Exploiting Sparsity in Automatic Differentiation on Multicore Architectures

Benjamin Letschert, Kshitij Kulshreshtha, Andrea Walther,
Duc Nguyen, Assefaw Gebremedhin, and Alex Pothen

Abstract We discuss the design, implementation and performance of algorithms suitable for the efficient computation of *sparse* Jacobian and Hessian matrices using Automatic Differentiation via operator overloading on multicore architectures. The procedure for exploiting sparsity (for runtime and memory efficiency) in serial computation involves a number of steps. Using nonlinear optimization problems as test cases, we show that the algorithms involved in the various steps can be adapted to multithreaded computations.

Key words: sparsity, graph coloring, multicore computing, ADOL-C, ColPack

1 Introduction

Research and development around Automatic Differentiation (AD) over the last several decades has enabled much progress in algorithms and software tools, but it has largely focused on differentiating functions implemented as serial codes. With the increasing ubiquity of parallel computing platforms, especially desktop multicore machines, there is a greater need than ever before for developing AD capabilities for parallel codes. The subject of this work is on AD capabilities for multithreaded functions, and the focus is on techniques for exploiting the *sparsity* available in large-scale Jacobian and Hessian matrices.

Derivative calculation via AD for parallel codes has been considered in several previous studies, but the focus has largely been on the source transformation ap-

Benjamin Letschert, Kshitij Kulshreshtha, Andrea Walther
Institut für Mathematik, Universität Paderborn, Germany, Benny.Letschert@web.de, kshitij@math.upb.de,andrea.walther@uni-paderborn.de

Duc Nguyen, Assefaw Gebremedhin, Alex Pothen
Department of Computer Science, Purdue University, IN, USA, {nguyend|agebreme|apothen}@purdue.edu

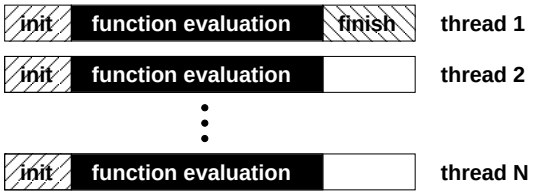

Fig. 1 Function evaluation of an OpenMP parallel code.

proach [1, 2, 3, 4, 11]. This is mainly because having a compiler at hand during the source transformation makes it relatively easy to detect parallelization function calls (as in MPI) or parallelization directives (as in OpenMP). Detecting parallel sections of code for an operator overloading tool is much harder since the corresponding parallelization function calls or directives are difficult or even impossible to detect at runtime. For that reason, the operator overloading tool ADOL-C [13] uses its own wrapper functions for handling functions that are parallelized with MPI. For parallel function evaluations using OpenMP, ADOL-C uses the concept of nested taping [8, 9] to take advantage of the parallelization provided by the simulation for the derivative calculation as well. In this paper we extend this approach to exploit sparsity in parallel.

By exploiting sparsity is meant avoiding computing with zeros in order to reduce (often drastically) runtime and memory costs. We aim at exploiting sparsity in *both* Jacobian and Hessian computations. In the serial setting, there exists an established scheme for efficient computation of sparse Jacobians and Hessians. The scheme involves four major steps: automatic sparsity pattern detection, seed matrix determination via graph coloring, compressed-matrix computation, and recovery. We extend this scheme to the case of multithreaded computations, where both the function evaluation and the derivative computation are done in parallel. The AD-specific algorithms we use are implemented in ADOL-C. The coloring and recovery algorithms are independently developed and implemented via ColPack [6], which in turn is coupled with ADOL-C. We show the performance of the various algorithms on a multicore machine using PDE-constrained optimization problems as test cases.

2 Parallel derivative computation in ADOL-C

Throughout this paper we assume that the user provides an OpenMP parallel program as sketched in Fig. 1. That is, after an initialization phase, calculations are performed on several threads, with a possible finalization phase performed by a dedicated single thread (say thread 1). The current "mode" of operation of ADOL-C when differentiating such OpenMP parallel codes is illustrated in Fig. 2. Here, the *tracing* part represents essentially the parallel function evaluation provided by the user. For computing the derivatives also in parallel, the user has to change in the

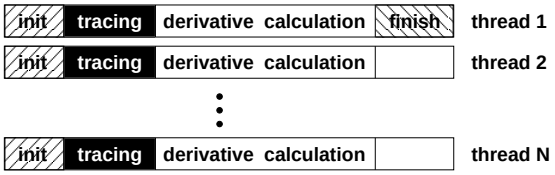

Fig. 2 Derivative calculation with ADOL-C for an OpenMP parallel code.

