PMSat user manual

Luís Gil[*]

July 2007

This manual describes how to use PMSat, to provide a clear description of its usage, functionalities and options.

1 How to use this manual

If you intend to install the program start reading from the following section; otherwise if you want to learn how to use it, start reading from the section Usage. There is also a quick start guide that explains the basic usage. The inputs and outputs of the program are explained as well as error messages. For doubts consult the FAQ section.

2 System requirements and installation

The program should be executed in a cluster or grid containing MPI 1.0 or higher.

To install just extract all the source files to one directory and compile them with the `make` command. You may need to edit the `Makefile` to set the variable `INCLUDE_DIR` where the directory `/mpich/include/` or other similar is located.

The executable (`pmsat`) and the input file may need to be installed in every computer (same directory) if they don't share a common file system.

You may need to set the login without requiring a password, from the head node to all other nodes on your cluster.

3 Quick start

`pmsat` is executed in paralle by the `mpirun` command:

```
mpirun -np number-of-CPUs pmsat [options] input-file [output-file]
```

[*]degree in Computer Science at IST – Technical University of Lisbon

The parameter `number-of-CPUs` is used to set the number of copies of the program to execute. p CPUs correspond to one master to manage the search and $p-1$ workers to solve the boolean formula.

The formula to test is given in a file with extension `.cnf` or `.bcnf`. This file must be placed in all computers (same directory) if they don't share a common file system.

The most important options of the program are `-n` and `-m` to set the number of variables to assume and the search mode respectively.

For k variables, the search modes *Sequential* (`s`) and *Random* (`r`) make 2^k tests while the *Few first* (`f`) and *Many first* (`m`) make only $2 \times k$ tests.

Example 3.1 *Setting the number of CPUs to 4 (3 workers and the master), the search mode to* Sequential *and the number of variables to assume to 5.*
```
mpirun -np 4 pmsat -m s -n 5 file.cnf
```

Example 3.2 *Setting the number of CPUs to 9 (8 workers and the master), the search mode to* Many first *and the number of variables to assume to 12.*
```
mpirun -np 9 pmsat -m m -n 12 file.cnf
```

You must provide a name for the output file to get and save the model of the formula. When it is not indicated, the result displayed in the screen is just SATISFIABLE or UNSATISFIABLE.

To abort the execution of the program press the keys `Ctrl+C`.

4 Usage

The `mpirun` command is used to run the program:
```
mpirun [mpirun_options...]  <progname> [options...]
```

Its main options are `-np` to indicate the number of processors where the program will run and `-machinefile` to indicate a file with the name of the servers where the application will run. p CPUs correspond to one master to manage the search and $p-1$ workers to solve the boolean formula.

The program `pmsat` is executed through `mpirun`:
```
mpirun -np number-of-CPUs pmsat [options] input-file [output-file]
```

If the computers of the cluster or grid do not share a common file system, the input file must be copied to every nodes where the program will run and placed in the same directory.

Several behaviors and features can be enabled by the options or from a configuration file described in the next section.

The minimal set of arguments you must give to invoke the program are:

1. amount of CPUs (k workers + 1 master);
2. name of the executable (`pmsat`);
3. input file.

The optional arguments are:

1. file with a list of machines that will run the program;
2. search mode, number of variables and other options;
3. output file, to save the solution of the formula.

The output file is used to save the model of the formula. When it is not indicated, the result displayed is just SATISFIABLE or UNSATISFIABLE. You must indicate the output file if you want to save the model of the formula.

The program may be executed without the `mpirun` command and perform a sequential search in the local machine, like the original MiniSAT. Its usage is `pmsat -m l input-file [output-file]` where `-m l` stands for *Local* search mode.

The program may be interrupted if you press `Ctrl+C`.

5 Options

To see the full list of options of the program (in Figure 5) type `pmsat -h`. It is not necessary to use the `mpirun` command.

Extra features are set as flags: `-c` to remove assumptions with conflicts, `-l` to share learnt clauses and `-r` to remove the learnt clauses after each execution of the solver.

