

Parallel Processing with Catalina

Table of Contents

Introduction.....	2
Using the Pragma Preprocessor.....	2
The Pragmas.....	5
Mandatory Pragmas.....	5
#pragma propeller worker.....	5
#pragma propeller begin.....	8
#pragma propeller end.....	8
Optional Pragmas.....	9
#pragma propeller factory.....	9
#pragma propeller start.....	10
#pragma propeller stop.....	11
#pragma propeller exclusive.....	11
#pragma propeller shared.....	11
#pragma propeller wait.....	12
#pragma propeller lock.....	13
#pragma propeller kernel.....	14
#pragma propeller extern.....	14
A few things to note.....	16
A Tutorial on using the Propeller Pragmas.....	18
Test 1.....	18
Sieve.....	21
Test 2.....	24
Test 3.....	26
Test 4.....	27
Test 5.....	28
Test 6.....	29
Test 7.....	30
Test 8.....	31
Test 9.....	33
Test 10.....	35
fftbench.....	36
Conclusion.....	38

Introduction

Those who have followed the development of Catalina¹ will know that one of its major goals has been to make it easy to utilize the power of parallel processing on the Propeller, using only the C language.

Catalina has had multi-threading and multi-cog capabilities almost from its inception, but they are implemented as libraries, which make the resulting programs non-portable, and can be complex for beginners to learn to use.

More recently Catalina introduced a simple "factory/worker" paradigm to simplify adding parallel features to C programs. In fact, it made it so simple to turn a serial program into a parallel program that the process can now be almost entirely automated, with just a few hints from the program developer required.

In the C language defining "pragmas" is an appropriate mechanism for doing this. It is the route taken by OpenMP² group, as well as some compiler manufacturers.

The C standards allow pragmas to be defined that give a compiler implementation-dependent information or hints about how to compile a program. Pragmas are ignored by compilers that do not implement them, and programs should generally be written so that they will still work – albeit perhaps slightly differently – even if the pragmas are ignored.

So Catalina has now defined pragmas that can be used to tell it how a serial C program should be "parallelized". With the addition of a few pragmas, it is now possible for a simple C program, even one consisting of just a few lines of code, to utilize the full power of the Propeller, and in such a way that the same program can still be compiled by any other C compiler. This document describes the new pragmas, and gives examples of their use.

First, note that all the pragmas discussed in this document start with the standard C keyword **#pragma**, followed by the word **propeller** and then the specific action and options, as follows:

```
#pragma propeller action [ options ... ]
```

Note that pragmas cannot be split across source lines. The whole pragma, including all options, must appear on a single line.

Catalina currently implements pragmas using a separate pragma preprocessor. The pragma preprocessor is provided for Windows and Linux (Ubuntu).

Using the Pragma Preprocessor

Starting with Catalina 4.6, the Pragma Preprocessor is now a fully integrated component of Catalina, although it can also be executed independently. To use the new pragma

¹ Catalina is a free, open-source ANSI C compiler for the Parallax Propeller. It is available from <https://sourceforge.net/projects/catalina-c>

² Open Multi-Processing – see www.openmp.org.

preprocessor in conjunction with Catalina, simply add the **-Z** switch to a normal Catalina command.

For example:

```
catalina -lc -lthreads -Z my_program.c
```

Note that the **-Z** option is *positional* – it must *precede* the names of the C files to be parallelized (parallelizing can also be turned off again with **-z** – i.e. a *lower* case z). Also note that parallelized programs must always be compiled with the threads library (hence the library option **-lthreads** in the example above).

The pragma preprocessor can also be executed as a stand-alone utility (called **parallelize**) which does not depend on the Catalina C preprocessor or C compiler itself. You can use the **parallelize** command to process C source files. If an output file name is specified (with the **-o** option) it will be used, otherwise if no output file is specified, the result will be saved in a file with **_p** appended to the file name.

For example:

```
parallelize my_program.c
```

```
parallelize my_program.c -o my_parallel_program.c
```

Then you can compile the resulting C program as normal (i.e. in the first case above *my_program_p.c*, or in the second case *my_parallel_program.c*). Note that if the input file contains no propeller pragmas, the output file will be identical to the input file, so you can safely use the **parallelize** command on *any* C program.

Several test programs are provided that illustrate the usage of various pragmas, which are described in detail in the next section of this document. The test programs are in the *demos/parallelize* subdirectory of your Catalina installation. Each of the test programs is described in the final section, which is structured as a tutorial. If you wish, you can go straight to that section to get a flavour of how the pragmas are used, and then come back here for the detailed description of the pragmas that are used by the programs.

To build all the test programs, use the **build_all** batch script in the *demos/parallelize* subdirectory, specifying your propeller platform, and your desired HMI and other options (see the *README.TXT* file for more details).

For each program, this script will generate *two* compiled versions of each program – one *serial* version and one *parallel* version. The serial version will have **_s** appended to the file name, and the parallelized version will have **_p**. For example, for the program *test_1.c*, the script will generate (on the Propeller 2) two binaries:

```
test_1_s.bin
```

```
test_1_p.bin
```

Note that you may see some warning messages when you execute the **build_all** script. These messages are normal, and are explained in the descriptions of each of the programs.

These programs can be loaded and run as normal using the **payload** command. For instance, on the Propeller 1, you might use the commands:

```
payload -i test_1_s.binary
```

```
payload -i test_1_p.binary
```

On the Propeller 2, you might use the commands:

```
payload -i -b230400 test_1_s.bin
```

```
payload -i -b230400 test_1_p.bin
```

NOTE: The output of the Catalina parallelizer can be compiled with other C compilers and other thread libraries, such as GCC and Posix Threads. For a complete example, see the file *custom_threads.h* in the *demos/parallelize* subdirectory of the Catalina installation.

The Pragmas

This section is intended for reference. New users should probably look at some of the test programs in the tutorial section before reading this section. The pragmas will make more sense after looking at a few examples.

Mandatory Pragmas

There are only three pragmas that *must* appear in a program for it to be parallelizable – a **worker**, **begin** and **end** pragma.

#pragma propeller worker

Options:

```
[worker_name] (input_variables)
[local(local_variables)]
[if(condition)]
[output(output_variables)]
[threads(count)]
[factory(name)]
[stack(size)]
[ticks(count)]
[memory(management)]
```

This pragma declares a worker thread, the input and local variables it will use, the variables it will output, and various options that affect the execution of the worker threads.

Most of the pragma options are not required just to parallelize a program. They are typically used for specifying variables to be shared between the workers, or for the management of resources that are used by the parallel program. The simplest form of a **worker** pragma has only an empty list of input variables and no options – i.e:

```
#pragma propeller worker(void)
```

The list of input variables represents the variables used by the worker code that must be initialized when the worker thread is started. In most (but not all) cases, the worker code will be within the body of a "for" loop, and the list will include the "for" loop variable.

The list of local variables allows you to specify variables that are *required* by the worker code, but which do not need to be *initialized* to a specific value when the worker code starts, and which are *not required* outside the worker once the worker code is complete.

The list of output variables must be a list of pointers to the actual variables, which must also appear in the local list if they are not in scope where the worker pragma is located. Only local variables can be output – input variables cannot be output since this would disrupt other workers that may be yet to be started. To output an input variable that is modified by the thread, it can be assigned to a local variable and then that variable can be

output. Typically, the last worker thread to execute will return the value of its output variables to the main thread, since this thread's output will overwrite any values already returned by other worker threads. For example:

```
#pragma propeller worker(int i) local(int j, int k) output(int *k)
```

This means that **i** is an input variable – it will be initialized to whatever value **i** has at the moment the worker thread is started. Variables **j** & **k** are local to the thread and will be initialized to zero when the thread is started (they should be explicitly initialized by the worker code if a different initial value is required). The variable **k** will be output when the worker thread completes.

A *condition* can be specified for output of variables by using the **if** option – the variables will only be output *if* the specified condition is met. Note that the condition applies to *all* the output variables – i.e. they are *all* output if the condition is met, otherwise *none* are output. For example:

```
#pragma propeller worker(int i) local(int j, int k, int l)
    if(j > 0) output(int *k, int *l)
```

The above example means both the variables **k** and **l** will be updated if the value of **j** is *greater than zero* on completion of the worker code. Otherwise, neither **k** nor **l** will be updated by this particular worker. For more information on using the **if** option, see the **wait** pragma. Since the **if** and **output** options are so closely related, it is good practice to keep them together in the pragma.

The threads count represents the maximum number of worker threads that will be executed *in parallel* – which may be different to the number of cogs that will be used (typically, it will be larger), and also different to the number that the program may *attempt* to create – if more threads are needed than the maximum thread count, the thread creation process will simply "stall" until some of the existing threads complete.

Note that the number of cogs used to execute threads is not specified by the worker – it is specified when the factory is defined – and if it is not specified then all available cogs will be used.

The factory name specifies the factory that will execute all the worker threads of this type. If not specified, the default name **_factory** will be used.

The stack size is the size of the stack allocated for each worker thread, in longs.

The default thread count is 10, and the default stack size is 200 longs for each worker. If you don't have much spare RAM, you can specify a lower number of worker threads. If you need a lot of worker threads, you can specify a smaller stack size. The stack size may need to be established by trial and error.

The ticks option determines the relative priority of this type of worker thread, compared to other types of worker threads *executed by the same factory* (it has no effect on threads executed by *other* factories). The more ticks, the more processor time is allocated to these worker threads. The default value is 100.

The memory management option can be used to specify the type of memory management to be used by the program – it can be **static** or **dynamic**. The default is **static**, which is faster and results in smaller code sizes, but since the memory required by *all* factories and workers is statically allocated, it can make the final program too large to load, especially on the Propeller 1. If this happens, try using **dynamic** memory management. Note that the memory management applies to *all* workers and *all* factories, not just the one on which the option is specified, and it cannot be changed once it is set to a specific option. This means that only one pragma needs to specify a memory management option, and if multiple pragmas do so, they must all specify the *same* option. Note that the memory management option can actually appear on *any* pragma.

The worker and factory names are optional – if not specified, the default names **_worker** and **_factory** will be used. This is important to know because you may see error or warning messages about these names, especially if you mix default workers and factories with those that are explicitly "named". In general, do not mix default and explicitly named entities – if you need more than one type of worker or more than one type of factory, each one should be explicitly named. Only use the default names when you have only one type of worker and one type of factory.

Note that you cannot use the names of any of the propeller pragma actions or options (e.g. **lock** or **threads**) as either a worker or a factory name.

The actual worker code segment that the thread will execute is identified elsewhere, using the **begin** and **end** pragmas – the **worker** pragma simply declares the worker. You can think of it as similar to a C "forward declaration".

A **worker** pragma may appear more than once, but if it does it must be exactly the same at each appearance, otherwise there may be a worker specification mismatch. A **worker** pragma **MUST** appear at file scope in the file that contains a corresponding **begin** pragma or a **start** pragma.

It is possible to create a worker thread with no input variables (by specifying an input variable list of **(void)**). If so, some other means of indicating the work the worker threads are to perform can be used, and also when the threads should terminate. For example, a global variable might be used to signal all workers.

