Parallel Probabilistic Computations on a Cluster of Workstations*

A. Radenski (a), A. Vann (a), and B. Norris (b)

(a) Computer Science Department, Winston-Salem State University,
Winston-Salem, NC 27110, USA, radenski@ga.unc, lvann@aol.com

(b) Computer Science Department, University of Illinois at Urbana-Champaign,
1304 W. Springfield Ave., Urbana, Illinois 61801, USA, brnorris@uiuc.edu

Probabilistic algorithms are computationally intensive approximate methods for solving intractable problems. Probabilistic algorithms are excellent candidates for cluster computations because they require little communication and synchronization. It is possible to specify a common parallel control structure as a generic algorithm for probabilistic cluster computations. Such a generic parallel algorithm can be glued together with domain-specific sequential algorithms in order to derive approximate parallel solutions for different intractable problems.

In this paper we propose a generic algorithm for probabilistic computations on a cluster of workstations. We use this generic algorithm to derive specific parallel algorithms for two discrete optimization problems: the knapsack problem and the traveling salesperson problem. We implement the algorithms on clusters of Sun Ultra SPARC-1 workstations using PVM, the parallel virtual machine software package. Finally, we measure the parallel efficiency of the cluster implementation.

1. Introduction

Probabilistic algorithms, such as Monte-Carlo trials [3], genetic algorithms [9], hill-climbing [9], and simulated annealing [1], are approximate methods for intractable problems, i.e., problems for which no efficient exact algorithms are believed to exist. The knapsack problem [8] and the traveling salesperson problem [7] are examples of intractable discrete optimization problems for which probabilistic algorithms seem promising.

The *knapsack problems* comprise a well-known class of combinatorial optimization problems. The *0/1 knapsack problem of size n* can be formulated as follows: Given the capacity of the knapsack, and the weights and the values of *n* objects, choose which objects to include in the knapsack so that the cumulative value of the objects is maximized without exceeding the capacity of the knapsack. Knapsack problems are hard to solve because of the vast number of possible knapsack assignments. Exact computations are theoretically possible but are not feasible because they have resource requirements that cannot be practically satisfied. For the 0/1 knapsack problem, a sub-optimal solution can be found by applying hill climbing and crossover

* This work is supported by NASA Grant NAG3-2011 and by NSF Grant CCR-9509223.

to a small set of possible solutions. In this paper, we demonstrate how such a solution can be derived from a generic probabilistic parallel algorithm.

The *traveling salesperson problem* is another famous combinatorial optimization problem: A salesperson must visit each of *n* cities and return to the initial city. The aim is to minimize the length of the tour. With the exception of very small problems, exhaustive search cannot deliver an optimal solution within reasonable time limits. A *sub-optimal* solution for the traveling salesperson problem can be found by simulated annealing of a set of several tours. In this paper, we demonstrate how such a solution for the traveling salesperson can be derived from the *same generic probabilistic parallel algorithm* that we employ to solve the *0/1 knapsack problem*.

A typical probabilistic algorithm tries to approximately solve a problem with a vast number of potential solutions by first generating a smaller set of candidate solutions. Then, the algorithm tries to improve the set of candidate solution by means of some probabilistic computations, such as simulated annealing, hill-climbing, crossover, or Monte-Carlo trials. Such probabilistic computations can be very intensive yet highly independent for the individual candidate solutions. As a consequence, a cluster of workstations can efficiently perform probabilistic computations with little necessary coordination between the processors.

It is possible to specify a common parallel control structure as a generic algorithm for probabilistic cluster computations. Such a generic algorithm implements process control and communication in a *problem-independent manner*. The generic parallel algorithm can be glued together with domain-specific *sequential* algorithms in order to derive approximate parallel solutions for different intractable problems.

In Section 2 we propose a generic algorithm for parallel probabilistic computations on clusters of workstations. In Sections 3 and 4 we use this generic algorithm to derive specific parallel algorithms for two discrete optimization problems: the knapsack problem and the traveling salesperson problem. In Section 5 we describe implementations of the algorithms on homogeneous clusters of workstations using PVM, the parallel virtual machine software package. In the same section, we present performance measurements on Sun Ultra SPARC-1 clusters of workstations. The last section relates our contributions to work performed by others and draws some conclusions.

