Software Engineering for Parallelism: An Exercise in Separating Design and Implementation

Maarten R. van Steen [a] , Henk J. Sips [b] and Hai-Xiang Lin [c]

[a]Erasmus University (Woudestein), Department of Computer Science, P.O. Box 1738, 3000 DR Rotterdam, e-mail: steen@cs.few.eur.nl

[b]Delft University of Technology, Department of Computational Physics, P.O. Box 5046, 2600 GA Delft, e-mail: henk@cp.tn.tudelft.nl

[c]Delft University of Technology, Department of Applied Mathematics and Informatics, P.O. Box 5031, 2600 GA, Delft, e-mail: lin@pa.twi.tudelft.nl

1. INTRODUCTION

The topic of how to exploit parallelism for performance enhancement of applications has already received considerable attention. In most cases, the subject has been treated in the context of algorithm development and scalability. In particular, attention has primarily focussed on *what* we should do in order to achieve improved performance. The question of *how* solutions should be implemented has been more or less been dealt with as an afterthought. The effect of this approach is not very surprising: to date, parallel implementations tend to be highly machine-dependent and often violate basic rules of well-structured software just for the sake of retaining efficiency.

If we are to come to a point that parallel application development can be considered a mature technology, it seems imperative that our attention is to be directed towards the development of methods and techniques that adhere to standard software engineering criteria. Fortunately, proposals in this direction are underway. For example, the Fortran community is now no longer seeking solutions in the form of basic programming environments, but has recognized that something needs to be done at the language level [4,10]. In addition, efforts are being undertaken for supporting the design phase of parallel applications [6,11]. In this paper we describe an approach followed within the *parTool* project [3]. In Section 2 we present our view on current practice, followed by a description of an alternative approach in Section 3. To illustrate the issues of concern, we discuss a real-world example in Section 4 and show how many problems can be avoided by our approach. We conclude our discussion in Section 5.

2. DESIGN VERSUS IMPLEMENTATION

When the development of parallel applications is more than just parallelization of a relatively simple sequential algorithm, such as in production applications, we are confronted with an elementary design problem. On the one hand, we need to design the application in such a way that dependencies with a target (parallel) implementation environment are reduced to the

minimum for these dependencies seriously affect the portability of our application. On the other hand, it is exactly this parallel environment which we identified in the first place as a solution to a non-functional requirement: getting performance. In other words, we are confronted with the problem of having to combine *portability* and *efficiency*. As it turns out, we have to cope with the fact that *design* and *implementation* of parallel solutions are more dependent on each other than in the case of sequential solutions.

From this perspective, there are two extreme solutions: (1) integrate design and implementation entirely, or (2) separate the two completely.

Integrating design and implementation

In practice, this has been the approach followed so far. Integration in this sense means that an implementation is considered to be a design, or *vice versa*. Implementations as designs are mainly achieved by assuming some type of target machine (shared-memory, distributed-memory), and subsequently using a commonly applied communication library. Although portability is achieved to a certain extent, it is obvious that we are dealing with, at the best, a very *low-level* design. Designs as implementations are very popular in those cases that the problem exhibits "natural" parallelism. Concurrent object-oriented applications are an exponent of this approach. However, developers then fail to make the important distinction between parallelism as a means for *modeling* a solution, and as a means for *implementing* it. What we see then, for example, is that considerable efforts are being spent on designing language concepts that support modeling parallel solutions, but which at the same time can be implemented efficiently (see e.g. [9]). However, this does assume something as a common implementation environment, which may be the case for sequential computers, but which is certainly not the case for parallel computers.

Separating design and implementation

This approach has been followed in the *parTool* project. In this case, one effectively develops a design based on an abstract machine model. Deriving an implementation from such a design means that an explicit mapping will have to be provided from that abstract machine to a target machine. This involves two aspects: (1) mapping data and computations onto respectively, memories and processors, and (2) providing schedules for data and computations. The important thing is, that mappings and schedules can be provided as *separate annotations* to a design. Deriving an implementation then reduces to the provision of annotations to a design. This approach is sketched in Figure 1, and is also the one which has been adopted in *parTool*. We shall elaborate on this separation of concerns in the next section.

