MassiveThreads User's Guide

September 2014

Kenjiro Taura
University of Tokyo
7-3-1 Hongo Bunkyo-ku Tokyo, 113-0033 Japan

Table of Contents

1 Getting Started

2 MassiveThreads Library

TODO: write about the library API itself.

3 Higher-Level Interfaces

3.1 Higher Level Interfaces Overview

MassiveThreads API described so far is still low level and bit burdensome as a parallel programming interface. MassiveThreads also provides higher level APIs, easier and more convenient APIs for programmers.

One is what we call TBB-compatible interface, that provides a subset of functions in Intel Threading Building Block. It does not only provide TBB-compatible interface, but also allows you to switch between various lightweight thread libraries implementing the same API. Currently supported libraries include MassiveThreads, Qthreads, Nanos++, and what we call a dummy scheduler. The last one elides task parallel primitives.

The other interface is what we call a task parallel switcher, with which you can write a single program running on top of even wider set of task parallel systems including OpenMP, Cilk, and TBB.

Besides providing a uniform API on various runtime systems, they serve another important purpose, which is to allow you to trace your task parallel programs with See ⟨undefined⟩ [DAG Recorder], page ⟨undefined⟩, a tracing tool described later in this manual. By programming in these APIs, rather than in the native API of the respective runtime systems, your are free from the burden of manually instrumenting your programs for tracing. To this end, we also provide headers to facilitate instrumentation of OpenMP and Cilk. They do not serve any purpose of making OpenMP and Cilk more convenient nor more uniform; they simply make instrumenting OpenMP and Cilk easier.

Here is the summary of choices of APIs and runtime systems.

API	**Header file**	**Runtime System**
TBB-compatible	mtbb/task_group.h	Intel TBB
TBB-compatible	mtbb/task_group.h	MassiveThreads
TBB-compatible	mtbb/task_group.h	Qthreads
TBB-compatible	mtbb/task_group.h	Nanos++
OpenMP-like	tpswitch/omp_dr.h	OpenMP
Cilk-like	tpswitch/cilk_dr.h	Cilk
TBB-compatible	mtbb/task_group.h	None (dummy)
Task Parallel Switcher	tpswitch/tpswitch.h	Intel TBB
Task Parallel Switcher	tpswitch/tpswitch.h	MassiveThreads
Task Parallel Switcher	tpswitch/tpswitch.h	Qthreads
Task Parallel Switcher	tpswitch/tpswitch.h	Nanos++
Task Parallel Switcher	tpswitch/tpswitch.h	OpenMP
Task Parallel Switcher	tpswitch/tpswitch.h	Cilk
Task Parallel Switcher	tpswitch/tpswitch.h	None (dummy)

3.2 TBB-Compatible Interface

3.2.1 TBB-Compatible Interface Overview

As of writing, it supports `task_group` class, `parallel_for` template function, and `parallel_reduce` template function. See respective sections of the TBB reference manual for their APIs.

3.2.2 Installing TBB-Compatible Interface

TBB-compatible interface is distributed as a part of MassiveThreads, so you do not do anything particular to install it besides the installation procedure of MassiveThreads.

After installation, the files constituting the API are installed as:

- *PREFIX*`/include/mtbb/task_group.h`
- *PREFIX*`/include/mtbb/parallel_for.h`
- *PREFIX*`/include/mtbb/parallel_reduce.h`

Note that they are under `mtbb` directory, instead of `tbb` directory as in the original TBB.

3.2.3 Writing Programs Using TBB-Compatible Interface

Using TBB-Compatible interface is a lot like using the regular TBB. You include `mtbb/{task_group,parallel_for,parallel_reduce}.h` instead of `tbb/{task_group,parallel_for,parallel_reduce}.h`, and use namespace `mtbb` instead of namespace `tbb`.

Here is a simple example (`bin_mtbb.cc`).

```
#include <mtbb/task_group.h>

long bin(int n) {
  if (n == 0) return 1;
  else {
    mtbb::task_group tg;
    long x, y;
    tg.run([=,&x] { x = bin(n - 1); });
    y = bin(n - 1);
    tg.wait();
    return x + y;
  }
}

int main(int argc, char ** argv) {
  int n = atoi(argv[1]);
  long x = bin(n);
  printf("bin(%d) = %ld\n", n, x);
  return 0;
}
```

Without DAG Recorder, you would compile this program as follows.

```
g++ --std=c++0x bin_mtbb.cc -lmyth-native
```

Remark 1: `--std=c++0x` is given to use C++ lambda expression at line 8, proposed in C++0x and standardized in C++11. GCC supports it since 4.5, when one of the following command line options `--std=c++0x, --std=gnu0x, --std=c++11`, or `--std=gnu11` is supplied. If your GCC does not support it, you could pass any callable object (any object supporting `operator()`). We use lambda expressions for brevity in this manual.

