PGPVM
Performance Visualization Support for PVM*
Version 2.0

Stéphane Bizard, Stephan Dalhem, Sébastien Veigneau
Brad Topol, Vaidy Sunderam, Anders Alund

April 2, 1997

Abstract

PGPVM is an enhancement package for PVM 3.3 that produces trace files for use with Paragraph, which is a tool for visualizing performance of parallel programs. This document presents the new release PGPVM2 which is more flexible and allows to trace any kind of PVM program.

1 Motivation

As computers are getting more and more connected via world-wide networks, a new chance of gaining computing power from the net arises. Thus, a collection of heterogeneous even small computers can now clusters computing power to exceed that of the sequential computers. However, it seems clear that the full potential of such distributed systems cannot be realized without similar advances in parallel software, just because parallel computation in a distributed environment is not so easy whereas it seems quite common to run large sequential scientific calculations on a single huge processor.

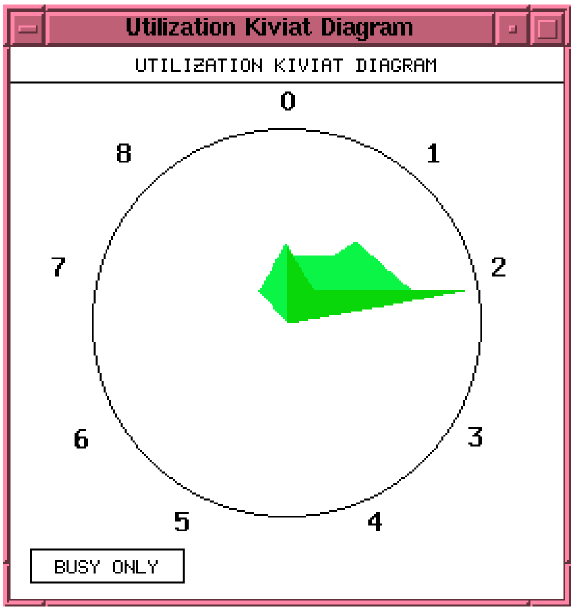
In other words, the software has to be adapted to the new situation. It means that new algorithms have to be developed or sequential ones to be adapted, but it also implies that the program must take into account the distribution of the computation onto the different machines. This is why PVM has been developed. The PVM software provides an environment to build and use distributed programs in an efficient manner. It provides a framework in which one sees a collection of heterogeneous computers as a single parallel virtual machine.

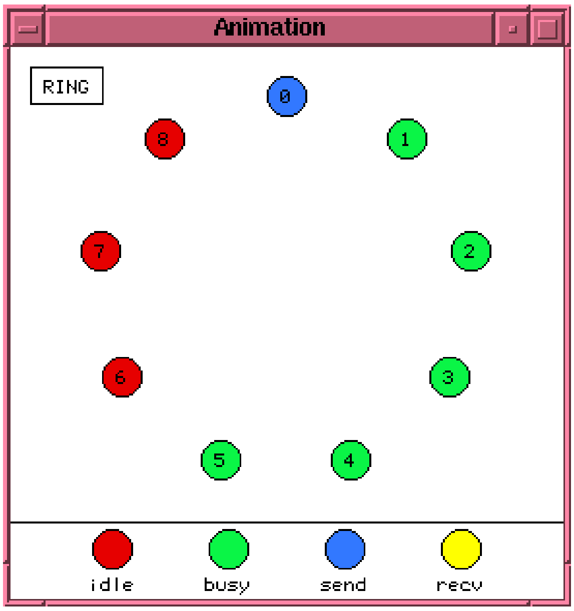
On the other hand, to be able to analyze and understand the behavior of parallel programs, in order to improve their general performance, it seems quite natural to use graphical visualization techniques. In fact, this is true for parallel computers including an advanced architecture but it remains true in the case of distributed systems. In both cases, the user will gain insight into the analysis of large volumes of trace data by using a special tool devoted to this purpose. Such tools exist, among which Paragraph which provides a detailed, dynamic, graphical animation of the behavior of message-passing parallel programs.

*This research was supported in part by the U. S. Department of Energy grant DE-FG05-91ER25105, Army Research Office grant DAAH04-96-1-0083, NASA grant NAG 2-828, and Procope.

