An Analytic Model for Colluding Processes

Stephen M. Watt
University of Western Ontario
London, Canada
www.csd.uwo.ca/~watt

Abstract—**We develop a quantitative framework in order to understand how OR parallelism can be used to reduce execution times. In this model, tasks may either succeed or fail and any one success completes the problem. We follow Hoare and call these tasks "colluding". We model the situation where tasks' execution times are not known in advance, but instead have some probability distribution of execution times. We show how expected serial and parallel execution times can be computed, and demonstrate how parallel execution can give a lower expected execution time than any serial order, even on a single processor. This model can be applied in domains, such as computer algebra, which use algorithms whose execution times cannot be readily predicted by examining the inputs.**

I. Introduction

Most of the work in parallel computing has focused on "AND" parallelism, where a number of tasks must all be performed and their results are assembled to give the final result. Another form of parallelism that occurs is "OR" parallelism, where the result of any one of a number of tasks is enough to solve the overall problem. This later situation may arise when there are a number of different methods to solve a problem, when there are a number of different starting points, or a number of parameter values that may be used. A more general situation also occurs, where some number of tasks must be completed, but not all of them. We consider OR parallelism here.

The most obvious useful setting for OR parallelism is a multiprocessor with each task allocated to a real processor. All tasks can run and the first one to complete gives the result and the others are be terminated. Another, less obvious, possibility is to run several tasks in parallel on a single processor and to take the first result. Several authors have noticed that this can give useful speedups in practice. We also note that this approach of interleaving computations is standard in theoretical work where one has to avoid non-halting computation.

In this article we present a model for analyzing the expected execution times of parallel computations. This model can be usefully applied when the running times of the tasks cannot be determined in advance, but some probability distribution of running times can be estimated. We show that if the precise running time of each task is not known in advance, or if tasks have some possibility of failing, then it can be the case that running the tasks in parallel on a single processor gives a smaller expected execution time than running them in any particular order sequentially. Interestingly, even when the probability distributions for all of the tasks are identical, it can be the case that running several in parallel yields a lower expected execution time than running them sequentially. In order to best understand the issues that pertain to OR parallelism, we restrict the investigation to its use on a single processor: How does running the tasks in parallel on a uniprocessor affect the execution time? Once this question has been addressed, then using a multiprocessor to exploit OR parallelism is not substantially different than using it to

The paper is organized as follows. Section II reviews the possible relations among parallel processes and draws attention to the notion of "colluding" processes, capturing the notion of OR parallelism. Section III presents a probabalistic model of task execution duration. Section IV introduces time-sharing into the execution model. Section V models collusion and Section VI analyzes the expected serial and parallel execution times for colluding processes. Section VII works an example to illustrate the concepts. Section VIII shows how even when all tasks' execution times have the same probability distribution, time sharing execution can still be beneficial. Finally, Section IX concludes the paper.

This work was part of an earlier study [1] that has not appeared elsewhere in the formal literature. The increased current interest in parallel computing suggests that the results may find use today, even though they are now 25 years old. We present them here with only minimal revision to stand independently.

II. Colluding Processes

Hoare has categorized the relationships between parallel processes based on the exchange of information between them [2]. We outline his classification here.

Disjoint processes are completely independent. They do not communicate and they do not share data. *Competing* processes also neither share data nor communicate, however, they do contend for resources such as disks and line printers. It is clear that on a uniprocessor these two forms of parallelism cannot lead to a speed up because the computations that must be performed for any task are independent of the results obtained by others. *Cooperating* processes are allowed to update common data but are not allowed to read it. Again, with this type of relationship between processes a certain amount of work must be done and the use of parallelism cannot possibly decrease the total execution time.

Communicating processes pass information between one another. This is done through shared variables which may be both updated and read or some other sort of message passing.

If several tasks are to be performed, it could be that the necessity of executing (or even completing, if execution has commenced) some of the tasks is determined by the results of other tasks. In this case, the order of execution will influence the total amount of work that is done. For example, one process may tell the others that it has "the result" and they may terminate.

A particular case of this would be processes which follow alternate strategies for attaining a common goal. Hoare calls such processes *colluding*. Colluding processes work together toward a common goal; when one process succeeds in accomplishing that which was desired, all the processes are terminated. The colluding processes may communicate to share partial results but aside from this the work spent on the processes which do not "succeed" is wasted.

We shall call tasks *collusive* if they may be executed as colluding processes. If, in performing one of a group of collusive tasks, the common goal is attained by a particular task, then we say that task *succeeds*. If in the execution of a group of collusive tasks, a task terminates without having attained the common goal, then we say that task *fails*.

If colluding processes can be exploited to give a decrease in computation time, then this fact will be of practical import only if there are real problems which take advantage of collusion. Kornfeld has reported timings in which colluding processes more than doubled the speed in a particular heuristic search program [3]. We give three broad problem categories which use collusion in essentially different ways:

- problems in which there are alternate methods for attaining the common goal
- problems for which there are several equally acceptable solutions
- divide and conquer problems.

We shall discuss each of these classes in turn.

Alternate Methods

In this category, a problem has a single ultimate goal and there is more than one known method of achieving it. In many problems it is not possible to determine which alternate method is the least expensive for given data without performing a costly analysis. The cost of the analysis may well outweigh the savings gained from using the most economical method.

This situation can occur if the operation to be performed can be done cheaply using special methods for certain types of input. This is illustrated in the following example.

a) ***Example:*** We consider two methods of computing the GCD of a pair of polynomials, each of which is much cheaper than the other under particular circumstances.

The first method is a generalization of Euclid's method for computing the GCD of integers, the subresultant PRS algorithm (see, for example, [4]). If the GCD of two polynomials is large, then this method finds it quickly, since only a few iterations are required. However, if the GCD of the polynomials is small, then this method becomes extremely costly due to the exponential growth of intermediate results.

The second method, the EZGCD finds the GCD of related polynomials in one variable and with coefficients in a finite field. From this GCD, the GCD of the original polynomials can be constructed [5]. The cost of this construction depends on the size of the final GCD — the larger the GCD the greater the cost.