To configure parameters we have: `-s` to set the method to choose the variables to assume, `-z` and `-t` to set the maximum size and amount of learnt clauses and `-a` to set the assumptions–CPUs ratio (*acr*).

All the previous options when omitted, assume a default value or retrieve it from a configuration file if available. The purpose of the file is to define new default values for parameters and flags without setting them in the command line. But if a configuration file is loaded and some option is given in the command line, it overrides the value in the file.

To manage configuration files there are the options `-g` and `-f` to generate and import configuration files respectively.

```
USAGE: mpirun -np number-of-CPUs ./pmsat [options] input-file [output-file]
Options:
  -h, --help  display this help and exit

  -v, --verbose  enable the verbose mode

  -n <value>, --number-of-vars  the number of variables to assume

  -m <arg>, --mode  assumptions generation / search mode:
         l - mode without assumptions to use just when number-of-CPUs is 1
         Progressive mode has <arg>:
         f - start from the assumptions with few literals
         m - start from the assumptions with many literals
         Equal mode has <arg>:
         r - test the assumptions randomly
         s - test the assumptions sequentialy

  -f <file>, --config-file  read a given configuration file.

  -g <file>, --generate-config  generate a configuration file and exit. The
program is able to work without a configuration file.

  The following options may be set in the configuration file
  (command line arguments override them):

  -c, --conflicts  detect and delete assumptions with conflicts

  -l, --learnts  enable the share of learnt clauses

  -z <value>, --learnts-max-size  set the max size of the learnt clauses to
share (default is 20)

  -t <value>, --learnts-max-amount  set the max amount of learnt clauses to
share (default is 50)

  -r, --remove-learnts  remove all the learnt clauses after each solve call.
                If its share is enabled they are sent before removal.
                By default the learnt clauses are kept.

  -a <value>, --assumps-cpus-ratio  set the ratio between the number of
  assumptions to solve and the worker CPUs (default is 3)
  It is used in the automatic calculation of the number of literals and mode.

  -s <arg>, --selection  methods to select the variables to assume with <arg>:
         o - variables with more occurrences(default)
         b - variables in the biggest clauses

  input-file: may be either in plain/gzipped DIMACS format or in BCNF.

  output-file: the file where the result is written.
```

Figure 1: Program's options

There are some precautions to have when editing the file: each line may have a maximum number of 60 characters and there can't be any space between the name of the option and the equal sign, because of the simplicity of the parser.

Example 5.1 *To create a configuration file.*
`pmsat -g minisat.conf`
The configuration file created by the program is displayed in Figure 2.

Example 5.2 *To load a configuration file and override the size and max amount of learnt clauses.*
`mpirun -np 4 pmsat -f minisat.conf -z 40 -t 100 file.cnf`

```
#keep comments in separate lines
#Do not insert spaces !

#max size of learnt clauses
LEARNTS_MAX_SIZE=20

#share learnt clauses ?
SHARE_LEARNTS=false

#remove learnt clauses after each solve?
REMOVE_LEARNTS=false

#share conflics ?
CONFLICTS=false

#max amount of learnt clauses to send
LEARNTS_MAX_AMOUNT=30

#Ratio between the number of assumptions.
#and the amount of CPUs.
#Used in automatic calculations of
#variables to assume and search mode
ASSUMPS_CPU_RATIO=3

#how select the variables to assume:
#can be more_occurrences or bigger_clauses
VARIABLE_SELECTION=more_occurrences
```

Figure 2: Configuration file

The options `-m` and `-n` can only be set by the user or calculated automatically and do not appear in the configuration file. This was decided because you may want to select different modes or assume a distinct amount of literals each time you run the program.

As was said, the most important options are precisely the previous two. Their values, when omitted, are set to respect the assumptions–CPUs ratio, i.e., create an amount of assumptions proportional to the number of workers without exceeding a threshold. The ratio should be bigger than 1 (default is 3).