2. A Generic Parallel Probabilistic Algorithm

We assume that a problem is defined by the type of its *solution* and by three *sequential* methods:

- a method to *improve* individual candidate solutions;
- a method to *test* if a *set* of candidate solutions contain an acceptable approximation;
- a method to *generate* a new set of candidate solutions for further improvements.

The algorithm maintains a *set s* of *n* candidate solutions and tries to improve them by means of one *master* and *p server* processes (Fig. 1). The master is connected to each server with a two-way communication *channel*. The master process sends $q = n/p$ candidate solutions to each server. The servers probabilistically *improve* their assigned candidate solutions in parallel and then send the improved versions back to the master. The master *tests* the set of current solutions and, if no more improvements are needed, terminates the servers and itself. Alternatively, if further improvements are needed, the master *generates* a new set of candidate solutions and again sends them to the servers.

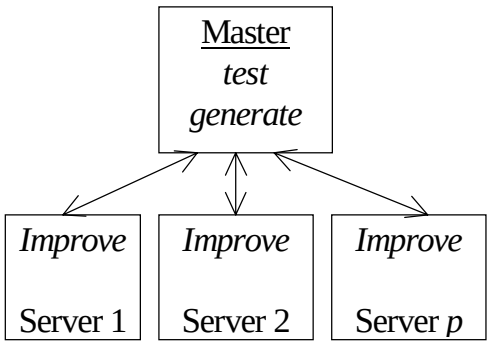

Figure 1. Master and Server processes.

```
const  n = ..;  {candidate solutions}
       p = ..;      {servers , n mod p = 0}
type solution = …;
       set = array[1..n] of solution;
procedure improve(
       var s: set; left, rght: integer); …
procedure generate(var s: set); …
procedure test
       (var s: set; var done: boolean); …
```

Figure 2. Parameters of the algorithm.

We specify and test this generic *master-server* algorithm using the publication language SuperPascal* [5]. Procedures *improve*, *test*, *generate*, and the type of the problem *solution*, are *parameters* of the generic algorithm (Fig. 2). Procedure *improve* operates on a subset *s[first..last]* of the set *s* of candidate solutions. Procedure *test* signals through its parameter *done* whether the current solution *s* is acceptable, i.e. if the current solution does not need more improvements. In case further improvements *are* needed, procedure *generate* can be invoked to restructure the current set *s* of candidate solutions; after this restructuring, the candidate solutions can be sent again to the servers for further improvements. The type of the *solution* and procedures *improve*, *test*, and *generate* are left unspecified in the generic algorithm because they vary significantly from one problem to another.

A net *c* of channels capable of transmitting messages of types *solution*, *boolean*, and *integer* is declared and opened as shown below:

```
type channel = *(solution, boolean, integer);   type net = array [1..p] of channel;
var c: net;                                     for i := 1 to p do open(c[i]);
```

The master distributes the *n* candidate solutions among the *p* servers by assigning *q = n/p* of them to each server. As shown in Fig. 3, procedure *master* repeatedly sends *q* candidate solutions to each server and receives them back improved. The master then tests if the improved candidate solutions are satisfactory, and if they are not, the master generates a new set for further improvements. The result from the test is broadcast to the servers so that they terminate when the problem is solved, or, alternatively, receive a new subset of *q* candidate solutions.

The master communicates *q* candidate solutions to and from server *i* as specified by procedures *sendToServer* and *receiveFromServer* (see Fig. 3).

Repeatedly, each server (1) receives from the master a subset of *q* candidate solutions, (2) applies procedure *improve* to that subset, (3) sends the improved candidate solutions back to

* Once it has been tested, a SuperPascal algorithm can be easily implemented in any practical parallel environment, e.g. C and PVM.

the master, and (4), receives a notification from the master if more improvements are needed. A specification of the *server* algorithm is presented in Fig. 4.

```
procedure master(var s: set; c: net);
var i: integer; done: boolean;
begin
    repeat
        for i := 1 to p do
            sendToServer(i);
        for i := 1 to p do
            receiveFromServer(i);
        test(s, done);
        for i:= 1 to p do
            send(c[i], done);
        if not done then generate(s);
    until done
end;
```

```
procedure sendToServer(i: integer);
var k, first, last: integer;
begin
    first := (i - 1)*q + 1; last := i*q;
    for k := first to last do send(c[i], s[k]);
end;

procedure receiveFromServer
    (i:integer);
var k, first, last: integer;
begin first := (i - 1)*q + 1; last := i*q;
    for k := first to last do
        receive(c[i], s[k]);
end;
```

Figure 3. Master algorithm.