Comparing these two methods we find the first is less costly when the GCD is large and the second is less costly when the GCD is small. We may take advantage of collusion in the following way: To compute the GCD of two polynomials, two processes are used — one using each method. When either of the processes produces the GCD, the goal has been attained. (On a uniprocessor, it would be desirable for one of the methods to give up gracefully when it discovered a problem was not one of its good cases, otherwise the parallel algorithm could take twice as long as the quicker method.)
□

Alternate Goals

Another class of problems well suited for collusion are those for which there are equally acceptable different solutions. An example of this would be to find a divisor of a large integer — any integer that exactly divides the given number is as good as any other. A more detailed example is the "satisficing search" problem which is analyzed later in this chapter.

Divide and Conquer

The *divide and conquer* approach is to divide a problem into subproblems, solve the subproblems, and combine the results [6]. In some divide and conquer algorithms the subproblems contribute different amounts toward the final solution, depending on the problem instance. This type of problem can use collusion in situations where not all the subproblems' results are needed to determine the final answer. In cases such as this, OR parallelism is used in conjunction with AND parallelism.

III. Execution Duration

The first step in building our mathematical model is to incorporate the expected execution duration for tasks. Exactly what do we mean by the "expected" duration? It is clear that any given program with particular input data will either require a certain fixed amount of execution time, if it terminates, or it will require an infinite amount of execution time, if it does not. However, even when a task is guaranteed to halt, to determine the exact time needed for execution may be just as costly as performing the execution in the first place. Therefore, it is not reasonable to assume that, in practice, the execution time required can be known prior to performing the task. Rather than assigning to each task an exact assessment of the required time, we shall treat it as a random variable, based on the behavior of the program over many inputs.

Both in theoretical models of computation and in real machines, computation proceeds in discrete steps. To perform a serious computation takes very many basic machine operations. In view of this, we can simplify the calculations that arise in using our model by taking execution time to be a continuous, rather than a discrete, variable.

To each task we will assign a probability density for the execution time. The choice of the density will be based on the overall behavior of the algorithm for the domain of input. From this density, we get the expected execution time.

*b) **Example**:* Consider the following Pascal procedure:

```
procedure action(x: real);
var k : integer;
begin
    k := trunc(1000 * sin(x)) mod 100;
    if k <> 37 then
        subaction1(x);
    subaction2(x)
end
```

It can be seen that with uniform random x the routine *subaction1* is called for roughly ninety-nine out of every one hundred inputs, while the routine *subaction2* is always called. Suppose that the computation of k takes time T_k and that the routines *subaction1* and *subaction2* take times T_1 and T_2 respectively. Then roughly 1% of the valid inputs will take time $T_k + T_2$ and the remaining 99% will take time $T_K + T_1 + T_2$.

If the inputs to this routine are uniformly distributed in the input domain, then the probability density for execution time is

$$p(t) = .01\ \delta(T_k + T_2 - t) + .99\ \delta(T_k + T_1 + T_2 - t)\ \ .$$

Here δ is the Dirac delta function, defined by

$$\delta(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{ikx} \mathrm{d}k$$

and having the properties

$$\begin{aligned} \delta(x - x') &= 0, \quad \text{if } x \neq x' \\ \int_b^a \delta(x - x')\, \mathrm{d}x' &= \begin{cases} 0, & \text{if } x < a \text{ or } b < x \\ 1, & \text{if } a < x < b \end{cases} \end{aligned}$$

□

As in the above example, it is sometimes possible that the input domain can easily be seen to be divided into a number of disjoint subsets where the time required for an instance of the problem depends only on the subset to which the input belongs. In fact, for *any* procedure the input domain may be partitioned based on the criterion of execution duration; there exists a partition of the input domain into classes such that all the elements (i.e. inputs) in a particular class require the same amount of execution time. As noted before, sufficiently analyzing a given element of the input domain to determine this membership may be exactly as costly as performing the operation. Instead of examining each input, we assign a weight to each of the classes in the partition. This gives the probability density for the execution time. The assignment of weights can be done either theoretically through analysis of the application, or empirically through simulation or collection of data on actual usage.

IV. Parallelism

There is a broad range of possible degrees of parallelism in executing processes for a set of tasks on a uniprocessor. At one extreme, we could execute the tasks completely serially (no parallelism at all). Another possibility would be to give all the tasks an equal share of the available processing time (complete parallelism). In the general case, it must be decided for each time interval what portion of the processor time each process should receive. This budgeting of time can be done either statically, before execution begins, or dynamically, with the time allotments based on the processes' dynamic behavior.

For simplicity of the model we shall use static time allotment. We do this by assigning a *time allotment function* $\nu_i(t)$, to each task $\mathbf{T}_i$. The function $\nu_i(t)$ has as a value the amount of processor time that the process for task $\mathbf{T}_i$ will have received after a total time t has passed. Suppose that the set of tasks to be executed is $\{\mathbf{T}_1, ..., \mathbf{T}_n\}$. The $\nu_i(t)$ may be any non-decreasing functions such that

$$\sum_{i=1}^{n} \nu_i(t) \leq t$$

and

$$\nu_i(0) = 0.$$

We allow the inequality in the definition so that overhead may be accounted for, if desired. So far, what we have said about the functions $\nu_i(t)$ allows the time variable to be either discrete or continuous. We will use a continuous variable for time. In this case, the derivative $\nu'_i(t)$ indicates the instantaneous proportion of the processor time which the process for task $\mathbf{T}_i$ is receiving at time t.

*c) **Example**:* If N tasks are all to receive an equal share of time, then we let

$$\nu_i(t) = \frac{t}{N}\ , \quad i = 1, ..., N.$$

□

*d) **Example**:* Suppose we have two tasks $\mathbf{T}_1$ and $\mathbf{T}_2$ which require times T_1 and T_2 to complete, respectively. If we execute task $\mathbf{T}_1$ and when it is done we execute task $\mathbf{T}_2$, then

$$\begin{aligned} \nu_1(t) &= \min(t, T_1) \\ \nu_2(t) &= \max(0, t - T_1)\ . \end{aligned}$$

□

V. Modelling Collusion

In our model of collusion we start with a set of tasks $\{\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_n\}$. The execution of each of the tasks can result in one of three possibilities:

1) It succeeds; the execution terminates and the remaining tasks need not be executed (or completed if execution has commenced).

2) It fails; the execution of the task terminates and the remaining tasks are unaffected.
3) It does not halt.

