Lecture Notes in Computer Science 1511

David O'Hallaron (Ed.)

Languages, Compilers, and Run-Time Systems for Scalable Computers

4th International Workshop, LCR '98
Pittsburgh, PA, USA, May 1998
Selected Papers

Springer

David O'Hallaron (Ed.)

Languages, Compilers, and Run-Time Systems for Scalable Computers

4th International Workshop, LCR '98
Pittsburgh, PA, USA, May 28-30, 1998
Selected Papers

Springer

QA76
.58
.W6
1998

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

David O'Hallaron
Computer Science and Electrical and Computer Engineering
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213-3891, USA
E-mail: droh@cs.cmu.edu

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Languages, compilers, and run time systems for scalable computers : 4th international workshop ; selected papers / LCR '98, Pittsburgh, PA, USA, May 28 - 30, 1998. David O'Hallaron (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1998
(Lecture notes in computer science ; Vol. 1511)
ISBN 3-540-65172-1

CR Subject Classification (1991): F.2.2, D.1.3, D.4.4-5, C.2.2, D.3, F.1, C.2.4, C.3

ISSN 0302-9743
ISBN 3-540-65172-1 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1998
Printed in Germany

Typesetting: Camera-ready by author
SPIN 10692671 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

98-45572

LIBRARY OF CONGRESS
DEC -2 1998
CIP

QA76
.58
.W658
1998

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

David O'Hallaron
Computer Science and Electrical and Computer Engineering
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213-3891, USA
E-mail: droh@cs.cmu.edu

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Languages, compilers, and run time systems for scalable computers : 4th international workshop ; selected papers / LCR '98, Pittsburgh, PA, USA, May 28 - 30, 1998. David O'Hallaron (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1998
(Lecture notes in computer science ; Vol. 1511)
ISBN 3-540-65172-1

CR Subject Classification (1991): F.2.2, D.1.3, D.4.4-5, C.2.2, D.3, F.1, C.2.4, C.3

ISSN 0302-9743
ISBN 3-540-65172-1 Springer-Verlag Berlin Heidelberg New York

Table of Contents

Expressing Irregular Computations in Modern Fortran Dialects 1
Jan F. Prins, Siddhartha Chatterjee, and Martin Simons

Memory System Support for Irregular Applications 17
John Carter, Wilson Hsieh, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, Leigh Stoller, and Terry Tateyama

MENHIR: An Environment for High Performance Matlab 27
Stéphane Chauveau and François Bodin

On the Automatic Parallelization of Sparse and Irregular Fortran Programs 41
Yuan Lin and David Padua

Loop Transformations for Hierarchical Parallelism and Locality 57
Vivek Sarkar

Dataflow Analysis Driven Dynamic Data Partitioning 75
Jodi Tims, Rajiv Gupta, and Mary Lou Soffa

A Case for Combining Compile-Time and Run-Time Parallelization 91
Sungdo Moon, Byoungro So, Mary W. Hall, and Brian Murphy

Compiler and Run-Time Support for Adaptive Load Balancing in Software Distributed Shared Memory Systems 107
Sotiris Ioannidis and Sandhya Dwarkadas

Efficient Interprocedural Data Placement Optimisation in a Parallel Library 123
Olav Beckmann and Paul H. J. Kelly

A Framework for Specializing Threads in Concurrent Run-Time Systems .. 139
Gregory D. Benson and Ronald A. Olsson

Load Balancing with Migrant Lightweight Threads 153
David Cronk and Piyush Mehrotra

Integrated Task and Data Parallel Support for Dynamic Applications 167
James M. Rehg, Kathleen Knobe, Umakishore Ramachandran, Rishiyur S. Nikhil, and Arun Chauhan

Supporting Self-Adaptivity for SPMD Message-Passing Applications 181
M. Cermele, M. Colajanni, and S. Tucci

Evaluating the Effectiveness of a Parallelizing Compiler 195
Dixie Hisley, Gagan Agrawal, and Lori Pollock

Comparing Reference Counting and Global Mark-and-Sweep on Parallel Computers 205
Hirotaka Yamamoto, Kenjiro Taura, and Akinori Yonezawa

Design of the GODIVA Performance Measurement System 219
Terrence W. Pratt

Instrumentation Database for Performance Analysis of Parallel Scientific Applications 229
Jeffrey Nesheiwat and Boleslaw K. Szymanski

A Performance Prediction Framework for Data Intensive Applications on Large Scale Parallel Machines 243
Mustafa Uysal, Tahsin M. Kurc, Alan Sussman, and Joel Saltz

MARS: A Distributed Memory Approach to Shared Memory Compilation . 259
M.F.P. O'Boyle

More on Scheduling Block-Cyclic Array Redistribution 275
Frédéric Desprez, Stéphane Domas, Jack Dongarra, Antoine Petitet, Cyril Randriamaro, and Yves Robert

