TOPAS - Parallel Programming Environment for Distributed Computing*

G. T. Nguyen, V. D. Tran, and M. Kotocova†

Institute of Informatics, SAS, Dubravska cesta 9, 84237 Bratislava, Slovakia
giang.ui@savba.sk

Abstract. In this paper, TOPAS, a new parallel programming environment for distributed systems is presented. TOPAS automatically analyzes data dependence among tasks and synchronizes data, which reduces the time needed for parallel program developments. TOPAS also provides supports for scheduling, dynamic load balancing and fault tolerance. Experiments show simplicity and efficiency of parallel programming in TOPAS environment.

1. Introduction

Nowadays, advances in information technologies have led to increased interest and use of clusters of workstations for computation-intensive applications. The main advantages of cluster systems are scalability and good price/performance ratio. One of the largest problems in cluster computing is software [1]. PVM [13] and MPI [14] are standard libraries used for parallel programming for clusters. Although these libraries allow programmers write portable high-performance applications, parallel programming still difficult. Problem decomposition, data dependence analysis, communication, synchronization, race condition, deadlock and many other problems make parallel programming much harder.

As parallel programming is difficult, there are efforts to make parallel compilers that can automatically parallelize sequential programs. However, in general, the parallel programs generated by parallel compilers are much slower than those written by experienced programmers using PVM/MPI. Parallel programs get more and more complex with many unsolved dependences at compilation time, distributed systems get heterogeneous, therefore, according to many experts, parallel programs generated by parallel compilers will not catch performance of PVM/MPI in near future [1][2].

TOPAS (Task-Oriented PArallel programming System, formerly known as Data Driven Graph - DDG [7][9]) is a new parallel programming environment for solving the problem. The objectives of TOPAS are as follows:

- to make parallel programming in TOPAS as easy as by parallel compilers, with the performance of programs in TOPAS is comparable with parallel programs written in PVM/MPI;

* This work is supported by the Slovak Scientific Grant Agency within Research Project No. 2/7186/20

† Department of Computer Science, FEI-STU, Ilkovicova 3, 81219 Bratislava,Slovakia

- to make parallel programs structured, easy to understand and debug, and to allow error checking at compilation time for removing frequent errors;
- to provide support for optimization techniques (scheduling and load balancing);
- to provide some facilities for Grid computing (heterogeneous architectures, task migration, fault tolerance).

The objectives are rather ambitious, but not unachievable. Section 2 shows the main features of parallel program in TOPAS, including scheduling and load balancing. Section 3 focuses on fault tolerance support in TOPAS environment, and Section 4 demonstrates real examples of parallel programs written in TOPAS.

2. Basic Ideas of TOPAS

Fig. 1 shows the steps of parallel programming development. At first, the program is divided into a set of tasks. Next, data dependence among tasks is analyzed; this is the basis for writing communication and synchronization routines in the next step. Finally, the codes of tasks are added. After these steps, programmers have a correct program; however, its performance may not be adequate. Therefore, the parallel program has to be optimized by scheduling that often requires task graphs of the parallel program.

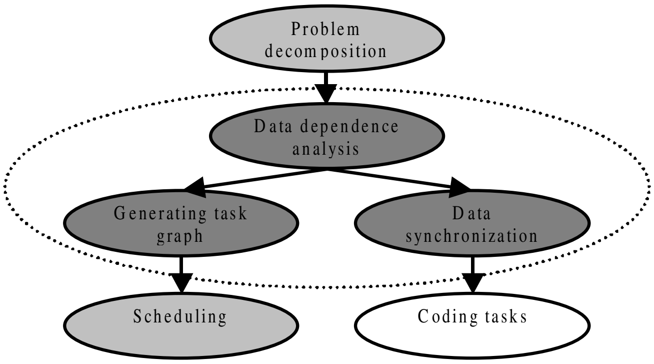

Fig. 1. Parallel program development

Except for the step of coding tasks that is similar to sequential programming, the other five steps can be divided into two groups. The first group consists of problem decomposition and scheduling, which have the following common features:

– these steps are important for performance. Every change in them may affect performance largely;
– there are many possible solutions for these steps and there is no general way to choose the best one.