Remark 2: Depending on your configuration, you might need to add `-I, -L,` and `-Wl,-R` options to the command line. For example, if you install MassiveThreads under `/home/you/local` (i.e., gave `/home/you/local` to `--prefix` of the `configure` command), the command line will be:

```
g++ --std=c++0x -I/home/you/local/include -L/home/you/local/lib -Wl,-R/home/you/loca
```

3.2.4 Choosing Schedulers Beneath the TBB-Compatible Interface

With the above command, you get a program that uses TBB-compatible API with MassiveThreads as the underlying scheduler. Roughly speaking, task_group's `run` method will create a thread of MassiveThreads library via `myth_create` and wait method will wait for all threads associated with the task group object to finish via `myth_join`.

The `mtbb/task_group.h` file allows you to use threading libraries other than MassiveThreads, by defining a compile time flag `TO_XXX`. Currently, you can choose from the original Intel TBB, MassiveThreads, Qthreads, Nanos++, or None. Flags you should give to them are listed below.

Runtime system	**Flag**
Intel TBB	`-DTO_TBB`
MassiveThreads	`-DTO_MTHREAD` (or nothing)
Qthreads	`-DTO_QTHREAD`
Nanos++	`-DTO_NANOX`
None	`-DTO_SERIAL`

The last one, None, elides all tasking primitives; `run(c)` executes `c()` serially and `wait()` is a noop.

In order to use `mtbb/task_group.h` with the scheduler you chose, you of course need to install the respective scheduler and link your program with it.

3.3 Task Parallel Switcher

TBB-compatible interface unifies various schedulers under the same, TBB-compatible interface. Task parallel switcher goes one step further by defining an API that can be mapped onto OpenMP and Cilk as well.

OpenMP, Cilk, and TBB's task_group interfaces are all conceptually very similar; they all define ways to create tasks and wait for outstanding tasks to finish, after all.

Yet there are idiosyncrasies that make defining truly uniform APIs difficult.

TODO: detail the following

- mk_task_group
- create_taskc
- create_task0

- create_task1
- create_task2
- create_taskA
- call_task
- call_taskc
- create_task_and_wait
- wait_tasks

4 DAG Recorder

4.1 DAG Recorder Overview

DAG Recorder is a tool to analyze the execution of task parallel programs. It records all relevant events in an execution of the program, such as task start, task creation, and task synchronization and stores them in a manner that is able to reconstruct the computational DAG of the execution.

4.2 Installing DAG Recorder

DAG Recorder is distributed as a part of MassiveThreads, so installing MassiveThreads automatically installs DAG Recorder too. DAG Recorder does not internally depend on MassiveThreads in any way, however; you can, for example, use DAG Recorder to analyze TBB or OpenMP programs.

After installation, files directly visible to the user are the following.

- *PREFIX*`/lib/libdr.so` — library
- *PREFIX*`/include/dag_recorder.h` — include file

where PREFIX is the path you specified with `--prefix` at `configure` command line.

In most cases, you do not have to directly include `dag_recorder.h`. TBB-compatible interface or aforementioned wrappers (`omp_dr.h` and `cilk_dr.h`) will automatically include it.

4.3 Writing Programs That Use DAG Recorder

4.3.1 Common Basics

Currently, DAG Recorder supports the following task parallel APIs.

- TBB or the TBB-compatible interface See ⟨undefined⟩ [Writing Programs Using TBB-Compatible Interface], page ⟨undefined⟩.
- OpenMP. #pragma task and #pragma taskwait
- Cilk and Cilkplus. spawn and sync

Making your programs ready for DAG Recorder involves replacing original task parallel primitives with equivalent, instrumented versions and inserting calls to start/stop recording at appropriate place. You also need to specify where to start/stop instrumentation and dump the result. We provide header files to make the instrumentation as straightforward as possible. How instrumentation is done depends on the programming model you chose and detailed in the following subsections.

4.3.2 Using DAG Recorder with TBB-Compatible Interface

If you are using TBB-Compatible Interface (see ⟨undefined⟩ [Writing Programs Using TBB-Compatible Interface], page ⟨undefined⟩), the instrumentation is most straightforward and least intrusive. Let's say you have a program including `mtbb/task_group.h` such as this.

```
#include <mtbb/task_group.h>

long bin(int n) {
  if (n == 0) return 1;
  else {
    mtbb::task_group tg;
    long x, y;
    tg.run([=,&x] { x = bin(n - 1); });
    y = bin(n - 1);
    tg.wait();
    return x + y;
  }
}

int main(int argc, char ** argv) {
  int n = atoi(argv[1]);
  long x = bin(n);
  printf("bin(%d) = %ld\n", n, x);
  return 0;
}
```

Instrumentation is turned on simply by giving `-DDAG_RECORDER=2` at the command line. What else you need to do is to insert calls to `dr_start`, `dr_stop`, and `dr_dump` at appropriate places like this (`bin_mtbb_dr.cc`).

```
#include <mtbb/task_group.h>

long bin(int n) {
  if (n == 0) return 1;
  else {
    mtbb::task_group tg;
    long x, y;
    tg.run([=,&x] { x = bin(n - 1); });
    y = bin(n - 1);
    tg.wait();
    return x + y;
  }
}

int main(int argc, char ** argv) {
  int n = atoi(argv[1]);
  dr_start(0);
  long x = bin(n);
  dr_stop();
  dr_dump();
  printf("bin(%d) = %ld\n", n, x);
  return 0;
```

```
}
```

As you will see already, you should insert:

- `dr_start(0)` at the point you want to start recording,
- `dr_stop()` at the point you want to stop recording, and
- `dr_dump()` at the point you want to dump the result.

