Abstract

Task graphs have been proved to be a very useful structure for solving co-synthesis problems. A tool, which will automate the task extraction procedure from the source code and will present the resulting task graphs, is required. In this paper, the creation process of such a task extraction tool is analyzed, focusing on four domains: the determination of the data dependencies, the estimation of each task’s execution time, the task construction and the final visualization of the graph.

Introduction

The basic challenge, when dealing with co-synthesis problems and parallel architectures, is to parcel independent tasks among the systems’ processors in an efficient way. The term efficient implies not only that the system’s performance will improve, but also that the program’s data flow will remain unaffected. To achieve this goal, there are two different approaches.

In the first one, the concurrent programming, the programmer is responsible for the biggest part of parallelization. Using parallel programming libraries and methodologies, depending on the machine’s architecture, (shared variable programming or message passing), he divides the tasks among processes which don’t interfere with each other. During execution, parallelization is achieved by routing the processes to different processors. However, this approach requires that the developer is familiar with multithreaded and distributed programming languages and that he is aware of the architecture to be used. He should also be able to determine the dependencies between processes and their synchronization.

In the second approach, the developer has an easier role. The program is written sequentially, as if it would be executed by a single processor. Then, a task extraction procedure is needed to determine the independent tasks, which could be executed in parallel and lead to a better performance. This method is independent of architectures and does not require further programming knowledge. Nevertheless, the manual task extraction is a difficult procedure and it could be time consuming, so a tool, which will automate the procedure of task extraction, is needed. The tool should present the tasks and their data dependencies in a form easily understood both by human and compilers. Task graphs are ideal for that role, because graphs can describe problems with clarity and precision and they can easily be generated and visualized, using already existing tools.

In this paper, a proposal for such a task graph extraction tool is discussed, focusing on four aspects, which are crucial for the further parallelization process. The first and most important is the determination of data dependencies between nodes (tasks) of the graph. This means that one should consider carefully what kind of information about the tasks (data, execution time etc) should be included in the graph and which method will be applied for the data dependence analysis. The second aspect is the estimation of each task’s execution time. The actual run time of a task in the program may be
different from the execution time of the task independently, so algorithms that solve that problem should be examined. Thirdly, it should define what a task is, what a task should include and if some lines of code should be treated sequentially in a single task, rather than in parallel. Finally, the output of the program and the visualization of the produced task graph, using already existing tools, should be discussed.

1. Data Dependencies

The basic rule during the parallelization process is the maintenance of the control and data flow of the program. This means that data dependent statements should retain their execution order. There are four main types of data dependencies: the flow dependence, when a variable is written in a statement and read in the following one, the anti dependence, where a variable is read in a statement and written in the following one, the output dependence, where a write is followed by another write and the input dependence, where a read is followed by another read.

The first operation of the tool is to take the source code written in C and retrieve the data dependences between statements. Thus, for scalar variables, data flow analysis techniques should be used, so as to determine the order of execution. This task is also performed by compilers in order to make appropriate transformations, optimize the source code and offer an efficient scheduling. As a result, the part of the Scale compiler, which performs the standard data flow analysis could be used to build the data dependences of the input source code in our implementation. However, these techniques are not enough to give accurate results in our case, as they are not able to set up the data dependences in more complicated situations, for example when dealing with for loops or arrays. ‘For loops’ are very crucial for the parallelization, because they are time consuming, if executed sequentially. Thus, the task extraction tool should be able to avoid false data dependencies between loops that could be parallelized. Nevertheless, parallelizing the loops’ iterations can affect severely the programs data flow, because there may be data dependencies between two instances of the same statement in different iterations (cross iteration dependence).

For that purpose, more sophisticated analysis is needed, so as to check the subscripts of any array contained in the loop and determine the existence or not of data dependencies. There are several array dependence tests which try to solve this problem. Their basic concept is to create an equation with the arrays subscripts and try to solve it ([1]). Our goal is to decide which one should be used as a part of the task extraction tool, according to their accuracy and efficiency.