Examples³:

```
#pragma propeller worker(void)

#pragma propeller worker worker_1(int i, int j) local(int k, float f)
    if ((j != 0) || (k == 0)) output(int *k, float *f)
    stack(200) factory(factory_1) threads(100)
    memory(dynamic)

#pragma propeller worker worker_2(int i) local(int j) stack(200)
    factory(factory_2) threads(100) ticks(1000)
```

³ Note that all pragmas must appear on a single source line, and cannot be spread across multiple lines as shown here and at various other places in this document.

#pragma propeller begin

#pragma propeller end

Options:

```
[worker_name]
```

These pragmas identify the beginning and end of the worker code segment that is to be executed by the worker threads – i.e. the code segment to be parallelized. Typically, they appear within the body of a "for" loop, and one of the worker input variables will be the loop iteration variable. But they can appear around *any* code that can be executed in parallel.

Note that the `worker_name` is optional, and if not specified the default name `_worker` will be used. Note also that the **begin** and **end** pragmas must be at the same lexical scope, and if the program has more than one worker, then *all* the workers should be named in their **begin** and **end** pragmas.

A corresponding **worker** pragma must appear at file scope within the same file as the worker code segment, and precede the use of the **begin** and **end** pragmas. Any variable used in the code segment must either appear within the input or local variable lists in the **worker** pragma, or be in scope (i.e. visible) at the location the **worker** pragma appears.

The **begin** and **end** pragmas cannot be nested, and therefore the `worker_name` on the end pragma is optional even if one is specified on the **begin** pragma – i.e. it is effectively ignored, but should be included for documentation (and because this may change in future!).

Note that the **begin** and **end** pragmas must appear at the same lexical scope, so that when the worker code segment they contain is removed and relocated by the pragma preprocessor, the resulting program is *still valid* according to standard C language rules. This means, for example, that you cannot have the **begin** pragma in a different C "block" than the **end** pragma, such as:

```
if (something) {  
    #pragma propeller begin  
}  
#pragma propeller end
```

Also, note that the worker code segment should be *well-behaved* – i.e. it should not exit except via the normal flow of execution reaching the end of the segment – in particular, it should not use **break** to exit the segment, or **goto** to jump out of the segment.

Examples:

```
#pragma propeller begin  
#pragma propeller end  
  
#pragma propeller begin worker_1  
#pragma propeller end worker_1
```


Optional Pragmas

There are several optional pragmas that *may* appear in a program to be parallelized. These pragmas are generally intended to give finer control over the behaviour of the program once it is parallelized, such as how the program will utilize various shared resources such as cogs, locks and Hub RAM.

#pragma propeller factory

Options:

```
[factory_name]  
[cogs(count)]  
[stack(size)]  
[lock(name)]  
[foreman(name)]
```

This pragma declares a factory, which is what executes worker threads using one or more cogs. Note that the factory name is optional, and If not specified the default name **_factory** will be used.

This pragma is not required if your program only needs one factory, and that factory uses the default name, default stack size, default lock, default foreman, and all the available cogs.

The cogs count specifies the *maximum* number of cogs that will be used by the factory. If less cogs than the specified count are available at run time, only the number of available cogs will be used. Multiple factories can be defined by specifying multiple **factory** pragmas. If there are multiple factories, you should specify the number of cogs for each. If you do not, the first factory declared will use all cogs that are available when the factory is started (see the **start** pragma) and there will be no cogs available for any other factories until it is stopped (see the **stop** pragma). If a factory is started and there are no free cogs available, the factory will be unable to execute any worker threads.

The default stack size is 50 longs. The factory itself does not generally need much stack – the workers in the factory each have their own stack space allocated (see the **worker** pragma). A factory typically only needs a larger stack if it has its own **foreman**.

Each factory has a foreman, which is the first thread to be executed on each factory cog. The foreman never exits, but normally does very little else other than set the kernel thread lock on the cog (remember, each cog must use the same kernel thread lock). However, it can perform other functions if desired.

If no foreman is specified, one with the default name **_foreman** will be created. If you instead want to use your own foreman, it should be declared similar to the following⁴:

```
static int my_foreman(int argc, char *not_used[]) {
    _thread_set_lock(my_lock);
    while (1) {
        _thread_yield();
    }
    return 0;
}
```

The only job the foreman is *required* to do is set the kernel thread lock (in the above case, it does so using a lock named **my_lock**). This should be the first thing it does. The foreman function should never exit.

If a lock is not specified then one with the default name **_kernel_lock** will be created (if it does not already exist) and initialized and will be used by any factories that do not specify a lock name. If the program is *already* multi-threaded then it **MUST** specify the same kernel thread lock that the program *already* uses when declaring any factories. For more details on using an existing lock, see the **lock** pragma.

Note that you cannot use the names of any of the propeller pragma actions or options (e.g. **lock** or **threads**) as a factory name.

This pragma must be used at file scope in the file that contains the worker code segment – i.e. the **begin** and **end** pragmas. It should precede the **worker** pragma.

Examples:

```
#pragma propeller factory
#pragma propeller factory factory_1 stack(200) lock(my_lock) cogs(2)
memory(static)
```

#pragma propeller start

Options:

```
[factory_name]
```

This pragma indicates when the program should start a factory. Note that the `factory_name` is optional, and if not specified the default name **_factory** will be used.

This pragma is not normally required – if the factory is not manually started using this pragma, the factory will be started automatically if any worker code is executed.

If the factory *is* manually started, then the automatic starting of the factory is suppressed, and so this pragma must appear at a point of execution that precedes the execution of any worker code. This is done to allow finer control over the starting and stopping of factories.

⁴ The arguments to the foreman function – i.e. `argc` and `argv` – are not actually used, but this is the standard Catalina definition of a thread function. Similarly, a thread function can return an int value, but in the case of a foreman the function would typically never exit.

The file that contains this pragma must also contain a **worker** pragma. However, note that the file does NOT need to contain a **factory** pragma.

This pragma will typically appear in the body of the C **main()** function.

Examples:

```
#pragma propeller start
#pragma propeller start factory_1
```

#pragma propeller stop

Options:

```
[factory_name]
```

This pragma indicates where the program should stop a factory. Note that the `factory_name` is optional, and if not specified the default name **_factory** will be used. Stopping the factory will stop all the worker threads, even if they have not completed their worker code.

A factory does not *need* to be terminated – this happens automatically when the program exits.

The file that contains this pragma must also contain a **worker** pragma. However, note that the file does NOT need to contain a **factory** pragma.

Examples:

```
#pragma propeller stop
#pragma propeller stop factory_1
```

#pragma propeller exclusive

#pragma propeller shared

Options:

```
[lock | extern] [exclusion_name]
```

These pragmas identify the start and end of an “exclusive” code segment – i.e. a code segment that can be executed by *only one* worker thread (of any type) at a time. While one worker thread is executing the exclusive code segment, all other worker threads will be temporarily “stalled” if they try and execute the same code segment, or *any other exclusive code segment with the same exclusion name*. Exclusive code segments cannot be nested, but can appear either within or outside a worker code segment. When the worker executing the exclusive code segments exits the segment, *one* of the stalled workers (if there are any) will be allowed to enter.

The optional exclusion name determines the name of the exclusion lock that will be used to control access to the exclusive code segments. The name is important because all the exclusive code segments that share the same name will use the same lock – i.e. if there are multiple exclusive code segments that share the same name, only *one such code segment* can be entered by any worker thread, but other worker threads may be executing

an exclusive code segment with a *different* name. If not specified then the default exclusion name **_region** will be used.

If neither the **lock** or **extern** option is specified, the exclusive code segment is considered *local*, because the exclusion will apply only within the file in which the **exclusive** pragma appears – i.e. if the same exclusion name appears in a local **exclusive** pragma in another file it will create a *different* exclusive code segment.

If either the **lock** or **extern** option is specified, the exclusion is considered *global* and if the same exclusion name is specified in an **exclusive** pragma in another file it will share the same exclusion lock. Note that only one file can specify the **lock** option in its **exclusive** pragmas, and any other files that want to share the same exclusion should specify the **extern** option their **exclusive** pragmas with the same exclusion name. Think of the **lock** option as identifying the file where the lock is declared, and all other files then declare it using the **extern** option so that it will not be re-declared (this is similar to the C use of the **extern** keyword, and also similar to the **lock** and **extern** pragmas).

Note that the **lock** and **extern** options are required *only* when the program to be parallelized consists of multiple files. While it is recommended that you specify the same **lock** or **extern** options on all the **exclusive** and **shared** pragmas in the same file that share the same exclusion name, it is actually only the first **exclusive** pragma in each file that determines whether the lock will be declared in the current file and whether it will be *local* or *global*.

Note that you cannot use the same exclusion lock name as both a *local* and a *global* exclusion lock. Doing so will prevent the program from compiling correctly.

Examples:

```
#pragma propeller exclusive
#pragma propeller shared
#pragma propeller exclusive my_region
#pragma propeller shared my_region
#pragma propeller exclusive lock my_region
#pragma propeller shared lock my_region
#pragma propeller exclusive extern my_region
#pragma propeller shared extern my_region
```

#pragma propeller wait

Options:

```
[worker_name]
[(condition)]
```

Without a condition option, this pragma signals that the program should wait for all worker threads of the specified type to complete. Note that the worker name is optional, and if not

specified the default name **_worker** will be used. Note that if one or more of the worker threads *never* completes, the program will wait forever.

With a condition option specified, this pragma signals that the program should wait for *either* the condition to be fulfilled (presumably, by one or more of the worker threads) or for all the worker threads to complete. This option allows a worker thread to signal that they have found a condition that allows the main thread to continue, even if other worker threads are still executing.

Note that if the condition is met, the program will continue *even though* some of the worker threads may still be executing. This allows (for example) the main thread to start processing an interim result generated by one of the worker threads, which will generally update an output variable using the same condition. If it later becomes essential for the main thread to wait for all the worker threads to complete, another **wait** pragma can be specified, without a condition.

For example:

```
// wait for either all workers to complete, or for any worker to set k > 0
#pragma propeller wait worker_1(k > 0)
if (k > 0) {
    // do something here to process the result signalled by a worker
}

// now wait for all worker to complete
#pragma propeller wait worker_1
```

This pragma can only appear in the same file as the **factory** pragma of the factory that executes the corresponding workers.

Examples:

```
#pragma propeller wait
#pragma propeller wait (k > 0)
#pragma propeller wait worker_1
#pragma propeller wait worker_1 ((i == 1) || (k == 1))
```

#pragma propeller lock

Options:

```
[lock_name]
```

This pragma signals that the program should use a specific kernel thread lock, which must be used in *all* multi-threading kernels, and therefore the same lock name must be specified in all **factory** pragmas.

This pragma is not normally required.

Using a **lock** pragma may be required in programs that are *already* multi-threaded, since those programs will typically *already* use a specific kernel thread lock in all the

multi-threaded kernels the program creates, and the same kernel thread lock must be used by all the factories created when the program is parallelized.

A **lock** pragma may also be required if the default lock name cannot be assumed for other reasons, such as when a program to be parallelized consists of multiple files. See also the **extern** pragma.

A **lock** pragma should appear in only one file. Other files that need visibility of the lock should use the **extern** pragma.

If the **lock** pragma is used, but the lock is *not* already initialized as the kernel thread lock by the program, then this can be done by using a **kernel** pragma.

