YAO: a generator of parallel code for variational data assimilation applications

Luigi Nardi*†, Fouad Badran†, Pierre Fortin‡ and Sylvie Thiria*

* LOCEAN, Laboratoire d'Océanographie et du Climat: Expérimentations et approches numériques.
UMR 7159 CNRS / IRD / Université Pierre et Marie Curie / MNHN.
Institut Pierre Simon Laplace. 4, place Jussieu Paris 75005, France.
† CEDRIC, Centre d'Etude et De Recherche en Informatique du CNAM. EA 1395,
292 rue St Martin Paris 75003, France.
‡ UPMC Univ Paris 06 and CNRS UMR 7606, LIP6, 4 place Jussieu, F-75252, Paris cedex 05, France

Abstract—**Variational data assimilation consists in estimating control parameters of a numerical model in order to minimize the misfit between the forecast values and the actual observations. The YAO framework is a code generator that facilitates, especially for the adjoint model, the writing and the generation of a variational data assimilation program for a given numerical application. In this paper we present how the modular graph specific to YAO enables the automatic and efficient parallelization of the generated code with OpenMP on shared memory architectures. Thanks to this modular graph we are also able to completely avoid the data race conditions (write/write conflicts). Performance tests with actual applications demonstrates good speedups on a multicore CPU.**

Keywords: data assimilation; automatic parallelization; shared memory architectures; OpenMP; dependence graph; numerical model; adjoint model

I. INTRODUCTION

Numerical models are widely used for studying physical phenomena. Most of the time, the model is used to forecast or analyze the evolution of the phenomenon. Since the model is imperfect, discrepancy between its forecast values and actual observations may be important due to model parametrization, numerical discretization, uncertainty on the initial conditions and boundary conditions. A new method, the so-called *data assimilation* [1], which uses both the numerical model of the phenomenon and the inverse problem method has been introduced to reduce this discrepancy. Data assimilation uses actual observations to constrain the control parameters (initial conditions, model parameters, ...) in order to force the numerical model (thereafter also referred to as the *direct* model) to reproduce the desired behavior. In *variational data assimilation* methods [2], this task is done by minimizing, with respect to control parameters, a cost function J which measures the misfit between the direct numerical model outputs and the observations. The minimization is done by the use of a gradient method, which requires calculating the gradient of J as a function of the control parameters. The gradient computation requires the product of the transpose Jacobian matrix of the direct numerical model with the derivative vector of J defined at the observation points. This product is also called *adjoint model*. Since the direct numerical model is usually very complex, the implementation of the programming code which represents the adjoint model is very complicated.

The YAO framework already presented in [3], [4] is a code generator dedicated to variational data assimilation. With YAO the user defines, using specific directives and C programming, the type of discretization and the specification of the numerical model. It then generates automatically the numerical and the adjoint model codes via C++ object-oriented programming. YAO has already been used with success on several actual applications in oceanography: *Shallow-water* [3], [4], *Marine acoustics* [5], [6], *Ocean color* [7], *PISCES* [8], *GYRE* configuration of *NEMO* [9].

The YAO formalism is based on a *modular graph*, which is similar to those used in automatic parallelization of nested loops. In this well-developed research field several concepts and algorithms have been introduced, which allow the analysis of nested loops, their decomposition and fusion [10], [11], [12]. The decomposition obtained is well suited to a multi-thread parallelization on shared memory architectures where no communications are required. In this paper, we show how the YAO modular graph enables us to integrate and adapt these algorithms in order to identify the available parallelism, and to allow the automatic generation of parallel code with YAO while completely avoiding the data race conditions (write/write conflicts). With the OpenMP directives, it is then possible to generate a multi-threaded parallel code that runs efficiently on shared memory architectures. This is an important improvement over the previous version of YAO [3] which can generate only sequential code. A large community in geophysics may thus automatically and transparently exploit decades of research in automatic parallelization and benefit from important speedups in computation times on multicore architectures without any additional effort and without any knowledge in parallel programming.

For the automatic generation of parallel code, the development of algorithms specific to YAO is necessary. Indeed, the software tools for automatic parallelization with OpenMP directives have specific constraints related to their design, and can therefore currently not be integrated in the YAO generator. For example, the CAPO toolkit [13] supports only Fortran

and relies on user interaction to improve the parallelization process. The Gaspard2 framework [14] enables automatic OpenMP code generation, but the available parallelism must be first specified by the user in an UML model. The PLuTo tool [15] can efficiently parallelize nested loops while taking into account, via tiling, data locality on multicore architectures with complex hierarchical memory. However PLuTo does not currently support object-oriented programming for input source codes and it has specific limitations (only SCoP programs with pure function calls, no dynamic branch conditions) that also prevents a direct integration in YAO. Finally and most importantly, as detailed further there are data race conditions (write/write conflicts) in the generated code. These data race conditions prevent any automatic parallelization from such tools according to their own data-dependency analysis. To our knowledge, none of these tools can automatically insert (for example) OpenMP *atomic* directives to avoid these race conditions and thus enable the parallelization. We here show how to efficiently accomplish this in YAO thanks to its modular graph.

Adapting state-of-the-art algorithms to YAO while relying on its modular graph has also several advantages. First, there is no constraint on the application code written by the user. Second, a high-level dependency graph is directly provided by the modular graph which enables to avoid the data-dependency analysis, to naturally obtain a coarse grain parallelism, and to possibly scale on real-life applications with thousands of statements.

In the following, we will first give a brief overview of the YAO framework in section II. Then, in section III we will show how the YAO modular graph can be used to automatically and efficiently parallelize the generated code on shared memory architectures. Performance results for two actual YAO applications on a multicore CPU are detailed in section IV. Finally, in section V concluding remarks are presented and future work discussed.