The simplest test is the GCD test ([2], [3]). It is able to prove the non existence of data dependencies, by determining that there is no solution to the subscripts equation, but it can’t do the same for existing ones. It is very efficient in terms of speed but there are cases where the test just assumes that there may be dependence, so its results are not accurate enough so as to be included in our tool. The Banerjee test is very fast, can also prove the non existence of a solution in a linear equation, but also the existence of a real solution, if there is one. However, because the variables in the arrays’ subscripts can only be integers, we are not interested in real solutions but in integer ones. The Banerjee test cannot make this distinction so it also gives inaccurate results. The I- test can be considered as an optimization of the Banerjee test, as it contains some conditions, in order to check if the solutions of the Banerjee test are real or integers. It is very efficient and accurate, apart from the case of coupled subscripts, where it returns a ‘maybe’ answer. Finally, the Omega test ([4]) uses the GCD test to
prove the non existence of dependencies. If this cannot be done, it uses the Fourier-Mortzkin Variable Elimination (FMVE) to prove the opposite. It is very accurate, but it lacks efficiency. From the experimental results in [3], we conclude that the Omega test is slightly more accurate than the I-test. The only case where the I-test fails to decide for the existence or not of dependencies is when the subscript variables cannot be bounded.

Taking these experimental results into consideration, the Omega test seems to be the most appropriate for the array data dependence in our tool. Furthermore, this selection is highly encouraged by the fact that the Omega test is also a part of the Scale compiler, so the corresponding part of the Scale source code could be used. In conclusion, during the first stage of the tool’s creation, the Scale compiler’s source code and Documentation should be examined, in order to retrieve the parts which deal with the scalar and array data dependence analysis. After a careful observation on how these parts achieve their goal, it should be determined how they could be adjusted in order to fit in our tool’s implementation.

**Time Estimation**

The execution time estimation of the programs’ tasks is another important process for the parallelization, so it must also be implemented in the task graph extraction tool. Various techniques have been developed for estimating the execution time of functions or programs ([5], [6], [7], [8], and [9]). Some of them are based on the usage of previous statistical analysis in order to make reliable predictions. The statistical data needed to use those algorithms include observations of execution times of the same program on the same hardware architecture, and the accuracy of their results are highly dependant on these previous historical data. However, in our case the computer architecture is not predefined, so the tool cannot include this kind of statistics. Moreover, the execution time estimation process refers to tasks rather than functions and complete programs. Thus, it is impossible to collect statistics for the execution times of all kind of statements and their combinations.

On the other hand, it would be better if the tool provided a measurement rather than an estimation of each tasks’ run time. The simplest method, also used by other task extraction tools, is to run its task as an independent one and put time stamps (for example using the system’s timer) in order to keep track of the starting and ending time. Then, the execution time is calculated as the difference between these two values. Thus, in the first stage we should compute the run times of all simple tasks (basic blocks). In the next step, some statements should be merged in order to produce bigger tasks (the reasons and corresponding techniques are examined in the next section). The challenging issue after the merging, which may end up in the formation of a complicated task, is to estimate its own execution time. One would argue that, since the run times of the fundamental tasks have been calculated, the total run time of the bigger task would be the summation of its component tasks’ run times. Nevertheless, this is not an accurate approach, since the total execution time may be proved to be much smaller than the summation. That can be explained by the architecture of nowadays processors, which are able to carry out more than one instruction per cycle, so that an instruction level parallelization is achieved. As an example, let’s assume that the two tasks in Figure 1 have to be merged, so as to create a bigger task. The first has a run time of 10ns and the second one a run time of 15 ns. If executed sequentially the total run time of both tasks would be 25ns. However, with the preprocessing in a compiler, it is clearly observer that the execution of the second
task could start instantly after the execution of statement (1), since the first task has produced the data needed for the second task.

```c
for (i=0; i<p; i++) {
    int a = rand()%100;
    data[i] = a*5 + d;
    printf("Original data: ");
}
```

```c
for (i=0; i<p; i++) {
    int a = rand()%100;
    data[i] = a*5 + d;
    printf("Original data: ");
}
```

```c
if (d == 0) {
    printf("Results: ");
    for (i=0; i<p; i++) {
        printf("d", results[i]+a);
    }
}
```

```c
if (d == 0) {
    printf("Results: ");
    for (i=0; i<p; i++) {
        printf("d", results[i]+a);
    }
}
```