If lock_name is not specified then the default name **_kernel_lock** will be used – however, note that in this case the lock must *still* be manually initialized as the kernel thread lock, even though it is the default name. This allows for finer control over the initialization of the kernel thread lock (which can be done using the **kernel** pragma).

Examples:

```
#pragma propeller lock
#pragma propeller lock my_lock
```

#pragma propeller kernel

Options:

```
[lock_name]
```

This pragma specifies that lock_name should be initialized as the kernel thread lock.

This pragma is not normally required. See the **lock** pragma for where it may be required.

If a lock_name is not specified then the default name **_kernel_lock** will be used. This allows for finer control over the initialization of the kernel thread lock.

Note that either a **lock** pragma or an **extern** pragma must appear in the same file, and before the **kernel** pragma appears.

Examples:

```
#pragma propeller kernel
#pragma propeller kernel my_lock
```

#pragma propeller extern

Options:

```
[lock_name]
```

This pragma signals that the program uses a specific lock, but that lock is defined in a different source file. It acts much like a normal C “extern” declaration.

This pragma is not normally required. An **extern** pragma is typically required only if the program to be parallelized consists of multiple files. Then a **lock** pragma should be used in

one file, and the **extern** pragma should be used in any other files where the lock must be visible.

If a `lock_name` is not specified then the default name **_kernel_lock** will be used – however, note that in this case the lock must *still* be manually initialized as the kernel thread lock, even though it is the default name.

Examples:

```
#pragma propeller extern  
#pragma propeller extern my_lock
```

A few things to note

There are a few other things to note about the current pragma preprocessor:

1. The pragma preprocessor defines the symbol `__PARALLELIZED` in the output source file. This allows a program to tell if it has been processed into a parallel version or not. There is an example of using this feature in the `test_8.c` test program.
2. The pragma preprocessor is still under development, and can be a bit fragile – i.e. unless the pragmas are specified with EXACTLY the correct syntax, and in the correct order, the pragma preprocessor may not process them correctly, and/or the resulting program may fail to compile or not execute correctly. But since the pragma preprocessor simply produces C code, the output of it can be checked to see if the pragmas have been interpreted correctly.
3. The worker code segment, indicated by the **begin** and **end** pragmas, will generally end up being moved to another location in the C source output. This can lead to compilation errors if this code refers to functions or variables that are not yet defined at the file location where the code ends up (which will be where the factory pragma is located, or where the worker pragma is located if the default factory is used). Normally, this can be resolved by judicious placement of the factory and worker pragmas, or by using an explicit **factory** pragma instead of just using the default factory. As a last resort, some entities in the original source file may have to be either relocated, or else a forward reference can be used to make sure they are "visible" to the parallelized code segment.
4. The pragma processing is currently done *separately from* and *before* the normal C preprocessor. This means you cannot use `#if ... #else ... #endif` constructs to affect the operation of propeller pragmas.
5. If a compilation error occurs, the line number reported by the compiler will generally refer to the *input* file, not the *output* file generated by the pragma preprocessor. However, if the line number makes no sense in the input file, try examining the output file instead. The output files are normally deleted by Catalina after being compiled, but you can specify that they not be deleted by using the `-u` command line option to Catalina. For example:

```
catalina -lc -lthreads -Z program.c -u
```

In this case, the `-u` option will result in the output of the parallelizer to be left in the file `program_p.c`.

6. The memory management option (discussed in the **worker** pragma description) can actually appear as an option to *any* pragma, but it is currently a global setting, so if it appears multiple times it must specify the same memory management each time. This may change in future, allowing different memory management for different factories or workers. The current memory management options are **static** or **dynamic**. The **static** option is the default, but since it statically allocates all the

RAM necessary for *all* factories and workers, it may not be the best option for programs that dynamically start and stop factories, or who only use a subset of workers at any one time – in such cases, statically allocating all the memory may be very wasteful of Hub RAM, and **dynamic** memory management might be a better option. The disadvantage of dynamic memory management is that it is slower to start and stop factories, and it also means the program must include the *malloc()* and *free()* C library functions, which adds a few kilobytes of additional code to the program.

7. The programs described in this document are all compiled by the **build_all** script to use the *integer* version of the standard C libraries, and the *tiny* version of the stdio library (hence the use of the **-lci** and **-ltiny** library options). If you compile these programs to use the normal floating point versions of the C library, or the full version of stdio, you will need to increase the stack size of any workers and factories that use these libraries – typically, a stack size of **400** would be required.

A Tutorial on using the Propeller Pragmas

Test 1

The first test program we will look at is a fairly trivial “Hello, world” type program. It is in the file *test_1.c*. Here is the actual program source code, in full (with comments and pragmas removed for clarity and brevity):

```
void main() {
    int i;

    printf("Hello, world!\n\n");

    for (i = 1; i <= 10; i++) {
        printf("a simple test ");
    }

    printf("\n\nGoodbye, world!\n");

    while(1); // never terminate
}
```

It is perhaps the simplest possible C program that can sensibly be turned into a parallel program!

And below is how the source code appears in the file *test_1.c*, with the addition of a few pragmas (shown in green).

```
#pragma propeller worker(void)

void main() {
    int i;

    printf("Hello, world!\n\n");

    for (i = 1; i <= 10; i++) {

        #pragma propeller begin
        printf("a simple test ");
        #pragma propeller end

    }

    printf("\n\nGoodbye, world!\n");

    while(1); // never terminate
}
```

Let’s explain those pragmas one by one. The first pragma is a **worker** pragma:

```
#pragma propeller worker(void)
```

This tells the pragma preprocessor to expect a parallelizable worker code segment that requires no variables (this is what **void** means). This pragma must appear before the

worker code segment itself, in a place in the source file where a function could be defined, and where any variable or function that the code needs to access is “visible” according to the standard C scope rules. It acts like a C “forward declaration” of the worker. In this case, since the worker uses no variables, anywhere near the beginning of the file is suitable.

The next two pragmas – the **begin** and **end** pragmas – identify the actual worker code segment:

```
#pragma propeller begin
// any code in here will become our "worker" code
#pragma propeller end
```

In this case, the worker code is a simple **printf** statement. But it may be much more complex. The most important thing to note here is that the **begin** and **end** pragmas must appear at the same lexical scope, so that when the worker code they contain is removed and relocated by the pragma preprocessor, the resulting program is *still valid* according to standard C language rules. This means, for example, that you cannot have the **begin** pragma in a different C “block” than the **end** pragma, such as:

```
if (something) {
    #pragma propeller begin
}
#pragma propeller end
```

Before we try to “parallelize” this program, let’s first execute it as a simple serial program, essentially just ignoring the pragmas. To do that, we will use the **build_all** batch script provided.

For example, if we want to execute the program on a Propeller 2 **P2_EVAL** board, we would first make sure our board is plugged in to the PC, and then enter a command such as⁵:

```
build_all P2_EVAL NATIVE
```

The program should compile with no errors. Then we can load and execute the serial version on our Propeller using a command like:

```
payload -i -b230400 test_1_s
```

⁵ This command assumes you are using a Propeller 2 evaluation board, which Catalina refers to as **P2_EVAL**. If not, specify the platform you do have as your first parameter to the **build_all** command – e.g. **C3**. The **build_all** script only needs to be executed once to compile all the test programs (except for test 9 – see the description of that test program for more details) so it will not be mentioned again.

This is what we should see as a result:

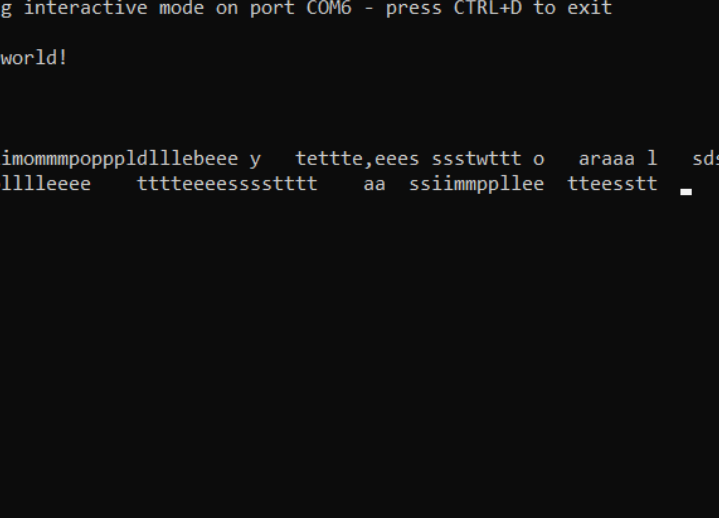
```
C Catalina Command Line - catalina_serial P2_EVAL test_1
— □ ×
Entering interactive mode on port COM6 - press CTRL+D to exit
Hello, world!
a simple test a simple test a simple test a simple test a simple test a simple t
est a simple test a simple test a simple test a simple test
Goodbye, world!
```

That all seems fairly straightforward – it is precisely what we should expect from the source code in the program.

Now, let's execute the *parallel* version of this program. We can do that by entering a command like:

```
payload -i -b230400 test_1_p
```

We will see something like this:



Catalina Command Line - catalina_parallel P2_EVAL test_1

Entering interactive mode on port COM6 - press CTRL+D to exit

Hello, world!

aaaa
s
sssiGiiimommmppopppldlllebeee y tette,eees ssstwttt o aaaa l sdsssi!iiim
mmppppplllleeee ttteeeessstttt aa ssiimmppllee tteesstt _

Woah! Surely that can't be right! What's gone wrong?

Well in fact *nothing* has gone wrong – this is exactly what you *should* expect to see. The 10 iterations of the “for” loop are now being executed *in parallel* with the main program, and the output of each **printf** statement is getting mixed up with the *other* **printf** statements⁶. If you look carefully, you will see that all the characters of the output are still there, and all the characters of each **printf** are in the correct order – it is just that the output of each is being jumbled together.

We will show how this can be addressed later, but for now we will just leave it and move on, because it is enough to show that our program *really is* being executed in parallel. In fact, our simple program is now being executed *on all the cogs available on the Propeller!*

Welcome to parallel processing!

Sieve

Now, from here we could go straight onto look at *test_2.c*, *test_3.c* etc, which go into more detail on the various pragmas and how their use affects the parallel execution of programs.

But before we do that, let’s look at a real example, which we can already do with not much more than what we have already learned.

This example is in the file *sieve.c*, and is a version of the Sieve of Eratosthenes⁷ – a classic algorithm for enumerating prime numbers.

⁶ Technically, reason the **printf** statements “intermingle” like this when executed in parallel is because Catalina uses *pre-emptive multitasking*. If instead of using Catalina threads, the Catalina parallelizer is used with *Posix* threads (which typically use only co-operative multitasking) then the individual **printf** statements may not intermingle. In such cases, it is difficult to see that they are being executed in parallel using such a simple example – but this will become more evident in the next few examples.

