Exploring Parallel Processing with Catalina C on the Parallax Propeller

by Ross Higson

Introduction

The Parallax Propeller chips are extraordinary beasts (see www.parallax.com/propeller for full details). The original Propeller chip (now known as the Propeller 1 or P1) had 8 32-bit processors with 512 32-bit longs of RAM local to each processor, and 32kb of shared RAM that could be used by all processors, with access to that RAM provided on a “round robin” basis, so each processor could access the shared RAM 1/8th of the time.

The new Propeller 2 (also known as the P2) also has 8 32-bit processors, but each one has 1024 32-bit longs of RAM local to each processor, and there is 512kb of shared RAM that can be accessed by all processors using a unique shared RAM design which means that each processor can potentially access the shared RAM at *any* time – but a different 1/8th slice of it at any *one* time, with each of the 8 processors in turn accessing a different 1/8th slice.

If this sounds complex, don’t worry. While it has some implications for the performance of the chip, particularly when executing high-level languages (which generally execute from the shared RAM), we can ignore this for the moment, and just think of the Propeller simply as a computer with 8 independent general-purpose processors.

Catalina is a free C89/C99 ANSI C compiler and run-time environment that was originally developed for the Propeller 1 by Ross Higson. It is based on **lcc** (the “little C compiler”), a portable retargetable C compiler developed by Christopher Fraser and David Hanson. Catalina is available on Windows and Linux.

Support for the new Propeller 2 chip is a recent addition to Catalina, and the Propeller 2 will be the main focus of this paper. However, it is worth pointing out that *all the programs* mentioned will also compile and run perfectly well on a Propeller 1 chip. They just need to be smaller, due to the smaller amount of RAM available on the Propeller 1.

The focus of this paper is exploiting the multi-processor capabilities of the Propeller. We will look at what is required to implement a parallel version of a very simple sequential algorithm in a high-level language – in this case, the C language – using the Catalina C compiler.

Terminology

Before we proceed, we need a little bit of terminology. We don’t need to go into all the details of the Propeller architecture (see the documentation available on the Parallax web site for that), but you will see many terms littered throughout nearly all Propeller code and documentation (including this one) that may be unfamiliar, so it is worth getting to know a few of them:

Cog	Each of the 8 processors is called a “cog” - this is because the way they operate and interact with the central hub is reminiscent of many small cogs rotating around a larger central drive gear which gives access to the shared resources to each of the cogs in turn.
Hub	The circuitry and resources shared between all processors (cogs) is known as the “hub”.
Cog RAM	The RAM that is dedicated to each processor is referred to as “cog RAM”.
Hub RAM	The RAM that is shared between all processors is referred to as “hub RAM”.

Register	Another name for the 512 (P1) or 1024 (P2) 32-bit longs of cog RAM – this is because as well as containing code for local execution, any of the longs (with a couple of exceptions) can be used as a general-purpose register by the Propeller’s native instruction set.
Kernel	A program that resides in cog RAM and executes other program code – code which is generally located in hub RAM.
Lock	A mutual-exclusion lock that can be used to coordinate access to shared resources. The hub provides 8 such locks, which can be acquired and used by any cog.
Spin	The original high-level language designed by Parallax for the Propeller. It is a byte-coded language interpreted by a kernel that executes in one or more cogs. Spin has built-in support for multi-processing, but being essentially a proprietary language it has had a mixed reception.
LMM	Large Memory Mode (or Model). The original Propeller executed native instructions only from cog RAM, and to execute code from Hub RAM a byte-oriented interpreted language called Spin was developed by Parallax. An early developer, Bill Henning, realized that a very tight loop of only a few native instructions in cog RAM could also fetch and execute native instructions from hub RAM, and he used this technique to extend the RAM available for native mode instructions from the limited cog RAM to the entire hub RAM. LMM code executes more slowly than cog code, but the LMM technique allows much larger programs to be created that can effectively use the native instruction set that was previously limited to executing only from cog RAM. This facilitated high-level language support on the Propeller 1, and the original set of high-level language compilers (including Catalina) was born. While LMM is sometimes also used on the Propeller 2, the Propeller 2 can also natively execute instructions from hub RAM – see also “NMM”.
CMM	Compact Memory Mode (or Model). This technique uses a similar execution mode to LMM, but makes use of a compressed instruction set that typically compresses programs to nearly half the size of the same LMM programs. Each instruction has to be uncompressed before execution, so CMM mode is even slower than LMM mode. But the limited memory on the Propeller 1 chip made it attractive, and it still finds use even on the larger Propeller 2.
NMM	Native Memory Mode (or Model). This is unique to the Propeller 2. It is also referred to as “Hub Execution” mode. It makes the Propeller a bit more similar to traditional processor architectures, where program code is generally in hub RAM, and the registers (i.e. the cog RAM) is generally used to hold data, not code.

