Aiding parallel programming with on-the-fly dependence visualisation

Oliver Sinnen, Ratha Long, Quoc Huy Tran
Department of Electrical and Computer Engineering, University of Auckland
Private Bag 92019, Auckland 1142, New Zealand
o.sinnen@auckland.ac.nz

Abstract—Parallel programming is notoriously difficult. This becomes even more critical as multicore processors bring parallel computing into the mainstream. In order to ease the difficulty, tools have been designed that help the programmer with some aspects of parallelisation. Unfortunately, the programmer is mostly left along when it comes to the difficult task of dependence analysis among the subtasks to be executed concurrently. This paper presents a new visual tool that supports the programmer with the dependence analysis in loops. This is very useful in combination with an automatically parallelising compiler or when loops are parallelised with OpenMP. The tool displays on-the-fly the dependences between the statements of the loop nest on which the developer is currently working. To maximise the usefulness of the tool, it is unobtrusive, customisable and flexible, and based on dependence analysis theory. A prototype was implemented for the Eclipse IDE as a plug-in that seamlessly integrates into the normal development process. The evaluation of the tool, including an evaluation against cognitive dimensions, demonstrates the usability and usefulness of the tool.

I. INTRODUCTION

The physical limits of processor technology have brought parallel computing into the mainstream. Modern processors have two or more cores. To benefit from them, programs have to be parallelised, i.e. written in such a form that they utilise the multiple cores at the same time. As a consequence, parallel computing has become more important than ever, but unfortunately, parallel computing is still a challenging problem.

The program to be parallelised has to be divided into subtasks, the dependences between them have to be analysed and they must be scheduled onto the processors of the parallel system [1]. To support the programmer in this difficult job, visual tools can help the programmer. However, many existing visual tools in the area of parallel computing are for debugging and performance analysis, e.g. [2]. Visual tools to support the programmer in the parallelisation process are lacking and there are only a few exceptions [3], [4]. This is especially true for tools that support the programmer in their known programming environments and do not force them to adapt to new environments.

In most computer programs, loops carry the lion’s share of the computational load and are therefore the primary target of parallelisation. Modern compilers, e.g. Intel C/C++ compilers and GCC [5], are able to automatically parallelise common loop types. An important condition for the successful parallelisation is that there are no dependences between the iterations of the loop. While the compiler will report for which loops parallelisation was possible and for which not, the programmer is usually left alone with the problem of analysing the code and avoiding or eliminating dependences.

In this paper we present a visual tool that supports the programmer with the dependence analysis. While the programmer is coding, our tool visualises the dependence structure of the loop (nest) the programmer is currently working on. This visualisation of the dependence structure can increase the developer’s understanding of the iterative code. Code can be rewritten to avoid dependences (e.g. antidependences) or the loop can be transformed to eliminate them [6]. Consequently, the visual tool supports the development process targeting an automatically parallelising compiler. The visual tool can also help the programmer to decide to enforce parallelisation (e.g. with OpenMP constructs [7]), when she or he is satisfied that there are no dependences, even if they can only be analysed partially by an automatic tool. Moreover, if dependences are detected early, the developer can even choose to implement an altogether different algorithm, which might be more suitable for parallelisation.

We developed a prototype tool that visualises the dependence structure of the loop on which the programmer is currently working on. The visual representation of the dependences is constantly updated and reflects the current state of the code. This tool was realised in Java for Java as an Eclipse plug-in and thereby stands in the same tradition as, say, a class browser for Object-Oriented programming. We implemented a Java parser and dependence analyser using accurate dependence tests. The tool is flexible and adaptable, for example the programmer can select the type of dependence to be displayed.

Our approach is different to previous ones, like the SUIF Explorer [3] or the 3D Iteration Space Visualizer [4], in some important points. The visualised dependence graph is the compact, cyclic dependence graph not the unrolled 3D iteration space [4]. More important, our tool is deeply integrated in a common development environment, and thereby enhances the programming experience instead of changing it.

Our testing and performance analysis revealed that the tool is sufficiently fast, robust and unobtrusive. More importantly, we analysed our tool proposal against cognitive dimensions, a common Human Computer Interaction (HCI) evaluation practise, where it shows its good properties.

