S&D Task Allocation Tool under PVM

L. Hluchý, M. Dobrucký, D. Dobrovodský

[#] 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 static and dynamic task allocation tool under PVM environment for distributed memory parallel systems. For the static allocation the objective function is used to evaluate the optimality of the allocation of a task graph onto a processor graph. Toghether with our optimisation method also augmented simulated annealing and heuristic move exchange methods are implemented. For dynamic task allocation the semidistributed approach was designed based on the division of processor network topology into independent and symmetric spheres. S&D task allocation tool is controlled by user window interface - allocation toolbox.*

Keywords: *distributed memory parallel systems, static and dynamic allocation, multicomputer, load balancing.*

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 allocation 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 allocation .

For static allocation 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 static allocation problem, [1 - 9]. Dynamic task allocation for parallel and distributed computing is an important goal to be achieved for various applications. The purpose of dynamic task allocation is to increase the system throughput in a dynamic environment, which can be done by balancing the utilization of computing resources and minimizing communication overheads among processors during run time. Dynamic task allocation approaches have been studied by numerous researcher [11 - 13]. This paper has 7 sections. Sections 2 and 3 describe the specification of cost function for static allocation and the diffusion method for optimisation of cost function. In section 4 is described dynamic task allocation. In section 5 is described the implementation of static task allocation under PVM. In section 6 is

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

described the implementation of dynamic task allocation under PVM. In section 7 is described S&D task allocation tool box.

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*, allocation A, the general form of the cost function *CF* can be built as a combination of two parts - so called **vertex cost function** F_{vertex} , **edge cost function** F_{edge}:

$$CF(A,t) = F_{vertex}(A,t) + F_{edge}(A,t)$$

Variable *t* is duration of the iteration step of the allocation 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 allocation so that allocation should ensure the minimum of external (interprocessor) communications.

3 The Diffusion Method for Static Allocation

The principle of our allocation 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 $f2_{ij}$ 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 :

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

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

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

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

w_k represents the computation weight of task *k*, *load(x)* is the total weight of all tasks allocated on the processor *x* (A*(k)*=*x*), *links*+1 means the number of those interested in

static allocation (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 allocation 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 allocation , 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 Dynamic Task Allocation

In the recent years the research of dynamic task scheduling in the parallel computer systems brought numerous different approaches. They vary from centralised to fully distributed ones. We have studied some of them from both categories. Centralised methods with a process responsible for making scheduling decisions (scheduler) running on the central node can result in exploitation of its whole computing capacity. If the set of nodes is too large and load information fluctuate rapidly, the communication overhead can overload the part of the interconnection network surrounding the central node. In the distributed approaches it is difficult to keep the node load information topical in the time. Messages with these information can increase the overall communication overhead. To avoid collisions when different schedulers are trying to allocate a task on the same node is time consuming.

We have chosen the compromise between centralised and distributed approach as is proposed in the paper [12] where the semi-distributed method of load balancing is described. The set of processors is divided into independent symmetric regions, called spheres. In each sphere, a processor equidistant from all other processors is selected as the scheduler for that sphere. Cumulative load information for each sphere is exchanged among the independent schedulers. Using knowledge about load of local nodes and information from other schedulers, the scheduler can place a requested task on the appropriate local node or transfer it into a less loaded sphere. Dividing the set of nodes into spheres considers granularity of the tasks. In the problems with small granules, the average lifetime of the process is short. Tasks arise and extinct more frequently than those in the problem with big granules. So the sphere served by one scheduler should be smaller in a way to avoid overload of the central node.

5 The Implementation of Static Allocation in PVM

The allocation 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 allocation 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 allocation 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 (Fig.1, Fig.2). 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.

```
#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.

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*" processes 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 allocation method is to be performed. If it is the case, all the other processes will stop and the master will do the allocation alone.

If the parallel allocation 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 allocation work.

The Diffusion (and some others) allocation methods 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 in Fig.3.

The simulated annealing allocation method uses another parallelisation possibility: a processing pool. The more parallel processes the better quality of the allocation (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 in 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 allocation  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,...) allocation method.

```
while (T > T_min)                    /* temperature is decreasing */
{
 obj_func[0] = Compute( &map[0] ); /* compute new allocation 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 The Implementation of Dynamic Task Allocation in PVM

The Parallel Virtual Machine (PVM) environment has a built-in support for dynamic task allocation. Using library function pvm_regrm() a task can register itself as a Resource Manager (RM). After registration every request for spawning a new PVM task is sent to the RM instead of the PVM daemon to which RM is registered. RM can decide where to spawn a new task with regard to the load balancing.