Flexible and Optimized IDL Compilation for Distributed Applications 288
Eric Eide, Jay Lepreau, and James L. Simister

QoS Aspect Languages and Their Runtime Integration 303
Joseph P. Loyall, David E. Bakken, Richard E. Schantz, John A. Zinky, David A. Karr, Rodrigo Vanegas, and Kenneth R. Anderson

The Statistical Properties of Host Load 319
Peter A. Dinda

Locality Enhancement for Large-Scale Shared-Memory Multiprocessors ... 335
Tarik Abdelrahman, Naraig Manjikian, Gary Liu, and S. Tandri

Language and Compiler Support for Out-of-Core Irregular Applications on Distributed-Memory Multiprocessors 343
Peter Brezany, Alok Choudhary, and Minh Dang

Detection of Races and Control-Flow Nondeterminism 351
Mindong Feng and Chung Kwong Yuen

Improving Locality in Out-of-Core Computations Using Data Layout Transformations 359
M. Kandemir, A. Choudhary, and J. Ramanujam

Optimizing Computational and Spatial Overheads in Complex Transformed Loops 367
Dattatraya Kulkarni and Michael Stumm

Building a Conservative Parallel Simulation with Existing Component Libraries 378
Chu-Cheow Lim and Yoke-Hean Low

A Coordination Layer for Exploiting Task Parallelism with HPF 386
Salvatore Orlando and Raffaele Perego

InterAct: Virtual Sharing for Interactive Client-Server Applications 394
Srinivasan Parthasarathy and Sandhya Dwarkadas

Standard Templates Adaptive Parallel Library (STAPL) 402
Lawrence Rauchwerger, Francisco Arzu, and Koji Ouchi

Author Index 411

This material may be protected by Copyright law (Title 17 U.S. Code)

Menhir: An Environment for High Performance Matlab

Stéphane Chauveau, François Bodin
schauvea@irisa.fr, bodin@irisa.fr

IRISA-INRIA
Campus de Beaulieu, 35042 Rennes, FRANCE

Abstract. In this paper we present MENHIR a compiler for generating sequential or parallel code from the MATLAB language. The compiler has been designed in the context of using MATLAB as a specification language. One of the major features of MENHIR is its retargetability that allows generating parallel and sequential C or Fortran code. We present the compilation process and the target system description for MENHIR. Preliminary performances are given and compared with MCC, the MathWorks MATLAB compiler.

1 Introduction

Current approaches for the development of numerical applications are frequently decomposed in two phases. First a prototype or specification is written using a popular high level tool such as MATLAB [10, 11]. Then, the application is rewritten in Fortran or C so efficiency can be achieved. When high performance is needed, the application is also parallelized. This manual process is error prone and very time consuming which makes MATLAB compilers very attractive, especially when the target is a parallel computer.

Automatically generating an efficient implementation from a MATLAB code encompasses two main aspects. First, the generated code must be efficient and able to exploit a wide range of architectures from sequential computers to parallel ones. Secondly the user must be able to change the generated code so it can be automatically inserted or exploited in an existing development environment. For instance, when the user has special data structures for implementing its matrices and the corresponding specific highly optimized libraries the compiler should be able to exploit them.

In this paper we present MENHIR (Matlab ENvironment for HIgh peRformance), a multi-target compiler for MATLAB 4.2[1]. The main feature of MENHIR is its target description system that allows addressing sequential and parallel systems. The user may add its own data structures and functions to this description, thus enabling better code generation. To exploit parallelism MENHIR relies on libraries such as ScaLapack [5].

[1] With a few restriction such as the operators eval, feval that take MATLAB statements as input are not supported.

In Section 2, we present a short summary of related tools. In Section 3, we overview the MENHIR's target system description (MTSD). In Section 4, we describe the type analysis and the code generation method used in MENHIR. Finally in Section 5, we present preliminary performance results on a set of MATLAB programs running on sequential and parallel architectures.

2 Related Works

Number of studies hava already been based on the MATLAB language. The first set of tools are interpreted MATLAB clones such as SCILAB and OCTAVE. Another class of tools proposes parallel extensions to MATLAB such as message passing in MultiMatlab [12] or a client-server mechanism in MathServer [8] or IPSC-MATLAB [4]. The main difference between those tools and MATLAB compilers is the lack of static type analysis.

Three existing compilers MCC, distributed by THE MATHWORKS INC, MATCOM [13], distributed by MATHTOOLS, and Falcon are available to generate efficient codes. The FALCON [9, 7] system encompasses a compiler and an interactive environment for transforming and optimizing MATLAB programs. The Falcon project was focused on the type analysis, one of the keypoint in compiling MATLAB. MENHIR differs from these two systems by two aspects. Firstly MENHIR relies on a retargetable code generator which Falcon, MATCOM and MCC do not. Secondly, MENHIR is able to exploit parallel numerical libraries.