3. THE *parTool* APPROACH

In order to facilitate the development of parallel solutions in a machine-independent fashion, we have devised two different programming techniques. One technique, realized in the programming language called *Booster*, is primarily used to design and implement numerical algorithms. We will not discuss Booster in this paper – the interested reader is referred to [8]. The second technique, and which we shall briefly present here, is referred to as *transaction-based programming*, or TBP for short [2,3]. As we shall explain below, the model underlying TBP is that of a machine executing transactions which act on associative storages. The important aspect of both Booster and TBP, is that in order to attain efficient solutions for a specific parallel machine,

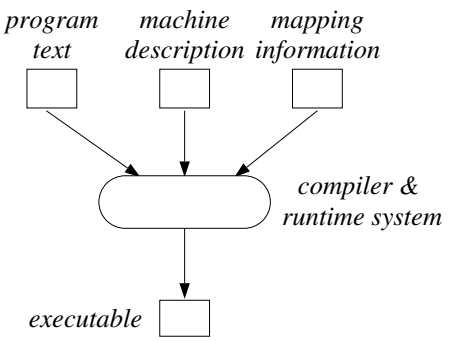

Figure 1. A development and execution architecture for parallel programs.

programs are *annotated* with mappings, expressing where and when computations should take place, as well information on the actual organization and placement of data objects.

3.1. Transaction-based programming

In order to explain how TBP actually works, we shall use a simple example. Consider an indexed set **I** of N numbers $\{(x,i)|0 \leq i < N\}$ that needs to be sorted, in the sense that

$$\forall (x,i),(y,i) \in \mathbf{I} :: x \leq y \Leftrightarrow i \leq j \tag{1}$$

Devising a parallel solution expressed in TBP is straightforward. The basic idea is to construct one or more *transactions* that act on a common storage place, in our case initially containing the unordered set **I**. For example, sorting this set can be described by means of a single transaction[1]:

transaction swap(i, j)
    select (a,i), (b,j) **where** $a < b$ **and** $i > j$
    insert (a,j), (b,i)
end

This transaction *removes* precisely two elements from the indexed set that do not meet criterion (1). This removal is expressed as a conditional **select** statement. The condition for removing the specified elements is expressed in a **where** clause. Instead of actually removing elements, it is also possible to specify that an element should only be read. In this case, a question-mark is used, such as, for example **select** ?(a,i).) If the selection succeeds, a transaction swap will subsequently insert the two elements back into the indexed set, but now with swapped indices. This is expressed by means of the **insert** statement.

[1]For the sake of simplicity we shall use an informal syntax to denote TBP programs. Also, we shall concentrate on the essentials only, implying initialization and description of storage place is omitted.

In order to understand the working of program the following should be noted. In the first place, transactions are executed *auto-concurrently*: if a transaction succeeds in selecting data objects, a new transaction is immediately spawned which, in turn, will attempt to access the storage place as well. If this transaction fails, then, of course, there is no need to spawn a next one. In addition, transactions are executed *atomically*, i.e. without mutual interference. Consequently, it is seen that as many transactions as possible will attempt to swap elements as necessary without ever interfering with each other. As soon as no more transactions can be spawned, implying that there is no more work to be done, it can be readily proved that the set **I** has been sorted.

The important issue of our solution, however, is that it merely states *what* should be done (namely sort the indexed set **I**). We have not described *how* sorting is to proceed – this is considered by us as an implementation issue that should be dealt with by means of annotations.

