CHEMICAL ENGINEERING TRANSACTIONS
VOL. 57, 2017
A publication of AIDIC
The Italian Association of Chemical Engineering
Online at www.aidic.it/cet
Guest Editors: Sauro Pierucci, Jiří Jaromír Klemeš, Laura Piazza, Serafim Bakalis

ISBN 978-88-95608- 48-8; **ISSN** 2283-9216

A Bottom-up Approach for Parallelizing CAPE Software

Karim Alloula[a], Jean-Pierre Belaud[a,b], Loïc Sanroma[c], Jean-Marc Le Lann[a,b]

[a] Université de Toulouse, INPT-ENSIACET, 4, allée Emile Monso, BP 44362, 31030 Toulouse Cedex 4, France
[b] CNRS, Laboratoire de Génie Chimique (UMR 5503), 4, allée Emile Monso, BP 44362, 31030 Toulouse Cedex 4, France
[c]Toulouse School of Economics, CNRS, University of Toulouse,21, allée de Brienne, 31015 Toulouse, France
Karim.Alloula@ensiacet.fr

One way to parallelize a CAPE software is to simulate unit operations simultaneously. This parallelization at the upper level -the flow sheet level- takes place usually in the context of distributed memory parallelism, where distinct computation nodes execute different pieces of code and exchange data with each other by means of a communication network. Taking advantage of today shared memory architectures, we suggest parallelizing the simulation codes from the lowest level -mathematical expressions and control structures- up to the upper level -the process-. Such an approach requires that the pieces of code to be parallelized should be written in an OpenMP compliant language, and that the source codes may be slightly modified. Assuming that these technical constraints are satisfied, from this systematic and progressive way to parallelize a CAPE software one can obtain an interesting speedup.

1. Introduction

For quite a long time, the CAPE community is interested in increasing the performance of the process simulators. This interest comes from the need for taking into account some new phenomena, while refining the description of the phenomena already simulated. The overhead associated to such an enriched simulation has to be counterbalanced by means of improved numerical methods, and more efficient hardware architectures. Those methods and architectures reach their summit within the context of parallel computing. In part two, this paper makes a very brief review of parallel computing which, in chemical engineering, is commonly applied at the flow sheet level. Part three introduces a way to use a shared memory parallelism when evaluating any instruction, set of instructions, or model within CAPE software. The simulation of a MTBE synthesis model validates this bottom-up approach.

2. Parallel computing at the flow sheet level

To summarize, when dealing with chemical engineering problems, the techniques coming from high performance computing (HPC) are mainly applied at the flow sheet level.
For a sequential modular simulation, several unit operations are simulated at the same time on a unique computer, or on distinct computers. In such a case, there is a need for a master task in charge of scheduling the elemental simulations, and forwarding the results of some unit operation to another, or to some others (Laganier et al., 1993).
For an equation-oriented simulation, a numerical solver running in a parallel environment handles the aggregated equations, either static or dynamic. A master task drives several other elemental tasks, each one being in charge of solving a sub problem on a computing node. The task definition, scheduling, and communication depend mainly on numerical considerations. This is the case with multi-frontal methods for solving linear systems, or with domain decomposition methods for solving partial differential equations.
For today's multi-physics simulations, several codes, each one devoted to a physical modeling, collaborate for simulating systems. Preisig and Lobo (2013) introduced the coupler technology providing « the infrastructure to bring such pieces to talk together and form a computation environment assembled for a specific purpose ». They detailed the case of nanotechnology where specialized simulation softwares are required from « very many different fields including quantum chemistry, molecular dynamics, micro-fluids, meso-scale population-

based behaviour simulations all the way to macroscopic flows and capacity behaviour, plant and control design ». Like sequential modular simulation, multi-physics simulation takes advantage of parallel computing at the “flow-sheet” level. However, in a multi-physics simulation the word “flow-sheet” has a quite different meaning than in a sequential modular simulation: the specialized simulation software replaces the unit operations, and the sets of information required and produced by the standalone pieces of software replace the pipes.
In all the cases, the parallel environment is made of computing nodes, a communication network for inter-node communication, and some additional software -the middleware- in charge of interfacing the process simulation code with the computing hardware. Today, the Message Passing Interface (MPI) is the main de facto standard for developing new parallel applications spread over several computing nodes. This middleware, devoted to coarse-grained parallelism, has been successfully applied to chemical engineering applications, as reported by Chen Z. et al. (2013).
Other parallelization models, such as grid computing (Papadopoulos and Linke, 2009), are not commonly used in the CAPE community. However, Gautier and Hamidi (2007) reports an attempt to exploit parallelism on services, provided by CAPE-OPEN compliant components.

