A Programming Tutorial for the Wisconsin Wind Tunnel

(Or How to Survive Your First Few Days on the Wisconsin Wind Tunnel)

Shubhendu S. Mukherjee, Alain Kägi, and Douglas Burger

wwt@cs.wisc.edu

January 10, 1995

1 Introduction

This tutorial gives a brief introduction to programming, compiling, and executing parallel shared-memory applications on the Wisconsin Wind Tunnel (WWT), a *virtual prototyping system*[1] [7]. The WWT currently runs only on a Thinking Machines CM-5, so we assume that the reader has access to one and knows how to log in and run programs and is familiar with basic Unix(TM) functionality.

The tutorial illustrates how to parallelize a simple sequential application; how to use the Cooperative Shared Memory (CSM) model [4] and different cache coherence protocols; how to execute, debug and profile parallel applications on the WWT; and how to use different network simulators. The tutorial should give you enough information to get started writing your own programs for the WWT.

2 Getting Ready

Before proceeding further, you should find the following directories that accompany the tutorial package:

`document/ templates/ sequential/ parallel/ parallel+cico/ bug/`

This document should be in `document/`. `templates/` contains template files for compiling and running programs, `sequential/` has a simple sequential matrix multiplication code,

[1] A virtual prototype is one that exploits similarities between the machine under design (the target) and an existing platform on which it is running (the host). It only needs to simulate those parts of the target that are not supported by the host. As a result, a virtual prototype is *faster* than a simulator, but *slower* than the host on which it runs.

`parallel/` contains the parallel version of the code in `sequential/`, `parallel+cico/` has the code in `parallel/` with CSM annotations and `bug/` has an example showing how to track memory faults.

Make your own copy of the directories listed above. Make them writeable using `chmod -R u+w *` in the directory to which you copied all these files.

3 A sequential program

You will find a simple program in the `sequential/` directory that implements multiplication of two square matrices. You should examine, compile and execute it. A Makefile is provided for this purpose.

This program forms the basis for the parallel version presented below.

4 A parallel program

Currently the WWT supports only the *Single Program Multiple Data (SPMD)* paradigm. Loosely speaking, in this paradigm all tasks execute the same code on different portions of the data. At the beginning of execution only one task is active on one node (typically node zero) of the target machine. This task is responsible for forking tasks on other nodes. Only one task can be forked on a target node. The forked tasks get private copies of all the global variables (along with the corresponding values just before the fork) created in the first active task.

Since no parallelizing compiler is available, you will have to parallelize your code by hand. The `Parmacs`[2] directives allow you to do this. The `Parmacs` directives are a set of macros that provides constructs for writing parallel programs.

The remainder of this section introduces some of the `Parmacs` directives supported by the WWT, presents a parallel program containing some of them, explains how to compile the program, shows how to run the resulting executable code, and briefly describes the output from a WWT run.

4.1 Parmacs directives supported by the WWT

Below we describe the `Parmacs` directives supported by the WWT. Note that the syntax of some of these directives are slightly different from the standard `Parmacs` specification.

4.1.1 `ENV: declaration`

`ENV` declares various data structures required by the WWT to support the `Parmacs` model. These data structures *must* be visible to all the non-header files. For example, at the beginning of the `parallel/mm.U` file you will find the following statements:

```
/* mm.U - matrix multiply */
/* Setup environment for WWT */
ENV
#include <stdio.h>      /* for stderr */
...
```

4.1.2 `INITENV(): statement`

`INITENV()` initializes the environment and must execute before any other non-declarative `Parmacs` directives. Look in Section 4.1.5 for an example of its usage.

4.1.3 `G_MALLOC(sh_mem): function`

Unless otherwise specified, all data are allocated in memory private to the allocating task created by `CREATE_ALL` (explained later), and inaccessible to all other tasks. If some data needs to be shared, the only option is to allocate shared memory using the directive `G_MALLOC()`. `G_MALLOC()` has the same interface as the standard Unix library call `malloc()`. For example, you can malloc shared space for an integer variable `shared_var` as

```
shared_var = (int *)G_MALLOC(sizeof(int))
```

`G_MALLOC` guarantees that the address it delivers is aligned on a cache block boundary. There is *no* `G_FREE` to free up shared space.

4.1.4 `CREATE_ALL(func) and WAIT_FOR_END(): statements`

In the `Parmacs` world, only one task is active (on processor zero) when the parallel program starts. We will call this task the *master*. The directive `CREATE_ALL()`[2] activates the other tasks (one on each of the other processors), which we call the *slaves*. This directive is passed the name of a function, *func*, that will be executed in parallel by the *slaves*. The call to `CREATE_ALL()` returns immediately.

