Wool 0.1 users guide

Karl-Filip Faxén

June 1, 2009

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 competitive with 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 (right now, the number of threads is a command line option decoded by the library). 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 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 Task definition

Tasks are introduced using task definitions of the form:

$$\texttt{TASK\_}n(rtype,\, name,\, argtype_1,\, argname_1,\, \ldots,\, argtype_n,\, argname_n)\{\, body \,\}$$

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*), $argtype_i$ is the type of the i'th argument and $argname_i$ 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:

$$rtype\; name(argtype_1\; argname_1,\, \ldots,\, argtype_n\; argname_n)\{\, body \,\}$$

(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:

$$\texttt{VOID\_TASK\_}n(name,\, argtype_1,\, argname_1,\, \ldots,\, argtype_n,\, argname_n)\{\, body \,\}$$

The corresponding function definition is:

$$\texttt{void}\; name(argtype_1\; argname_1,\, \ldots,\, argtype_n\; argname_n)\{\, body \,\}$$

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

2.2 Spawning tasks

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

$$\texttt{SPAWN}(name,\, e_1,\, \ldots,\, e_n)$$

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.

2.3 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.*

2.4 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.

2.5 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` $name(ixvartype\, ixvar,\, argtype_1\, argname_1,\, \ldots,\, argtype_n\, argname_n)$

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 there each leaf in the parallel divide and conquer tree only executes one loop body.

2.6 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:

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

This will cause the loop body function to be invoked $e_{high} - e_{low} + 1$ times, logically in parallel but possibly sequentially to limit overhead. Note that the return type is `void`; a loop does not return anything.

2.7 Main program

The Wool library contains the `main` function, 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.

3 Running Wool Programs

The Wool library decodes the following flags controlling its operation:

-p <n> Number of workers started. If this option is not given, it defaults to a single worker.

-s <n> Number of stealable tasks in the task pool; only the oldest **n** tasks become stealable, decreasing overhead in recursive divide-and-conquer applications while potentially leading to loss of parallelism. If this option is not given, the default is $3 + \log_2 N$ where N is the number of workers. If the work is well balanced, this gives about eight stealable tasks for each worker.

-t <n> The (initial) size of the task pool of each worker. Defaults to 1000 tasks per pool.

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.

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). There is because the library itself defines the `main` function of the program; in addition, as 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 as a loop 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.