To each task $\mathbf{T}_i$ we assign two probability density functions, $p_i(t)$ and $q_i(t)$. The density $p_i(t)$ gives the probability that the task will succeed when it has consumed a total time t. The density $q_i(t)$ gives the probability that the task $\mathbf{T}_i$ will fail when it has consumed a total time t. If the execution of task $\mathbf{T}_i$ halts, then

$$\int_0^\infty [p_i(t) + q_i(t)]\ \mathrm{d}t = 1.$$

VI. Expected Execution Times

We now develop formulas for the expected completion times of groups of collusive tasks as we have modelled them. We first examine the time for serial execution of the tasks and then the time for parallel execution. After this, examples are given to compare the expected execution duration of serial versus parallel computations.

Collusive Tasks Executed Serially

We have a set of N collusive tasks that are to be executed one after another until one of them succeeds or until they all have failed. Let the tasks be labelled $\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_N$, according to the order in which they would be executed if none were to succeed. Now for each task $\mathbf{T}_i$ let $p_i(t)$ and $q_i(t)$ denote the probability densities for success and failure, respectively, as a function of the time consumed by the process for the task.

We will now derive formulas for the probability densities, over time, for success of any task or failure of all tasks in the group. Let $p_{1..m}(t)$ denote the probability that one of the first m tasks (i.e. $\mathbf{T}_1, ..., \mathbf{T}_m$) succeeds when a total time t has been spent on all the tasks together. Let $q_{1..m}(t)$ denote the probability that the m-th task fails at time t.

By definition, we have

$$\begin{aligned} p_{1..1}(t) &= p_1(t) \\ q_{1..1}(t) &= q_1(t)\ . \end{aligned} \quad (1)$$

One of the first m ($m \geq 2$) tasks succeeds at time t if either (i) one of the first $m-1$ of them succeeds at this time or (ii) all of the first $m-1$ tasks fail and the m-th task succeeds after consuming the remaining time to t. Thus,

$$p_{1..m}(t) = p_{1..m-1}(t) + \int_0^t q_{1..m-1}(t')\ p_m(t-t')\ \mathrm{d}t'.$$

If the m-th task fails at time t, the task $\mathbf{T}_{m-1}$ must have failed at some time prior to t. Then task $\mathbf{T}_m$ will have failed after consuming the time remaining to time t. We therefore have

$$q_{1..m}(t) = \int_0^t q_{1..m-1}(t')\ q_m(t-t')\ \mathrm{d}t'. \quad (2)$$

Taking Laplace transforms of (1) through (2), we obtain the expressions

$$\begin{aligned} \tilde{p}_{1..1}(s) &= \tilde{p}_1(s) \\ \tilde{q}_{1..1}(s) &= \tilde{q}_1(s) \\ \tilde{p}_{1..m}(s) &= \tilde{p}_{1..m-1}(s) + \tilde{q}_{1..m-1}(s) \cdot \tilde{p}_m(s) \\ \tilde{q}_{1..m}(s) &= \tilde{q}_{1..m-1}(s) \cdot \tilde{q}_m(s), \end{aligned}$$

where $\tilde{f}(s)$ denotes the Laplace transform of $f(t)$. Solving these recurrences and putting $m = N$, we have

$$\begin{aligned} \tilde{p}_{1..N}(s) &= \tilde{p}_1(s) + \tilde{q}_1(s) \cdot \tilde{p}_2(s) + \cdots \quad (3) \\ &\quad + \tilde{q}_1(s) \cdots \tilde{q}_{N-1}(s) \cdot \tilde{p}_N(s) \\ &= \sum_{i=1}^{N} \tilde{p}_i(s) \cdot \prod_{j=1}^{i-1} \tilde{q}_j\ (s) \\ \tilde{q}_{1..N}\ (s) &= \prod_{i=1}^{N} \tilde{q}_i\ (s). \quad (4) \end{aligned}$$

Taking the inverse transform gives the probability densities for $p_{1..N}(t)$ and $q_{1..N}(t)$.

The performance of the N collusive tasks will be complete under either one of two mutually exclusive conditions: (i) one of them has succeeded or (ii) they have all failed. The expected execution time is therefore

$$\langle t \rangle_{\mathrm{ser}} = \int_0^\infty t\ [p_{1..N}\ (t) + q_{1..N}\ (t)]\ \mathrm{d}t. \quad (5)$$

*e) **Example**:* Suppose we have two tasks, $\mathbf{T}_1$ and $\mathbf{T}_2$, with

$$\begin{aligned} p_i(t) &= a_i \lambda_i e^{-\lambda_i\ t} \\ q_i(t) &= (1-a_i) \lambda_i e^{-\lambda_i\ t} \end{aligned}$$

for $0 \leq a_i \leq 1$, $\lambda_i > 0$. This gives

$$\begin{aligned} \tilde{p}_i(s) &= \frac{\lambda_i\ a_i}{s+\lambda_i} \\ \tilde{q}_i(s) &= \frac{\lambda_i(1-a_i)}{s+\lambda_i} \end{aligned}$$

so that

$$\begin{aligned} \tilde{p}_{1..2}(s) &= \frac{\lambda_1\ a_1}{s+\lambda_1} + \frac{\lambda_1(1-a_1)}{s+\lambda_1} \cdot \frac{\lambda_2\ a_2}{s+\lambda_2} \\ \tilde{q}_{1..2}(s) &= \lambda_1 \lambda_2 \frac{(1-a_1)(1-a_2)}{(s+\lambda_1)(s+\lambda_2)} \end{aligned}$$

and

$$p_{1..2}(t) + q_{1..2}(t) = \\ a_1 \lambda_1 e^{-\lambda_1 t} - (1-a_1)\ \lambda_1\ \lambda_2 \frac{e^{-\lambda_1 t} - e^{-\lambda_2 t}}{\lambda_1 - \lambda_2}\ .$$

Then the expected execution time is

$$\langle t \rangle_{\mathrm{ser}} = \int_0^\infty t\ [p_{1..2}(t) + q_{1..2}(t)]\ \mathrm{d}t = \frac{1}{\lambda_1} + \frac{1-a_1}{\lambda_2}$$

□

Collusive Tasks Executed In Parallel

Here we have N collusive tasks $\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_N$, that are to be performed in parallel. The execution of these colluding processes will continue until one of them succeeds or until all of them have failed. As for the serial case, let $p_i(t)$ and $q_i(t)$ denote the probability densities for success and failure respectively with respect to the amount of time consumed by the process for task $\mathbf{T}_i$. Let $\nu_i(t)$ denote the time allotment function for task $\mathbf{T}_i$.

