

Using vfAnalyst To Parallelize Your Program

vfAnalyst is a tool that provides guidance for engineers parallelizing their sequential C code. It analyzes concurrency and data dependencies so that a correct multi-threaded program can be created. If done incorrectly, a parallel program will, at best, not work identically to its sequential version. At worst, it will simply not work, and the bugs will be elusive and take a long time to identify. vfAnalyst guides the user around these pitfalls to ensure that the resulting parallel program is semantically identical to the sequential original.

This paper provides an example of how vfAnalyst can be used. The example is described first in order to motivate usage of the tool. More background discussion on the nuts and bolts of parallelization is then provided for those looking for a deeper analysis of how parallelization is achieved and for how the strategies found using vfAnalyst can be implemented.

This whitepaper makes reference to various kinds of data dependencies as well as various aspects of the vfAnalyst user interface. If you are not familiar with dependencies or the vfAnalyst GUI, you can find more information in the [vfAnalyst whitepaper](#).

A parallelization example

In this section, we take a specific program and show how it can be parallelized to improve its performance. This is a real-world program, and, while not huge, at about 2000 lines of code, the parallelization we show will be non-trivial and non-obvious. What's particularly powerful about the process is the fact that no detailed knowledge of the code being parallelized is needed.

The methodology focuses on loops and uses three steps:

1. Identify a high-delay loop. In general, you would pick the highest-workload loop, but if there is a deep call stack with more or less the same delay, you want to go deeper into the call stack to find a suitable number of partitioning options.
2. Ask vfAnalyst to show you what partitions are available for that loop.
3. Select the desired partition according to the data synchronization cost; the resulting performance improvement will be reflected in the display, and the relative width of the invocation you chose will shrink accordingly.

The cost of data synchronization remains generic in vfAnalyst, but can be more specific in vfSoftware. In general, a partition that requires less data communication as possible is preferred over one requiring more. Given options with similar amounts of data, you have a tradeoff between synchronization overhead and latency. More frequent synchronization of small amounts of data means less delay while waiting for data and therefore less latency. But

synchronization requires overhead. Too much synchronization can actually slow the system down due to that overhead. Synchronization is discussed in more detail below.

We'll start by looking for the loop with the highest delay. *Loop_585* was selected in Figure 1. Although it's not the longest loop, the others are too far up the call chain to provide useful partitions.



Figure 1. Step 1

Having chosen this as our highest-delay invocation, we now right-click and ask for a list of the best partitioning options by selecting “Optimize loop”. The options are presented in a brief table as shown in Figure 2.

Project	Profile	Partitions			
Partition id	Threads	Speedup	Streams	Apply	
Partition 1	2	1.4	4	Apply	

Figure 2. Step 2

The benefits and costs of each option are included in the table. The costs are currently simply expressed in terms of the number of streams that must be broken to execute the partition; these streams must be replaced by communication channels in the parallelized version. Only one option is provided in the table above, creating two threads that cross four streams for

Using vfAnalyst To Parallelize Your Program

loop performance that's 1.4 times faster than the sequential loop. We can now click "Apply" next to that partition to have vfAnalyst create the parallel version, which is shown in Figure 3. Note that, although we got a loop speedup of 1.4x, the speedup to *main()*, that is, the whole program, is 1.14x since only that one loop has been accelerated.



Figure 3. Step 3

This same process is then repeated on what is now the highest-delay loop (which happens to be in *Loop_702*), and continue this process until either we have reached our goal or we have reached the maximum level of parallelism that the program allows. It's illustrative to see the options available for the next loop to be parallelized. Diving deeper yields a long loop in function *IDCT()*, and three partitioning options are available, as shown in Figure 4. Two provide a speedup of 1.9x, one provides a speedup of 1.3x. Of the two faster options, one creates 3 threads, the other two threads. They all cross two streaming dependencies. In general, fewer threads are better, and so the third partition would be selected if 1.9x were needed.

Project	Profile	Partitions			
Partition id	Threads	Speedup	Streams	Apply	
Partition 1	3	1.9	2	Apply	
Partition 2	2	1.3	2	Apply	
Partition 3	2	1.9	2	Apply	

Figure 4. Step 2 for a loop with more partitioning options

Note that this specific methodology is not required; it's simply a straightforward, deterministic way to parallelize a program that you know little or nothing about. Even though the acceleration of a specific invocation may appear nominal, when accumulated, significant speedups can be achieved. Given greater knowledge and experience, you could go to any invocations in any order to target specific parallelization strategies. You may even identify partitioning opportunities that vfAnalyst didn't find by examining the dependencies: the only abiding rule is that partitions occur only across compute, streaming, and anti-dependencies.

