6-1992

Parallel Monte Carlo Trials

Per Brinch Hansen
Syracuse University, School of Computer and Information Science, pbh@top.cis.syr.edu

SU-CIS-92-11

Parallel Monte Carlo Trials

Per Brinch Hansen

June 1992

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, NY 13244-4100

Parallel Monte Carlo Trials [1]

PER BRINCH HANSEN

School of Computer and Information Science
Syracuse University
Syracuse, New York 13244, U.S.A.

June 1992

SUMMARY

The best results of Monte Carlo methods are generally obtained by performing the same computation many times with different random numbers. We develop a generic algorithm for parallel execution of Monte Carlo trials on a multicomputer. The generic algorithm has been adapted for simulated annealing and primality testing by simple substitutions of data types and procedures. The performance of the parallel algorithms was measured on a Computing Surface.

KEY WORDS Multicomputer algorithms Monte Carlo trials
Simulated annealing Primality testing
Computing Surface

INTRODUCTION

Monte Carlo methods are algorithms that use random number generators to simulate stochastic processes. Probabilistic algorithms have been applied successfully to combinatorial problems, which cannot be solved exactly because they have a vast number of potential solutions.

The most famous example is the problem of the *traveling salesperson* who must visit n cities. No computer will ever be able to find the shortest tour through 100 cities by examining all the 5×10^{150} possible tours. For practical purposes, the problem can effectively be solved by *simulated annealing* [1, 2]. This Monte Carlo method has a high probability of finding a near-optimal tour of 100 cities after examining a random sample of one million tours. The chance of finding a good solution can be increased by selecting the shortest tour found in, say, 10 annealing runs.

Another example is *primality testing* of large integers. It is not feasible to determine whether or not a 150-digit integer is a prime by examining all the 10^{75} possible divisors. In practice, this problem can also be solved by Monte Carlo trials. The

Miller-Rabin algorithm tests the same integer many times using different random numbers [3, 4]. If any one of the trials shows that a number is composite, then this is the correct answer. However, if all trials fail to prove that a number is composite, then it is almost certainly prime. The probability that the algorithm gives the wrong answer after, say, 40 trials is less than 10^{-24}.

Simulated annealing and primality testing illustrate a general characteristic of Monte Carlo methods: Due to the probabilistic nature of the algorithms, the best results are obtained by performing the computation many times with different random numbers.

The advantage of using a multicomputer for *Monte Carlo trials* is obvious [5]. When the same problem has been broadcast to every processor, the trials can be performed simultaneously without any communication between the processors. Consequently, the processor efficiency is very close to 1 for nontrivial problems.

We will develop a generic algorithm for parallel Monte Carlo trials on a multicomputer. We will then show how the generic algorithm can be adapted for simulated annealing and primality testing by simple substitutions of data types and procedures. The performance of the parallel algorithms has been measured on a *Computing Surface.*

SEQUENTIAL PARADIGM

We assume that a Monte Carlo method is defined by a Pascal procedure of the form

```
procedure solve(a: problem; var b: solution;
  seed: integer)
```

The procedure parameters define a problem a of some type, a solution b of some type, and an initial *seed* of a random number generator. Since the procedure and its parameter types vary from one Monte Carlo method to another, we deliberately leave them unspecified at this point.

Algorithm 1 is a sequential paradigm for Monte Carlo trials. The same problem is solved m times using the trial numbers $1, 2, \ldots, m$ as distinct initial seeds. The m solutions are collected in a table.

```
const m = ...{trials};
type table = array [1..m] of solution;

procedure compute(a: problem; var b: table);
var trial: integer;
begin
  for trial := 1 to m do
    solve(a, b[trial], trial)
end
```

Algorithm 1

PARALLEL PARADIGM

We will rewrite Algorithm 1 for a multicomputer. In theory this is an easy task: First, we broadcast the same problem to p processors. Each processor then performs m/p trials. Finally, we collect m solutions from the processors. To simplify things a bit we will assume that the number of trials m is divisible by the number of processors p. The local *quota* $q = m/p$ is the number of trials per processor.

A straightforward implementation of this scheme requires a master processor that communicates directly with p server processors. Unfortunately, present multicomputers only permit each processor to communicate with a few neighboring processors. For p larger than, say 4, the data must be transmitted through a chain of processors. The simplest way to do this is to use a *pipeline* with p *nodes* controlled by a *master* process (Fig. 1).