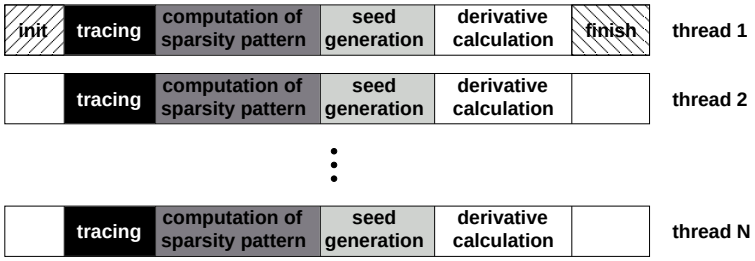

Fig. 3 Derivative calculation with ADOL-C for an OpenMP parallel code when exploiting sparsity.

function evaluation all double-variables to adouble-variables, include the headers adolc.h and adolc_openmp.h, and insert the pragma omp parallel firstprivate(ADOLC_OpenMP_Handler) before the trace generation in the initialization phase. Then, ADOL-C performs a parallel derivative calculation using the OpenMP strategy provided by the user as sketched in Fig. 2. Hence, once the variables are declared in each thread, the traces are written on each thread separately during the tracing phase. Subsequently, each thread has its own internal function representation. This allows for the computation of the required derivative information on each thread *separately* as described in [8].

3 Parallel sparse derivative computation

In this work, we extend this functionality of ADOL-C such that *sparse* Jacobians and Hessians can be computed efficiently in a parallel setting. Figure 3 illustrates the approach we take for parallel, sparsity-exploiting derivative computation. As in Fig. 2 derivatives on each thread are computed separately, but this time, the per-thread computation is comprised of several steps: automatic sparsity pattern detection, seed matrix generation and derivative calculation.

3.1 Sparsity pattern detection

In the case of a Jacobian matrix, we propagate in parallel on each thread the so-called index domains

$$\mathscr{X}_k \equiv \{j \leq n : j - n \prec^* k\} \quad \text{for} \quad 1 - n \leq k \leq l$$

determining the sparsity pattern corresponding to the part of the function on that thread. Here, n denotes the number of independent variables, l denotes the number of intermediate variables, and $\prec^*$ denotes precedence relation in the decomposition of function evaluation into elementary components. Since it is not possible to exchange data between the various threads when using OpenMP for parallelization, the layout of the data structure storing these partial sparsity patterns has to allow a possibly required reunion of the sparsity pattern, for example during the finalization phase performed by thread 1. However, since the user provides the parallelization strategy, this reunion can not be provided in a general way.

To determine the sparsity pattern of the Hessian of a function $y = f(x)$ of n independent variables, in addition to the index domains, so-called nonlinear interaction domains

$$\left\{ j \leq n : \frac{\partial^2 y}{\partial x_i \partial x_j} \not\equiv 0 \right\} \subseteq \mathscr{N}_i \,, \quad \text{for} \quad 1 \leq i \leq n$$

are propagated on each thread. Once more, each thread computes only the part of the sparsity pattern originating from the internal function representation available on the specific thread. Therefore, in the Hessian case also, the data structure storing the partial sparsity patterns of the Hessian must allow a possibly required reunion to compute the overall sparsity pattern. Again, this reunion relies on the parallelization strategy chosen by the user.

3.2 Seed matrix determination

A key need in compression-based computation of an $m \times n$ Jacobian or an $n \times n$ Hessian matrix A of known sparsity pattern is determining an $n \times p$ *seed* matrix S of minimal p that would be used in computing the compressed representation $B \equiv AS$. The seed matrix S in our context encodes a *partitioning* of the n columns of A into p groups. It is a zero-one matrix, where entry (j,k) is one if the jth column of the matrix A belongs to group k in the partitioning and zero otherwise. The columns in each group are pair-wise structurally "independent" in some sense. For example, in the case of a Jacobian, the columns in a group are structurally orthogonal to each other. As has been shown in several previous studies (see [5] for a survey), a seed matrix can be obtained using a *coloring* of an appropriate graph representation of

the sparsity pattern of the matrix A. In this work we rely on the coloring models and functionalities available in (or derived from) the package ColPack [6].