`dr_start` takes a pointer, which may be zero, to `dr_options` data structure as the argument. ⟨undefined⟩ [Controlling the Behavior of DAG Recorder], page ⟨undefined⟩ for options you can specify.

Here are the command lines to compile this program, with and without DAG Recorder

- with DAG Recorder:

```
g++ --std=c++0x bin_mtbb_dr.cc -DDAG_RECORDER=2 -ldr -lmyth-native
```

- without DAG Recorder:

```
g++ --std=c++0x bin_mtbb_dr.cc -lmyth-native
```

The reason why you set DAG_RECORDER to "2" is historical. There was a version one, which have become obsolete by now.

You could switch to other schedulers in the way described already. See ⟨undefined⟩ [Choosing Schedulers Beneath the TBB-Compatible Interface], page ⟨undefined⟩. For example, you will get the original TBB scheduler with the following command line.

```
g++ --std=c++0x bin_mtbb_dr.cc -DTO_TBB -DDAG_RECORDER=2 -ldr -ltbb
```

4.3.3 Using DAG Recorder with OpenMP

OpenMP uses directives (`pragma omp task` and `pragma omp taskwait`) to express task parallel programs. It almost always uses `pragma omp parallel` and `pragma omp single` (or `pragma omp master`) to enter a task parallel section. Here is an equivalent program to our example, written in the regular OpenMP.

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

long bin(int n) {
  if (n == 0) return 1;
  else {
    long x, y;
#pragma omp task shared(x)
    x = bin(n - 1);
#pragma omp task shared(y)
    y = bin(n - 1);
#pragma omp taskwait
    return x + y;
  }
}

int main(int argc, char ** argv) {
```

```
  int n = atoi(argv[1]);
#pragma omp parallel
#pragma omp single
  {
    long x = bin(n);
    printf("bin(%d) = %ld\n", n, x);
  }
  return 0;
}
```

We need to instrument these pragmas, for which we defined equivalent *macros* (not pragmas) in a header file `tpswitch/omp_dr.h`. This is not as straightforward as we hope, but we do not know any good mechanism to introduce a new pragma or redefine existing pragmas.

`tpswitch/omp_dr.h` defines the following macros.

- `pragma_omp_task(`*`clauses, statement`*`)`
- `pragma_omp_taskwait`
- `pragma_omp_parallel_single(`*`clauses, statement`*`)`

Without DAG Recorder, they are expanded into equivalent OpenMP pragmas in an obvious manner:

- pragma_omp_task(*clauses*, *statement*) =

  ```
  #pragma omp task clauses
    statement
  ```

- pragma_omp_taskwait

  ```
  #pragma omp taskwait
  ```

- pragma_omp_parallel_single(*clauses*, *statement*)

  ```
  #pragma omp parallel clauses
  #pragma omp single
  {
    statement
  }
  ```

So, here is DAD Recorder-ready version of the above program.

```
#include <stdio.h>
#include <stdlib.h>
#include <tpswitch/omp_dr.h>

long bin(int n) {
  if (n == 0) return 1;
  else {
    long x, y;
    pragma_omp_task(shared(x),
                    x = bin(n - 1));
    pragma_omp_task(shared(y),
```

```
                    y = bin(n - 1));
    pragma_omp_taskwait;
    return x + y;
  }
}

int main(int argc, char ** argv) {
  int n = atoi(argv[1]);
  pragma_omp_parallel_single(, {
      dr_start(0);
      long x = bin(n);
      dr_stop();
      printf("bin(%d) = %ld\n", n, x);
      dr_dump();
    });
  return 0;
}
```

This source code can be compiled with and without DAG Recorder.

- Without DAG Recorder:

  ```
  g++ -fopenmp bin_omp_dr.cc
  ```

- With DAG Recorder:

  ```
  g++ -fopenmp -DDAG_RECORDER=2 bin_omp_dr.cc -ldr
  ```

4.3.4 Using DAG Recorder with Cilk and Cilkplus

TODO: write about instrumenting Cilk and Cilkplus

- `tpswitch/cilk_dr.h`
- `spawn_(spawn f(x))`
- `sync_`
- `cilk_begin`
- `cilk_return(x)`
- `cilk_void_return`

4.3.5 Using DAG Recorder with tpswitch.h

Just give `-DDAG_RECORDER=2` and respective linker options (e.g., -lmyth-native -ldr -lpthread) to the command line.

TODO: more detailed and reader-friendly description.

4.4 Running Your Programs with DAG Recorder

4.4.1 Basics of Running Your Programs with DAG Recorder

Once you obtained an executable compiled and linked with DAG Recorder, you can run it just normally.