Ignoring the other processes for now, the probability that the process for task $\mathbf{T}_i$ succeeds before a total time t is spent (on all processes) is given by

$$P_i(t) = \int_0^{\nu_i(t)} p_i(t')\ \mathrm{d}t'\ . \tag{6}$$

Similarly, ignoring the other processes, the probability that the process for task $\mathbf{T}_i$ fails by time t is

$$Q_i(t) = \int_0^{\nu_i(t)} q_i(t')\ \mathrm{d}t'\ . \tag{7}$$

We now derive formulas for the probability densities for the success of any process or the failure of all processes. Let $P_*(t)$ denote the probability of success in any of the processes by time t and let $Q_*(t)$ denote the probability of failure of all the processes by time t.

The probability that a success occurs by a given time is given by

$$P_*(t) = P_1(t)\ \cup\ P_2(t) \cup\ \cdots\ \cup\ P_N(t)\ .$$

Here, the associative operator $\cup$ is the inclusive *or*, defined to be $a + b - ab$. That is, if a and b are probabilities, then $a \cup b$ is the probability of a or b or both. It is simple to prove

$$\bigcup_{i=1}^{n} a_i = 1 - \prod_{i=1}^{n} (1 - a_i)$$

Therefore the probability density for success at time, t, $p_*(t)$, is

$$\begin{aligned} p_*(t) &= \frac{d}{dt} P_*(t) \\ &= \frac{d}{dt} \left[1 - (1 - P_1(t))\ (1 - P_2(t)) \cdots (1 - P_N(t))\right] \end{aligned}$$

The probability that all tasks have failed by time t is

$$Q_*(t) = Q_1(t) \cdot Q_2(t) \cdots Q_N(t).$$

This directly gives us the probability density function for failure at time t, $q_*(t)$:

$$q_*(t) = \frac{d}{dt}\ Q_*(t) = \frac{d}{dt} \left[Q_1(t) \cdot Q_2(t) \cdots Q_N(t)\right].$$

The execution of the N colluding processes will be complete if one of the processes succeeds or if all of them have failed. These two conditions are mutually exclusive. The probability density for execution completion is $p_*(t) + q_*(t)$. The expected execution duration is therefore given by

$$\begin{aligned} \langle t \rangle_{\mathrm{par}} &= \int_0^{\infty} t\ (p_*(t) + q_*(t))\ \mathrm{d}t \\ &= \int_0^{\infty} t\ \frac{d}{dt}\ (P_*(t) + Q_*(t))\ \mathrm{d}t. \end{aligned} \tag{8}$$

Integrating by parts, we obtain the expression

$$\langle t \rangle_{\mathrm{par}} = \lim_{L \to \infty} t\ (P_*(t) + Q_*(t)) \Big|_0^L - \int_0^L (P_*(t) + Q_*(t))\ \mathrm{d}t. \tag{9}$$

*f) **Example**:* Suppose again that we have two tasks, $\mathbf{T}_1$ and $\mathbf{T}_2$, with

$$p_i(t) = a_i\ \lambda_i\ e^{-\ \lambda_i\ t} \qquad q_i(t) = (1 - a_i)\ \lambda_i\ e^{-\ \lambda_i\ t}$$

for $0 \leq a_i \leq 1$, $\lambda > 0$. For both tasks let the time allotment function be $\nu_i(t) = t/2$. Then we have

$$P_i(t) = a_i (1 - e^{-\lambda_i\ t/2}) \quad Q_i(t) = (1 - a_i)\ (1 - e^{-\lambda_i\ t/2})$$

which implies

$$\begin{aligned} P_*(t) + Q_*(t) \ &= \ 1 - (1 - a_1)\ e^{-\lambda_2 t/2} - (1 - a_2)\ e^{-\lambda_1 t/2} \\ &\quad + (1 - a_1 - a_2)\ e^{t(\lambda_1 + \lambda_2)/2} \end{aligned}$$

Using (9), the above expression yields

$$\langle t \rangle_{\mathrm{par}} = 2\ \left[\ \frac{1 - a_2}{\lambda_1} + \frac{1 - a_1}{\lambda_2} - \frac{1 - (a_1 + a_2)}{\lambda_1 + \lambda_2}\ \right].$$

□

VII. Example: Satisficing Search

Several types of search problems may be distinguished based on the aim of the search. In a *satisficing search* [7] there is a set of items, a subset of which have some particular property, and the goal of the search is to find any element of the set with that property. An example is the search for a block of storage in a *first-fit* storage allocation algorithm. Several classes of satisficing search are treated in the literature. In this section we model the type of satisficing search in which the items may be examined in any order. This is known as *unrestricted satisficing search* or *satisficing search without order constraints*.

We can model satisficing search without order constraints in the following way: for each element (e_i) there is a probability (p_i) that the element has the goal property and there is a fixed time (t_i) required to examine the element. If we have N elements, then we have the tasks $\mathbf{T}_1 ... \mathbf{T}_N$ of examining the elements $e_1, ..., e_N$, respectively.

From the above description, we see that the density functions for the probabilities of success and of failure with the *i-th* task are

$$p_i(t) = p_i\ \delta(t - t_i) \tag{10}$$

$$q_i(t) = \bar{p}_i\ \delta(t - t_i) \tag{11}$$

where δ is the Dirac delta function and $\bar{p}\ _i = 1 - p_i$.

We now find the expected time for serial and parallel execution of the tasks $\mathbf{T}_1, ..., \mathbf{T}_N$. Because the time dependence is given by delta functions, it is quite possible to derive the expected times using discrete methods. However, to illustrate the use of the formulas derived in the previous sections we will use the more general method.