Parallelization background

Parallelization of a program is simple in concept but can be more complex in actual implementation. The purpose of this section is to present the essential elements of parallelization and add a bit more rigor to the example presented above.

While a thorough understanding of these background topics is not needed in order to make effective use of vfAnalyst and vfSoftware, it will help you understand how to make the most effective partitioning decisions.

It's almost always about loops

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 5). If not, then the faster tasks end up waiting for the longest task to complete (Figure 6).

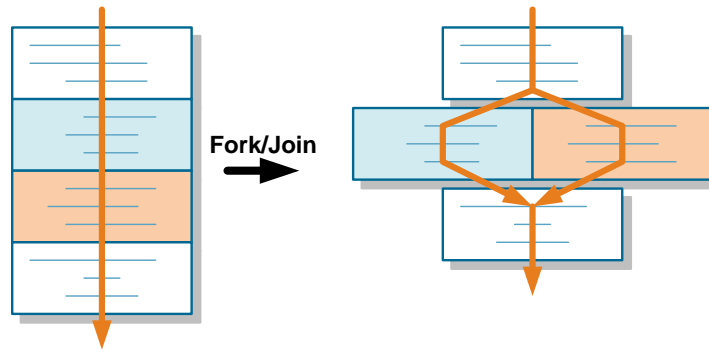


Figure 5. Balanced fork/join parallelization

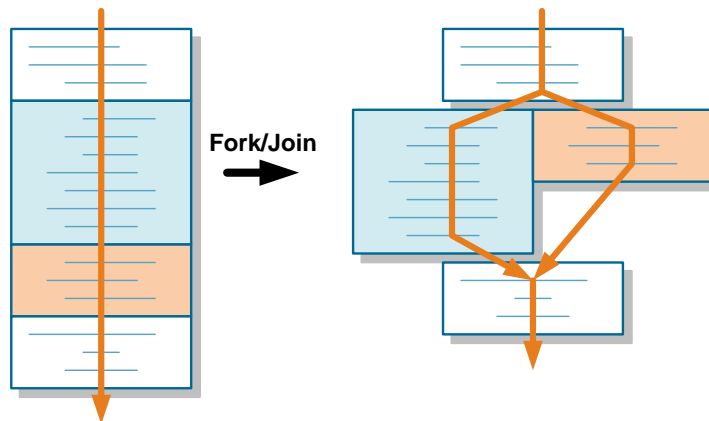


Figure 6. Unbalanced fork/join parallelism

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. vfAnalyst will show such opportunities as having no dependencies between the invocations being forked.

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 7, with no (or negligible) added latency.

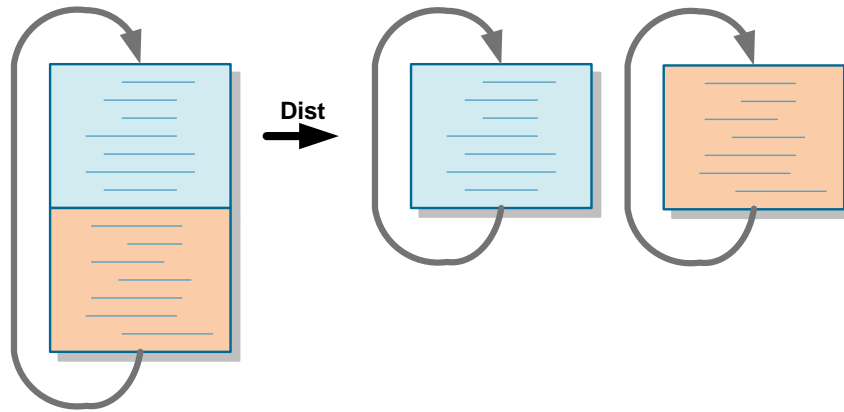


Figure 7. Simple loop distribution

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 8, with the “P” triangles indicating data production and the “C” triangles indicating data consumption.