The Sieve of Eratosthenes

The best way to to explore parallel programming from a practical perspective is with a concrete example of a complete algorithm that is simple, familiar, and easy to turn from a sequential algorithm to a parallel algorithm so that we can easily measure the benefits.

Surprisingly, good examples of such algorithms are not so easy to find. But the “Sieve of Eratosthenes” is one such. Some people may remember the sieve from their school days – it is a means of enumerating prime numbers by simply writing down a sequential list of *all* numbers and then crossing any any numbers that are multiples of any others in the sieve, other than 1 or itself (recall that a prime number is a number that is not a multiple of any numbers other than 1 and itself). When you have crossed out all the numbers you can, the ones that are left in the sieve are all the *prime* numbers.

Without further ado, let’s see the Sieve of Eratosthenes, implemented in C:

```
/******************************************************************************
 *                                                                            *
 *                         The Sieve of Eratosthenes                          *
 *                                                                            *
 ******************************************************************************/

#include <stdio.h>
#include <stdlib.h>

/*
 * define the size of the sieve (if not already defined):
 */
#ifndef SIEVE_SIZE
#define SIEVE_SIZE   100
#endif

/*
 * main : allocate and initialize the sieve, then eliminate all multiples
 *        of primes, then print the resulting primes.
 */
void main(void){
   unsigned long i, j;
   unsigned long k = 1;
   unsigned char *primes = NULL;

   // 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;
   }

   // eliminate multiples of primes
   for (i = 2; i < SIEVE_SIZE/2; i++) {
      if (primes[i] == 0) {
         for (j = 2; i*j < SIEVE_SIZE; j++) {
            primes[i*j] = 1;
         }
      }
   }

   // print the resulting primes, starting from 2
   for (i = 2; i < SIEVE_SIZE; i++) {
      if (primes[i] == 0) {
         printf("prime(%d)= %d, ", k++, i);
      }
   }

   while(1);
}
```

You will find all the programs discussed in this document, and scripts to compile those programs, in the folder **demos\sieve** in the directory where you installed Catalina (you must be using Catalina 4.3 or later). You will find the program listed above in a file called **sieve_original.c**.

You can very easily execute this program on the Propeller (1 or 2). Here are the actual commands to use to do so using the Catalina C compiler (**catalina**) and its program loader (**payload**) on a Propeller 2[1]:

```
catalina -p2 sieve_original.c -lci -C TTY
payload sieve_original -i -b230400
```

If you are using a Propeller 1, omit the **-p2** parameter:

```
catalina sieve_original.c -lci -C TTY
payload sieve_original -i
```

When you load and execute the program using **payload**, you will see output similar to this:

```
Entering interactive mode on port /dev/ttyUSB1 - press CTRL+D to exit

prime(1)= 2, prime(2)= 3, prime(3)= 5, prime(4)= 7, prime(5)= 11, prime(6)= 13,
prime(7)= 17, prime(8)= 19, prime(9)= 23, prime(10)= 29, prime(11)= 31, prime(12
)= 37, prime(13)= 41, prime(14)= 43, prime(15)= 47, prime(16)= 53, prime(17)= 59
, prime(18)= 61, prime(19)= 67, prime(20)= 71, prime(21)= 73, prime(22)= 79, pri
me(23)= 83, prime(24)= 89, prime(25)= 97,
```

So, the 25th prime number is 97. Very interesting ... but let's face it, a sieve size of only 100 numbers is a bit lame. We can do that with a pencil and paper, and very likely did so in school!