Now, the PGPVM software has been developed to connect PVM and Paragraph. More explicitly, PGPVM is an enhancement package for PVM 3.3 that produces trace files for use with standard Paragraph. PGPVM attempts to give an accurate portrayal of applications by minimizing the perturbation inherent with this type of monitoring. PGPVM does not utilize standard PVM tracing but instead its own buffered tracing techniques to provide more accurate monitoring information. Further, PGPVM provides a shellscript that performs some postprocessing and produces a <file>.trf, *i.e.* a standard Paragraph trace event file. Tachyon removal and clock synchronization are performed during postprocessing when necessary.

The new version PGPVM2, has been developed to be more flexible than PGPVM. It allows to trace any PVM program and provides new possibilities with respect to the interface between PVM and Paragraph. PGPVM2 uses a new external task, named pg_administrator, that counts all tasks that want to be traced. In particular, this administrator is able to simulate a parallel virtual machine having a fixed number of processors. This is useful at the level of Paragraph when for instance a PVM program uses, say, 100 tasks while at most 8 are effectively running in parallel. The program which has been developed for a 8-processor parallel computer can thus be tested using PVM and analyzed using PGPVM2 because Paragraph will only see 8 processors. For instance, it gives the following Paragraph visualization windows:





which respectively describe the utilization of individual processors (or more precisely processes) together with the overall load balance across processes, and the status of each process (busy, idle, sending, receiving) in the virtual parallel system represented as a graph whose nodes denote processes and whose arcs represent communication events between processes. This is more convenient even if one can learn how to decode Paragraph trace files, as suggested by the following example:

-3 -901 0.023579 0 -1 0	trace_start clock 23578 node 0
-3 -52 0.061272 0 -1 1 2 -1	recv_blocking clock 61271 node 0
-3 -901 0.182373 1 -1 0	trace_start clock 182373 node 1
-3 -21 0.189360 1 -1 3 2 432 5 0	send_entry clock 189359 node 1 to 0 type 5 lth 432
-4 -21 0.189726 1 -1 0 0xb2	send exit clock 189725 node 1
-3 -52 0.189773 1 -1 1 2 -1	recv_blocking clock 189772 node 1

```
-4 -52 0.190360 0 0 3 2 432 5 1          recv_waking clock 190359 node 0 from 1 type 5 lth 432
-3 -21 0.193675 0 -1 3 2 136 6 1          send entry clock 193674 node 0 to 1 type 6 lth 136
-4 -52 0.236901 1 0 3 2 136 6 0          recv_waking clock 236901 node 1 from 0 type 6 lth 136
-4 -21 0.240374 0 -1 0                                              send exit clock 240373 node 0
-3 -52 0.240422 0 -1 1 2 -1                                      recv_blocking clock 240421 node 0
-3 -21 0.433116 1 -1 3 2 640 5 0          send entry clock 433115 node 1 to 0 type 5 lth 640
-4 -52 0.434116 0 0 3 2 640 5 1          recv_waking clock 434116 node 0 from 1 type 5 lth 640
-3 -21 0.439659 0 -1 3 2 200 6 1          send entry clock 439658 node 0 to 1 type 6 lth 200
-4 -21 0.440059 0 -1 0                                              send exit clock 440059 node 0
-3 -52 0.440105 0 -1 1 2 -1                                      recv_blocking clock 440104 node 0
-4 -21 0.445879 1 -1 0                                              send exit clock 445879 node 1
-3 -52 0.445931 1 -1 1 2 -1                                      recv_blocking clock 445930 node 1
-4 -52 0.446151 1 0 3 2 200 6 0          recv_waking clock 446150 node 1 from 0 type 6 lth 200
-3 -901 0.446960 2 -1 0                                           trace_start clock 446960 node 2
```

Before using PGPVM2, let us see what must be done to install the library.

2 Installing the PGPVM2 Library

Before the PGPVM2 library can be correctly compiled, please make sure of the following:

① You have installed PVM release 3.3 or later.

② You are in the pgpvm2 directory that you must create and have untarred pgpvm2.tar. To untar, type `tar -xvf pgpvm2.tar`. Further, the $PVM_ROOT has been set and the standard PVM program aimk exists and is found by your PATH!

③ You are able to write to $PVM_ROOT/include and to $PVM_ROOT/lib as PGPVM2 needs to install itself.

④ You are able to write to the executable path for PVM tasks in order to install the pg_administrator program from the ADMIN subdirectory.

As previously touched upon, application programs must be recompiled and linked to the newly created PGPVM2 library, libpgpvm2.a. Moreover, the program must link to the libpvm3.a library. PGPVM2 does not rely on the PVM group library, libgppvm.a, and this need not be linked unless the application uses the group routines. Further, the application must `#define USE_PGTRACE`. This must be done before the header file `#include "pgpvm2.h"` so place this macro in the source code or set it during compilation. Please refer to the Makefile.aimk in the EXAMPLE/C and EXAMPLE/FORTRAN subdirectories for proper compilation techniques.

