Cougar: An Interactive Tool for Cluster Computing

Barbara Chapman, Tien-Hsiung Weng, Oscar Hernandez, Zhenying Liu, Lei Huang, Yi Wen, Laksono Adhianto
University of Houston, Houston, TX
chapman@cs.uh.edu

***Abstract*-- Cougar Compiler is a tool designed to help the programmer understand the structure of a sequential or parallel Fortran program. We support the de facto standards OpenMP and MPI, as well as the mixed mode OpenMP/MPI model, which can be used to write programs for executions on SMP clusters. The user may query the system interactively and view the results obtained by our program analysis via a graphical interface. This analysis includes up-to-date dependence tests, array section analysis and parallel dataflow analysis.**

In addition to representing a program's structure, Cougar is able to automatically generate OpenMP code and assist in its optimization as well as check for common parallel programming errors such as race conditions. We plan to make Cougar available to the community.

***Index Terms*-- Parallel Programming Tools, OpenMP, MPI, Fortran 77 support, Program Analysis.**

I. Introduction

IT is much harder to write parallel programs than sequential ones, because developers need to know details of the target computer architectures (which may include shared or distributed memory systems, DSMs, clustered SMPs, etc), and of the paradigm used to describe parallelism (MPI, HPF, OpenMP, Pthreads, etc) in addition to understanding their application; moreover, they must devise a strategy for exploiting parallelism in their problem solution. Parallel application developers almost always begin with a sequential program, and then incrementally transform it into a parallel program, optimizing it to improve parallelism as well as its efficiency on individual CPUs.

Cougar [7] is a software development tool designed to help parallel application programmers create optimized and scalable parallel applications for a variety of platforms including clustered SMPs. It accepts Fortran 77 source code with OpenMP and/or MPI constructs, performs interactive program analyses and source-to-source transformations to help create an alternative parallel version. It has browsing capabilities to help a programmer attain better understanding of how a given parallel program works, with a graphical user interface that displays the overall structure of the source code as well as detailed information for a given code segment. Some of the main features of Cougar's displays include the program's callgraph, callsite information, data dependence analysis, control flow analysis, global data structure usage and I/O information within a program. Its data dependence analysis is powerful, since Cougar implements a hierarchical dependence test including GCD, Banerjee, and the I-test. The information provided by Cougar gives an application developer an in-depth view of his/her source program.

Cougar provides a variety of features for manual parallelization efforts, particularly those that adopt the SPMD programming model. OpenMP/MPI mixed mode codes may potentially provide a more efficient parallelization strategy than pure MPI for an SMP cluster. Therefore, Cougar supports the hybrid MPI/OpenMP programming model. Help for inserting OpenMP constructs is offered, as are features for improving the use of OpenMP in a code. Since it is efficient to treat MPI and OpenMP [20] separately, we expect that a programmer will often integrate MPI and OpenMP programs that are designed and optimized separately, or add OpenMP to an existing MPI code. Since the thread-safety of the MPI implementation is then a concern, we help check for this and other potential programming problems.

In the past few years we have seen a widespread acceptance of OpenMP as a portable model for shared memory parallel programming. Unfortunately, experiments [16] using PGI 3.2-4, Hitachi, NEC, SGI 7.3.1.1, Omni 1.3, and Guide 4.0 compilers showed that many possible optimizations are missing from current OpenMP compilers. In addition to being a user-level tool with the interactive features described in this paper, Cougar serves as a testbed for us to learn more about compiler optimization for OpenMP codes.

Cougar is an on-going research project developed at the University of Houston, with funding from NSF, DOE, Dell Computer Corp. and NASA. Its functionality is currently being ported to the Open64 compiler infrastructure so that future versions will be able to accept Fortran90/95. When this process is complete, we intend to make Cougar available to the community and contribute to the set of tools currently available that supports cluster programming.

In the next section, we describe interactive features for displaying a program's structure and its use of data. Then we describe how Cougar supports the insertion and optimization

This work was supported in part by the U.S. Department of Energy under Grant W-7405-ENG-36 and the National Science Foundation under Grant ACI 96-19019.

of OpenMP in a sequential program or an MPI process. Issues specifically related to cluster computing follow in Section IV. We conclude by summarizing related work and our plans for future work.