So let's first crank it up a bit … we can specify a larger sieve size on the command line as follows:

```
catalina -p2 sieve_original.c -lci -C TTY -D SIEVE_SIZE=400000
payload sieve_original -i -b230400
```

A sieve size of 400,000 is only possible on the Propeller 2. On the Propeller 1, you would instead only be able to go to a sieve size of about 12,000 – i.e:

```
catalina sieve_original.c -lci -C TTY -D SIEVE_SIZE=12000
payload sieve_original -i
```

If you do this, you will find that the 1438th prime is 11987 (on the Propeller 1) or that the 33860th prime is 399989 (on the Propeller 2).

Now, that's more like it! We can't very easily do *that* with a pencil and paper!

From here on, as we work through different versions of the sieve program, we will not give the individual commands. Instead, you should just compile all the programs using the **build_all** batch script, specifying the Propeller platform you have as the first parameter, and any memory model or Human/Machine Interface (HMI) options as subsequent parameters. This script will automatically apply sieve sizes appropriate for the Propeller 1 or 2.

For example:

```
build_all P2_EVAL TTY NATIVE
```

or:

```
build_all FLIP TTY
```

We will assume that all the programs mentioned have been compiled using these scripts, and executed using **payload** (don't forget the **-i**).

Note that on both the Propeller 1 and the Propeller 2, you should specify **TTY** as your HMI option (it is the HMI option that uses least RAM, allowing us the biggest sieves), and on the Propeller 2 you should specify **NATIVE** mode (the fastest). On the Propeller 1 you do not need to specify any mode - LMM will be used.

1 On the Propeller 2, serial programs use 230400 baud by default, so you must specify either set the environment variable **PAYLOAD_BAUD** to 230400, or specify this baud rate using **-b** on each payload command.

Parallelizing the Sieve Algorithm

The Sieve of Eratosthenes is a very simple sequential (or “single-threaded”) algorithm. So can we turn it into a program that can exploit the parallel processing capabilities of the Propeller?

Of course, the answer is *“Yes we can!”*

The first thing is to consider the following part of the program:

```
   ...

   // eliminate multiples of primes
   for (i = 2; i < SIEVE_SIZE/2; i++) {
      if (primes[i] == 0) {
         for (j = 2; i*j < SIEVE_SIZE; j++) {
            primes[i*j] = 1;
         }
      }
   }

   ...
```

This is pretty much the the entire sieve algorithm in a couple of lines of code. The inner loop here does a very simple job – it iterates through the `primes` array eliminating multiples of each number identified in the outer loop. But for a sieve containing hundreds of thousands of numbers, both the inner and the outer loops must be executed hundreds of thousands of times. That’s going to take some time.

But what if we could execute multiple instances of that inner loop *in parallel* – i.e. on *multiples* of the numbers in the outer loop at the same time? That would speed up the algorithm quite dramatically.

It is worth noting here that each iteration of the inner loop is *independent* – and so they can be executed in any order. Yes, it is true that if we happen to (say) start to eliminate multiples of 4 before eliminating all the multiples of 2, then we will end up duplicating some work – but this does not invalidate the process, and if we can truly do these operations in parallel, then this potential duplication of work will not even cost us any time!

These facts make the sieve algorithm a good candidate for the application of parallel processing, and the Propeller is a good candidate for the job!

How? Well, the first (and easiest) step is to move the inner loop into a function of its own. We do that in the file `sieve_single_threaded.c`, which contains the following sections of code – our new function – which simply implements the inner loop of the original algorithm:

```
   ...

/*
 * eliminate_multiples: eliminate all multiples of a prime from the sieve
 /
void eliminate_multiples(unsigned long int i) {
   unsigned long j;
   // eliminate multiples
   for (j = 2; j < SIEVE_SIZE; j++) {
      primes[i*j] = 1;
   }
}

   ...
```

And the new outer loop that invokes this function is as follows:

```
   ...

   // eliminate multiples of primes
   for (i = 2; i < SIEVE_SIZE/2; i++) {
      if (primes[i] == 0) {
         eliminate_multiples(i);
      }
   }

   ...
```

This may look like we are adding overhead to our algorithm, and so we are – we are adding a function call. But the overhead of adding one function call per iteration of the outer loop is small compared to the execution time of the inner loop *within* that function – and doing this makes the next steps conceptually easier.