When an application has no need for PGPVM2's tracing, simply remove USE_PGTRACE macro (either from source or the Makefile, wherever you placed it) and recompile. The other modifications do not have to be removed and the application will execute as if PGPVM2 routines had not been linked to the application.

3 What Must Be Done to Correctly Compile the Examples

Several example programs are provided into the subdirectory `EXAMPLE` of `pgpvm2`. Two subdirectories (`C` and `FORTRAN`) contain respectively `C` and `Fortran` example programs that demonstrate how to use the `PGPVM2` library. Before the examples can be correctly compiled, please make sure of the following:

① You are in the `EXAMPLE/C` subdirectory of `pgpvm2` and the `$PVM_ROOT` has been set accordingly and that the standard `PVM aimk` exists, and can be found from your `PATH`!

② The `Fortran` examples assume that you have placed `pgpvm2` as a subdirectory of `pvm3`. If this is not true, you must edit the includes in `master1.f` and `slave1.f`. Please note there are two includes in the `slave1.f` example.

③ The `C` examples should work fine no matter where you place the directory as long as ① above is satisfied.

4 Using PGPVM2 with C Programs

Before `PGPVM2` can be utilized, you must compile it. Please refer to the `Readme` file.

Using `PGPVM2` requires only three minor modifications to a `PVM` application. First, the application must include a new header file, "`pgpvm2.h`". This should go directly under the standard "`pvm3.h`" header file. The new header file provides macros that replace normal `PVM` routines with calls to the `PGPVM2` library. Calls to the `PVM` library in the application source code need not be modified. The source does however need to be recompiled.

The second modification is the addition of the `pg_startadmin(char *outfile, char *host, int nbt_admin_max)` library routine in the first spawned task. The name `outfile` corresponds to the output trace file name, which is chosen by default (when the `outfile` name is `""`) to be `pgfile.<uid>` in the directory `/tmp`. The `host` parameter specifies where the `pg_administrator` program must be spawned. One can set `host` to the empty string `""` to let `PVM` choose where to spawn the `pg_administrator` program but in this case, one has to know where it has been spawned in order to get the `outfile` on the disk of the corresponding computer. The integer `nbt_admin_max` gives the number of simultaneous spawned tasks which want to be administrated by the `pg_administrator`. This option gives a way to simulate (at the level of `Paragraph`) a parallel virtual machine having a fixed number of processors. A value of `nbt_admin_max` equal to `0` means that there is no limitation on the number of simultaneous spawned tasks producing trace information.

The `pg_startadmin` routine must be executed before any `pg_tids` call, but since the `pg_startadmin` call should be in the first spawned task, it simply means that the function call to `pg_startadmin` should be before the `pg_tids` call in the first spawned task. This brings us to the last modification needed to use `PGPVM2`, that is the addition of the `pg_tids(char *option)` in which the `option` string specifies whether the task will produce trace data or not. This is done by setting the parameter `option` to either `"Y"` or `"y"` to mean "yes", and either `"N"` or `"n"` otherwise. Note that a call to the `pg_tids` function with either "Y" or "N" must be performed before any PVM function call, or more precisely before any PVM function involved in the tracing process, such as pvm_send, pvm_recv, and so on.

The exact behavior of the first call to the pg_tids function is to declare a new task to the administrator. Moreover, this first call informs the administrator whether the task is or not traced from now on. If the task wants its trace to stop, it must call pg_close(). Then, if it wants to produce again some trace data, it must call pg_tids("Y").

In the first release of PGPVM, the pg_tids(int *tid, int nprocs) library routine had a completely different behavior. The array of tid's (nprocs denoted the number of elements in the array) informed PGPVM of which processes should produce tracing information. All processes that wished to participate in tracing had to call pg_tids() and all had to pass the same tids array all other processes passed. In PGPVM2, this is more flexible.