Finally, the algorithm from Fig. 5 creates the network displayed on Fig. 1 by (1) opening p channels and by (2) running in parallel one master and p server processes.

```
procedure server(c: channel; i: integer);
var s: set; first, last, k: integer;
    done: boolean;
begin first := (i - 1)*q + 1; last := i*q;
    repeat
        for k := first to last do
            receive(c, s[k]);
        improve(s, first, last);
        for k := first to last do
            send(c, s[k]);
        receive(c, done)
    until done;
end;
```

Figure 4. Server algorithm.

```
procedure compute(var s: set);
var c: net; i: integer;
begin
    for i := 1 to p do
        open(c[i]);
    parallel
        master(s, c) |
        forall i := 1 to p do
            server(c[i], i)
    end
end;
```

Figure 5. Master-server network

Given this generic parallel algorithm, one can derive a parallel algorithm for a particular problem by specifying the particular type of its *solution* and the problem-specific, sequential procedures *improve*, *test*, and *generate*. (One has to also specify two less significant procedures, *initialize* and *summarize* which are not discussed in this paper.) Finally, procedure *compute* can be invoked to solve concrete problem instances.

We use the generic master-server algorithm to derive parallel algorithms for two different discrete optimization problems: the knapsack problem and the traveling salesperson problem. We achieve this by defining the type of the *solution* for the each specific problem, and by defining concrete sequential versions of procedures *improve*, *test*, and *generate*.

3. Deriving a Parallel Evolutionary Algorithm for the Knapsack Problem

We consider a 0/1 knapsack problem of size *m*: Choose, which objects to include in a knapsack so that the cumulative value of the objects is maximized without exceeding the knapsack capacity [8].

We derive a *parallel evolutionary algorithm* for the knapsack problem from the generic master-server algorithm as follows. We represent the candidate solutions as binary chromosomes, so that *true* in position *k* means that object *k* is chosen and *false* means it is not:

```
type bitArray = array[1..m {problem size}] of boolean;
    solution {chromosome} = record
        bit: bitArray;
        fitness: real;
    end;
```

The fitness of the chromosome is measured by the cumulative values of the chosen objects. Procedure *improve* is defined to perform extensive hill-climbing on a number of replicas of each chromosome *s[i], i = first, first+1, …, last*. The most fitted chromosome, which results from the hill-climbing, becomes the improved version of *s[i]*, as specified below:

```
procedure improve(var s: set; first, last: integer);
var i: integer;
begin for i := first to last do
    {generate a temporary population consisting of a number of clones of s[i] and
    perform hill-climbing on each member of that population; replace s[i] with the
    best fitted chromosome obtained by hill-climbing}
end;
```

Procedure *generate(var s: set)* is defined to modify the set *s* of chromosomes by applying the genetic *crossover* operator [9]. Finally, procedure *test(var s: set; var done: boolean)* is defined to accept the current fittest chromosome after a predefined number of generations. Procedure *test* also maintains a copy of the best candidate solution currently found by this algorithm. A complete specification of the knapsack algorithm is available in [10].

4. Deriving a Parallel Simulated Annealing Algorithm for the Traveling Salesperson Problem

We consider the popular traveling salesperson problem: Find the shortest tour to visit each of *m* cities once and return to the initial city. We derive a *parallel simulated annealing algorithm* for the traveling salesperson problem from the generic master-server algorithm as follows. Similarly to [4, Ch.11], a *city* is represented by its real coordinates in the plane. A candidate *solution* is an array of different cities, also called a *tour*, and the length of the tour is the sum of the distances of successive cities:

```
type city = record x, y: real end;
    solution {tour} = array [1..m {problem size}] of city;
```

Procedure *improve(var s: set; first, last: integer)* is defined to improve tours *s[first], s[first+1], …,s[last]* by simulated annealing, as specified below:

procedure *improve*(**var** *s*: *set*; *left*, *right*: *integer*);

var *i*: *integer*;

begin for *i* := *first* **to** *last* **do** *{apply simulated annealing to tour s[i]}* **end**;

Procedure *generate(var s: set)* is null, i.e., specifies no actions. Finally, procedure *test(s: set; var done: boolean)* is defined to always return *done = true* and, therefore, terminate the master and the server. Thus, in this algorithm each server improves only one subset of $q = n/p$ candidate solutions. A complete specification of the traveling salesperson algorithm is available in [10].

