

MIR: Installation and Usage

Ananya Muddukrishna

ananya@kth.se

1 Introduction

MIR is a task-based runtime system library. MIR is written using C99 and scales well for medium-grained task-based programs. MIR provides a direct interface for writing task-based programs. A subset of OpenMP 3.0 tasks is also supported. MIR is flexible - the user can experiment with different scheduling policies. Locality-aware scheduling and data distribution support for NUMA systems is also present. MIR supports extensive performance analysis and profiling features. Users can quickly solve performance problems using detailed thread-based and task-based performance information provided by MIR.

2 Requirements

Building MIR as a basic task-based runtime system requires:

- Machine with x86 architecture
- Linux kernel later than January 2012
- GCC and Binutils
- Scons - a build system

2.1 Optional requirements

Extended features of MIR such as profiling and locality-aware scheduling require these additional software:

- libnuma - for data-distribution and locality-aware scheduling on NUMA systems

- PAPI - for reading hardware performance counters during thread-based profiling
- Paraver - for visualizing thread execution traces
- Python 2.X and 3.X - for executing profiling scripts
- Intel Pin sources - for profiling instructions executed by tasks during task-based profiling
- R - for executing profiling scripts
- These R packages:
 - igraph - for task graph processing
 - RColorBrewer - for colors
 - gdata, plyr - for data structure transformations
- Graphviz - for task graph plotting

3 Directory Structure

The MIR source repository is structured as shown below. Directories are shown using /. Nesting is shown using indentation and +.

```

/ src - Core source files
  +/ arch - Architecture-based source files
  +/ scheduling - Scheduling policy source files
/ scripts - Various profiling and testing scripts
  +/ task-graph - Scripts for task-based profiling
/ test - programs for testing
  +/ omp - OpenMP programs
  +/ bots - BOTS programs written using MIR programming interface
  +/ with-data-footprint - Programs where tasks have explicit data footprint

```

4 Building

Building the basic MIR library is simple. Follow below steps:

1. Set `MIR_ROOT` environment variable to the MIR source repository path

```
$ export MIR_ROOT=<<MIR source repository path>>
```

2. Ensure `MIR_ROOT/src/SConstruct` matches your build intention
3. Build

```
$ cd $MIR_ROOT/src  
$ scons
```

4.1 Data-distribution and locality-aware scheduling on NUMA systems

Follow below steps to enable support for NUMA systems.

1. Install `libnuma` and `numactl`
2. Create an empty file called `HAVE_LIBNUMA`

```
$ touch $MIR_ROOT/src/HAVE_LIBNUMA
```

3. Clean and rebuild MIR

```
$ cd $MIR_ROOT/src  
$ scons -c  
$ scons
```

5 Task-based Programming Interface

MIR provides a direct task-based programming interface and supports a subset of OpenMP 3.0 tasks.

5.1 Direct task-based programming interface

MIR provides a basic interface for task-based programming. Look at `mir_public_int.h` in `MIR_ROOT/src` for interface details and programs in `MIR_ROOT/test` for interface usage details. A simple example program is shown below.

```

#include "mir_public_int.h"

void foo(int id)
{
    printf(stderr, "Hello from task %d\n", id);
}

struct foo_wrapper_arg_t
{
    int id;
};

void foo_wrapper(void* arg)
{
    struct foo_wrapper_arg_t* farg = (struct foo_wrapper_arg_t*)(arg);
    foo(farg->id);
}

int main(int argc, char *argv[])
{
    // Initialize the runtime system
    mir_create();

    // Create as many tasks as there are threads
    int num_workers = mir_get_num_threads();
    for(int i=0; i<num_workers; i++)
    {
        struct foo_wrapper_arg_t arg;
        arg.id = i;
        mir_task_create((mir_tfunc_t) foo_wrapper, &arg, sizeof(struct
        foo_wrapper_arg_t), 0, NULL, NULL);
    }

    // Wait for tasks to finish
    mir_task_wait();

    // Release runtime system resources
    mir_destroy();

    return 0;
}

```

5.2 OpenMP 3.0 task-based programming subset

MIR also supports a subset of OpenMP 3.0 tasks. Only the `task` and `taskwait` constructs are currently supported.

The `parallel` construct is deprecated. MIR creates a team of threads during initialization - when `mir_create` is called. Threads are released when `mir_destroy` is called.

Tips to write OpenMP 3.0 task programs supported by MIR:

- Initialize and release the runtime system explicitly by calling `mir_create` and `mir_destroy`.
- Do not think in terms of threads. Think in terms of tasks.
- Do not use the `parallel` construct to share work.
- Do not use barriers to synchronize threads.
- Use the `task` construct to parallelize work. Use clauses `shared`, `first-private` and `private` to indicate the data environment.
- Use `taskwait` to synchronize tasks.
- Use `mir_lock` instead of the `critical` construct or use native locks such as `pthread_lock`.
- Use gcc atomic builtins for flushing and atomic operations.
- Look at examples in `MIR_ROOT/test/omp`.

A simple set of steps we use often when producing MIR supported OpenMP programs is as follows:

1. When parallel execution is required, create a `parallel` block with `default(none)` followed immediately by a `single` block.
2. Use the `task` construct within the `single` block to parallelize work.
3. Synchronize tasks using the `taskwait` construct explicitly. Do not rely on implicit barriers and taskwaits.
4. Parallelizing work inside a master task context is helpful while interpreting profiling results.
5. Compile and link with the native OpenMP implementation (`libgomp`) and check if the program runs correctly.
6. Comment out the `parallel` and `single` blocks, initialize the MIR runtime system right in the beginning of the program and release it at the end of the program, include `mir_public_int.h`, compile and link with the appropriate MIR library (`opt/debug/prof`) to produce the executable.