Serial Execution

Taking Laplace transforms of (10) and (11), we obtain

$$\tilde{p}_i(s) = p_i\, e^{-\,st_i} \qquad \tilde{q}_i(s) = \bar{p}_i\, e^{-st_i}.$$

Using (3) and (4), we therefore have

$$\begin{aligned}\tilde{p}_{1..N}(s) &= p_1 e^{-st_1} + \bar{p}_1 p_2 e^{-s(t_1+t_2)} + \cdots \\ &\quad + \bar{p}_1 \cdots \bar{p}_{N-1} p_N e^{-s(t_1+\cdots+t_N)} \\ \tilde{q}_{1..N}(s) &= \bar{p}_1 \bar{p}_2 \cdots \bar{p}_N e^{-s(t_1+\cdots+t_N)}\end{aligned}$$

Taking the inverse Laplace transforms we find

$$\begin{aligned}p_{1..N}(t) &= p_1\delta(t_1 - t) + \bar{p}_1 p_2 \delta(t_1 + t_2 - t) + \cdots \\ &\quad + \bar{p}_1 \cdots \bar{p}_{N-1} p_N \delta(t_1 + \cdots + t_N - t) \\ q_{1..N}(t) &= \bar{p}_1\bar{p}_2 \cdots \bar{p}_N \delta(t_1 + \cdots + t_N).\end{aligned}$$

The expected execution time is therefore

$$\begin{aligned}\langle t\rangle_{\text{ser}} &= \int_0^\infty t\,(p_{1..N}(t) + q_{1..N}(t))\,\mathrm{d}t \\ &= p_1 t_1 + \bar{p}_1 p_2 (t_1 + t_2) + \cdots \\ &\quad + \bar{p}_1 \cdots \bar{p}_{N-1} p_N (t_1 + \cdots + t_N) \\ &\quad + \bar{p}_1 \cdots \bar{p}_N (t_1 + \cdots + t_N) \\ &= \sum_{i=1}^{N-1} \left[p_i \cdot \prod_{j=1}^{i-1} \bar{p}_j \cdot \sum_{j=1}^{i} t_j \right] + \prod_{j=1}^{N-1} \bar{p}_j \cdot \sum_{j=1}^{N} t_j\end{aligned}$$

Not surprisingly, the expected execution time depends on the values of the p's, the t's, and the order in which the tasks are executed. If we know the values for the p's and t's it is natural to ask in what order the tasks should be executed to minimize the expected processing time. This is known as the *least cost testing sequence* problem [8].

To solve this problem, consider the effect of exchanging the order of two tasks,$\mathbf{T}_k$ and $\mathbf{T}_{k+1}$. In the original order we have

$$\langle t\rangle_{\text{ser}_{k,k+1}} = A + \rho p_k(\tau + t_k) + \rho\bar{p}_k p_{k+1}(\tau + t_k + t_{k+1}) + B,$$

where

$$\begin{aligned}A &= \sum_{i=1}^{k-1} \left(p_i \cdot \prod_{j=1}^{i-1} \bar{p}_j \cdot \sum_{j=1}^{i} t_j \right) \\ B &= \sum_{i=k+2}^{N-1} \left(p_i \cdot \prod_{j=1}^{i} \bar{p}_j \cdot \sum_{j=1}^{i} t_j \right) + \prod_{j=1}^{N-1} \bar{p}_j \cdot \sum_{j=1}^{N} t_i \\ \rho &= \prod_{i=1}^{k-1} \bar{p}_i \qquad\qquad \tau = \sum_{i=1}^{k-1} t_i.\end{aligned}$$

With the interchange we have

$$\langle t\rangle_{\text{ser}_{k+1,k}} = A + \rho p_{k+1}(\tau + t_{k+1}) + \rho\bar{p}_{k+1} p_k(\tau + t_{k+1} + t_k) + B$$

Therefore

$$\begin{aligned}\langle t\rangle_{\text{ser}_{k,k+1}} - \langle t\rangle_{\text{ser}_{k+1,k}} &= \rho \Big[p_k(\tau + t_k) - p_{k+1}(\tau + t_{k+1}) \\ &\quad + \bar{p}_k p_{k+1}(\tau + t_k + t_{k+1}) \\ &\quad - \bar{p}_{k+1} p_k(\tau + t_{k+1} + t_k) \Big] \\ &= \rho\,[p_{k+1} t_k - p_k t_{k+1}]\end{aligned}$$

This shows that the task with the smaller value of t_i/p_i should be executed first. Since any permutation can be obtained from successive transpositions, the optimal order for sequentially executing the tasks will be $\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_N$ when

$$\frac{t_1}{p_1} \le \frac{t_2}{p_2} \le \cdots \le \frac{t_N}{p_N}\,.$$

If any of the ratios are in fact equal, then more than one ordering is optimal. This solution to the least cost testing sequence problem has been given by a number of authors, the earliest apparently being Mitten [9].

Parallel Execution

For simplicity, we shall use the time allotment function $\nu_i(t) = t/N$ for all processes. This is less than optimal, since once some of the tasks have failed there is more processor time available. However, taking advantage of this available time adds to the complexity of the analysis without significantly affecting the results.[1]

We label the tasks in such a way that

$$t_1 \le t_2 \le \cdots \le t_N \tag{12}$$

Now, using (10) in (6), we find that

$$P_i(t) = \int_0^{t/N} p_i\delta(u - t_i)\mathrm{d}u = p_i\mathbf{U}(t/N - t_i)$$

where $\mathbf{U}$ is the Heaviside unit step function. Similarly, using (11) in (7), we see

$$Q_i(t) = \bar{p}_i\mathbf{U}(t/N - t_i).$$

Therefore, we have

$$\begin{aligned}P_*(t) &= 1 - [1 - p_1\mathbf{U}(t/N - t_1)] \cdots [1 - p_N\mathbf{U}(t/N - t_N)] \\ Q_*(t) &= \bar{p}_1 \cdots \bar{p}_N \mathbf{U}(t/N - t_N),\end{aligned}$$

the latter justified by (12). The expected execution time is

$$\langle t\rangle_{\text{par}} = \lim_{L\to\infty} t(P_*(t) + Q_*(t))|_0^L - \int_0^L (P_*(t) + Q_*(t))\mathrm{d}t.$$

[1]In particular, we could use a model for time allocation where, after a process fails, the time formerly allocated to it is split equally amongst the remaining processes. Call this model M. We give the results for this model in footnotes for comparison.