To illustrate the use of PGPVM2, let us consider a simple example. The following program corresponds to the source father.c in the EXAMPLE/C subdirectory of pgpvm2. It starts with the first modification discussed previously, that is the header file pgpvm2.h included just after pvm3.h:

```
#include <stdio.h>
#include <malloc.h>
#include "pvm3.h"                       /* includes pvm3.h */
#include "pgpvm2.h"                     /* includes pgpvm2.h after pvm3.h */
```

Then, it continues with several macros and since it is the first spawned task, it contains the function call to pg_startadmin in which we have specified that the maximal number of tasks simultaneously producing trace information will be NBSONS1+NBSONS2+1. Then, the program specifies that it wants also to produce trace information by running pg_tids("y"):

```
#define SONS1           "son1"
#define SONS2           "son2"
#define NBSONS1         5               /* nb son1 tasks */
#define NBSONS2         3               /* nb son2 tasks */

void main()
{ int i;                                /* variable for loop */
  int mytid;                            /* my task id */
  int nbs=NBSONS1+NBSONS2;              /* total number of sons */
  int *tids;                            /* tids array */
  int info;                             /* pvm info */

  pg_startadmin("", "", NBSONS1+NBSONS2+1);
  pg_tids("y");
```

The PGPVM2 library provides two extra functions pg_beglab(int label) and pg_endlab(int label) that allow a program to associate an initeger label (at the level of Paragraph) for a section of its source. The pg_beglab begins the section while the pg_endlab ends it. This is the case in the next lines of father.c:

```
  pg_beglab(1);

  tids=(int *)malloc(sizeof(int)*(nbs+1));
  assert(tids!=NULL);
  tids[0]=mytid=pvm_mytid();

  info=pvm_spawn(SONS1, (char **)0,
                 PvmTaskDefault, "", NBSONS1, tids+1);
  info=pvm_spawn(SONS2, (char **)0,
                 PvmTaskDefault, "", NBSONS2, tids+1+NBSONS1);

  pvm_initsend(PvmDataDefault);
  pvm_pkint(&nbs, 1, 1);
  pvm_pkint(tids, nbs+1, 1);
  for (i=1;i<nbs+1;i++) pvm_send(tids[i],1);

  for (i=1;i<nbs+1;i++) pvm_recv(-1,-1);

  sleep(1);

  pg_endlab(1);
```

This can be useful to count the number of tasks simultaneously executing a certain part of their code, as given by the submenu `Count` of the menu `Tasks` of `Paragraph`. Coming back to `father.c`, the program ends by spawning again several tasks:

```
  info=pvm_spawn(SONS1, (char **)0,
                 PvmTaskDefault, "", NBSONS1, tids+1);
  info=pvm_spawn(SONS2, (char **)0,
                 PvmTaskDefault, "", NBSONS2, tids+1+NBSONS1);

  pvm_initsend(PvmDataDefault);
  pvm_pkint(&nbs, 1, 1);
  pvm_pkint(tids, nbs+1, 1);
  for (i=1;i<nbs+1;i++) pvm_send(tids[i],1);

  for (i=1;i<nbs+1;i++) pvm_recv(-1,-1);

  free(tids);
  pvm_exit();
  exit(0);
}
```

Now, we provide two other `PVM` programs which are spawned by the `pvm_spawn` commands of `father.c`. Here is the first one, named `son1.c`:

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

```
#include <malloc.h>
#include <assert.h>
#include "pvm3.h"
#include "pgpvm2.h"

void main()
{ int mytid;                            /* task identifier */
  int *tids;                            /* tids array */
  int nbs;                              /* number total of sons */

  pg_tids("y");

  pg_beglab(2);

  pvm_recv(-1, 1);
  pvm_upkint(&nbs, 1, 1);
  tids=(int*)malloc(sizeof(int)*(nbs+1));
  assert(tids!=NULL);
  pvm_upkint(tids, nbs+1, 1);

  pvm_initsend(PvmDataDefault);
  pvm_send(tids[0], 1);

  pg_endlab(2);

  free(tids);
  pvm_exit();
  exit(0);
}
```

Note that it involves a section labeled by 2. Now, the second task is called `son2.c` and its source involves the two labels 3 and 4:

```
#include <stdio.h>
#include <malloc.h>
#include <assert.h>
#include "pvm3.h"
#include "pgpvm2.h"

void main()
{ int *tids;                            /* tids array */
  int nbs;                              /* number total of sons */

  pg_tids("y");

  pg_beglab(3);

  pvm_recv(-1, 1);
```

```
  pvm_upkint(&nbs, 1, 1);
  tids=(int*)malloc(sizeof(int)*(nbs+1));
  assert(tids!=NULL);
  pvm_upkint(tids, nbs+1, 1);

  pg_endlab(3);

  sleep(3);

  pg_beglab(4);

  pvm_initsend(PvmDataDefault);
  pvm_send(tids[0], 1);

  pg_endlab(4);

  free(tids);
  pvm_exit();
  exit(0);
}
```