II. YAO PRESENTATION

A. The modular graph

We present here the concept of modular graph, which is fundamental in YAO, as well as the *forward* and *backward* procedures: more details can be found in [3], [4]. We first define the following terms.

- A *module* is an entity of computation; it receives inputs from other modules or from an external context[1] and it transmits outputs to other modules or to an external context.
- A *connection* is a transmission of data from a module to another or between a module and an external context.
- A *modular graph* is a data flow graph composed by a set of several interconnected modules; it summarizes the sequential order of the computations.

---

[1]An external context is an entity which initializes and retrieves the computation of certain modules.

In order to perform data assimilation, at each time step a modular graph is traversed by the *forward* procedure and then by the *backward* procedure.

1) The forward procedure: the input data set of a module F_p is a vector denoted $\mathbf{x}_p$ and its output data set is a vector denoted $\mathbf{y}_p$ ($\mathbf{y}_p = F_p(\mathbf{x}_p)$). As a consequence, a module F_p can be executed only if its input vector $\mathbf{x}_p$ has already been processed, which implies that all its predecessor modules have been executed beforehand. Thus there are only *flow* dependencies [10] between modules. Since we suppose that the modular graph is acyclic, it is then possible to find a module ordering, i.e. a topological order, which allows us to correctly propagate the calculation through the graph. If we denote by $\mathbf{x}$ the vector corresponding to all the graph input data, provided by the external context, the *forward* procedure enables to calculate the vector $\mathbf{y}$ corresponding to all the graph output values. The modular graph defines an overall function Γ and makes it possible to compute $\mathbf{y} = \Gamma(\mathbf{x})$. The function Γ has a physical meaning: it represents a direct numerical model M, with respect to the YAO formalism. The *forward* procedure allows us to compute the outputs of the numerical model according to its inputs. The incoming connections from the external context of Γ could be, for example, initializations or boundary conditions. Outgoing connections transmit their values to compute, as an example, a cost function.

2) The backward procedure: this procedure enables the computation of the adjoint of the cost function J with respect to the control parameters. We suppose that for each module F_p, with an input vector $\mathbf{x}_p$ and receiving in its *output data points* a "perturbation" vector $\mathbf{dy}_p$, we can compute the matrix product $\mathbf{dx}_p = \mathbf{F}_p^T \mathbf{dy}_p$, $\mathbf{F}_p^T$ being the transposed Jacobian matrix of the module F_p calculated at point $\mathbf{x}_p$. It is possible [4] to compute the gradient of J with respect to the control parameters by traversing the modular graph in a reverse topological order and executing local computations on each module in order to compute $\mathbf{dx}_p$. It has been shown [3], [4] that this reversed traversal leads to a back propagation on the modular graph characterised by additions (accumulations) of several local computations. Each of these additions is computed in an intermediate step and then back propagated. Thus there are *flow* and *output* dependencies [10].

3) YAO formalism: running simulations or data assimilations using an operational numerical model M requires the definition of a modular graph representing the sequence of the computations. A numerical model operates on a discrete grid, where the physical process is computed at each grid point I and at each time step t. As the phenomenon under study is quite the same at each grid point, only the modular subgraph representing a grid point is needed. YAO obtains Γ by duplicating this subgraph for each I and t.

In the YAO formalism, the user must define a set of basic functions $\{F_1, F_2, \ldots F_k\}$ which has to be applied to each grid point I and at each time step t. The user has to define also the dependencies between these functions. From this information, YAO generates the overall modular graph Γ. The modules of the modular subgraph $\Gamma_{I,t}$ are denoted by $F_p(I, t)$, where I

(a)					(b)
ctin F_1 from F_1	i	j-1	t-1		order YA1
ctin F_2 from F_1	i	j+1	t		order YA2
ctin F_2 from F_3	i-1	j+1	t		F_1 F_3
ctin F_2 from F_3	i	j	t-1		
ctin F_2 from F_4	i+1	j	t		
ctin F_3 from F_1	i-1	j	t-1		order YB1
ctin F_4 from F_3	i	j	t		order YB2
ctin F_4 from F_2	i	j+1	t		F_2 F_4

Fig. 1. (a) Part of the specification language used by the user with 2D space modules. The second *ctin* directive specifies the connection from F_1 at point (i,j+1,t) to F_2 at point (i,j,t). (b) The *order* directives indicate the ordering in which we compute the functions F_p and the ordering in which we traverse the grid.

Fig. 2. RDG issued by the *ctin* directives of Fig. 1a.

represents a grid point (1D, 2D or 3D), t is a time step and F_p a basic function. Thus, a module is the computation of the function F_p at grid point I and at time t. We denote by i (respectively j and k) the indices of the first axis (resp. the second and third axis). An edge from a source module $F_s(I', t')$ to a destination module $F_d(I, t)$ corresponds to a data transmission from $F_s(I', t')$ to $F_d(I, t)$ (s may be equal to d).

The modular graph is similar to the Expanded Dependence Graph (EDG) used for the parallelism detection in nested loops [10]. The main difference with the modular graph being that in the EDG the nodes represent one operation (the instance of a statement) while the nodes of the modular graph are a set of operations (the instance of a function composed by a set of statements) represented by the module $F_p(I, t)$. Thus, the granularity of the nodes differs. In practice the dimension of a YAO basic function depends on the application and on the user design. In general a YAO module has some dozens of statements but in particular cases it may be very much larger.