3.2. Annotating a program with mapping information

Although the program can exploit parallelism in abundance, this need not imply that this will also be done in the most efficient way. In particular, in order to maintain the atomicity property, it may turn out that an actual implementation of TBP uses a conservative scheduling mechanism as such an implementation must support *any* TBP program. It is here that a developer may provide additional, application-specific information in order to realize efficiency. By specifying a so-called *mapping*, a developer can add explicit control-flow information to a program.

For example, assume that we want to execute our sorting program according to an odd-even strategy which consists of two phases. During the first phase, all even-indexed elements are compared to their respective next element, and if necessary, swapped. The second phase yields that all odd-indexed elements are compared to their respective neighbor. Obviously, during each phase all comparisons and possible swaps can be done in parallel. Moreover, it is easy to see that these operations are completely independent, so that any atomicity property is automatically maintained. Assuming that N is odd, we can add the following *schedule* to our original program:

```
schedule odd-even
    while true do
        pardo( 0 ≤ k < N / 2 ) execute swap( 2k, 2k + 1 )
        pardo( 0 ≤ k < N / 2 ) execute swap( 2k + 1, 2(k + 1) )
    endwhile
end
```

The schedule odd-even is provided as additional information to a compiler: it provides information on *how* a solution is to be obtained. In our case, the mapping states that the runtime system should first execute all transactions swap($2k$,$2k + 1$) *in parallel*, and after that, continue with the transactions swap($2k + 1$,$2(k + 1)$). This sequence of parallel executions is to be repeated until there is no more work to do. It is important to note that execution of a transaction may still fail, despite that it has been explicitly scheduled. As soon as *all* scheduled transactions fail, the program terminates. This means that the runtime system is still responsible for termination detection. Alternatively, we could have provided explicit termination criteria for the runtime system (in particular, criteria denoting when the set is ordered).

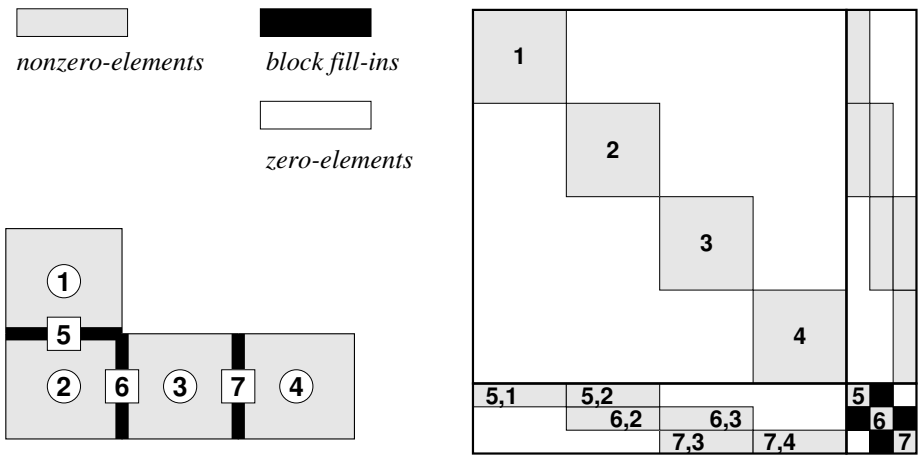

Figure 2. A decomposition of a structure into 4 subdomains (1–4), yielding 3 interfaces (5–7), and its resulting sparse DBBD matrix.

4. PARALLEL BLOCK FACTORIZATION

One of the application-oriented subprojects within *parTool* concerned the parallelization of the finite-element system DIANA[2] [1]. DIANA is a general-purpose FEM system based on the displacement method. The package comprises a large number of program modules that support analyses of linear statics, eigenvalues, stability, fracture mechanics, potential flows, as well as nonlinear analysis and dynamic analysis. The part that we are concerned about here, is the parallelization of the direct solver as described in [7]. At the heart of the parallel solver we find an intelligent domain decomposer which produces a so-called *double bordered block diagonal* (DBBD) matrix, which is a rather complex sparse data structure. As an example, Figure 2 shows a simple domain (and its decomposition) and its resulting matrix.