The rest of this paper is organised as follows. Section II
reviews the necessary background of data dependence analysis. Our visual tool proposal is presented in Section III, some implementation details are discussed in Section IV and its performance is evaluated in Section V. The paper concludes in Section VI.

II. BACKGROUND

Dependence analysis [6], [8], [9], [10] distinguishes between two kinds of dependence: data dependence and control dependence. The latter represents the dependence relations evoked by the control structure of the program, which is established by conditional statements like if...else. This paper focuses on data dependence.

For data dependence, one can further distinguish between three different types, namely flow dependence, antidependence and output dependence. In imperative/procedural programming languages, these dependence types occur through the writing to and reading from variables or memory locations. The following example quickly illustrates the dependence types:

\[
\begin{align*}
1: & \quad a = 2 \\
2: & \quad v = a \times 5 \\
3: & \quad u = v + 2 \\
4: & \quad v = 3 \times 7
\end{align*}
\]

Every line of this program segment is a statement of the form \textit{variable} = \textit{expression}. The statements are entities between which we define the dependence relations. A dependence occurs if a variable is used in more than one place, of which at least one is a write.

**Flow dependence** is given when a variable is first written to and later read from, e.g. variable \texttt{a} is written in line 1 and read in line 2. **Antidependence**, as the name suggests, corresponds to the converse case: a variable is first read and then written to, e.g. line 3 reads \texttt{v} before line 4 writes to it. Finally, **output dependence** is the occurrence of two writes to the same variable, e.g. both line 2 and line 4 write to variable \texttt{v}. In all cases of dependence, executing the statements in a different order, for instance line 2 before line 1, generally leads to a wrong result, in the sense that it would differ from the outcome of the sequential execution of the original program.

It is common to represent the dependences of a code segment as a directed dependence graph [6], [10]. The nodes of the dependence graph represent the statements and the directed edges represent the dependences between them. Figure 1 depicts the dependence graph for the program segment above, with labels for the different types.

![Dependence graph example program](image)

Figure 1. Dependence graph for example program

Antidependence and output dependence are dependences that only arise through variable reuse and can therefore be eliminated. For example, the antidependence and the output dependence disappear when we use a new variable \texttt{w} in line 4 instead of \texttt{v}. In contrast, flow dependences are inherent to the computation and therefore called real dependences.

A. Data dependence in loops

Data dependences in loops follow the same basic concept, but differ in two important points: statements in loops are executed multiple times and loops often contain array variables.

The following example presents a loop over the index variable \(i\) with a loop body, or loop kernel, consisting of the two statements, or tasks, \(S\) and \(T\).

\[
\begin{align*}
\text{for } i = 2 & \text{ to } 100 \text{ do} \\
& S: \quad A(i) = B(i + 2) + 7 \\
& T: \quad B(i + 2) = A(i - 1) + C(i + 1)
\end{align*}
\]

The statements \(S\) and \(T\) are executed multiple times, once for each value of \(i\), \(i = 2, 3, \ldots, 100\). Dependence can arise between these instances of \(S\) and \(T\) through references to the same elements of the arrays \(A\) and \(B\) (array \(C\) is only read, hence not incurring dependences). Within one iteration (intra iteration), i.e. between the instances \(S(j)\) and \(T(k)\), with \(j = k\), it can be easily verified that there is no dependence as the accessed array elements are different. However, dependence does arise between instances of different iterations, called inter iteration dependence. For example, instance \(T(3)\) is flow dependent on \(S(2)\): the output variable of task \(S(2)\), \(A(2)\), is the input of \(T(3)\). The same can be observed for the relation between \(T(4)\) and \(S(3)\), and between \(T(5)\) and \(S(4)\), and so on. The dependence distance, that is the number of iterations between the two task instances forming a dependence relation, is 1. In general, it can be stated that the instance \(T(i + 1)\) is dependent on the instance \(S(i)\). Such a dependence is called **uniform** since its distance is constant.

