Processor Allocation in Parallel Systems

Maria Calzarossa, Miroslav Kubíč
Dipartimento di Informatica e Sistemistica
Università di Pavia
via Abbiategrasso 209
I-27100 Pavia, Italy

ABSTRACT

Parallel systems are characterized by a large number of processors interacting together and cooperating for the execution of programs usually structured as a set of independent tasks. When the tasks are executed in parallel, a significant reduction of the elapsed time can be achieved. The allocation of the processors to the various programs (and tasks) plays a critical role in that it highly influences the performance of the individual programs as well as of the parallel systems. A metric for the evaluation and the comparison of different allocation policies is presented. The complexity of the programs is analyzed and an estimate of the order of time for their execution is given.

1 Introduction

Software systems, i.e., programs, to be executed on parallel systems are usually structured as a set of largely independent tasks. When each task is allocated to a different processor, a significant reduction of the global elapsed time can be obtained.

The typical performance metrics used to evaluate parallel systems are speedup and efficiency [2]. In such a case, an effective use of a system, i.e., of its processors, is measured in terms of the number of simultaneously active processors.

Since the number of processors available is limited, it is fundamental to find out "the best" allocation strategy for subdividing the processors among the tasks belonging to different programs.

The allocation policies can be classified into two categories, namely, static and dynamic policies (see e.g., [8]). A dynamic policy allows programs to request and relinquish processors at run time, according to their needs. A static scheduling gives a program, at the beginning of its execution, a constant number of processors which are held till the end of the execution itself, even though they are no more in use. When a static approach is adopted, the choice of the number of processors needs to be made prior to execution, e.g., at compile time. An interesting solution to the problem of mapping programs into a set of processors is presented in [1], where the optimization criterion adopted is the communication time. The number of processors allocated to a program equals the number of tasks it consists of. The mapping of the tasks into the processors tries to minimize the time needed for the communications among the tasks.

Minimizing communications, with a constant number of processors, does not always yield the best allocation strategy. Indeed, the parallel attributes of a program, such as, parallelism, granularity, synchronism, resource sharing, parallel structures, are usually concerned with how the components of the program are put together and how the elements in the system are used by the program. Hence, the total execution time of a program can be expressed as the sum of three components: computation time, communication time and contention times (internal, that is, within a particular program, or external, that is, among different programs). The proportion of these three components on the total time is strictly related to the parallel attributes of the programs (see e.g., [3, 7]). When a large number of processors is allocated to a program, both the internal contention time and the throughput of the system decrease. Because of the differences in the computational requirements of individual tasks, it can easily happen that a few processors are idle most of the time, a few are only partly busy and none is overloaded. In such a case, the total number of processors allocated to a program can be decreased with no penalty for the program itself and with some possible benefits for the system throughput.

In this paper we present a heuristic approach for the evaluation of static allocation policies. We start with an initial allocation where each task is given its own processor and we propose an algorithm for the estimation of the expected utilization of the individual processors. Such an estimation can be used for further optimizations.

The paper is organized as follows. Section 2 introduces the basic ideas of the proposed approach and presents the various steps of the procedure for the computation of the "mean rough utilization". A simple example of the application of the methodology is described in Section 3. Finally, a few ideas for future work are discussed in Section 4.
2 The method

The basic idea of our methodology is the estimation of the utilization of the individual processors with respect to a specific mapping. The utilization of a processor is defined as the ratio between the busy time and the total time. The values of these two measures depend on the input parameters of the program to be executed and on some unpredictable variables, such as boolean expressions in loops and branch parameters. Hence, we will only be able to estimate, by means of simple heuristics, the order of complexity of the programs. The order of complexity of the tasks performed on individual processors represents the busy times of the processors themselves. To predict the total time we use the notion of threads, and in particular of vital threads, which determine the execution time of the whole program.

The main steps of the procedure for the computation of the mean rough utilization are:

1. Definition of an initial mapping of the program and identification of the vital threads.
2. Computation of the order of complexity of the vital threads and of the work of individual processors.
3. Estimation of the utilization factor \( u_f \) of processor \( i \).
4. Computation of the mean rough utilization of the whole system.

The details of the operations performed in each step of the procedure will be described in the following sections.

2.1 The threads of control

Let us consider a simple example of a parallel program (see Figure 1) which will be adopted as a means of explanation of our methodology.