3. Parallel computing at the expression level

3.1 Main principles

The previous part summarized the use of some parallelization models in an inter-node context. The last few years were characterized by a renewed interest in intra-node parallelism, where several tasks are executed simultaneously in a graphical process unit or by one computing node, either on one, two, up to eight multi-core processors. This paper suggests a new way to take advantage of one of the most common type of intra-node parallelism: shared memory parallelism. Instead of dealing with an inter-node parallelism at the process level, we investigate the benefits of parallelizing on each computing node the lowest level task: expression evaluation.
Any piece of code, either an arithmetic expression, an assignment, a control structure, a routine, or the whole program, can be viewed as a tree which leaves are the lexical tokens of the programming language -numbers, variables, operators or key words- and where nodes are subexpressions. Consequently, the sequential execution of any straight-line piece of code (without loops, tests, gotos, …) is a depth-first traversal process of this parsing tree. More generally, the sequential execution of any piece of code (with loops, tests, gotos, …) is a depth-first traversal process of some tree, built dynamically from the parsing tree. In order to take advantage of a shared memory parallelism during code execution, an obvious way seems to evaluate simultaneously all the nodes at the same level during the tree traversal process. Unfortunately, such an approach has to deal with at least two extra issues coming with parallel computing: data dependence and load balancing (we omit talking about synchronization and race conditions). The data dependence is the fact that, even in a parallel computing context, some variable values have to be computed first by some instructions before being used by some others. This paper does not address this obstacle for parallelization.
We focus on the load-balancing problem, coming from the fact that the evaluation costs associated to each node in the parsing tree may be very different due to subexpression differences in length, depth and/or semantics. Instead of distributing the subexpression evaluations to a thread pool, each thread evaluating immediately a fixed number of subexpressions, we create tasks for the deferred evaluation of some subexpressions, queue these tasks, and dispatch them to threads for execution as soon as they become inactive. Introducing a queuing mechanism between the producers -the evaluation tasks to be executed- and the consumers -the threads in charge of executing these tasks-, we hope that the load-balancing problem will be treated efficiently.
This task mechanism may solve the load-balancing problem, but it requires working with the tricky task and thread features. In order to hide this additional complexity to the CAPE software developers, the suggested approach is not applied directly to the original code, but to a corresponding parsing tree structure built automatically from the original code. The programming language keywords and operators are overloaded to create the nodes and leaves of this tree and to change the evaluation semantics. The overloading mechanism allows us to make tasks evaluate subexpressions in parallel. It also extends the semantics domain of the original code. From the original mathematical expressions, operator and function overloading build functional expressions to which formal transforms can be applied (Alloula et al., 2009). We no longer work with a numerical calculation system but with a computer algebra system. The most important formal transform this calculation system comes with is derivation, which allows obtaining very accurate Jacobian or Hessian matrices improving the numerical solving procedures.

3.2 Expected benefits

Our intent is to offer the CAPE software developer a means to reduce the computation time of some simulation code by spreading expression evaluations over the several cores available in a computing node, a computing node being a personal computer or a shared memory element in a HPC cluster. The main benefit of the presented approach is to hide from the developer the entire burden related to shared memory parallelization. The CAPE software engineer should not have to know how to parallelize certain pieces of code. He, or she, subcontracts this work to a dedicated software library.
The instructions in the original code are not modified. Thanks to the overloading mechanisms provided by the CAPE software programming language, the sequential instructions seamlessly become parallel instructions. However, because one should be able to indicate precisely which pieces of straight-line code should be parallelized, the overloading, and consequently the parallelization, takes place only for instructions involving variables of some predefined type. This bottom-up approach where, progressively, sequential instructions become instructions which subparts are executed in parallel, should improve the computing performance continuously.