```
$ ./bin_mtbb_dr 20
bin(20) = 1048576
```

You will find the following three files generated under the current directory.

- `00dr.dag` — The DAG file. This is the primary file generated by DAG Recorder, from which other files are derived
- `00dr.gpl` — The parallelism file. This is a file showing the actual and available parallelism, in a gnuplot format.
- `00dr.stat` — The summary stat file. This is a text file showing, among others, the number of tasks, total work time (time spent in the application code), critical path, the number of steals, etc. The contents of this file will be explained later.

Run this program with setting environment variable `DR=0`, and you can run the program with DAG Recorder turned off.

```
$ DR=0 ./bin_mtbb_dr 20
bin(20) = 1048576
```

It still imposes a small overhead (essentially, looking up a global variable + branch) for each tasking primitive. We believe this overhead is rarely an issue, but if you want to completely eliminate this overhead, compile the program without `DAG_RECORDER=2`.

4.4.2 Controlling the Behavior of DAG Recorder

The behavior of DAG Recorder can be controlled either from within the program or by environment variables; you can pass a pointer to `dr_options` structure to `dr_start`, which has been 0 in the examples we have shown so far. If the argument to `dr_start` is null (zero), options can be set via environment variables. We will illustrate how they work.

First about environment variables. Run the program with setting the environment variable `DR_VERBOSE` to `1`, and you will see the list of environment variables and their values printed by `dr_start`. You will also see messages about files generated by `dr_dump`.

```
$ DR_VERBOSE=1 ./bin_mtbb_dr 10
DAG Recorder Options:
dag_file (DAG_RECORDER_DAG_FILE,DR_DAG) : 00dr.dag
stat_file (DAG_RECORDER_STAT_FILE,DR_STAT) : 00dr.stat
gpl_file (DAG_RECORDER_GPL_FILE,DR_GPL) : 00dr.gpl
dot_file (DAG_RECORDER_DOT_FILE,DR_DOT) : (null)
text_file (DAG_RECORDER_TEXT_FILE,DR_TEXT) : (null)
gpl_sz (DAG_RECORDER_GPL_SIZE,DR_GPL_SZ) : 4000
text_file (DAG_RECORDER_TEXT_FILE_SEP,DR_TEXT_SEP) : |
dbg_level (DAG_RECORDER_DBG_LEVEL,DR_DBG) : 0
verbose_level (DAG_RECORDER_VERBOSE_LEVEL,DR_VERBOSE) : 0
chk_level (DAG_RECORDER_CHK_LEVEL,DR_CHK) : 0
uncollapse_min (DAG_RECORDER_UNCOLLAPSE_MIN,DR_UNCOLLAPSE_MIN) : 0
collapse_max (DAG_RECORDER_COLLAPSE_MAX,DR_COLLAPSE_MAX) : 1152921504606846976
node_count_target (DAG_RECORDER_NODE_COUNT,DR_NC) : 0
prune_threshold (DAG_RECORDER_PRUNE_THRESHOLD,DR_PRUNE) : 100000
alloc_unit_mb (DAG_RECORDER_ALLOC_UNIT_MB,DR_ALLOC_UNIT_MB) : 1
pre_alloc_per_worker (DAG_RECORDER_PRE_ALLOC_PER_WORKER,DR_PRE_ALLOC_PER_WORKER) :
```

```
pre_alloc (DAG_RECORDER_PRE_ALLOC,DR_PRE_ALLOC) : 0
writing dag to 00dr.dag
dr_pi_dag_dump: 28648 bytes
writing stat to 00dr.stat
writing stat to 00dr.gpl
not writing dot
not writing text
bin(10) = 1024
```

Uppercase names within parentheses are environment variables you might want to set. They start with a prefix `DAG_RECORDER_` and many of them have a shorter version that begin with `DR_`. The list will change as our experiences accumulate. Below is the list of frequently used variables (consider other variables are still experimental).

variable	**default**	**description**
`DR_DAG`	`00dr.dag`	The name of the DAG file
`DR_STAT`	`00dr.stat`	The name of the summary stat file
`DR_GPL`	`00dr.gpl`	The name of the parallelism file
`DR_DOT`	`00dr.gpl`	The name of the DAG file in a graphviz format, which can be converted into viewable images by the dot command. You need to have graphviz package installed in yours system
`DR_TEXT`	none	DAG file, but formatted in a human-readable text file. Specify this when you want to inspect raw data
`DR_TEXT_SEP`	`|`	The field delimiter used in the text-formatted DAG file
`DR_VERBOSE`	0	Set verbosity
`DR_COLLAPSE_MAX`	a huge value	Determine how aggressively the DAG Recorder collapses subgraphs. Specifically, the value determines an upper bound of time (in clock cycles) any single node resulted from collapsing a subgraph can span. In other words, any single node in the DAG represents either a true single node (i.e., performed no tasking primitives) or a subgraph that took shorter than this number of clocks. The default is a huge value, which means the system can collapse subgraphs as much as it can. Set it to a small value to guarantee a minimum resolution.

Let us move on to the second method, which is to control the behavior from your program. As briefly noted above, this is done by passing a pointer to `dr_options` structure to `dr_start`. See *PREFIX*`/include/dag_recorder.h` for the list of fields. Note that field names were also displayed with `DR_VERBOSE=1` option above. For example, the line:

```
dag_file (DAG_RECORDER_DAG_FILE,DR_DAG) : 00dr.dag
```

tells you `dag_file` is the field name you want to set to change the file name of the DAG file.

When you change some of these fields, you will want to leave other fields to their default values. `dr_options_default(opts)` is the function that fills the structure pointed to by `opts` with default and environmentally-set values. So, the typical sequence you want to use will be:

```
dr_options opts[1];
```

```
dr_options_default(opts);
opts->dag_file = ...;
opts->whatever_you_want_to_change = ...;
   ...
dr_start(opts);
```

4.5 dag2any DAG to any data converter

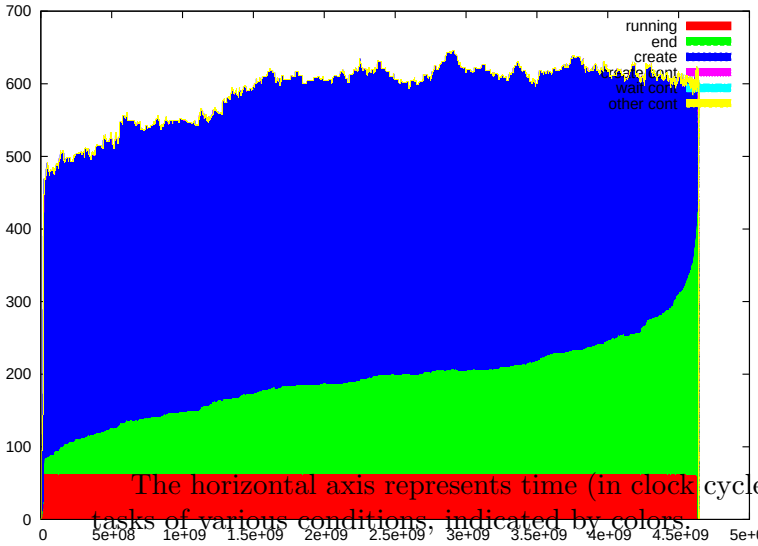
about dag2any

4.6 Viewing Recorded Data

Tools to view DAG Recorder data are still ad-hoc; ideally there should be a single tool to view the same data from many angles. As of writing, there instead is an interactive tool to show parallelism profile and a set of files derived from the DAG data, viewable by standard tools such as gnuplot. We will continue to work on developing tools to analyze DAG data from many angles and unify their user interfaces.

4.6.1 Viewing Parallelism Profile with gnuplot

By default, programs traced by DAG Recorder generates a parallelism profile as a gnuplot file. You can simply view it by gnuplot. A parallelism profile looks like this.



The horizontal axis represents time (in clock cycles) and the vertical axis the number of tasks of various conditions, indicated by colors.

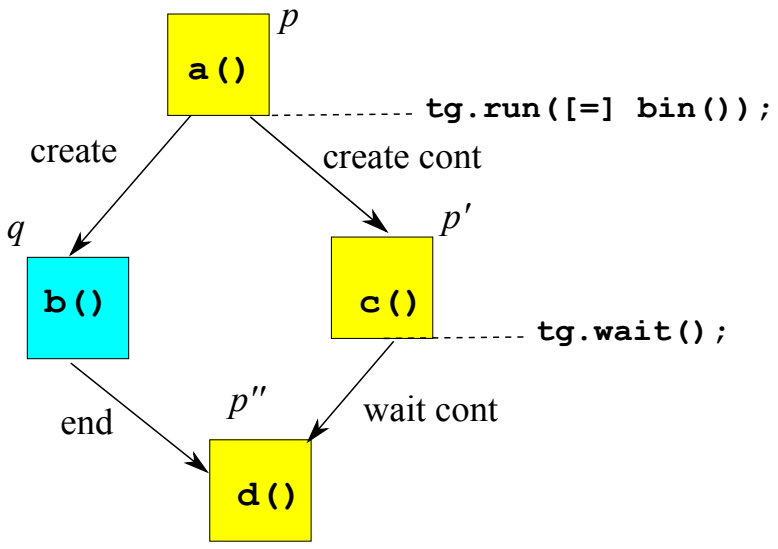
- "running" means the number of actually running tasks. The number of running tasks should never exceed the number of workers used by the execution. In the graph above, it is constant around 64. As you will have guessed already, it was an execution with 64 cores.
- all other colors mean the number of "available" or "runnable" but not running tasks; a task is available when all its predecessors in the DAG have finished. Available tasks are classified by the type of event that made them runnable.

- "end" means the task became available as its awaiting task finished.
- "create" means the task became available as its parent created it.
- "create cont" means the task became available as it created a task and continues.
- "wait cont" means the task became available as it reached synchronization point (i.e., issued tg.wait() in TBB, sync in Cilk, pragma task wait in OpenMP, etc.) and child tasks have already finished by that point.
- "other cont" means the task became available as it performed any operation that might enter the runtime system. In practice, you will never see this event.

For example, consider the following program:

```
#include <mtbb/task_gorup.h>
int main() {
  mtbb::task_group tg;
  a();
  tg.run([=] b());
  c();
  tg.wait();
  d();
}
```

and the DAG resulting from executing this program.



The label of an edge indicates how the node it points to is classified when its source node has finished. For example, the node q is counted as *create*, from the time when p finished (i.e., the task entered `tg.run([=] { b(); })`) to the time when q started.

p" becomes available when *both* q and p' finished, so how it is classified depends on which of them finished last. If q finished later than p', it is classified as *end*; otherwise as *wait cont*.

In most cases, your primary interest will be in "running." If this stays constant around the number of workers used, it means the same number of cores are maximally utilized (as

long as the operating system runs each worker on a distinct core). If it is not the case, that is, there are intervals in which the number of running tasks is lower than the number of workers used, you should check if there are enough *available* tasks.

If there are no or little available tasks in an interval, it means your program did not have enough tasks in that interval, so you might have to consider increasing the parallelism in that interval. In some cases you have simply left some section of your code left not parallelized at all, which is easily visible in the parallelism profile. A tool drview will help you spot source code locations when this happens. see ⟨undefined⟩ [Viewing DAG file with drview], page ⟨undefined⟩.

If, on the other hand, available tasks are abundant, it means the runtime system, for whatever reasons, was not able to fully exploit available parallelism. There are several possible reasons for this.

- Your tasks are too fine grained, so you observe the overhead of task creation or task stealing. For example, let's say a runtime system takes 10000 cycles from the point a task is created until the point it actually gets started, it is not counted as running during that interval of 10000 cycles. If average task granularity is only, say, 5000 cycles, then on average only 33% (5000/15000) of CPU time will be spent on actually running tasks. With a 64 workers execution, you will observe about 20 running tasks. The more overhead the runtime system imposes, the less number of running tasks you will observe.
- The runtime system somehow imposes constraints on workers that can run certain tasks, so some available tasks are left unexecuted when workers meeting the condition are busy on other tasks. A typical example is OpenMP tied tasks and TBB (where all tasks are tied); tied tasks cannot migrate once started by a certain worker.

4.6.2 Visualizing the DAG via graphviz

You can generate the DAG captured by DAG Recorder, by setting environment variable `DAG_RECORDER_DAG_FILE` (or `DR_DAG`) to the filename you want to have it in. The file is a text file of a graphviz dot format, which can then be transformed into various graphics format by graphviz tool dot.

Since a program easily generates a DAG of millions or more nodes, this feature will be useful only for short runs.

For example, you can see the DAG by any SVG viewer by the following procedure.

```
$ DR_DAG=00dr.dot ./a.out
$ dot -Tsvg -o 00dr.svg 00dr.dot
$ any-svg-viewer 00dr.svg
```

See graphviz package and dot manual for further information about the dot tool.

When you use this feature to visualize the true topology of the DAG your program generated, you might want to turn off the subgraph contraction algorithm DAG Recorder implements to save space. To this end, you can set `DR_COLLAPSE_MAX` environment variable to zero.

```
$ DR_COLLAPSE_MAX=0 DR_DAG=00dr.dot ./a.out
$ dot -Tsvg -o 00dr.svg 00dr.dot
$ any-svg-viewer 00dr.svg
```

4.6.3 Understanding Stat File

By default, programs traced by DAG Recorder generates a small text file that summarizes various pieces of information of the execution. You can view it by any text editor. Here is an example.

```
create_task           = 1048575
wait_tasks            = 1048575
end_task              = 1048576
work (T1)             = 1313026836
delay                 = 9031849743
no_work               = 11285973
critical_path (T_inf) = 91285263
n_workers (P)         = 4
elapsed               = 2589040638
T1/P                  = 328256709.000
T1/P+T_inf            = 419541972.000
T1/T_inf              = 14.384
greedy speedup        = 3.130
observed speedup      = 0.507
observed/greedy       = 0.162
task granularity      = 9601.938
interval granularity  = 3200.645
dag nodes             = 5242877
materialized nodes    = 351
compression ratio     = 0.000067
end-parent edges:
 266182 7 7 1
 1 253506 16 2
 0 8 280326 5
 1 4 9 248486
create-child edges:
 266204 0 0 0
 0 253527 0 0
 0 0 280342 0
 0 0 0 248502
create-cont edges:
 266187 7 6 4
 2 253514 9 2
 3 7 280329 3
 4 3 0 248495
wait-cont edges:
 266183 0 1 0
 0 253531 0 0
 1 0 280361 0
 0 0 0 248498
other-cont edges:
 0 0 0 0
```