Often a program requires a post-processing phase, which cannot operate in parallel and cannot proceed until all parallel tasks complete. The directive `WAIT_FOR_END()` synchronizes all *slaves* and deactivates them as soon as they return from *func*. When `WAIT_FOR_END()` returns, only the *master* is active. This directive should only be invoked by the *master*.

Here is an example from `parallel/mm.U`, where the *master* creates a process to execute the function `Multiply` on each of the other processors. `CREATE_ALL(Multiply)` returns immediately and the *master* does its share of the work by calling `Multiply`. At the end, the *master*

[2]The `Parmacs` model supports a different macro, CREATE, that creates one process on one processor. Hence, multiple calls to CREATE are necessary to fork multiple tasks, one on each processor. Although this macro is supported on WWT, you should use CREATE_ALL instead of CREATE because forking one task at a time is inefficient.

synchronizes the *slaves* by calling WAIT_FOR_END and then prints the output.

```
CREATE_ALL(Multiply);
Multiply();
WAIT_FOR_END();
printf(...);
...
```

4.1.5 XX_NUM_NODES & XX_NODE_NUMBER: constants

The constant XX_NUM_NODES denotes the number of processors available to the parallel program. The constant XX_NODE_NUMBER denotes the processor id on which a process is running, where $0 \leq$ XX_NODE_NUMBER $<$ XX_NUM_NODES. A process created by the macro CREATE_ALL can read the processor id from XX_NODE_NUMBER and decide to do its own share of work. For example, parallel update of a linear array A of size XX_NUM_NODES can be done by the following code:

```
ENV
int *A;
main()
{
  INITENV();
  A = (int *)G_MALLOC(sizeof(int) * XX_NUM_NODES);
  ...
  CREATE_ALL(Update);
  Update();
  ...
  WAIT_FOR_END();
  ...
}

Update()
{
  A[XX_NODE_NUMBER] = 0;
  ...
}
```

4.1.6 Locks

- LOCKDEC(*lock*) : declaration
- LOCKINIT(*lock*) : statement
- LOCK(*lock*) : statement
- UNLOCK(*lock*) : statement

The WWT implements *MCS locks* in software [6]. Declare a shared lock variable *lock*, using `LOCKDEC`(*lock*) . Initialize it using `LOCKINIT`((*lock*)). Use the directives `LOCK`(*lock*) and `UNLOCK`(*lock*) to lock and unlock, respectively, the variable *lock*. For example, you can atomically increment a shared variable, `I`, in the following way:

```
ENV
struct GlobalSpace
{
   int I;
   LOCKDEC(lock);
} *g;

main()
{
  INITENV();
  g = (struct GlobalSpace *)G_MALLOC(sizeof(struct GlobalSpace));
  g->I = 0;
  LOCKINIT(g->lock);
  ...
  LOCK(g->lock);
  g->I++;
  UNLOCK(g->lock);
  ...
}
```

4.1.7 `BARRIER(dummy, num_procs): statement`

The directive BARRIER(*dummy*, *num_procs*) sets up a barrier, which holds back processors until *num_procs* - 1 other processors reach any BARRIER directive. There is, however, a caveat here. *num_procs* must be equal to `XX_NUM_NODES`. Or, in other words, you cannot have a partial barrier. *dummy* is a dummy argument for our purpose. You *do not* have to declare it as any variable. The original `Parmacs` specification requires the name of a barrier variable in its place. For example, you can synchronize all processors at a point in the following way:

```
ENV
...
main()
{
  INITENV();
  CREATE_ALL(test_function);
  test_function();
  WAIT_FOR_END();
  ...
}

test_function()
{
```

```
    ...
    BARRIER(dummy, XX_NUM_NODES);
    ...
}
```

4.2 Timing

- CLOCK(*vtime*): statement

A call to the macro CLOCK(*vtime*)[3] returns the *virtual time*[4] in thousands of cycles in *vtime*. This macro can be used to find the virtual time at any point during execution. For example, you can measure the virtual time for the parallel section of your code in the following way:

```
ENV
unsigned start_time, end_time, total_time;
...
main()
{
  INITENV();
  ...
  CLOCK(start_time);
  CREATE_ALL(...);
  ...
  WAIT_FOR_END();
  CLOCK(end_time);
  /* total_time = total virtual time to execute the parallel section */
  total_time = (end_time - start_time) * 1000;
}
```

4.3 Discussion of the example code

In `parallel/`, you will find a parallel version of the sequential matrix multiply program presented earlier. You should examine it, but postpone compiling and executing until you have read the next two subsections.

There are many ways to parallelize matrix multiplication. We have chosen to decompose the problem into computations of complete rows of the result matrix. Most of the code is straightforward. However, the work scheduler requires some explanation. The scheduler allows dynamic load balancing by distributing work where there is demand. Tasks compute a row of the result matrix at a time, picking a row in order, starting at the row with the smallest index. A shared variable, `I`, records the lowest row number that still requires computation. An idle task reads `I`'s value and increments it. A lock ensures correctness by guaranteeing that these operations occur atomically.