B. User specifications and code generation

This section presents two YAO directives, *ctin* and *order*, on which relies the YAO automatic code generation. These directives generate nested loops, which allow us to traverse the modules $F_p(I, t)$ in the correct ordering.

1) order and ctin directives: the *ctin* directive has the following syntax "**ctin from** F_s **to** F_d **list of coordinates**". Such a directive represents one edge (or connection) of the modular graph which is then automatically replicated by YAO on space and time. *list of coordinates* represents, for a generic point I and time step t of the destination module F_d, the point I' and the time t' of the source module F_s (with $t \geq t'$). The difference $I' - I$ represents the *distance vector* of the source module with respect to the destination module; we denote its coordinates by d_i, d_j or d_k. The user has to specify in the *list of coordinates* the distance vector and $d_t = t' - t$ as a function of the generic point I of the destination module, which is the same in all connections. Fig. 1a gives an example of *ctin* directives.

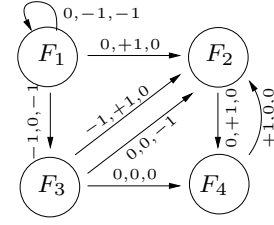Every *ctin* directive generates an edge from F_s to F_d labeled

by the distance vector $I' - I$ and $t' - t$. The resulting graph is a directed multigraph[2] which represents all dependencies between basic functions. This multigraph corresponds to the Reduced Dependence Graph (RDG) [10] used for the automatic generation of parallelism in nested loops[3]. Fig. 2 presents the RDG of the former example. Since the space dimension is two, the edges are labeled by (d_i, d_j, d_t) which indicates that the destination module, at time t and at point (i, j), takes its inputs from the source module at time $t + d_t$ and point $(i + d_i, j + d_j)$ with $d_i, d_j \in \mathbb{Z}$ and $d_t \in \mathbb{Z}_{\leq 0}$.

The YAO *order* directive allows the user to define a traversal of the modular graph following a topological order. This directive allows to visit all the grid points of the space, and permits the generation of nested loops. The user specifies one *order* directive for each dimension of the space. Thus, a program generated by YAO, contains an outermost loop, that represents the time, within which the user defines, thanks to the *order* directives, the different loops that allows the traversal of the space for each time step. In general, we have several ways to traverse a space. In the *order* directive, *YA1* (*YAO Afterward axis 1*) means that we are managing the i loop and we go along this axis in an ascendant way. *YA2* means the same but for the j axis, whereas *YB1* (*YAO Backward axis 1*) means that we go along the i axis in a descendant way. Fig. 1b gives an example of such *order* directives.

An *order* directive is coherent if and only if it ensures the correct computation of the functions it contains. YAO has an internal procedure which verifies the coherence, with respect to the RDG, of the *order* directives. In the following, we thus assume that all the *order* directives specified by the user are coherent.

The different *order* directives are executed in a sequential way in the scheduling specified by the user. The outermost loop (ascendant or descendant) of one *order* directive corresponds to one axis; in the remainder we use l to denote this axis and r the number of basic functions contained in this outermost loop. As explained in [4], if l is ascendant necessarily all the distances d_l of the edges between the r modules are negative or null due to the coherence hypothesis. Symmetrically if l is descendant all the distances d_l are positive or null.

---

[2]A directed multigraph is a graph with multiple parallel edges.

[3]As for the analogy between the EDG and the modular graph, the RDG has one statement per node while the YAO RDG has a basic function (a set of statements) per node.

```
loop i ascendant
  loop j ascendant
    F₁(i,j,t).forward(F₁(i,j-1,t-1))
    F₃(i,j,t).forward(F₁(i-1,j,t-1))

loop i descendant
  loop j ascendant
    F₂(i,j,t).forward(F₁(i,j+1,t), F₃(i-1,j+1,t),
                      F₃(i,j,t-1), F₄(i+1,j,t))
    F₄(i,j,t).forward(F₃(i,j,t), F₂(i,j+1,t))
```

Fig. 3.   YAO generator translation of the directives of Figs. 1a and 1b.

Fig. 4.   Addition, represented by the symbol $\sum$, in the back propagation of the *backward* procedure. These two connections represent a data transfer between two time steps. The two modules F_3 and F_1 perform the transfer towards F_1 at time step $t - 1$. This partial graph example case is given by the YAO directives shown in Fig. 1.

2) Generation of the forward and backward procedures: in Fig. 3 we give the translation, performed by the YAO code generator, of the *ctin* and *order* directives given in Figs. 1a and 1b. This represents the translation, in a pseudo code language, of the *forward* procedure. Each *order* directive generates one loop, one for each dimension of the space. The way we traverse the axes, ascendant or descendant, and the scheduling of the modules are the ones specified in the *order* directives. For each object of each module class F_p the local *forward* function (a C++ method) is called with the output of its predecessor modules as inputs. The local *forward* functions are defined, for each basic function, by the user. It has to be noticed that all *forward* functions are thread-safe because they compute the result with respect to the generic grid point I, as shown in Fig. 3. The nested loops allow us to compute the output of the modules for all the grid points and for one time step. An overall loop, not shown in the figure, which allows us to traverse the time steps in an incremental order t, $t + 1$, $t + 2$, etc., encompasses all the local *forward* functions. The time loop may be considered as a computation barrier where at current time t, all the computations for time $t' < t$ are done.

