Wool 0.1.8 users guide

Karl-Filip Faxén

April 1, 2016

Abstract

This document gives a short reference to the Wool library. Wool is a library providing lightweight tasks on top of `pthreads`. Its performance is superior to that of Cilk and the Intel TBB, at least in terms of overhead.

1 Introduction

Wool grew out of an attempt to understand how to design a really low overhead user-level task scheduler. For convenience, Wool is a C-library rather than being implemented in a compiler code generator or as a preprocessor. However, Wool uses macros and inline functions to make the overhead involved in integrating the task operations into the operations of the program small.

A Wool program starts execution with the library initializing itself, in particular starting one OS thread for each physical processor. These threads are called workers. Each worker has a set of data structures for implementing task management, in particular a pool of tasks that are ready to execute. An important design decision in Wool is that this pool is a stack that grows and shrinks roughly in step with the main stack. This simplifies memory management enormously.

Parallelism is introduced by spawning tasks, which roughly corresponds to doing an asynchronous function call. Spawning a task is implemented by allocating a task descriptor on the top of the task stack and initializing it with some administrative information, including a pointer to the code of the task, and some arguments to the task. The application program using Wool is responsible for joining with every spawned task; if the task has not been stolen by another worker while in the task pool, it is then executed by the same worker that spawned it.

Parallel `for` loops (also known as `doall` loops) are also supported. Due to the lack of nested functions in C, named loop bodies are defined out of line and invoked by the `FOR` macro.

2 API Reference

This section gives a reference to the Wool API. There are constructs for defining, spawning and synchronizing with tasks. These constructs are all macros; the task definition macros have rather complicated definitions. The task definition macros are arity specific; the macros for defining tasks of arity one is different from those for defining tasks of arity two and so on. This is the reason why the Wool header file is distributed in the form of a shell script that takes an integer n and produces a header file with task definition macros up to arity n.

2.1 Creating parallelism with tasks and loops

This section describes the macros that are used to create and manage parallelism. Wool allows to use `SPAWN`, `CALL`, `SYNC`, or `FOR` in any C function, although such calls will be slower when located in an ordinary C function than when located in a task or loop body. This is because they need to recompute the task stack pointer and use operations of the underlying thread library (`pthreads`) to find the per worker data structures. There is no change of syntax; the same macros are used in C functions and in tasks/loop bodies.

2.1.1 Task definition

Tasks are introduced using task definitions of the form:

$$\texttt{TASK\_}n(\mathit{rtype},\ \mathit{name},\ \mathit{argtype}_1,\ \mathit{argname}_1,\ \ldots,\ \mathit{argtype}_n,\ \mathit{argname}_n)\{\ \mathit{body}\ \}$$

Here, n is the number of arguments to the task (its arity), *rtype* is the return type of the task (it returns objects of type *rtype*), $\mathit{argtype}_i$ is the type of the i'th argument and $\mathit{argname}_i$ is its name. Finally, *body* is the code that is executed when the task is invoked.

A task as defined above closely corresponds to a function with the following definition:

$$\mathit{rtype}\ \mathit{name}(\mathit{argtype}_1\ \mathit{argname}_1,\ \ldots,\ \mathit{argtype}_n\ \mathit{argname}_n)\{\ \mathit{body}\ \}$$

(In fact, a function with a very similar definition is part of the implementation of the task definition.) In particular, the same identifiers are visible in *body* and the same way should be used to return from it.

There is also a second form that is used for tasks that do not return a result:

$$\texttt{VOID\_TASK\_}n(\mathit{name},\ \mathit{argtype}_1,\ \mathit{argname}_1,\ \ldots,\ \mathit{argtype}_n,\ \mathit{argname}_n)\{\ \mathit{body}\ \}$$

The corresponding function definition is:

$$\texttt{void}\ \mathit{name}(\mathit{argtype}_1\ \mathit{argname}_1,\ \ldots,\ \mathit{argtype}_n\ \mathit{argname}_n)\{\ \mathit{body}\ \}$$

Both `TASK` and `VOID_TASK` are indexed families of macros. Valid indices are determined by the argument to `wool.sh` when generating `wool.h`.

2.1.2 Spawning tasks

The task *name* with arity n is spawned with arguments e_1 to e_n by:

`SPAWN`($name$, e_1, …, e_n)

This expression, which does not return a value, causes the new task to be placed in the task pool of the executing worker, so that it can be stolen by other workers.

