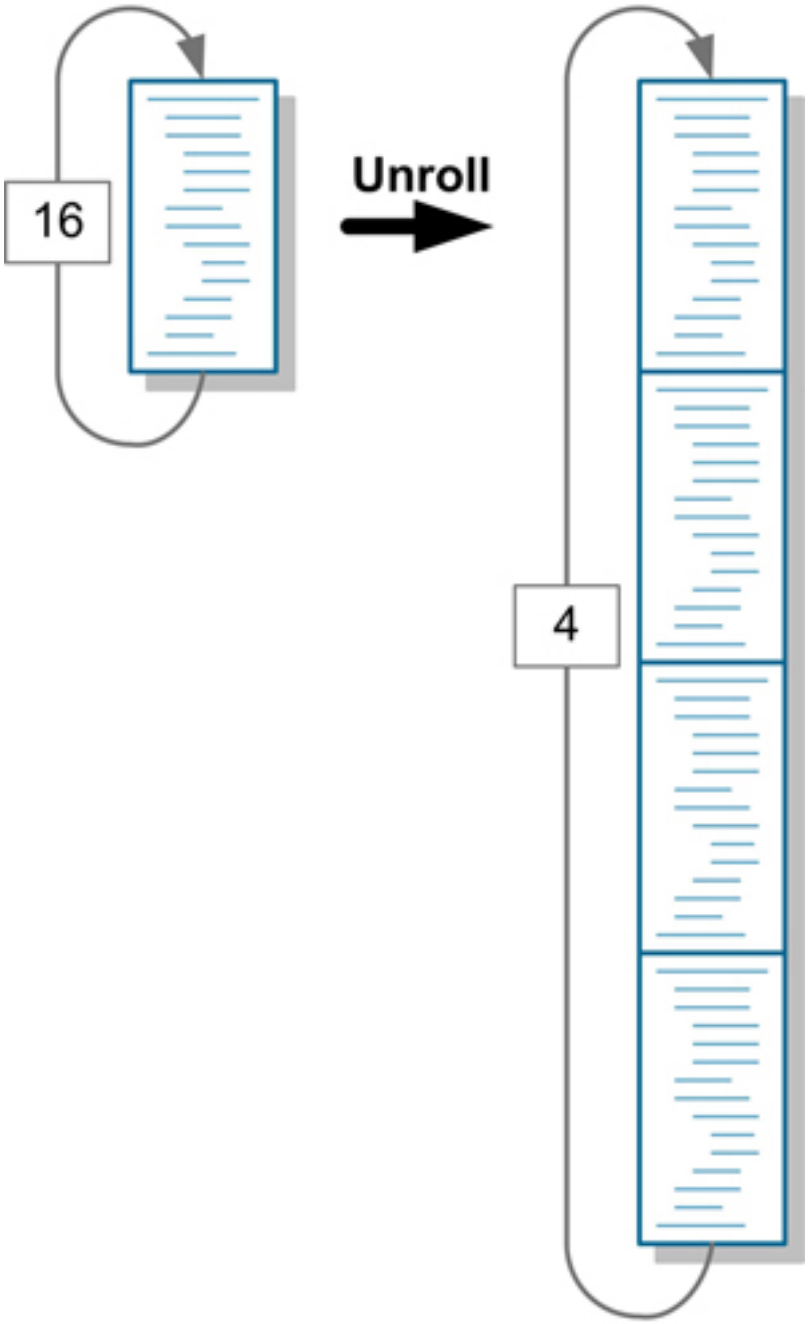

Figure 5. Loop unrolled four times.

As an example of loop parallelization, two processors can split up the number of loop iterations by unrolling once and then distributing. In this way, the first processor takes the odd iterations and the second processor takes the even ones; this is illustrated in Figure 6.