Once the application has been modified as described and recompiled by the following command (in the `EXAMPLE/C` subdirectory):

```
$ aimk father son1 son2
making in HPPA/ for HPPA
   cc -I/usr/local/lib/pvm3/include -DUSE_PGTRACE -o
      /users/pvm/bin/HPPA/father ../father.c -L/usr/local/lib/pvm3/lib/HPPA
      -lpvm3 -lpgpvm2
   strip /users/pvm/bin/HPPA/father
   cc -I/usr/local/lib/pvm3/include -DUSE_PGTRACE -o
      /users/pvm/bin/HPPA/son1 ../son1.c -L/usr/local/lib/pvm3/lib/HPPA
      -lpvm3 -lpgpvm2
   strip /users/pvm/bin/HPPA/son1
   cc -I/usr/local/lib/pvm3/include -DUSE_PGTRACE -o
      /users/pvm/bin/HPPA/son2 ../son2.c -L/usr/local/lib/pvm3/lib/HPPA
      -lpvm3 -lpgpvm2
   strip /users/pvm/bin/HPPA/son2
```

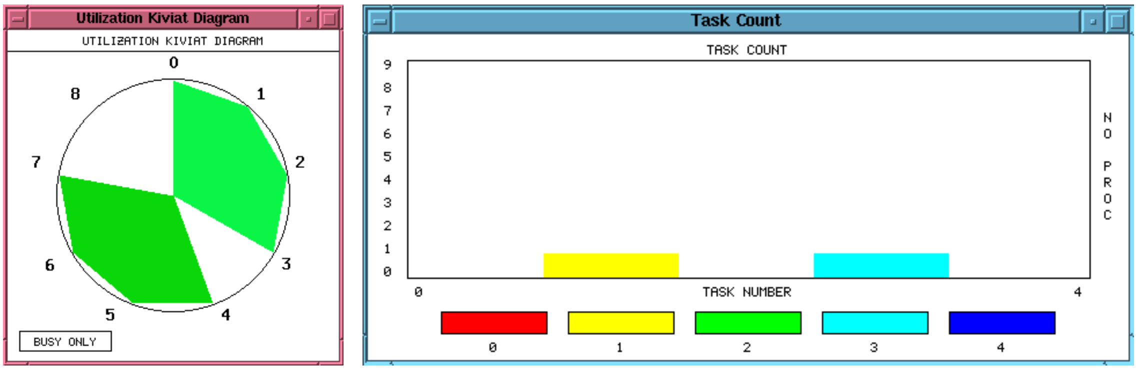
upon execution `PGPVM2` produces a single tracefile named `pgfile.<uid>` in the `/tmp` directory. This tracefile can be found on the host machine where `pg_startadmin` has started the `pg_administrator` program. To convert the output file into a file readable by `Paragraph`, the user should run `PGSORT` on `/tmp/pgfile.<uid>`, as illustrated by the following lines:

```
$ PGSORT /tmp/pgfile.516
Working... initial sort
Working... clocksync
```

```
Working... 2nd sort
Working... converter
Working... final sort
```

This shellscript converts the file to standard `Paragraph` trace file format and the trace file will appear in the directory as `pgfile.<uid>.trf`, that is `pgfile.516.trf` in the previous example. An advanced feature described previously allows users to dictate the pathname and filename of where the tracefile is created, and many find this more convenient than retrieving the file from `/tmp`.

Once the file `pgfile.516.trf` has been produced, the user may run `Paragraph` by the command `PG pgfile.516.trf` in order to visualize the behavior of the program:



5 Using PGPVM2 with Fortran Programs

Before `PGPVM2` can be utilized, you must compile it. Please refer to the `Readme` file.

Using `PGPVM2` with `PVM Fortran` programs is only slightly more tedious. Although the user does not add an extra header file nor a `#define USE_PGTRACE` as is the case with `C` programs, the user must modify the names of certain `PVM` routines. For example, the user must change all calls from `pvmfmytid()` to `pgfmytid()`. The next figure lists the mandatory name modifications for `PVM Fortran` applications:

Old Routine Name	New Routine Name
pvmfmytid()	pgfmytid()
pvmfexit()	pgfexit()
pvmfrecv()	pgfrecv()
pvmfprecv()	pgfprecv()
pvmnrecv()	pgfnrecv()
pvmtrecv()	pgftrecv()
pvmfmcast()	pgfmcast()
pvmfsend()	pgfsend()
pvmfpsend()	pgfpsend()
pvmfspawn()	pgfspawn()

The only other modifications are the addition to the source code of the `pgfstartadmin()`, `pgftids()` `pgfbeglab()`, `pgfendlab()` and `pgfclose()` library routines to the source code.