One can observe further dependence relations in the above loop example. The instance \(S(3)\) is flow dependent on \(T(2)\), caused by the array element \(B(5)\), and \(S(5)\) is flow dependent on \(T(3)\), caused by \(B(7)\). These are two examples of the dependence pattern caused by the output variable \(B(i + 2)\) of \(T\) and the input variable \(B(i + 2)\) of \(S\). Other dependence pairs of this pattern are: \(S(7)\) depends on \(T(4)\), \(S(9)\) depends on \(T(5)\), \ldots, \(S(99)\) depends on \(T(50)\). As the distance of these dependence relations is not constant – the minimum distance is 1 and the maximum is 49 – it is said to be **non-uniform**.

The dependence graphs of loops also follow the same principle as seen in Figure 1, with the difference that we also indicate the dependence distance next to every edge. Section III will show several examples of dependence graphs for loops.

B. Double loops – loop nests

The next logical step in dependence analysis of loops is to extend the described concepts to double and multiple nested loops. Below a double loop over the indices \(i\) and \(j\), containing the two statements, or tasks, \(S\) and \(T\) in the kernel.

\[
\begin{align*}
\text{for } i = 0 & \text{ to } 5 \text{ do} \\
& \text{for } j = 0 \text{ to } 5 \\
& S: \quad A(i + 1, j) = B(i, j) + C(i, j) \\
& T: \quad B(i + 1, j + 1) = A(i, j) + 1
\end{align*}
\]

What the index variable is to the single loop, is now, in a straightforward generalisation, an index vector of two
dimensions. An instance of the double loop kernel, i.e., an iteration, is determined by the two corresponding values of the index variables $i$ and $j$. Also, an instance of one of the tasks $S$ and $T$ is denoted by $S(i, j)$ and $T(i, j)$, respectively. The extension to a more general nest of loops follows a similar pattern – every loop simply contributes one dimension to the index vector.

By examining the tasks of the double loop, it becomes apparent that instance $S(i + 1, j + 1)$ depends on instance $T(i, j)$, caused by the references to the elements of array $B$, and instance $T(i + 1, j)$ depends on $S(i, j)$, caused by the references to the elements of array $A$. As a logical consequence of the generalisation from the index variable to an index vector, the dependence distance is also expressed as a distance vector. For the identified dependence relations of the double loop, the distance vectors are $(1, 1)$, for $S(i + 1, j + 1)$ depending on $T(i, j)$, and $(1, 0)$, for $T(i + 1, j)$ depending on $S(i, j)$. So there are two uniform dependences, as the distance vector is constant for every dependence.

How to automatically detect and calculate dependences, so called dependence tests, is discussed in Section IV.

### III. VISUALISATION

In general, parallelisation of a loop, i.e., the current execution of the iterations, is not possible if there are inter iteration dependences. Only very few and specific cases can be automatically transformed by a parallelising compilers [6]. Hence, being aware and avoiding dependences is an essential aspect of the parallelisation process.

#### A. Objectives

The aim of the proposed visualisation tool is to visualise the dependence structure of loops, while the programmer is implementing them. When designing the visualisation tool, we had the following objectives.

- Display all common detectable dependences
- Indicate when uncertain – Indicate for which code parts a conclusive dependence analysis is not possible (e.g., due to possible side effects). This allows the developer to take decisions on enforcing parallelism, even if a parallelising compiler would not do it.
- Integrate in normal programming environment – The tool should seamlessly integrate into the normal development process.
- Unobtrusive – The tool should not distract the programmer from the normal development flow. Ideally, the programmers do not have to change or adapt their development steps or perform additional steps.
- Display all available information in a familiar and intuitive format – Using the dependence graph representation is a well known and tested visualisation concept of dependence structures. Developers familiar with dependence analysis theory do not need to learn anything new.
- Automatic and consecutive update – The tool should always reflect the current state of the code without requiring the developer to request an update.
- Customisable and flexible – To optimise the efficiency the developer must be able to customise the visualisation. For example to focus on certain dependence types or to change visualisation aspects.

#### B. Eclipse Plug-in

To achieve the objectives of the visualisation tool, it has been developed as a plug-in for the Eclipse IDE [11]. As the target language (the language to be analysed) Java was chosen, but most of the techniques and approaches are directly applicable to other imperative languages, e.g., C/C++. Initially Java was chosen for the proof-of-concept implementation due to the familiarity of the authors with Java. Yet Java is also particularly interesting for such a tool as there are no production compilers that perform automatic parallelisation of loops. At first sight this seems to contradict the purpose and usefulness of the tool. However, there is an unofficial implementation of OpenMP for Java, called JOMP [12]. When the developer parallelises a loop with the parallel-for directive of OpenMP, she or he must guarantee that there are no dependences between the iterations. The developer is left alone with the necessary dependence analysis to ensure this. Hence, the proposed dependence visualiser can be very helpful in such a situation and not only when developing for a parallelising compiler.