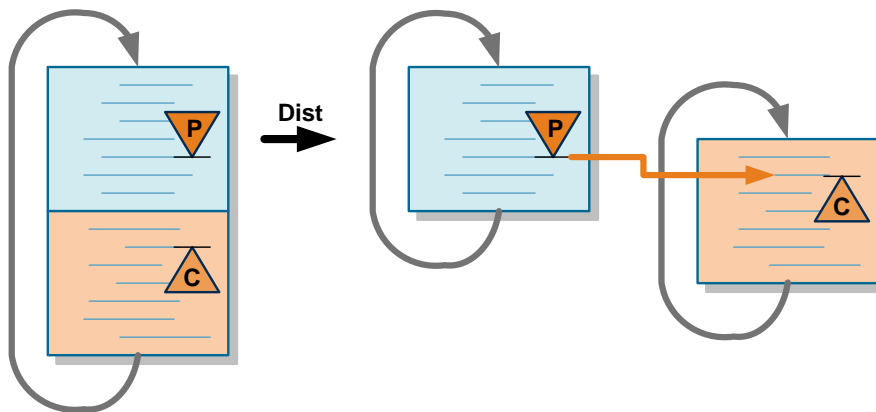


Figure 8. Loop distribution with data synchronization

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 9.

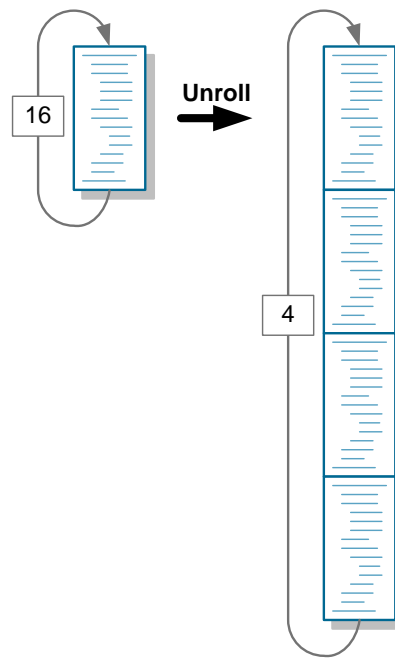


Figure 9. Loop unrolled 4 times

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 10.

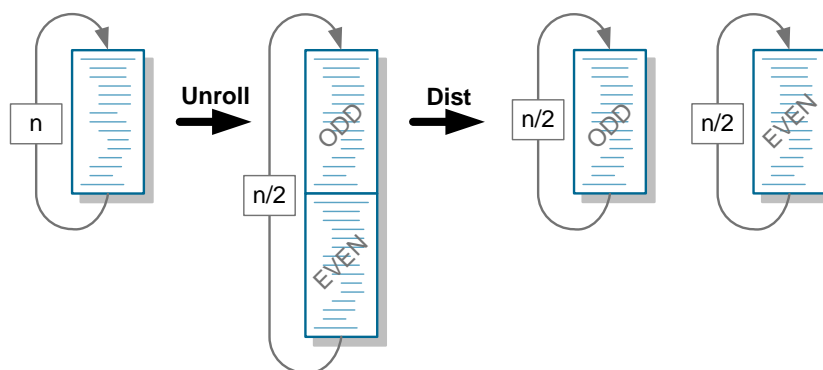


Figure 10. Loop parallelization by unrolling and distribution

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

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.

The parallelization implications of loop-carry dependencies are described in more detail in the [vfAnalyst whitepaper](#).

Synchronization

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 11).

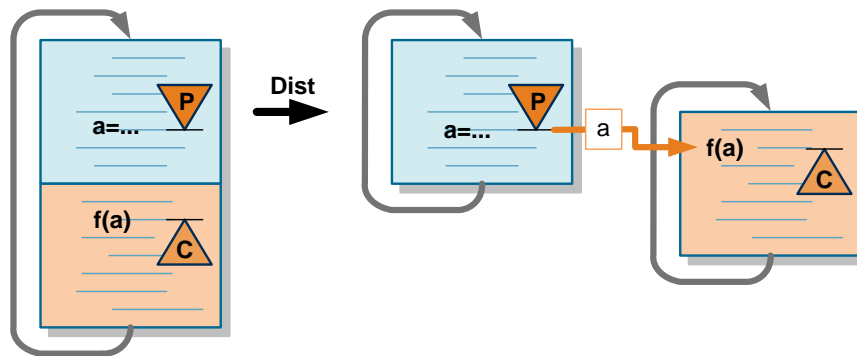


Figure 11. Synchronizing a single variable

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 12).