The program, written in a CSP-like language [5], consists of four tasks: \( P, S, U1 \) and \( U2 \). Just the communication commands are shown because they potentially start the busy time of the processors. The meaning of \( U1 ? x \) is ‘wait till you receive \( x \) from \( U1 \)’. \( U2 ! x \) means ‘send \( x \) to \( U2 \)’. The meaning of \( S || U1 || U2 \) is ‘start simultaneously the parallel execution of \( S, U1 \) and \( U2 \). In our example, the only operation performed by \( P \) is to start the simultaneous execution of \( S, U1 \) and \( U2 \).

In what follows, the sequences of statements between communication commands will be called subtasks. A subtask can be part of higher level loops (e.g., \( U1 \)) or branches (e.g., \( U2 \)).

Figure 2 shows the representation of the parallel program of Fig. 1 by means of an oriented graph. The nodes are the subtasks and the edges, oriented ‘from sender to receiver’, represent the communications. Such a graph will be used for our purposes in order to determine the threads in the program and their order of complexity.

An important information which can be derived from an oriented graph is represented by the threads of control, i.e., the paths followed during the execution of a program. The threads, consisting of a finite set of subtasks, can be executed in parallel or are exclusive to each other. We are particularly interested in the identification of the vital threads, which determines the minimum and maximum execution time of a program.

From Figure 2, the following vital threads can be tracked:

1. \( P = \overline{U1.1, U1.2, U1.3} - S2.1 - S1.1 - S1.2 \)
2. \( P = \overline{U1.1, U1.2, U1.3} - S1.1 - S1.2 \)
3. \( P = \overline{U1.1, U1.2, U1.3} - U1.4 \)
4. \( P = U2.1 - S1 - S2.1 - U2.3 \)
5. \( P = U2.1 - S2.1 - U2.3 \)
6. \( P = U2.1, U2.2, U2.3 \)

Another important notion is the definition of subthread, i.e., the thread which can be seen as part of a vital thread. In our example, the thread \( P = U2.1 - S2.1 \) is a subthread because after the subtask \( S2.1 \) the control goes to \( U2.3 \) which ends the program execution.

\[ P :: [S || U1 || U2] \]

\[ S :: \text{declarations} \]

\[ U1 :: \text{declarations} \]

\[ U2 :: \text{declarations} \]

\[ U1 ? \text{op}(x); \]

\[ \text{subtask} \]

\[ U1 ! \text{res}(x); \]

\[ \text{subtask} \]

\[ [U2 ? \text{op}(x); \]

\[ \text{subtask} \]

\[ U2 ! \text{res}(x); \]

\[ \text{end.} \]

Figure 1 - Example of a parallel program written in a CSP-like language.
2.2 The order of complexity

The order of complexity of a program or of a subtask is determined by means of simple heuristics. The basic idea is that each construct of a parallel language fits into one of the following categories: loops, branches, simple statement sequences.

The order of time spent in a loop is determined by the depth of loop nesting which is the only information we can derive from the source code of a program. The execution time of the branches depends on which branch is chosen, which, in turn, depends on some boolean expressions. In such a case, we can just determine the lower (min) and upper (max) bounds of the execution time, which correspond to the "lightest" and "heaviest" branch, respectively.

The time spent in a single statement sequence is given by the length of the sequence and by the complexity of the statements it consists of.

The algorithm for estimating the order of complexity of a program can be summarized in the following expert-system-like rules:

- a single statement has basically the order of 0;
- if a single statement contains a "complex formula", increment the order by one;
- if a single statement contains time-demanding standard functions (e.g., sin, cos, log), increment the order by one;
- for each loop-nest increment the order of the innermost statement by one;
- branching statements have two orders: the first one corresponding to the heaviest branch (max) and second one corresponding to the lightest branch (min);
- if the number of statements with the order of complexity equal to L exceeds a predefined threshold, substitute all of them with a single statement characterized by the order of complexity equal to L + 1.

2.3 The utilization factor

As already pointed out in the previous section, the order of complexity of a program is not defined by a single value. Because of the existence of branches, we can only express the order of complexity in terms of its lower and upper bounds, that is, by means of a pair (min, max).

We now introduce a new performance measure, the utilization factor uf, of processor i, obtained by comparing two pairs of values: the order of complexity of a vital thread and the order of complexity of the work of the individual processor i.

As a consequence, uf is defined by a pair (uflow, ufupp), determined according to the rules stated in Table 1.

A 3-valued logic, with values 0, 1, (0, 1) is used to determine the utilization factor. This approach is borrowed from the Dempster-Shafer probabilistic theory[4] where the probability distribution is assigned to subsets of the domain rather than to the discrete values.

