The mapping tool running distributed under PVM

L. Hluchý, D.Dobrovodský, M. Dobrucký

[*] Institute of Computer Systems, Slovak Academy of Sciences
Dúbravská cesta 9, 842 37 Bratislava, SLOVAKIA
e-mail: upsyhluc@savba.sk

Description of the mapping tool's current state

The mapping tool we have developed is up to now implemented under the UNIX operating system as distributed parallel program running under PVM (Parallel Virtual Machine ver.3.3). It can be run in the background, because all the inputs are supplied via command-line parameters. It uses the SW-graph file as the input for the description of the parallel application to be mapped and creates an output file in which the result (the location vector) is stored.

The communication smoothing tool is implemented as separate tool, because the multiprocessor communication routing policy (configuring) is hardware dependent (supplied by a vendor).

The dynamic load-balancing processing is under development and can be implemented in the MPVM environment, that is partially architecture dependent, i.e. task moving is allowed only between the hosts with the same architecture.

Usage rules of the mapping tool

Current implementation of the mapping tool is as follows. Being started the mapping tool detects number of hosts running in PVM environment, i.e. gets info about current parallel hardware. User can supply the inputs as command-line parameters : SW-graph file name, HW-graph file name, chosen mapping method and its parameters. If there is no parameter, help on the usage is printed (see Fig. 1.).

```
usage: pmap SW_graph_data_file [HW_graph_data_file [mode [comm [iter]]]]
    or: pmap  - [HW_graph_data_file] ...for obtaining the HW_data from PVM

HW_graph_data_file = file_name OR "-" as working PVM
mode =  quick: 0,1,2,3,4 (greedy)
           smart: diffusion: D, Boillat: B, diffusion_fixed: F
           heavy: heuristic_move_exchange: H
           all: A
comm =  communication_weight_magnifier
    iter =  iteration_loop_divisor
```

Fig. 1: The help on the mapping tool usage

---

[*] SEPP progress report appendix - Task 4 : Mapping and Load Balancing

SW-graph describes the application parallel program that should be mapped. If it is not specified, our tool can measure the interconnection throughputs between the PVM-hosts and store obtained info into the HW-graph file for the further usage. Its default name is "hw.out" and can be specified as the second command-line parameter (the first should be in this case the dash sign "-").

HW-graph describes the target where an application parallel program have to be mapped. If it is missing (or is there dash sign instead), the current presented PVM will be the target and the measurement of the interconnection throughputs between the PVM-hosts takes place (it may take some time : because it can be time-dependent, this measurement should be done in the larger period of time or in the supposed run-time-of-the-day, i.e. in the other day time.)

As third command-line parameter can be the chosen mapping method. It can be one of the {0,1,2,3,4,D,B,F,H,A} character (see Fig.1).

Via the next two parameters the user can tune the Boillat or Diffusion methods.

The status parameters and results are printed on the screen and are also written into the output file, which name is derived from the input file names by their concatenation (with ".") and adding the ".out" suffix. There are followed items there :

- SW_graph, HW_graph - the input file names
- #hw_nodes - the number of detected PVM hosts or the size of the HW-graph
- #nghbr - the maximum of the processor neighbours (estimated from the measured interconnection throughput values) or diameter of the HW graph (for transputer-like multiprocessors)
- "GREEDY","SMART","HEAVY" - the chosen method (8 methods including Diffusion, Boillat, HME)
- #tasks - the number of parallel tasks to be mapped (SW-graph size)
- #channels - the number of communication channels between parallel tasks

Our mapping tool can animate the progress of the mapping process (methods D,B,F) in the "xterm" - type window. Then it writes the followed output values to the file and screen:

- Imb - the imbalancing percentage = the quality of the solution
- Comm - the communication cost = the quality of the solution
- Time - the time spent on the mapping process
- Node Location - the mapping output vector (is written only to the file)

Our tool can run distributed in the PVM environment (actually 3 iterative methods D,B,F). It supplies mapping methods, that are grouped into 3 groups according their quality and also time consumption in the **quick** (fast simple greedy methods suitable for large tasks), **smart** (our diffusion method, Boillat method and method derived from our diffusion method, all suitable for medium size tasks) and **heavy** ("heuristic move-exchange", more time-consuming but with the best achieved quality suitable for small size tasks). The user is allowed to tune some parameters for these methods via command-line parameters (number of iteration, communication weight magnifier etc.).