```
0 0 0 0
0 0 0 0
0 0 0 0
```

- The first three items show the number of events:

  `create_task` The number of times tasks are created, not including the main task.

  `wait_tasks` The number of times wait operations are issued. Each wait may wait for multiple tasks, so this number may not match create_task

  `end_task` The number of times tasks are ended. This should be `create_task + 1`. `+1` is because the former does not include the main task, but `end_task` does.

- Then there are three numbers showing the breakdown of the total time spent by the execution.

  `work (T1)` The cumulative time (clock cycles) spent in executing the application code. Total across all cores. This does not include time spent in the runtime system (e.g., task creation overhead). If the application perfectly scales, this number should be constant no matter how many cores you used for execution. This is the area of the "running" region in the parallelism profile graph.

  `delay` The cumulative time available tasks are not executed despite there are "spare" cores not executing any task. This is the area of "available" region below the horizontal line at the number of cores in the parallelism profile graph. This value would be zero under a hypothetical "genuinely greedy" scheduler, a scheduler which immediately dispatches any available task to if any available core, without any delay or whatsoever.

  `no_work` The cumulative time cores spent without available tasks. This is the area not filled by running or available tasks below the horizontal line at the number of cores in the parallelism profile graph.

  The following is a conceptual model to understand what each of them is. Imagine we stop all workers at each processor cycle and count the number of tasks running ($= R$), as well as the number of tasks available but not running ($= A$).

  Let W = the number of workers. In this setting,

  - `T1` is the total of R over all cycles
  - `delay` is the total of min(A, W - R) over all cycles
  - `no_work` is the total of min(0, W - R - A) over all cycles

  Observe that at any point, the sum of the three terms is always W. Therefore, it always holds that

  `T1 + delay + no_work` = W x elapsed time

  In other words, `T1, delay`, and `no_work` give a *breakdown* of the whole execution time. Perfectly scalable executions have `T1` approximately the same as that of serial execution and have both `delay` and `no_work` nearly zero. They in general give you a quantitative information on why your application does not ideally scale.

  Applications that do not have enough parallelism will have large `no_work`, those that have enough parallelism that cannot be utilized by the runtime system will show a

large `delay` value, and those that have their work time increased (presumably due to cache misses due to inter-core communication, false sharing, or capacity overflows on shared caches) will show a `T1` value significantly larger than that of serial execution.

- Nine metrics that follow give you a better idea about the speedup.

`critical_path (T_inf)`	Critical path of the DAG. This is the longest time spent in a path in the DAG. The time does not include time spent in the runtime system.
`n_workers (P)`	The number of workers that participated in the execution. This is the value DAG Recorder observed during execution and, in rare occasions, may not match the number of cores you asked the runtime system to use. If, for example, the program was so short lived or created so few tasks that some cores were not used at all, you may observe a number smaller than the number you specified.
`elapsed`	Elapsed time (clock cycles) of the application. As we stated above, `elapsed` x `P` should match the sum of `T1, delay,` and `no_work`.
`T1/P`	This is simply `T1` divided by `P`. This gives an obvious lower bound on achievable elapsed time.
`T1/P+T_inf`	This is simply `T1` divided by `P`. This gives an upper bound of elapsed time by a hypothetical greedy scheduler. If the scheduler is "greedy enough" (available tasks will be executed quickly enough as long as there is an available core), the elapsed time you observed should be close to this value.
`T1/T_inf`	This is simply `T1` divided by `T_inf`, or the "average parallelism" of the execution. In general, if you hope your application to scale, this value should be much larger than the number of cores you hope to utilize.
`greedy speedup`	The speedup that should be achieved by a hypothetical greedy scheduler. It is, `T1` divided by `T1/P+T_inf`.
`observed speedup`	The actual speed up observed, which is `T1` divided by `elapsed time`.
`observed/greedy`	The ratio of the above two terms. It indicates how greedy the scheduler was.

- The following two terms give you an idea about granularity

task granularity	This is the average number of cycles between to task creations. That is, `T1` divided by the number of tasks.
interval granularity	This is the average number of cycles spent in a single DAG node, or cycles between any two consecutive task parallel operations (e.g., a task creation followed by a sync).

- Three terms that follow give you the number of DAG nodes and the effectiveness of the DAG contraction algorithm.

dag nodes	The number of DAG nodes if there would be no contraction.
materialized nodes	The number of nodes after DAG contraction. If `DR_COLLAPSE_MAX=0` (DAG contraction turned off), this should equal to dag nodes. If this value is large (default) and you use only a single core, this is always one!
compression ratio	The ratio between the two. DAG contraction is more effective (thus the value is small) when many large subgraphs are executed in a single core, and thus are contracted.

- Finally, there are five matrices that describe the number of edges in the DAG connecting two nodes executed by a pair of workers. Specifically, each matrix is P x P matrix (where P is the number of workers) whose *P[i,j]* element (i : row number, j : column number) is the number of edges of a respective type connecting from a node executed by worker i to a node executed by worker j. Five matrices are:

end-parent edges	This matrix counts edges from the last node of a task to the node that follows a wait operation that synchronized with the task.
create-child edges	This matrix counts edges from a task creation node to the first node of the created task.
create-cont edges	This matrix counts edges from a task creation node to its continuation in the same task.
wait-cont edges	This matrix counts edges from a synchronization node (a node that ends by issuing OpenMP `taskwait`, TBB `task_group::wait()` method, Cilk `sync` statement, etc.) to its continuation in the same task.
other-cont edges	This matrix counts edges from a node that ends by entering the runtime system for any reason other than task creation or synchronization to the node that starts after the operation.

4.6.4 Viewing DAG file with drview

`drview` is a tool that shows parallelism profile of an execution and allows you to zoom into an interval in it. This way it helps you pinpoint tasks executing when parallelism was low.

Prerequisites: `drview` is a python script that relies on the following libraries.

- matplotlib (Debian package name: python-matplotlib)
- gtk (Debian package name: python-gtk2 and perhaps python-gtk2-dev)

Please make sure you should be able to import respective python modules (`matplotlib` and `gtk`).

To use `drview`, you first need to convert the .dag file generated by DAG Recorder into SQLite3 format using `dag2any` tool described above. Then you pass the resulting SQLite3 file to `drview`.

TODO: We are planning to improve this crude interface, so you can directly give a `.dag` file to drview.

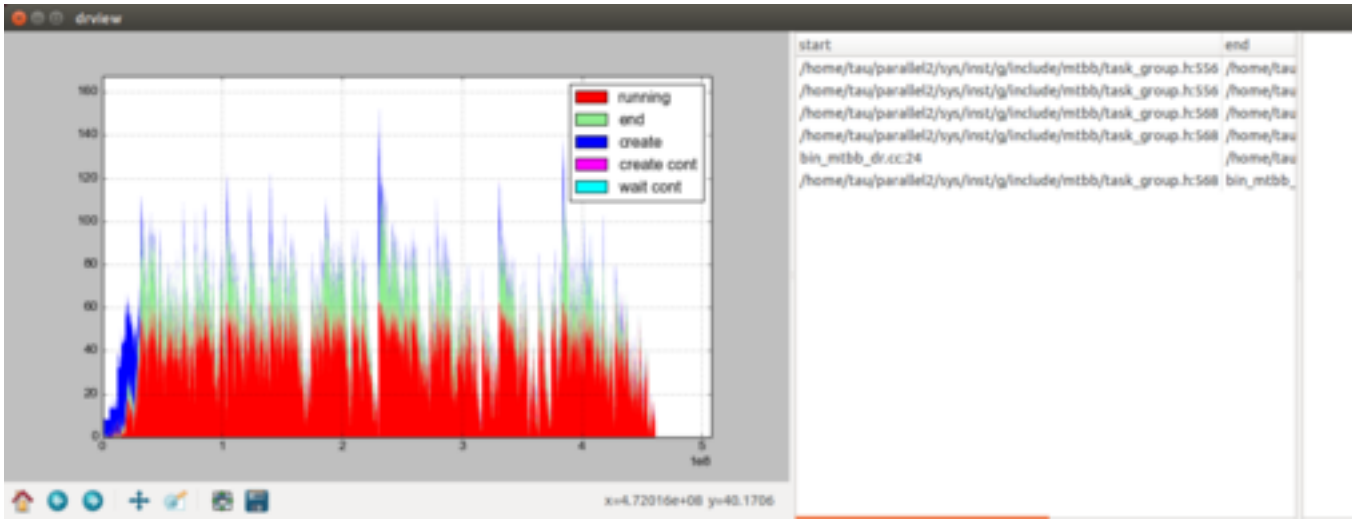
```
$ dag2any --sqlite 00dr.sqlite 00dr.dag
```

