Project exercise: Matrix Multiplication in SCOOP

19 May 2004

Implementing a concurrent object-oriented matrix multiplication application with GUI in SCOOP

The goal of this project is to implement completely three concurrent object-oriented matrix multiplication applications with a graphical user interface (GUI). Below you can see one such possible matrix multiplication application.
Consider two \( n \times n \) matrices \( A \) and \( B \) in figure 1 below (here \( n \) is 3).

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 1
\end{pmatrix}
\begin{pmatrix}
4 & 3 & 2 \\
5 & 1 & 3 \\
6 & 4 & 3
\end{pmatrix}
= 
\begin{pmatrix}
32 & 17 & 17 \\
77 & 41 & 41 \\
122 & 65 & 65
\end{pmatrix}
\]

Figure 1: Matrix multiplication of \( C = A \times B \)

The multiplication of the matrix \( A \) and \( B \) with \( n \) rows and \( n \) columns yield the matrix \( C \), which is also a \( n \times n \) matrix. The goal of this exercise is to write three concurrent matrix multiplication application with different algorithms using SCOOP which can multiply such two \( n \times n \) (where \( n > 0 \)) matrices \( A \) and \( B \) and display the result in a third matrix \( C \).

Algorithms for matrix multiplication

According to [1] there are basically two kinds of algorithms for matrix multiplication: iterative parallelism relying on shared variables and the use of so-called peers for distributed matrix multiplication relying on message passing. Please note, that in SCOOP the distinction between shared variables and message passing is not made explicitly. Below you will find the description of one matrix multiplication algorithm for iterative parallelism and two algorithms for distributed programming. Your goal is to implement all of these three algorithms. The algorithms below are all presented in a pseudo-language which should be straightforward to read and understand, not unlike Pascal or Modula-2. All three algorithms below are taken from [1] and are described there in more detail.

Algorithm 1: Iterative Parallelism

You can find the complete description of this algorithm in [1], pages 13 to 16. Consider again two \( n \times n \) matrices \( A \) and \( B \) and a result matrix \( C \). We can declare three variables as following:

\[
\text{integer } a \ [n, n], \ b \ [n, n], \ c \ [n, n]
\]

The indices range from 0 to \( n - 1 \). Note that in Eiffel indices of an array can start with an arbitrary integer number.

The sequential matrix multiplication can be implemented with the following algorithm:

```plaintext
for [i = 0 to n - 1] {
    for [j = 0 to n - 1] {
        -- compute inner product of \( a[i, \ast] \) and \( b[\ast, j] \)
        c[i, j] := 0
        for [k = 0 to n - 1] {
            c[i, j] := c[i, j] + a[i, k] * b[k, j]
        }
    }
}
```

[1] notices that two operations are independent if the write set of each is disjoint from both the read and write sets of the other. The read set of an operation contains the
variables it reads but does not alter, and similarly the write set of an operation contains the variables that it alters and possibly also reads. If two operations are independent, they can be then executed in parallel. For matrix multiplication, the computations of inner products are independent operations.

The concurrent algorithm can be implemented in the following way:

```c
co [i = 0 to n - 1, j = 0 to n - 1] { -- all rows and all columns
  c[i, j] := 0
  for [k = 0 to n - 1] {
    c[i, j] := c[i, j] + a[i, k] * b[k, j]
  }
}
```

In the above concurrent algorithm the new keyword `co` for concurrent execution is introduced instead of the two “for loops” of the sequential algorithm. This means that for all `i` iterating from 0 to `n – 1` and also for `j` iterating from 0 to `n – 1` separate activities executing the body of the `co` statement will be launched. Altogether `n \times n` activities (in SCOOP context processors) will be executed in parallel.

