Distributed Mapping Tool 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

Abstract: *This paper describes the mapping algorithm for distributed memory, parallel message passing systems using the objective function to evaluate the optimality of the mapping of a task graph onto a processor graph. Our optimisation method is compared with other ones, some of them issuing from artificial intelligence or operations research. In our experiments randomly generated tasks and processor graphs are used.*

Keywords: *static and dynamic mapping, multicomputer, scheduling, load balancing, smoothing.*

1 Introduction

Optimal planning of parallel program execution in distributed environment of a multicomputer on the basis of message passing solves the response speed. The theory of optimal mapping comes out from the assumption that the program execution time depends upon uniform load of the processors and upon interprocessor communication minimisation. In this paper, our attention is concentrated on the diffusion method for static mapping.

For static mapping a parallel program can be represented by a task graph $S(V,E)$, vertices $V=\{1,2...q\}$ represent the program tasks and edges E represent mutual communication. The topology of available technical resources can be represented by processor interconnection graph $H(C,E_p)$, vertices of which, $C=\{1,2,...p\}$, represent processors, and edges E_p represent communication channels or links. We assume the homogenous structure of the multicomputer system. The communication cost depends on message size and on distance between source and destination. Interprocessor distance d_{ij} between processors i and j is given by routing rules, i.e. as minimal number of hops from processor i into the processor j. Clearly, if $d_{ij}=0$ then $i=j$. Hardware graph detection includes methods for multiprocessor system inspecting and diagnostics (seeking connections and node parameters in the structure). Many optimisation methods have been created to solve the mapping problem, [1 - 9]. Their basic principles are common with the method used in other areas of research.

2 The Cost (Objective) Function Definition

To specify an appropriate optimisation goal it is necessary to create a cost function, which provides a realistic evaluation of the communication and computation overhead. For the given graphs S and H, mapping M, and routing RA, the general form of the cost function CF can be built as a combination of three parts - so called **vertex cost function** F_{vertex} , **edge cost function** F_{edge} and the **routing cost function** F_{rout} :

$$CF(M,t) = F_{vertex}(M,t) + F_{edge}(M,t) + F_{rout}(M,t,RA)$$

[#] *Funded by Commission of the European Communities DG12 under contract number CIPA-CT93-0251 and Grant Agency for Science of Slovak Academy of Sciences*

Variable *t* is duration of the iteration step of the mapping algorithm. *Fvertex(M,t)* expresses the effect of computation loads so that individual physical processors would be loaded in a uniform way. *Fedge(M,t)* expresses the effect of communication volume on mapping so that mapping should ensure the minimum of external (interprocessor) communications. *Frout(M,t,RA)* expresses the effect of imbalance of physical communication links. Our contribution describes the solution of two phase cost function optimisation: load balancing with communication minimisation and then traffic load balancing.

3 The Diffusion Method for Static Mapping

The principle of our mapping approach can be represented by the following procedure [10]. In the starting condition all tasks are located at the root node of the multicomputer network. The tasks are transferred by centrifugal force *f1* (following from the requirement of uniform processor load), against which centripetal force *f2* is acting, trying to keep the processes with mutual communication as close to each other as possible. This way, uniform load is obtained and this method can be considered as an improvement of the "pure" load balancing method. The resulting sum of forces is a vector with components for each task stored in a node and directed to each communication link. A task and a direction with maximum (positive) value is chosen. The following relation is the definition of changes in the cost function, where $\alpha(t)$=const. and function *g(t)* hides the weight coefficient:

$$\Delta CF(M) = \alpha(t) \ . f1(t) + g(t).f2(t)$$

where *t* is the duration of iteration step. Individual forces $f1_{ijk}$ and $f2_{ijk}$ are calculated in each iteration for each node *i* from set of nodes *C* and for each possible transfer direction (into node *j*) of each task *k* from set of tasks *V* located on it according to the following relations :

