PMP: Poor Man's Parallelizer. User Guide

Peter Leikauf, Seminar for Applied Mathematics,
Swiss Federal Institute of Technology ETH, CH-8092 Zürich

January 8, 2001

1 Table of Contents

2 Purpose

The PMP System (Poor Man's Parallelizer) provides an easy, robust, platform-independent mechanism for parallelizing big tasks consuming very large amounts of CPU time (like a few weeks) in a heterogeneous environment, provided these tasks can be subdivided into independent sub-tasks. Exhaustive searching is an example for such a task. Utmost care has been taken to make sure the software only needs very basic libraries found on nearly any operating system: a standard ANSI-C library and internet sockets. Only the node acting as server will make use of a file system. Furthermore, a running system will survive (within reasonable limits) all the typical incidents that may occur from time to time, such as failure of one or several clients, failure of the server itself or even failure of the whole network. Besides, a run can be interrupted and restarted at any time.

3 How it Works

To make use of PMP, it is first necessary to create a set of N subtasks, numbered from 0 to N-1. One specific node of a network acts as the server whose address must not change during the run. The server software does not act by itself but assignes to each connecting client a task (identified by its number), which is then supposed to be handled by that client. If a client reports some task as completed (see below how this works), the server marks this task accordingly and assigns a new task to the reporting client - until all tasks are completed. The number of clients taking part can increase and decrease all the time. It is important to understand that the server does NOT monitor the activities of the clients - only when a client reports some task as done, then this task is taken off the todo-list. This leads to a certain overhead near the end of a run (there are always multiple runs for the last few tasks), but makes the whole system extremely simple and robust.

To survive hardware or network incidents, a client, after having done its work, repeatedly tries to connect to the server for a certain time (currently once per minute for 100 times) before giving up. This allows for many kinds of interruptions, including shutting down the server node for some time due to hardware or network maintenance.

4 Setting up a Run

Using PMP only makes sense if the CPU time for completing one subtask is reasonably large. As a rule of thumb, one hour to one day is a good choice. Shorter times could cause unwanted network overhead, longer times might lead to regrettable loss in case of client failure. So the first step is to figure out a good number of subtasks to be completed. Before starting the server process (program `BTServer`), we have to create a new status file (called `BTServer.DAT`). This file must contain exactly two lines initially: the number of tasks in the first line and a zero followed by the same number in the second line. So, for setting up a run for 120 000 tasks, we would have to create a file `BTServer.DAT` looking like this:

```
120000
0 120000
```

After that, the server process can be invoked, normally as a background job, by the command `./BTServer &`.

From now on, a client can be started anywhere, provided it is able to handle a subtask and to connect to the server process. This could be on the same machine as the server (which may only make sense with multi-processor hardware), on a local network or anywhere the internet can be accessed. Invoking a client process is done as follows on a Unix system:

```
./BTClient <n> <serveraddr> <cmd> [arg1][arg2]...[argN]
```

where `<n>` is a client identifier, usually 1. Its only purpose is to uniquely identify a client process in case more than one of them should be running on the same node (again, this is usually only done on a multi-processor machine). `<serveraddr>` is the internet address of the server node, such as `"xy.anywhere.net"` or `"212.23.56.193"` or even `"localhost"`. `<cmd>` is the command to be invoked by the client (as a subprocess) to solve an assigned subtask (optionally followed by `arg1..argN)`, which will then be followed by the subtask number itself. As an example, suppose each client has a `perl` script called `solver.pl` which is able to solve a subtask, given as the only command line parameter. Thus, `./solver.pl 5` would solve subtask 5. To invoke a client process now, we would typically type:

```
./BTClient 1 xy.anywhere.net ./solver.pl &
```

The standard output of the subprocess invoked by the client is then reported to the server who puts it into the server log file. Its structure are completely up to the user, e.g., for an exhaustive search task it could be empty if nothing was found or some number otherwise. Once such a client is started, it usually quickly gets its first subtask from the server and starts running the appropriate command as a separate subprocess, currently with minimum priority (`nice +20`). The client will only terminate after it was told by the server that all tasks have been completed. However, it is possible at any time to kill a client and restart a new one.

5 Monitoring Progress

The server process collects all assigned tasks and the reported results of every client run in a log file called `BTServer.log`. Each line of it contains the

exact time, the connecting node including its client identifier, the number of seconds it took to complete its run and the result (= output of the command invoked by the client). This logfile is being flushed on every contact and can be observed, e.g. with `tail BTServer.log`. Depending on the structure of a subtask solver's output, it may be scanned for results more or less easily with tools such as `grep`. As an example of a non-trivial logfile analyzer, the tool `BTCheck.pl`, a small `perl` program, can be taken as a template. It nicely handles the case when the output of a client run is 0 when nothing was found, or a list of numbers otherwise.

Besides the log file, the already mentioned status file `BTServer.DAT` is redundantly maintained on every contact and lists the completed tasks in human-readable, but compressed format. Should one of these files be destroyed by an unusual accident such as media failure, the run could at least be set up again (although this would require looking into the server program `BTServer.c)`. This may be important if a run takes several months to terminate.

6 Interrupting and Stopping

Shutting down the server process `BTServer` doesn't do any harm to a run, provided it is restarted within the period in which the clients try to connect to the server. Thus, a run can even survive a hardware or network reconfiguration and - most importantly - a reboot of the server node.

After a run has completed, the active client processes will be notified on their next contact and will then immediately stop. This is the normal way a run ends itself - only the server has to be shut down manually some time afterwards.

To end a run before completion, it thus suffices in principle to shut down the server and wait long enough for the last client to stop. However, to do a clean shutdown it is better to kill the client processes first to avoid unnecessary network traffic.

7 PMP on a Supercomputer

If some or all of the client nodes are part of a supercomputer with independent nodes (such as a Beowulf cluster), it is convenient to maintain a list of the active client nodes and a small script for stopping/starting all of them at

once using tools like `rsh` on Unix systems. Furthermore, since the log- and status-files are flushed upon every contact with a client, it may be a good idea to put them onto a local file system and not onto an NFS in order to reduce hidden network traffic.

8 Technical Information

The PMP system consists of these files:

```
BTServer.c     server program
BTClient.c     client program
SIS.h, SIS.c   small library used by both server and client
BTCheck.pl     sample Perl script for analyzing the server log file
```

The C programs will compile on Linux and SunOS systems and can easyly be adapted to work also on Windows/Mac or other Unix variants.

`BTServer` acts as a mere dispatcher and takes modest CPU time even when running for months, provided the average elapsed time of a client process is within reasonable limits (see above).

`BTClient` creates a new subprocess for every subtask (currently with lowest priority). There are three reasons to do it this way:

1. software independency (`BTClient` has no direct link to whatever it takes to solve a subtask),
2. simplicity and robustness (`BTClient` might run for months and must not be affected by problems of the subtask-solving software such as memory leaks),
3. different process priorities (although it can be meaningful to assign low priority to subtask-solvers, `BTClient` itself needs normal priority to be able to contact the server within the timing constraints).

9 Suggested Improvements

- Put server constants (like retry interval) into a config file
- Create a simple logfile analyzer reporting completed tasks
- Use `perl/python` instead of C, since those are ubiquitous today