As presented in section II-A2, the *backward* procedure traverses the modular graph in a reverse topological order. For ease of presentation we do not detail the pseudo code of the *backward* procedure as we did for the *forward* procedure. These are very similar. However it is important to point out the addition (accumulation) in the back propagation, explained in [3], [4] and specific to the *backward* procedure. This accumulation results in *output* dependencies which may arise between two time steps. This computation is briefly explained in Fig. 4. The y_p variables ($p \in 1, 2, 3$) are the outputs of the local *forward* functions. The propagation allows us to provide the predecessor module computations to the successor modules. On the other side, the back propagation allows us to back propagate the gradient of J by using the Jacobian matrix (J_p in the figure) for computing dx_p. The back propagation of several dx_p (dx_3 and dx_2 in Fig. 4) which have the same predecessor enforces the addition of the dx_p (the symbol $\sum$ in the figure).

III. ALGORITHM FOR AUTOMATIC PARALLELIZATION

A. Parallelization of the forward procedure

In section II we have noted an interesting similarity between the YAO formalism and the theories of compilation and of automatic parallelization of nested loops [10]. Thanks to this
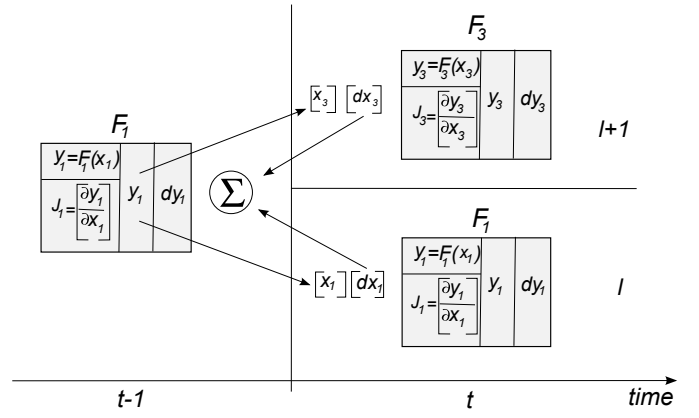
similarity we can adapt these techniques and algorithms to the YAO automatic code generator. We thus propose here to integrate and adapt such algorithms in order to automatically parallelize the *forward* procedure generated by YAO on shared memory architectures with multi-thread programming. No communication is required and we just have to maximize the number of parallel loops. Because of the strong time dependencies in all data assimilation applications the temporal loop is not parallelized and we focus on data parallelism at each time step. The domain decomposition between threads is performed as a 1D block distribution on the space and we rely on a static load balancing since in all the current YAO applications the computation load of each module is constant for each grid point. Our goal is thus to label, as "parallel" or "not parallel", each outermost *order* directive so that the corresponding loop can be generated as parallel or sequential in the final code (with OpenMP directives). In order to maintain the coherence hypothesis of one given nest of *order* directives we opted not to change or invert the *order* defined by the user. However we can still use techniques such as a loop distributions possibly followed by loop fusions in order to detect the maximum available parallelism and to reduce the synchronization points.

Since the temporal loop is not considered in the parallelization algorithm, the edges whose $t' - t$ are negative can be removed from the RDG. The remaining graph is shown in Fig. 5, we denote it $\overline{\mathrm{RDG}}$. This is obtained by removing all $dt = -1$ connections and writing only the signs of the distance vector components. Thus $(0, +)$ means a distance vector equal to $(0, +1)$.

Considering some nested *order* directives which have as outermost axis l and a connection from F_s to F_d we consider a connection as *critical* with respect to these nested order directives if:

- F_s and F_d are contained by the nested directives,
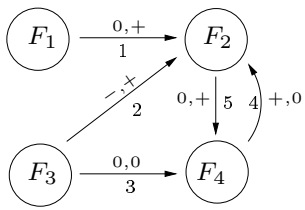- $d_t = 0$ and $d_l \neq 0$.

Fig. 5. $\overline{\text{RDG}}$ obtained by simpli-
fication of the RDG of Fig. 2. The
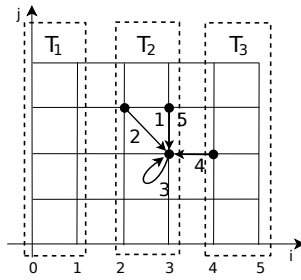edges are numbered from 1 to 5.

Fig. 6. *Flow* dependencies between
3 threads T_1, T_2 and T_3. Same edge
numbers as in Fig. 5.

The analysis of the $\overline{\text{RDG}}$ highlights the critical connections
which prevents parallelization because of *flow* dependencies
between threads, as presented in Fig. 6. The connection from
F_4 to F_2 and from F_3 to F_2 (edges #2 and #4 in Figs. 5
and 6) results in a *flow* dependency between the couples of
threads (T_1, T_2) and (T_2, T_3) because $d_l \neq 0$ (in this example
l is the i axis). This is not the case for the connections #1,
3 and 5 because the two corresponding grid points belong to
the domain computed by one thread, as shown in Fig. 6. The
connection #2 is *not critical*, since F_3 and F_2 are not in the
same nested loops (see Figs. 1b and 3). In this example only
connection #4 is *critical*.

For the analysis of one outermost loop l composed by
functions F_1, F_2, ..., F_r we consider the subgraph G_l of
the $\overline{\text{RDG}}$ limited to the r basic functions and the edges
between them. On the edges of this subgraph we retain only
information concerning the distance d_l [4]. As far as the d_l value
is concerned we retain only the sign of d_l, $(-, +)$ if $d_l \neq 0$
and 0 if $d_l = 0$. The analysis of G_l allows to decompose the
loop in several loops preserving the computation coherence
hypothesis. Taking into account that the *forward* functions are
thread-safe, we can apply the Allen-Kennedy algorithm [12]
to decompose the loop in parallel loops as follows.