[3] Statement.

[4] The virtual time is the number of simulated target machine cycles since the beginning of execution. It is unaffected by how fast the WWT is running on its host, the CM-5.

Option	Description
-a	Rewritten a.out file name (default: a.out.vt)
-c#	All instructions have cost # (default: Cypress "601 #'s")
-d	Count down for WWT (default: up)
-i	Print information about program operation
-T	Directory for temporary files
-w	Executable runs on WWT

Table 1: Command line options for the virtual timer `vt`

4.4 Compilation

Complications arise during compilation because a parallel program (a) contains `Parmacs` directives and (b) runs on the WWT, a *virtual prototype*, and *not* on an actual machine. Hence, the compilation process has a few more steps than usual. You should now examine the file `templates/Makefile-skeleton`. You should use this file as a template for compiling your code on the WWT. We assume you are familiar with the *SunOS make*. We are going to describe only what is specific to the WWT.

We have filled in the blanks of the skeleton makefile for the parallel program in `parallel/`. Before building the executable, you should make sure that the directories `${WWT_ROOT}/bin` and `${WWT_ROOT}/scripts` are on your search path.

You need to create your program as a set of `.U` (similar to `.c`) and `.H` (similar to .h) files in the C language. The following actions will be taken by the `Makefile`. These files `.U` and `.H` files will be preprocessed by the `m4` macro processor and converted to the corresponding `.c` and `.h` files. A C compiler will convert these into object files, which will be linked by the linker `ld`. Finally, `vt` will instrument the target binary to keep track of virtual time on the target machine. The `vt` options are listed in Table 1. `make -f Makefile` produces an executable ready to run on the WWT.

You need to set an environment variable, `WWT_ROOT`, to the root of the WWT hierarchy. The root directory should contain the `include/` and `lib/` directories. In the `Makefile` you need to set the variables `PROTOCOL` to the cache coherence protocol (see Section 6), `MODEL` to the model you are using for writing shared-memory applications, which in our case is `Parmacs`, `TARGET` to the target binary you want to create and `OBJS` to the object files. `OBJS` should also contain another object file, `/lib/crt0.o`, which provides a pointer to the environment needed by some library objects. You can set additional flags through the variables, `OTHER_CFLAGS` (optimization level, the Wind Tunnel flag `WWT`, the cache coherence protocol etc.), `OTHER_LDFLAGS` (flags to `ld`) and `OTHER_VTFLAGS` (flags to the virtual timer `vt`. See Table 1). Set the `OTHER_VTFLAGS`, for example, to `-c1`. The resulting `VT` in the statistics file (see Section 4.6), `WWT.stat.`N, will denote the total number of cycles executed by your program, instead of the total virtual time taken by the program to execute on the target machine. You do not need to change any other variable in `Makefile`.

You should now compile the files in `parallel/` by typing in

```
make -f Makefile
```

4.5 Execution

Several ways exist to run the WWT with your target program. The first method is fine for simple runs. You might also want to use this while you are developing and debugging code. The second method, which uses the `benchmarks tree`, is more extensive, systematic and powerful. There are a collection of scripts in `${WWT_ROOT}/Scripts` to facilitate the manipulation of the benchmarks tree. This method should be used for extensive and systematic experimentation with different benchmarks and cache coherence protocols once you are sure your code is debugged and stable.

The WWT comes with this distribution as two binaries, namely, `dir1sw` and `dirix`, corresponding to the cache coherence protocols, Dir_1SW and Dir_iX respectively (Section 6). To run the target code on WWT with Dir_1SW, you should make a call like:

```
dir1sw <dir1sw switches> <target> <target arguments>
```

To use Dir_iX, replace `dir1sw` by `dirix`. The `dir1sw` and `dirix switches` are explained in Section 9.

However, you might not be able to run your code by simply typing in the above command in the command-line. The way you submit runs changes, depending on whether or not you have the Distributed Job Manager (DJM) running on the CM-5 at your site.

Section 4.5.1 explains how to submit jobs to the DJM. Section 4.5.2 explains how to use the same scripts as in Section 4.5.1 in the absence of the DJM. Section 4.5.3 describes how to use the `benchmarks tree` structure for doing controlled experiments.

4.5.1 Simple runs using the Distributed Job Manager

The standard way is to submit your job on a CM-5 to the Distributed Job Manager (DJM) with a file describing what and how things should be done. The file `submit` in `parallel/` is an example. `submit` contains DJM directives (commented expressions, like #JSUB ...) and a call to one of the WWT binaries, `dir1sw` or `dirix`, as shown before. The file `submit-skeleton` can be used as a template for submitting runs to the DJM. Most of the comments from the `submit` script in `parallel/` have been removed.