The plug-in was conceived as an additional view of the currently edited source text file in the Eclipse IDE. While the user writes the source code in the text editor window, our tool monitors and analyses this source code. The additional view visualises the dependence structure of the loop(s) enclosing the cursor. Figure 2 shows a screenshot of Eclipse with the dependence visualiser in action. On the left is the normal Java code text editor panel provided by the Eclipse IDE. On the right the dependence graph of the loop enclosing the cursor in the text editor panel is displayed in our visualisation tool panel.

The plug-in monitors the text editor for modifications and changes of the cursor position. Whenever there is inactivity for a certain period of time, currently one second, the plug-in parses the code and creates the dependence graph or updates it in the right view for the loop nest enclosing the cursor.

The visualisation panel draws the dependence graph with the corresponding components:

- Nodes correspond to line numbers of source file
  Each node of the dependence graph corresponds to a line number in the source code text file and is drawn as a blue circle. The default layout of the nodes is linear in vertical direction ordered by ascending line numbers. Code lines that cannot be satisfactorily analysed in terms of dependence, for example when there is a function call, are drawn in red to indicate potential side effects, see Figure 2. Those code lines are excluded from the rest of the dependence analysis so that the user can still see all analysable dependences.
- Edges of dependence graph are displayed as directed lines
  The edges of the dependence graph are drawn as lines directed in the order imposed by the dependence. Intentionally, each detectable dependence is reflected by
Figure 2. Overview of dependence visualisation tool

To distinguish the different dependence types, the directed lines are coloured and labelled differently as given in Table I. A dependence is scalar if the variable incurring it is a scalar and not an array. A dependence is unknown if two statements access the same array variable, but the tool is unable to analyse if the same array elements are accessed. For example, this is the case when the array subscripts are non-linear functions of the index variables.

<table>
<thead>
<tr>
<th>Dependence type</th>
<th>Colour</th>
<th>Edge label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow</td>
<td>Blue</td>
<td>F</td>
</tr>
<tr>
<td>Anti</td>
<td>Red</td>
<td>A</td>
</tr>
<tr>
<td>Output</td>
<td>Green</td>
<td>O</td>
</tr>
<tr>
<td>Scalar</td>
<td>Orange</td>
<td>S</td>
</tr>
<tr>
<td>Unknown</td>
<td>Black</td>
<td>?</td>
</tr>
</tbody>
</table>

Table I

DEPENDENCE COLOURS AND LABELS

Figure 3 shows simple loops and their corresponding dependence graph as displayed in our tool.

1) Customisable, interactive, flexible: To maximise the usefulness of the dependence visualiser its design includes customisation and interaction functionality. When the user clicks on a node in the dependence graph, the corresponding line number is highlighted in the source code. All nodes can be freely positioned in their panel in order to allow the user to get a better perspective or to position the nodes according to the expected dependence structure. Per default the nodes are in a vertical linear layout ordered by ascending line numbers.
Alternatively, a press on a button in the Eclipse tool bar will bring the nodes into circular layout.

Furthermore, each dependence type can be excluded from the display, i.e. it is not shown, or made exclusive, i.e. only that dependence is shown, with simple button presses in the Eclipse tool bar. This functionally helps the developer in involved dependence situation to focus on certain dependences. An example for an involved dependence graph is depicted in Figure 4.

As already mentioned, the dependence visualiser works with multiple loops in a single source file, because only the dependence graph for the loop nest enclosing the cursor position in displayed. Moreover, our tool also handles multiple text files open in different tabs, as the dependence graph is automatically shown for the text file currently worked on.

Figure 4. Filtering dependence display

IV. IMPLEMENTATION

As already mentioned, a prototype of the proposed dependence visualiser was realised as an Eclipse IDE plug-in for Java. Being a plug-in, the tool fully integrates into the normal Java development process supported by Eclipse, including editors, debuggers, compilers etc.. The plug-in was implemented in Java, based on several technologies.