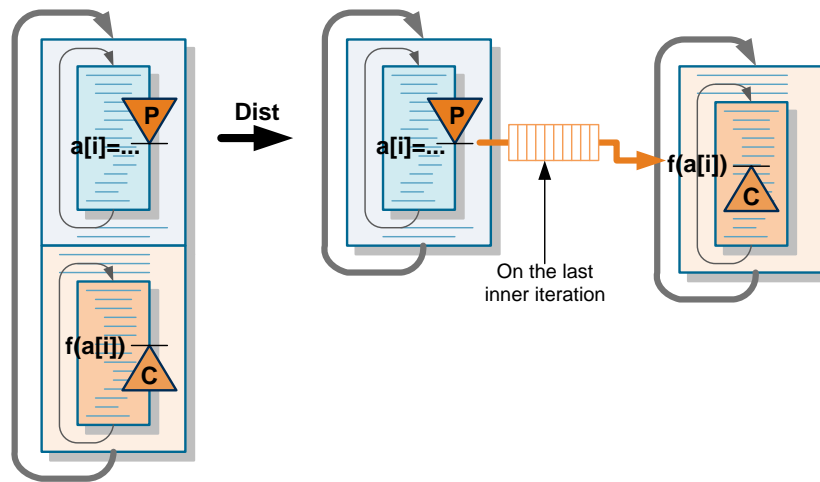


Figure 12. Synchronizing an array

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.

vfAnalyst is able to identify various data synchronization patterns, the simplest of which is the FIFO streaming pattern (for other patterns, please refer to the [document describing the recognized patterns](#)). These are depicted as streaming dependencies, and the properties of the dependency tell you what kind of data would have to be synchronized if you were to parallelize across that dependency.

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, vfAnalyst will show the generic synchronization cost so that you can pick the one that best meets your overall performance needs.

Handling data dependencies

As seen above, dependencies may create the need to synchronize data. But some dependencies require handling in different ways, and certain ones may actually block parallelization. The following discusses the impact of the various kinds of data dependencies

as recognized by vfAnalyst. For more information on these dependencies, please refer to the [vfAnalyst whitepaper](#).

Streaming dependencies

Streaming dependencies can always be managed by providing the appropriate synchronization as discussed above as long as all of the writes are on one side of the partition and all of the reads are on the other side. vfAnalyst will only recommend partitions that meet this condition. Some other configurations could be managed, but add synchronization complexity and overhead, and are best avoided.

Compute dependencies

It is usually possible to parallelize across compute dependencies, but the affected variables must also be synchronized. This ensures that there is no semantic change in the functioning of local variables. Synchronizing compute dependencies adds to the cost of parallelization.

Memory dependencies

Memory dependencies are more complicated. The easiest approach is simply not to parallelize across them. If that's not an option, then they must be considered case by case. In general, the semantics involved will change, which may or may not be acceptable. For example, when writing to a file, a sequential program will write in a given order. A parallelized version might write in a different order. Only the designer can decide whether that semantic change is acceptable. For this reason, memory dependencies take a longer time to resolve – leading to the recommendation to avoid cutting them if possible.

Anti-dependencies

It is possible to parallelize across an anti-dependency as long as the reads are on one side of the partition and the writes are on the other side. vfAnalyst will only recommend partitions that meet this condition. Parallelization is facilitated by creating a local version of the variable causing the anti-dependency at the beginning of the invocation with the reads in it. This results in more storage being used, and increases the cost of parallelization.

Implementing vfAnalyst's recommendations

Once a parallelization strategy has been decided, you can implement the solution in your program. Since vfAnalyst is system-agnostic, it will not provide system-specific implementation details. The exact steps to be taken vary by system, but follow the same general procedure.

Unrolling loops

Some compilers can handle loop unrolling through directives. The specifics of how this works will vary by compiler, and relying on those capabilities will restrict you to a specific compiler unless you go back and change the directives when changing compilers. To do this in a compiler-independent way, you can specifically cut and copy the loop body code as many times as you want to unroll the loop.

Implementing threads

Threads must be created for parallel tasks. This might be implemented through the use of *posix* calls creating the threads. In general, it's considered good practice to terminate the threads when their execution is complete.

An alternative way to avoid the overhead of thread creation and destruction is to create a pool of threads, referred to as *worker threads*, when the program initializes. Then, instead of creating and destroying threads, you can activate existing idle threads from the pool, returning them to the pool when complete. In this manner, you may be able to re-use threads, ultimately creating fewer threads than would be needed if you explicitly created a thread for each parallel task.

Implementing data communication

vfAnalyst will tell you which data structures need to communicate, as well as any other details required for a robust implementation.

