12th IMEKO TC4 International Symposium
Electrical Measurements and Instrumentation
September 25–27, 2002, Zagreb, Croatia

THE EXAMPLE OF WORKLOAD ALLOCATION ALGORITHM IN DISTRIBUTED COMPUTING ENVIRONMENT

Nadža Milanović, Mladen Boršić

Faculty of Electrical Engineering and Computing, Zagreb, Croatia

Abstract – To simplify the usage of a computer network as a virtual parallel machine, as well as to introduce the opportunity of distributed parallel computing to Visual Basic users, a DCOM-based software infrastructure has been implemented. This infrastructure is based on a master-slave model. Algorithm for workload allocation in distributed computing system used with this software infrastructure is proposed in this article. Algorithm is used in finding the simple solution of traveling salesman problem and with more complex schedule feasibility analysis.

Keywords: data decomposition, workload, functional decomposition, task distribution

1. INTRODUCTION

In today's world of local area networks, Internet and server-client applications, distributed computing can be good option for solving time-consuming problems. There are numerous software packages and environments that enable distributed computing. In general, workload is allocated to processes in the process pool. Processes are distributed through the computers on the network. On each computer particular task (process) is performed as a part of a workload. Due to a problem, being solved, workload allocation can be based on functional or data decomposition. Furthermore, workload decomposition algorithm for allocation of the tasks is also very important. The algorithm should enable efficient and quick task distribution and should be resistant to dead locks. In the paper, functional and data workload decomposition are explained in short and algorithm for task distribution used in developed software infrastructure for distributed computing [8] is proposed.

2. FUNCTIONAL DECOMPOSITION

Functional decomposition, divides the work based on different operations or functions. Different work processes perform different operations concurrently. Simple example of functional decomposition is workload allocation with respect to the three stages of typical program execution: input, processing and result output. In functional decomposition, such application may consist of three separate and distinct programs, each one dedicated to one of the three phases. Parallelism is obtained by concurrently executing the three programs and by establishing a "pipeline" (continuous or quantized) between them. The example of functional decomposition is trivial and the term functional decomposition is generally used to signify partitioning and workload allocation by function within the computational phase. The computational or processing stage, typically, contains several different sub algorithms, which are performed on the same data set (*MISD – Multiple-instruction single-data)*. These algorithms are then performed in separated programs and executed concurrently.

3. DATA DECOMPOSITION

Data decomposition or partitioning, assumes that the overall problem involves applying computational operations or transformations on one or more data structures and, further, that these data structures may be divided and operated upon. In data decomposition data set is divided in subsets and working processes get different data subset and perform equal operations on given data subsets.

Data partitioning can be done *statically*, where each process knows *a priori* its share of workload, or *dynamically*, where a control process (e.g. master process) allocates parts of the workload to processes as and when they become free. The allocation of workload to working processes is called *scheduling*. In *static* scheduling, individual process workloads are fixed and in *dynamic* scheduling individual process workloads vary as the computation progresses. In most distributed computing environments dynamic scheduling should give better results because of the dynamic nature of the distributed environment. In distributed environments, computers or workstations that take part in distributed computing can have different CPU speeds, different memory sizes and different workloads that are not part of distributed computing. Because of that, some working processes complete their portion of workload much faster or much slower than the others. In dynamic scheduling, parts of workload are given to free working processes and the whole computing process can be done faster.

Furthermore in many examples, data and functional decompositions cannot be completely separated. As a part of a distributed computing, both decompositions are possible. Part of the working processes can perform tasks with different functions (functional decomposition), and another part can perform same operations on different data sets (data decomposition).

4. COMPONENTS OF SOFTWARE INFRASTRUCTURE FOR DISTRIBUTED COMPUTING BASED ON DCOM

Program components used for distributed computing based on DCOM and described in [9] are organized according to master – slave model. The master – slave model of distributed computing is a collection of closely related processes, which typically execute the same code, perform the computations on different portions of the workload and periodically exchange the intermediate results. Example of process organization in master – slave model is shown in Fig. 4.1. In this model, a separate control program, termed the *master*, is responsible for process spawning, initialization, collection and display of the results, and timing of the functions. The *slave* computers perform the actual computation. The master allocates their workloads, statically or dynamically.