II. STANDARD ANALYSIS CAPABILITIES

The standard analysis of Cougar helps the parallel programmer gain an in-depth understanding of how a sequential or parallel program is structured and how it behaves. It attempts to give initial support to the programmer who plans to manually parallelize or restructure an application.

A. The Callgraph And Callsite Information

The program structure is represented in the form of a call graph, which contains a node for each procedure in the program, and a directed edge linking a pair of nodes if and only if the procedure corresponding to the source node may invoke the sink node's procedure at run time. The user can select an alternative view of the program structure by selecting the call tree option as shown below in Figure 1. In this display form, a single procedure may correspond to multiple nodes, since it will be separately displayed with a link from each node representing a procedure that may call it at run time. An additional feature that is being currently implemented is the ability to collapse/expand hierarchically the nodes of the callgraph, in order to summarize regions of the call graph for large applications. The callgraph is built via a standard call graph algorithm, which creates a graph structure that can be easily traversed.

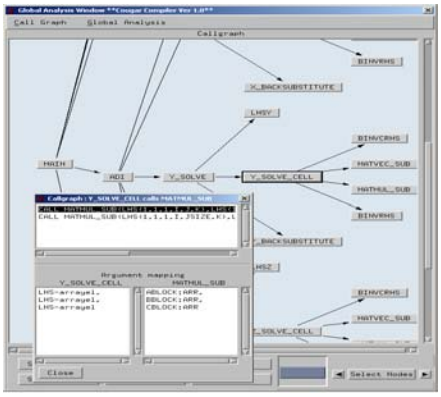

Figure 1. Callgraph and Callsite Information for the NAS BT Benchmark.

When a procedure is selected, either from a list of subroutines and functions, or by clicking on a corresponding node in the program structure display, the source code text of the procedure is displayed in a separate window.

The user may request information on the calls that one procedure makes to another one. The simplest way to obtain data on this is to click on the corresponding edge in the call graph. As a result, a list of the calls in the source code is displayed. If an individual call is selected from this list, the formal and actual arguments are displayed and can be compared.

Array section analysis is performed in order to calculate summary information on the possible side effects of calls on the program's arrays.

B. Flowgraph

Before attempting to parallelize a sequential program, it is important to visualize how the different blocks of statements are organized inside each procedure. In order to understand the structure of an individual subroutine or function, it may be necessary to obtain an overview of its control flow.

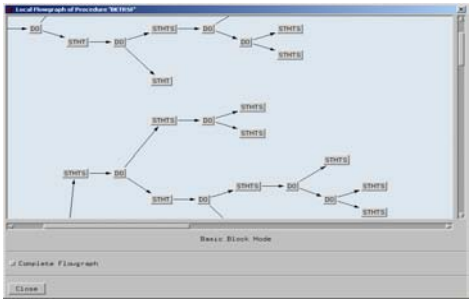

Figure 2. Control Flowgraph**.**

Cougar uses the results of intraprocedural flow analysis to summarize and display the control flow of a procedure. Nodes in the display represent basic blocks, or sequences of consecutive executable statements that do not contain any internal control flow. Basic blocks are identified and linked in the high-level intermediate representation of Cougar for use here and in data flow and data dependence analysis. A directed edge links a node to all possible successor nodes in the procedure, so that each possible execution path within the procedure is displayed. All procedure calls, loop headers, and conditional constructs in the selected routine are explicitly depicted in the graphical flowgraph.

Standard dataflow analysis can represent control flow in a sequential program or in an MPI process. However, we need parallel dataflow analysis [10][19] to represent and optimize a parallel code. Our tool represents the flow of control in an OpenMP program via a parallel flowgraph which takes account of intra-thread control flow and synchronization among threads. Such a graph has two kinds of nodes, conventional nodes and OpenMP directive nodes, and three kinds of edges: sequential edges, parallel edges, and synchronization edges. It can show users the parallel regions and synchronization points within a procedure. This parallel flow graph is also the basis for a variety of OpenMP optimizations. We are using it to accomplish reaching definition analysis, which considers intra-thread, inter-thread data flow and synchronization, in order to privatize variables. It is also used to perform synchronization analysis, cross-loop data dependence and array section analysis for removing unnecessary barriers, merging parallel regions and detecting race conditions, etc.