In ColPack, a Jacobian (nonsymmetric) matrix is represented using a *bipartite graph* and a Hessian (symmetric) matrix is represented using an *adjacency graph*. With such representations in place, we obtain a seed matrix suitable for computing a Jacobian J using a *distance-2 coloring* of the column vertices of the bipartite graph of J. Similarly, we obtain a seed matrix suitable for computing a Hessian H using a *star coloring* of the adjacency graph of H [7]. These colorings yield seed matrices suitable for direct recovery, as opposed to recovery via substitution, of entries of the original matrix A from the compressed representation B.

Just as the sparsity pattern detection was done on each thread focusing on the part of the function evaluation on that thread, the colorings are also done on the "local" graphs corresponding to each thread. For the results reported in this paper, we use parallelized versions of the distance-2 and star coloring functionalities of ColPack.

3.3 Derivative calculation

Once a seed matrix per thread is determined, the compressed derivative matrix (Jacobian or Hessian) is obtained using an appropriate mode of AD. The entries of the original derivative matrix are then recovered from the compressed representation. For recovery purposes, we rely on ColPack. In Fig. 3 the block "derivative calculation" lumps together the compressed derivative matrix computation and recovery steps.

4 Experimental results

We discuss the test cases used in our experiments in Sect. 4.1 and present the results obtained in Sect. 4.2.

4.1 Test cases

We consider optimization problems of the form

$$\min_{x \in \mathbb{R}^n} f(x), \quad \text{such that} \quad c(x) = 0, \tag{1}$$

with an objective function $f : \mathbb{R}^n \mapsto \mathbb{R}$ and a constraint function $c : \mathbb{R}^n \mapsto \mathbb{R}^m$, ignoring inequality constraints for simplicity. Many state-of-the-art optimizers, such as Ipopt [12], require at least first derivative information, i.e., the gradient $\nabla f(x) \in \mathbb{R}^n$ of the target function and the Jacobian $\nabla c(x) \in \mathbb{R}^{m \times n}$. Furthermore, they benefit con-

siderably in terms of performance from the provision of exact second order derivatives, i.e., the Hessian $\nabla^2\mathscr{L}$ of the Lagrangian function

$$\mathscr{L} : \mathbb{R}^{n+m} \mapsto \mathbb{R}, \quad \mathscr{L}(x,\lambda) = f(x) + \lambda^T c(x).$$

Optimization tasks where the equality constraints represent a state description as discretization of a partial differential equation (PDE) form an important class of optimization problems having the structure shown in (1). Here, sparsity in the derivative matrices occurs inherently and the structure of the sparsity pattern is not obvious when a nontrivial discretization strategy is used.

In [10] several scalable test cases for optimization tasks with constraints given as PDEs are introduced. The state in these test cases is always described by an elliptic PDE, but there are different ways in which the state can be modified, i.e., controlled. For four of the test problems, serial implementations in C++ are provided in the example directory of the Ipopt package. From those, we chose the `MittelmannDistCntrlDiri` and the `MittelmannDistCntrlNeumA` test cases for our experiments. These represent optimization tasks for a distributed control with different boundary conditions for the underlying elliptic PDE. Inspecting the implementation of these test problems, one finds that the evaluation of the constraints does not exploit the computation of common subexpressions. Therefore, when taking the structure of the optimization problem (1) into account, a straightforward parallelization based on OpenMP distributes the single target function and the evaluation of the m constraints equally on the available threads. The numerical results presented in Sect. 4.2 rely on this parallelization strategy.