5. Cluster Implementation and Performance Evaluation

We reprogram the generic probabilistic algorithm for a 10Mbps Ethernet cluster of Sun Ultra SPARC-1 workstations using the programming language *C* and the software package *PVM* [13]. From the generic implementation, we derive implementations of the evolutionary and the simulated annealing parallel algorithms. For this purpose we reprogram from *SuperPascal* to *C* the serial code for procedures *improve, test,* and *generate,* which we then compile with the *C/PVM* implementation of the generic algorithm. A more detailed description of the derivation of a C/PVM implementation can be found in [10].

Table 1. Knapsack.

p	T(p)	E(p)
1	337	1.00
2	178	0.95
4	98	0.86
8	61	0.69
16	47	0.45

Table 2. Traveling Salesperson.

p	T(p)	E(p)
1	1153	1.000
2	578	0.997
4	291	0.991
8	146	0.987
16	74	0.974

Table 1 shows the average run time *T(p)* in seconds for parallel solution of a randomly generated knapsack problem of size 400 on 1, 2, 4, 8, and 16 workstations, with a set of 16 candidate solutions. The processor efficiency, $E(p) = T(1)/(p*T(p))$ is high for a small number of processors.

Table 2 shows the average run time *T(p)* in seconds for parallel computation of an optimal tour of 400 cities on 1, 2, 4, 8, and 16 workstations, with a set of 16 candidate solutions. The processor efficiency *E(p)* is higher than 97%.

Figures 6 and 7 graphically represent the wall-clock execution time and speedup of the algorithms. Both algorithms are expected to scale well on the set of candidate solutions. The traveling salesperson algorithm scales well for large number of processors on the problem size as well, while the knapsack algorithm has more limited scalability due to the communication patterns inherent to the algorithm.

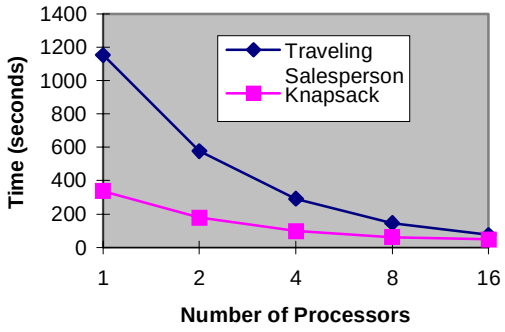

Figure 6. Wall-clock execution time.

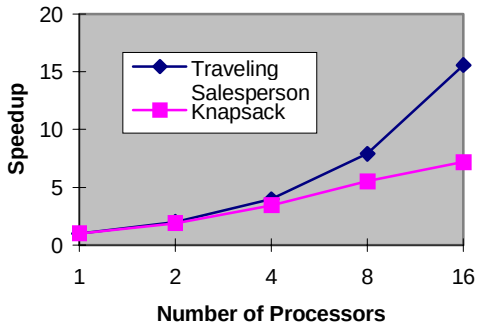

Figure 7. Speedup.

6. Conclusions

In this paper, we have proposed a generic probabilistic algorithm for probabilistic cluster computations. From the generic algorithm, we have derived an evolutionary algorithm for the knapsack problem and a simulated annealing algorithm for the traveling salesperson problem. The derivation consists in gluing together the parallel generic algorithm with *sequential* problem-oriented code. We have implemented these algorithms on a homogeneous cluster of Sun Ultra SPARC-1 workstations using C and PVM. The processor efficiency of the implementations is over than 97% for the traveling salesperson implementation and fairly high for the knapsack implementation. We have received compatible results from experiments with a DEC Alpha 3000 cluster.