To submit the job type

`jsub submit`[5]

The reply should read

`Job submitted successfully.  Job id is` M.

where M is some number.

Upon successful execution of your program you will receive a notification through mail, and you will find the following files in the directory from which you submitted the job: `submit.o`M and `WWT.stat.`N, where N is some number not necessarily equal to M. The first file contains a summary of the execution session and the second file contains some WWT statistics about the execution of your program. The contents of the `WWT.stat.`N file will be explained in Section 4.6.

You should now examine the `submit` file in `parallel/` and try to run the code in the directory, if DJM is present at your site.

4.5.2 Simple runs without the Distributed Job Manager

To submit a job in the absence of the DJM, type

`/bin/csh submit`

At the end of the execution only the `WWT.stat.`N file will be created.

You should now try the above in the `parallel/` directory, if DJM is not installed at your site.

4.5.3 Benchmarks tree

The `benchmarks tree` is set up for large-scale experimentation and *cannot* be used without the DJM. Figure 1 illustrates the structure. The default root directory is derived from the environment variable `${WWT_ROOT}`. However, as will be explained later, you can change this root. `benchmarks.src` contains the source code and input files for the benchmarks. The `Protocol1..n/` directories contain the Makefile necessary to compile the target code with the particular protocol. `benchmarks/` contains links to the `inputs/` and `src/` directories in `benchmarks.src`. The `runs/` directory under `benchmarks/` has the runs for the different experiments, named `Expt1..n/`. Each such experiment directory must have two basic files - `Makefile` and `experiment_settings`, and links to any other input files required. The contents of `experiment_settings` will be explained later in this section.

You should manipulate the `benchmarks tree` only through three basic scripts - `wwt_prepare`, `wwt_build` and `wwt_run`. You will find man pages for all of these.

Before using `wwt_prepare` you must set up the first experiment directory, typically `dir1sw`[6], which should contain a `Makefile` and an `experiment_settings` file. `wwt_prepare` can then cre-

[5] `jsub` is a DJM command. Check the man page for details.

[6] You can change this default directory through the `-d` switch to `wwt_prepare`.

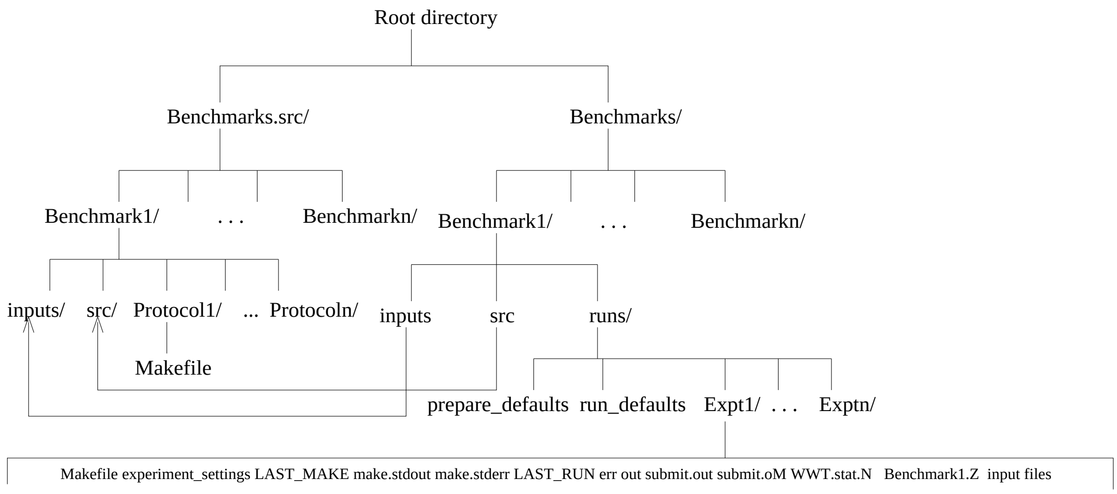

Figure 1: Benchmarks Tree

ate the experiment directories you want to set up. It reads from `prepare_defaults` names of files (which must include `Makefile` and `experiment_settings`) to be copied from the default directory (typically `dir1sw`) and names of input files which must be linked from the `inputs/` directory. The `prepare_defaults` file should be written in `perl` syntax. A typical `prepare_defaults` file would look like:

```
@COPY = ("Makefile", "experiment_settings");
@LINK = ("test.geom");
```

However, you have to go into the experiment directory created and modify the files `Makefile` and `experiment_settings` for your purpose.