This version of the sieve program also adds a few other housekeeping items, such as timing calculations, which will allow us to determine how long the algorithm takes to execute. We time the outer loop using the system clock counter (which makes the results independent of the actual clock frequency) and print it once the outer loop is complete. These changes do not alter the basic operation of the sieve algorithm - that comes in the next step.

But first, lets execute this version, because this is the first version that we can *time*. If we do so on a Propeller 2, we will see output similar to the following:

```
Entering interactive mode on port /dev/ttyUSB1 - press CTRL+D to exit

starting...
... done - 227292729 clocks

press a key to see results
```

This tells us it took the program 227,292,729 clocks to complete the sieve algorithm on a sieve size of 400,000 numbers. [2]

This is the number we will have to beat – and beat substantially – if we are going to claim that a parallel version of this algorithm is worth the additional resources and effort.

2 The actual number may differ depending on the version of Catalina being used.

Multithreading

Before we start worrying about using multiple *physical* processors, we will turn our `eliminate_multiples()` function into something that we can execute using multiple *logical* processors.

Why? Well, consider our sieve program – in an ideal world, eliminating primes from an array of 400,000 numbers would require up to 200,000 separate sub-tasks (i.e. executions of the `eliminate_multiples()` function). For the fastest result, these should all be executed *simultaneously* – but we don't have 200,000 physical processors. Nor are we ever likely to! But we want to make best use of whatever processors we *do* have.

It is also the case that the sub-tasks we have to execute in order to implement our algorithm may not map well to physical processors – they may be too simple (as they are in the case of our sieve program) and not worth the effort of assigning individually to a physical processor, or else too complex, possibly requiring multiple physical processors to implement each sub-task.

So what we do instead is first execute our sub-tasks on *logical* processors, and then assign those logical processors to whatever *physical* processors we have available.

This also has the benefit that we are not constrained should the physical processors not all be available when we need them – we divide the algorithm logically, not physically, knowing that we will always be able to execute our logical processors using whatever physical processors happen to be available at the time, using multi-threading if necessary. In purely practical terms, it would be silly to design an algorithm that depended on having *200,000* physical processors available, and then discovering at run time that you had only *five!*

With C, as with other fundamentally sequential languages, we can most easily do this by using *threads* – i.e. independent threads of execution that can potentially be executed in parallel. The overhead of starting or stopping a thread – especially one executing on another physical processor – is still quite high, so we definitely don't want to have to do this for each *individual* sub-task – this would potentially negate any benefit unless we truly *did* have a very large number of physical processors.

Instead, what we we will do is create a *worker thread* – a logical process that stays active as long as we need it, and which – once started – we can use to perform sub-tasks as we need without further overhead. Minimizing the overhead of assigning a task to a worker thread, or determining when that work is complete, will make a significant difference to the final outcome.

So before we move on to true parallel processing using multiple physical processors, we will turn our program into a multi-threaded version that still executes on a single physical processor. This will generally not give us any improvement in execution time. When a multi-threaded program is run on a single physical processor, the threads are not really executed in parallel – at best, they are executed by interleaving the execution of small chunks of each of them until all of them are eventually complete. The overheads of this interleaving is why multi-threading may end up *slowing down* an algorithm instead of speeding it up. But we will do it anyway, because while this step is a little complex, it makes the subsequent step – i.e. mapping these worker threads onto physical processors – very much simpler

So let's have a look at the version of the sieve called `sieve_multi_threaded.c` – this version turns the `eliminate_multiples()` function into a worker thread, and uses Catalina's thread functions to execute multiple instances of them using the capabilities built into Catalina's multi-threading kernel.