Please, refer to the section dealing with the utilization of PGPVM2 with C programs for more details about these functions, respectively called in C programs pg_startadmin(), pg_tids(), pg_beglab(), pg_endlab() and pg_close().

In particular, one can use labels to identify specific parts of your program by using the pgfbeglab and pgfendlab functions. For instance, to start a section labelled by 1, one has to insert the following line:

```
call pgfbeglab(1, info)
```

where info is an integer, which is set to a value different from PvmOk if the label is already in use. In the same way, use:

```
call pgfendlab(1, info)
```

to end the labelled section.

Furthermore, the internal C function pg_veriflab() is replaced by pgfveriflab() in Fortran programs. Refer to the previous C example or to the Fortran examples in the subdirectory EXAMPLE/FORTRAN.

Note that all strings in the Fortran routines must be explicitly null (\0) terminated. Where one uses the empty string "" as a parameter in C PGPVM2 functions, one should use a "*" with a null termination in the associated Fortran routines. For example, from master1.f, we use:

```
call pgfstartadmin('BRAD\0', '*\0', nproc+1)
```

or:

```
call pgfstartadmin('*\0', '*\0', nproc+1)
```

Once the application has been modified as described and recompiled, upon execution PGPVM2 produces a single tracefile named pgfile.<uid> in the /tmp directory. This tracefile can be found on the host machine where the first task has spawned the pg_administrator program. Then, as in the case of a C program, running PGSORT pgfile.<uid> converts the file to standard Paragraph trace file format and the trace file will appear in the directory as pgfile.<uid>.trf. An advanced feature described previously allows users to dictate the pathname and filename of where the tracefile is created, and many find this more convenient than retrieving the file from /tmp.

5.1 PGPVM2 – Fortran Compilation Details

As previously touched upon, application programs must be recompiled and linked to the PGPVM2 library, libpgpvm2.a. This is in addition to linking to the standard PVM libraries that Fortran programs link to. Please refer to the Makefile.aimk in the EXAMPLE/FORTRAN subdirectory of PGPVM2 for proper compilation techniques.

When an application has no need for PGPVM2's tracing capabilities, simply comment out both pgfstartadmin() and pgftids(), and recompile. The other modifications do not have to be removed and the application will execute as if PGPVM2 routines had not been linked to the application.

6 PGPVM2 – Advanced Features

The first version of PGPVM had three advanced features, pg_outfile() (pgfoutfile() for Fortran), pg_close() (pgfclose() for Fortran) and pg_chprefix() (pgfchprefix()).

Currently, in the new release PGPVM2, pg_chprefix() and pg_outfile() had been removed. However, it is still possible to specify the output file name by calling the function pg_startadmin() (pgfstartadmin()).

The only advanced feature is the routine pg_close() (pgfclose() for Fortran). The pg_close() routine allows the production of trace events to terminate before pvm_exit() is called. If this call is not used, the process produces trace events up until pvm_exit(). This necessarily implies that all nodes that participate in tracing need to eventually call pg_close() or pvm_exit(). After having stopped the production of trace events by a pg_close(), nodes can produce again trace information by calling pg_tids("Y"), as previously explained.

7 PGPVM2 – Another C Example

Now that you know how to use PGPVM2, let us come to a "real" example in which we want to distribute a *quick-sort* on a parallel virtual machine. This example, which is also included in the subdirectory EXAMPLE/C, demonstrates how the visualization of an execution can be useful to analyze a parallel program.

We consider a distributed sort in which each *slave* has in charge to sort one part of an array of integers, while the *master* that runs slaves has to merge the sorted arrays at the end of the computation. Here is first the source of the slave named dis_qsort_slave.c:

```
#include <stdlib.h>
#include <assert.h>
#include "pvm3.h"
#include "pgpvm2.h"

#define TAG_UNSORTED                    1
#define TAG_SORTED                      2

int increase(a, b) char *a; char *b;
{ if (*a==*b) return 0;
  if (*a>*b) return 1;
  return -1;
}
```

```
void main()
{ char *data;
  int mytid, buffer_id, bytes, typetag, nb, myrank, tid;

  mytid=pvm_mytid();

  pg_tids("y");

  pg_beglab(4);
  buffer_id = pvm_recv(-1, -1);
  pvm_bufinfo(buffer_id, &bytes, &typetag, &tid);
  pvm_upkint(&myrank, 1, 1);
  pvm_upkint(&nb, 1, 1);
  data=(char *)malloc(sizeof(char)*nb);
  assert(data!=NULL);
  pvm_upkbyte(data, nb, 1);
  pg_endlab(4);

  pg_beglab(5);
  qsort((void *)data, nb, sizeof(char),
        (int (*)(const void *, const void *))increase);
  pg_endlab(5);

  pvm_initsend(PvmDataRaw);
  pvm_pkint(&myrank, 1, 1);
  pvm_pkbyte(data, nb, 1);
  pvm_send(tid, TAG_SORTED);
  free(data);
  pvm_exit();
  exit(0);
}
```