⁷ For more details, see https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Here is the source code in full, with pragmas included. You will find it in the file `sieve.c`:

```
#include <stdio.h>
#include <stdlib.h>
// define the size of the sieve:
#define SIEVE_SIZE 12000
unsigned char *primes = NULL;

#pragma propeller worker(unsigned long i) local(unsigned long j) stack(60)
// main : allocate and initialize the sieve, then eliminate all multiples
//         of primes, then print the time taken, and all the resulting primes.
void main(void){
    unsigned long i, j;
    unsigned long k = 1;
    unsigned long count;
    // allocate a byte array of suitable size
    primes = malloc(SIEVE_SIZE);
    if (primes == NULL) {
        // cannot allocate array
        exit(1);
    }
    // initialize sieve array to zero
    for (i = 0; i < SIEVE_SIZE; i++) {
        primes[i] = 0;
    }
    t_printf("starting ...\n");
    #pragma propeller start
    // remember starting time
    count = _cnt();
    // eliminate multiples of primes
    for (i = 2; i < SIEVE_SIZE/2; i++) {
        if (primes[i] == 0) {
            #pragma propeller begin
            for (j = 2; i*j < SIEVE_SIZE; j++) {
                primes[i*j] = 1;
            }
            #pragma propeller end
        }
    }
    #pragma propeller wait
    // calculate time taken
    count = _cnt() - count;
    t_printf("... done - %ld clocks\n", count)

    t_printf("\npress a key to see results\n");
    k_wait();
    // print the resulting primes, starting from 2
    for (i = 2; i < SIEVE_SIZE; i++) {
        if (primes[i] == 0) {
            t_printf("prime(%d)= %d, ", k++, i);
        }
    }
    while(1);
}
```

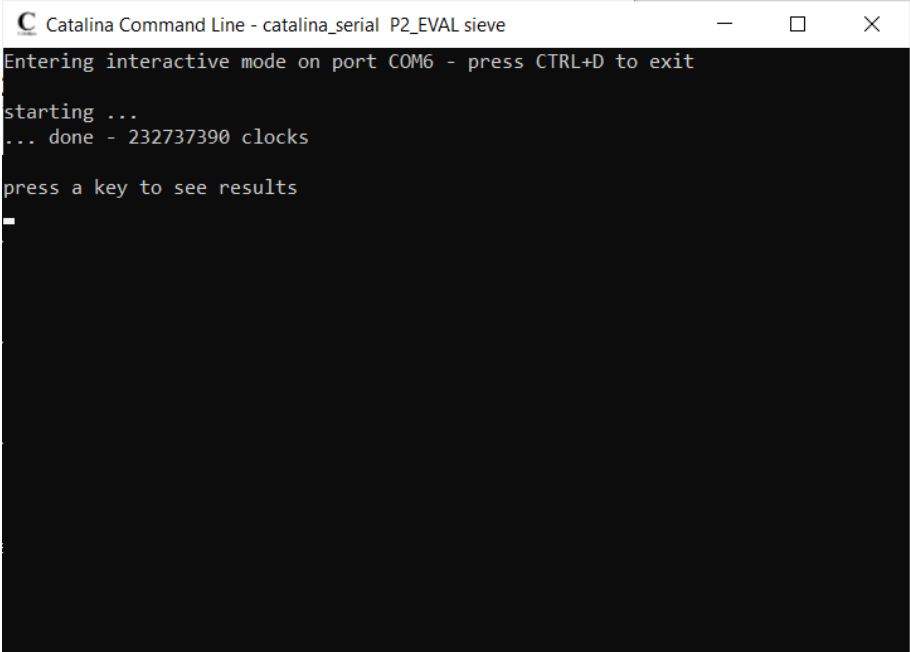
You should recognize several of the pragmas already – the **worker**, **begin** and **end** pragmas do the same job as we have already seen in the *test_1.c* program. The **worker** pragma looks a bit more complex – e.g. it now identifies the variables that the worker code segment needs to use. For more details on what this means, see the description of the **worker** pragma given elsewhere in this document.

There is a new **start** pragma, which manually starts the factory that will execute the workers. We do not strictly *need* this pragma, but it makes the program more efficient by avoiding the need to check if the factory has been started each time we execute the worker code.

Also, there is a new pragma towards the end of the program – a **wait** pragma. We will go into that in the next example program. For now, let's just execute the Sieve program – first as a serial program, then as a parallel program. If you have executed the **build_all** script, you can execute the *serial* version of the program on a Propeller 2 with a command like:

```
payload -i -b230400 sieve_s
```

This is what we should see as a result:

A screenshot of a terminal window titled "Catalina Command Line - catalina_serial P2_EVAL sieve". The terminal shows the following text: "Entering interactive mode on port COM6 - press CTRL+D to exit", "starting ...", "... done - 232737390 clocks", and "press a key to see results". A cursor is visible on the line "press a key to see results".

```
Catalina Command Line - catalina_serial P2_EVAL sieve
Entering interactive mode on port COM6 - press CTRL+D to exit
starting ...
... done - 232737390 clocks
press a key to see results
_
```

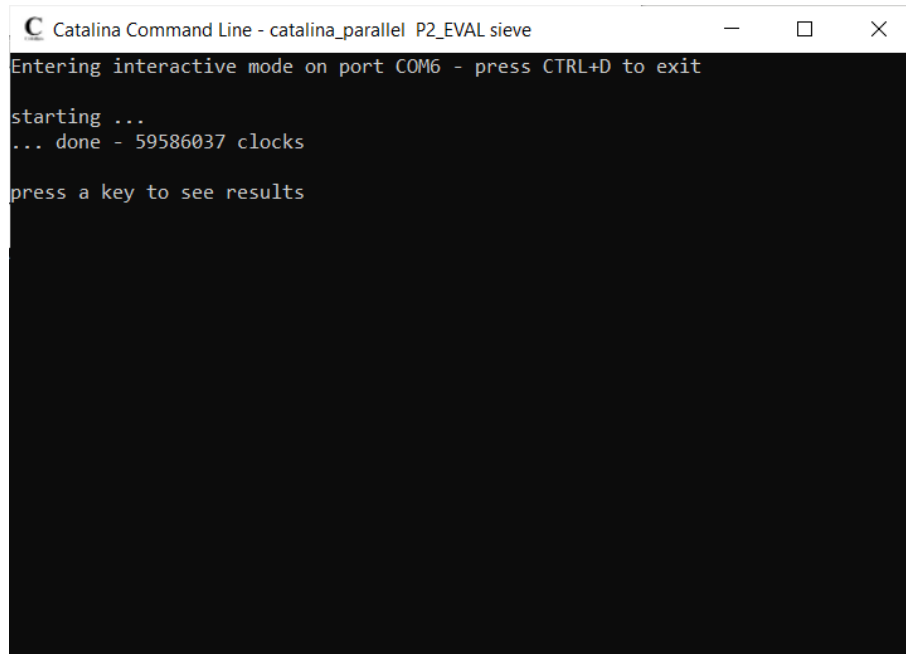
The important detail to notice is the time taken – in this case, 232,737,390 clocks⁸.

Now execute it as a parallel program, using a command like:

```
payload -i -b230400 sieve_p
```

⁸ The actual numbers will be different on the Propeller 1. But the final outcome will be similar.

This is what we should see as a result, again on a Propeller 2:



```
Catalina Command Line - catalina_parallel P2_EVAL sieve
Entering interactive mode on port COM6 - press CTRL+D to exit

starting ...
... done - 59586037 clocks

press a key to see results
```

This time, the program took 59,586,037 clocks.

By *parallelizing* this program – essentially doing nothing more than adding a few pragmas – we have enabled it to run using *all the available Propeller cogs* and in doing so have made the program execute nearly *4 times faster!*

As you work your way through the remaining test programs in this tutorial, you may feel overwhelmed by the sheer *number* of pragmas in some of the programs – in some cases they seem to outnumber the actual lines of C source code! But keep in mind that this is not typical. The test programs have been specifically *designed* to illustrate the uses of the various pragmas.

This is one reason we took this detour into the Sieve program – real programs don't often *need* many pragmas. We will finish our tutorial with another “real world” program – *fftbench.c* - a Fast Fourier Transform Benchmark program – to reinforce this point.

Test 2

The next program we will look at is in the file *test_2.c*. In this file you will see a few more complicated pragmas. First of all, there are *two worker* pragmas:

```
#pragma propeller worker worker_1(int i) stack(150)
#pragma propeller worker worker_2(int i) stack(150)
```

This just means that we have two different types of worker. Here they are:

```
#pragma propeller begin worker_1
printf("a simple test %d ", i);
#pragma propeller end worker_1
```


and

```
#pragma propeller begin worker_2
printf("A SIMPLE TEST %d ", i);
#pragma propeller end worker_2
```

The most important thing to note here is that our workers use a variable – `i` – which must be specified in the worker pragmas, so that each worker gets a copy of that variable at run time. If we do not declare the variables in our **worker** pragmas and they are not “visible” at the point of that pragma is in the source file according to the normal C scope rules, we will get an error about the variable being undefined when we compile our program.

We can also see two **wait** pragmas:

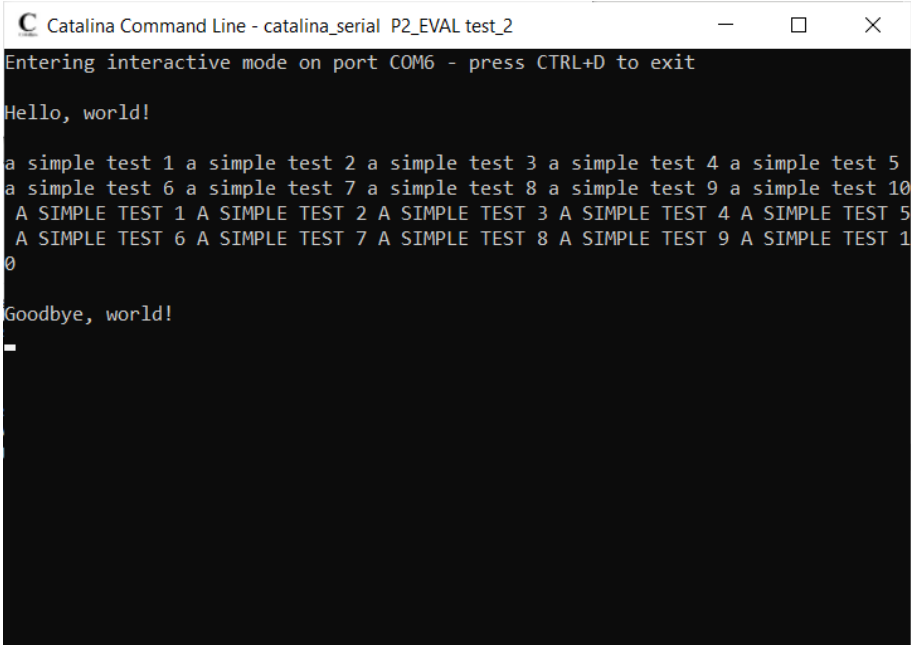
```
#pragma propeller wait worker_1
#pragma propeller wait worker_2
```

These pragmas tell the program to wait until all the workers of the types specified have completed.