2.1.3 Synchronizing with tasks

A previously spawned task called *name* can be synchronized with its parent as follows:

`SYNC`($name$)

This expression has the type of the task *name*, that is, if *name* was defined by `TASK_`n($rtype$, $name$, …){ *body* }, the type is *rtype* while if the task definition was of the form `VOID_TASK_`n($name$, …){ *body* }, the type is `void`.

A `SYNC` matches the last unsynced `SPAWN`, making it synced so that the previous unsynced spawn becomes the new last unsynced spawn. This behavior simplifies memory management by allowing the task pool to be maintained as a stack; a `SPAWN` pushes a task on the stack and a `SYNC` pops the top task off of the stack and synchronizes with it.

Synchronization can entail one of three different actions on the part of the calling task:

- If the task was not stolen, it is invoked directly by a function call to the task's work function. This is by far the most common case, and the inclusion of the task name in the sync syntax allows the call to be an ordinary direct C function call that can even be inlined.
- If the task was stolen and the thief has completed executing the task, its result value (if any) is extracted from the task and returned.
- If the task was stolen and has not completed, the calling task becomes blocked. The default behavior in the current version of Wool is that the worker executing the blocked task attempts to steal a task from the thief, a strategy called *leap frogging.*

2.1.4 Invoking tasks directly

As an optimization, Wool provides a direct invocation macro for tasks. Thus the expression

`CALL`($name$, e_1, …, e_n)

is equivalent to

`SPAWN`($name$, e_1, …, e_n), `SYNC`($name$)

except more efficient (and briefer). A direct task invocation becomes a simple direct function call with the required hidden argument added.

2.1.5 Loop bodies

A loop body is defined using the `LOOP_BODY` family of macros, as follows:

`LOOP_BODY_`n($name$, $grainsize$, $ixvartype$, $ixvar$, $argtype_1$, $argname_1$, ..., $argtype_n$, $argname_n$){ *body* }

Here, *name* is the name of the loop body (used when invoking the loop), *grainsize* is a lower bound on the number of cycles the loop body takes to execute (used by Wool to balance parallelism versus overhead), *ixvar* is the index variable of the loop and *ixvartype* is its type (typically an integer type like `int`, `long`, `unsigned long long` or similar). Finally, the rest are loop invariant parameters that will have the same values in all iterations, given when invoking the loop with `FOR`.

The loop body *body* performs *one* iteration of the loop and should be written as if it were the body of a C function (which we refer to as the *loop body function*) declared as:

`void` $name$($ixvartype$ $ixvar$, $argtype_1$ $argname_1$, ..., $argtype_n$ $argname_n$)

It is ok to have a very light weight loop body; Wool will implement the parallel loop using a tree of divide and conquer tasks down to a certain level, where a loop will be executed calling the loop body function. Since the call is direct, and the loop body function is marked for inlining, the effect is that the body will be inlined into the loop. The number of iterations that are executed sequentially in this way depends on the *grainsize* value; the cutoff is tuned so that if the loop body really takes *grainsize* cycles to execute, about 1% of execution time should be spent in spawning and syncing with tasks in the parallel divide and conquer tree. That is, if the overhead on a particular machine is S cycles for a spawn/sync pair, then

- if *grainsize* is greater than about $100 \times S$, the sequential loop will iterate only once, and
- otherwise the cutoff will be $100 \times S/grainsize$ iterations.

Lying about *grainsize* gives programmers control over the trade off between parallelism and overhead; an underestimate will give lower overhead (more sequential iterations) whereas an overestimate will result in fewer iterations thus exposing more fine grained parallelism and potentially giving better load balancing. There should however be little need for the latter, since the size of the generated tasks at the leaves of the parallel tree is about $100 \times S$, which for a typical value of S of 20 cycles becomes 2000 cycles. This is very small compared to useful task sizes (typically on the order of 100k cycles), and almost all stealing will happen closer to the root with bigger tasks. Thus an underestimate of *grainsize* is unlikely to yield problems with loss of parallelism in practice.

There is a symbolic constant `LARGE_GRAIN` that is equal to $100 \times S$. Using this value for *grainsize* ensures that each leaf in the parallel divide and conquer tree only executes one loop body.

2.1.6 Invoking loops

A parallel loop is invoked with iteration bounds e_{low} and e_{high} and loop invariant arguments e_1 to e_n as follows:

$$\texttt{FOR}(name,\ e_{low},\ e_{high}, e_1,\ \ldots,\ e_n)$$

This will cause the loop body function to be invoked $e_{high} - e_{low}$ times as in a C for loop of the form:

```
for( i = e_low; i < e_high; i++ ) { ... }
```

The iterations are executed logically in parallel but possibly sequentially to limit overhead. Note that the return type is `void`; a loop does not return anything.