- Calculate the Strongly Connected Components (SCCs) of
  G_l.
- Consider the reduced Directed Acyclic Graph (DAG),
  denoted by $G_{l/SCC}$, by shrinking each SCC down to a
  single vertex and by drawing one, and only one, edge
  between two SCCs if there is at least one edge from the
  first to the second in the graph G_l. If at least one of the
  edge in G_l which connects these two SCCs is labeled
  by non 0 (that is to say either $-$ or $+$), then label the
  corresponding edge in $G_{l/SCC}$ by this value. Else, if all
  the labels are 0, then label the corresponding edge in
  $G_{l/SCC}$ by 0.
- Sort in a topological order the $G_{l/SCC}$ graph and enu-
  merate all the SCCs following this order. For each SCC
  generate an l loop which computes its basic functions.

This decomposition is a maximum loop distribution of the

---

[4]All the distances d_l in G_l are either ≤ 0 or ≥ 0 if the l loop is respectively
ascendant or descendant, as they correspond to the same outermost loop.

---

initial loop, in other words we can not further decompose
without breaking the coherence hypothesis.

We can analyze each SCC loop in order to see if we
can perform a domain decomposition on the l axis. For a
particular SCC we consider all the edges of the graph G_l
between two basic functions being part of this SCC. If at
least one of these connections is labeled by $+$ or $-$, namely
if $d_l \neq 0$, the SCC is considered to be not parallelizable. The
loop is parallelizable if all these edges are labeled by 0. In
other words it is parallelizable if it does not contain any *flow*
dependency between threads. We label by p and $\bar{p}$ the loops
which are respectively parallelizable and not parallelizable.
Such maximum loop distribution gives the largest number of
parallel loops. The critical connections of the $\overline{\text{RDG}}$ have been
minimized.

B. Reducing synchronization points

The previous section applies the Allen-Kennedy algorithm
to YAO thanks to the analogies between the EDG and the
modular graph. This algorithm allows us to automatically
label as parallel or not the SCCs resulting in maximum
loop distribution. As far as performance is concerned, this
loop distribution is not the best solution because it increases
the number of synchronization points. Following Kennedy-
McKinley [11] it is possible to propose a loop fusion algorithm
that will reduce the number of synchronization points. As the
$G_{l/SCC}$ is a DAG, we can reorganize the SCCs in levels. The
levels are numbered from $k = 1$ to $k = M_{level}$ where M_{level}
is the maximum number of levels. The first level, $k = 1$,
contains the SCCs without predecessors; the predecessors of
a SCC at level k, with $k > 1$, are located in the preceding
levels k', $k' \leq k - 1$, with at least one predecessor located
at level $k - 1$. Thanks to the level reorganization there are
no edges between two vertices at the same level. For each
level it is then possible to merge all vertices labeled as p and
separately all vertices labeled as $\bar{p}$. We obtain a reduced graph
with the same number of levels but with one or two vertices
per level. If a level contains two vertices they are mandatory
labeled as p and $\bar{p}$.

The fusion process can be extended to the vertices located
at two consecutive levels as follows: for all levels k and $k+1$

- merge two vertices labeled as $\bar{p}$, this gives a new $\bar{p}$ vertex;
- merge two vertices labeled as p which are not connected
  by a *critical* edge $(d_l = 0)$, this gives a new p vertex.

The fusion process between different levels may modify the
vertex level repartition. However the modification can affect
only some levels: it does not impact the levels which precede
k. Algorithm 1 allows to manage the fusion of the vertices
with the level technique which maintains the highest degree
of parallelization. The final reduced graph is treated by YAO
which generates code according to the following steps.

- Sort in a topological order the final reduced graph and
  enumerate all its vertices following this topological order.
- Write one block of *order* directives for each vertex. These
  order directives have the same axes as the one provided

Algorithm 1 Fusion with levels approach.

1: Organize the graph $G_{l/SCC}$ in M_{level} levels. The vertices are labeled either p or $\bar{p}$.
2: Traverse the graph and for each level merge the vertices of the same label. Update edges and their labels (0, $-$ or $+$).
3: $k := 1$
4: **while** $k < M_{level}$ **do**
5:     Consider two consecutive levels k and $k + 1$:
6:     **if** there are two vertices labeled by p and there is no *critical* edge between them **then**
7:         Fusion the two in one vertex labeled by p
8:     **else**
9:         **if** there are two vertices labeled by $\bar{p}$ **then**
10:             Fusion the two in one vertex labeled by $\bar{p}$
11:         **end if**
12:     **end if**
13:     **if** a fusion has been performed **then**
14:         Reorganize the new reduced graph in levels and update M_{level}
15:     **else**
16:         $k := k + 1$
17:     **end if**
18: **end while**

Fig. 7. Subdomain decomposition with $d_{l_{max}} = 1$ for threads T_1, T_2 and T_3. Each thread domain is decomposed into two *border* subdomains with $d_{l_{max}}$ grid points in the parallel dimension, and one *inner* subdomain.

by the user and contain the basic functions merged in the vertex.

C. Parallelization of the backward procedure

The same algorithm can also be applied to the *backward* procedure, which results in a complete parallelization of all computations at each time step. The total elapsed time in a YAO application is mainly composed by the *forward* and the *backward* elapsed times. Making parallel these two procedures means that most of the application has been optimized. Profiling measurement on some YAO applications showed that roughly 99 percent of the total elapsed time is, in general, in these procedures.

The RDG used for the *backward* procedure is the same as for the *forward* procedure but the arrows are reversed with respect to the original RDG. These two RDGs have the same SCCs. As the outermost loops also have the same axis, the same method used to parallelize the *forward* procedure is also valid to parallelize the *backward* procedure. Likewise, it is easy to see that the rules used to merge loop blocks previously introduced remain valid for the *backward* procedure. Thus, parallel *order* directives obtained by the decomposition/merging methods defined for the *forward* procedure can be fully retained for the *backward* procedure.