Problem sizes. The results we obtained for the `MittelmannDistCntrlDiri` and `MittelmannDistCntrlNeumA` showed similar general trends. Therefore, we present results here only for the former. We consider three problem sizes $\tilde{n} \in \{600, 800, 1\,000\}$, where $\tilde{n}$ denotes the number of inner grid nodes per dimension. The number of constraints (number of rows in the Jacobian ∇c) is thus $m = \tilde{n}^2$. Due to the distributed control on the inner grid nodes and the Dirichlet conditions at the boundary nodes, the number of variables in the corresponding target function (number of columns in the Jacobian ∇c) is $n = \tilde{n}^2 + (\tilde{n}+2)^2$. Further, the Hessian $\nabla^2\mathscr{L}$ of the Lagrangian function is of dimension $(n+m) \times (n+m)$. The number of nonzeros in each $m \times n$ Jacobian is $6 \cdot \tilde{n}^2$. Here, five of the nonzero entries per row stem from the discretization of the Laplacian operator occurring in the elliptic PDE, and the sixth entry comes from the distributed control. Similarly, the number of nonzeros in each $(n+m) \times (n+m)$ Hessian is $8 \cdot \tilde{n}^2$. The two additional nonzeros in the Hessian case come from the target function involving a sum of squares and a regularization of the control in the inner computational domain. Table 1 provides a summary of the sizes of the three test problems considered in the experiments.

$\tilde{n}$	m	n	$n+m$	nnz (∇c)	nnz $(\nabla^2 \mathscr{L})$
600	360 000	722 404	1 082 404	2 160 000	2 880 000
800	640 000	1 283 204	1 923 204	3 840 000	5 120 000
1 000	1 000 000	2 004 004	3 004 004	6 000 000	8 000 000

Table 1 Summary of problem sizes used in the experiments.

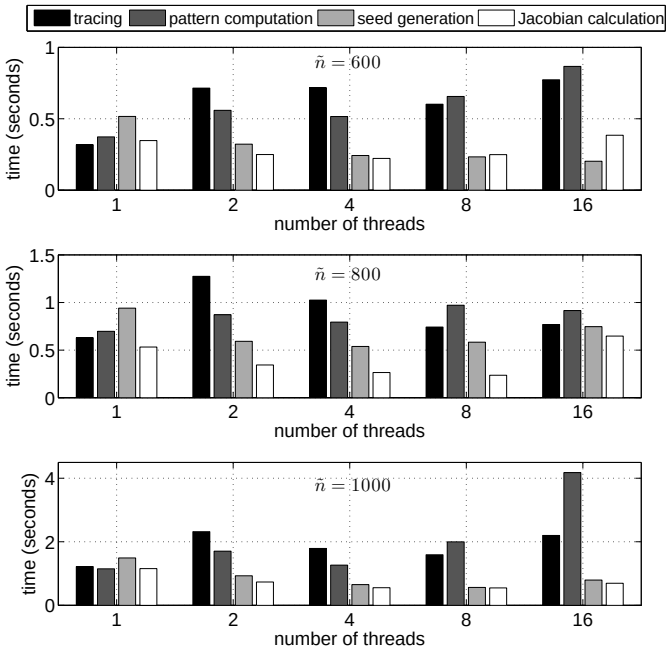

Fig. 4 Timing results for multithreaded computation of the Jacobian ∇c when sparsity is exploited. Three problem sizes are considered: $\tilde{n} = 600$ (top), $\tilde{n} = 800$ (middle), and $\tilde{n} = 1000$ (bottom).

4.2 Runtime results

The experiments are conducted on an Intel, Fujitso-Siemens, model RX600S5 system. The system has four Intel X7542, 2.67GHz, processors each of which has six cores; the system thus supports the use of a maximum of 24 cores (threads). The node memory is 128 GByte DDR3 1066, and the operating system is Linux (CentOS). All codes are compiled with gcc version 4.4.5 with -O2 optimization enabled.

Figure 4 shows runtime results on the computation of the Jacobian of the constraint function for the three problem sizes summarized in Table 1 and various number of threads. Figure 5 shows analogous results for the computation of the Hessian of the Lagrangian function. The plots in Fig. 4 (and Fig. 5) show a breakdown of the total time for the sparse Jacobian (and Hessian) computation into four constituent