When L exceeds Nt_N, we have

$$\begin{aligned}
\langle t\rangle_{\text{par}} &= \lim_{L\to\infty} L(1-\bar p_1\cdots\bar p_N+\bar p_1\cdots\bar p_N)\\
&\quad -\int_0^{Nt_1}(1-1)\mathrm{d}t-\int_{Nt_1}^{Nt_2}(1-\bar p_1)\mathrm{d}t\\
&\quad -\int_{Nt_2}^{Nt_3}(1-\bar p_1\bar p_2)\mathrm{d}t-\cdots\\
&\quad -\int_{Nt_{N-1}}^{Nt_N}(1-\bar p_1\cdots\bar p_{N-1})\mathrm{d}t\\
&\quad -\int_{Nt_N}^{L}(1-\bar p_1\cdots\bar p_N+\bar p_1\cdots\bar p_N)\mathrm{d}t\\
&= Nt_1p_1+Nt_2\bar p_1p_2+\cdots\\
&\quad +Nt_N\bar p_1\cdots\bar p_{N-1}p_N+Nt_N\bar p_1\cdots\bar p_N
\end{aligned}$$

This may be written as[2]

$$\langle t\rangle_{\text{par}}=\sum_{i=1}^{N-1}\left(p_i\cdot\prod_{j=1}^{i-1}\bar p_j\cdot Nt_i\right)+\prod_{j=1}^{N-1}\bar p_j\cdot Nt_N \tag{13}$$

Comparison

We now compare the expected execution duration for serial and parallel execution of a satisficing search. First, we demonstrate that a non-optimal ordering of serial execution can have an expected execution time *greater* than for parallel execution. Then we show that the optimal order for serial execution gives an expected execution time *less* than for parallel execution.[3]

We show that a non-optimal ordering of serial execution may be expected to require more time than a parallel execution by giving a simple example. Consider the case where $N=2$ and $t_1=t, t_2=10t, p_1=p_2=39/40$. The expected time required for parallel execution is less than $5t/2$ while the expected time for serial execution is greater than $10t$ if task $\mathbf{T}_2$ is performed before $\mathbf{T}_1$.

To show that *optimal* serial execution is better than parallel execution, for this problem we assume that the time required for each task is greater than zero and that each task has a non-zero probability of success. Let the tasks be labelled $\mathbf{T}_1,\mathbf{T}_2,...,\mathbf{T}_N$ so that $t_1\le t_2\le ...\le t_N$. We define the following notation:

P_k = the expected time to execute $\mathbf{T}_1,...,\mathbf{T}_k$ in parallel;
$\tilde S_k$ = the expected time to execute $\mathbf{T}_1,\mathbf{T}_2,\mathbf{T}_3,...,\mathbf{T}_k$ serially and in that order;
S_k^0 = the expected time to execute $\mathbf{T}_1,...,\mathbf{T}_k$ serially in an optimal order.

Since $S_k^0\le\tilde S_k$, we have

$$P_k-S_k^0\ge P_k-\tilde S_k.$$

Therefore, we can prove P_N is greater than S_N^0 by showing $P_N-\tilde S_N>0$. This we do by induction.

[2]For model M, replace Nt_k with $\sum_{i=1}^{k-1}t_j+(N-k+1)t_k$ in (13).
[3]Both these results still hold if parallel execution is based on model M.

For the basis of the induction, consider the case when $N=2$:

$$\begin{aligned}
P_2 &= 2(t_1p_1+t_2\bar p_1)\\
\tilde S_2 &= p_1t_1+\bar p_1(t_1+t_2).
\end{aligned}$$

Subtracting, we find

$$P_2-\tilde S_2=p_1t_1+\bar p_1(t_2-t_1)>0.$$

For the inductive step, we note that

$$\begin{aligned}
P_N &= \frac{N}{N-1}P_{N-1}+N\bar p_1\cdots\bar p_{N-1}(t_N-t_{N-1})\\
\tilde S_N &= \tilde S_{N-1}+t_N\bar p_1\cdots\bar p_{N-1}
\end{aligned}$$

Taking the difference and using (13), we see that

$$\begin{aligned}
P_N-\tilde S_N &= (P_{N-1}-\tilde S_{N-1})+\frac{P_{N-1}}{N-1}\\
&\quad +\bar p_1\cdots\bar p_{N-1}\big((N-1)t_N-Nt_{N-1}\big)\\
&> (P_{N-1}-\tilde S_{N-1})+\ \bar p_1\cdots\bar p_{N-1}\,t_{N-1}\\
&\quad +\bar p_1\cdots\bar p_{N-1}\,((N-1)\,t_N-N\,t_{N-1})\\
&= (P_{N-1}-\tilde S_{N-1})\\
&\quad +(N-1)\bar p_1\cdots\bar p_{N-1}(t_N-t_{N-1}).
\end{aligned}$$

So, for all $N\ge 2$,

$$P_N-S_N^0\ \ge\ P_N-\tilde S_N\ >\ 0.$$

This completes the proof.

VIII. The Existence of Extremely Collusive Densities

In the previous sections we have shown how to determine the expected times for serial and parallel execution of collusive tasks running on a uniprocessor. The expressions derived in these sections are quite general, allowing for each task to have a distinct probability density. Quite often, however, we can expect the tasks to share the same probability density, while maintaining independence. This leads us to consider the following question:

Given a number of collusive tasks with a common probability density for execution time, can the expected time for performing the tasks in parallel be *less* than the expected time for performing the tasks serially, *even on a uniprocessor*?

We define an *extremely collusive density* to be one for which the expected time of two tasks using parallel execution on a single processor is less than for serial execution. In this section, we show that densities with this property do indeed exist.

We demonstrate the existence of extremely collusive densities by explicitly exhibiting an example.

Let us begin with a problem for which N identical tasks must be performed in the worst case. Call these tasks $\mathbf{T}_1...\mathbf{T}_N$. With each task, $\mathbf{T}_i$, we associate a probability (p_i) that it may find the solution to the overall problem and a time (t_i) which it would take to do this. We also associate with each task the time (T_i) that would be required to compute a partial result if it does not solve the entire problem. For each task, we assume that $T_i > t_i$. :p. From this description, we see that the probability distribution functions for success and for "failure" are, respectively given by

$$\begin{aligned} p_i(t) &= p_i\delta(t-t_i) && (14)\\ q_i(t) &= \bar{p}_i\delta(t-T_i). && (15)\end{aligned}$$

In our analysis we will ignore the time required to determine the appropriate subproblems and to combine the results. This is not because the time is necessarily negligible, but because it is exactly the same regardless of whether the tasks are executed serially or in parallel.