However the parallelization of the *backward* procedure has a further difficulty in terms of synchronization. This synchronization is enforced by the addition (accumulation) presented in II-B2. As shown in Fig. 4, in a parallel context this addition may result in a data race condition (write/write conflicts) if the back propagations of dx_p are performed concurrently by several threads. This kind of synchronization may arise between two time steps. Hence, the analysis of the $\overline{\text{RDG}}$ is not sufficient to point out all the data race conditions of the *backward* procedure.
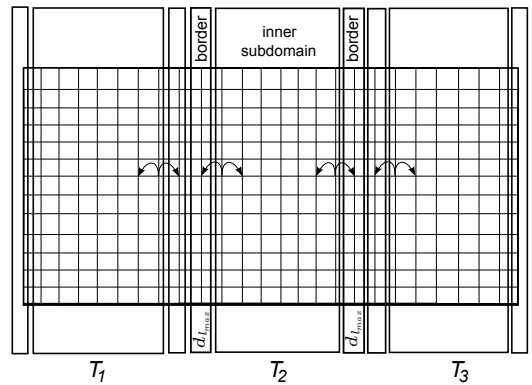
This can be solved with OpenMP *atomic* directives which ensure that each addition is performed atomically. However these atomic instructions are costly, as well as numerous in the *backward* parallel code, which prevents us from obtaining good parallel performances in practice. In order to avoid these OpenMP *atomic* directives, we rely on the distance vectors of the RDG to determine the maximum $|d_l|$, denoted $d_{l_{max}}$. In the 1D block decomposition, we can now further decompose each thread domain in three subdomains: two *border* subdomains with $d_{l_{max}}$ grid points in the parallel dimension, and one *inner* subdomain with usually much more than $d_{l_{max}}$ grid points in the parallel dimension. An example with $d_{l_{max}} = 1$ is presented in Fig. 7.

Data race conditions are now avoided by ensuring that all threads compute the three subdomains in the same ordering. OpenMP barrier directives are required between each subdomain computation. If there are too many threads, or if the computational space is too small, the thread domains may be too small: in these special cases the parallel code generated by YAO will have to rely on OpenMP *atomic* directives to run correctly in parallel without any data race condition.

Taking all this into account, the overall parallelization algorithm ensures the parallelization of all the computations done by a YAO generated application. It gives a domain decomposition with respect to the outermost loop l, which can then be automatically parallelized in the final generated code thanks to OpenMP directives. Furthermore if a multi-level parallelization is desired, it is then possible to apply the same algorithm for each subloop. We emphasize that the parallel code generated by YAO respects the *order* and the *ctin* directives which implies that the result of the parallel code is the same as the sequential code.

D. Marine acoustics example

This section presents an example of the decomposition algorithm on a 2D modular graph taken from one of the actual YAO applications. The Marine acoustics example has a small number of functions F_i, which allows us to easily show the evolution of the RDG graph. We use the same function names
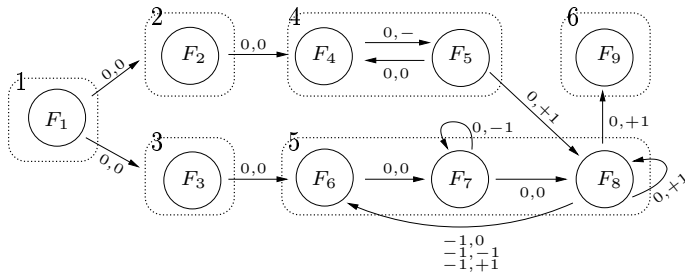
Fig. 8.   RDG: the dashed lines are the Strongly Connected Components.

order YA1
  order YA2
    $F_1\ F_3\ F_2\ F_5\ F_4\ F_6\ F_7$
  order YB2
    $F_8\ F_9$

Fig. 9.   *order* directives defined by the user for the Marine acoustics example.

order YA1 (**parallel**)
  order YA2
    $F_1\ F_3\ F_2\ F_5\ F_4$

order YA1 (**non parallel**)
  order YA2
    $F_6\ F_7$
  order YB2
    F_8

order YA1 (**parallel**)
  order YA2
    F_9

Fig. 10.   *order* directives recomputed by the algorithm for the Marine acoustics example.

Fig. 11. $G_{l/SCC}$ where the double circle represents a non parallelizable vertex.

Fig. 12.   Fusion of the vertices 2 and 3 in a new p vertex called 2,3.

Fig. 13.   Fusion of the vertices 1,2,3 and 4 in a new p vertex called 1,2,3,4.