3.3 Technical requirements

In order to obtain the aforementioned benefits, the programming language used for simulation should support both, a shared memory-programming model and overloading facilities. For the time being, we make the choice to parallelize only codes written with the FORTRAN language. Starting with its 90 version, FORTRAN comes with assignment and operator overloading facilities, and provides shared-memory features through the OpenMP directives, data types, environment variables and functions. Another choice could have been to deal with codes written in C++, because this language comes also with overloading and OpenMP. Our choice of FORTRAN, rather than C++, is justified by the language of the underlying library providing the parallel facilities: eXMSL, the computer algebra system introduced in Alloula et al. (2009). eXMSL is written in FORTRAN 95, so it can be naturally interfaced with codes written in the same programming language. However, the main principles of our approach could be implemented with codes written in C++ if the parallel evaluation process, the overloaded functions and the overloaded operators were coded in this language.
In the functional programming context of a computer algebra system, the OpenMP **`TASK`** feature provides the means for seamlessly distributing sub expressions evaluations over the execution cores. Because eXMSL is in charge of the whole evaluation process, nearly all the task mechanism is hidden from the chemical engineer who programs models. Only one set of OpenMP directives appears at the beginning of the program, and is closed at the end of the program as illustrated in Figure 1. Initially, a parallel region is created using the **`PARALLEL`** directive. This directive starts a team of threads, which will be in charge of parallel executions. However, the **`SINGLE`** directive specifies that only one thread in the team executes the enclosed code. This single thread executes one task: calling the procedures `Validate_Abs`, `Validate_And`, … enclosed within the **`TASK`** directive. Therefore, from the point of view of the chemical engineer the whole execution flow seems to be sequential!

```
!$OMP PARALLEL
!$OMP SINGLE
!$OMP TASK
        CALL Validate_Abs
        CALL Validate_And
        CALL Validate_Append
        CALL Validate_Apply
        CALL Validate_ArcCos
        CALL Validate_ArcCosh
        CALL Validate_ArcSin
!$OMP END TASK
!$OMP END SINGLE
!$OMP END PARALLEL
```

Figure 1: OpenMP directives to be placed in the end user code.

Of course, this is not the case. Behind the scene, hidden to the end user, other OpenMP directives appear inside the eXMSL library. These directives will take advantage of the threads created initially by the **`PARALLEL`** directive for executing simultaneously several tasks, that is to say blocks of instructions together with their associated data. Figure 2 shows some of the OpenMP directives included in the eXMSL library, just to give a flavor of the asynchronous execution mechanism based on tasks. **`TASK`** directives create as many tasks as

the number of parts in the expression to be evaluated: one task for the expression head, and one task for each subexpression. Those tasks become available for further execution by the pool of threads. The **TASKWAIT** directive insures that all the expression parts are evaluated before returning the evaluation result. Notice that the number of tasks, i.e. the expression length plus one, and the number of threads are independent.

```
IF (traverse_heads) THEN
!$OMP TASK DEFAULT(SHARED)
       CALL setHead(transformedExpression, depthFirstTraversalLeaves1(factory,
getHead_(expr), traverse_heads, transform))
!$OMP END TASK
ELSE
       CALL setHead(transformedExpression, getHead_(expr))
END IF
DO  index = 1, length
!$OMP TASK DEFAULT(SHARED) FIRSTPRIVATE(index)
       CALL setSubExpression(transformedExpression, index,
depthFirstTraversalLeaves1(factory, getSubExpression_(expr, index), traverse_heads,
transform))
!$OMP END TASK
END DO
!$OMP TASKWAIT
```

Figure 2: OpenMP directives driving the parallel execution in the eXMSL library.