$$\forall i, j \in C \mid j = neighbour(i) : \quad a_{ij} = \frac{load(i) - load(j)}{links(i)+1} + eps \qquad load(x) = \sum_{k \mid M(k)=x} w_k$$

$$\forall k \in V \mid M(k) = i : \qquad f_{ijk}(t) = \alpha(t) \ . f1_{ijk}(t) + g(t).f2_{ijk}(t)$$

$$f1_{ijk} = \begin{cases} w_k & \text{. . . . . if } w_k \le a_{ij} \\ \dfrac{(links(i)+1).a_{ij} - w_k}{links(i)} & \text{. . . else} \end{cases}$$

$$f2_{ijk} = \sum_{\forall m \in V \wedge M(k)=i} CC_{km} - \sum_{\forall m \in V \wedge M^*(k)=j} CC^*_{km}$$

w_k represents the computation weight of task *k*, *load(x)* is the total weight of all tasks allocated on the processor *x* (M*(k)*=*x*), *links*+1 means the number of those interested in load balancing (the number of neighbours plus the node itself), *eps* is a random-generated variable with values 1 and 0 whose use will allow a change at equal conditions as well. *CC* - the communication cost of a channel - is given by the product of the channel cost and of the cost for the distance of communicating processes (*CC*=0 if no communication). *CC** is the communication cost if the task *k* were moved, the difference of *CC* and *CC** gives the communication improvement (similarly *M** is the mapping if the task *k* were moved).

The total number of iterations must be sufficient for location stabilisation. This number is obtained from the experiments and is a trade-off between the time consumption and the result accuracy. We estimate this number from q - the number of tasks to be mapped. Standard- nominal value was estimated as $7q/2$.

The function *g(t)* that we use was obtained from programming experiments in the transputer environment for some applications. Our experiments aimed at looking for suitable face of function *g(t)* (linear, nonlinear, increasing and decreasing intervals,...) and the number of iterations. The function *g(t)* is suggested to have the following properties: During the first *n* iterations it can be kept near zero or increases slowly from zero. The reason is that all tasks are located at the same processor at the start of mapping, and there is a need to keep the communication force low to enable task distribution. In the *n*-th iteration step *g(t)* reaches the value where the influence of *f1* and *f2* is comparable. Then *g(t)* decreases to zero, and thus the influence of communications becomes weaker. The algorithm can be implemented in the distributed environment as well.

4 The Traffic Load Balancing

For the traffic load balancing (smoothing) as the second phase of cost function optimisation we used simulated annealing. The algorithm consists of the following steps:

1. Creation of the shortest path table on the basis of the algorithm for the shortest paths. We used the matrix algorithm [11]. The input for this algorithm are hardware and software graphs.
2. Creation of the routing table which is at the beginning initialised to the shortest path table.
3. There is a randomly chosen path and randomly chosen processor on it and then is randomly chosen neighbouring processor and this new state is evaluated under rule of simulated annealing. Cost function is represented by relation:

$$Frout(M, t, RA) = \gamma(t).\sum_{li=1}^{T} \left| AD_{li} - \overline{AD} \right|$$

AD_{li} means the amount of data transferred by link *li*, $\overline{AD}$ stands for average amount of data per link. The *RA* variable expresses the dependence of *Frout* upon the directing algorithm selected specifying the message delivery from the source to destination. The change following from this evaluation is written down in the routing table.

5 The Implementation in PVM

The mapping tool we have developed up to now is 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. Our tool supports also various other mapping methods (simulated annealing [1], Boillat [4], heuristic move exchange [2] ... some of them have sequential version only).

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

The mapping tool is written in C language under UNIX and using PVM library. Having started it spawns itself into several identical copies across all the current PVM-host processors (see Fig.1.). 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.

```
#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.1: The distributed initialising and interconnection throughput measurement

In the first part, our tool obtains the PVM-host number ("*nproc*"). If there are more than one PVM-host and the HW-graph info is not present, the measurement of the interconnection throughputs will start. The master process spawns "*nproc-1*" copies, one per each PVM-host. Then each of the "*nproc*" process spawns its twin on the same host and communication between these 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 it 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 new 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.