```
writing sqlite3 to 00dr.sqlite
basics:  ....................................
nodes:   ....................................
edges:   ....................................
strings: ....................................
committing
$ drview 00dr.sqlite
```

This will bring up the user interface window.

BUG: The initial configuration of panes is far from satisfactory. Please adjust their sizes manually by grabbing borders between panes. I am still trying to figure out how to configure their sizes.

After manually adjusting pane sizes, you will obtain something like this.



On the leftmost pane, you see the parallelism profile, the same information you can see by the gnuplot-formatted parallelism profile. see ⟨undefined⟩ [Viewing Parallelism Profile with gnuplot], page ⟨undefined⟩.

On the center pane is the list of DAG nodes executed. Each row represents a group of nodes that share the same start and end positions. They are ordered by the total number of cycles spent in the group of tasks. If you double-click on a row, the right pane shows the source code of the corresponding location. By clicking somewhere in the “start” or “end” column, the source code pane will display the group’s start or end position, respectively.

The most useful feature of this tool is that you can zoom into an interval of your interest in the parallelism pane. Hold the left button of the mouse pushed and specify a rectangular region in the parallelism pane, and you will see the parallelism and the task panes redrawn to reflect the tasks executed in the selected interval. This way, you can easily know the source locations of low parallelism.

4.7 Querying Recorded Data