4.1. The basic algorithm

The main issue here is to factorize the matrix into a form $L \cdot D \cdot L^T$, where L is a lower triangular block matrix with unit diagonals, and D a diagonal (block) matrix. In practice this means that the following two sets of equations need to be solved:

$$L_{ii} \cdot D_{ii} \cdot L_{ij}^T = A_{ii} - \sum_{k \in S(i)} L_{ik} \cdot D_{kk} \cdot L_{ik}^T, (1 \le i \le N_{\text{sub}} + N_{\text{int}}) \quad (2)$$

$$L_{ij} = (A_{ij} - \sum_{k \in S(i) \cap S(j)} L_{ik} \cdot D_{kk} \cdot L_{jk}^T) \cdot L_{jj}^{-T} \cdot D_{jj}^{-1}, (1 \le j \prec i \le N_{\text{sub}} + N_{\text{int}}) \quad (3)$$

A_{ij} denotes a submatrix of the DBBD-matrix A. For example, each of the shaded regions shown in Figure 2(b) denotes a (non-zero) submatrix of A. The number of subdomains and interfaces is denoted as N_{sub} and N_{int},respectively. Each set $S(i)$ denotes the indices that are relevant for

[2]DIANA is a registered trademark of the TNO Institute for Building and Construction Research.

the computation. In particular, we have, regardless of the actual matrix, that

$$S(i) = \{k | 1 \leq k \prec i\}$$

Now, of course, equation (2) need only be carried out for all non-zero block matrices L_{ik}, and similarly, the summation in equation (3) need only be carried out for non-zero L_{ik} and L_{jk}^T. Anticipating our further discussion, we identify these non-zero block matrices by means of the sets $C(i)$ which are defined as:

$$C(i) = \{1 \leq k \prec i | L_{ik} \text{ is non-zero } \} \subseteq S(i)$$

The important thing to note is that contrary to $S(i)$, $C(i)$ is problem-specific and needs to be calculated separately for each domain decomposition. We shall return to this later.

Solving the above sets of equations by means of a TBP-program is actually quite straightforward. A few things should be kept in mind, though. In the first place, transactions should always represent basic operations. In other words, if several computations need to be done with implicit or explicit data dependencies between them, then each computation should be expressed by a single transaction. For example, equation 2 actually consists of two data dependent computations: (1) calculate a sum, and (2) factorize the result. Such dependencies should be translated into two different transactions. In our case, we can use the following specifications:

transaction sum-1(i, k)
  select A_{ii}, ?L_{ik}, ?D_{kk} **where** $k \prec i$
  compute $A_{ii} \leftarrow A_{ii} - L_{ik} \cdot D_{kk} \cdot L_{ik}^T$
  insert A_{ii}
end

transaction factorize-1(i)
  select ?A_{ii}, L_{ii}, D_{ii}
  factorize $L_{ii} \cdot D_{ii} \cdot L_{ii}^T = A_{ii}$
  insert L_{ii}, D_{ii}
end

Repeated execution of transaction sum-1(i,k) will eventually yield that the block matrix A_{ii} is replaced by

$$A_{ii} \leftarrow A_{ii} - \sum_k L_{ik} \cdot D_{kk} \cdot L_{ik}^T$$

In the meantime, transaction factorize-1(i) will continue factorizing A_{ii}, thereby removing previous calculated matrices L_{ii} and D_{ii} and replacing them accordingly. In the end, the correct factorization will have taken place. Note that as long as the complete sum is not available, all calculated values for L_{ii} and D_{ii} are, in fact, irrelevant.