Experienced programmers often do these steps better than parallel compilers. The other three steps: data dependence analysis, data synchronization and generating task graphs have the following common features:

– each of these steps has only one solution. There are simple algorithms to get it;

– these steps have little effect on performance (note that inter-processor communication is minimized by problem decomposition and scheduling);
– these steps require a lot of work and most frequent errors in parallel programming occur here.

In TOPAS, the performance critical steps (problem decomposition and problem scheduling) are done by programmers and the remaining are done automatically by TOPAS runtime library. This way, writing parallel programs written in TOPAS is as easy as by parallel compiler, while the parallel programs in TOPAS can achieve performance comparable with parallel programs in PVM/MPI.

Parallel programming in TOPAS

The basic units of parallel programs in TOPAS are tasks. Each task is a sequential program segment that can be assigned to a processor; once it starts, it can finish without interaction with other tasks. To extend, tasks can be understood as the sequential parts between two communication routines in PVM/MPI.

Each task in TOPAS consists of a function/subprogram that contains task code and a set of variables the task uses. Tasks are created by calling *create_task()* function with its code and its variables as parameters. Each variable may have read-only (*ro*), read-write (*wr*) or write-only (*wo*) access.

Here is an example of a small sequential program

```
1. a = 10
2. b = a + 10
3. c = a * 5
4. d = b + c
```

and its version in TOPAS (assuming that each line is a task and functions *assign(), incr(), mult(), sum()* have corresponding codes)

```
1. ddg_create_task(assign, wo(a))
2. ddg_create_task(incr, ro(a), wo(b))
3. ddg_create_task(mult, ro(a), wo(c))
4. ddg_create_task(sum, ro(b), ro(c), wr(d))
```

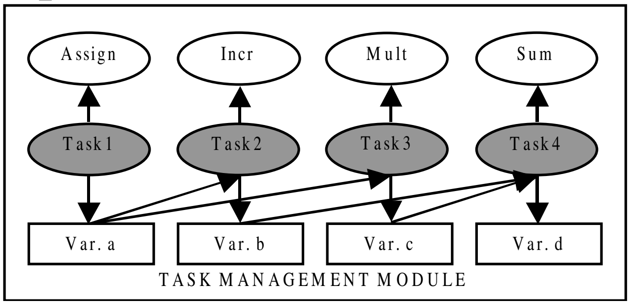

Fig. 2. Memory structures of program in TOPAS

Fig. 2 shows memory structures of the parallel program in TOPAS. Each task is represented by a task object that has pointers to its code and its variables. Using the memory structure, TOPAS runtime library can detect the data dependence among tasks and synchronize data automatically. Real programs in TOPAS are shown in

Section 4. If several tasks write to the same variable, then multi-version technique is used to solve the data dependence among tasks. Each task that writes to the variable creates a new version of the variable. Versions are created dynamically when needed, and they are removed from memory if there are non-unexecuted tasks that use the version. More details on multi-version variable are given in [7]. In any case, the complexity of data dependence analysis and creating internal memory structures of programs in TOPAS is proved to be linearly increasing with the number of tasks.

Applications in TOPAS are compiled as single executable files, which are available on every processor. When a program runs, it creates tasks, schedules, distributes and executes tasks according to the generated schedule. Tasks in TOPAS are executed in non-preemptive mode. A task is executed by calling the code of the task, which is a function in High Level Languages (C/C++, Fortran) with its parameters. When a task is finished, the output data of the task become available for other tasks. According to the memory structures, TOPAS can test all tasks, which use the output data of the executed task, if they are ready. If some of the tasks, which use the output data of the executed task, are assigned to another processor, one copy of the output data is sent to the target processor. Assume that N is the number of tasks, $T_{overhead}$ is the overhead of TOPAS runtime library, T_{total} is the total execution time and $T_{average}$ is the average task execution time. As $T_{overhead}$ is linearly increased with N, there does exist a constant k that $T_{overhead} \leq k.N$. The relative overhead of TOPAS runtime library (the ratio $T_{overhead} / T_{total}$) is

$$\frac{T_{overhead}}{T_{total}} \leq \frac{kN}{T_{total}} = \frac{k}{T_{total}/N} = \frac{k}{T_{average}}$$

In other words, the relative overhead of TOPAS runtime library does not increase with the number of task. If average task execution time is long enough, then the relative overhead is negligible. Section 4 shows the relative overhead of TOPAS runtime library of real problems.