You should then run `wwt_build` to build the benchmark binary in the same directory. The binary will automatically be stripped of the symbol table and compressed (e.g. `Benchmark1.Z`). `wwt_build` will create the files `LAST_MAKE`, `make.stdout` and `make.stderr`, which should contain the execution history of the build.

Use `wwt_run` to run the experiments you set up. `wwt_run` reads the default settings for the experiment from the `run_defaults` file and specific settings from the file `experiment_settings` in the specific experiment directory. `run_defaults` is written in `csh` syntax. It contains default environment variable settings for the experiment runs. It sets default values for two sets of variables. The first set relates to the Wind Tunnel and second to the DJM. The WWT related variables are SIM_NAME (name of the protocol binary), SIM_ARGS (arguments to the protocol binary, not including the name of the target code), CMD_NAME (name of the target binary), CMD_ARGS (arguments to the target binary) and CMD_IN (name of the file from which the target reads its input: optional). The DJM variables are JSUB_NPROC (number of processors), JSUB_CPU (cpu time), JSUB_MEM (memory requirements of WWT + target code), JSUB_ME (informs DJM to send you mail at the end of execution), JSUB_EXP (exports all these environment variables to the job) and JSUB_XTRA (extra DJM variables). A typical `run_defaults` file would look like:

```
#!/usr/misc/tcsh -f

if ( ! $?SIM_NAME ) setenv SIM_NAME "${WWT_ROOT}/bin/dir1sw"
if ( ! $?SIM_ARGS ) setenv SIM_ARGS ""
if ( ! $?CMD_NAME ) setenv CMD_NAME "mp3d"
if ( ! $?CMD_ARGS ) setenv CMD_ARGS "24000 32 50 test.geom"
# if ( ! $?CMD_IN )
if ( ! $?JSUB_NPROC ) setenv JSUB_NPROC "-nproc 32"
if ( ! $?JSUB_CPU ) setenv JSUB_CPU "-cputime 60min"
if ( ! $?JSUB_MEM ) setenv JSUB_MEM "-memory 145mb"
if ( ! $?JSUB_ME ) setenv JSUB_ME "-mail_end"
if ( ! $?JSUB_EXP ) setenv JSUB_EXP "-export"
if ( ! $?JSUB_XTRA ) setenv JSUB_XTRA "-server mendota"
```

The `experiment_settings` file must at least define the environment variable EXP_FLAGS to the protocol binary arguments (experiment flags). The variable SIM_ARGS is set to EXP_FLAGS. The variables defined in `run_defaults` can be set to specific values peculiar to the experiment in this file. A typical `experiment_settings` file would look like:

```
setenv EXP_FLAGS "-NPT -R01 -n 4"
setenv CMD_ARGS  "1000 32 5 test.geom"
setenv JSUB_CPU  "-cputime 10min"
```

`wwt_run` will create the files `err`, `out`, `submit.out`, `submit.oM` and `WWT.stat.`N. All these files except the last one should always be created when you run `wwt_run` correctly. These contain the execution history of the experiment run.

Three additional scripts - `wwt_batch` (to submit jobs in batches to the DJM), `wwt_jq` (to check the status of your job in DJM) and `wwt_jrm` (to delete jobs in batches) will help you to run and check your experiments. Check the corresponding man pages for details.

You can do regression tests on the `benchmarks tree` using the `wwt_regress` script. Check `${WWT_ROOT}/regression` for examples and the man page of `wwt_regress` for details.

You can create and manipulate the `benchmarks tree` with any specified root (other than `${WWT_ROOT}`) through the command line switch, `-w`, to any of the relevant scripts. There are other options listed in the man pages which you can explore. Check the `${WWT_ROOT}/benchmarks` and `${WWT_ROOT}/benchmarks.src` directories for examples.

4.6 WWT.stat.N

The statistics file, `WWT.stat.`N, contains a wealth of information about the execution of your program on the target machine. You should examine the sample `WWT.stat.2429` file in `parallel/`.

The following information is in the `WWT.stat.`N file:

- WWT execution parameters
- Virtual time statistics and breakdown for computation, TLB misses, cache misses etc.
- Protocol (e.g. Dir_1SW/Dir_iX) dependent statistics
- CSM statistics, like number of check_outs, check_ins, prefetches, prefetch distance distribution etc. for Dir_1SW
- Message counts and protocol transitions
- Broadcast statistics (for Dir_1SW)
- Cache statistics, like number of shared and private misses etc.