A similar approach is followed for the set of equations 3, which also yields two transactions:

transaction sum-2(i, j, k)
select A_{ij}, ?L_{ik}, ?D_{kk} **where** $k < \min\{i,j\}$
compute $A_{ij} \leftarrow A_{ij} - L_{ik} \cdot D_{kk} \cdot L_{jk}^T$
insert A_{ij}
end

transaction factorize-2(i, j)
select ?A_{ij}, L_{ij}, ?L_{jj}, ?D_{jj} **where** $j < i$
compute $L_{ij} \leftarrow A_{ij} \cdot L_{jj}^{-T} \cdot D_{jj}^{-1}$
insert L_{ij}
end

A few remarks about this program are in order. In the first place, note again that, analogous to transaction factorize-1, L_{ij} is repeatedly replaced by transaction factorize-2, eventually resulting in the correct (infinitely repeated) calculation of L_{ij}. In the second place, one must realize that a transaction can never succeed before all its input is available. In particular, this means that no two transactions sum-1(i,k_1) and sum-1(i,k_2) (with $k_1 \neq k_2$) can ever execute simultaneously as both select A_{ii} for removal from the (not further specified) storage place. A similar reasoning shows that no two transactions sum-2(i,j,k_1) and sum-2(i,j,k_2) can ever execute at the same time[3]. In the third place, one may question if the program sketched so far will ever stop. In fact, it will not: as can be verified, transactions factorize-1 and factorize-2 will continue to factorize the matrix A, even when this does not yield new results. In order not to diverge too much from our original problem, we shall leave program termination for what it is.

4.2. Towards efficiency

In order to obtain an *efficient* solution, the original factorization scheme needed to be completely refined. In effect, in order to combine efficiency and portability, the actual implementation of the factorization scheme as devised in [7], makes explicit use of a dependence graph. This graphis constructed in advanced and gradually updated as operations such as expressed by the transactions above, are executed. In other words, a combination of efficiency and portability is obtained through a form of adaptive scheduling of operations while the factorization is being performed.

This strategy can be followed for transactions as well. The main problem with our solution is that the execution mechanism underlying the TBP program will have to determine when transactions can be selected for execution. This is a costly operation. Alternatively, we can explicitly provide scheduling information to avoid almost all runtime-decisions. This information is used by the compiler to adapt the runtime system in the sense that the generic transaction scheduler is replaced with a specific one.

To start with, we can easily derive an efficient (partial) schedule for computing the sets of equations (2) and (3), respectively.

[3]It should be noted, however, that we could have exploited parallelism even further by calculating both sums according to some parallel algorithm, such as recursive doubling.

```
schedule sum-factor-1( i )
    for( 1 ≤ k ≺ i ) execute sum-1(i,k);
    execute factorize-1(i);
end

schedule sum-factor-2( i,j )
    for( 1 ≤ k ≺ min{i,j} ) execute sum-2( i,j,k );
    execute factorize-2( i,j );
end
```

Each schedule describes that we should first compute the complete sum before doing the actual factorization. We now at least have some improvement, for we can simply add this mapping information to our program as follows:

```
schedule parallel-block-factorization
    pardo( 1 ≤ i ≤ N_sub + N_int )
        sum-factor-1( i );
        pardo( 1 ≤ j ≺ i )
            sum-factor-2( i, j );
        endpar
    endpar
end
```

Effectively resulting in the *sequential* execution of each pair of transactions (sum-1(i),factorize-1(i)) and (sum-2(i,j),factorize-2(i,j)), respectively, but *parallel* execution of all these pairs taken together. Note that any data-dependency is always preserved regardless of the mapping information. In other words, if we explicitly schedule a transaction that cannot yet execute on account of the lack of input, then execution will be postponed by the runtime system.