Slaves use label 4 when they are receiving data, and label 5 when they are sorting integers. The master (dis_qsort_master.c) initiates the computation by creating the slaves as illustrated by the following lines:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "pvm3.h"
#include "pgpvm2.h"

#define TAG_UNSORTED                    1
#define TAG_SORTED                      2

#define NB_TASKS_PER_PROCESSOR          1
#define MAX_NB_TASKS                    1024
```

```
#define QSORT_TASK_NAME                    "dis_qsort_slave"
#define INPUT_FACTOR                       1000

void main(argc, argv) int argc; char *argv[];
{ int i, j, rank, nb, nbdata, minipos;
  char *data, *sorteddata, minival;
  int mytid, tids[MAX_NB_TASKS], nbtids, route;
  int startsubarray[MAX_NB_TASKS], indicessubarray[MAX_NB_TASKS],
      maxpossubarray[MAX_NB_TASKS];
  int nhost, narch, pvm_info;
  struct pvmhostinfo *hostp;

  assert(argc==2 || argc==3);
  nb=atoi(argv[1])*INPUT_FACTOR;

  data=(char *)malloc(sizeof(char)*nb);
  assert(data!=NULL);
  sorteddata=(char *)malloc(sizeof(char)*nb);
  assert(sorteddata!=NULL);
  mytid=pvm_mytid();
  route=pvm_setopt(PvmRoute, PvmRouteDirect);
  pvm_info=pvm_config(&nhost, &narch, &hostp);
  assert(pvm_info>=0);
  if (argc==3) nbtids=atoi(argv[2]);
  else nbtids=NB_TASKS_PER_PROCESSOR*nhost;
  assert(nbtids<=MAX_NB_TASKS);

  /* random initialization of the array to sort */
  for (i=0;i<nb;i++) data[i]=(char)rand();

  pg_startadmin("", "", nbtids+1);
  pg_tids("y");
  for (i=0;i<nhost-1;++i)
  { if (nbtids/nhost)
      pvm_spawn(QSORT_TASK_NAME, (char **)NULL, PvmTaskHost,
                hostp[i].hi_name, nbtids/nhost,
                tids+i*(nbtids/nhost));
  }
  pvm_spawn(QSORT_TASK_NAME, (char **)NULL, PvmTaskHost,
            hostp[i].hi_name,
            (nbtids/nhost)+((nbtids%nhost)?(nbtids%nhost):0),
            tids+i*(nbtids/nhost));

  pg_beglab(1);
  nbdata=nb/nbtids;
  for (i=0;i<nb%nbtids;++i)
  { startsubarray[i]=indicessubarray[i]=i*(nbdata+1);
    maxpossubarray[i]=startsubarray[i]+nbdata;
```

```
}
for (j=0;j<nbtids-nb%nbtids;++j)
{ startsubarray[i+j]=(nb%nbtids)*(nbdata+1)+j*(nbdata);
  indicessubarray[i+j]=startsubarray[i+j];
  maxpossubarray[i+j]=startsubarray[i+j]+nbdata-1;
}
for (i=0;i<nbtids;++i)
{ pvm_initsend(PvmDataRaw);
  pvm_pkint(&i, 1, 1);
  nbdata=maxpossubarray[i]-startsubarray[i]+1;
  pvm_pkint(&nbdata, 1, 1);
  printf("[%d]", nbdata);
  pvm_pkbyte(data+startsubarray[i], nbdata, 1);
  pvm_send(tids[i], TAG_UNSORTED);
}
printf(" distributed...\n"); fflush(stdout);
pg_endlab(1);
```

Then, the master waits for sorted arrays and ends the computation by merging all these arrays in order to produce a global sorted array:

```
/* receiving sorted arrays */
pg_beglab(2);
for (i=0;i<nbtids;++i)
{ pvm_recv(-1, TAG_SORTED);
  pvm_upkint(&rank, 1, 1);
  pvm_upkbyte(data+startsubarray[rank],
              maxpossubarray[rank]-startsubarray[rank]+1, 1);
}
pg_endlab(2);