The parallel execution is possible only because the eXMSL library handles a data structure mapping the instructions in the end user code. A parallel evaluation process of the data structure replaces the sequential evaluation of the original instructions. In order to build this data structure by means of the assignment and operator overloading, the declaration part of the end user code has to be modified: for a given set of instructions to be parallelized, all the numerical variables, previously declared as integers, reals or complexes, become symbols declared with the eXMSL-defined type `Expression`. Figure 3 details how the variable declarations are modified. The first instruction `USE Exmsl_` imports the eXMSL library features in the user code, the most important feature being the derived type `Expression`. `R` and `xs` variables, declared with the FORTRAN predefined types `REAL` or `DOUBLE PRECISION` in the original sequential code, are now declared as references to `Expression` data. `x` variable, declared as a real array in the original sequential code, is now declared as a reference to an `Expression` array. Initially, `R`, `xs` and `x` are null references. The `link` instructions attach them to expressions. Here, those expressions are symbols. From those symbols, thanks to overloading mechanisms, composite expressions are built automatically. For example, from `x(1)+x(4) == 3.0D0`, eXMSL builds a composite expression made of two symbols `x(1)` and `x(4)`, one real number `3.0D0`, and two operators `+` and `==`. The built-in FORTRAN `+` operator has been overloaded to produce an `Expression` reference `x(1)+x(4)` from the two `Expression` arguments `x(1)` and `x(4)`. The built-in `==` operator has been overloaded to produce an `Expression` reference `x(1)+x(4) == 3.0D0` from the `Expression` argument `x(1)+x(4)` and the `REAL` argument `3.0D0`. Several equality expressions are built the same way and are referenced by the different elements of an explicit array. Then, the `FindRoot` eXMSL function takes three array arguments and builds a new `Expression` which head is the predefined `FindRoot` symbol and which parts are respectively the system of nonlinear equations, the unknown variables and the initial conditions. Finally, the reference to the `FindRoot` composite expression is transmitted to `evaluationStep`, the main eXMSL evaluation function, in charge of the parallel process explained before.
To summarize, from Figure 3 one can notice that the modifications to the original sequential code occur in the declaration part. In the instruction part, syntax remains unchanged but, thanks to the overloading mechanisms, the semantics is enriched:

- Expression parts are evaluated in parallel.
- Mathematical models are formulated in a more user-friendly manner. For example, to solve a system of nonlinear equations, one can directly write a list of equality expressions instead of a residual function.
- In addition to the numerical evaluation features coming by default with the FORTRAN language, the end user can take also advantage of the several computer algebra facilities coming with the eXMSL library like differentiation, operations on sets, replacements and so on.

```
PROGRAM CombustionOfPropaneInAir

USE Exmsl_

IMPLICIT NONE

TYPE(Expression), POINTER :: R => NULL(), xs => NULL()
TYPE(Expression), DIMENSION(:), POINTER :: x => NULL()

CALL prolog
!$OMP PARALLEL
!$OMP SINGLE
!$OMP TASK
CALL link(R, Symbol('R'))
CALL link(x, Symbol('x', 10))
CALL link(xs, Symbol('xs'))
CALL evaluationStep( &
                   FindRoot( &
                             (/ &
                             x(1)+x(4) == 3.0D0, &
                             2*x(1)+x(2)+x(4)+x(7)+x(8)+x(9)+2*x(10) == R, &
                             2*x(2)+2*x(5)+x(6)+x(7) == 8, &
                             2*x(3)+x(5) == 4*R, &
                             x(1)*x(5) == 0.193D0*x(2)*x(4), &
                             x(6)*SQRT(x(2)) == 0.002597D0*SQRT(x(2)*x(4)*xs), &
                             x(7)*SQRT(x(4)) == 0.003448D0*SQRT(x(1)*x(4)*xs), &
                             x(8)*x(4) == 1.799D-5*x(2)*xs, &
                             x(9)*x(4) == 2.155D-4*x(1)*SQRT(x(3)*xs), &
                             x(10)*x(4)**2 == 3.846D-5*xs*x(4)**2, &
                             R == 40.0D0, &
                             xs == SUM(x) &
                             /), &
                             (/ R, x, xs /), &
                             (/ 0.0D0, 3.0D0, 4.0D0, 80.0D0, 0.00236450211271867 9D0,
0.0006D0, 0.0013D0, 0.06D0, 3.5D0, 26.4D0, 0.004D0, 100.0D0 /) &
                                                ) &
                   )
CALL unlink(R)
CALL unlink(x)
CALL unlink(xs)
!$OMP END TASK
!$OMP END SINGLE
!$OMP END PARALLEL
CALL epilog

END PROGRAM CombustionOfPropaneInAir
```

Figure 3: Modifying variable declarations to take advantage of the eXMSL library facilities.

3.4 Pitfalls and drawbacks

From the previous presentation, our approach seems to be seductive because when applied to an original code the modifications are few and furthermore they are located in the declaration parts only. However, as usual, pitfalls and drawbacks appear when entering into the details.

The presented approach suggests to do parallel computing at the expression level so, by default, at the instruction level. This kind of “fine-grained parallelism” is implemented using OpenMP tasks. Consequently, our approach can result in a too finely grained task parallelism. As detailed in Schmidl et al. (2012), “if the execution time of a task is very small, this overhead can consume more CPU cycles than the task’s actual execution. In this case, it would be more efficient to execute the task’s body immediately without separating it into a task”. Our answer to this pitfall is to make our instructions bigger by grouping several independent instructions into one list of expressions, which is executed in parallel.