C. I/O Analysis

Some manual modifications require the programmer to deal

with the program's I/O explicitly. To support this, Cougar gathers details of I/O statements in a code; this I/O information can be sorted and displayed either by procedure units or by I/O device units. Since OpenMP does not support parallel I/O while MPI does, the details could be used to reorganize I/O in MPI programs or I/O accesses in general. Cougar takes advantage of its scanning/parsing capabilities to store this information in a simple database that can be accessed anytime within the program and can be sorted accordingly.

D. Global data usage analysis

Some Fortran programs make heavy use of global data, declared in common blocks. Such common blocks are often accessed in all procedures of a program. It may be necessary for the application developer to determine where the variables in the common block are actually referenced, to obtain an overview of the nature of these references, and to find out whether data in the common block is identically declared throughout the program. Cougar identifies where the common blocks are accessed within the different procedures. Results of this global data usage analysis can be displayed and used to detect inconsistencies in declarations between procedures.

E. Data Dependence

Data dependence information is essential in any effort to reorganize a sequential program to obtain a parallel counterpart. However, even though this concept is well understood by many application developers, it is notoriously difficult to detect the dependences that exist in even small regions of code. Cougar uses the separability test, gcd, Banerjee and the I-test in a hierarchical algorithm to detect all these dependencies efficiently and as accurately as possible.

The objective of the data dependence tests is to determine unequivocally if a pair of data accesses, either array elements or scalar variables, inside the range of a loop nest may refer to the same memory location. In particular, a programmer will want to identify loop carried dependences, and dependence cycles, which require extra effort during parallelization.

The Cougar implementation first ensures that all loops are in normal form. Then dependence analysis is performed separately for each program unit. The dependence analysis evaluates all scalar references. Only those array references that are suitable for the available dependence tests are analyzed. Cougar linearizes multi-dimensional arrays instead of testing each dimension separately to avoid loss of accuracy when evaluating each dimension independently.

The data dependence analyzer tests for true, anti and output dependence. In the case of subscript dependence test, Cougar uses a hierarchical test, where it first applies the separability test, gcd and then the I-Test. If the I-Test returns a case for "Banerjee test residue" [13], it invokes the Banerjee test to evaluate the subscript expression with the known limits. If the algorithm fails to determine whenever or not there is a dependence, it will assume there is one, which will be reflected in the dependence graph.

The I-test was selected as our main dependence test because it is highly efficient and accurate enough to provide dependence information on demand. The speed of the I-test is comparable with the Banerjee test but it is more accurate since it tries to find integer solutions within the intermediate-value theorem solution range. For the Perfect Benchmarks, the I-Test outperformed the Omega test in terms of speed by a factor of 55 on average [17]. When the accuracy of the Omega test and the I-test were compared using the Perfect benchmarks, Omega proved only 3% more dependences than the I-test. [17].

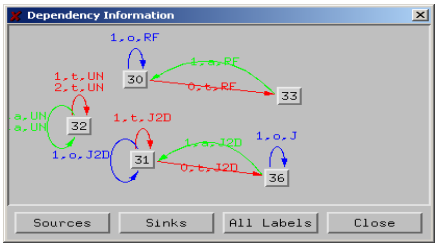

Figure 3. Data Dependence Graph

III. OPENMP SUPPORT

Cougar provides support to help an application programmer to create a shared memory parallel program manually or to transform from a sequential one. It then improves a parallel OpenMP program and performs a safety check of an OpenMP program. Features include identifying parallel loops and parallel regions, merging parallel regions, performing barrier elimination, identifying private variables and reduction variables, optimizing for memory locality reuse, and checking for the possibility of race conditions within a parallel DO construct.