Apart from the standard toolkits provided by the Eclipse platform for plug-in and extension development, the Graphical Editing Framework (GEF) is employed for the drawing of and the interaction with the dependence graph. GEF is a rich and powerful framework, based on the Model Control View (MVC) model, for developing graphical applications with Eclipse. It provides all the necessary components such as circular shapes (the nodes) and smart connections (the dependence edges) and functions for their management (positioning, highlighting, dragging etc.).

To be able to visualise the dependence graph of a loop, the source code must be parsed and understood by our tool. For dependence analysis we do not need a fully featured Java compiler, but only a parser that can extract all necessary elements to perform the dependence calculations. Among the components we need are: index variables, their bounds, their increment/decrement, the statements of the loop kernel, access variables and subscript functions for array variables. We can obtain this information by generating an Abstract Syntax Tree (AST) for the source file and extracting from it the relevant information. To generate the AST we employed Java Compiler Compiler (JavaCC) a compiler construction toolkit [13]. JavaCC takes as input a grammar definition for a programming language (here Java) and generates a Java-based parser program as the output. This parser creates an AST for a Java source code file as input, which allows us to easily extract all elements for the dependence analysis. Note that our parser makes certain assumptions, which would not be acceptable for a normal compiler. For example, it assumes that there is no aliasing or overlap of array variables. This is acceptable for a prototype and should be improved for a production version.

A. Dependence tests

After extracting the elements of a loop with the parser, i.e. index variables, accessed variables etc., the large remaining task is to determine which array elements are accessed by each statement and if there is an overlap. Dependence relations caused by scalars are trivial to detect. For arbitrary subscript functions of arrays this is a very difficult problem, as we must determine for which index values, within the bounds of the loop, the subscript functions are equal. Fortunately, in the large majority of programs the subscript functions are linear functions of the index variables, e.g. the examples of Section II, and we can tackle the problem. To illustrate the well known method [9] we employed consider the following code segment.

\[\text{for } I = p \text{ to } q \text{ do} \]
\[S1: \ X(a \cdot I + a0) = \ldots \]
\[S2: \ \ldots \ldots \ldots = X(b \cdot I + b0)\]

Here \(X\) is a one dimensional array and \(p, q, a, a0, b, b0\) are integer constants determinable at compile time. The output variable of statement \(S1\) and input variable of \(S2\), namely the array \(X\), can potentially cause \(S2\) to be flow dependent on \(S1\), or \(S1\) to be antidependence on \(S2\), or both for different values of \(I\). Consider two instances of the index variable \(I = i, j\). Let an instance of the output variable of \(S1\) be \(X(a \cdot i + a0)\) and input variable of \(S2\) be \(X(b \cdot j + b0)\). The two instances will access the same array element, and hence cause a dependence if:

\[a \cdot i - b \cdot j = b0 - a0\]

with

\[p \leq i \leq q\]
\[p \leq j \leq q\]

If a solution exists to this set of equations and inequalities, then a dependence exists, otherwise there is none. For nested loops the number of equalities and equalities increases, but the
principle is the same. Such sets of equations and inequalities for integer variables are called Integer Linear Programs (ILP) and solving ILPs is in general an NP-hard problem. For dependence analysis, however, the problem instances are small in their very large majority, and an optimal solution can quickly be obtained, for example with the Omega test [14] and even further accelerated with other, albeit less precise, techniques [9]. Instead of implementing an Omega solver in our tool, we simply employed GLPK (GNU Linear Programming Kit) [15], an ILP solver. This is as accurate as possible, but not ideal in terms of performance. Still, it suffices for our prototype as only one loop nest is analysed at a time, which the performance evaluation in Section V confirms. In the future better dependence tests and more advanced dependence analysis techniques will further improve the tool [8].

As we are not only interested in whether or not a dependence exists, but also in the dependence distance/vector, we formulate the ILPs in such a way that we obtain the minimum and maximum dependence distance. When minimum and maximum are the same, the dependence in uniform, otherwise it is non-uniform.

V. Evaluation

The essential evaluation criterion for a visual tool is its usability. Before this is addressed, we first need to evaluate the correctness and performance of the tool.