Algorithm 2: Distributed Matrix Multiplication (coordinator/worker interaction)

You can find the complete description of this algorithm in [1], pages 23 to 24. This algorithm and the next one (Algorithm 3) rely on message passing. As in the first algorithm we assume two `n \times n` matrices `A` and `B` and a result matrix `C`. We can declare three variables as following:

```c
integer a[n, n], b[n, n], c[n, n]
```

This algorithm employs a coordinator process and an array of independent worker processes as it is illustrated in the figure 2 below.

```
Figure 2: Coordinator/worker interaction
```

The keyword `process` (used below in the code of the algorithm) allows introducing processes. Worker process `i` computes row `i` of result matrix `c`. In order to compute it, it needs row `i` of source matrix `a` and the entire source matrix `b`. Each worker receives these values from a separate coordinator process. The worker then computes its row of results and sends them back to the coordinator. The `send` and `receive` statements, which are also used below in the code are message passing primitives.
The code for the workers is given below:

```
process worker [i = 0 to n – 1] {
    integer a [n]  -- row i of matrix a
    integer b [n, n] -- all of matrix b
    integer c [n]  -- row i of matrix c
    receive initial values for vector a and matrix b
    for [j = 0 to n – 1] {
        c [j] := 0
        for [k = 0 to n – 1] {
            c [j] := c [j] + a [k] * b [k, j]
        }
    }
    send result vector c to the coordinator process
}
```

Below you will find the code for the coordinator process:

```
process coordinator
    integer a [n, n]  -- source matrix a
    integer b [n, n] -- source matrix b
    integer c [n, n]  -- source matrix c
    initialize a and b
    for [i = 0 to n – 1] {
        send row i of a to worker [i]
        send all of b to worker [i]
    } 
    for [i = 0 to n – 1]
        receive row i of c from worker [i]
        print the results, which are shown now in matrix c
}
```

The coordinator process initiates the computation and gathers and prints the results. The coordinator also first sends each worker the appropriate row of \( a \) and all of \( b \). Then the coordinator waits to receive a row of \( c \) from every worker.

Algorithm 3: Distributed Matrix Multiplication (circular pipeline/peers)

You can find the complete description of this algorithm in [1], pages 24 to 26. As in the first and second algorithms we assume two \( n \times n \) matrices \( A \) and \( B \) and a result matrix \( C \). We can declare three variables as usual as following:

```
integer a [n, n], b [n, n], c [n, n]
```

Contrary to the second algorithm, this algorithm assumes that each worker has only one column of \( b \) at a time instead of the entire matrix. In order for worker \( i \) to compute all of row \( i \) of matrix \( c \), it has to acquire all columns of matrix \( b \), which can be done — as illustrated in the figure 3 below — by using a circular pipeline and circulating the columns among the worker processes.
Below you will find the code for the worker process:

```eiffel
process worker [i = 0 to n – 1] {
    integer a [n]  -- row i of matrix a
    integer b [n]  -- one column of matrix b
    integer c [n]  -- row i of matrix c
    integer sum = 0 -- storage for inner products
    integer next_col = i -- next column of results
    receive row i of matrix a and column i of matrix b
    compute  c [i, i] = a [i, *] * b [* , i]
    for [k = 0 to n – 1] {
        sum := sum + a [k] * b [k]
        c [next_col] := sum
    } -- circulate columns and compute rest of c [i, *]
    for [j = 1 to n – 1] {
        send my column of b to the next worker
        receive a new column of b from the previous worker
        sum := 0
        for [k = 0 to n – 1]
            sum := sum + a [k] * b [k]
        if (next_col = 0)
            next_col := n – 1
        else
            next_col := next_col – 1
        end if
        c [next_col] := sum
    }
    send result vector c to the coordinator process
}
```

One should note that the coordinator process used in the algorithm above is just responsible for displaying the result vector and has no other important functionality, so the name process might be confusing. This algorithm employs an interprocess relationship that is called *interacting peers* (or just peers).

**Organizational issues of the project exercise**

The definitive deadline for the submission of the project is the **28.06.2004 9:00 am local time**. You should send all the source code (but without the EIFGEN folder and the .epr file of an EiffelStudio project) of all your projects and also all the documentation (PDF format, MS Word format) in a single zipped file via email to: Volkan.Arslan@inf.ethz.ch. The projects should be done by groups of two people. It
can be also done individually. It is recommended to create one Eiffel project for each algorithm, since the graphical visualization of the different algorithms will be different for each one. Nevertheless — if you want — you can write one application and implement all the three algorithms in that one application.