5 Cooperative Shared-Memory

Hill et al. [4] and Larus et al. [5] describe the Cooperative Shared-Memory (CSM) model in detail. In CSM, programmers bracket the use of the shared data by a *check_out* command before the first use of the data and a *check_in* command after the expected last use of the data. There is a prefetch annotation which allows you to prefetch data thereby overlapping communication with computation. Note that these annotations only help you to improve performance and *do not affect the correctness of the program in any way.*

The annotations referred to above are called *Check-In/Check-Out* (CICO) annotations. These are available to you as macro definitions in `${WWT_ROOT}/include/model.h`. This file is automatically included through the `ENV` declaration. See the example in `parallel+cico/`. You can use the following annotations: `CO_X` (check out exclusive), `CO_S` (check out shared)[7], `PF_X` (prefetch exclusive), `PF_S` (prefetch shared) [8] and `CI` (check_in). All of these annotations take one argument - the address of the data you are referring to. For example, to check out a shared variable A in the exclusive mode you would say

```
CO_X(&A)
```

However, the above annotations only allow to manipulate shared data at the granularity at which the cache coherence protocol is implemented (currently it is 32 bytes). If you need to manipulate larger data structures you need to add a `M_` to the beginning of each of these annotations. These annotations require an additional argument - the size of the data structure. For example, if you have a structure defined as

[7]The default mode is check out shared that arises on a read reference. Hence, you may skip this annotation when you want to check out a variable as shared read-only copy.

[8]We have neither used nor tested the `PF_S` annotation.

`struct mystruct {int A; int B} C;`

Then you can check out exclusive the whole structure by the following annotation:

`M_CO_X(&C, sizeof(struct mystruct))`

If the size of your data structure is less than a cache block (granularity at which coherence is maintained), then you might want to issue a `CO_X` only when you are the beginning of the cache block. Also, you might want to `CI` only when you have reached the last data item in the cache block. Macros are available in `${WWT_ROOT}/include/align.h` for this purpose. However, you do not have to explicitly include this header file because that is already done by `ENV`. These macros are just a prefixed version of the macros `CO_X`, `CO_S`, `PF_X`, `PF_S` and `CI`, the prefix being `ALIGNED_`. The following example should illustrate the use of these aligned macros.

```
ENV
int *A;
...
main()
{
  int i;
  INITENV();
  ...
  A = (int *)G_MALLOC(sizeof(int) * 200);
  ...
  for (i=0;i<200;i++)
    {
      ALIGNED_CO_X(&A[i]);
      A[i] = ...;
      ALIGNED_CI(&A[i]);
    }
}
```

In the above code, `CO_X` and `CI` will be executed once instead of eight times for each iteration, assuming a 32-byte cache block and four-byte integer.

There are some additional macros available in `${WWT_ROOT}/include/align.h`. These macros are

To compile your program with CICO, add the -DCICO flag to the OTHER_CFLAGS macro in the Makefile. You can selectively turn on check_in, check_out and prefetch by the following flags: -DCHECKOUT (check_out exclusive), -DCHECKIN, -DPREFETCH_X and -DPREFETCH_S. These annotations are available with all three of the currently distributed cache coherence protocols (Section 6) - namely, Dir_1SW, Dir_1SW^+ and Dir_iX.

You do not have to modify the `submit` file to run your program with CICO.

6 Cache coherence protocols

The following cache coherence protocols are supported by the WWT: Dir_1SW [4], Dir_1SW^+ [8] and Dir_iX [1]. The examples are set up for Dir_1SW. To use Dir_1SW^+, pass -NPT and -RO1 as arguments to the `dir1sw` binary (in `submit`). On an N-processor system, Dir_iX supports Dir_iB for i=0,N and Dir_iNB for i=1,N. To use `dirix` you have to recompile the target program. Go into the Makefile and change the macros PROTOCOL to dirix and OTHER_CFLAGS to -O2 -DWWT -DDIRIX. Re-make your program. You also have to change the `submit` file. The default for dirix is Dir_nNB, on an n-processor system. To convert this to a broadcast protocol you have to specify the flag -BR. To limit the number the hardware pointers to i, specify the flag -DP i. The `submit` file has two examples of dirix on a four-processor system - namely, Dir_2B and Dir_4NB.

7 Tracking memory faults

You will find a tool called `pcfind` in `${WWT_ROOT}/bin` that translates a program counter into a source file and line number of the target binary. The executable must be compiled with the `-g` flag. See the man page of `pcfind` for details. There is an example in the `bug/` directory. `bug.U` is the source file containing an MMU fault. Look into the `session` file for the execution history. The MMU fault will produce a `CMTSD_printf.pn.N` file which will tell you the address needed by `pcfind` to track the fault.

8 Profiling program execution

You can profile your program with a tool called `xcprof`. `Xcprof` needs a trace file which can be generated by supplying the argument `-xc n` to the `${WWT_ROOT}/bin/dir1sw` binary[9], where n can be either 1 or 2. Level 1 annotates only source lines. Level 2 annotates both source lines and data structures. You have to label the data structures yourself. Two macros, `SH_LABEL` and `PRIV_LABEL`, are provided to allow labeling of memory regions. These macros are enabled by compiling the target program with -DPROFILE. For memory regions not labeled, the default is to label each cache block with its virtual address.