Streaming dependencies will be the most common requirement, and there are two types that may be encountered: FIFO and Windowed FIFO.

FIFO streaming dependencies typically use a FIFO to transfer data, as the name suggests. There are numerous ways to create a software FIFO, and those details are beyond the scope of this paper. The critical dimensions of the FIFO are the size of each entry of the FIFO and the FIFO depth. The size is determined by the amount of data per sync; the depth should be no less than the loop distance, as shown in Figure 13. (For more information on the loop distance, please refer to the [vfAnalyst whitepaper](#)).

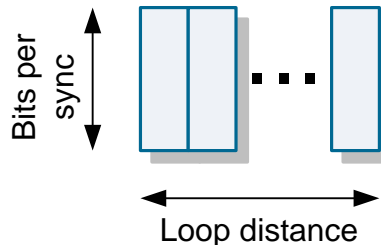


Figure 13. FIFO sizing

A **Windowed FIFO** is more complex, requiring an additional buffer. One implementation is to allocate a buffer for each data sync, either copying data to the buffer outright or, if the algorithm requires, filling the buffer in a random access manner. A FIFO can then be created, with a pointer to the buffer being placed in the FIFO; the reading side can then perform random access reads on the buffer. The size of the buffer is specified by vfAnalyst; the size of a FIFO entry is the size of a pointer; and the FIFO depth is determined by the loop distance.

Another way of implementing a windowed FIFO is to have each entry of the FIFO be a buffer; each entry would then be of the required buffer size, with the depth being set by the loop distance. This is illustrated in Figure 14.

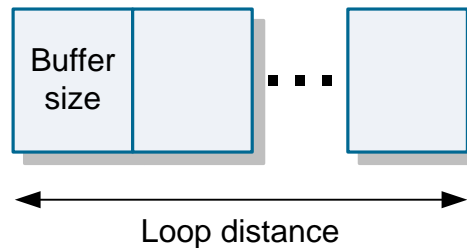


Figure 14. Windowed FIFO sizing

The data structures should be declared just before the loops being parallelized are entered and destroyed just after the loops are exited. However, in the case of nested loops, it is more efficient to create them before the outside loop is entered and destroy them after the outside loop is exited even if they are only used for an inner loop. This saves the overhead of re-creating and re-destroying the data structures each time through the outer loop.

In the sequential program, the writing portion of the algorithm places data in some memory location; the reading portion reads data from that memory location. Once parallelized, those write and read locations must be modified. Instead of writing to memory, the producing task should write to the FIFO; likewise, the consuming task should read from the FIFO.

A synchronization command must then be given to indicate that the FIFO entry is complete. The location of the synchronization command will be determined by the granularity of synchronization. If done one data element at a time, then it can come after the FIFO write; if done after an array is filled in a loop, then it would occur after the loop completes. Similarly, the reading side needs to check for data availability, and this would either occur before the read or before loop entry, depending on the granularity chosen.

In summary, then, handling streaming dependencies means 1) creating and destroying the necessary data structures and 2) changing the writes and reads to refer to those structures. This is schematically illustrated in Figure 15 for a FIFO dependency (with the two possible locations shown for the synch and FIFO-checking operations).