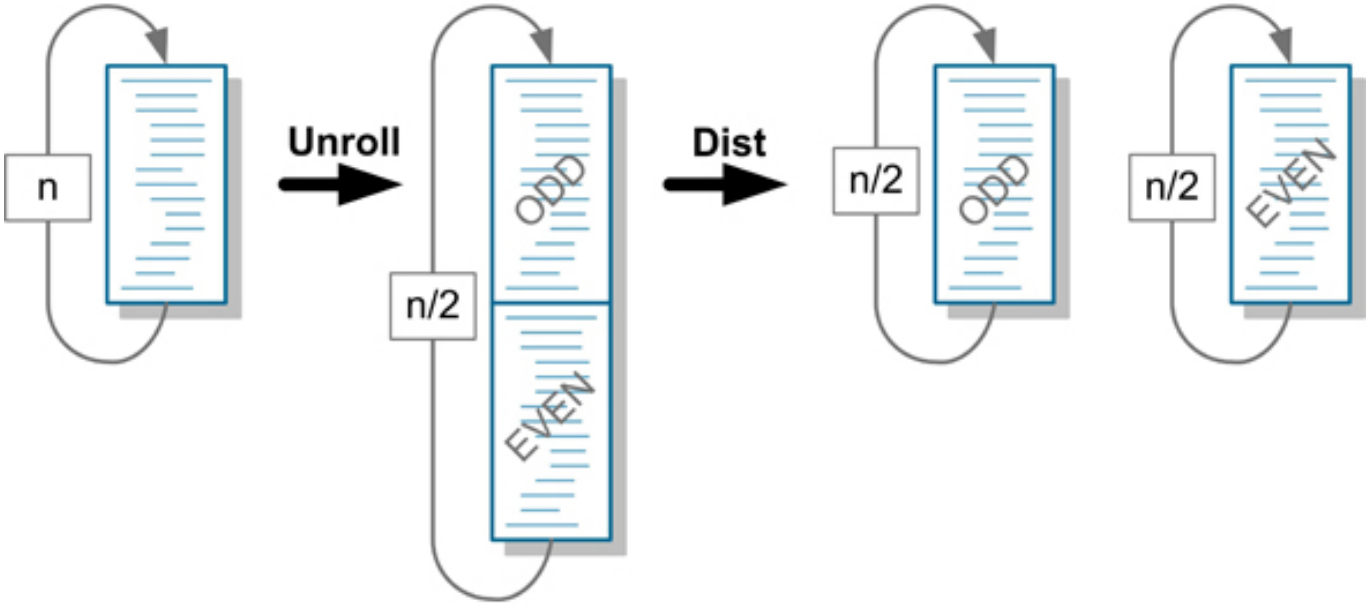

Figure 6. Loop parallelization by unrolling and distribution

If the loop is completely unrolled and there are as many processors as iterations, then all iterations can be executed in parallel.

Loop-carry Dependencies and Synchronization
The degree of parallelization possible depends on how data needs to be communicated between iterations. This is the same synchronization issue as discussed for simple loop distribution, only applied to unrolled loops, where each copy of the loop body represents an iteration of the original loop. Dependencies between one loop iteration and another are referred to as loop-carry dependencies. Loop-carry dependencies require synchronization if parallelizing across them.

Synchronization involves the communication of data from one task to another. There are a variety of ways to implement it – including dedicated hardware – so there is no one right or wrong way. Each carries a level of overhead that may be high or low, a consideration to be taken into account when making parallelization choices.
Depending on the program, synchronization might happen more or less frequently. Higher frequency indicates a higher level or “granularity” of parallelization; infrequent synchronization indicates “chunkier” parallelization.

For example, if a simple loop is distributed and a single value is calculated that must be synchronized, then synchronization will occur with each loop iteration for a small piece of data (Figure 7).