Because parallelization is achieved at the expression level, our approach works seamlessly with the control structures, loops or conditions, present in the original sequential code. However, in order to make instructions bigger, the end user can take advantage of the control structure overloading provided by eXMSL, which allows converting control structures in expressions too.

4. Application to the MTBE synthesis

4.1 Presentation

The case study consists in simulating a reactive Rayleigh distillation model, which couples kinetics and thermodynamics. Thermodynamic data and chemical constants are taken from Chen F. at al. (2002). Alloula et al. (2013) details an index reduction method, transforming this index two differential algebraic system to an index one system. The simulation times obtained during this previous work, where any computation was sequential, are compared with a parallel version of the simulation code.

4.2 Main results

The most important result for this case study is that the speed-up, associated to the shared memory parallelism introduced before, is almost linear for a number of cores up to twenty-four. Such an improvement was made possible because the mathematical expressions involved in such a problem were big enough, and because we finely tuned the task granularity, avoiding unnecessarily nested tasks. In the general case, when the proposed parallelization is applied as introduced in section 3, a value of the expected speed-up cannot be guessed before runtime. However, as a rule of thumb, we can state that the bigger the mathematical expressions in the original code, the higher the speed-up obtained with the parallelized code.

5. Conclusions

An innovative approach to exploit an intra-node parallelism in CAPE software has been detailed. While successfully implemented into a legacy calculation system, where everything is an expression made of sub expressions, it requires some extra work to deal with most of the existing simulators coded using procedural or object-oriented languages. Fortunately, this extra work needs not to be achieved in one shot: first, one can move only the lowest levels (mathematical expressions, thermodynamic models …) to a parallel implementation, and then tackle the top levels (unit operations and flow sheet).
From the case study, a promising result has been obtained: the overhead generated by the OpenMP library, and by some of the specific features of our calculation system (garbage collection, common sub-expression sharing and cache mechanism), remains small when compared to the speed-up coming from the parallel evaluations of expressions parts. Furthermore, this intra-node parallelization can be efficiently combined with an inter-node parallelization at the flow sheet level, leading to a hybrid model of programming CAPE codes.

Reference

Alloula K., Monfreda F., Théry Hétreux R., Belaud J.-P., 2013, Converting DAE model to ODE models: application to reactive Rayleigh distillation, Chemical Engineering Transactions, 32, 1315-1320, DOI: 10.3303/CET1332220

Alloula K., Belaud J.-P., Le Lann J.-M., 2009, A co-operative model combining computer algebra and numerical calculation for simulation, Computer-Aided Chemical Engineering, 26, 889-894.

Chen F., Huss R. S., Doherty M. F., Malone M. F., 2002, Multiple steady states in reactive distillation: kinetic effects, Computers & Chemical Engineering, 26, 81-93.

Chen Z., Chen X., Shao Z., Yao Z., Biegler L. T., 2013, Parallel calculation methods for molecular weight distribution of batch free radical polymerization, Computers & Chemical Engineering, 48, 175-186.

Gautier T., Hamidi H.-R., 2007, Re-scheduling invocations of services for RPC grids, Computer Languages, Systems & Structures, 33, Issues 3–4, 168-178.

Laganier F. S., Le Lann J.-M., Joulia X., Koehret B., Morari M., 1993, Simultaneous modular dynamic simulation: Application to interconnected distillation columns, Computers & Chemical Engineering, 17, Supplement 1, S287-S297.

Papadopoulos A. I., Linke P., 2009, A decision support grid for integrated molecular solvent design and chemical process selection, Computers & Chemical Engineering, 33, Issue 1, 72-87.

Preisig H.A., Lobo S.C., 2013, The role of couplers in chemical model-based engineering, Chemical Engineering Transactions, 32, 1405-1410, DOI: 10.3303/CET1332235

Schmidl D., Philippen P., Lorenz D., Rossel C., Geimer M., an Mey D., Mohr B., Wolf F., 2012, Performance analysis techniques for task-based OpenMP applications, IWOMP'12 Proceedings of the 8th international conference on OpenMP in a Heterogeneous World, 196-209, DOI: 10.1007/978-3-642-30961-8_15