Here are the significant new portions of that program – a new **eliminate_multiples()** function, and a global array (**my_prime**) which the main program can use to communicate with each of the individual worker threads:

```
   ...

/*
 * define a place to put the prime number a worker thread should process
 * (0 means thread is either not started, or is waiting for a new prime)
 */
static unsigned long int my_prime[NUM_WORKERS] = { 0 };
   ...

/*
 * eliminate_multiples: eliminate all multiples of a prime from the sieve
 *
 * This is our worker thread function - note that the "me" parameter passed is
 * a thread number, which the worker uses to look up their allocated prime
 * in the array "my_prime" - this is just an easy way of passing a parameter
 * other than a plain int, and also indicating completion of our task.
 */
int eliminate_multiples(int me, char *unused[]) {
   unsigned long i;
   unsigned long j;

   while (1) {
      // get my allocated prime
      while ((i = my_prime[me]) == 0) {
         idle();
      }
      // eliminate multiples
      for (j = 2; i*j < SIEVE_SIZE; j++) {
         primes[i*j] = 1;
      }
      // indicate we are done
      my_prime[me] = 0;
   }
   return 0;
}

   ...
```

You should recognise the section highlighted in red as the *inner* loop from the original sieve algorithm.

The code *before* the section in red is the worker thread waiting to be assigned a task to process, and the code *after* it is the worker thread indicating that it has completed its assigned task.

A significant change to note in this version is that the **eliminate_multiples()** function no longer directly accepts the number it is to work on. Instead, it accepts a thread number (**me**) on start-up that tells it where to look in a global array (**my_prime**) to find its assigned task. The number to be processed by the worker is put into the appropriate entry of the array by the main program, and the worker replaces that value with zero when it is finished processing the number.

This provides an easy and efficient way for the main program to assign a task to each worker thread, and for that thread to report the completion of that task back to the main program.

Here is the code to start a worker thread, implemented in a new `create_worker()` function. The function must allocate stack space for the worker thread to use, and then uses the functions `_thread_start()` and `_thread_ticks()` from the Catalina thread library (the details of these functions not that important here, but if you are interested you can consult the Catalina documentation for more details on these and any of the other Catalina thread library functions):

```
   ...

/*
 * create_worker: create a worker thread.
 */
void *create_worker(_thread worker, int stack_size, int ticks, int argc, char *argv[])
{
   long *s = NULL;
   _thread *w = NULL;

    s = malloc(STACK_SIZE*4);
    if (s != NULL) {
       w = _thread_start(worker, s + STACK_SIZE, argc, argv);
       if (ticks > 0) {
          _thread_ticks(w, ticks);
       }
    }
    return w;
}

   ...
```

And here is the code that calls the `create_worker()` function to create some worker threads:

```
   ...

   // create workers
   for (i = 0; i < NUM_WORKERS; i++) {
      if (create_worker(&eliminate_multiples, STACK_SIZE, 100, i, NULL) == NULL) {
          t_printf("Cannot create worker\n");
          exit(1);
      }
   }

   ...
```

In this program, the worker threads are never specifically terminated (they will be terminated when the program terminates) but in the real world you would probably stop the worker threads once they are no longer needed. You can do this using the thread library function `_thread_stop()`. Typically, the threads would terminate *themselves* on some signal – such as if they were passed -1 as the number to process, or if a global variable was set to a particular value.

It is worth noting that the `NUM_WORKERS` value is quite arbitrary – we can start as many worker threads as we like, because these are *logical* processors, not *physical* ones. It is true that how many we start will partly depend partly on how many *physical* processors we have to execute them, but it mostly depends on the function the logical processor implements – in some cases we will start many more logical processors than we have physical processors, and in other cases we may start only one (it can be hard to predict how many logical processors will best accomplish a given algorithm in the least time, and trial and error may be the easiest way to determine this).

The new version of the sieve algorithm that assigns tasks to the worker threads is shown below. You will recognise the section highlighted in red as the *outer* loop of the sieve algorithm, and rest of the code is the program waiting for a free worker thread, and then assigning it a task to do whenever it finds one:

```
   ...
/*
 * next_worker - move to the next worker
 */
#define next_worker(j)  (j = (j + 1)%NUM_WORKERS)

   ...

   // eliminate multiples of primes
   j = 0;

   for (i = 2; i < SIEVE_SIZE/2; i++) {
      if (primes[i] == 0) {
         // find a free worker thread, waiting as necessary
         while (my_prime[j] != 0) {
            next_worker(j);
            if (j == 0) {
               idle();
            }
         }
         // found a free worker thread, so give them some work!
         my_prime[j] = i;
         next_worker(j);
      }
   }

   ...
```

One final piece of new code is required - once it has assigned all the work to worker threads, the sieve algorithm must wait until all the worker threads have completed their assigned tasks before it can declare the algorithm complete.