A. OpenMP parallelization

Our tool first traverses the callgraph and flowgraph and identifies all potential parallel loops using the results of data dependence tests and array section analysis. Essentially, any loop with no loop-carried dependences is a good candidate for parallelization via the "parallel do" directive. In some cases we may increase the number of parallel loops by applying transformations, such as induction variable substitution, variable privatization, or constant propagation to eliminate dependences. As part of the optimization process described below. loop splitting is carried out to separate parallel from non-parallel parts.

Initially, each parallel region is assumed to consist of one parallel loop; adjacent parallel regions may subsequently be merged. An attempt is made to recognize reduction variables which have reduction operation such as "*", "+", "-", "min", "max", so that these do not prevent parallelization, and may be identified and dealt with via OpenMP's "reduction" clause.

B. Program optimization

We provide support for OpenMP program improvement

such as the identification of private variables, reduction variables, merging of parallel regions, barrier elimination, and memory locality reuse.

Privatization is one of the most important means to increase parallelism, eliminate false sharing and improve data locality. A variable is privatizable in a loop if there are no loop carried true dependences in this variable within the loop. Dataflow analysis enables us to determine which scalar variables can be replicated across loop iterations. For instance in Figure 4, the reference to temp is shared by different iterations, which also causes a race condition. This can be solved using scalar expansion or privatization. The analysis determines that `temp` is independent in all iterations; and can be simply privatized by adding a PRIVATE clause. Array privatization is similar to scalar privatization and has been studied extensively [14][22]. If elements of an array are defined before use in those loop iterations where they occur, then the array is a candidate for privatization in this loop; this is done by examining the subscripts of an array that do not contain any induction variables.

```
I1=0                        C$OMP PARALLEL DO  PRIVATE(Temp)
DO I=1,N                    DO I=1,N
   DO J=1,M                    DO J=1,M
      Temp = A(I, J)              Temp = A(I,J)
      A(I,J) = C(I,J)             A(I,J) = C(I,J)
      C(I,J) = Temp               C(I,J) = Temp
   ENDDO                       ENDDO
ENDDO                       ENDDO
```

Figure 4. Scalar privatization for data locality and race condition identification

Standard data dependence analysis has been used to detect loop parallelism [4][23], but it is not appropriate for evaluating variable references across loops, or a larger program region. For these we use array section analysis. Currently our tool relies on a standard, efficient triplet notation based on regular sections [11], rather than more precise methods such as simple sections [3], and region analysis [21]. It summarizes array regions defined and used within a loop nest, an OpenMP construct and a sequence of statements by performing union operations. Then, we compute data dependences between these constructs or between the construct and a sequence of statements by performing an intersection operation. This data dependence information can be used further for performing barrier elimination, merging parallel regions, array privatization, and determining whether a NOWAIT clause can be safely added to an OpenMP construct.

Merging two or more parallel regions into one big region reduces fork-join overhead. We first check the dependences between the two regions; when there is no dependence between the regions, merging is performed and can be optimized by eliminating the barrier. Otherwise, a barrier is needed to merge the two regions. Figure 5 shows two parallel regions close to each other with sequential code in between them. The sequential code is parallelized first by adding the MASTER construct. Since there is no dependence between the three regions, we merge them as shown in Figure 6 and add NOWAIT to eliminate the implicit barrier. If there are dependences between these constructs, we cannot remove the implicit barriers or must add an explicit BARRIER, since the MASTER construct does not have an implicit one.

Our optimization algorithm is an improvement of [1], which is currently based on intra-procedural analysis. The algorithm can be used for a variety of loops from *rectangular* to *non-rectangular* loops (in the latter, one loop variable may depend on those of enclosing loops) and from *single* loops to *semi-perfectly nested loops*. A semi-perfectly nested loop is an extension of a perfectly nested loop that may include assignment statements between loop headers and loop end statements.

```
C$OMP PARALLEL DO
      DO K = 2, NDATA-3
         Q(K,K-1)=H(K)
         Q(K,K)=2.0*(H(K)+H(K+1))
         Q(K,K+1)=H(K+1)
      ENDDO
C$OMP END PARALLEL DO
      Q(NDATA-2,NDATA-3)=H(NDATA-2)
      Q(NDATA-2,NDATA-2)=2.0*(H(NDATA-2)+H(NDATA-1)
$OMP PARALLEL DO
      DO K = 1, NDATA-1
         S(K)=(Y(K+1)-Y(K)) / H(K)
      ENDDO
$OMP END PARALLEL DO
```