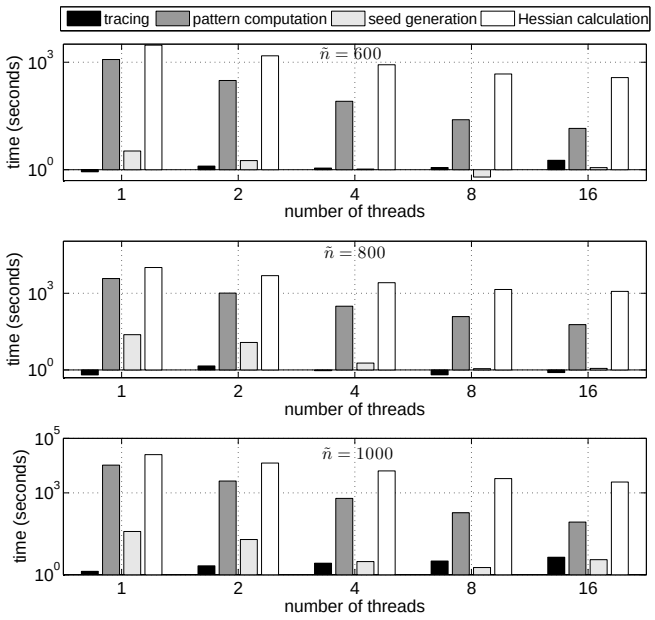

Fig. 5 Timing results for multithreaded computation of the Hessian $\nabla^2 \mathscr{L}$ when sparsity is exploited. Three problem sizes considered: $\tilde{n} = 600$ (top), $\tilde{n} = 800$ (middle), and $\tilde{n} = 1000$ (bottom).

parts: tracing, sparsity pattern detection, seed generation, and derivative computation. The results in both figures show the times needed for the "distributed" (across threads) Jacobian and Hessian computation, excluding the time needed to "assemble" the results. We excluded the assembly times as they are nearly negligibly small and would have obscured the trends depicted in the figure. (The assembly time is less than 0.03 sec for $\tilde{n} = 600$ and less than 0.09 sec for $\tilde{n} = 1\,000$ for the Jacobian case, and less than 0.17 sec for the Hessian case for both sizes.)

Note that the vertical axis in Fig. 4 is in linear scale, while the same axis in Fig. 5 is in log scale, since the relative difference in the time spent in the four phases in the Hessian case is too big. Note also the magnitude of the difference between the runtimes in the Jacobian and Hessian cases: the runtimes in the various phases of the Jacobian computation (Fig. 4) are in the order of seconds, while the times in some of the phases in the Hessian case (Fig. 5) are in the order of thousands of seconds. We highlight below a few observations on the trends seen in Fig. 4 and Fig. 5.

- *Tracing:* In the Jacobian case, this phase scales poorly with number of threads. A likely reason for this phenomenon is that the phase is memory-intensive. In the Hessian case, tracing accounts for only a small fraction of the overall time that its scalability becomes less important.

- *Sparsity pattern detection:* The routine we implemented for this phase involves many invocations of the `malloc()` function, which essentially is serialized in an OpenMP threaded computation. To better reflect the algorithmic nature of the routine, in the plots we report results after subtracting the time spent on the `malloc`s. In the Jacobian case, the phase did not scale with number of threads, whereas in the Hessian case it scales fairly well. A plausible reason for the poorer scalability in the Jacobian case is again that the runtime for that step (which is about one second) is too short to be impacted by the use of more threads.
- *Seed generation:* For this phase, we depict the time spent on coloring (but not graph construction) and seed matrix construction. It can be seen that this phase scales relatively well. Further, the number of colors used by the coloring heuristics turned out to be optimal (or nearly optimal). In particular, in the Jacobian case, for each problem size, 7 colors were used to distance-2 color the local bipartite graphs consisting of n column vertices and m/N row vertices on each thread, where N denotes the number of threads. Since each Jacobian has six nonzeros per row this coloring is optimal. In the Hessian case, again for each problem size, 6 colors were used to star color the local adjacency graphs (consisting of $n+m$ vertices) on each thread.
- *Derivative computation:* This phase scales modestly in both the Jacobian and Hessian cases.
- *Comparison with dense computation:* The relatively short runtime of the coloring algorithms along with the drastic dimension reduction (compression) the colorings provide enables enormous overall runtime and space saving compared to a computation that does not exploit sparsity. The runtimes for the dense computation of the Jacobian for $\tilde{n} = 600$, for example, are at least three to four orders of magnitude slower requiring hours instead of seconds even in parallel (we therefore omitted the results in the reported plots). For the larger problem sizes, the Jacobian (or Hessian) could not be computed at all due to excessive memory requirement to accommodate the matrix dimensions (see Table 1).