Let’s first run this program as a normal serial program. To do this, use a command like:

```
payload -i -b230400 test_2_s
```

You should see something like this as the result:

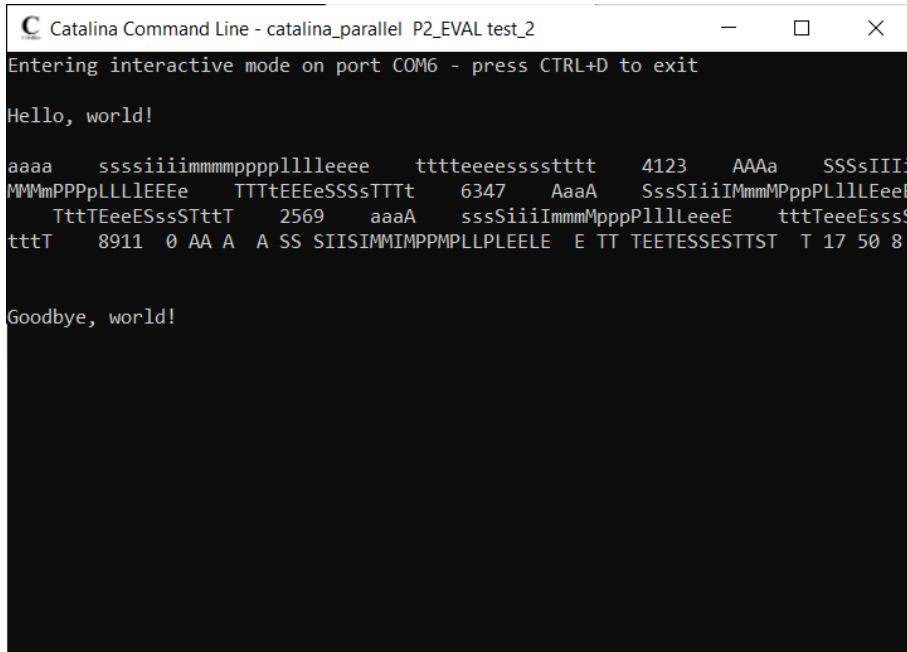


```
Catalina Command Line - catalina_serial P2_EVAL test_2
Entering interactive mode on port COM6 - press CTRL+D to exit
Hello, world!
a simple test 1 a simple test 2 a simple test 3 a simple test 4 a simple test 5
a simple test 6 a simple test 7 a simple test 8 a simple test 9 a simple test 10
A SIMPLE TEST 1 A SIMPLE TEST 2 A SIMPLE TEST 3 A SIMPLE TEST 4 A SIMPLE TEST 5
A SIMPLE TEST 6 A SIMPLE TEST 7 A SIMPLE TEST 8 A SIMPLE TEST 9 A SIMPLE TEST 10
Goodbye, world!
```

Now let’s run this program as a parallel program. To do this, use a command like:

```
payload -i -b230400 test_2_p
```

You should see something like this as the result:



```

Catalina Command Line - catalina_parallel P2_EVAL test_2
Entering interactive mode on port COM6 - press CTRL+D to exit

Hello, world!

aaaa  ssssiimmmpplllleeee  tttteesssstttt  4123  AAAa  SSSsIIIi
MMmPPpLLLLEEe  TTTtEEeSSStTTt  6347  AaaA  SssSIiIMmmMPpPLllLeeeE
TttTeeESssSTttT  2569  aaaA  sssSiiiIimmMPppPlllLeeeE  tttTeeEsssS
tttT  8911  0 AA A  A SS SIISIMIMPPMPLPLEELE  E TT TEETESSESTTST  T 17 50 8

Goodbye, world!

```

Again, our output is all jumbled – but note that the last line (“Goodbye, world!”) is no longer part of the jumble. This is because we told our program to wait until all the workers had completed before proceeding, so that last **printf** is not jumbled up with the rest of them.

We will learn another technique for preventing this “jumbling” in the next example.

Test 3

The next program we will look at is in the file `test_3.c`. In this file you will see a few new pragmas in the worker code. Here they are:

```

#pragma propeller begin worker_1
#pragma propeller exclusive
printf("a simple test %d ", i);
#pragma propeller shared
#pragma propeller end worker_1

```

and

```

#pragma propeller begin worker_2
#pragma propeller exclusive
printf("A SIMPLE TEST %d ", i);
#pragma propeller shared
#pragma propeller end worker_2

```

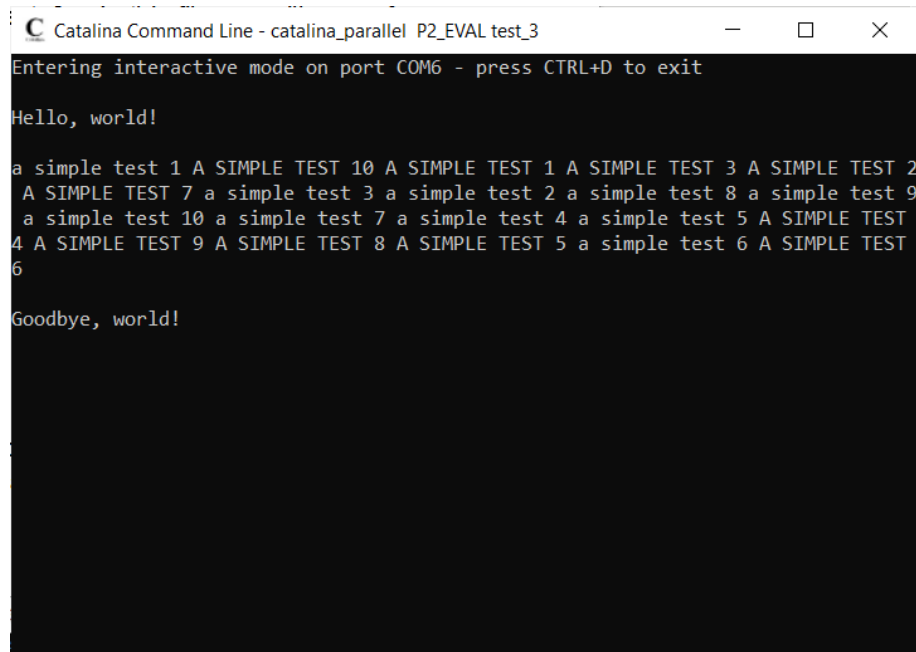
The **exclusive** and **shared** pragmas define a code segment that only one worker can execute at a time. This means that each **printf** statement will be executed separately, even though the workers are executing in parallel. Note that we would not normally make the whole section of worker code “exclusive”, but in this case there is only one line of code in each worker code segment!

You can execute the program serially yourself, but you can probably predict what the output will be. Let’s just execute it as a parallel program.

To do this, use a command like:

```
payload -i -b230400 test_3_p
```

You should see something like this as the result:



```
Catalina Command Line - catalina_parallel P2_EVAL test_3
Entering interactive mode on port COM6 - press CTRL+D to exit

Hello, world!

a simple test 1 A SIMPLE TEST 10 A SIMPLE TEST 1 A SIMPLE TEST 3 A SIMPLE TEST 2
A SIMPLE TEST 7 a simple test 3 a simple test 2 a simple test 8 a simple test 9
a simple test 10 a simple test 7 a simple test 4 a simple test 5 A SIMPLE TEST
4 A SIMPLE TEST 9 A SIMPLE TEST 8 A SIMPLE TEST 5 a simple test 6 A SIMPLE TEST
6

Goodbye, world!
```

No more jumbling! But examine the output a bit more carefully, and you will see that we are still executing the workers in parallel – you can tell this because there are upper case and lower case versions of the messages are being intermingled, even though each `printf` is being executed exclusively. Also, the messages are not coming out strictly in numerical sequence. But if we specifically needed to achieve *that*, we would probably not try running them in parallel!

Test 4

The next program we will look at is in the file `test_4.c`. In this file you will see a few changes in the **factory** and **worker** pragmas. Here they are:

```
#pragma propeller factory factory_1 cogs(2)
#pragma propeller factory factory_2 cogs(2)

#pragma propeller worker worker_1(int i) threads(4) factory(factory_1)
#pragma propeller worker worker_2(int i) threads(4) factory(factory_2)
```

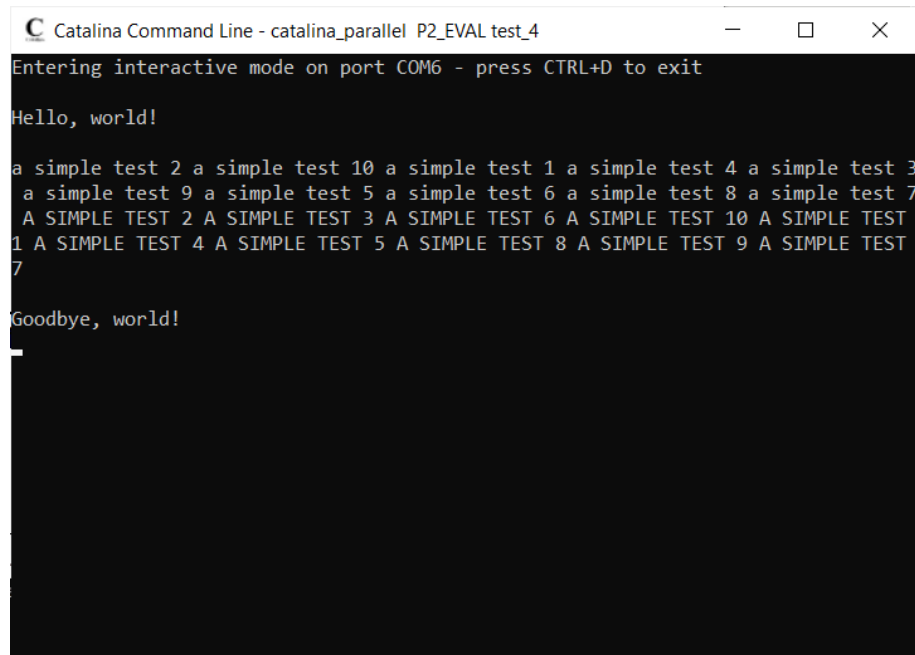
Notice that we now have *two* factories, and that we assign one type of worker to each factory.

One important thing to note here is that if we have multiple factories and want them to execute in parallel, we cannot simply default to assigning all the available cogs to each factory – in this case, we assign two cogs to each factory. If we did not, then when we attempted to start the second factory the program would stall because there are no cogs available.

You can execute *test_4.c* yourself as a serial and parallel program using the following commands:

```
payload -i -b230400 test_4_s
payload -i -b230400 test_4_p
```

The output is similar to the output of *test_3.c* but there is one difference that you may not spot immediately (**hint**: look at the location of the **wait** pragmas, and then look at how this affects the output of the upper and lower case messages):



```
Catalina Command Line - catalina_parallel P2_EVAL test_4
Entering interactive mode on port COM6 - press CTRL+D to exit

Hello, world!

a simple test 2 a simple test 10 a simple test 1 a simple test 4 a simple test 3
a simple test 9 a simple test 5 a simple test 6 a simple test 8 a simple test 7
A SIMPLE TEST 2 A SIMPLE TEST 3 A SIMPLE TEST 6 A SIMPLE TEST 10 A SIMPLE TEST
1 A SIMPLE TEST 4 A SIMPLE TEST 5 A SIMPLE TEST 8 A SIMPLE TEST 9 A SIMPLE TEST
7

Goodbye, world!
```

Test 5

The next program we will look at is in the file *test_5.c*. In this file we introduce the idea of using our own lock, and also our own foreman, rather than using the defaults. Consequently you will see a **lock** pragma, and a few changes in the **factory** pragmas.