Figure 5. OpenMP code after identifying the parallel loops and parallel regions

```
C$OMP PARALLEL
C$OMP DO
      DO K = 2, NDATA-3
         Q(K,K-1)=H(K)
         Q(K,K)=2.0*(H(K)+H(K+1))
         Q(K,K+1)=H(K+1)
      ENDDO
C$OMP END DO NOWAIT
C$OMP MASTER
      Q(NDATA-2,NDATA-3)=H(NDATA-2)
      Q(NDATA-2,NDATA-2)=2.0*(H(NDATA-2)+H(NDATA-1)
C$OMP MASTER
$OMP  DO
      DO K = 1, NDATA-1
         S(K)=(Y(K+1)-Y(K)) / H(K)
      ENDDO
$OMP END DO
$OMP END PARALLEL
```

Figure 6. Merging of parallel regions and barrier elimination

Our algorithm consists of three steps: In the first step, already described above, data references are privatized and reduction variables are identified to enable parallelization. The second step is to reorder the nested loops according to the potential cache lines accessed in every loop iteration (the loop with the highest level of reuse based upon both *temporal* and *spatial locality* is innermost and that with the least reuse is in outermost position) and to the loop's *parallelizability* (i.e.

parallel loops are in outermost position, and non-parallel loops are in innermost position). The latter goal has higher priority. This reordering aims to maximize both parallelism and memory locality reuse. The last step of the optimization algorithm is to tile the innermost loop (if legal) according to the *cache line size*, the *cache size*, *loop stride*, and array's dimension. The *tile size* computation is adopted from [1]. We are also investigating the combination of *loop tiling* and *array padding* in this step to reduce *false-sharing*. The latter happens when several processors are continually updating different references that reside in the same cache line. The array padding transformation can reduce the false sharing problem by aligning the column of an array to a cache line boundary.

IV. Cluster computing issues

A. Thread-Safety Analysis for MPI libraries

Thread-safety is a term used to describe a routine that can be called from multiple programming threads without unwanted interaction between the threads. By using thread-safe routines, the risk that one thread will interfere and modify data elements of another thread is eliminated by circumventing potential data race situations with coordinated access to shared data. There is a thread-safety problem associated with MPI calls when mixing MPI and OpenMP.

Although a large number of MPI implementations are thread-safe, this cannot be guaranteed. Thus Cougar helps a programmer analyze a hybrid MPI/OpenMP code to determine whether the code may lead to problems in this respect, and can help to design thread-safe MPI/OpenMP codes. The corresponding MPI calls located in OpenMP parallel regions can be identified and displayed. Furthermore, a programmer may be able to reorganize this code interactively, select and insert synchronization directives from a list of directives: CRITICAL, MASTER and SINGLE.

On the other hand, as mentioned in [8], environment variables are not propagated by mpirun, so the programmer will need to explicitly set the requested number of threads with OMP_NUM_THREADS. If omp_set_num_threads() subroutine is not called explicitly, such a call is inserted interactively in Cougar after the programmer specifies the total number of threads.

B. SPMD Style

We expect that future clusters will provide direct access to remote memories via single-sided communication mechanisms such as get and put operations. For such systems, it may become possible to provide OpenMP directly, as is currently the case for cache-coherent DSMs and clusters that have software DSM installed. For such platforms, OpenMP programs will need to exhibit good data locality if high performance is to be obtained. It is possible to obtain data locality as well as to minimize false sharing between threads on an SMP via privatization of data. A systematic use of privatization that separates array elements that are shared from those elements of the same arrays that are not shared requires extensive program modification. The so-called SPMD style of OpenMP programming is a realization of this approach and it has been shown to provide scalable performance that is superior to a straightforward parallelization of loops even for SMPs with a small number of processors [5][6]. Unfortunately, it is a complex task to transform a loop-parallel OpenMP code into an equivalent SPMD program. An onging effort aims to provide functionality within Cougar to translate OpenMP programs into SPMD mode code to enable the programmer to achieve high performance.