---

**Figure 1:** Two dependent tasks and the result of their merging

<table>
<thead>
<tr>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>--</td>
</tr>
</tbody>
</table>

```c
for (i=0; i<p; i++) {
    int a = rand()%100;
    data[i] = a*5 + d;
    printf("Original data: ");
}
```

```c
for (i=0; i<p; i++) {
    int a = rand()%100;
    data[i] = a*5 + d;
    printf("Original data: ");
}
```

```c
if (d == 0) {
    printf("Results: ");
    for (i=0; i<p; i++) {
        printf("d", results[i]+a);
    }
}
```

```c
if (d == 0) {
    printf("Results: ");
    for (i=0; i<p; i++) {
        printf("d", results[i]+a);
    }
}
```

---

**Figure 2:** The actual execution of the merged tasks by a single processor

**Task Construction**

After having found the data dependencies and computed the execution times of all basic tasks (statements), the tool must decide which tasks should be merged together and produce bigger tasks. In case there is no data dependence between two tasks it is obvious that they should be parallelized, since it is highly unlikely to achieve a better execution time if merging them. On the other hand if the tasks are dependent, the decision can be more complicated. Parallelization has as its final goal the reduction of the program’s execution time. Taking into consideration that the merging phase could end in a task which is executed faster, as explained in the previous section, and that
the data transmission from one task to another has a cost in time ($T_{\text{total}}$), we conclude that the tool should be able to calculate these times and compare them.

More specifically, the transmission time depends on two factors: A startup time ($T_{\text{startup}}$), a parameter which shows the transmitting capability of the architecture (the bandwidth) ($B$) and the volume of the data to be transmitted ($D$).

$$T_{\text{total}} = T_{\text{startup}} + \frac{D}{B}.$$ 

The parameters $T_{\text{startup}}$ and $B$ should be given as inputs to the tool in the beginning of its execution. Variable $D$ depends on the data required to be transmitted and can be measured by the input data it takes. If the total transmission time is bigger than the run time of the task, it means that parallelization would lead to poor performance, so the tool should calculate its execution time, if executed in the same processor merged with another task. To achieve that, the two tasks’ lines of code are merged together, and the result is executed individually. If the transmission time is smaller than the run time though, the tool should decide not to merge them, as it may be more efficient to parcel them in different processors during the scheduling.

The problem becomes even more complicated, if we consider that there will probably be more than two independent tasks in a program. In that case the decision that needs to be made is not only if there should be a merging between the tasks, but also which merging would be more efficient in terms of execution time. In all cases, first the parallel approach should be investigated and only if it does not promise efficient results should the merging path be considered.

Graph Visualization

After the task construction, the tool must be able to present the results with clarity, so that they can be used in the next stage of scheduling the tasks to the systems’ processors. Therefore, a graph visualization tool is needed.

The Visualization of Compiler Graphs (VCG) tool ([10]) is the most appropriate in our case. It takes the specifications of the graph as an input from a text file, which will be generated by the task extraction tool, and creates the graph. The language used for representing the graph in a text file is the Graph Description Language (GDL) ([12]). The graph is, therefore, specified as a structure with nodes, edges and attributes. Each node has a unique title and a number of attributes and each edge is specified by the titles of its source and target nodes and its attributes. The following example demonstrates the use of the GDL language.

```json
graph: {
  node: { title: "Task1" label: "datain1,dataout1,time1" }
  node: { title: "Task2" label: "datain2,dataout2,time2" }
  edge: { source: "Task1" target: "Task2"
           label: "passing data: D" }
}
```
The shape and style of nodes and edges can be determined by adding the statement shape: NodesShape or linestyle: EdgeLineStyle next to the corresponding node or edge definition.

The final layout depends on the time limit set by the user. If there is no time limit, it performs in the normal mode, trying to create the best possible layout, a task which can be time consuming in case of many nodes and edges. However, fast results can be produced by specifying the limit, so the tool switches to fast mode. Moreover, it is possible to focus on a region of the graph, by declaring and folding subgraphs and by hiding nodes and edges which are of no interest. It is possible to choose among different shapes of nodes and edges, in order to achieve the best visualization and it can convert the final graph in other modes (postscript or bitmap). Finally, it is distributed under the GNU license and it applies in all UNIX like systems.