This is what that section of code looks like:

```
   ...

   // wait till all worker threads complete
   j = 0;

   while (1) {
      if (my_prime[j] != 0) {
         idle();
      else {
         next_worker(j);
         if (j == 0) {
            break;
         }
      }
   }

   ...
```

If we execute this this version on a Propeller 2, we will find that it takes 285,508,385 clock ticks to complete the sieve algorithm. So we have indeed gone backwards! But this was expected, so let's just proceed to the next step and see how we do when we introduce multiple *physical* processors.

Multiprocessing

The final step on our path to parallelizing our sieve algorithm is to map our *logical* processors onto the available *physical* processors.

But if you are expecting this step to be more difficult than the previous one, then think again. As was hinted in the previous section, the hard part was making it possible for our sub-tasks to be executed by a worker thread. The next step is quite trivial by comparison.

To do this, instead of just creating worker threads in the context of our main program (and which would just execute on the same physical processor as the main program) we will first create a *factory*, assigning it as many cogs to work with as we want, and then create the workers *in the context of the factory*.

The factory is responsible for assigning the workers it is given to whatever cogs it has available. And, importantly, we generally don't need to get involved in the details of how this is done.

Here is the code that does it:

```
...

// create a factory with any available cogs
f = create_factory(ANY_COG, STACK_SIZE, kernel_lock);
if (f == NULL) {
    t_printf("Cannot create factory\n");
    exit(1);
}

// create workers who will work in the factory
for (i = 0; i < NUM_WORKERS; i++) {
   if (create_worker(f, &eliminate_multiples, STACK_SIZE, 100, i, NULL) == NULL) {
      t_printf("Cannot create worker\n");
      exit(1);
   }
}
...
```

The rest of the final version of our sieve program is mostly the same as the previous version, except that we don't even have to provide the `create_worker()` function any longer – this can also done by the factory, because it requires exactly the same code for any worker. But note that the factory version of `create_worker()` takes the factory as its first parameter.

You can find this final version of our sieve program in the file **sieve_multi_processor.c**.

But before we explain a bit more about what it going on here, let's *execute* this version, to see if we have actually achieved any improvement in the execution time. It would certainly be a bit pointless if we had not!

When we do that, we find that on a Propeller 2, our final version of the sieve takes 61,941,305 clocks to execute. Remember that our original version took 227,292,729 clocks.

So our parallel version of the Sieve of Eratosthenes, using (in this case) 5 physical processors to implement the sieve algorithm instead of 1, is almost *4 times faster* than the original version.

In theory, one might expect that this multi-processor version should be *5 times* faster. And, in fact it comes close – but it close to 5 times faster than the *multi-threaded* version, not the single-threaded version. So we lost some ground when we turned our single-threaded version into a multi-threaded version, but we then gained many times more than that back again when we turned the multi-threaded version into a multi-processor version.

I call *that* a win!

On a Propeller 1, we don't do quite so well in the end, but we still end up with a performance improvement of over *3 times* that of the single threaded version.

So how was this achieved? Well, one of the main purposes of the thread factory is to *isolate* us from such details, because they are not important to the job at hand. We can just *use* the thread factory, without understanding the internals. However, the internal details are all available in the file **thread_factory.c**, and they take only about a page of fairly simple C code. The current (prototype) implementation of the thread factory uses some of the same functions we used in our own **create_worker()** process. But we no longer need to do this ourselves if we use the factory, so in fact our multi-processor version is also *simpler* than our multi-threaded version.

It is worth studying **sieve_multi_processor.c** to see how the pieces fit together, and also to see how a similar process might be applied to other sequential algorithms.