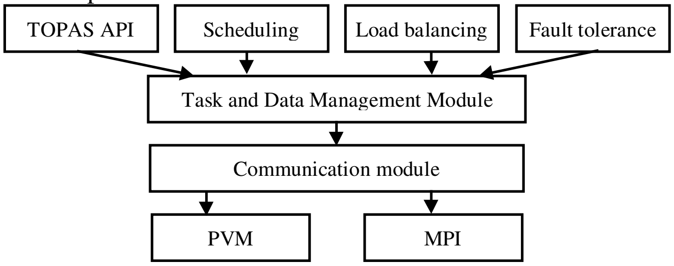

Fig. 3. TOPAS architecture

The architecture of TOPAS library is shown in **Fig. 3**. The kernel of TOPAS library is Task and data Management Module (TMM) that manages all tasks and data, executes tasks and synchronizes data. Communication module creates interface between TMM and underlying communication libraries (PVM, MPI, Nexus) and make parallel programs in TOPAS independent from communication library. TOPAS API is the application-programming interface, which allows programmers to create tasks and variables. Other modules (scheduling, load balancing and fault tolerance) use the memory structures in TMM.

Scheduling in TOPAS

Scheduling is one of the most important techniques to improve the performance of parallel programs. It distributes tasks among processors and arranges task executions in order to get the performance goals such as minimizing the inter-processor communications, balancing load of processors and/or overlapping communications by computations. There are many scheduling algorithms available however their application for programs in PVM/MPI is rather limited [3][4][6][8]. Most scheduling algorithms need the task graphs of parallel programs as input and generating the task graph from a parallel program written in PVM/MPI is difficult. TOPAS solves the problem completely by using the memory structure in TMM. If each task and its output variables are considered as a node in task graph; then the memory structures can be considered as the task graph of the program

When a parallel program in TOPAS runs, after tasks have been created, TMM calls function *ddg_schedule()*, which receives the task graph as input parameter and returns a generated schedule as output parameter. Programmers can customize scheduling by overwriting the *ddg_schedule()*. There are many choices: they can call one of the implemented scheduling algorithms in TOPAS library, write their own scheduling algorithms or save the task graph in a file, use external scheduling tools to schedule and then read the generated schedule back from a file. In any case, when programmers change the scheduling algorithms they do not have to modify the other parts of codes. This is a considerable advantage of parallel programming in TOPAS: in classical PVM/MPI programming, change task distributions and execution orders often require major modification in the parallel program source code. TOPAS runtime library also eliminates frequent errors in parallel programming, including race conditions and deadlocks. Default scheduling in TOPAS library is done by Dominant Sequence Clustering [12] algorithm. Program codes are not affected by changing task distribution or execution order of new scheduling algorithms provided by programmers.

Dynamic load balancing

If the behavior of parallel programs is unpredictable, e.g. tasks are created dynamically, then dynamic load balancing is one of the best ways to optimize the execution of parallel programs. Dynamic load balancing algorithms try to keep all processors busy by moving tasks from heavily loaded processors to lightly loaded ones. One of the largest obstacles of using dynamic load balancing is task migration [10]. Migration of a running task requires large overhead: stopping the task, saving the state of the task, sending the state and code to the target processor, restarting the task with the saved state. When a system is heterogeneous, codes of tasks generated in different platforms can differ from each other and do not support code migration. Task migration also often requires additional work from programmers or supports from operating system.

In TOPAS, moving task from a processor to another can be done easily by sending its task object to the target processor. The task code is already available in the target processor because the same executable file runs on each processor (like SPMD programs). Therefore the cost of task migration is sending 70-80 bytes (the size of the task object) to the target processor. Task migration in TOPAS can be done in systems with heterogeneous architectures due to the fact that programmers do not have to move the code. From the viewpoint of data synchronization among tasks, it is not significant if a task is placed in the source or the target one and data are sent to the target processor if necessary.