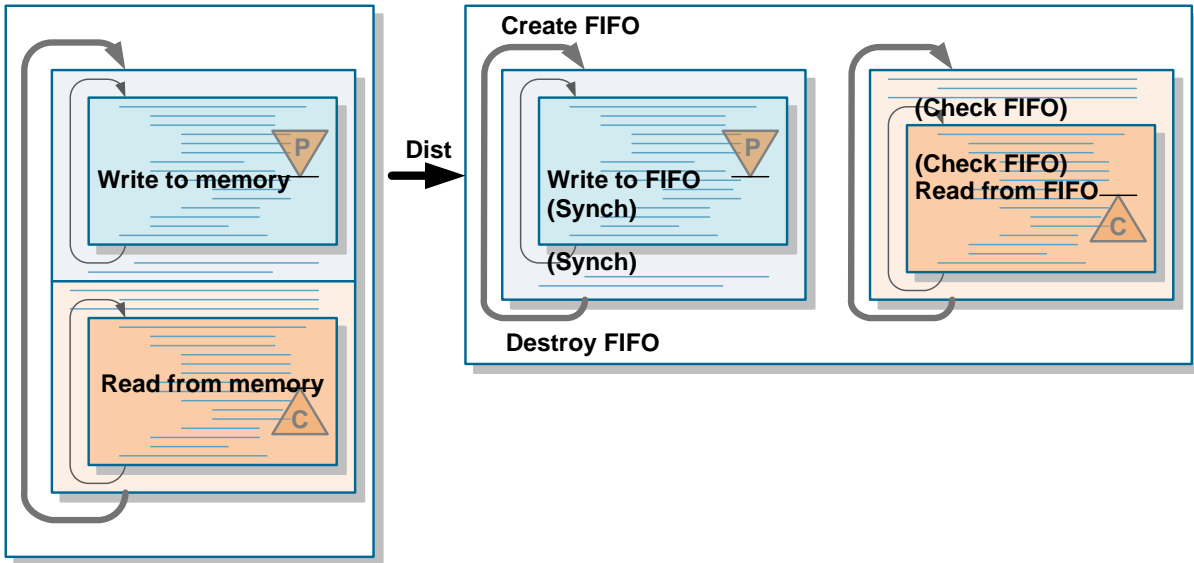


Figure 15. Handling streaming dependencies

Compute dependencies can also be communicated by FIFOs. This essentially means communicating any local variables that are required by the task being communicated with. vfAnalyst will indicate which these are. You can pack the variable values together for each FIFO entry, unpacking them on the reading side. In this case, the size of each FIFO entry would be the combined storage requirement for the variables being communicated. The depth of the FIFO would be no less than the loop distance. This sizing is illustrated in Figure 16.

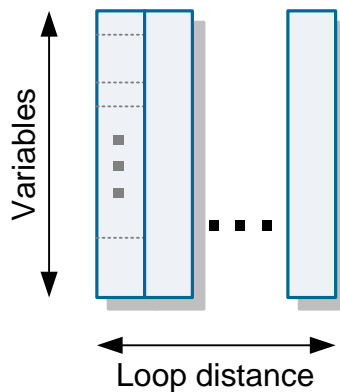


Figure 16. Compute dependency FIFO sizing

As with streaming dependencies, the data structures have to be created outside the loops. Then, right before the new task or thread is created, the current value of those variables should be written to the FIFO (including any packing algorithm if used). They should be read (and unpacked) on the consuming side before the thread continues. This is shown schematically in Figure 17.

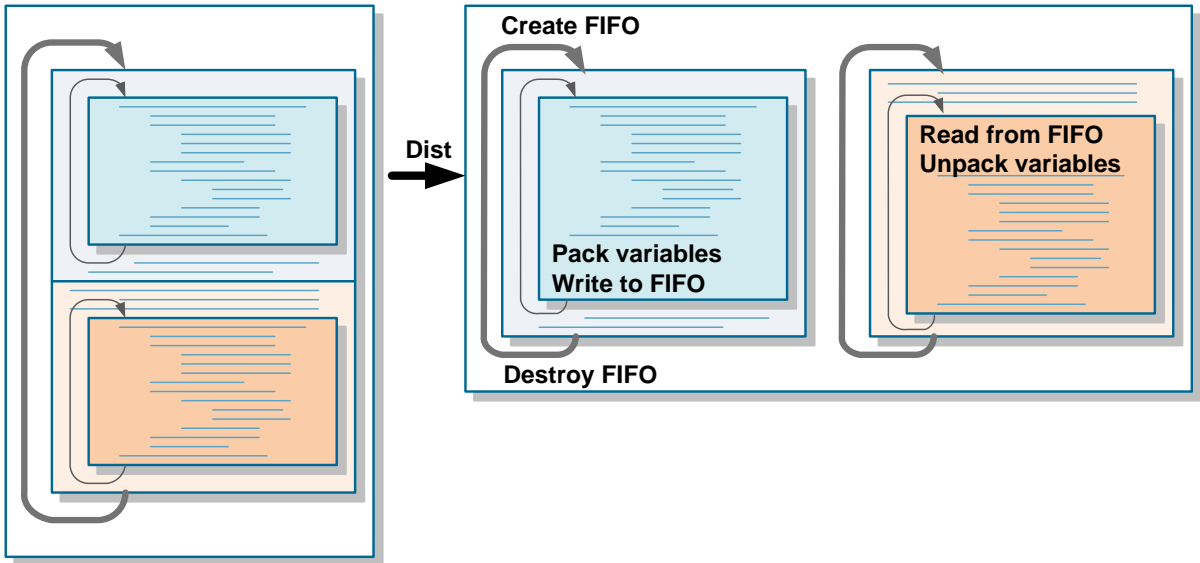


Figure 17. Handling compute dependencies

It may be possible to create one FIFO for both streaming and compute dependencies, using a FIFO entry size that is the combined size required for both streaming and local variable data.

