SICSA Summer School on Advances in Programming Languages
Swan Programming Lab

Hans Vandierendonck

August 19th

This document guides you through programming with Swan.

You should have been set up with a copy of Swan on your local directory. It is available from `https://github.com/hvdieren/swan`.

You will notice difference in notation between theory and practice. This is due to Swan being implemented as a library rather than through a compiler. A detailed overview of the Swan API is provided in appendix.

1 Compiling Swan Programs

Swan contains two major parts: the scheduling library (`swan/schedulers`) and a directory of benchmarks (`swan/benchmarks`). There are also some utils (`swan/utils`) and example programs used for testing (`swan/tests` and `swan/tutorial`). When providing pathnames, we will assumes that the top-level swan directory is called `swan`.

First, enable a recent gcc compiler (version 4.8.2):

```
$ scl enable devtoolset-2 bash
```

To compile, step into the top-level swan directory and follow these steps (these commands are assuming a bash shell):

```
$ ./configure --disable-fortify-source
$ make util-dir scheduler-dir
```

If you type simply '`make`' you will do a lot more compilation than necessary for this tutorial.

Now you should have the scheduler library `swan/schedulers/libschedulers.a` and also several library files under `swan/utils`.

Now go into the directory `swan/tutorial` and study the code in `fib.cc` and check what it does. Then, compile using

```
$ make fib
```

Execute `fib` using all available cores with argument 32:

```
$ time ./fib 32
```

Execute `fib` using 1 core:

```
$ time NUM_THREADS=1 ./fib 32
```

or:

```
$ export NUM_THREADS=1
$ time ./fib 32
```

What you should see is that the parallel execution is faster than the single-core execution, although it won't be as much faster as the number of cores in your computer. The Fibonacci number written out should be the same regardless of the number of threads used.

2 Pipeline Parallelism

1. In the directory `swan/tutorial`, review the source code in `pipeline.cc`.
2. Compile the program and execute it several times using 8 threads.

   ```
   $ NUM_THREADS=8 ./pipeline 10
   ```

Observe the differences in output. Does every execution produce an output that is correct given the dependences specified in the program?

3. Now change the program at lines 70 and 71 and change the declarations of `object_t<...>` to `unversioned<...>`. Also change the `call` at line 79 to a `spawn`. Recompile and reexecute the program. Explain the impact on program output.

3 Dataflow in Linear Algebra

The task dataflow model has received strong support because of its natural fit to linear algebra. Linear algebra operations, such a matrix multiply, transpose and matrix factorisations, are expressed recursively as primitive operations on blocks of the original matrices. In this instance, we will take a closer look at an implementation of LU factorization on sparse matrices.

1. Move to the directory `swan/benchmarks/sparse_lu/src_wfo` and inspect the file `sparse_lu.cc`. The main algorithm is in the function `LU`, which takes as arguments a 2-dimensional array of points to blocks of a matrix. NULL pointers represent missing (zero) blocks as the matrix is sparse.

2. Inspect the source code to understand the parallelism in this algorithm. Try to estimate how much parallelism is in the code by inspecting the loop bounds.

3. Compile the code (run `make`) and run it with several thread counts (from 1 to the number of cores in your machine). You may need to repeat the measurements a few times to obtain stable execution times.

4. Analyse the performance scalability of the code. How well does it scale? Did you expect this from analysing the code?

4 Hyperqueue

Now let's turn to an example using the hyperqueue.

1. Review the source code of `swan/tutorial/hyperqueue.cc`.

2. Compile the program and execute it. The program takes four parameters as follows: `./hyperqueue <n> <producers> <consumers> <delay>`, where

`<n>`	Number of items to send through queue
`<producers>`	Number of producer tasks
`<consumers>`	Number of consumer tasks
`<delay>`	Sleep time in microseconds applied for each item pushed or popped

   Execute the program as follows:

```
$ NUM_THREADS=1 ./hyperqueue 10 1 1 0
```

   Inspect the program output.

3. Now unset the number of threads (or set it to the maximum number of threads in your machine) and re-execute. Explain the observed behavior.

4. Increase the number of items in the queue (first positional argument) to 100 or 1000 and re-execute. Re-execute a few times. Explain the observed program outcome.