3. Fault Tolerance in TOPAS

As the size of computer systems increases unsteadily, the probabilities that faults may occur in some nodes of the system increase as well. Therefore, it may be necessary to ensure that the applications may continue in the system despite the occurrence of faults. This is very important in Grid computing, as computational nodes are not managed by one owner and communications via Internet are unreliable.

Traditional system recovery applies in fixed order: fault detection and location, FT reconfiguration, restarting system with valid data (e.g. from the last valid checkpoint). When hardware redundancy is not available, software redundancy is unavoidable.

Output data checkpoints

The basic concept of fault tolerance in TOPAS is to keep spare instances of output data in other processors as checkpoints. Assume that only one single hardware permanent fault can occur in a node; then

- one spare instance of the output data is sufficient, if the output data is required by successors that are allocated in the same node
- no spare instance is necessary, if the output data is required by successors, which are allocated in at least two different nodes. After a fault occurrence, if one node is faulty, another instance of output data is still available in another healthy node.

Each node needs information about the system and task states in its memory to maintain its duties. Such information can be created when an application starts and it is updated when each task finishes. The information can be divided to the data-flow graph of the application, states of all nodes, tasks and data, a list of data instance locations. There is a problem to keep low number of inter-node communications, otherwise the system performance decrease significantly. If a spare instance is created, its location must be announced to all nodes and this can cause a large number of broadcasts. Therefore, in TOPAS, spare instances are located in the neighbor node of the successor node. If the output data is required by at least two tasks in different nodes, locations of all data instances are in data-flow graph of the application.

Removing obsolete output data

In TOPAS, when fault-tolerant feature is not supported, removing unnecessary output data is quite simple. Each node has the data-flow graph of the application and knows about the need of all data in the node exactly. There is no requirement to know if another node will need checkpoints (spare instances of data) allocated in the node. When the fault-tolerant feature of TOPAS environment is supported, the use of multi-version variables is unavoidable and the situation is more complicated. Output data checkpoints are kept until they are unnecessary for all tasks. The moment when no task needs certain data version is not easy to determine. Broadcast messages and task synchronization are frequently mentioned in literature. They usually cause a large number of inter-node communications and/or keep large amount of obsolete data.

Because of such disadvantages, TOPAS removal method has been designed. When a task is running, the node has own information about finished and unfinished tasks in the system. The information is collected

– using the data-flow graph of the application, which is available on each node to determine all finished task predecessors
– when there is a spare instance of an output data, the node knows the task, to which the spare instance belongs. In this case, the owner of the spare instance must be finished and its predecessors are finished too.

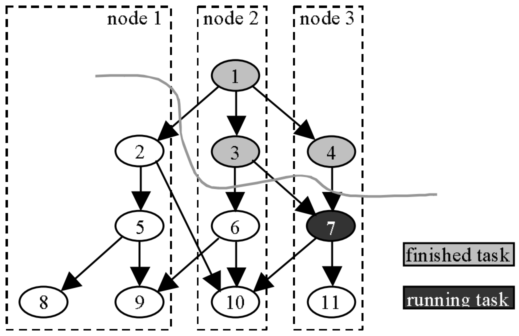

Fig. 4. Detected finished tasks from the third node viewpoint

From the viewpoint of a certain node, the set of all tasks is divided into subset of finished tasks and a subset of remaining tasks. The subset of remaining tasks contains also finished tasks that are not detected by the node. The data on the gray boundary curve (**Fig. 4**) must be kept and the remaining data can be removed. The advantage of this method is no need of broadcast and synchronization. The disadvantage is that not all obsolete data are removed. In **Fig. 4**, the multiprocessor system has three nodes. Task 7 is one of running tasks. The third node determines that task 1, 3 and 4 are finished. Task 2 had been finished on the first node, but it has not been detected by this node. The data moved from task 1 to task 4 and from task 1 to task 3 can be removed, e.g. the third node can remove such data instances, which are located on it.