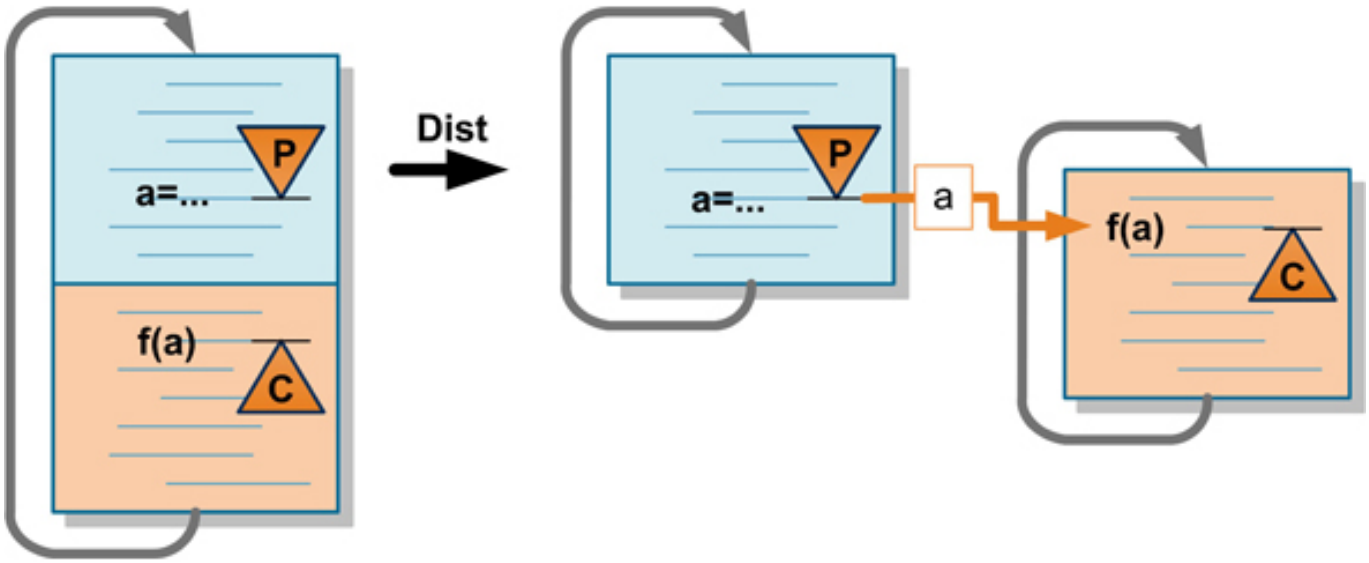

Figure 7. Synchronizing a single variable.

Nested loops provide a counter-example. If the loop being distributed contains two loops, one generating data and one consuming the generated data, and if the loop is split between those two inner loops, then one task will produce a number of data points; the other task will consume them.

Synchronization will not occur until the entire inner loop has finished generating data. So if the inner producing loop in the first task iterates 16 times and fills an array with 16 values, then synchronization will transfer those 16 values to the second task, whose inner loop will then iterate 16 times to read them (Figure 8).