V. Related Work

Although a variety of tools exist to support MPI and/or OpenMP application development, most of those in use assume that a parallel program has been created, and their features aim to assist the user in its optimization. They typically instrument a parallel program and represent its behavior, possibly with additional information. However, the user must learn to interpret the reported information and know how to apply it to achieve an improved program version. Vampir, a vendor-independent tool, and Cray's Apprentice are two examples of commercial tools that graphically analyze the performance of MPI programs. Vampir is now being extended to analyze the performance of OpenMP programs in the Intone [2] project. Sun's FORGE performance tool represents a new kind of performance tool that is able to provide feedback on the execution behavior of OpenMP programs entirely based upon hardware counters and thus without instrumentation.

Support for the time-consuming task of parallel application creation based upon an analysis of the source code was pioneered in Parascope. Many of the features developed in that project are still among those most needed by application developers. Recent efforts are similarly based upon compiler infrastructures, and the features they provide range from passive display of program analysis results to sophisticated interactive restructuring operations. The SUIF explorer [15] is an example of an interactive parallelizer; it can run programs sequentially to collect loop profiling information and dynamic data dependency information. The Parallelization Guru of the Explorer is able to guide users through the parallelization process, but it does not automatically transform code.

Captools [12] is a computer-aided toolset that can help users transform a sequential Fortran 77 program into a parallel version with MPI, OpenMP (CAPO module) or a combination of both. This tool is widely used among researchers to generate parallel code; it employs its own library, CAPLib, to generate generic parallel code that can be mapped to SHMEM or MPI so that the fastest available implementation can be created for a given platform. The analysis performed for large applications is powerful, extensive and slow, sometimes taking several days, and the information provided may or may not be relevant to the user, according to his needs. Our tool attempts to provide good analysis comparatively rapidly and to be more strategic when presenting information to the user.

The WPP compiler comes with a tool, Aivi [18], that graphically displays analysis results of programs that have been processed by it. WPP realizes a basic loop parallelization by inserting OpenMP directives around parallel loops. Aivi helps the user detect larger parallel regions by enabling navigation of a program to find outer loops enclosing code even when they are in a different procedure. The user may then be able to parallelize outer loops manually.

Foresys [9] is a commercial software suite for reengineering Fortran applications developed by Simulog. It can be used to maintain multiple program versions and to modernize legacy codes, and provides a broad range of interactive displays and transformation features. Recent work with similar goals as ours has extended its functionality to the creation of OpenMP programs. However, the tool does not include automatic parallel code optimization or related analyses. They also do not user guidance for the task of transforming from sequential to parallel form. The non-commercial TSF module developed at INRIA provides novel features for defining and using custom program transformations within the Foresys environment.

VI. CONCLUSIONS AND FUTURE WORK

Cougar is an interactive tool that provides detailed information about a Fortran program that may contain OpenMP/MPI constructs. The basic information displayed in our graphical tool is general-purpose and could be employed in many situations, from analyzing legacy sequential Fortran 77 code to helping users reconstruct parallel code. Currently, our support for the creation of MPI code is limited to a display of the control flow; our features can be used in conjunction with a performance tool to help improve existing codes. We are currently focusing on helping create OpenMP code. Since Cougar performs source-to-source translation only, it is not be dependent on any particular platform. We expect that OpenMP and MPI will be used together frequently and that many porting efforts will begin with an MPI code; we also expect that OpenMP will be implementable on some clusters in future, and we are working toward supporting SPMD style OpenMP, which promises to provide good performance on such systems. We will extend our parallel data flow analysis and improve the quality of our interprocedural analysis, both of which are of vital importance for OpenMP programs in the near future.

VII. REFERENCES

[1] L. Adhianto, B. Chapman, D. Lancaster, I. Wolton, M.Leuschel, and A. Kneer. "A Guidance Tool to Improve Memory Locality Reuse and To Exploit Hidden Parallelism in Loop Nests." *Technical Report* DSSE-TR-2001-4, September 2001.