Current implemented methods of mapping tool under UNIX OS are as follows :

5 greedy : 2x Random, to most Free, Largest first to most free, Fit largest first to free&near;
3 smart : our Diffusion, Boillat, Fixed (modificated diffusion method)
1 heavy : Heuristic Move Exchange

Another heavy mapping method - Simulated Annealing is implemented as the separate program and will be integrated into our mapping tool later, likewise smoothing tool as well.

There is in the mapping tool a possibility to enlarge the number of identical copies of the mapping tool in the PVM environment via adding redundant mapping-processes to fit the desired HW-graph size and thus run in parallel. This possibility is automatically used if the number of added processes would be small (not exceeding the user defined process-wasting limit per host).

How it works in parallel

The mapping tool is written in C language under UNIX and using PVM library as one-file program. Having started it can spawn itself into several identical copies across all the current PVM-host processors. Only the main process (the master) manages the input and output data (stored in files). Other copies work as co-processors and communicate with the master and also among themselves.

In the first part, our tool obtains the PVM-host number ("nproc"). If there are more than 1 PVM-host there and the HW-graph info is not read, the measurement of the interconnection throughputs will start. The master process spawn "nproc-1" copies, one per each PVM-host. Then each of the "nproc" process spawns its twin on the same host and communication between this twins will be measured and stored. Next, the communication among all the first started "nproc" processes (without twin) will be measured and the results will be sent to the master. If the measurement is done, the master will read the input command-line parameters and will decide whether the sequential mapping method is to be performed. If this is the case, all the other processes will stop and the master will do the mapping alone.

If the parallel mapping method is chosen, the master will arrange the next work as follows. If the number of parallel copies (2*nproc) is not enough (less than the HW-graph size), some copies will be spawned across the PVM to reach the equivalence, i.e. each copy will work for desired HW-graph vertex (target processor). Then the master sets the initial status (location vector) and spreads input data. The unnecessary processes (if 2*nproc > HW_graph_size) will be self-terminated and thus they will not participate on the mapping work.

The 3 smart mapping methods work iteratively. After each iteration is completed, the computed data are interchanged among all the participants. This implementation is synchronized, because the information about task location need to be up-to-date in all the participants. After the last iteration is done, all the result data are present at the master and need not be colected. This parallel procedure is described on the Fig.2.

The "simulated annealing" mapping method use another method: processing pool. The more parallel processes the better is quality of the mapping (the search space is well reconsidered). After certain temperatures the partial results are interchanged and the best solving-status will be chosen.

This parallel procedure is described on the Fig.3.

```
for (ii = 1; ii <= iter; ii++)                /* iterative method */
{ tsk = Compute( ii, &target );                /* choose task and new location for it */
  SynchData( tsk, target );                    /* partial results exchange */
  Animate( ii );                               /* animation of the mapping progress */
}

SynchData(tsk,tgt) int tsk,tgt;
{int j;
pvm_initsend( PvmDataDefault );                /* create/initialize send buffer */
pvm_pkint( &tsk, 1,1 );  pvm_pkint( &tgt, 1,1 );       /* data packing */
pvm_bcast( GROUP, MSGTAG );                    /* broadcast to all but me */
for(j=1;j<nproc;j++)                           /* receive from all but me */
 {pvm_recv( -1, MSGTAG );                      /* source undetermined */
  pvm_upkint( &tsk, 1,1 ); pvm_upkint( &tgt, 1,1);   /* data unpacking */
  if (tsk >= 0) Move(tsk,tgt);                 /* task location changing */
 }
}
```

Fig.2: The distributed iterative (diffusion, Boillat,...) mapping method

```
while (T > T_min)                     /* temperature is decreasing */
{
 obj_func[0] = Compute( &map[0] );/* compute new mapping vector */
 SynchData( map, obj_func );          /* partial results exchange */
 GeneticChange();                     /* mutate, exchange, cross-over - genetic change on the pool
                                         of partial results */
}

SynchData(map,obj) int map[][], obj[];
{int j;
pvm_initsend( PvmDataDefault );                /* create/initialize send buffer */
pvm_pkint(map[0], sizeof(map[0]),1);  pvm_pkint(obj[0], 1,1 );  /* data packing */
pvm_bcast( GROUP, MSGTAG );                    /* broadcast to all but me */
for(j=1;j<nproc;j++)                           /* receive from all but me */
 {pvm_recv( -1, MSGTAG );                      /* source undetermined */
  pvm_upkint( map[j], ); pvm_upkint( obj[j], 1,1);   /* data unpacking */
 }
}
```