A. Correctness

Our proposed tool is based on well established dependence analysis theory. Further, the selected visualisation as a dependence graph is the common and intuitive method to visualise such dependences. With these premises the evaluation of the correctness of the tool is limited to the evaluation of our implementation. We performed such an evaluation in form of a black-box testing with a large set of loops exploring all distinguished dependences in loops with different nesting levels. The dependence graphs produced by the loop were compared to the expected ones and it was thus verified that the implementation works correctly for this test set.

B. Performance

As our plug-in is a visual tool for an interactive development environment, the performance objective is that the delay for dependence analysis and graph drawing is below the human perception threshold. We tested this by creating again a large set of loops with different nesting levels and number of statements in the loop kernel. Remember that the number of separate loop nests within one source code file is not relevant for the dependence analysis, as this is always limited to the loop nest around the cursor position.

For the very large majority of our test cases, the dependence analysis and the graph drawing was perceived as instantaneous by the testers after the inactivity threshold triggered the update of the dependence graph. Only for loops with more than 3 nesting levels and loop kernels with more than 200 statements a perceivable delay of 1-2 seconds was registered. Also note that the dependence calculation is done in the background in a separate thread, thus not obstructing the user.

In conclusion, this is already a sufficient performance, especially considering that loops with more than 3 nest levels and loops with a large number of statements in the kernel are rather rare. In any case, the results demonstrate the validity of the concept and the potential as a useful tool. Note also that the implementation is a prototype and is easily accelerated with faster dependence tests (see Section IV-A), responsible for the largest fraction of computation time. This performance result is also not unexpected as similar dependence analysis (Omega solver) is performed by modern compilers, e.g. the GNU Compiler Collection (GCC) [5].

C. Usability

To evaluate and analyse the usability in a systematic form, we perform a cognitive dependence evaluation [16], a common Human Computer Interaction (HCI) practise. In this evaluation method, the design is tested against several cognitive dimensions:

- Abstraction gradient – The dependences are represented as a graph, abstracting the source code into a spatial concept. Graphs in general are a well known abstraction notation and dependence graphs in particular are the standard method for dependence representation. Moreover, the target users of the tool are programmers for which graphs are a familiar concept.
- Closeness of mapping – The notation of a dependence graph is a very close and intuitive notation of dependences: the direction of an edge corresponds to the precedence order enforced by the dependence. The tail node must be executed before the head node.
- Consistency – The notations used are consistent throughout: statements are always represented with a node while edges, independent of the type, always represent the dependence between statements.
- Hidden dependency – Intentionally, the steps of the dependence analysis are completely hidden from the user and only the result, i.e. the graph, is shown. The relation between the nodes and the line numbers is clear and can be highlighted by clicking on a node. In the future it is planned to highlight the variables incurring the dependences in the corresponding statements when a dependence connection is clicked.
- Progressive evaluation – The dependences are analysed and depicted as soon as the developer is inactive or completes a line. Displayed are the components that were possible to analyse, while statements that cannot be analysed and unknown dependences are highlighted. Hence, the tool provides useful information at every step of the development, not only for complete loops.
- Viscosity – A ‘reconfigurable’ layout as employed means that the user can move the graphical components (e.g. nodes, connections, etc.) as desired.
- Visibility and Juxtaposibility – The default layout of the nodes matches the sequential ordering of their counterparts (statements) in the source code. Users can reposition
statements and filter the display of dependence types to improve the visibility.

In summary, our proposed tool performs very well against the desired properties of a visual tool.

VI. CONCLUSIONS

We proposed in this paper a visual tool to aid the parallel programmer in the difficult task of dependence analysis in loops. Based on the dependence analysis concepts found in parallelising compilers, the tool displays on-the-fly the dependence graph of the loop statements. Such a tool can support the programmer when developing for a parallelising compiler or when using OpenMP. It is especially beneficial when the dependences can only be partially analysed and the tool support enables the user to take decisions on enforcing parallelisation. The evaluation of the tool revealed satisfactory performance and, more importantly, high usability. In the future we intend to extend the concept of the tool, by implementing dependence elimination and transformation techniques that can be applied to the dependence graph with the corresponding modification of the source code.

REFERENCES