2.1.7 Support for separate compilation

In previous versions of Wool, a task had to be defined in the same file that it was used since there was no way of declaring the identifiers that a task definition creates. There was, so to speak, nothing to put in the header file. We have now split the functionality of the `(VOID_)TASK_`n macros into separate declaration and implementaion macros which take the form

$$\begin{array}{l}\texttt{TASK\_DECL}(RT, F, T_1, \ldots, T_n)\\ \texttt{VOID\_TASK\_DECL}(F, T_1, \ldots, T_n)\\ \texttt{TASK\_IMPL}(RT, F, T_1, V_1, \ldots, T_n, V_n)\\ \texttt{VOID\_TASK\_IMPL}(F, T_1, V_1, \ldots, T_n, V_n)\end{array}$$

where RT is the return type, F is the task name, the T_i are the argument types, and the V_i are the argument names. The declaration macro goes into the header file and the implementation macro into the C file. A task declaration can also be used in a single file program as a forward declaration allowing mutually recursive tasks. For convenience and compatibility, the old `(VOID_)TASK_`n macros are still provided, each defined as a pair of a corresponding declaration and implementation.

The non inlined functions defined by these macros are global, not static, so you should avoid identically named tasks in the same program.

2.2 Main program

There are two possible program structures.

2.2.1 Main as task (a.k.a old style programs)

In this style, the Wool library provides the `main` function (in the source file `wool-main.c`), so it gets control at program start up. After initialization it invokes the task called `main`, which the program should define as a task with two arguments, an `int` and a `char**`, and returning an `int`, thus:

```
TASK_2(int, main, int, argc, char**, argv)
{
  ...
}
```

The library decodes a few flags controlling things like the number of workers and a few other parameters, but it passes the rest of the arguments to the main task, so a Wool program can also have command line arguments.

2.2.2 Explicit initialization (a.k.a. new style programs)

In this style, `main()` is an ordinary C function which explicitly starts and stops the Wool run-time system using the following functions:

- `int wool_init(int argc, char **argv)` which initializes the run time system using the supplied command line options and returns the number of remaining arguments after eating the Wool options. Remaining options are shifted to the front, so after the call, `argv` can be used as if the Wool options had not been there and the return value gives the number of remaining arguments.

- `void wool_fini(void)` which stops the run time system, in particular it terminates the workers and prints any statistics gathered to `stderr`.

Sometimes it is inconvenient to decode options and then immediately start Wool. For instance if the application also is interested in the command line arguments but has a lengthy initialization phase which should be done before the run-time system starts, so that it is not covered by the timing information collected by the run-time system. For that reason the initialization of Wool can be split in two using the following functions (in the order given here):

- `int wool_init_options(int argc, char **argv)` which just decodes the supplied command line options and returns the number of remaining arguments after eating the Wool options. Remaining options are shifted to the front, so after the call, `argv` can be used as if the Wool options had not been there and the return value gives the number of remaining arguments.

- `void wool_init_start(void)` which starts the run-time system (in particular the worker threads).

2.3 Additional library functions

There are a few additional library functions that can be used, once the library is initialized, to find out information about the way the program runs.

- `int wool_get_nworkers(void)` which returns the number of workers, including the root worker (the main thread of the program).

- `int wool_get_worker_id(void)` which returns the worker id of the worker running the current task, in the interval 0 to $N-1$ if there are N workers. Note that the library must have been started (`wool_init()` or `wool_init_start()`) for it to be legal to call this function. Note also that since Wool never moves a running task from one worker to another, the return value should be stable; calling the function repeatedly from the same task should yield the same value.

3 Running Wool Programs

3.1 Command line options

The Wool library functions `wool_init()` and `wool_init_options()` decode some flags which controls the operation of Wool. Wool programs that pass the command line arguments to any of these functions (in particular old style Wool programs do this) implement the following flags, among others:

-p <n> Number of workers started. If this option is not given, it defaults to one worker per processor in the affinity set of the main thread.

Decoding of options stops when an unknown option is found, and the rest of the arguments (starting from the offending option) are passed to the `main` task of the program. Since there are several more flags implemented, it is best to give an argument-decoding Wool program a double dash (`--`) before any options decoded by the program itself; Wool interprets a double dash as end of its options.

3.2 Controlling the number of workers and their affinities

By default, Wool runs one worker on each processor in the affinity set of the main thread. Each worker has its affinity set to a single processor. Given that affinities are advisory rather than mandatory, it is not certain that it will run on that processor, but it is very likely. The main thread always runs worker 0, which gets the first processor in the set, with subsequent workers getting subsequent processors, in order. The Linux command `taskset` can be used to control the affinity set of the main thread of the program.