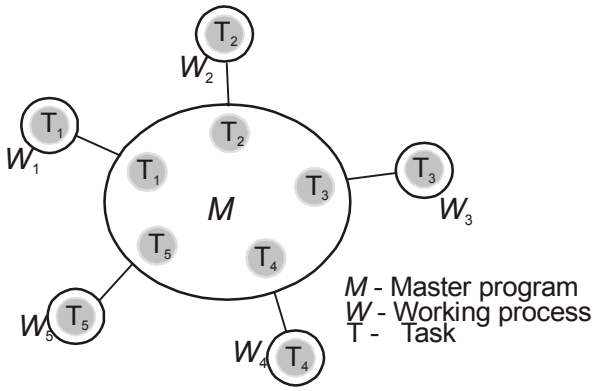

Fig. 4.1. Master – slave model of distributed computing

Shortly, in every session of distributed computing, exist one master computer and several slave computers. DCOM components, parts of the software infrastructure for distributed computing, are created on each computer that is taking part in distributed computing. *Master program* is started on one of the computers in the network and that computer becomes master computer. That program, after preparing the data and tasks, creates an instance of *master communication manager* object, one of the DCOM components. *Master communication manager* object assists in broadcasting the call to the other computers in network and preparing the slave computers for distributed computing. It also collects data about slave computers such as data about the connection, IP address, slave status (busy, idle), time spent on computing etc. First, on each slave computer *slave communication manager* object is created. This DCOM component is also part of the software infrastructure and helps in preparing the slave computer for distributed computing. In addition to *slave communication manager* component, which is problem independent, *working component* must be created on slave computer. This component is problem dependant, and it actually solves the tasks given by the master. When all of the slave computers are prepared and working components are created on them, master program begins with the workload allocation.

Another problem that should be considered in distributed computing is a consistency of the data set in the distributed computing system. The problems, solved with distributed computing, could be divided in two categories. Solving problems from the first category, data set in the system is not altered. In that case, *master program* sends the data set to slaves only once, and gives them the tasks until all are solved. On the other hand, solving some problems affects the data set during the computation and the consistency of the data set must be maintained with the periodical synchronization. The problem dependent functions, built in the *working component*, are responsible for the synchronization. The *master program* records the state of the master data set and keeps a log of transactions performed on the slave machines. Transactions for the synchronization of the data set are included in the new task that is sent to the slave process and are applied as the part of a new task.

In proposed master–slave model, master program controls the workload allocation (data and tasks distribution). The workload allocation algorithm used in [9] is presented as an example in the chapter 5.

After all the tasks have been solved, *master program* calls the *master communication manager*'s function for closing all *working component* objects on the slave computers and thus returning the slave computers in initial state. *Master program* then analyzes the results and presents them to the user.

5. WORKLOAD ALLOCATION ALGORITHM EXAMPLE

Solving the problem with distributed computing, involves task distribution to working computers until all of the tasks have been solved. In order to be able to distribute tasks, master program have to prepare tasks. Master program creates two data arrays with data about tasks. First array, i.e. `TA_1`, contains data linked to the problem. Elements of that array specify tasks and work that should be done as a part of the process of solving the problem. The other array, i.e. `TA_2`, is problem independent array. Elements of this array define task parameters used for workload allocation algorithm and each element corresponds to one element of the array `TA_1`.

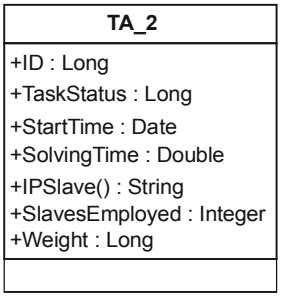

Fig. 5.1. The element structure of the array `TA_2`

Separation of task data in two proposed arrays is not necessary, but it enables easier organization of the program components and employment of the same components for

solving different problems. The structure of the array `TA_1` changes with the problem, but the structure of the array `TA_2` is the same for all applications of distributed computing infrastructure. The element structure of array `TA_2` is shown in Figure 5.1.