The number of data that must be kept in the whole system is smaller than *O(n)* where *n* is the number of application tasks and is greater than *O(p)* where *p* is the level of parallelism of the parallel application. When a faulty node is detected and located, TOPAS system diagnosis reports to all remaining nodes *"i-th node is down"* and applications have to restart from the last valid checkpoint. In addition, nodes in multiprocessor systems are assumed to be fail-silent, i.e. they only send correct messages or nothing when faults occur.

Task reconfiguration

The memory structures, automatic data synchronization, runtime scheduling and task migration of TOPAS provide the basic services for implementing fault-tolerance features. When a fault occurs, TOPAS moves the unexecuted tasks on the faulty processor to a healthy one. After fault occurrence, new task locations are determined using reconfiguration algorithms with the criteria of fast recovery time and a graceful degraded performance. Of course, no solution can fulfil these criteria in the best way and an optimal solution is somewhere between them. Scheduling algorithms can be used and can provide the best performance for the system after reconfiguration, but do not give any guaranty for fast recovery time. Usually, the code of application program is small in comparison with data. In TOPAS, task codes are available on every node and there is no problem to reorganize the task-running order. The problem is in moving existing data due to task reallocations. Such data can be large and cause communication delay and long reconfiguration time. Default reconfiguration algorithm finds a suitable node to reallocate each task in the faulty node with the criterion to minimize the amount of data needed to move. It also provides possible good load balancing for the rest of the system. Similarly to like TOPAS scheduling, programmers can customize reconfiguration algorithm by overwriting the *ddg_reconfiguration()* to achieve their desired goals [5].

In healthy nodes, running tasks finish their work as normally, then refresh their states and start to realize new configuration. After reconfiguration, running tasks in the faulty node are added to waiting queues for processors on other healthy nodes and will be restarted with valid input data. Waiting tasks in the faulty node are added to waiting queues in other nodes. All remaining tasks continue from their last states.

Details of fault tolerance in TOPAS are also given in [11].

4. Case Study

In this section, Gaussian elimination algorithm is implemented as parallel program in TOPAS. This chosen problem is representative for large classes of parallel applications, so it can prove the usability of TOPAS library.

```
#define N 1200
main()
{ float a[N][N];
  init(a);                 // initial the values of a
  for (int i = 0; i < N-1; i++)
    for (int j = i+1; j < N; i++)
    { coef = a[j][i] / a[i][i];
      for (int k = i+1; k < N; k++)
        a[j][k] = a[j][k] - coef*a[i][k];
     }
     print(a);             //print the result values of a
}
```

Gem01: Sequential Gaussian elimination algorithm

The sequential Gaussian elimination algorithm - GEM is described in *Gem01*. We concentrate only on GEM, the input and output functions *init()* and *print()* are not considered. The tasks are defined from the lines inside two outer loops, (line 7, 8, 9 in *Gem01*). Before defining a task, the code of the tasks has to be moved to a function. Finally, the task is created from the code (*Gem02*). It can be seen in the final version of GEM in TOPAS, that there are no communication routines in the program. All data synchronization is done by TOPAS runtime library.

```
1. #include "ddg.h"
2. #define N 1200
3. typedef float vector[N];
4. void ddg_main()
5. { ddg_var_array<vector> arr(N);
6.   init(a);
7.   for (int i = 0; i < N - 1; i++)
8.     for (int j = i+1; j < N; j++)
9.       ddg_create_task(compute,ddg_direct(i),
                         ddg_ro(arr[i]), ddg_rw(arr[j]));
10.    print(a);
11. }
12. void compute(ddg_var<int> i,
                 ddg_var<vector> i_line, ddg_var<vector> j_line)
13. {  coef = j_line[i] / i_line[i];
14.    for (int k = i + 1; k < N; k++)
15.      j_line[k]=j_line[k] -    i_line[k]*coef;
16. }
```