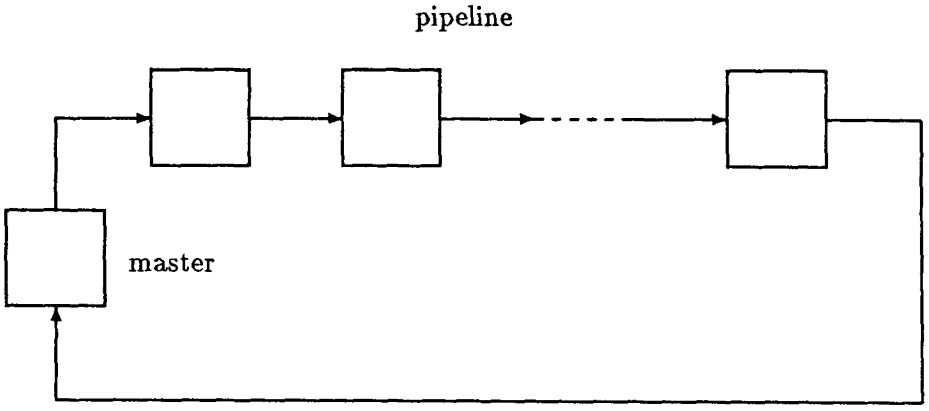

Fig. 1 Master and pipeline.

The parallel processes will be programmed in Pascal extended with statements for parallel execution and message communication.

The execution of k statements $S_1, S_2, \ldots, S_k$ as *parallel processes* is denoted

parbegin $S_1|S_2|\ldots|S_k$ **end**

The parallel execution continues until every one of the k processes has terminated.

The *parallel for* statement

parfor i := 1 **to** k **do** S(i)

is equivalent to

parbegin S(1) | S(2) |...| S(k) **end**

We assume that parallel processes run on separate processors without shared memory. Parallel processes communicate through synchronous *channels* only.

Each process has an input channel and an output channel. The *input* and *output* of a value a are denoted

inp?a out!a

The *master* process outputs a problem a to the pipeline and inputs m solutions from the pipeline (Algorithm 2).

```
procedure master(a: problem; var b: table;
  inp, out: channel);
var trial: integer;
begin
  out!a;
  for trial := 1 to m do inp?b[trial]
end
```

Algorithm 2

The m trials are distributed as follows among the p nodes:

node	first trial	last trial
1	1	q
2	q+1	2q
...	...	...
i	(i−1)q+1	iq
...	...	...
p	(p−1)q+1	pq (=m)

Pipeline node number i goes through two major phases:

1) *Broadcasting phase*: The node inputs a problem a. If the node is followed by another node, it outputs the problem to its successor:

```
inp?a;
if i < p then out!a
```

2) *Trial phase*: The node solves the same problem q times:

```
q := m div p;
for j := 1 to q do
begin
  solve(a, b, (i - 1)*q + j);
  collection phase
end
```

After every iteration each node holds a single solution. The nodes now go through a minor phase:

2.1) *Collection Phase*: Each node outputs its most recent solution b and copies the most recent solutions produced by its predecessors in the pipeline:

```
out!b;
for k := 1 to i − 1 do
begin inp?b; out!b end
```

In short, the pipeline outputs p solutions at a time to the master.

Together these pieces define the complete algorithm for a pipeline node (Algorithm 3).

```
procedure node(i: integer; inp, out: channel);
var a: problem; b: solution; j, k, q: integer;
begin
  inp?a;
  if i < p then out!a;
  q := m div p;
  for j := 1 to q do
  begin
    solve(a, b, (i − 1)*q + j);
    out!b;
    for k := 1 to i − 1 do
    begin inp?b; out!b end
  end
end
```

Algorithm 3

Algorithm 4 defines the parallel network shown in Fig. 1.

```
const m = ...{trials};
  p = ...{processors};
type table = array [1..m] of solution;

procedure compute(a: problem; var b: table);
type net = array [0..p] of channel;
var c: net; i: integer;
begin
  parbegin
    master(a, b, c[p], c[0]) |
    parfor i := 1 to p do
      node(i, c[i − 1], c[i])
  end
end
```

Algorithm 4

We have developed a generic algorithm for parallel execution of Monte Carlo trials on a multicomputer (Algorithms 2–4). We will now use this paradigm to solve two different problems.