The trace file generated has the name `XCPROF.N`, where N is the same as in `WWT.stat.`N described in Section 4.5.1. The `-O2` option in the OTHER_CFLAGS in the Makefile should be changed to `-g`.

Refer to the man page for details on `xpcrof`.

[9] There is no support for profiling in `dirix`.

9 Parameters to dir1sw and dirix

The binaries `dir1sw` and `dirix` offer you some flexibility in setting parameters for the target machine through command-line switches. These switches and their default values are listed in Table 2. The table does not list the switches for the network simulators described in the next section.

10 Network Simulation

This section describes the range of options for simulating interconnection networks with the Wisconsin Wind Tunnel. Burger and Wood describe the network simulator in more detail in [3].

Our simulated networks assume unidirectional virtual channels with one virtual channel per physical channel. The network is wormhole-routed and the routers are loosely based on Dally's Torus router. Routing is statically determined; adaptive routing schemes are currently under development.

10.1 Simulation Strategies

A range of simulation strategies has been implemented, which trade off speed versus accuracy in simulation. The flag *-icn X* determines which strategy to use, where X is one of the following:

c: Constant. This is the original WWT network model. Each message travelling through the network will take z cycles, where z is set with the *-nl* option. The default for z is 100. The quantum latency, set with the *-ql* option, should be less than or equal to z. The default quantum latency is also 100. This is the only network option that is topology-independent. This model tends to overestimate latency for the default value of 100.

p: Contention-free. The network latency is computed as the sum of the message length plus its distance through the network. The quantum latency should be set to 15 cycles for this option, which is the minimum end-to-end traversal time of any message. This model significantly underestimates network latency. A topology should be specified.

r: Random. The network latency is the sum of the message length plus network distance travelled, plus a random variable that accounts for contention. In addition to a topology, a random variable distribution and the accompanying parameters should be specified. (Using the *-dis*, *-r1*, and *-r2* flags described below). This model should also use a quantum latency of 15 cycles, and is the most accurate "fast" model currently available. The slowdown (generally a factor of 2 to 3) is almost entirely due to the reduction in quantum latency.

a: Approximate. This is an expensive discrete event simulator which reduces error to generally less than 5%. It will slow simulations down from 3 to 9 times, and should be run with a topology and a maximum quantum latency of 15 cycles.

e: Baseline. This is the most accurate simulation module, which performs an event-driven simulation which simulates each flit utilizing each channel. It slows simulations from 7 to 20 times (for 32 processors), and should be run with a topology specified and a maximum quantum latency of 15 cycles.

10.2 Topologies

Currently three classes of topologies have been implemented. The *-top* X option chooses the topology to be run. Legal values for X are:

n: None. This is the default, and is only used with the constant model. A real topology should be specified for any model other than the constant one.

k: Kary-ncube. An n-dimensional cube of degree k, where k is automatically chosen for each dimension. The simulator attempts to balance each of the n dimensions according to powers of 2. For example, a three-dimensional 32 processor system would be a $4x4x2$ cube. Links are unidirectional, and wrap around in each dimension. Routing is done in dimension-order, from lowest to highest. Dimensionality is selected with the *-d* N option, where N is the desired number of dimension. Legal values for N range from 2 to 10, and is set to a default of 2.

f: Fat-tree. This represents a tree in which constant bandwidth is maintained at each level of the tree. The processing nodes are at the leaves. Multiple routers form single logical internal nodes of the tree at higher levels. Routing is randomized on the way up the tree and deterministic on the way down. The degree of the tree is set by the *-ary* V argument, where V can range from 2 to 10. V has a default value of 4.

r: Ring. This is a ring-based topology that is intended for use with the SCI simulator. This topology is currently under development.

10.3 Random Variable Distributions

The random module allows 7 distributions to be used to generate random contentions. The *-dis* X option chooses the distribution. *-r1* Y and *-r2* Z provide the necessary parameters for the distributions as described below. The *-st* N option chooses the random number stream to use; the streams are numbered between 0 and 99 inclusively. Legitimate values for X are:

d: Dynamically chosen distribution. This chooses a distribution based on Y and Z, which represent the mean ($E[X]$) and coefficient of variation (Cx) of contention in the equivalent baseline run.

u: Uniform distribution. Y and Z represent the lower and upper bounds, respectively, for a uniform distribution.

e: Exponential distribution. Y is the parameter for the exponential distribution, while Z is a dummy argument that doesn't need to be specified.

h: Hyperexponential distribution. Y is $E[X]$, and Z is Cx.

r: Erlang-k distribution. Y is $E[X]$, and Z is the number of stages in the distribution (k).

g: Gamma distribution. Y represents the α parameter, and Z represents the β parameter.

n: Normal distribution. Y represents the μ parameter, and Z represents the σ parameter.