Elements of the array `TA_2` enable task management in distributed computing. Attribute `ID` is a simple identification number of each array element. Every element of the `TA_1` array also has this attribute and with those are elements of two arrays related. `TaskStatus` attribute contains data about the state of the task. The task can be in several different states during the process of distributed computing. It can be *unsolved* and none of the working processes is solving it at the time, it can be *in process*, which means that one or more working processes are solving the task. If some working process has solved the task and reported the results to the master program, the task is marked as *solved*. Attribute `StartTime` contains the latest time when the task has been given to one of the working processes for solving, and the attribute `SolvingTime` contains time needed for solving the task. In the attribute `SlavesEmployed` number of working processes that were given the task is stored. The `IPSlave` attribute of the `TA_2` array is also an array that contains the IP addresses of all the working computers that are solving the task at the time. Finally, in the attribute `Weight` is a number that describes the weight of the task in comparison with the other tasks.

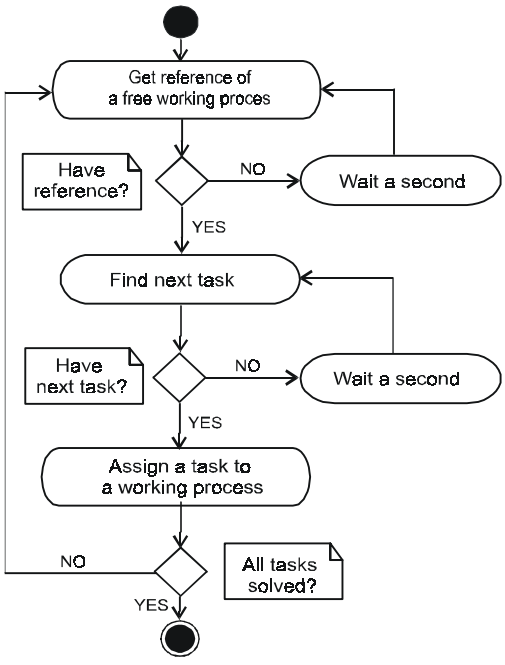

Fig. 5.2. Workload allocation algorithm flow

Apart from these two arrays with the data about tasks, master program also has to have data about the working computers. These data are also stored to an array. Each element of the array corresponds to one working computer and the data of the elements describe working computers. For example master program keeps IP address of the working computer, state of the working computer (free, busy or unavailable), time spent on computing and number of tasks solved. Based on the data from that array, master program chooses free working computer for the next task.

After the workload is divided in tasks, two arrays are prepared and the pool of working computers is ready, master program begins with the allocation of tasks to working processes and starts the process of distributed computing. The flow of the workload allocation algorithm is shown in Fig. 5.2.

In the first step of the algorithm, master program gets a reference of a free working process. The free working process is selected from the working process pool. Master program chooses free working computer that has minimum of computing time. In that way balanced employment of the working computers is achieved. If there is no available free working process, master program waits a second and than tries again. While waiting, master program process responds from working computers and accepts the results of tasks being solved. In that way it is possible that in one or more waiting cycles some working computers become free.

After the free working process is found, master program selects an unsolved task from the task array. Pseudo code for task selection algorithm is shown in Fig. 5.3.

```
/* Task selection */
for i = 0, NumOfTasks - 1
  if TA_2(i).TaskStatus(i) <> TASK_SOLVED
    if TA_2(i).SlavesEmployed = 0
      UnsolvedTask = i
      TaskGiven = False
      i = NumOfTasks + 1
    else
      tmpTime = MaxSolvingTime
      tmpWeight = MaxWeight
      for j = 0, NumOfTasks - 1
        if (TA_2(j).TaskStatus = TASK_SOLVED) &
          (TA_2(j).Weight >= TA_2(i).Weight) &
          (tmpWeight >= TA_2(j).Weight)
                tmpTime = TA_2(j).SolvingTime
                tmpWeight = TA_2(j).Weight
      if TA_2(i).SlavesEmployed *
      (Now - TA_2(i).StartTime) >= tmpTime
        UnsolvedTask = i
        TmpSlvsEmpl = TA_2(i).SlavesEmployed
        TaskGiven = False
/* End */
```