/* merging sorted arrays */
pg_beglab(3);
for (i=0;i<nb;++i)
{ for (minipos=0;minipos<nbtids;++minipos)
    if (indicessubarray[minipos]<=maxpossubarray[minipos]) break;
  minival=data[indicessubarray[minipos]];
  for (j=0;j<nbtids;++j)
    if (indicessubarray[j] <= maxpossubarray[j] &&
        data[indicessubarray[j]]<minival)
    { minival=data[indicessubarray[j]];
      minipos=j;
    }
  sorteddata[i]=minival;
  indicessubarray[minipos]++;
}
pg_endlab(3);
```

```
  free(data);
  free(sorteddata);
  pvm_exit();
  exit(0);
}
```

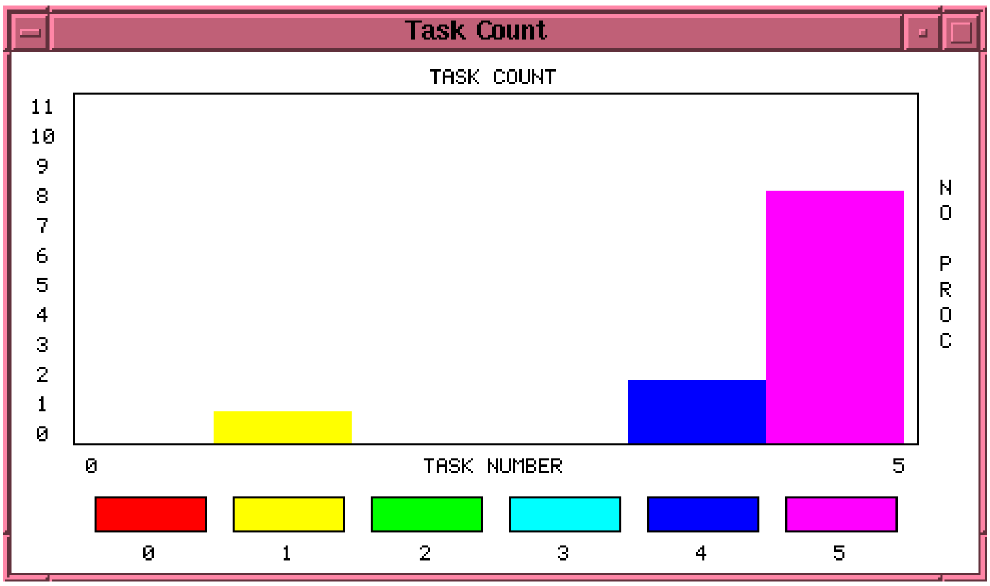
Now, for instance, suppose we want to know whether the partial `qsort()` function calls are really performed in parallel on the virtual machine, we should compile these programs and link them to the `PGPVM2` library with the `aimk` command. We then run the master program with the following command to mean that we want to sort 5000000 integers using 10 tasks:

```
$ dis_qsort_master 5000 10
pg_tids: initialization completed
[500000][500000][500000][500000][500000][500000][500000]
[500000][500000][500000] distributed...
pg_exit: calling pvm_exit
```

and call `PGSORT`:

```
$ PGSORT /tmp/pgfile.516
Working... initial sort
Working... clocksync
Working... 2nd sort
Working... converter
Working... final sort
```

Now, we use `Paragraph` to answer to the previous question about the number of concurrent `qsort()` calls performed on blocs of 500000 integers:



This figure confirms that almost all slaves are performing their quick sort in parallel, because it shows that 8 processes are simultaneously executing their section labelled 5, *i.e.* the `qsort()` part of the program.

8 Obtaining Paragraph and PGPVM2

`Paragraph`, a performance visualization tool created by Michael T. Heath and Jennifer E. Finger, is available free over the internet. For instructions on how to obtain `Paragraph`, simply send an electronic mail message to `netlib@ornl.gov` containing the text, "`send index from paragraph`".

`PGPVM2` can be obtained by anonymous `Ftp` from `ftp://phalanstere.univ-mlv.fr/` in the subdirectory `/pub/lang/pvm/PGPVM2`.

9 Comments or Questions

Please send comments and questions regarding either the first or the second release of `PGPVM` to Brad Topol (`topol@cc.gatech.edu`), Vaidy Sunderam (`vss@mathcs.emory.edu`) or Sébastien Veigneau (`Sebastien.Veigneau@univ-mlv.fr`).