```
#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.2: Subroutines for the distributed initialising and interconnection throughput measurement

The Diffusion (and some others) mapping method work iteratively. After each iteration is completed, the computed data are interchanged among all the participants. This implementation is synchronised, 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 collected. This parallel procedure is described on the Fig.3.

The simulated annealing mapping method uses another parallelisation possibility: a processing pool. The more parallel processes the better quality of the mapping (the search space is well reconsidered). After certain states (temperatures) the partial results are

interchanged and the best partial solving will be chosen or the genetic algorithms (cross-over, mutation, exchange ...) take place. This parallel procedure is described on the Fig.4.

```
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.3: 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 */
 Decrease(T);                                   /*    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.4: The distributed simulated annealing with the genetic approach

6 Experimental results

We have compared (Fig.5 - 7) diffusion method with methods given in [4] - iteration method (Boillat), [2] - heuristic move-exchange (HME) and [1] - simulated annealing (SA). Model calculations were performed for nonregular graphs and grids. Graphs were generated by random graph generator. The software graph size was limited by 160 nodes (tasks), hardware graph size was limited by 64 nodes (processors), connected in transputer-like network (max. four neighbours per node). The cost for path length was given as follows: 0 as the cost for communication on the same processor, 1 as the cost for communication between direct neighbours and *2n* as the cost for communication across n-1 intermediators.

The augmented simulated annealing method was specified by T_{max} = 0.5, T_{min} = 0.05, *decrease* = 0.8. Number of moves for each temperature was *3pq*, where *p* is the number of processors and *q* is the number of tasks to be mapped.

The length of queue for HME was 3, the number of random moves was proportional to the number of mapped tasks (*2q*).

The number of iterations for Boillat and our diffusion methods was selected as the results of the experiments. The influence of the number of iterations on the quality of solution for these two algorithms was compared. Results obtained for Boillat algorithm showed that saturation of solution quality can be achieved when the number of iteration is equal nominal value $NOM = {}^{7q}/_{2}$, where *q* is number of tasks to be mapped. For diffusion algorithm we can observe this saturation when the number of iteration equals ${}^{NOM}/_{4}$, so to obtain the comparable results it is sufficient to choose the number of iteration ${}^{NOM}/_{4}$ or *q*.