Here is the part we *do* need to know - the definition of the thread factory itself, and its main functions. This is from the file **thread_factory.h**:

```
   ...

/*
 * declare a type for our factory:
 */
typedef struct factory_struct FACTORY;

/*
 * create_factory : create a factory with the specified number of cogs, the
 *                  specified stack size for each cog, and use the specified
 *                  kernel lock.
 */
FACTORY *create_factory(int num_cogs, int stack_size, int kernel_lock);

/*
 * create_worker : create a worker thread to work in the specified factory,
 *                 using the specified worker function, and with the specified
 *                 stack size and ticks between context switches. The worker
 *                 will be passed the specified argc and argv parameters.
 */
void *create_worker(FACTORY *factory, _thread worker, int stack_size, int ticks,
int argc, char *argv[]);

   ...
```

The key functions are the ones that first *create* the factory, and then create workers *in* that factory.

It is worth pointing out that you can create a factory that uses *all* the available cogs (by specifying the value **ANY_COG**) or with a *specified number* of cogs. We can create a factory which uses just one cog if we want to, and we can also create multiple factories if we want to. This might be useful if (for instance) we had several different algorithms to implement, and we did not want the performance of any one of them to affect the performance of the others.

Another thing to note is that we are not limited to only assigning one *type* of worker to each factory. We can create different worker threads for doing different tasks in the same factory. Unlike a real factory, *our* factory does not care what its workers actually *do*.

The process we used in this paper to speed up the Sieve of Eratosthenes algorithm can be used on other sequential algorithms. The difficult part is *identifying* the sub-tasks of an algorithm that can be improved by executing them in parallel. Many sequential algorithms simply do not *have* any suitable sub-tasks. The two main things we need to look for are:

1. That the sub-tasks of the algorithm are *independent* (otherwise we need to worry about their interactions); and
2. That there are *enough* sub-tasks, or they are *complex* enough, to make executing them in parallel worthwhile – remember that during the process of parallelizing our algorithm we introduced necessary overheads – and we don't want to end up with the benefits outweighing the costs.

Conclusion

We have shown how even a very simple algorithm such as the Sieve of Eratosthenes can have its execution speed dramatically improved by exploiting parallelism.

And we have also shown that the Parallax Propeller (1 or 2) is a great chip for demonstrating parallel processing, and that Catalina provides a simple means of exploiting the parallel processing capabilities offered by the Propeller chips, using the C language.

The Propeller chips are attractive because they offer a very low cost in terms of "bang for buck", and are appropriate for both hobbyists and professional uses.

Catalina has had multi-threading support built-in almost since its inception, but the "thread factory" (demonstrated here) is a recent innovation, and will be improved and added to in future releases of Catalina. It is partly the result of Chip Gracey (designer of the Propeller chips) asking whether it was possible for threads to execute *anywhere* (he meant on any cog) and not care where. He was, as usual, quite correct. Sometimes you can't see the wood for the trees!

High-level language support on the Propeller 1 has always been a tight fit, and language developers have had to go to extraordinary lengths to make them work. The result is that while all the programs discussed here will happily *execute* on the Propeller 1 (albeit with smaller sieve sizes), the *benefits* of multi-processing in a high-level language on that chip are somewhat muted because of the higher overheads incurred in supporting high-level languages on that chip *at all*. But we have demonstrated that even on the Propeller 1 they can still be worthwhile.

But on the Propeller 2, exploiting the parallelism it offers can really make a difference! The Propeller 2 not only has features that facilitate the implementation of high-level languages (such as hub execution mode, and fast hub reads and writes), it also has 16 times as much Hub RAM to use, and versions with even more RAM and/or with cogs are already on the drawing board.

This makes the Propeller a perfect platform for exploring and exploiting parallel programming - and Catalina will continue to refine its multi-processing support to take advantage of that.

Appendix

There is another version of the sieve program in the **`demos\sieve`** directory, in the file **`sieve_bitmapped.c`**. This version uses an array of bits instead of an array of bytes to hold the sieve array, and therefore can be compiled to process a much larger sieve – sieve sizes of well over 3,000,000 numbers are possible on the Propeller 2. However, because it uses bit manipulation rather than byte manipulations, it cannot be "parallelized" as easily – the worker thread operations would not be independent because one worker thread could overwrite the results of another worker thread should the numbers they were working on at the same time be in the same 32 bit long.