[2] E. Ayguade, M. Brorsson, H. Brunst, H.-C. Hoppe, S. Karisson, X. Martorell, W. E. Nagel, F. Schlimbach, G. Utrera and M. Winkler. "OpenMP Performance Analysis Approach in the INTONE Project." *Third European Workshop on OpenMP* (EWOMP 2001), 2001

[3] V. Balasundaram, K. Kennedy, "A Techniques for Summarizing Data Access and its Use in Parallelism Enhancing Transformations," *Proceedings of ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, Jun. 1989.

[4] Banerjee. "Dependence Analysis for Supercomputing." *Kluwer Academic Publishers*, Boston, MA, 1988.

[5] B. Chapman, F. Bregier, A. Patil and A. Prabhakar. "Achieving Performance under OpenMP on ccNUMA and Software Distributed Shared Memory Systems." *Special Issue of Concurrency Practice and Experience*. 2001

[6] B. Chapman, A. Patil and A. Prabhakar. "Performance Oriented Programming for NUMA architectures." *Proc. WOMPAT 2001*, LNCS, Springer Verlag. 2001.

[7] B. Chapman, O. Hernandez, A. Patil and A. Prabhakar, "Program Development Environment for OpenMP on ccNUMA and NUMA Platforms," *Proc. Large Scale Scientific Computations 2001*, Sozopol

[8] R. Eigenmann, T. Mattson. "SC'01 Tutorial: Advanced OpenMP." *Proc. Supercomputing 2001*, Denver, Colorado, Nov. 10-16, 2001.

[9] Foresys, see http://www.simulog.fr/is/fore5.htm

[10] D. Grunwald and H. Srinivasan. "Data Flow Equations for Explicitly Parallel Programs. University of Colorado at Boulder." *Technical Report* CU-CS-605-92. Boulder, Colorado 1992.

[11] P. Havlak and K. Kennedy. "An implementation of interprocedural bounded regular section analysis." *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350--360, July 1991.

[12] C.S. Ierotheou, S.P. Johnson, M. Cross, and P. Legget, "Computer Aided Parallelisation Tools (CAPTools) – Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes," *Parallel Computing*, 22 (1996) 163-195. http://captools.gre.ac.uk/.

[13] X. Kong, D. Klappholz and K.Psarris. "An Improved Dependence Test for Automatic Parallelization and Vectorization." *IEEE Transactions on Parallel and Distributed Systems*, 2(3):342-349, July 1991.

[14] Zhiyuan Li. "Array privatization for parallel execution of loops." In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 313--322, 1992.

[15] Shih-Wei Liao "SUIF Explorer: An interactive and interprocedural parallelizer" Ph.D. Dissertation, Department of Computer Science, Stanford University, August 2000.

[16] M. Muller. "OpenMP Optimization Techniques: Comparison of Fortran and C Compilers." *Third European Workshop on OpenMP* (EWOMP 2001), 2001

[17] Kleanthis Psarris and Konstantinos Kyriakopoulos. "Data Dependence in Practice." *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*. Newport Beach, CA, October 1999.

[18] Makoto Satoh etl. "Interprocedural Parallelizing Compiler WPP and Analysis Information Visualization too Aivi" *Second European Workshop on OpenMP* (EWOMP 2000), 2000

[19] S. Satoh, K. Kusano, and M. Sato. "Compiler Optimization Techniques for OpenMP Programs." *Second European Workshop on OpenMP* (EWOMP 2000), 2000

[20] L. A. Smith, M, Bull. "Development of Mixed Mode MPI/OpenMP Applications. Technology watch report," available: http://www.ukhec.ac.uk/publications/tw/mixed.pdf, The University of Edinburgh

[21] R. Triolet, F. Irigoin, P. Feautrier, "Direct Parallelization of CALL Statements," *Proceedings of ACM SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, July 1986, pp. 176-185.

[22] P. Tu. "Automatic Array Privatization and Demand-Driven Symbolic Analysis." PhD thesis, Department of Computer Science, University of Illinois at UrbanaChampaign, August 1995.

[23] M.J. Wolfe. "Optimizing Supercompilers for Supercomputers." *The MIT Press*, Cambridge, MA, 1989.