Fig.8 shows the comparison of traffic load balancing; *SM* means smoothing, *SP* means routing on the shortest paths.

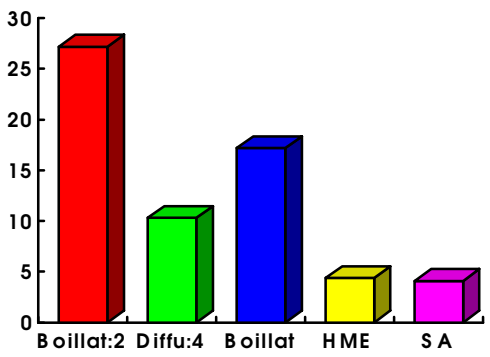

Figure 5: Comparison of load imbalances [%]

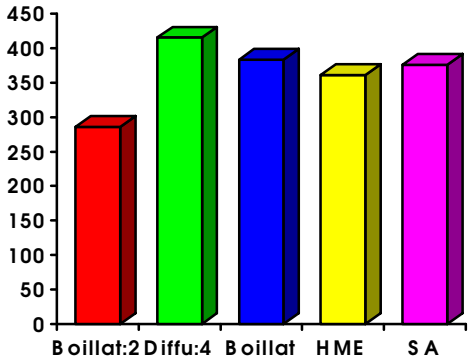

Figure 6: Comparison of communication costs

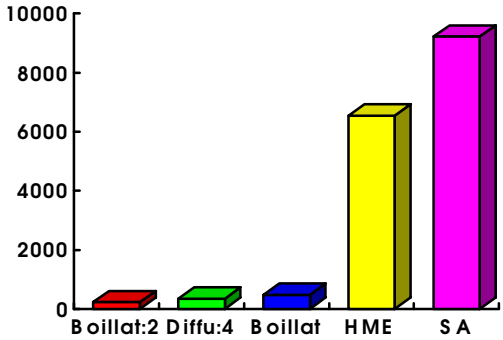

Figure 7: Comparison of time consumption [s]

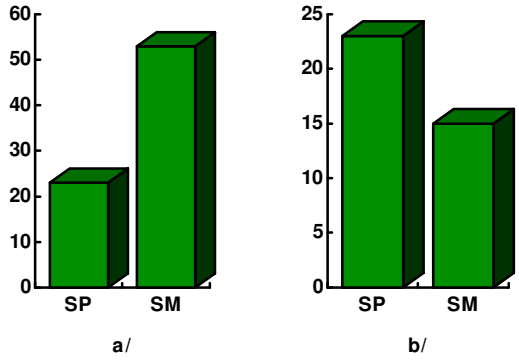

Figure 8: Influence of SM on the path length (a) and communication imbalance (b) [%]

7 Conclusion

Beside the algorithms given above, we have analysed also other approaches: - modified algorithm [3] and genetic algorithms. Our aim was not to evaluate the best optimisation strategy for static mapping only, but also to show which methods can be used in dynamic load-balancing. Further, our attention will be concentrated on the comparison of static traffic load-balancing with the using adaptive routing algorithms. Together with parallel programs monitoring we are implementing the integrated programming tool for automatic development of parallel programs in the PVM environment.

References

[1] KVASNIÇKA, V. - POSPÍCHAL, J. - BISKUPIÇ, S.: Task Allocation Problem Solved by Augmented Simulated Annealing. Central European Journal for Operation Research and Economics, in print.

[2] SELVAKUMAR, S. - MURTHY, S.R.C.: An efficient heuristic algorithm for mapping parallel programs onto multicomputers. Microprocessing and Microprogramming, Vol.36, 1992/93, pp. 83-92.

[3] BOKHARI, S.H.: On the Mapping Problem. IEEE Transactions on Computers, Vol. C-30, 1981, No.3, pp. 207-214.

[4] BOILLAT, J.E. - ISELIN, N. - KROPF, P.G.: MARC: A Tool for Automatic Configuration of Parallel Programs. In: Welch, P. et. al. (Eds.): Transputing 91, IOS Press 1991, pp.311-329.

[5] van LAARHOVEN, P.M.J. - AARTS, E.H.L.: Simulated Annealing: Theory and Applications. Reidel, Dordrecht, 1987.

[6] BOLLINGER, S.W. - MIDKIFF, S.F.: Heuristic Technique for Processor and Link Assignment in Multicomputers. IEEE Transaction on Computers, Vol.40, 1991, No.3, pp. 325-333.

[7] BIANCHINI, R.P. - SHEN, J.P.: Interprocessor Traffic Scheduling Algorithm for Multiple-Processor Networks. IEEE Transactions on Computers, Vol.36, 1987, No.4, pp.396-409.

[8] LEE, S.Y. - AGARWAL, J.K.: A Mapping Strategy for Parallel Processing. IEEE Transactions on Computers, Vol.36, 1987, No.4, pp.433-442.

[9] COROYER, C. - LIU, Z.: Effectiveness of Heuristics and Simulated Annealing for the Scheduling of Concurrent Tasks-An Empirical Comparison. In: Bode, A.-Reeve, M.-Wolf, G. (Eds.): Proceedings PARLE'93. Springer 1993, pp.452-463.

[10] HLUCHÝ, L. - DOBRUCKÝ, M. - DUDÁK, M.: Solving Method for Optimal Load Balancing and Communication Minimisation. In: Plander, I. (Ed.): Proceedings of the Sixth International Conference on Artificial Intelligence and Information - Control Systems of Robots, World Scientific 1994, pp.297-302.

[11] CHEN, W.K.: Theory of nets: flows in networks. John Wiley & Sons, Inc., 1990, 493 pp.