The aiSee tool ([11]) started as the development of VCG for Windows systems, but now it is also available both for Linux and Windows. It includes some improved features compared with the VCG such as the visualization of hierarchical graphs, where a subgraph can be a node and can be unfold, depending on the user’s choice, the support of animating layout, which means that the nodes move whenever there is a change in the layout, the support of conversion in more formats (same as VCG plus PNG and SVG), the inclusion of more styles for nodes and edges and finally the capability of visualizing larger graphs.

Finally, another tool available both for Windows and Linux systems is the uDraw ([14]), which was formerly known as the daVinci tool. It is distributed under the Lesser Gnu Public License (LGPL) and it is able to visualize a great variety of graphs, but it gives the developer the opportunity to use an application programming interface (API) designed for this tool and connect his application to uDraw. The input files used to load the graph are written in a text format called term representation, which is specified in the API’s documentation ([15]). Generally, it seems the syntax used is more difficult to understand than the GDL’s syntax. A simple representation of our previous example would be:

```
[ 1 ("Task1", n("T1", [a("OBJECT", "Task1")]),

[ 1 ("Task1 – Task2", e["T2", "Task1 – Task2",

[ 1 ("Task2", n("T2", [a("OBJECT", "Task2")],[]))])])
]
```

The first line defines a node task1 (T1), the second defines an edge from task1 to task2 and the third defines the child node Task2 (T2).

**Time plan**

The first priority in the project is to gather all tools needed for establishing the data dependencies. By the 1\textsuperscript{st} of June a familiarity with the Scale compiler will have been achieved and by the 20\textsuperscript{th} of the same month the parts of its source code, which are needed for the task extraction tool, will have been inspected. These include the data dependence test for scalar variables and for arrays (the Omega test). By the 10\textsuperscript{th} of July the data dependencies part and the calculation of each tasks execution time part should
be completed, so until the 1st of August our focus will be on the task construction, which seem to be the most difficult part. After that, the decision on the visualization tool will be made, so the next two weeks the program’s output will be adjusted to the language the visualization tool uses. The next 5 days will be dedicated on performing several experiments using the program, noting the results and evaluating them. During the last 10 days of August the final thesis will be written, containing the stages of the tool’s creation and discussing the results of the experimental tests.

Conclusion

The creation of a task extraction tool must be able to perform efficiently four operations: determination of the data dependences, estimation of each task’s execution time, graph hierarchical construction and graph visualization. In this paper several tools and methods have been discussed in order to conclude to the most appropriate ones.

As far as the data dependencies are concerned, the Scale compiler is proved to be very helpful, since it is implemented in Java and it contains the basic flow analysis needed for the determination of data dependencies between scalar variables. In addition, it contains a java implementation of the Omega Test. The Omega Test is a tool for array data dependence analysis, a procedure which is essential for the determination of more complex dependencies, usually found in ‘for’ loops. After an evaluation of different tests, the Omega test seems to be the most effective one.

After the examination of some techniques used for estimating the execution time of a program, it is clearly seen that in our tool’s case, in which accuracy is very important, a calculation rather than estimation of the execution time is needed. Taking into consideration that each programming language provides functions for the program’s communication with the system’s timer, the task extractor could execute each task independently and keep track of its execution time. However, in the next phase of task construction, when tasks may need to be merged together in order to achieve maximum efficiency, the tool should merge the tasks’ source codes, execute them again as one and keep track of the new time as well. The reason is that each processor is able to execute multiple instructions per cycle, so the execution time of merged tasks will not be the summation of their individual execution times.

The task construction stage is responsible for the conclusion of the tasks creation. Using the data dependencies and the execution times the tool decides which tasks can be merged and if this merging will result in a more efficient final graph. This process may contain several issues, if more than two tasks can be merged. The decision should be always made based on the maintenance of the data flow and the best resulting execution time.

The final tool’s output will be an ASCII file, which will contain the representation of the program’s final graph. The language used should be readable by one of the graph visualization tools examined (VCG – aiSee, uDraw). VCG and aiSee use the GDL language, which uses simple and clear syntax to define the graph with graph oriented terms (nodes, edges, attributes), while uDraw uses a more complicated language defined in an API, which is used for the program’s communication with the visualization tool.

References