Gem02: Final version in TOPAS

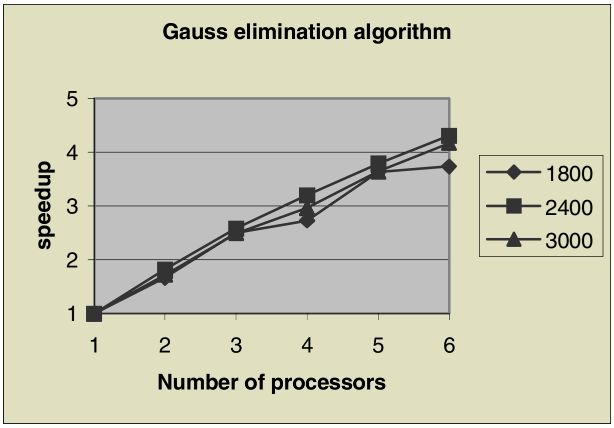

Fig. 5. Speedups of Gaussian elimination algorithm

Experiments are performed on a PC cluster of six Pentium 500 MHz connected by 100Mb Ethernet. The speedup of Gaussian elimination algorithm in TOPAS is shown in **Fig. 5**. TOPAS runtime library causes about 0.3% overhead of total execution time.

When TOPAS provides fault-tolerant feature, runtime library causes larger overhead, i.e. about 1.3% to 1.9% total non-fault execution time.

5. Conclusion

In this paper, the TOPAS parallel programming environment is described. TOPAS does not only allow programmers to write parallel programs easily but also provides facilities for scheduling, dynamic load balancing and fault tolerance features. Experiments have demonstrated the usability of TOPAS on real problems, the simplicity of parallel programming in TOPAS, the performance of TOPAS programs and the overhead of TOPAS runtime library. The latest work in TOPAS is involved in make TOPAS library to Grid computing.

References

1. H. El-Rewini, T.G. Lewis: Distributed and Parallel Computing. Manning Publication, USA, 1998.
2. Kennedy: Compilers, Languages and Libraries. The Grid: Blue Print for a New Computing Infrastructure, pp. 181-204, Morgan Kaufmann, 1999.
3. Senar M.A., Cortes A., Ripoll A., Hluchy L., Astalos J.: Dynamic Load Balancing. Parallel Program Development for Cluster Computing. Nova Science Publishers, USA, 2001.
4. H. El-Rewini, H. H. Ali, T. Lewis: Task Scheduling in Multiprocessing Systems. Manning Publication, USA, 1999.
5. L. Hluchy, M. Dobrucky, J. Astalos: Hybrid Approach to Task Allocation in Distributed Systems. Computers and Artificial Intelligence, vol.17, No.5, pp. 469-480, 1998.
6. B. A. Shirazi, A. R. Hurson, K. M. Kavi: Scheduling and Load Balancing on Parallel and Distributed Systems. IEEE Computer Society Press, 1995.
7. V. D. Tran, L. Hluchy, G. T. Nguyen: Parallel Programming Environment for Cluster Computing. CLUSTER'2000, pp. 395-396, November 2000, Germany. IEEE Computer Society Press.
8. L. Hluchy, M. Dobrucky, D. Dobrovodsky: Distributed Static and Dynamic Load Balancing Tools under PVM. First Austrian - Hungarian Workshop on Distributed and Parallel Systems, Miskolc, Hungary, 1996, pp.215-216.
9. V. D. Tran, L. Hluchy, G. T. Nguyen: Parallel Program Model for Distributed Systems. EuroPVM/MPI'2000, pp. 250-257, September 2000, Hungary. Springer Verlag.
10. M. Richmond, M. Hitchens: A New Process Migration Algorithm, Operating System Review, 31(1), 1997, 31-42.
11. G. T. Nguyen, V. D. Tran, L. Hluchy: DDG Task Recovery for Cluster Computing. PPAM'2001, Poland, September 2001. Springer Verlag. To appear.
12. T. Yang, A. Gerasoulis: DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. IEEE Transaction on Parallel and Distributed Systems, Vol. 5, No. 9, pp. 951-967, 1994.
13. PVM: Parallel Virtual Machine http://www.epm.ornl.gov/pvm/pvm-home.html
14. MPI - Message Passing Interface http://www.erc.msstate.edu/mpi/