4 Building the library

Currently, the library is built as an object file that is linked to the program (that is at least what the make file does). Since the code is rather small and performance sensitive, the present strategy seems reasonable.

There are a number of build time options that are important. These affect both the header file `wool.h` and the implementation of the library in `wool.c`.

MAX_ARITY The header file `wool.h` is generated by a shell script called `wool.sh` which takes an argument n and generates task definition macros for arity 1 to n and loop body macros for arity 0 to $n-2$ (the loop body definition macro for arity i uses the task definition macro for arity $i+2$). The default is 10.

TASK_PAYLOAD All task descriptors are the same size in the current implementation, simplifying (and thus speeding up!) the management of the task pool. This parameter controls the size in bytes needed to store the arguments of the largest task in the program. The default is **MAX_ARITY** $\times$ 8, allowing each argument in the maximum arity task to be a `double`. The argument area is guaranteed to be aligned on an 8 byte boundary, which is typically the strictest alignment requirement for conventional data types in current processor implementations. If you use larger arguments than 8 bytes (for instance structs), you may need to use this option, as well as if you only use smaller arguments and wish to save memory.

FINEST_GRAIN This controls the number of iterations executed sequentially at the leaves of a divide and conquer tree implementing a parallel loop. Wool will use the *grainsize* value given in the loop body definition to ensure that computations cheaper than **FINEST_GRAIN** are executed sequentially.

COUNT_EVENTS Setting this parameter to 1 enables code which counts various events (spawns, steals, ...) and prints statistics to standard error, while setting it to 0 builds with this code disabled, which is also the default.

5 Performance analysis

In this section we give some brief ideas about how to figure out whether a Wool program runs efficiently and how to improve it if not.

5.1 Critical path profiling

The *critical path* of a computation is the longest sequence of data dependent operations of the computation; the *critical path length* or *span* of a computation is the time to execute the critical path, which is also the time to execute the program on an unlimited number of processors (ignoring overhead). The average parallelism is given by the dividing the execution time on one processor by the span.

For a work stealing scheduler like Wool, the span gives an upper bound on the execution time of a computation on a given number of processors:

$$T_p < T_\infty + \frac{T_1}{p}$$

Here, T_p is the execution time on p processors, T_∞ is the span and p is the number of processors. The two terms on the right hand side are, roughly, the execution time if we ignore resource constraints and the execution tme if we ignore dependencies. Both are lower bounds on the execution time so we have

$$T_p > \max(T_\infty, \frac{T_1}{p})$$

as a combined lower bound.

For computations with frequent steals, the overhead of stealing must be taken into account when computing span. We do this during the span measurement by, at each join point, simulating the optimal scheduling desicion given a certain overhead. The span S of the fork-join region is computed as

$$S = \min(\max(S_1, S_2) + T_S, S_1 + S_2)$$

where S_1 and S_2 are the span of the two parallel subcomputations and T_S is the stealing overhead. Basically, if the smaller subcomputation is cheaper than the stealing overhead, we compute the span as if the subcomputations were run sequentially.

Reasonable values for the stealing overhead is 1000-2000 cycles for our 2.3GHz dual quadcore Opteron and 500-1000 cycles for our 700 MHz Tilera machine.

Wool can be built with instrumentation for finding the span by defining the preprocessor symbol `WOOL_MEASURE_SPAN` to the value 1, either using `-D` at the compiler command line or by saying `make WOOL_MEASURE_SPAN=1`. The program should then be run on a single processor and the total execution time, span and an estimate of speedup on different numbers of processors are output on `stderr`. The desired stealing overhead can be set at run time using the `-c` option to the Wool program. The unit is whatever the time base we use measures; for x86s we use the time stamp counter which gives processor cycles and for the Tilera the unit is likewise processor cycles. The pie time profile prints the frequency of the measurement clock.

5.2 Pie time profiling

Wool can be built to collect information about the time spent on different kinds of work, which we call *pie time profiling* since one might present the infomation as a pie chart. This time is collected per worker and summed by category, so it is CPU time rather than elapsed time. Therefore, the sum of the categories should be close to the sum of user and system time as reported by for instance the Linux `time` command. The categories are:

Startup Time spent on startup, for instance creating workers, up until the first successful steal (or exit, for workers with no successful steals).

Work Time spent executing the application, including the part of Wool related to inlined (never stolen) tasks.

Overhead Time spent executing successful steal operations.

Search Time spent looking for work, that is, executing failed steal operations.

Exit Time spent in final synchronization, from the last exit from application code until the worker itself exits.