The search modes conduct the search by different paths and for k literals, the *Sequential* and *Random* modes will make 2^k tests, while the *Few first* and *Many first* will make only $2 \times k$ tests.

6 Examples

Here you can find a set of examples with several different ways to invoke the program.

Example 6.1 *Running the program in 5 CPUs (4 workers and the master) with search mode* Few first *and sharing learnt clauses. The number of variables is automatically calculated.*

```
mpirun -np 5 pmsat -m f -l file.cnf
```

Example 6.2 *Running the program in 6 CPUs (5 workers and the master), sharing conflicts and selecting the variables in bigger clauses.*

```
mpirun -np 6 pmsat -c -s b file.cnf
```

Example 6.3 *Running the program in 10 CPUs (9 workers and the master), sharing conflicts, removing learnt clauses after each execution of the solver and giving an output file.*

```
mpirun -np 10 pmsat -c -r file.cnf out.txt
```

Example 6.4 *Running the program in 8 CPUs (7 workers and the master), reading a configuration file, sharing learnt clauses and setting them to have a maximum size of 10 literals (overriding the value given in the config file).*

```
mpirun -np 8 pmsat -f minisat.conf -l -z 10 file.cnf
```

7 Inputs and Outputs

The input with boolean formulas are given in a file in DIMACS/CNF format (.cnf) or BCNF (.bcnf).

The DIMACS/CNF file format is composed by:

1. comments, that is, lines beginning with the character `c`;
2. a preamble containing informations about the instance: the file format, the number of variables and clauses: `p format nvars nclauses`;

3. the clauses, each one encoded as a sequence of non-null numbers ranging from -nvars to nvars and separated by zero. Positive numbers represent the corresponding variables and negative numbers denote their negations.

It is not necessary that every variable appear in the formula, as one can see in the example where the variable 2 is not used.

Example 7.1 (DIMACS/CNF file)
```
c comments here
c next is preamble
p cnf 5 4
1 -3 4 0
-1 3 0
-5 -4 0
5 -3 0
```

The result of the search (SATISFIABLE/UNSATISFIABLE) is printed to the screen and the model may be saved in a file if a name is given in the arguments.

The other interesting output produced is a file with the extension `.time` containing all the statistics of the execution like the number of calls and the times spent. It shows:

- master's initialization time to read the input file, parse and setup the parameters;
- conditions of the execution: number of workers, variables, search mode, variables's selection mode, erased assumptions and options of learnt clauses;
- workers's execution statistics indicating the number of times the solver ran, sets of learnt clauses sent and received and the CPU time (user plus system time) spent in computation and by the master to process worker's data;
- master finalization time to write outputs and post processing;
- total time as the sum of master's initialization and finalization times and $worker + master$ time from the worker that took more time;
- total wall time to see the influence of the system's load and the delay caused by communication.

The name of this file is made following a pattern, by joining the name of the input file and the conditions of execution: number of CPUs, search mode, amount of variables and method to choose them, existence of conflicts and learnt clauses (and their settings).
A XML file with the same content is also created, to allow a statistic processing by automatic tools.

Example 7.2 *After solving the file* `fpga10_11_uns_rcr.cnf` *in 3 CPUs (2 workers and master),* Random *mode, assuming 3 variables, removing assumptions with conflicts and sharing learnt clauses, the program will create a file named* `fpga10_11_uns_rcr.cnf-3-r-3-o-c-l-z20-t50.time` *with the statistics:*

```
Master initialization time: 0.005998 secs

Workers: 2
Variables to be assumed: 3
Search mode: r
Variable's selection mode: o
Erased assumptions: 2
Learnt max amount: 50
Learnts max size: 20

Worker 1:
solve() was executed 2 times
Total time spent by worker: 41.090568 secs
Total time spent by master with this worker: 0.001000 secs
Databases received: 1
Databases sent: 2

Worker 2:
solve() was executed 4 times
Total time spent by worker: 41.034564 secs
Total time spent by master with this worker: 0.001000 secs
Databases received: 1
Databases sent: 4

Master finalization time: 0.000000 secs

Total CPU time: 41.097566 secs

Total wall time: 42.758681 secs
```