We now find the expected time for serial and for parallel execution of the tasks. After this we compare the results for the special case when the tasks share the same distribution. Doing this we find that for certain ranges of p_i and t_i/T_i these densities are extremely collusive.

g) Serial Execution: The Laplace transforms of (14) and (15) are

$$\tilde{p}_i(s) = p_i e^{-st_i} \qquad \tilde{q}_i(s) = \bar{p}_i e^{-sT_i}.$$

Using formulas (3) and (4), this gives us

$$\begin{aligned} \tilde{p}_{1..N}(s) &= p_1 e^{-st_1} + \bar{p}_1 p_2 e^{-s(T_1+t_2)} + \cdots \\ &\quad + \bar{p}_1\cdots\bar{p}_{N-1}p_N e^{-s(T_1+\cdots+T_{N-1}+t_N)} \\ \tilde{q}_{1..N}(s) &= \bar{p}_1\cdots\bar{p}_N e^{-s(T_1+\cdots+T_N)}.\end{aligned}$$

Taking the inverse Laplace transforms yields

$$\begin{aligned} p_{1..N}(t) &= p_1\delta(t_1-t) + \bar{p}_1p_2\delta(T_1+t_2-t)+\cdots \\ &\quad + \bar{p}_1\cdots\bar{p}_{N-1}p_N\delta(T_1+\cdots+T_{N-1}+t_N-t) \\ q_{1..N}(t) &= \bar{p}_1...\bar{p}_N\delta(T_1+\cdots+T_N-t).\end{aligned}$$

This gives an expected execution time of

$$\begin{aligned} \langle t\rangle_{\rm ser} &= p_1t_1 + \bar{p}_1p_2(T_1+t_2)+\cdots \\ &\quad + \bar{p}_1\cdots\bar{p}_{N-1}p_N(T_1+\cdots+T_{N-1}+t_N) \\ &\quad + \bar{p}_1...\bar{p}_N(T_1+\cdots+T_N) \\ &= \sum_{i=1}^{N}\left[p_i\cdot\prod_{j=1}^{i-1}\bar{p}_j\cdot\left(t_j+\sum_{j=1}^{i-1}T_j\right)\right] \\ &\quad + \prod_{j=1}^{N}\bar{p}_j\cdot\sum_{j=1}^{N}T_j. && (16)\end{aligned}$$

As before, it is natural to ask in what order the tasks should be executed to minimize the expected execution time. Using a method similar to that employed in section 3.6, we find that the optimal ordering of the tasks is to have

$$\phi(1)\le\phi(2)\le...\le\phi(N),$$

where

$$\phi(i) = (t_i - T_i) + \frac{T_i}{p_i}.$$

h) Parallel Execution: Again, for simplicity, we shall use the time allotment function $\nu_i(t) = t/N$ for all processes. Then, using (14) and (15) in (6) and (7), we have

$$\begin{aligned} P_i(t) &= p_i\mathbf{U}(\frac{t}{N}-t_i) \\ Q_i(t) &= \bar{p}_i\mathbf{U}(\frac{t}{N}-T_i)\end{aligned}$$

Without loss of generality, let $t_1\le t_2\le\cdots\le t_N$. Also let

$$T_{MAX} = \max(T_1,...,T_N).$$

Then we have

$$\begin{aligned} P_*(t) &= 1-[1-p_1\mathbf{U}(\frac{t}{N}-t_1)]\cdots[1-p_N\mathbf{U}(\frac{t}{N}-t_N)] \\ Q_*(t) &= \bar{p}_1\cdots\bar{p}_N\mathbf{U}(\frac{t}{N}-T_{MAX}).\end{aligned}$$

Assuming $L > NT_{MAX}$, the expected execution time is

$$\begin{aligned} \langle t\rangle_{\rm par} &= \lim_{L\to\infty} t\,(P_*(t)+Q_*(t))\,|_0^L \\ &\quad -\int_{NT_{MAX}}^{L} 1\cdot{\rm d}t - \int_{Nt_N}^{NT_{MAX}}[P_*(t)+Q_*(t)]{\rm d}t \\ &\quad -\int_0^{Nt_1}P_*(t){\rm d}t - \int_{Nt_1}^{Nt_2}P_*(t){\rm d}t - \cdots \\ &\quad -\int_{Nt_{N-1}}^{Nt_N}P_*(t){\rm d}t \\ &= \lim_{L\to\infty}\ L - [1]_{NT_{MAX}}^{L} - [1-\bar{p}_1]_{Nt_1}^{Nt_2} - \cdots \\ &\quad -[\,1-\bar{p}_1\cdots\bar{p}_{N-1}\,]_{Nt_{N-1}}^{Nt_N} \\ &\quad -[\,1-\bar{p}_1\cdots\bar{p}_N\,]_{Nt_N}^{NT_{MAX}} \\ &= N\,t_1\,p_1 + N\,t_2\,\bar{p}_1\,p_2+\cdots \\ &\quad +N\,t_N\,\bar{p}_1\cdots\bar{p}_{N-1}\,p_N + NT_{MAX}\,\bar{p}_1\cdots\bar{p}_N.\end{aligned}$$

This may be expressed as

$$\langle t\rangle_{\rm par} = \sum_{i=1}^{N}\left(p_i\,\cdot\,\prod_{j=1}^{i-1}\bar{p}_j\,\cdot\,Nt_i\right)+\prod_{j=1}^{N}\bar{p}_j\,\cdot\,NT_{MAX}. \qquad (17)$$

i) Comparison: We now compare the expected serial execution time to the expected parallel execution time for the special case of this example where

$$\begin{array}{ccccccccc} p_1 &=& p_2 &=& \cdots &=& p_N &=& p \\ t_1 &=& t_2 &=& \cdots &=& t_N &=& t \\ T_1 &=& T_2 &=& \cdots &=& T_N &=& T.\end{array} \qquad (18)$$

We show that even in this special case parallel execution may have a better expected execution time than serial execution.