First, we must declare our lock:

```
#pragma propeller lock my_lock
```

Now we can use this lock in our **factory** pragmas:

```
#pragma propeller factory factory_1 cogs(2) lock(my_lock)
#pragma propeller factory factory_2 cogs(2) lock(my_lock) stack(150)
                                foreman(my_foreman)
```

Because we have defined our own lock, we must also set it as our own kernel thread lock using the **kernel** pragma – this is done in the *main()* function:

```
#pragma propeller kernel my_lock
```

Also, since we have declared that `factory_2` will use a custom foreman, we have to define a suitable foreman function – we define it as follows:

```
static int my_foreman(int argc, char *argv[]) {
    _thread_set_lock(my_lock);
    printf("Foreman standing by on cog %d!\n\n", _cogid());
    while (1) {
        _thread_yield();
    }
    return 0;
}
```

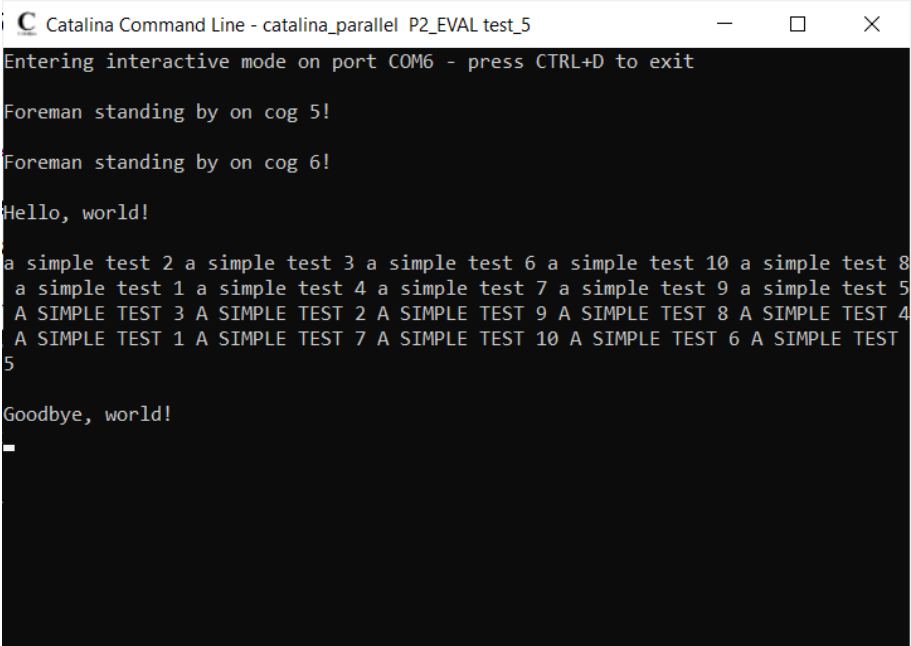
Because our factories now do things other than just execute worker code (in this case the foreman prints a message), we should manually start the factories. This allows us to determine more precisely when they are started:

```
#pragma propeller start factory_1
#pragma propeller start factory_2
```

You can execute `test_5.c` yourself as a serial and parallel program using the following commands:

```
payload -i -b230400 test_5_s
payload -i -b230400 test_5_p
```

When executed as a serial program, the output will be similar to the output of `test_4.c`. When executed as a parallel program, the output of `test_5.c` will be as follows:



```
Catalina Command Line - catalina_parallel P2_EVAL test_5
Entering interactive mode on port COM6 - press CTRL+D to exit

Foreman standing by on cog 5!

Foreman standing by on cog 6!

Hello, world!

a simple test 2 a simple test 3 a simple test 6 a simple test 10 a simple test 8
a simple test 1 a simple test 4 a simple test 7 a simple test 9 a simple test 5
A SIMPLE TEST 3 A SIMPLE TEST 2 A SIMPLE TEST 9 A SIMPLE TEST 8 A SIMPLE TEST 4
A SIMPLE TEST 1 A SIMPLE TEST 7 A SIMPLE TEST 10 A SIMPLE TEST 6 A SIMPLE TEST
5

Goodbye, world!
```

Defining your own custom foreman function is never required just to parallelize a serial program, but it may be handy if you are designing a parallel program from scratch.

Test 6

The next program we will look at is in the files `test_6a.c`, `test_6b.c` & `test_6c.c`. In this example we show that the **factory** and **worker** pragmas, and the worker code, do not all

need to appear only in a single source file. This becomes important in large C programs, which are often implemented using multiple source files. Consequently you will see a few changes in the **factory** and **worker** pragmas. Refer to these files for more details.

An important point to note is that since the files are compiled separately (and linked only when generating the final executable) each file must contain all the information (in terms of pragmas) it needs – so the **worker** pragma has to appear in multiple files, and must be identical in each file. Also note that the **factory** pragma appears in a different file to the **start** pragma, and the use of the **extern** pragma.

You can execute *test_6* yourself as a serial and parallel program using commands similar to the following:

```
payload -i -b230400 test_6_s
payload -i -b230400 test_6_p
```

Note that you may get warnings when you compiled this program, such as:

```
Warning : No code has been defined for worker 'worker_1'
Warning : No code has been defined for worker 'worker_2'
```

This is the parallelizer warning you that some files contain a *declaration* of a worker, but no actual *code segment* for that worker. In this case, this is to be expected – it is in fact the point of this example!

The output of *test_6* is similar to *test_4.c*.

Test 7

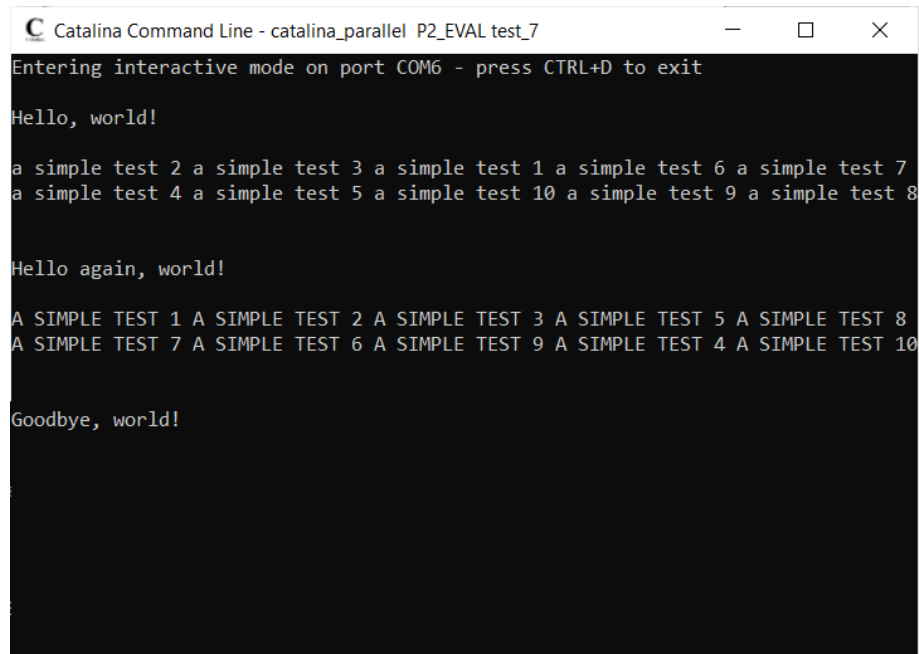
The next program we will look at is in the file *test_7.c*. In this file we show how to stop one factory and start another. Or we could restart the same factory again. This is how we can have multiple factories that both use all the available cogs – this is fine as long as they do not try and execute *at the same time*. The relevant **stop** pragma is fairly trivial, and must occur before we start another factory that will try and use the same cogs:

```
#pragma propeller stop factory_1
#pragma propeller start factory_2
```

You can execute *test_7.c* yourself as a serial and parallel program using the following commands:

```
payload -i -b230400 test_7_s
payload -i -b230400 test_7_p
```

Here is the output when executed as a parallel program:



```

Catalina Command Line - catalina_parallel P2_EVAL test_7
Entering interactive mode on port COM6 - press CTRL+D to exit

Hello, world!

a simple test 2 a simple test 3 a simple test 1 a simple test 6 a simple test 7
a simple test 4 a simple test 5 a simple test 10 a simple test 9 a simple test 8

Hello again, world!

A SIMPLE TEST 1 A SIMPLE TEST 2 A SIMPLE TEST 3 A SIMPLE TEST 5 A SIMPLE TEST 8
A SIMPLE TEST 7 A SIMPLE TEST 6 A SIMPLE TEST 9 A SIMPLE TEST 4 A SIMPLE TEST 10

Goodbye, world!

```

Test 8

The next program we will look at is in the file *test_8.c*. In this file we show something completely different – i.e. how we can use the pragmas to simply execute different parts of a program in parallel on different cogs. This example does not start multiple instances of any workers. It simply defines one worker that occupies one thread on one cog, and another worker that occupies one thread on another cog. Then, executing both segments of the program in parallel is quite trivial. Here are the relevant **factory** and **worker** pragmas:

```

#pragma propeller factory ping cogs(1)
#pragma propeller factory pong cogs(1)

#pragma propeller worker ping(void) threads(1) factory(ping)
#pragma propeller worker pong(void) threads(1) factory(pong)

```

And here is the significant part of the **main()** function:

```

#pragma propeller begin ping
ping();
#pragma propeller end ping

#pragma propeller begin pong
pong();
#pragma propeller end pong

```

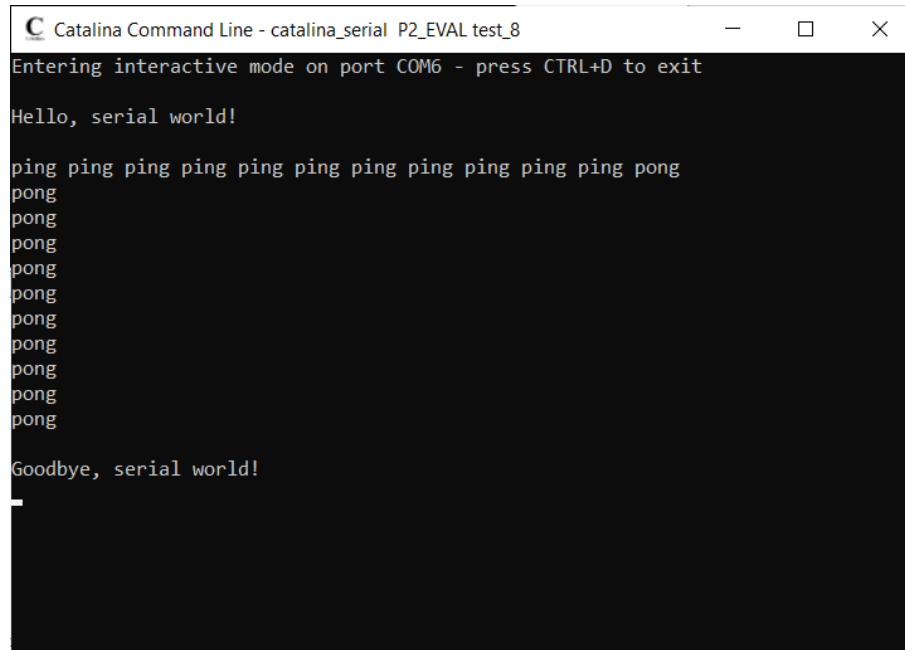
Here, **ping()** and **pong()** are not loops – they are just ordinary C code (in these cases, just a single function call each, but they could be more complex). Consult the file *test_8.c* for details. Also, note that the names given here to the factories and the workers actually has nothing to do with the names of the functions they are being used to execute in this

example – but calling them by the same names as the functions makes everything a little easier to understand.