as [6], [5]. This YAO application deals with marine acoustics and allows to assimilate some actual observations of acoustic pressure in order to retrieve some geoacoustic parameters like celerity, density and attenuation. In [6] the basic functions are denoted by $n(z)$, C, B, bet, gam, R, X_t, ψ and ψ_{fd}. To make it simpler we denote them respectively by $F_1, \ldots, F_9$. Fig. 8 shows the *RDG* composed by *r=9* basic functions and the edges labeled with the coefficient signs of the *ctin* directives. In this figure the SCCs are outlined by the dashed lines and numbered from 1 to 6. The *order* directives specified by the user are given in Fig. 9. In this case, the outermost loop is related to the ascendant i axis. After computing the $G_{l/SCC}$ graph, we label each vertex and we proceed with the level reorganization, as presented in Fig. 11, where M_{level} equals 5; the single circle is a parallelizable vertex (p) and the double circle is a non parallelizable vertex ($\bar{p}$). Fig. 12 shows the fusion of the vertices 2 and 3 labeled by p in a new vertex called 2,3 of the same label. This is done in the initialization phase of the algorithm (step 2). Then the vertices 1 and 2,3 can be merged in a new vertex called 1,2,3 which is parallel too. The two vertices are located on levels $k = 1$ and $k = 2$. A level reorganization reduces M_{level} to 4. The same operation is done on the modules 1,2,3 and 4, followed again by a level reorganization (M_{level} reduced to 3). The topological order is then: [1,2,3,4], [5], [6] as shown in Fig. 13. The algorithm ends because 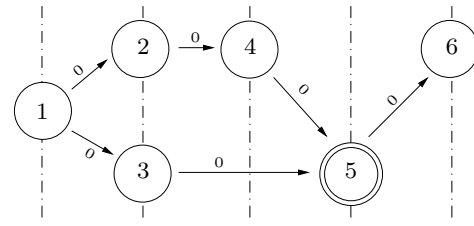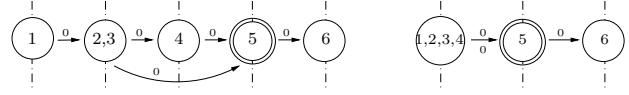it is no longer able to fusion and the level counter has reached M_{level} equals 3. This topological order is translated in an ordering of the modules. The final scheduling respects the ordering given by the user and corresponds to: $[F_1\ F_3\ F_2\ F_5\ F_4]$, $[F_6\ F_7\ F_8]$, $[F_9]$ or [n(z) B C gam bet], [R X_t ψ], [ψ_{fd}]. The final *order* directive decomposition is given in Fig. 10. With the keywords *parallel* and *non parallel* the figure outlines the outermost loops (*order* directives) that the algorithm has recognised as parallel or not.

IV. PERFORMANCE RESULTS

We now present the performance results of the parallel code generated by YAO for two actual applications of data assimilation: the Shallow-water and the Marine acoustics applications mentioned before. These tests have been performed on a server, located at Polytech Paris-UPMC (Paris, France), composed of one AMD Magny-Cours Opteron 6168 processor and 16 GB of memory. This processor has 12 cores running at 1.9 GHz which have private L1/L2 (64KB/512KB) caches and share two 6MB L3 caches. All computations are performed in double precision.

The RDG of the Shallow-water application is composed by 6 SCC (each SCC contains one basic function), see [3] for more details. The parallelization algorithm returns that all SCCs are parallelizable. Fig. 14 shows the elapsed time and the parallel speedup for an increasing number of cores used (with one OpenMP thread per core) and for different computational space sizes, with both OpenMP *atomic* directives and our subdomain decomposition. The data race conditions in the *backward* procedure are more efficiently avoided with our subdomain decomposition which clearly offers better performance than the *atomic* directives. We emphasize that such OpenMP code automatically generated by YAO is equivalent to a (nontrivial) manual parallelization, and offers good speedups (up to 9.4 on 12 cores). Moreover for a fixed number of cores the speedup increases with the computational space size since this increases the computational granularity of each thread.

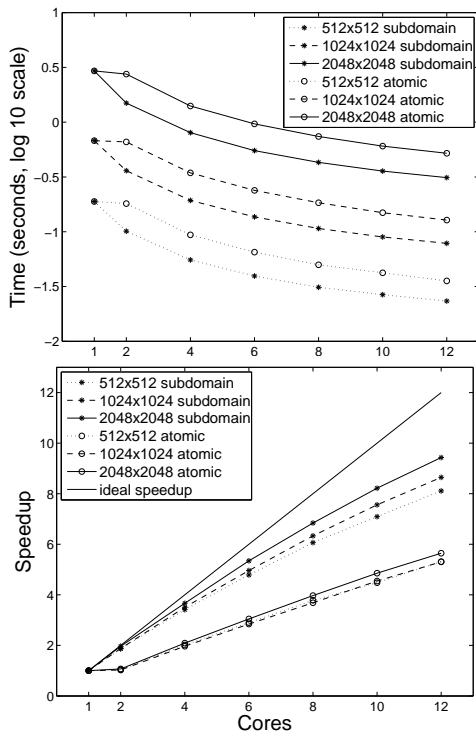The performance results on the Marine acoustics are very

Fig. 14.   Shallow-water performance measurements for three 2D computational space sizes over one time step (time averaged over 30 time steps). Both performances with OpenMP *atomic* directives and subdomain optimization are shown. The time encompasses both the *forward* and the *backward* procedures.
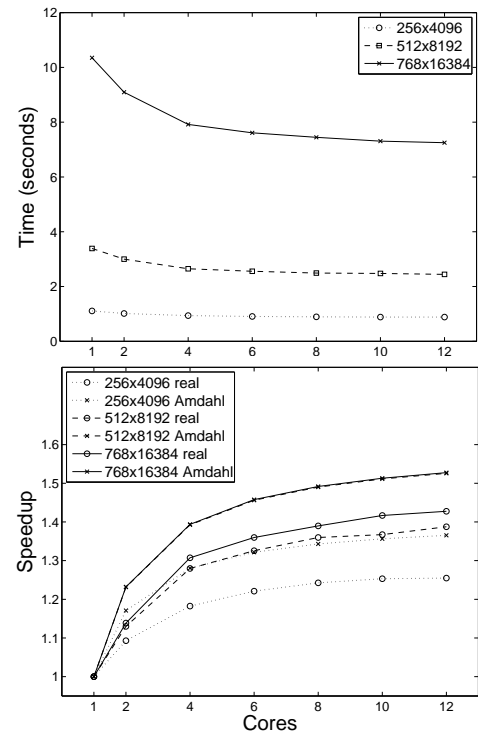

Fig. 15.   Marine acoustics performance measurements for three 2D computational space sizes over one time step (time averaged over 30 time steps). The time encompasses both the *forward* and the *backward* procedures.

different. In section III-D we have shown that the parallelization algorithm does not parallelize the whole GDR. Three modules, which unfortunately contain most of the computation, are excluded from the parallel region. Fig. 15 shows the elapsed time and the parallel speedup, as well as the theoretical maximum speedup according to Amdahl's law for this application. The parallel speedup is very limited, but the code generated by YAO offers most of the speedup available for this application. Again, the performance gain increases with the computational space size.