Fig.3: The distributed simulated annealing with the genetic approach

```
#define LOOPS 20                        /* repeat transfer */
mytid = pvm_mytid();                    /* enroll in PVM */
me = pvm_joingroup(GROUP);              /* join the named group - get my instance number */
pvm_config(&nproc, &narch, &hi);        /* get configuration info */
myhost = Findhost(mytid);               /* get my host identifier */
if(me == 0) for (i=0; i<nproc; i++)     /* spawn accross all the PVM hosts */
     if (i != myhost) pvm_spawn(taskname, (char**)0, 1,hi[i].hi_name, 1,&j);
if(me < nproc)pvm_spawn(taskname, (char**)0, 1,hi[myhost].hi_name, 1,&child); /* spawn twin */
if(me < 2*nproc)pvm_barrier(GROUP, 2*nproc);     /* wait until all are spawned */
if(me ==  0) SpreadHgRd();                       /* master proces spreads "hgrd" status value */
else if(me < 2*nproc) GetHgRd();                 /* other (included twin) */
else {hgrd=FALSE; pvm_joingroup(GROUP1);}        /* adding process */
/* ------------------------------------------------------------------------------------------------- */
if(hgrd){                                        /* if the HW info has to measure */
for (j=0; j<LOOPS; j++) if (me < nproc) Measure(child); else Reply(pvm_parent());
if (me < nproc) for (j=0; j<LOOPS; j++) for (k=0; k<nproc; k++)    {       /* each with every */
                        pvm_barrier(GROUP, nproc);          /* synchronize */
                        if ( me == k ) for (i=0; i<nproc; i++)          /* each with every */
                                {if (i!=me) Measure(pvm_gettid(GROUP,i));}
                        else Reply(pvm_gettid(GROUP, k));               /* each */
                                                                    }
if ( me == 0 ) for (i=1; i<nproc; i++) Recv_row(pvm_gettid(GROUP,i));
else if (me < nproc) Send_row();                /* result collecting */
        }
```

Fig.4: The distributed initializing and interconnection throughput measurement

```
#define BLOCK 400        /* test message size */
#define GROUP "pmapgrp"
#define GROUP1 "pmap1grp"
#define MSGTAG 0
int mess[BLOCK];

Measure(dest) int dest;
{ int t, myhosttid, bufid; struct timeval start, stop, delta;
bufid=pvm_initsend( PvmDataDefault );
pvm_pkint( mess, BLOCK, 1 );
pvm_hostsync((myhosttid=pvm_tidtohost(mytid)),&start,&delta);
pvm_setsbuf(bufid); /* hostsync use rbuf and sbuf  ! */
pvm_send( dest, MSGTAG );               bufid=pvm_recv( dest, MSGTAG );
pvm_hostsync(myhosttid,&stop,&delta);
pvm_setrbuf(bufid); /* hostsync use rbuf and sbuf  ! */
pvm_upkint( mess, BLOCK, 1 );
t = stop.tv_sec - start.tv_sec; t *= 1000;      /*  s -> ms */
t += (stop.tv_usec - start.tv_usec)/1000;       /* us -> ms */
Hg[myhost][findhost(dest)] += t;
}

Reply(dest) int dest;
{
pvm_recv( dest, MSGTAG );
pvm_upkint( mess, BLOCK, 1 );
pvm_initsend( PvmDataDefault );
pvm_pkint( mess, BLOCK, 1 );       pvm_send( dest, MSGTAG );
}

Send_row()
{pvm_initsend( PvmDataDefault );pvm_pkint( Hg[myhost], nproc, 1 );
pvm_send( pvm_gettid(GROUP,0) , MSGTAG+1 );}

Recv_row(dest) int dest;
{pvm_recv( dest, MSGTAG+1 );pvm_upkint( Hg[findhost(dest)], nproc, 1 );}

int Findhost(tid) int tid;
{int i,j, t;
t=pvm_tidtohost(tid);
i=-1; for (j=0; j<nproc; j++) if (hi[j].hi_tid == t) i=j;
if (i<0) exit(2); return(i);
}
```

Fig.5: Subroutines for the distributed initializing and interconnection throughput measurement