The application responsible for load balancing presented by us consist of four types of processes (Fig.5). The main process has the user interface and is designed for a determination of spheres, a creation of schedulers and a control of whole system for load balancing. The Scheduler is a process designed to make task placement decisions. The Agent running on the each node collects information about average load. Based on these information Scheduler is updating its database. RMs running on each node catch spawn requests from tasks and invoke the scheduler to decide where the tasks should be placed. Requests to create a new task are sent to the RMs running on destination nodes. They are responsible for the task creation on their nodes. To achieve global load balanced, schedulers inform each other about average load of the sphere. In Fig.6 modified message exchange model after request for spawning a new task is given.

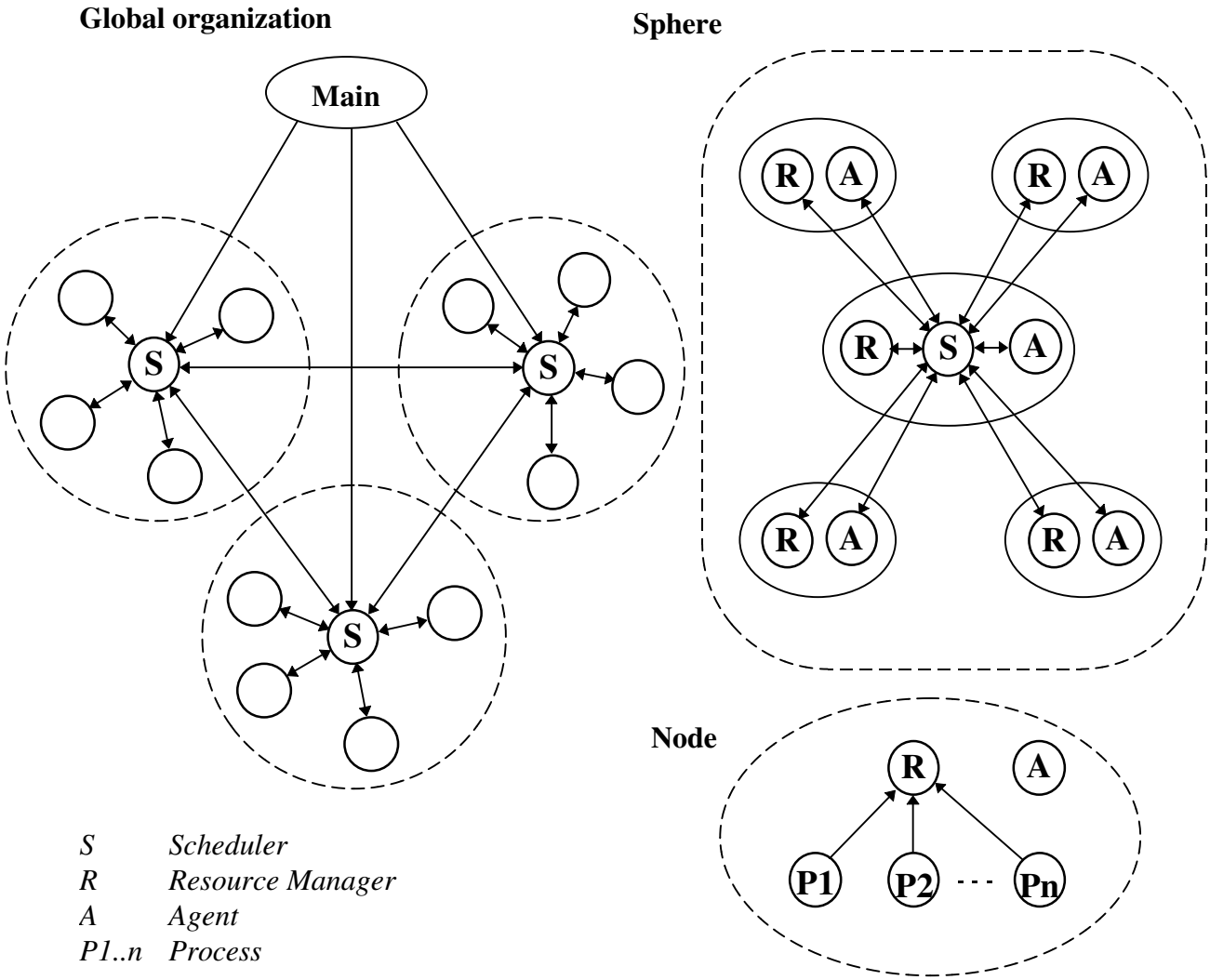
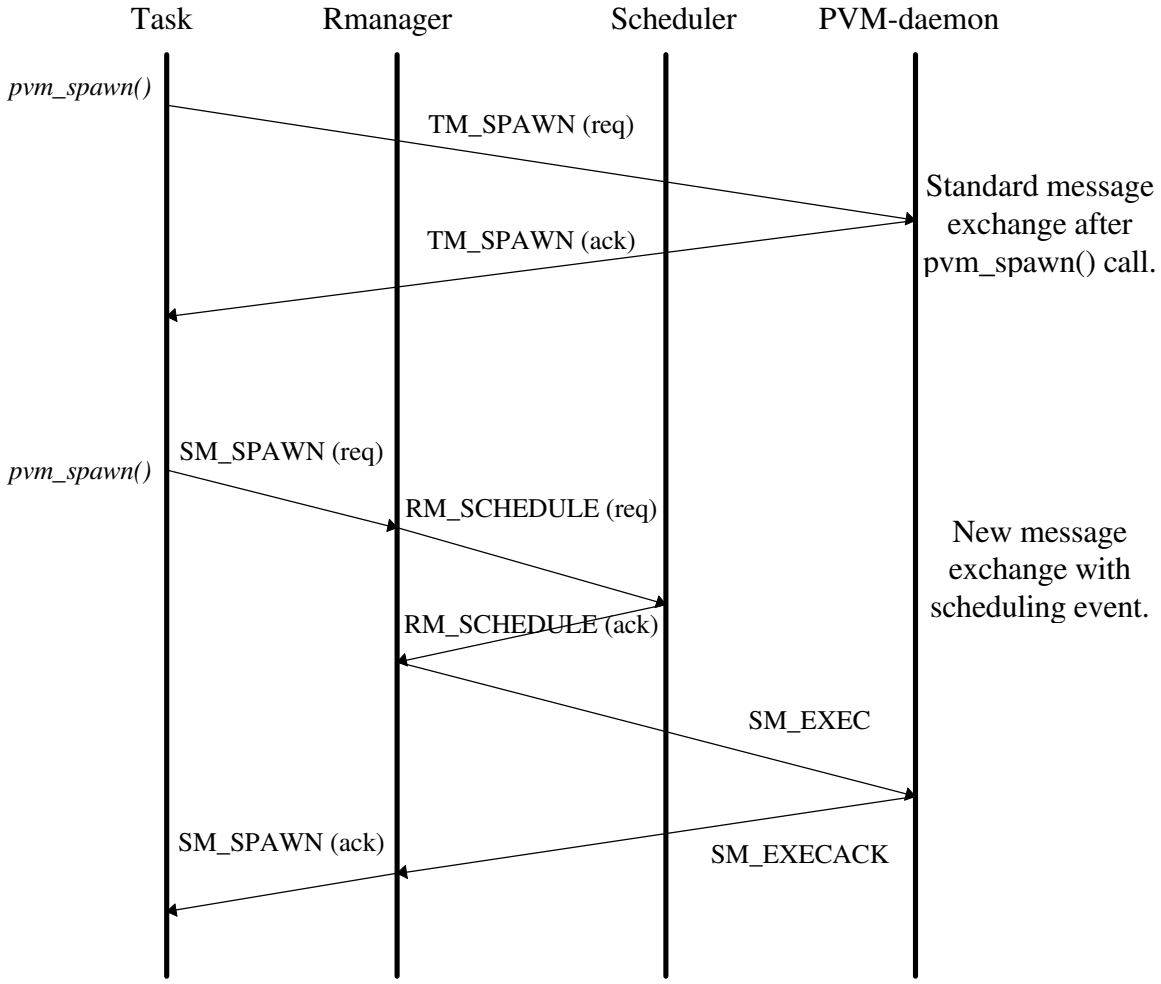