V. CONCLUSION AND PERSPECTIVES

In this paper, we have shown how the modular graph formalism of YAO allows addressing the issue of automatic parallelization of the code generated by YAO. Indeed, the YAO modular graph is generated by a reduced graph, which is similar to the Reduced Dependence Graph (RDG) used in automatic parallelization of nested loops. This similarity allows the adaptation to YAO of the algorithms that were developed in this research field. We have thus presented here how the Allen-Kennedy [12] and Kennedy-McKinley [11] algorithms can be integrated and adapted in order to enable the automatic parallelization, via multiple threads on shared memory architectures, of the application code generated by YAO. In the *backward* procedure the modular graph is furthermore used to decompose each thread domain into three subdomains, whose appropriate sizes enable us to completely avoid the race

conditions of this *backward* procedure. We have also presented the performance results of the parallel generated code with OpenMP on a multicore CPU for two actual applications: the Shallow-water and the Marine acoustics. We obtain good speedups (for example up to 9.4 on 12 cores) within the limits of the parallelism available in the application.

More advanced transformations (unimodular transformation, loop inversion, SIMD vectorization, tiling, ...) have already been developed in automatic loop parallelization, especially via the *polyhedral model* [10], [15]. We are currently studying if and how this polyhedral model can be integrated in the YAO framework. In the future, we also plan to investigate the automatic generation of MPI code from OpenMP code in the YAO context in order to automatically scale data assimilation applications on distributed memory architectures. It can be noticed that the subdomain decomposition between border and inner subdomains, presented here to avoid race conditions, may help overlap MPI communications with computation in order to obtain the best speedups in this distributed memory context: here again, the modular graph of YAO may be very useful to automatically determine this subdomain decomposition for any variational data assimilation application.

ACKNOWLEDGMENT

REFERENCES

[1] E. Kalnay, *Atmospheric Modeling, Data Assimilation, and Predictability*. Cambridge University Press, 2003.

[2] O. Talagrand, "Assimilation of Observations, an Introduction," *J. Meteor. Soc. Japan*, vol. 75, pp. 191–209, 1997.

[3] L. Nardi, C. Sorror, F. Badran, and S. Thiria, "YAO: A Software for Variational Data Assimilation Using Numerical Models." in *LNCS 5593, Computational Science and Its Applications - ICCSA 2009*, pp. 621–636.

[4] L. Nardi, "Formalisation et automatisation de YAO, générateur de code pour l'assimilation variationnelle de donnèes," PhD, CNAM, 2011.

[5] F. Badran, M. Berrada, J. Brajard, M. Crépon, C. Sorror, S. Thiria, J.-P. Hermand, M. Meyer, L. Perichon, and M. Asch, "Inversion of Satellite Ocean Colour Imagery and Geoacoustic Characterization of Seabed Properties: Variational Data Inversion Using a Semi-automatic Adjoint Approach," *J. of Marine Systems*, vol. 69, pp. 126–136, 2008.

[6] M. Berrada, "Une approche variationnelle de l'inversion, de la recherche locale à la recherche globale par carte topologique: application en inversion géoacoustique," PhD Thesis, UPMC, France, 2008.

[7] J. Brajard, C. Jamet, C. Moulin, and S. Thiria, "Use of a Neuro-variational Inversion for Retrieving Oceanic and Atmospheric Constituents from Satellite Ocean Colour Sensor: Application to Absorbing Aerosols," *Neural Networks*, vol. 19, no. 2, pp. 178–185, 2006.

[8] A. Kane, S. Thiria, and C. Moulin, "Développement d'une Méthode d'Assimilation de Données in Situ dans une Version 1D du Modèle de Biogochimie Marine PISCES," Master's thesis, LSCE/IPSL, CEA-CNRS-UVSQ, 2006.

[9] G. Madec, *NEMO ocean engine*. LOCEAN, Paris, France: Note du Pôle de modélisation de l'Institut Pierre-Simon Laplace No 27, 2008.

[10] A. Darte, Y. Robert, and F. Vivien, *Scheduling and automatic parallelization*, Birkhaüser, Ed., 2000.

[11] K. Kennedy and K. McKinley, "Typed fusion with applications to parallel and sequential code generation," Tech. Rep., 1993.

[12] R. Allen, D. Callahan, and K. Kennedy, "Automatic decomposition of scientific programs for parallel execution," in *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. NY, USA: ACM, 1987, pp. 63–76.

[13] H. Jin, M. A. Frumkin, and J. Yan, "Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes," in *ISHPC 2000*. Springer-Verlag, pp. 440–456.

[14] J. Taillard, F. Guyomarc'h, and J.-L. Dekeyser, "A Graphical Framework for High Performance Computing Using An MDE Approach," in *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2008*. USA: IEEE CS, pp. 165–173.

[15] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI 2008*. USA: ACM SIGPLAN, pp. 101–113.