Our work is inspired by Brinch Hansen [4] who first described a generic Monte-Carlo parallel algorithm for multi-computers, then used it to develop a simulated annealing algorithm for the traveling salesperson problem and a primality testing algorithm. The Monte-Carlo algorithm is targeted to multi-computers and builds a pipeline of processes - a structure that exploits existing direct processor connections. A pipeline cannot be mapped efficiently on clusters of workstations, because the connections may have diverse topology, or could even be inaccessible to the programmer. Our proposed master-server process configuration eliminates this problem by leaving the control of the physical connections to the run-time system, e.g. PVM in our implementations. Our master-server algorithm is more general than the pipelined Monte-Carlo algorithm because the former provides multiple interactions between the master and the server processes, while the latter sends work to the pipelined processes only once. Therefore, the Monte-Carlo algorithm cannot be used to derive evolutionary algorithms, which require multiple interactions between a master process on the one end, and server processes that perform the actual computations on the other.

Our parallel evolutionary algorithm for the 0/1 knapsack problem uses extensive parallel hill-climbing and limited serial crossover. This is in contrast to typical genetic algorithms, which employ extensive crossover and limited mutations. This difference is essential for parallel execution because hill-climbing is independent for individual chromosomes, while crossover requires coordination of the whole population [2]. Besides, experiments with an

earlier version of the knapsack evolutionary algorithm have demonstrated significantly better optimization performance than genetic algorithms.

Parallel probabilistic algorithms have found diverse applications in computational biology [14], inference [6], simulation [16], sorting [15], prefix computation [11]. Research in theory of probabilistic algorithms [12, Ch. VII] has been concentrated on algorithms that solve problems exactly for poly-logarithmic expected running time. Probabilistic algorithms from this class have been proposed for a variety of problems, such as parallel prefixes, sorting, maximal matching and maximal independent sets of vertices in undirected graphs, matrix rank and normal form computations. The disadvantage of this class of algorithms is that their running time is indeterminate and, therefore, their worst cases can be very bad.

7. References

1. Aarts E. and J. Korst. *Simulated Annealing and Boltzmann Machines*, John Wiley, Chichester, 1989.
2. Dorigo M. et al. Parallel genetic algorithms: Introduction and overview of current research, In: *Parallel Genetic Algorithms*, J. Stender, editor, IOS Press, Amsterdam, 1993, 5-42.
3. Fishman G. S.. *Monte Carlo: Concepts, Algorithms, and Applications*, Springer Verlag, New York, 1996.
4. Hansen B. *Studies in Computational Science: Parallel Programming Paradigms*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1995.
5. Hansen B. SuperPascal - a publication language for parallel scientific computing, *Concurrency - Practice and Experience*, 6, No 5, 1994, 461-483.
6. Kozlov A. et al. Parallel Implementation of Probabilistic Inference, *Computer, 29, No 12*, 1996, 33-40.
7. Lawler E. et al. *The Traveling Salesperson Problem: A Guided Tour of Combinatorial Optimization*, John Wiley, Chichester, 1985.
8. Martello S. and P. Toth. *Knapsack Problems*, John Wiley, Chichester, 1990.
9. Michalewicz Z. *Genetic Algorithms + Data Structures = Evolution Programs*, 2nd ed., Springer-Verlag, Berlin, 1994.
10. Probabilistic Master-Server Algorithms, ftp://saturn.cs.wssu.edu, directory pub/paradigms/mastserv/, 1997.
11. Reif J. H. Probabilistic Parallel Prefix Computation, *Computers and Mathematics with Applications*, 26, No 1, 1993, 101-110.
12. Smith J. R.. *The design and Analysis of Parallel Algorithms*, Oxford University Press, New York, 1993.
13. Sunderam V. PVM: A Framework for Parallel Distributed Computing, *Concurrency: Practice and Experience, 2, No 4*, 1990, 315-339.
14. Thompson E. et al. The Gibbs Sampler On Extended Pedigrees: Monte Carlo Methods For The Genetic Analysis Of Complex Traits, Tech. Report 193, Department of Statistics, University of Washington, 1990.
15. Vitter J.S. et al. Algorithms for Parallel Memory II: Hierarchical Multilevel Memories, Tech. Report CS-92-05, Department of Computer Science, Brown University, 1992.
16. Yoshimura S. et al. Life Extension Simulation Of Aged Reactor Pressure Vessel Material Using Probabilistic Fracture Mechanics Analysis On A Massively Parallel Computer, *Nuclear Engineering and Design, 158, No 2-3*, 1996, 341-350.