Further, the **work**, **overhead** and **search** categories are divided into steal and leap subcategories. The steal subcategory counts the time spent when the steal routine is called from the main loop of the worker wheras the leap subcategory is associated with leapfrogging steals, that is, when steal is called from the join operation. Steal work is the application code invoked by successful ordinary steals wheras leap work is application code invoked by successful leapfrogging steals.

In addition to aggregate times, time per operation is also reported for the following operations (also distinguished as leap frogging related or ordinary):

Succesful steal The part of a succesful steal until the invocation of application code.

Failed steal The complete time of a failed steal.

Signal The part of a successful steal after the application code returns.

Pie time profiling is enabled by defining the preprocessor symbol `WOOL_PIE_TIMES` to the value 1 either using `-D` at the compiler command line or by saying `make WOOL_PIE_TIMES=1`.

5.3 Performance debugging of Wool code

Here are some hints about assessing the performance of a Wool program. An important starting point is to always choose an input to the program that makes it run at least on the order of a second or so. This might be tricky when scaling to many workers, as you want to compare the same workload when examining scalability. Sometimes I use different input sizes for overlapping ranges of processor counts.

5.3.1 Checking spawn/sync overhead

Typically, you develop a Wool program starting from a sequential one. By comparing the execution time of the sequential version to the execution time of the Wool version on a single processor, you get an idea of the impact of inlined (not stolen) tasks. These times should be very close unless your program is extremely fine grained, like the `fib` example program in the Wool distribution where each task performs on the order of ten cycles of work.

5.3.2 Measuring speedup

Compare execution time of the Wool program on different numbers of processors. If the execution time on p processors is close[1] to $\frac{1}{p}$ times the execution time on one processor, you're fine. Otherwise read on.

5.3.3 Do the Wool workers run all the time?

Wool is quite aggressive in using the CPU, effectively busy waiting while looking for work. Hence the sum of user and system time (as reported by `time`) should be close to the real time times the number of workers. If it is more, a miracle has occurred. If it is less, maybe the machine was not as quiet as intended.

5.3.4 Do I have too little parallelism?

Do critical path profiling, as described above. If the results are in line with the observed (bad) speedup you need to consider the algorithms you use. Are they as parallel as you think? Also check your initialization code; is it sequential? Maybe you can parallelize it?

If the program is nondeterministic in the sense that the shape of the dependence graph depends on timing details, the span measurement can be misleading. Consider a program with a shared memo table. The first time a function is called with a certain argument, the result is recorded in the memo table so that subsequent invocations with that argument can reuse the previously computed result. In a single processor execution, the first call with a given argument occurs in the same place in the call tree every time, but in a parallel execution the call (task) tree is traversed in different orders depending on nondeterministic timing details. Hence different invocations with the same argument may be the first one, arbitrarily affecting the value of the span computation.

5.3.5 Where is my program spending its time?

Time to do pie chart profiling. If your program scales well, the CPU time reported when measuring pie times should be similar when increasing the number of processors, but if it is not, check the categories that increase:

Work This means that the application code runs slower on larger number of processors. There are three broad classes of causes I have run across here:

- The program overtaxes shared resources. If your processors are not cores but hardware contexts which share execution units, as on hyperthreaded Intel CPUs, this is normal and probably a sign that your code runs well without the hyperthreading. But it could also be due to for instance limited memory bandwidth (a memcopy benchmark will not scale well) or shared cache size.

[1] Close means *close*, not like within a factor of 2. Wool is supposed to be fast!

- Communication effects; when a worker executes stolen code, the data is not in the cache of the thief but must be read from the original processor. The Cholesky example program behaves like this for small matrix sizes.
- Extra synchronization in the application. If you for instance have a shared data structure protected by a lock, the time spent waiting ends up here. If you have lock free operations, retry time does likewise.

Steal search This is a typical sign of insufficient parallelism, so you should have been alerted by the critical path profiling. So if that one looks good, but you still get dramatic increses in steal search time, this might be a problem with the Wool scheduler.

Leap search If your critical path profile looks good, but you get swamped by leap search time, it means that the transitive leap frogging algorithm used by Wool is not strong enough to handle the joins in the program. This is possible since the Wool scheduler is not completely greedy; when blocked at a join (sync), transitive leap frogging only allows you to steal a subcomputation of the one for which you're waiting. This is uncommon (I've never seen such programs in the wild) but possible. Try to divide large tasks into smaller; maybe you wanted large tasks to avoid overhead?

Overhead I've never seen a program where overhead is significant *and* not much smaller than search, so do drop me an email!