Fig.5: Block scheme of dynamic task allocation.



TM - Task message, SM - Scheduler message, RM - Resource manger message

Fig.6: Modified message exchange model.

7 The S&D Task Allocation Toolbox

We have implemented the S&D task allocation toolbox in the window Motif-like environment. In Figs 7,8 examples of the toolbox window are given.

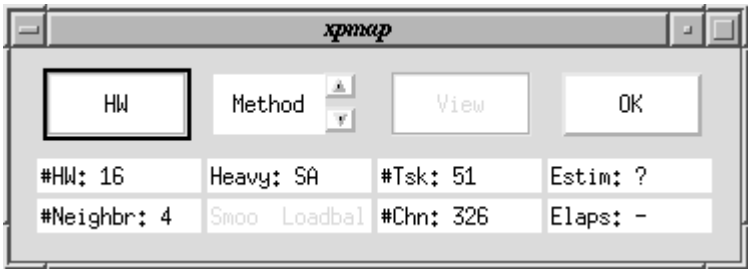


Fig. 7: Starting point - the user program (with 51 parallel tasks and 326 channels among them) is to be allocated onto parallel hardware machine with 16 processors (hypercube)

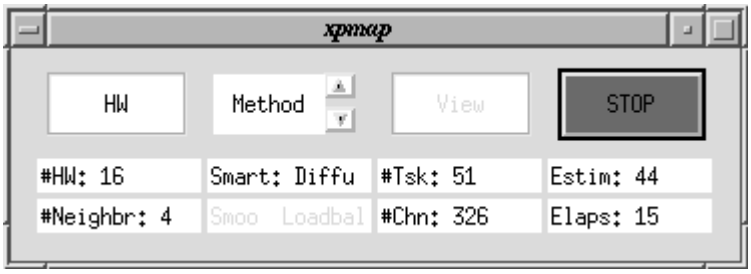


Fig. 8: Running phase - the Diffusion method was chosen.
This is the 15th iteration from 44.

When the S&D task allocation toolbox (S&D TATB) is started it has to read info about SW-graph (description of the application parallel program - task graph) and info about parallel hardware (hardware topology). Then the S&D TATB window appears.

In the bottom part of the S&D TATB window the status parameters are given :

- #HW - the number of PVM-hosts (size of the HW-graph)
- #Neighbr - the maximum of the processor neighbours - diameter of the HW-graph
- "Heavy: SA" - example of the chosen method
- "LoadBal" - indicator of 2-state switch for the dynamic load-balancing (dynamic task allocation)
- #Tsk - the number of parallel tasks to be mapped (SW- graph size)
- #Chn - the number of communication channels between parallel tasks
- "Estim" - estimated [time] consumption (it depends on the chosen method and SW-HW-graphs sizes)
- "Elaps" - progress in the [time] spending on the allocation

The execution button "**OK**" is available only if the allocation process is allowed to be started. Its face can be changed either to recommend further activities (that should be done, i.e. "HW?" if no HW- graph info was found in input file) or "Stop", "Wait" or "Restart". The allocation can be interrupted and started again, for example with the less time-consuming allocation method.

View: Our allocation tool can show the result progress (i.e. imbalance percentage, communication costs, time consumption, number of iterations ...) and also animates it.

Method: The user can change the allocation method. Allocation methods 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 suitable for a medium size tasks) and **heavy** (simulated annealing and "heuristic move-exchange", more time consuming but with the best achieved quality suitable for smaller tasks).

HW: After starting, our tool reads info about SW-graph and HW-graph. Activating this button shows the "pvm:" - number of the live PVM - nodes.

8 Conclusion

In this paper we proposed the cost function and diffusion method for the static task allocation for distributed memory parallel computers. Implementation is done in the distributed form under PVM. Together with the diffusion method also augmented simulated annealing, "heuristic move-exchange" method and several methods for faster allocation (greedy methods) were implemented. In our S&D task allocation tool we have implemented dynamic task allocation based on the semi-distributed approach. We suppose two phase optimisation for allocation of task graph into distributed memory parallel computers: in the first phase static task allocation is done (before compilation) and then during run time dynamic task allocation is executed which balances the utilization of computing resources and minimizes communication overheads among processors. For user we provide S&D task allocation toolbox in which he has possibilities to choose a static allocation method and switch on the dynamic task allocation. Implementation is performed on the generic Unix operating system (Solaris 2.4, IRIX 5, Linux 1.2.x) and PVM 3.3.

In the future we want to verify our S&D task allocation tool on the basis of concrete applications. So far randomly generated graphs were used for testing the S&D task allocation tool.

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 allocation parallel programs onto multicomputers. Microprocessing and Microprogramming, Vol.36, 1992/93, pp. 83-92.

[3] BOKHARI, S.H.: On the Allocation 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 Allocation 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] CHANG H.W. D., OLDHAM W.J.B.: Dynamic Task Allocation Models for Large Distributed Computing Systems. IEEE Trans. on Parallel and Distributed Systems, Vol. 6., No. 12, December 1995, pp. 1301-1315.

[12] AHMAD I., GHAFOOR A.: Semi-Distributed Load Balancing for Massively Parallel Multicomputer Systems. IEEE Trans. on Software Engineering, Vol. 17, No. 10, October 1991, pp. 987-1004

[13] LIN H.Ch., Raghavendra C.S.: A Dynamic Load - Balancing Policy with a Central Job Dispatcher (LBC). IEEE Trans. on Software Engineering, Vol. 18, No. 2, February 1992., pp. 148-158.