The directly-programmed example shown previously written using the OpenMP 3.0 task subset supported by MIR is shown below.

```

int main(int argc, char *argv[])
{
    // Initialize the runtime system
    mir_create();

    // #pragma omp parallel default(none)
    // {
    // #pragma omp single
    // {
    // Master task context: helpful for interpreting profiling results.
    #pragma omp task
    {
        // Now parallelize the work involved
        // Work in this case: create as many tasks as there are threads
        int num_workers = mir_get_num_threads();
        for(int i=0; i<num_workers; i++)
        {
            #pragma omp task firstprivate(i)
            foo(i);
        }

        // Wait for tasks to finish
        #pragma omp taskwait
    }
    // Wait for master task to finish
    #pragma omp taskwait
    // }
    // }
    // Release runtime system resources
    mir_destroy();

    return 0;
}

```

Look at test programs in `MIR_ROOT/test` for advanced usage examples.

6 Testing

MIR does not have a dedicated test case (unit testing) suite yet. The `fib` test program (Fibonacci number calculation) is recommended for testing.

```

$ cd $MIR_ROOT/test/fib
$ scons -c
$ scons
$ echo "Executing verbose build"
$ ./fib-verbose

```

```
$ echo "Executing debug build"
$ ./fib-debug
$ echo "Executing optimized (production) build"
$ ./fib-opt
```

Other test programs in `MIR_ROOT/test` can also be used for testing.

6.1 Compiling and linking

Look at `SConstruct` - the `scons` build file - of each test program to understand how to compile and link with the MIR library. Observing `scons` build messages is also recommended.

6.2 Configuration

MIR has several configurable options which can be set using the environment variable `MIR_CONF`. Set the `-h` flag to see available configuration options.

```
$ cd $MIR_ROOT/test/fib
$ scon
$ MIR_CONF="-h" ./fib-opt 3
```

7 Profiling

MIR supports detailed thread-based and task-based profiling.

7.1 Thread-based profiling

Thread states and events are main performance indicators in thread-based profiling.

Enable the `-r` flag to get detailed per-thread state and event information in files with `.rec` extension. Each `rec` file represents a worker thread. The name of each `rec` file begins with the Unix time when the runtime system was initialized. The `rec` files can be inspected individually or combined and visualized using `Paraver`.

```
$ rm *.rec *.prv *.pcf
$ MIR_CONF="-r" ./fib-opt
$ $MIR_ROOT/scripts/mirtoparaver.py *--config.rec
$ wxparaver *--paraver.prv
```

A set of files matching the pattern «Unix time»-state-time*.rec are also created when -r is enabled. These files contain thread state duration information which can be aggregated for analysis without Paraver.

```
$ $MIR_ROOT/scripts/get-state-stats.sh <<Unix time>>
$ cat state-file-acc.info
```

Hardware performance counters can be read during thread events. This process is not fully automated and needs work from the user.

First install PAPI. Then set the PAPI_ROOT environment variable and create a file called HAVE_PAPI in MIR_ROOT/src. Next enable the preprocessor definition MIR_RECORDER_USE_HW_PERF_COUNTERS in MIR_ROOT/src/mir_defines.h. Next enable hardware performance counters of interest in MIR_ROOT/src/mir_recorder.c|h. Rebuild MIR.

```
$ export PAPI_ROOT=<<PAPI install path, typically /usr>>
$ touch $MIR_ROOT/src/HAVE_PAPI
$ cat mir_defines.h mir_recorder.{c|h}
$ scon
```

Performance counter readings will now be added to .rec files produced by enabling thread-based profiling (-r flag). The readings can either be viewed on Paraver or aggregated for analysis outside Paraver.

```
$ $MIR_ROOT/scripts/get-event-counts.sh <<.prv file>>
$ cat event-counts-*.txt
```

7.2 Task-based profiling

Task are first-class citizens in task-based profiling.

Enable the -i flag in MIR_CONF to get basic task-based information in a file called mir-stats.

```
$ MIR_CONF="-i" ./fib-opt
$ cat mir-stats
```


Enable the -g flag to generate and plot the fork-join task graph that unfolded during execution.

```
$ MIR_CONF="-g" ./fib-opt
$ Rscript ${MIR_ROOT}/scripts/task-graph/mir-fork-join-graph-plot.R mir-
task-graph color
```

MIR can also generate an instruction-level profile of tasks. Instruction-level task profiling support is restricted to programs compiled using GCC. To enable instruction-level profiling support, first get PIN sources and set the below environment variables.

```
$ export PIN_ROOT=<<Pin source path>>
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<<Pin install path>>
```

Next, edit `PIN_ROOT/source/tools/Config/makefile.unix.config` and add `-fopenmp` (note the -) to variables `TOOL_LDFLAGS_NOOPT`, `TOOL_CXXFLAGS_NOOPT`.

Next build `mir_outline_function_profiler.so` - the Pin tool which profiles tasks.

```
$ cd $MIR_ROOT/scripts/task-graph
$ make PIN_ROOT=$PIN_ROOT
```

View profiling options using `-h`.

```
$ $PIN_ROOT/intel64/bin/pinbin -t $MIR_ROOT/scripts/task-graph/obj-
intel64/mir_outline_function_profiler.so -h -- echo
```

Look at `MIR_ROOT/docs/ATG.pdf` for more information on instruction-level profiling. Also, the file `MIR_ROOT/test/fib/profile-test.sh` shows how to automate instruction-level profiling.