5. Now increase the number of producers to 10. Explain the impact on program output.

6. Now increase the number of consumers to 10. Explain the impact on program output.

5 Programming Exercises

5.1 An Irregular Pipeline

Pipelines are often irregular, which can pose difficulties and/or inefficiencies for programming models. In this exercise, you will create a pipeline parallel program where a pipeline stage is skipped from time to time.

1. Copy the file `swan/tutorial/pipeline.cc` to `swan/tutorial/irpipeline.cc` and add the target `irpipeline` to the variable `EXAMPLES` in `swan/tutorial/Makefile`.

2. Add a pipeline stage B2 to the program that takes a single argument with in/out side effect and multiplies that value by -1. Stage B2 is executed between stages B and C and is passed in the object `obj_bc`. However, B2 is executed only if the loop counter `i` is even.

   Edit the code as requested, compile and execute. Check that the output is correct.

5.2 Matrix Multiply

Study the matrix multiply code in `swan/tutorial/matmul.cc`. Again, data is laid out in blocked format. For your convenience, initialization and correctness checking code has been parallelized.

1. Parallelize the function `multiply_matrix` by adding task side effect annotations to the calls to the function `mult_add_block` and turning these calls into `spawn` statements. Don't forget to insert a `sync` statement and to call `multiply_matrix` using `run`.

   Compile the code and execute it on one thread and on multiple threads. Is the code correct? Does performance scale?

2. Parallelize again the function `multiply_matrix` without using side effect annotations but using only the Cilk-style `spawn` and `sync` statements.[1]

   Compile the code and execute it on one thread and on multiple threads. Is the code correct? How does performance compare against the task dataflow version?

[1] It is understood that proper Cilk methodology requires you to recursively decompose the iteration space into smaller pieces as this increases the efficiency of the scheduler. We will leave that as homework.

A Appendix: Swan API

Swan is currently implemented as a C++11 library. As such, the syntax used in practice deviates slightly from the desired syntax. The section shows the details.

A.1 Variable Annotation

Variables on which dependence tracking is to be performed must be annotated at declaration time. There are two annotations: `object_t` (the equivalent of the term `versioned` used in publications and presentations) and `unversioned`. The difference between the two is that renaming may be applied to a `object_t` but not to an `unversioned` variable.

The annotations are applied as C++ templates to the intended variable type:

```
template<typename T>
class object_t;

template<typename T>
class unversioned;
```

Example:

```
unversioned<int> i; // an integer variable with dataflow support
object_t<double> d; // a double variable with dataflow support
object_t<std::string> s; // a string object with dataflow support
object_t<float[]> array( 128 ); // an 128-element array of floats
```

In each case, dataflow dependences are tracked on the object as a whole. E.g., for the array of `float`, tasks are assumed to touch the whole array or none of it.

Hyperqueues are declared with type `hyperqueue`:

```
hyperqueue<int> queue; // a queue of integers
```

A.2 Argument Annotation Types

Variables of type `object_t`, `unversioned` and `hyperqueue` should never be passed to function arguments. The `spawn`, `call` and `leaf_call` functions explicitly forbid this. These variables should be cast to one of the argument annotation types (a.k.a. dependence types) listed below.

Applicable to `object_t<T>` and `unversioned<T>`	
`indep<T>`	The task will access argument in read-only manner
`outdep<T>`	The task will over-write every single element of the argument. After writing an element, it may also read that element.
`inoutdep<T>`	The task may read and write individual elements of the argument, or the argument as a whole.
`cinoutdep<T>`	The task may read and write the argument as with `inoutdep<T>`. Moreover, this task may be reordered relative to other tasks applied to the same argument with the same annotation, relative to dependence constraints introduced by other task arguments. This annotation guarantees mutually exclusive access to the argument.
`reduction<M>`	The task will apply a reduction operation that is both commutative and distributive. The type `<M>` is struct that defines a base type, an initialisation function and the binary reduction operation.
Applicable to `hyperqueue<T>`	
`pushdep`	The task may push on the hyperqueue.
`popdep`	The task may pop from the hyperqueue.

Note that all these variable and dependence types live in the namespace `obj`.

A.3 Statements

The key statements in Swan are `run`, `spawn`, `call`, `leaf_call` and `sync`.