You can execute `test_8.c` yourself as a serial and parallel program using the following commands:

```
payload -i -b230400 test_8_s  
payload -i -b230400 test_8_p
```

Here is the output when executed as a serial program:



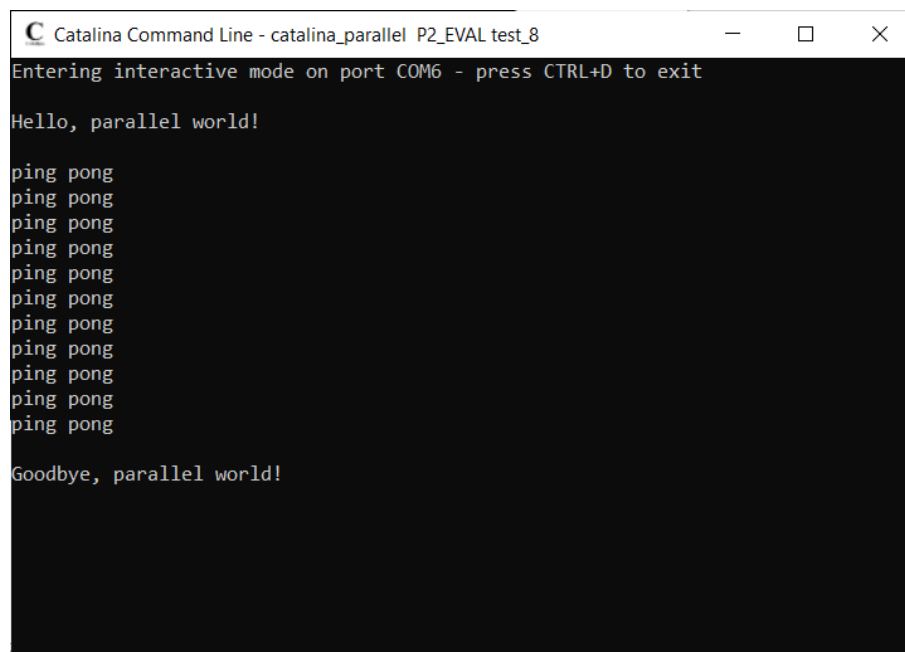
```
Catalina Command Line - catalina_serial P2_EVAL test_8
Entering interactive mode on port COM6 - press CTRL+D to exit

Hello, serial world!

ping ping ping ping ping ping ping ping ping ping ping pong
pong
pong
pong
pong
pong
pong
pong
pong
pong
pong
pong

Goodbye, serial world!
```

And here is the output when executed as a parallel program:



```
Catalina Command Line - catalina_parallel P2_EVAL test_8
Entering interactive mode on port COM6 - press CTRL+D to exit

Hello, parallel world!

ping pong
ping pong
ping pong
ping pong
ping pong
ping pong
ping pong
ping pong
ping pong
ping pong
ping pong
ping pong
ping pong

Goodbye, parallel world!
```

Neat, eh?

This technique can be extended to run as many different parts of the program on as many different cogs as we have available.

Test 9

The next program we will look at is in the file `test_9.c`. In this file we show how to use a *condition* to determine when a worker thread will output variables, and also the effect a condition has on a **wait** pragma.

This is a complex example, and it also requires us to edit the program to see the effects of the different wait pragmas. You should study the program to understand why it behaves the way it does when the different wait pragmas are used.

First of all, here is our worker pragma:

```
#pragma propeller worker(int game) local(int winner)
    output(int *winner) if(winner > 0)
    threads(NUM_GAMES) stack(60)
```

Notice the **if** option defines a *condition*, which says that this worker will only output the **winner** variable back to the main thread *if* that variable has a value **> 0**.

Next, note that there are two different variants of the wait pragmas – one *with* a condition, and *without* (one is commented out). These are around line 102 of `test_9.c`:

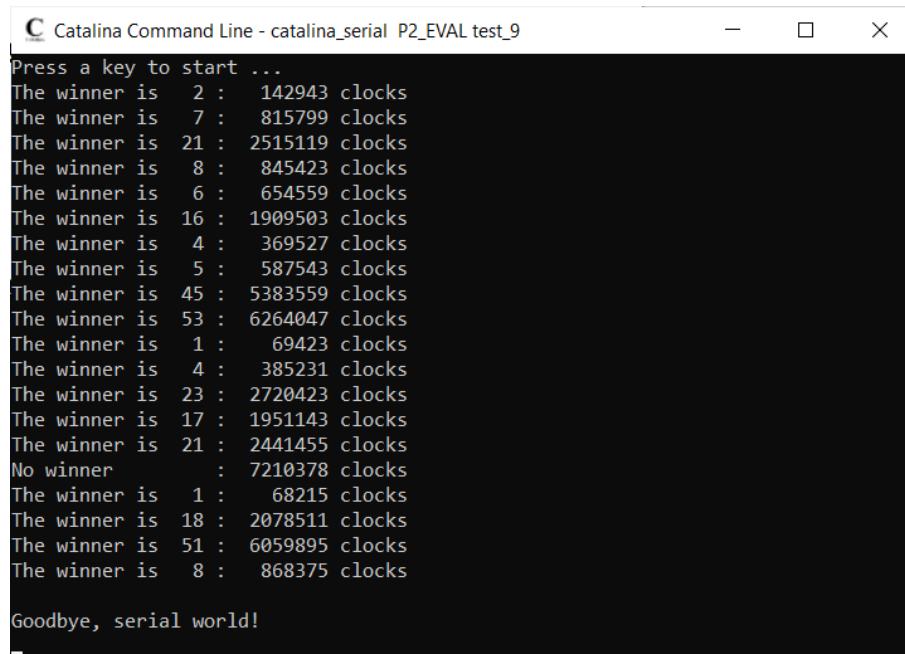
```
#pragma propeller wait
//#pragma propeller wait (winner > 0)
```

You should first execute `test_9.c` yourself as a serial program using the following command, just to see what it outputs:

```
payload -i -b230400 test_9_s
```

The program will run 20 trials of a simple game simulation, and you should see something like this⁹:

⁹ The program uses random numbers to simulate a game of chance, so the precise output will be different on every execution.



```

Catalina Command Line - catalina_serial P2_EVAL test_9
Press a key to start ...
The winner is 2 : 142943 clocks
The winner is 7 : 815799 clocks
The winner is 21 : 2515119 clocks
The winner is 8 : 845423 clocks
The winner is 6 : 654559 clocks
The winner is 16 : 1909503 clocks
The winner is 4 : 369527 clocks
The winner is 5 : 587543 clocks
The winner is 45 : 5383559 clocks
The winner is 53 : 6264047 clocks
The winner is 1 : 69423 clocks
The winner is 4 : 385231 clocks
The winner is 23 : 2720423 clocks
The winner is 17 : 1951143 clocks
The winner is 21 : 2441455 clocks
No winner : 7210378 clocks
The winner is 1 : 68215 clocks
The winner is 18 : 2078511 clocks
The winner is 51 : 6059895 clocks
The winner is 8 : 868375 clocks

Goodbye, serial world!

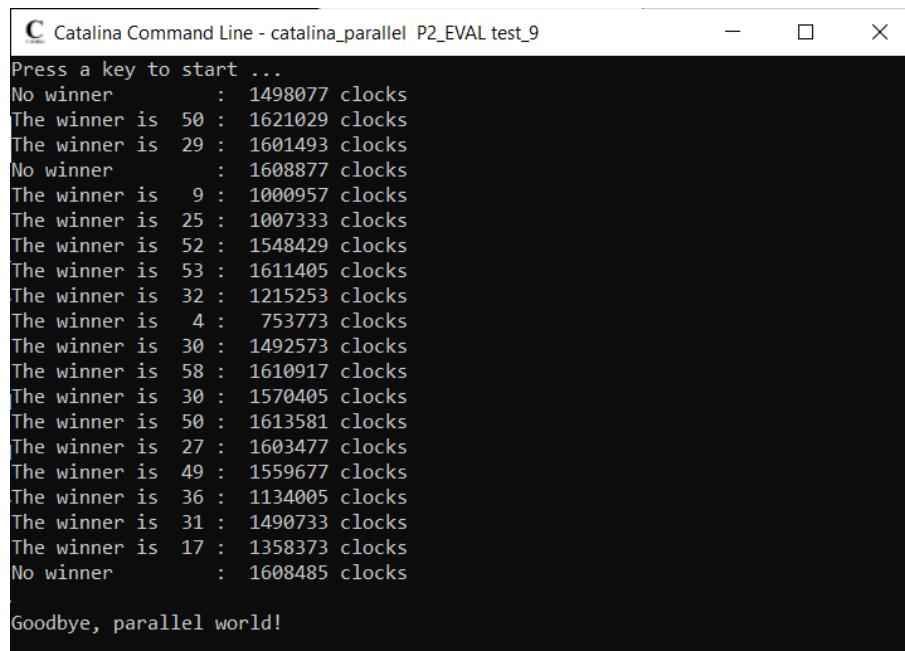
```

Here, we are mostly interested in the *shortest* time, and the *longest* time taken for the trials – in this case the program takes from 68,215 to 7,210,378 clocks to execute a trial (the longest time taken will always be a trial where no winner is found).

Next, execute `test_9.c` yourself as a parallel program using the following command:

```
payload -i -b230400 test_9_p
```

This time, you should see something like this:



```

Catalina Command Line - catalina_parallel P2_EVAL test_9
Press a key to start ...
No winner : 1498077 clocks
The winner is 50 : 1621029 clocks
The winner is 29 : 1601493 clocks
No winner : 1608877 clocks
The winner is 9 : 1000957 clocks
The winner is 25 : 1007333 clocks
The winner is 52 : 1548429 clocks
The winner is 53 : 1611405 clocks
The winner is 32 : 1215253 clocks
The winner is 4 : 753773 clocks
The winner is 30 : 1492573 clocks
The winner is 58 : 1610917 clocks
The winner is 30 : 1570405 clocks
The winner is 50 : 1613581 clocks
The winner is 27 : 1603477 clocks
The winner is 49 : 1559677 clocks
The winner is 36 : 1134005 clocks
The winner is 31 : 1490733 clocks
The winner is 17 : 1358373 clocks
No winner : 1608485 clocks

Goodbye, parallel world!

```

This time, the program takes from 753,773 to 1,608,877 clocks to execute a trial (again, the longest time taken will always be a trial where no winner is found). So by parallelizing this program we have managed to reduce the *longest* time taken by a factor of 4.5 times,

but the *shortest* time taken looks like it has actually *increased* – this is because our wait pragma causes us to wait for all the worker threads to complete, *even though* one of the threads may have found a winner.