<table>
<thead>
<tr>
<th>condition</th>
<th>uflow</th>
<th>ufupp</th>
<th>'physical' meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_p &lt; min_p</td>
<td>0</td>
<td>0</td>
<td>underutilized</td>
</tr>
<tr>
<td>min_p ≥ max_p</td>
<td>1</td>
<td>1</td>
<td>overutilized</td>
</tr>
<tr>
<td>all other cases</td>
<td>0</td>
<td>1</td>
<td>undecided</td>
</tr>
</tbody>
</table>

Table 1 - Definition of the Utilization Factor
$min_T$ and $max_T$ denote the order of complexity of the vital threads. $min_p$ and $max_p$ indicate the order of complexity of the work of an individual processor.

The rationale behind the definition of the utilization factor can be easily explained in terms of its physical meaning. An 'over-utilized' processor is considered busy most of the time because there is at least one-order difference between its busy time and the execution time of the program. The same considerations apply to the 'under-utilized' case. In the 'undecidable' case, both the extreme situations are possible.

2.4 The mean rough utilization

Let $N$ be the number of processors allocated to a program. Let $u_f$ be the utilization factor of processor $i$. The mean rough utilization of the processors is defined by the pair $(mrulow, mrupp)$, where:

$$mrulow = \frac{1}{N} \sum_{i=1}^{N} (u_{f_{low}})_i$$

and

$$mrupp = \frac{1}{N} \sum_{i=1}^{N} (u_{f_{upp}})_i$$

The $mrulow$ represents the estimate of the fraction of processors that are not idle most of the time. Values close to 1 indicate that the processors are highly utilized. Values close to 0 indicate a sort of 'wasting' of the resources.

The optimal allocation strategy will then maximize the $mrulow$ and minimize the global number of processors $N$ allocated to each program.

The details of the algorithm for the computation of the $mrulow$ can be found in [6].

3 Example

As an illustration of the proposed methodology, let us consider a subtask consisting of 8 statements. In this example, we suppose that the values of the threshold on the number of statements of the same order of complexity is equal to 6.

Figure 3 shows the source code of the subtask together with the values of the minimum and maximum order of complexity for each statement. These values have been derived according to the rules presented in Sect. 2.2.

Hence, the minimum and maximum order of complexity for the subtask are:

$$min = 6 \cdot 1 + 2 \cdot 0 = 2$$

$$max = 2 \cdot 2 + 4 \cdot 1 + 2 \cdot 0 = 3 \cdot 2 = 2$$

Suppose we have already analyzed the complexity of all the vital threads and found out that the order of complexity of the whole program is $(3,5)$. We can then conclude that the utilization factor $u_f$ of the processor which executed the subtask is equal to $(0,0)$.

4 Conclusions

The paper introduces a methodology for the evaluation of the effectiveness of processor allocation in parallel systems. The estimate of the processor utilization obtained prior executing the program itself is very important with many respects. For example, a programmer, who has a rough description of the program he plans to implement would like to be able to predict its performance before investing further effort in coding and testing the program. Another possible use of this

1. read(x);
2. $y := x + 2$;
3. if $y > 5$ then $z := \sin(y) + 0.5$;
   else for $i := 1$ to $n$ do
      begin $x := x + 1$;
      $y := y \cdot x$;
      $z := \sin(y) + 0.5$;
   end;
4. for $i := 1$ to $m$ do
   begin $x := 3 \cdot z + 0.1$;
   if $x > 0.7$ then $x := \sqrt{x}$;
   if $x > 0.5$ then
      for $j := 1$ to $p$ do
      begin $z := y \cdot x$;
   end;
5. $z := \sin(x)$
6. $y := 3 \cdot z + 1$;
7. $z := ((0.5 \cdot y)/(x + 2) - 0.8/(2 \cdot y + 3 \cdot z + 0.1 \cdot z))$;
8. write(y);

Figure 3 - Example of a subtask
methodology is in the compiler design. At compile time, the compiler is faced with many alternatives on how to divide the computational work, how to allocate portions to different processors and how to get them to work together. A knowledge of the performance of the various alternatives will help the compiler in making the "best" decision.

Some important aspects of our methodology remain to be investigated. An improvement with respect to the 'undecidable' case must be sought. Furthermore, the value of the threshold as well as an estimate of the computation time of standard functions should be derived from measurements performed on real applications. The relationship between the value of the mru and the overall system performance has also to be analyzed.

5 References