8 Hints for better performance

There are no better settings to get a good performance. Through experience, you may see what the options make the program take less time to solve the formulas. Here are some, that may not always be the best:

- keep the granularity high: select an amount of variables large enough to generate more assumptions than the amount of workers (4 or 5 assumptions for each worker);
- nevertheless, try a search with few variables and workers (2 or 3). Some problems are easily solved;

- avoid the *Sequential* mode. The others provide better performance;
- do not use the option `-r` to remove the learnt clauses because they will guide the following searches;
- use only conflicts when you have many assumptions and in *Random* and *Sequential* modes. The same for learnt clauses;
- after solving take a look at the file with the statistics because it gives a clear picture where the time was spent.

9 Error messages

The error messages that the program can present are:

- `ERROR! Could not open file: filename` – the file does not exits or you don't have permissions to open it.
- `ERROR! Not a BCNF file: filename` – the file has an incorrect format.
- `ERROR! BCNF file in unsupported byte-order: filename` – similar to previous error.
- `PARSE ERROR! Unexpected char:` – a character different than a digit appeared in the input file.
- `ERROR! configuration file filename not found! Setting default values...` – the file with the given name was not found and the program will continue assuming the default internal values for the parameters.
- `ERROR! Could not open file: filename` – could not open the input file.
- `ERROR! Cannot write output to file!` – the program could not write the results for the output file due lack of permissions, disk space or other reason.
- `ERROR! Number of literals to assume is bigger than number of variables in formula!` – if you tried to assume more variables than those in the formula.
- `ERROR! the number of CPUs is 1 but the selected mode is not LOCAL!` – if the program was invoked without `mpirun` and with a search mode different than *Local*. The program will set the search mode to *Local* and continue.
- `ERROR! the number of CPUs is greater than 1 but the selected mode is LOCAL!` – the inverse situation, with many CPUs and the *Local* mode. The search mode is recalculated and the program continues.

10 FAQ

What is MPI ?

MPI stands for Message Passing Interface and is a specification of an interface to create parallel programs and manage the communication between the processes. It is considered the de-facto standard for parallel computing. There are several vendors that provide MPI as a library to be called by the parallel programs. These programs are executed in clusters of servers or workstations.

Can I use MPI just in Unix/Linux, or is available for Windows also ?

There are several MPI implementations from different vendors for both operating systems. We used MPICH from Argonne National Laboratory that provides versions for Windows and Unix/Linux.

Can I run MPI in a heterogeneous (Linux and Windows) clusters ?

You should get that information from your MPI vendor. There are implementations that just work on homogeneous clusters (same operating system and version, architecture, libraries) while others may be used in heterogeneous clusters, most of the times with some restrictions (e.g. the size of datatypes).

What are the contents of the file with the list of machines: IPs or hostnames ?

The file given after the option `-machinefile` contains the hostnames of the machines where the processes will be created.

What is the average speedup of the program ? When should I use parallel vs sequential ?

The performance varies with the problem to solve. We noticed that the program is faster, but in certain problems or with certain options it takes the same time or even more. You should use the parallel program with problems that take much time to solve by the sequential program, for instance more than 300 seconds. The objective is to decrease the execution time in sets of ten or even hundreds of seconds.

But is there a general idea based in the size of the problem and the amount of processors ?

No. There are small problems that take a lot of time to solve and big problems that are solved really fast.

What is super–linear speedup ? Does it happens in the program ? Why ?

Speedup is time of the sequential algorithm divided by the time in taken in p processors. Super–linear speedup happens when the speedup achieved is bigger than the number of processors used to solve the problem.

A super–linear speedup occur specially with SAT problems. Their cause is related to how quickly each partition of the subspace is solved and the solution is found. The solving process is made by a search algorithm that is guided by the constraints of the problem. This means that each partition takes a different time to be explored, even when they have the same dimensions. Sometimes the partition with the solution is quickly explored leading to a super–linear speedup.