Anti-dependencies are handled by using local copies of the affected variables; the most straightforward way is to have a local copy in each thread. Writes and reads are changed to reference the local version rather than the original.

If the dependency is wholly contained in the invocation being split, then in fact the original global declaration was probably not needed; local scope would have been better, and the original declaration can be replaced by the local declaration, with all references changing to the local variable, as shown in Figure 18.

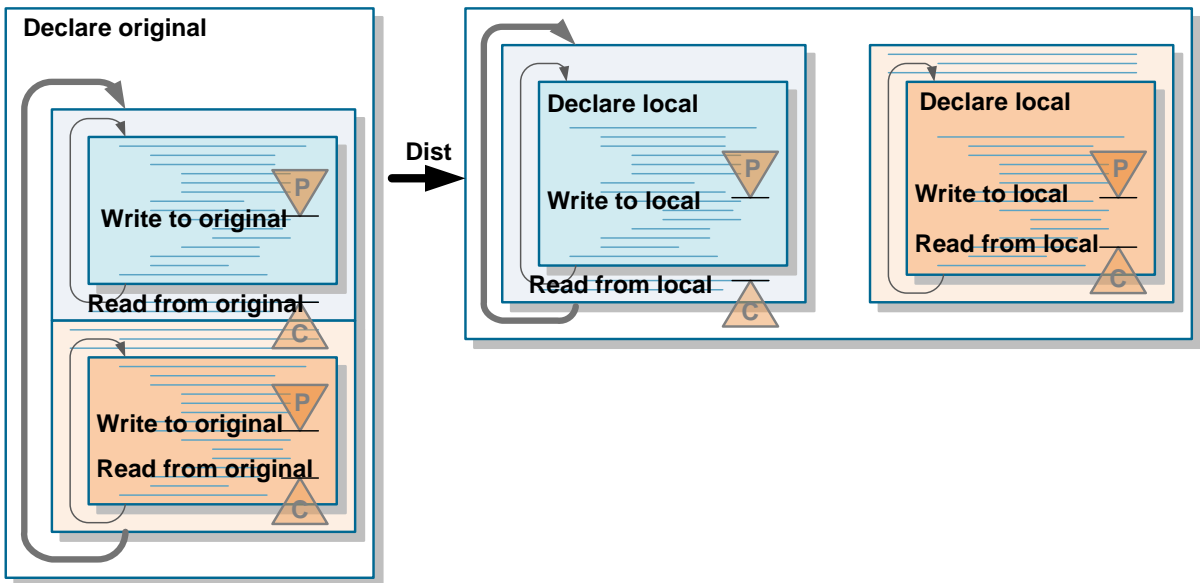


Figure 18. Handling an anti-dependency where local scope suffices

If the affected variable is also used outside the scope of the anti-dependency, then the local declaration is created in addition to the original. If there are no further reads after the parallelized invocation, then a local version can be used in the “second” partition since the written values will not be needed once the invocation completes; this is shown in Figure 19