SIMULATED ANNEALING

The first Monte Carlo method we will consider is simulated annealing. In [2] we discussed simulated annealing and illustrated the method by a sequential Pascal algorithm for the traveling salesperson's problem:

```
procedure anneal(var a: tour;
  Tmax, alpha: real; steps,
  attempts, changes: integer)
```

This procedure replaces an initial random tour of n cities by a near-optimal tour.

Parallel annealing trials can be implemented by making the following substitutions in Algorithms 2–4:

1. The problem and solution types are both replaced by

```
const n = ...{cities};
type city = record x, y: real end;
  tour = array [1..n] of city;
```

2. The solution procedure is replaced by

```
procedure solve(a: tour; var b: tour;
  seed: integer);
begin
  initialize(seed); b := a;
  anneal(b, sqrt(n), 0.95,
    trunc(20*ln(n)), 100*n, 10*n)
end
```

The programming details are explained in [2].

We reprogrammed simulated annealing trials in occam for a Computing Surface with T800 transputers. Table I shows the run time T_p for parallel computation of 10 different tours of 400 cities on 1, 5, and 10 transputers.

Table I

p	T_p s	E_p
1	8367.2	1.0000
5	1674.8	0.9992
10	838.3	0.9981

The processor efficiency $E_p = T_1/(pT_p)$ is very close to 1.

PRIMALITY TESTING

Our second Monte Carlo method is primality testing of a large integer p. (This integer should not be confused with the number of processors, which is also called p.)

We discussed the Miller-Rabin algorithm in [4] and illustrated it by a sequential Pascal function

```
function witness(x, p: integer): boolean
```

The boolean value of this function defines whether or not an integer x is a *witness* to the compositeness of the integer p.

Parallel primality trials can be implemented by making the following substitutions in Algorithms 2–4:

1. The problem type is replaced by type integer.
2. The solution type is replaced by type boolean.
3. The solution procedure is replaced by

```
procedure solve(p: integer; var sure: boolean;
  seed: integer);
begin
  initialize(seed);
  sure := witness(random(1, p − 1), p)
end
```

In practice, the procedure must be reprogrammed to perform *multiple-length arithmetic* on large integers represented by arrays of digits:

```
type number = array [...] of integer
```

Parallel primality trials were also tested on a Computing Surface. Table II shows the run time T_p for 40 primality tests of a random integer with 160 decimal digits. (Here p denotes the number of processors.) The trials were performed in parallel on 1, 10, 20, and 40 transputers.

Table II

p	T_p s	E_p
1	1503.3	1.0000
10	150.4	0.9995
20	75.2	0.9995
40	37.7	0.9969

The processor efficiency E_p is practically 1.

FINAL REMARKS

This is one of several papers that demonstrate the benefits of writing generic parallel algorithms, which can be adapted to different applications [6, 7, 8]. In this case, we developed a generic algorithm for parallel execution of Monte Carlo trials on a multicomputer. We then modified this algorithm to solve two different problems: Finding a near-optimal tour of n cities by simulated annealing, and testing the primality of an n-digit integer by the Miller-Rabin method [2, 4]. The parallel algorithms were implemented in occam for a Computing Surface. The processor efficiency was very close to 1.

I thank Jonathan Greenfield for constructive suggestions.

REFERENCES

1. S. Kirkpatrick, C. D. Gelatt and M. P. Vechi, 'Optimization by simulated annealing', *Science, 220*, 671–680 (1983).

2. P. Brinch Hansen, 'Simulated annealing', School of Computer and Information Science, Syracuse University, Syracuse, NY, 1992.

3. M. O. Rabin, Probabilistic algorithms for testing primality, *Journal of Number Theory, 12*, 128–138 (1980).

4. P. Brinch Hansen, 'Primality testing', School of Computer and Information Science, Syracuse University, Syracuse, NY, 1992.

5. S. Ulam, *Science, Computers & People: From the Tree of Mathematics*, Birkhauser, Boston, MA, 1986.

6. P. Brinch Hansen, 'The all-pairs pipeline', School of Computer and Information Science, Syracuse University, Syracuse, NY, 1990.

7. P. Brinch Hansen, 'A generic multiplication pipeline', School of Computer and Information Science, Syracuse University, Syracuse, NY, 1991.

8. P. Brinch Hansen, 'Parallel divide and conquer', School of Computer and Information Science, Syracuse University, Syracuse, NY, 1991.