Fig. 5.3. Pseudo code for task selection algorithm

Task selection algorithm is based on the values of the `TA_2` array elements. Task unassigned to any working process (`TA_2(i).SlavesEmployed = 0`) is automatically selected so task selection algorithm ends. If all tasks are already assigned to working processes, one can be assigned again to a free working process. It is possible that the task is not solved because its solving is time consuming or working computer that is solving the task is slow. In order to hasten the solving process, same task can be assigned to one or more working computers. In that case the first solution is accepted. However, next task is found based on the time already spent for its solving, time spent for solving other tasks with same or similar weight and maximum time spent for solving already solved tasks. If time already spent on solving the task is shorter than the time spent on solving a

task with same or similar weight, task is not selected as a next task and will not be additionally assigned to free working process. More time is given to the working process to solve the task. If there is no comparable task solved, maximum time is taken for the evaluation. In the case that none unsolved task is selected, master program waits and than repeats the task selection procedure. While waiting, master program receives the results from working computers, task parameters are changing and unsolved task can be found. After the unsolved task is selected, it is assigned to a working process from the step one. This procedure is repeated until all of the tasks are solved.

6. CONCLUSION

Proposed algorithm for workload allocation is used for solving the problems with implemented software infrastrucuture for distributed computing based on DCOM [9]. Distributed computing was achieved on process pool of ten computers. Two problems were taken as test applications. First problem was traveling salesman problem that was solved with the trivial recursive algorithm. The task set generated for that problem was uniform and all tasks were of the similar weight. The other problem was schedule feasibility analysis. With this problem task generated set was rather diverse with different task weights. For both problems workload allocation algorithm resulted in better performance and less time needed for solving the problems.

REFERENCES

[1] Geoffrey C. Fox, Roy D. Williams, Paul C. Messina; **Parallel Computing Works**; 1994, Morgan Kaufmann Publishers, ISBN 1-55860-253-4; http://www.npac.syr.edu/copywrite/pcw/

[2] Ian Foster; **Designing and Building Parallel Programs**; Adison-Wesley, 1995, ISBN 0-201-57594-9; http://www-unix.mcs.anl.gov/dbpp/

[3] Anna Hondroudakis**, Computer Science Research in HPC**, Version 1.0, Technology Watch Report, February 1996, Edinburgh Parallel Computing Centre, The University of Edinburgh, http://www.epcc.ed.ac.uk/epcc-tec/documents.html

[4] Fadlallah, G.; Lavoie, M.; Dessaint, L-A.; **Parallel computing environments and methods**, Proceedings International Conference on Parallel Computing in Electrical Engineering. PARELEC 2000. IEEE Comput. Soc. 2000, pp. 2-7.

[5] **Cluster Computing** – http://www.mcc.ac.uk/hpc/cluster_computing/

[6] Baker, M., **Cluster Computing White Paper, Final Release, Version 2.0**, 2000, 23 December, http://www.dcs.port.ac.uk/~mab/tfcc/WhitePaper/final-paper.pdf

[7] C. Wang, Y. M. Teo; **Supporting parallel computing on a distributed object architecture**; The Jurnal of Systems and Software 56 (2001) 261-278

[8] N. Milanović, V. Mornar; **A Software Infrastructure for Distributed Computing Based on DCOM**; ITI 2001 – 23rd International Conference on Information Technology Interfaces, Pula, Croatia, June 19 – 22, 2001; IEEE Catalog Number 01EX491, ISBN 953-96769-3-2, ISSN 1330-1012, p. 63-68

[9] N. Milanović; **Parallel Virtual Machine Based on Component Object Model**; Master thesis; March, 2002; University of Zagreb, Faculty of Electrical Engineering and Computing

Authors: M. Sc. **Nadža Milanović**, Department of Fundamentals of Electrical Engineering and Measurements, Faculty of Electrical Engineering and Computing, Unska 3, HR-10000 Zagreb, CROATIA, Tel: +385 1 6129 806, Fax: +385 1 6129 616, E-mail: nadza.milanovic@fer.hr

Prof. dr. sc. **Mladen Boršić**, Department of Fundamentals of Electrical Engineering and Measurements, Faculty of Electrical Engineering and Computing, Unska 3, HR-10000 Zagreb, CROATIA, Tel: +385 1 6129 600, Fax: +385 1 6129 616, E-mail: mladen.borsic@hmd.tel.hr