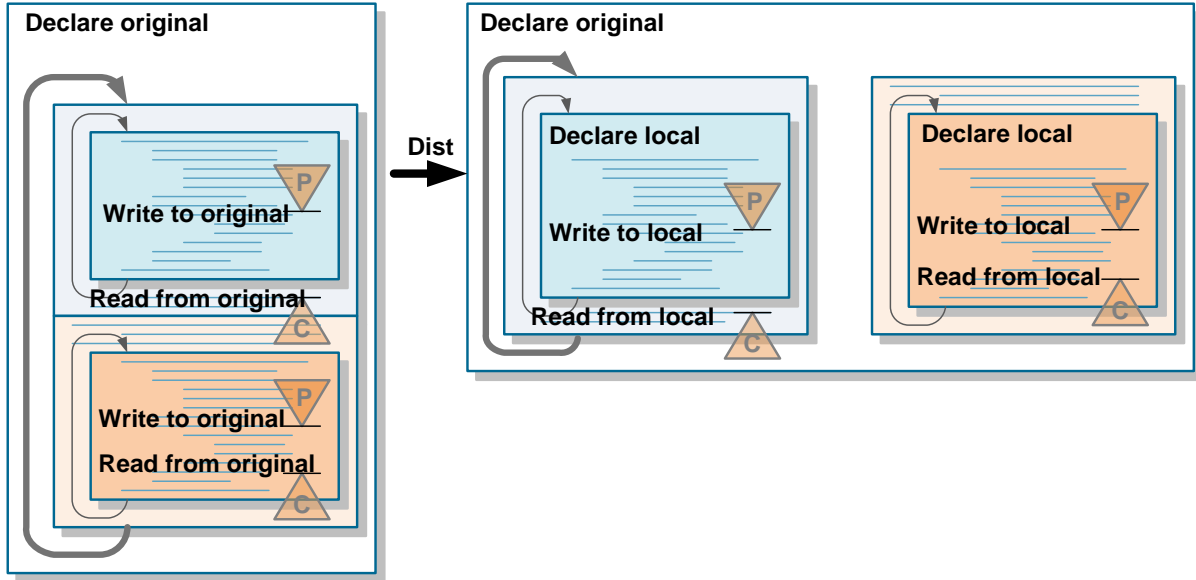


Figure 19. Handling an anti-dependency when the both global and local scope are needed

If there are further reads after the parallelized invocation (which would be reflected by a memory dependency from the second write to that read), then the “second” partition should maintain its reference to the global variable so that the writes will be available afterwards; all references from the write on in the “first” partition are changed to reference the local variable. This is illustrated in Figure 20.

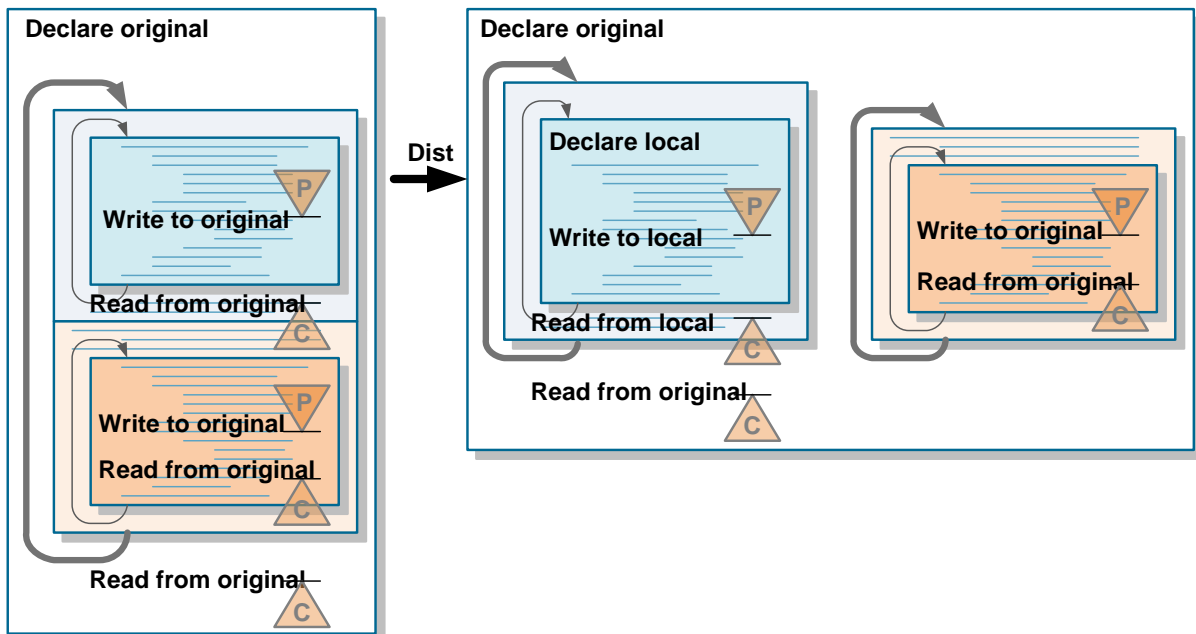


Figure 20. Handling an anti-dependency when the last written value must be preserved

The new thread would keep its original references to the variables, while the old thread would change writes and/or reads to access the local copies instead of the original variables. An example implementation is shown schematically in Figure 20.

Memory dependencies cannot be handled by any single method. In general, cutting memory dependencies will create a semantic change, so the handling of that cut has to be managed on a case-by-case basis.

Conclusion

By using the partitioning algorithms in vfAnalyst, it is simple to identify opportunities to create parallel versions of existing sequential code. Combined with an understanding of how the parallelism can be implemented allows for much faster creation of parallel code than is possible using manual techniques alone.