A.4 run

`run` starts a parallel region. Due to absence of a compiler at the moment, it is not practical to implement parallel/serial reciprocity, i.e., the ability to arbitrarily nest parallel and serial code regions. `run` enters a parallel region while `leaf_call` leaves the parallel region. Upon exit form a parallel region using `leaf_call`, it is not possible to re-enter a parallel region in Swan.

A.4.1 Format

```
template<typename TR, typename... Tn>
TR run( TR (*func)( Tn... ), Tn... args );
```

A.5 spawn

A spawn statement indicates that the spawned function may execute in parallel with the calling function given that the semantics of dataflow dependences and `sync` statements are respected. Spawned functions can take any of the dependence types as argument.

Spawned functions with a non-void return value require that a memory location is identified where the return value of the function can be stored. The memory location must remain alive until the next `sync` statement is encountered. Such a memory location is declared using the `chandle` type.

Here is an example:

```
int fib( int n ) {
    if( n < 2 )
        return n;
    else {
        chandle<int> x;
        spawn( fib, x, n-1 );
        int y = call( fib, n-2 );
        ssync();
        return x + y;
    }
}
```

A.5.1 Format

```
template<typename TR, typename... Tn>
typename std::enable_if<std::is_void<TR>::value>::type
spawn( TR (*func)( Tn... ), chandle<TR> & ch, Tn... args );

template<typename TR, typename... Tn>
typename std::enable_if<std::is_void<TR>::value>::type
spawn( TR (*func)( Tn... ), Tn... args );

template<typename TR, typename... Tn>
typename std::enable_if<!std::is_void<TR>::value>::type
spawn( TR (*func)( Tn... ), chandle<TR> & ch, Tn... args );
```

A.6 call

`call` explicitly labels function calls. The requirement for this is an artefact of the implementation of Swan as a library. Swan internally allocates stack frames and links them in a cactus stack. Every segment of the cactus stack should be large enough to hold the stack data for a number of function calls. However, now and then it is necessary or useful to explicitly reserve more stack space, which can be done by annotating a normal function call with `call`.

Note that `call` statements accept only arguments with `outdep` and `pushdep` annotations. The reason hereto is that a call must always be able to execute immediately (called function is not suspended). The argument annotations that allow this are `outdep` (due to renaming) and `pushdep` (due to allocating a fresh queue segment). Using `call` statements with annotations is useful a.o. for writing pipeline parallel code.

A.6.1 Format

```
template<typename TR, typename... Tn>
TR call( TR (*func)( Tn... ), Tn... args );
```

A.7 leaf_call

`leaf_call` enters serial code execution. Again due to the internal stack management of the library implementation of Swan, it is prudent to annotate library functions such as `printf` with `leaf_call`.

A.7.1 Format

```
template<typename TR, typename... Tn>
TR leaf_call( TR (*func)( Tn... ), Tn... args );
```

A.8 sync

The *sync* statement is represented by the function `ssync`. The name is `ssync` rather `sync` to avoid clashes with the *sync* system function.

A.8.1 Format

```
void ssync();
```

A.9 Argument Passing

Yet another side effect of implementing Swan as a library are some restrictions on argument passing. Swan implements most of the x86-64 calling conventions within `run`, `call`, `spawn` and `leaf_call`. It accepts all scalar types as arguments, pointer types and the Swan-specific argument annotation types. It does not accept reference types and has limited support for class types.

For class types, only types without non-trivial copy constructors and without non-trivial destructors are supported, i.e., the copy constructor and the destructor must be the implicitly-declared default function and it should not have virtual functions or virtual base classes. Arguments of such class types are passed by invisible reference at the assembly level, which is not supported by Swan. More details on these class types are in the x86-64 ABI specification.

It is possible to pass arguments with class types with trivial copy constructors and trivial destructors by value in Swan. Hereto, it is necessary to declare the non-static data members of the class type. For example:

```
struct struct_type {
    int x, y;
    char * p, * q;
    unsigned char * u;
    static int s; // not passed by argument
};
namespace platform_x86_64 {
    template<size_t ireg, size_t freg, size_t loff>
    struct arg_passing<ireg, freg, loff, struct_type>
        : arg_passing_struct5<ireg, freg, loff, struct_type,
                              int, int, char *, char *, unsigned char *> {
        // empty
    };
}

void f( struct_type s );
struct_type s;
spawn( f, s ); // works
```

Note that declaring the data members is rarely required. In any case, it is possible to pass arguments of class type by pointer.