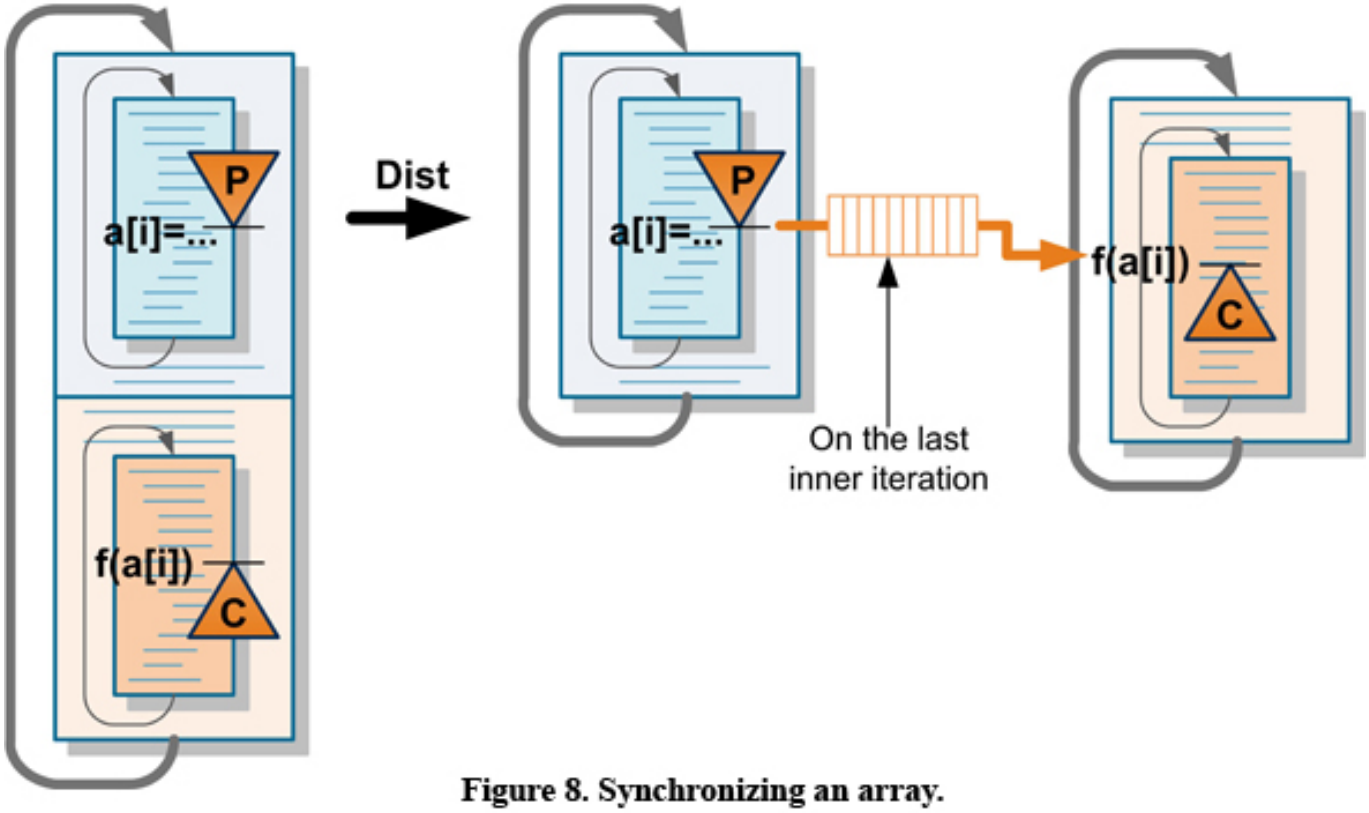

Figure 8. Synchronizing an array.

To the extent that one task is waiting on another for data, the more data transferred at a time, the longer the waiting task has to sit around until the data is ready. So more things can be done in parallel more quickly if data is transferred smaller chunks at a time.

However, synchronization does involve some overhead. The overhead may be as small as a simple indicator that data is available, or may involve locking and unlocking data structures and copying data. This overhead directly leads to the latency introduced by parallelization. Therefore, the more overhead there is for each synchronization, the more data you want to transfer in each synchronization. If you have zero to little overhead, then data can be synchronized more frequently with less penalty.

The dependency properties include the number of synchronizations (“syncs”) and the amount of data per synchronization. Multiplied together, these give you the total data synchronized.

In general, given various parallelization choices, it’s better to transfer less data than more. In a tie, there’s a tradeoff between the latency introduced by doing fewer synchronizations with more data each time and the overhead introduced by doing more synchronizations with less data each time. When you explore the different ways you can partition your program,

Until recently, attempting to identify potential ways to parallelize an application has required a huge amount of time. However, new cloud-based tools, such as vfAnalyst from Vector Fabrics, are making this onerous task much easier. By uploading your source code to the cloud service the tools can quickly identify the areas most likely to benefit from parallelizing. vfAnalyst for example, presents this visually and aids developers focus their efforts on the sections that will give a performance improvement rather than those parts of the application that cannot be parallelized.

Source URL (retrieved on *06/20/2013 - 2:11am*):
http://www.ecnmag.com/articles/2011/01/managing-loops-and-data-synchronization-are-keys-successful-parallelization