References

[1] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[2] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfieldand Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston Inc. The Dryden Press. Saunders College Publishing., 1987.

[3] Douglas C. Burger and David A. Wood. Accuracy vs. Performance in Parallel Simulation of Interconnection Networks. In *To Appear in the Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[4] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V).*

[5] James R. Larus, Satish Chandra, and David A. Wood. CICO: A Shared-Memory Programming Performance Model. In Jeanne Ferrante and Tony Hey, editors, *Portability and Performance for Parallel Processors*, chapter 5, pages 99–120. John Wiley & Sons, 1994.

[6] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[7] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[8] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993. Also appeared in *CMG Transactions,* Spring 1994.

Options		Description	Default	Legal
Short	**Long**			**values**
		Both dir1sw and diriX		
-a	-assoc	cache associativity	4	2^n, $1 \le n \le 5$
-acb+	-acob-entries	# of async checkout buffer entries	4000	> 0
-aa	-alloc-all	allocate sh. mem. on all nodes	*off*	*on/off* *
-an	-alloc-non-proc	allocate sh. mem. non-processing nodes only	*off*	*on/off*
-b	-block	cache block size (in bytes)	32	32×2^n, n=0,1,...
-c	-cache	cache size (in bytes)	256 KBytes	-b $\times$ -a $\times$ 2^n, n=0,1,...
-cpf	-cpf-alloc	# of CPFs to allocate initially	128	> 0
-le	-loop-on-error	loop on error instead of exiting	*off*	*on/off*
-ll	-local-latency	target local message latency (in cycles)	10	≥ 0
-mc	-msg_cost	target message send/receive cost (in cycles)	5	≥ 0
-n	-nodes	# of target nodes to use for processing	*#hn* **	1,2,...,*#hn*
-nl	-net-latency	target network message latency (in cycles)	100	> 0 ***
-pfb++	-pfb-entries	# of prefetch buffer entries	4000	> 0
-pnc	-pn-core	dump PN core files (CMTSD_core...)	*off*	*on/off*
-pm	-pvt-magic	make private data "magic" (not cached)	*off*	*on/off*
-pt	-print-trace	send debug trace to file instead of buffer	*off*	*on/off*
-ql	-quantum-length	simulation quantum length (in cycles)	100	> 0, $\le$ -nl
-s	-shared-static	make static data shared	*off*	*on/off*
-tlb	-tlb-entries	# of TLB entries	64	> 0
-tr	-trace	turn on debug tracing	*off*	*on/off*
-ts	-trace-size	# of trace buffer entries	100	> 0
-v	-verb	verbosity level	0	0,1,...,4
-xc ++	-xcprof-profiling	xcprof profiling level	0	0,1,2
-CA	-cache-access	cache access cost (in cycles)	3	≥ 0
-CF	-cache-fill	cache fill cost per block (in cycles)	1	≥ 0
-DA	-dir-access	directory access cost (in cycles)	10	≥ 0
-DF	-dir-fill	directory fill cost per block (in cycles)	1	≥ 0
		dir1sw		
-RMX	-Read-miss-excl	Read misses prefer exclusive copies	*off*	*on/off*
-ACO++	-Async-co	CO's are asynchronous (non-blocking)	*off*	*on/off*
-ASIS	-Get-asis	read misses send GET_ASIS message	*off*	*on/off*
-RO1	-RO-one	keep pointer for single shared copy	*off*	*on/off*
-RO1X	-RO-one-x	keep xor for two shared copies (implies RO1)	*off*	*on/off*
-NPT	-No-pairwise-traps	handle pairwise conflicts in hardware	*off*	*on/off*
-TLAT	-trap-latency	directory trap response latency (in cycles)	5	≥ 0
-TACK	-trap-ack	directory trap acknowledge latency (in cycles)	50	≥ 0
-TC	-trap-cost	directory trap handler runtime (in cycles)	200	≥ 0
		diriX		
-DP	-Directory-Pointers	# of directory pointers	*#hn*	1,2,...,*#hn*
-BR	-Broadcast	switch to diriB from diriNB	*off*	*on/off*
-SWIC	-Swap-in-Cache	enable swapping in cache	*off*	*on/off*

Table 2: Switches to `dir1sw` and `dirix`

* *on* (*off*) implies whether the switch is specified (or not)

** *#hn* = number of host nodes

*** WWT runs slow (fast) for small (large) values of quantum length. Try `-ql 1000 -nl 1000` during debugging

\+ ACO (asynchronous check out) does not work

++ These switches, although present, do not work for `dirix`