5 Conclusion

We demonstrated the feasibility of exploiting sparsity in Jacobian and Hessian computation using Automatic Differentiation via operator overloading on multithreaded parallel computing platforms. We showed experimental results on a modest number of threads. Some of the phases in the sparse computation framework scaled reasonably well, while others scaled poorly. In future work, we will explore ways in which scalability can be improved. In particular, more investigation is needed to improve the scalability of the sparsity pattern detection algorithm used for Jacobian computation (Fig. 4) and the tracing phase in both the Jacobian and Hessian case. Another direction for future work is the development of a parallel optimizer that could take advantage of the distributed function and derivative evaluation.

Acknowledgements The experiments were performed on a computing facility hosted by the Paderborn Center for Parallel Computing (PC^2). The research is supported in part by the U.S. Department of Energy through the CSCAPES Institute grant DE-FC02-08ER25864 and by the U.S. National Science Foundation through grant CCF-0830645.

References

1. Bücker, H.M., Rasch, A., Vehreschild, A.: Automatic generation of parallel code for Hessian computations. In: M.S. Mueller, B.M. Chapman, B.R. de Supinski, A.D. Malony, M. Voss (eds.) OpenMP Shared Memory Parallel Programming, Proceedings of the International Workshops IWOMP 2005 and IWOMP 2006, Eugene, OR, USA, June 1–4, 2005, and Reims, France, June 12–15, 2006, *Lecture Notes in Computer Science*, vol. 4315, pp. 372–381. Springer, Berlin / Heidelberg (2008). DOI 10.1007/978-3-540-68555-5_30
2. Bücker, H.M., Rasch, A., Wolf, A.: A class of OpenMP applications involving nested parallelism. In: Proceedings of the 19th ACM Symposium on Applied Computing, Nicosia, Cyprus, March 14–17, 2004, vol. 1, pp. 220–224. ACM Press, New York (2004). DOI 10.1145/967900.967948. URL `http://doi.acm.org/10.1145/967900.967948`
3. Conforti, D., Luca, L.D., Grandinetti, L., Musmanno, R.: A parallel implementation of automatic differentiation for partially separable functions using PVM. Parallel Computing **22**, 643–656 (1996)
4. Fischer, H.: Automatic Differentiation: Parallel computation of function, gradient and Hessian matrix. Parallel Computing **13**, 101–110 (1990)
5. Gebremedhin, A.H., Manne, F., Pothen, A.: What color is your Jacobian? Graph coloring for computing derivatives. SIAM Review **47**(4), 629–705 (2005)
6. Gebremedhin, A.H., Nguyen, D., Patwary, M., Pothen, A.: ColPack: Software for graph coloring and related problems in scientific computing. Tech. rep., Purdue University (2011)
7. Gebremedhin, A.H., Tarafdar, A., Manne, F., Pothen, A.: New acyclic and star coloring algorithms with applications to Hessian computation. SIAM J. Sci. Comput. **29**, 1042–1072 (2007)
8. Kowarz, A.: Advanced concepts for Automatic Differentiation based on operator overloading (1998). PhD Thesis, TU Dresden
9. Kowarz, A., Walther, A.: Parallel derivative computation using ADOL-C. In: W. Nagel, R. Hoffmann, A. Koch (eds.) Proceedings of PASA 2008, Lecture Notes in Informatics, Vol. 124, pp. 83 – 92. Gesellschaft fr Informatik (2008)
10. Maurer, H., Mittelmann, H.: Optimization techniques for solving elliptic control problems with control and state constraints. II: Distributed control. Comput. Optim. Appl. **18**(2), 141–160 (2001)
11. Utke, J., Hascoët, L., Heimbach, P., Hill, C., Hovland, P., Naumann, U.: Toward adjoinable MPI. In: Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering, PDSEC-09 (2009). DOI http://doi.ieeecomputersociety.org/10.1109/IPDPS.2009.5161165
12. Wächter, A., Biegler, L.: On the implementation of a Primal-Dual Interior Point Filter Line Search algorithm for large-scale nonlinear programming. Math. Program. **106 (1)**, 25–57 (2006)
13. Walther, A., Griewank, A.: Getting started with ADOL-C. In: U. Naumann, O. Schenk (eds.) Combinatorial Scientific Computing. Chapman-Hall (2012).
see also http://www.coin-or.org/projects/ADOL-C.xml