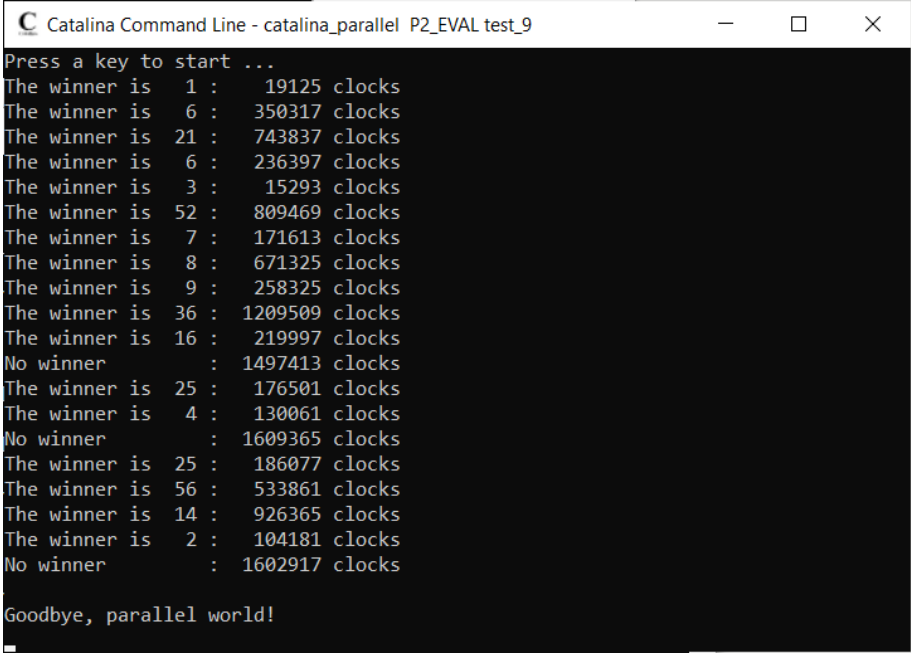
We can fix this problem by editing the program to use the *other* version of the wait pragma:

```
//#pragma propeller wait
#pragma propeller wait (winner > 0)
```

Then re-run the **build_all** script and run the program again as a parallel program:

```
payload -i -b230400 test_9_p
```

This time, you should see something like this:



```
Catalina Command Line - catalina_parallel P2_EVAL test_9
Press a key to start ...
The winner is 1 : 19125 clocks
The winner is 6 : 350317 clocks
The winner is 21 : 743837 clocks
The winner is 6 : 236397 clocks
The winner is 3 : 15293 clocks
The winner is 52 : 809469 clocks
The winner is 7 : 171613 clocks
The winner is 8 : 671325 clocks
The winner is 9 : 258325 clocks
The winner is 36 : 1209509 clocks
The winner is 16 : 219997 clocks
No winner : 1497413 clocks
The winner is 25 : 176501 clocks
The winner is 4 : 130061 clocks
No winner : 1609365 clocks
The winner is 25 : 186077 clocks
The winner is 56 : 533861 clocks
The winner is 14 : 926365 clocks
The winner is 2 : 104181 clocks
No winner : 1602917 clocks
Goodbye, parallel world!
```

This time, the program takes from 15,293 to 1,609,365 clocks to execute a trial (again, the longest time taken will be a trial where no winner is found). So by parallelizing this program *and* using a condition we have managed to reduce the *longest* time taken by a factor of 4.5 times, without sacrificing the advantage of exiting the loop when we find a winner (i.e. the *shortest* time taken).

This demonstrates the advantages of **wait** pragmas *with a condition*. But it is important to also note the condition specified in the **worker** pragma, which prevents the winner that has been found by one thread being overwritten by another thread that has *not* found a winner. It is *common* for the conditions in the worker and wait pragmas to be identical, but it is not actually *required*.

Test 10

Test 10 is a demonstration that, while **begin** and **end** pragmas cannot be *nested*, one parallelized worker segment *can* execute *another* parallelized worker segment, provided it

is in a different part of the source code (such as within a function). The program itself is very similar to the previous example (i.e. Test 9).

Here are our worker pragmas:

```
#pragma propeller worker play(int game) local(int winner)
    output(int *winner) if(winner > 0)
    threads(NUM_GAMES) stack(60)

#pragma propeller worker dice(int roll) local(int result)
    output(int *result) if (result >= 9996)
    threads(10) stack(60)
```

Here is the relevant worker code segment for the **play** worker, which just calls the *play()* function:

```
#pragma propeller begin play
winner = play(game);
#pragma propeller end play
```

But now examine the *play()* function:

```
int play(int game) {
    ...
    #pragma propeller begin dice
    result = random(10000);
    #pragma propeller end dice
    ...
}
```

So the *play()* function itself contains a parallelized code segment called **dice**. This is perfectly acceptable.

You can execute *test_10.c* yourself as a serial and parallel program using the following commands:

```
payload -i -b230400 test_10_s
payload -i -b230400 test_10_p
```

When you do do, and study the times displayed, you will learn another very important lesson – i.e. that sometimes parallelizing a program can make it *slower*, not *faster*!

In this case, this is because the code in the **dice** code segment is so trivial that the overheads of parallelizing it make the whole program run slower. But this is just an example – we did it mainly to demonstrate that we *can*!

fftbench

Our final example is another “real world” program – a common benchmark used on the Propeller. It is again instructive to examine the source code to see how *few* pragmas are typically required in the real world to effectively parallelize a program.

First, a **worker** pragma near the top of the program:

```
#pragma propeller worker(int slice, int slices, int firstLevel, int
lastLevel) local(int s, int slen) threads(1<<LOG2_SLICES) ticks(1000)
```

In the body of the FFT algorithm we have **begin**, **end** and **wait** pragmas:

```
for (slice = 0; slice < slices; slice++) {
    #pragma propeller begin
    s = FFT_SIZE * slice / slices;
    slen = FFT_SIZE / slices;
    butterflies(&bx[s], &by[s], firstLevel, lastLevel, slices, slen);
    #pragma propeller end
}
#pragma propeller wait
```

In the C **main()** function we have a **start** pragma:

```
#pragma propeller start
```

That's all that's required – very similar to the pragmas we used to parallelize the Sieve program earlier.

You can execute *fftbench.c* yourself as a serial and parallel program using the following commands:

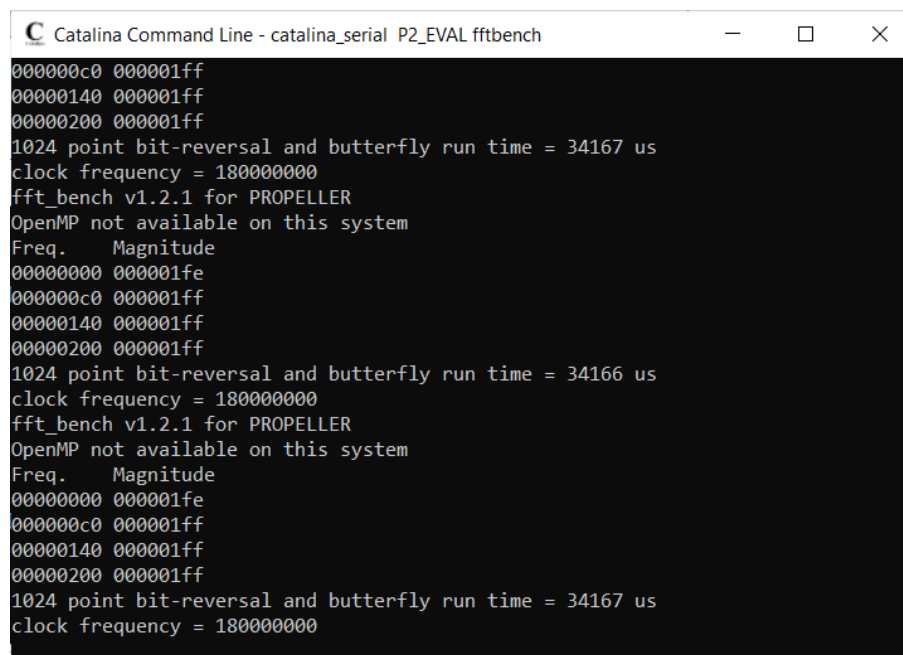
```
payload -i -b230400 fftbench_s
payload -i -b230400 fftbench_p
```

Note: you may get a warning similar to the following when you compiled the *fftbench* program:

```
Warning: non-propeller pragma ignored
```

This is the parallelizer indicating that it has found a non-propeller pragma in the program – this warning is just in case a pragma has been incorrectly specified (e.g. by accidentally omitting the word **propeller** – in such cases, the pragma would otherwise be silently ignored).

Here is the output when executed as a serial program:



```
Catalina Command Line - catalina_serial P2_EVAL fftbench
000000c0 000001ff
00000140 000001ff
00000200 000001ff
1024 point bit-reversal and butterfly run time = 34167 us
clock frequency = 180000000
fft_bench v1.2.1 for PROPELLER
OpenMP not available on this system
Freq.    Magnitude
00000000 000001fe
000000c0 000001ff
00000140 000001ff
00000200 000001ff
1024 point bit-reversal and butterfly run time = 34166 us
clock frequency = 180000000
fft_bench v1.2.1 for PROPELLER
OpenMP not available on this system
Freq.    Magnitude
00000000 000001fe
000000c0 000001ff
00000140 000001ff
00000200 000001ff
1024 point bit-reversal and butterfly run time = 34167 us
clock frequency = 180000000
```

This tells us the algorithm took 34,167 microseconds to perform the FFT calculations.

Here is the output when executed as a parallel program:

```

Catalina Command Line - catalina_parallel P2_EVAL fftbench
000000c0 000001ff
00000140 000001ff
00000200 000001ff
1024 point bit-reversal and butterfly run time = 14162 us
clock frequency = 180000000
fft_bench v1.2.1 for PROPELLER
OpenMP not available on this system
Freq.    Magnitude
00000000 000001fe
000000c0 000001ff
00000140 000001ff
00000200 000001ff
1024 point bit-reversal and butterfly run time = 14182 us
clock frequency = 180000000
fft_bench v1.2.1 for PROPELLER
OpenMP not available on this system
Freq.    Magnitude
00000000 000001fe
000000c0 000001ff
00000140 000001ff
00000200 000001ff
1024 point bit-reversal and butterfly run time = 14165 us
clock frequency = 180000000

```

This tells us the algorithm took 14,165 microseconds to perform the FFT calculations.

So, again, just like the Sieve example we saw earlier in this tutorial, just by adding a few pragmas to our program we have enabled it to run using multiple cogs and in doing so have made the program execute *2.5 times faster!*

Conclusion

Catalina's propeller pragmas allow an easy way to significantly improve the performance of a C program by allowing it to use all the available cogs on the Propeller – and without compromising the portability of the C program.

The Catalina parallelizer is still under development – but is already fully functional, and already sufficiently powerful for real-world applications.

The Catalina parallelizer can also be used stand-alone, to allow parallelized programs to be compiled with other C compilers and other thread libraries, such as GCC and Posix Threads¹⁰.

Feedback on the pragmas and the pragma preprocessor is welcome – particularly in cases where pragmas are correctly specified in an input program, but the output program does not behave correctly. Feedback can be emailed to ross@thevastydeep.com, or posted in the Parallax forums (<https://forums.parallax.com>).

¹⁰ The pragma preprocessor essentially only needs to know how to start and stop threads on cogs to be used with *any* C compiler that has a threads library. An example of using it with GCC and Posix threads is provided.