Substituting from (18), the formulas (16) and (17) reduce to

$$\begin{aligned} \langle t\rangle_{\rm ser} &= \frac{1-\bar{p}^N}{1-\bar{p}}\cdot[(1-\bar{p})\,t+\bar{p}\,T] \\ \langle t\rangle_{\rm par} &= N\cdot\left[(1-\bar{p}^N)\,t+\bar{p}^NT\right].\end{aligned}$$

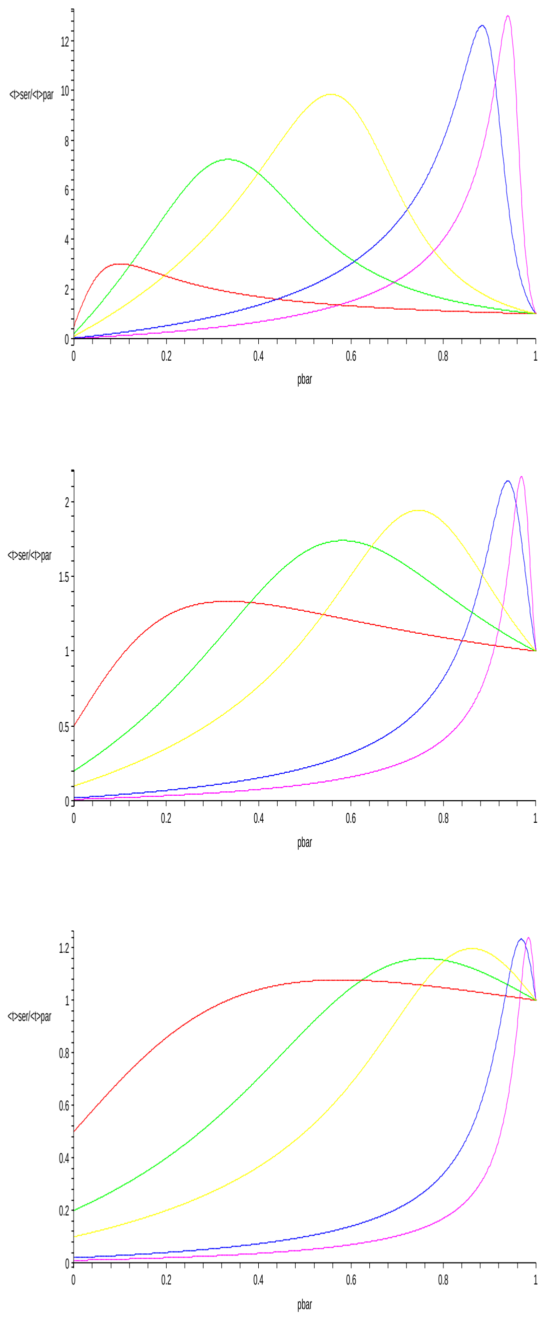

Fig. 1. $\langle t\rangle_{\rm ser}/\langle t\rangle_{\rm par}$ for various values of $\bar p$, N and t/T.

Serial execution of the tasks is expected to take longer than parallel execution when the ratio

$$\frac{\langle t\rangle_{\rm ser}}{\langle t\rangle_{\rm par}} = \frac{1}{N} \cdot \frac{1-\bar p^N}{1-\bar p} \cdot \frac{(1-\bar p)\, t + \bar p\, T}{(1-\bar p^N)\, t + \bar p^N\, T}$$

is greater than unity.

Examining this expression, we see that when p approaches zero, the value of the ratio approaches one and when p is one the value of the ratio is $1/N$, as would be expected. For small values of t/T, we find that as p increases from zero the value of the ratio increases from 1, reaches a maximum, decreases back past 1, and eventually reaches the minimum value of $1/N$ when $p = 1$. The cross-over point, where the value of the ratio is one, may be given in terms of $\bar p$:

$$\bar p = \frac{(N-1)\, t}{(N-1)\, t + T} + O\left(\bar p^N\right).$$

For example, when $T = 10t$ and $N \geq 5$, ignoring the $O(\bar p^N)$ term gives the value of p correct to within 1%.

Thus when t is small compared to T, the expected parallel time is less than the serial time when $\bar p$ is between 0 and $\sim (N-1)t/[(N-1)t + T]$. When $\bar p$ is between $\sim (N-1)t/[(N-1)t+T]$ and 1, the expected time for serial execution is the smaller.

To be very explicit, let

$$p(t) = \frac{1}{2}\delta(t-1)$$
$$q(t) = \frac{1}{2}\delta(t-4).$$

This is an extremely collusive density. We give graphs to show the ratio $\langle t\rangle_{\rm ser}/\langle t\rangle_{\rm par}$ as a function of $\bar p$ for various values of N and t/T. (See Figure 1.)

IX. Conclusion

We have provided a model of OR parallelism that allows different strategies to be evaluated for expected execution time. This can be used to determine which tasks to run in parallel and which to run serially to minimize expected time. This model has been used to show why parallel execution, even on a single processor, can be more effective than serial execution.

References

[1] S.M. Watt, “Bounded Parallelism in Computer Algebra”, Ph.D. Thesis, University of Waterloo, 1985.

[2] C.A.R. Hoare, “Parallel Programming: An Axiomatic Approach”, pp.11-42 in *Language Hierarchies and Interfaces*, ed. F. L. Bauer and K. Samelson, Springer-Verlag Lecture Notes in Computer Science No. 46, Berlin (1976).

[3] W.A. Kornfeld, “The Use of Parallelism to Implement a Heuristic Search”, pp. 575-580 in *The Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81)*, (August 1981).

[4] W.S. Brown and J.F. Traub, “On Euclid’s Algorithm and the Theory of Subresultants”, *J. ACM* **18** (4) pp.505-514 (1971).

[5] J. Moses and D.Y.Y. Yun, “The EZ-GCD Algorithm”, pp. 159-166 in *Proceedings of the A.C.M. Annual Conference*, Atlanta (1973).

[6] J.L. Bentley, “Multidimensional Divide and Conquer”, *Comm. ACM* **23** (4) pp. 214-229 (1980).

[7] H.A. Simon and J.B. Kadane, “Optimal Problem-Solving Search: All-or-None Solutions”, *AI* **6** pp. 235-247 (1975).

[8] H.W. Price, “Least-Cost Testing Sequence”, *J. Indust. Engineering* (July-August 1959).

[9] L.G. Mitten, “An Analytic Solution to the Least Cost Testing Sequence Problem”, *J. Indust. Engineering* p. 17 (January-February 1960).