But, of course, we can do better. In particular, it should also be possible to avoid any computations involving zero-valued block matrices. This may seem as a rather modest type of optimization, but in fact, practice shows that for the type of problems for which the parallel block factorization was originally devised, more than 95 % of the submatrices of the matrix A constructed after domain decomposition was actually completely zero-valued. Consequently, it is even *mandatory* to avoid zero-valued block matrices in the computation. To this aim, we need the sets $C(i)$ as introduced previously, and which are defined as:

$$C(i) = \{1 \leq k \prec i | L_{ik} \text{ is non-zero } \} \subseteq S(i)$$

Using these sets, we can now devise an *adaptive* scheduling, in the sense that the scheduling behavior is now dependent on the problem-instance for which the program has originally been designed. To this aim, we need only slightly modify our mapping information as follows:

```
schedule sum-factor-1( i )
    for( k ∈ C(i) ) execute sum-1(i,k);
    execute factorize-1(i);
end

schedule sum-factor-2( i,j )
    for( k ∈ C(i) ∩ C(j) ) execute sum-2( i,j,k );
    execute factorize-2( i,j );
end

pardo( 1 ≤ i ≤ N_sub + N_int )
    sum-factor-1( i );
    pardo( j ∈ C(i) )
        sum-factor-2( i, j );
    endpar
endpar
```

Apart from some minor details, this is in fact a description of the algorithm as devised in [7].

5. CONCLUSIONS

Software engineering for parallel applications is still in its infancy, and much research is needed to provide workable solutions. The approach of separating design and implementation issues by introducing annotations may be such a solution. But, of course, there is more to it than just this. In particular, we also need to devise methods and techniques that allow us to combine various forms of parallelism. An important issue that should also be addressed and which we have purposefully neglected in this paper, is the design of (dynamic) datastructures that are suitable for parallel implementation environments. In fact, designing such a datastructure, was a major research effort with respect to the parallelization of DIANA.

REFERENCES

1. R. de Borst, G.M.A. Kusters, P. Nauta, and F.C. de Witte. "diana – a comprehensive but flexible finite element system." In C.A. Brebbia, (ed.), *Finite Element System Handbook*. Springer-Verlag, Berlin, 1985.
2. E. de Jong. *Transaction-based Programming*. Ph.D. thesis, Department of Computer Science, University of Leiden, December 1992.
3. E. de Jong, E.H.R.M. Paalvast, H.J. Sips, and M.R. van Steen. "High-Level Specification Tools for Parallel Application Development." In P. Dewilde and J. Vandewalle, (ed.), *Proceedings CompEuro '92*, pp. 163–168, The Hague, 1992. IEEE.
4. HPF Forum. "High Performance Fortran Language Specification." Technical Report, Rice University, Houston, Texas, May 1993.
5. L.H. Jamieson. *Characterizing Parallel Algorithms*, In L.H. Jamieson and D.B. Gannon and R.J. Douglass, (ed.), *The Characteristics of Parallel Algorithms*, chapter 3, pp. 65–100. MIT Press, Cambridge, Mass., 1987.

6. I.E. Jelly, I. Gorton, and J. Gray. "Using PARSE for Transputer Software Development." In R. Grebe and J. Hektor and S. Hilton and M.R. Jane and P.H. Welch, (ed.), *Transputer Applications and Systems*, volume 2. IOS Press, Amsterdam, 1993.
7. H.X. Lin. *A Methodology for the Parallel Direct Solution of Finite Element Systems*. Ph.D. thesis, Delft University of Technology, Department of Applied Mathematics and Computer Science, October 1993.
8. E.M. Paalvast, H.J. Sips, and L.C. Breebaart. "Booster: a high-level language for portable parallel algorithms." *Applied Numerical Mathematics*, 8:177–192, 1991.
9. M. Papathomas. "Concurrency Issues in Object-Oriented Languages." In D. Tsichritzis, (ed.), *Object-Oriented Development*, pp. 207–246. University of Geneva, July 1989.
10. PCF. "PCF Fortran, Version 3.0." Technical Report ANSI X3H5, Kuck & Associates, Champaign, Ill., April 1990.
11. C. Steigner, R. Joostema, and C. Groove. "PAR-SDL: Software Design and Implementation for Transputer Systems." In R. Grebe and J. Hektor and S. Hilton and M.R. Jane and P.H. Welch, (ed.), *Transputer Applications and Systems*, volume 2. IOS Press, Amsterdam, 1993.