Grading of the project

The grading of the project will be done according to several criteria which are listed below:

- Quality of the design (according to [3]):
  - Extendibility
  - Re-usability
  - Minimalism
- Quality of the code:
  - Quality of contracts (Design by contract: preconditions, postconditions, invariants, loop variants and loop invariants, check assertions)
  - Respect of style guidelines (formatting, etc.)
  - Presence of detailed comments for class indexes and features.
- Documentation:
  - User guide: describes how to use the application (can be short)
  - Developer guide: describes the architecture (BON method recommended), main classes, the relationship between concurrent (separate) classes, limitations of the application, and how to extend it. Also the difficulties encountered during the design and implementation of the application
- Functional and non-functional correctness:
  - Correct result of the matrix multiplication computation
  - Correct implementation of the algorithms regarding their specification
  - General visual quality of the application (GUI)

It should be noted, that the applications should reflect the concurrent matrix multiplication visually, but the GUI part of the application is not the most important part regarding the grading. The emphasis of the grading lies definitively on the correctness and the quality of the design.

You can find most of the aspects listed above in [3]. The Eiffel Style guidelines are described in detail in chapter 26 *A sense of style* of [3]. Also the EiffelStudio 5.4 tool can format compiled Eiffel classes according to the *Eiffel Style guideline* when the *Clickable view* button in the Editor tool is pressed. It is recommended to use the BON method for the architecture documentation of the applications. You can find an example using BON about the architecture of a conference management system in the BON book [6] on pages 231-269. Especially the relationships between classes and clusters (client-supplier and inheritance relationship) are important which are illustrated in the diagram on page 267 and the complete static architecture is illustrated on page 268. You should have at least these both kinds of diagrams in your architecture documentation.

Technical setup and project information

As a language Eiffel will be used. You can find an introduction to Eiffel “Eiffel: The Essentials” at

http://se.inf.ethz.ch/teaching/ss2004/0004/slides/eiffel_the_essentials.pdf
and also in [3]. An introduction to BON — Business Object Notation — can be found in [6] or online at

http://www.bon-method.com

The “concurrency” of the application should be achieved by using the SCOOP model. The Eiffel library called SCOOPLI implements most functionality of the SCOOP model. The documentation of SCOOPLI can be found in [5] or online at

http://se.inf.ethz.ch/people/arslan/data/scoop/workshops/Plzen_2003/SCOOPLI_on_.NET.pdf

and in [4] or online at

http://www2.inf.ethz.ch/~meyer/publications/concurrency/scoop_ieu.pdf

The GUI of the application should be programmed using the portable GUI library EiffelVision2 (see the online documentation at http://docs.eiffel.com/ or the shipped documentation for all libraries of EiffelStudio in general and EiffelVision2 in particular) shipped with the integrated development environment (IDE) EiffelStudio 5.4. There is an interactive Vision 2 Tour application (vision2_tour.exe) shipped with EiffelStudio 5.4 in the folder $ISE_EIFFEL\vision2_tour\spec\windows\bin\, where $ISE_EIFFEL denotes the installation folder of EiffelStudio 5.4. The html file at $ISE_EIFFEL\vision2_tour\readme.html describes how to use the interactive Vision 2 Tour application.

You can download a free version of EiffelStudio 5.4 from

http://www.eiffel.com/downloads/

for Windows and .NET. Please make sure that you download and use only the Windows version of EiffelStudio 5.4, since the SCOOPLI library relies on the .NET Framework. If you don’t have the .NET Framework 1.1, you can download it at


If you have already Visual Studio.NET then very likely the .NET Framework is already installed. Additionally you will need a C compiler (even if we produce .NET IL code, since some libraries of EiffelStudio rely on external C libraries) either the Borland C compiler which is shipped with EiffelStudio 5.4 and which you can install during the installation of EiffelStudio 5.4 or the Microsoft C compiler which is not shipped with EiffelStudio, but for example in Visual Studio .NET.

For applying the MVC (Model View Controller) architecture, the event library described in [2] and available online at


can be used.

REFERENCES