3 Menhir's Target System Description (MTSD)

The MTSD is a major component of MENHIR. Its goal is to describe the property and implementation details of target system. The target system, in our case, is a programming language such as Fortran, C or other and a linear algebra library. The MTSD indicates to MENHIR how to implement the matrix data structures as well as all the MATLAB operators and functions. MENHIR code generation algorithm is independent from the target system. In Table 1, we present the main constructs of the MTSD. These constructs indicate to MENHIR how to allocate and access matrices and declare the library functions which implements MATLAB operators and functions.

As shown in Table 1.b, the data structures are described in an object oriented manner. Each data structure is described in a `class` construct which members specify its properties. The fields `elem`, `shape`, `minreal`, `maxreal`, `minimag` and `maximag` respectively indicate the basic type of the elements of the data structure, the rank of the structure and a complex value interval. The `prop` field is used to declare properties about the content of the objects. These properties are declared as shown in Table 1.a. They are associated to classes, variables and expression results and propagated by the type analysis presented in Section 4.1. The identifier `bname` indicates that the data structure is inherited from class `bname`. This mechanism allows to define new classes with different properties but which share the same implementation. In example Table 1.b, the class `UTMatReal`

is declared to the compiler to be an upper triangular matrix with element values in 0 to 2^{16} which is implemented with the same data structure as the real matrix (i.e. `MatReal`). The default data structures to be used by the code generator are declared as shown Table 1.c.

The declaration of the accesses to data structure elements is shown Table 2.g while code generation for memory allocation is illustrated in Table 1.j. The `assign` and `cast` constructs are provided to allow the compiler to copy and convert data structures from one type to another. Cast operations are necessary to ensure that the result of operators/functions can be converted in a format suitable to the routines exploiting the results. They are shown Table 1.f.

MATLAB operators and functions are declared as shown in Table 1.d and 1.e. The code to generate is an expression or a sequence of statements. For each parameter in the list, an attribute (*att*) and a class name (*cname*) are given. The attribute indicates how the parameter is used in the target code ; `out` for a result, `const` for a constant variable, `expr` for an expression (i.e. without an address) etc. The type indicates the data structure `class` name. Contrary to operators, MATLAB functions are declared in two parts (Table 1.e). First some target subroutines are declared. Then, they are gathered in the `function` construct. MATLAB control statements are defined similarly as shown in Table 2.h.

The conform statement iterators given in Table 2.i are used by the compiler to implement conform statement, such as the point-wise addition of two matrices, in an efficient manner that minimizes the number of temporary variables. For instance, if we consider the MATLAB expression $R = B - A * x$, an implementation based on library function calls would imply to generate the code in two parts, first $T = A * x$ and then $R = B - T$ resulting in poor performance. Instead MENHIR generates the following C code using the `loop` construct:

```
...
for (tmp136 = 1; tmp136 <= tmp128; tmp136++) {
   for (tmp135 = 1; tmp135 <= tmp127; tmp135++) {
      (*tmp133++) = ((*tmp129++)-
         ((*tmp131++)*tmp94));
   }
   tmp129 = &tmp129[tmp130];
   tmp131 = &tmp131[tmp132];
   tmp133 = &tmp133[tmp134];
}
...
```

As shown by this example, the generated code can be close to what "hand programming" would produce. In the case of parallel code generation, a similar principle, shown in Table 2.k, is used to describe the scanning of a distributed matrix local elements.

Syntax	Example
a) Properties `property` *name* ;	`property DIAG ;` `property UPPERTRI ;`

b) Data Structure
class *name* : *bname*
`elem =` *id*
`shape =` *scal,row,col,matrix*
`minreal =` *constant* ;
`maxreal =` *constant* ;
`minimag =` *constant* ;
`maximag =` *constant* ;
`prop =` *list of properties*;
end
	`class UTMatReal: MatReal`
	`elem = Real ;`
	`shape = matrix;`
	`minreal = 0 ;`
	`maxreal =` 2^{16};
	`prop = UPPERTRI;`
	`end`

c) Default Data Structure `default` *shape*, *elemtype* [,*prop*] = *cname*	`default matrix,complex=MatComplex;` `default matrix,real,DIAG=DiagMatReal;`
d) MATLAB Operators `inline` *name* (*att cname var1,...*) `=>` *statement 1* `=> ...` ; `inline` *res name* (*att cname var1,...*) "*expression*" ;	`inline real @op_add(real A,int B)` `"(A+((double)B))" ;`

e) MATLAB Functions
`inline func1(`*parameters*`)`
`=> code ;`
;
`function` *res* = *name*(*parameter*)
`add` *func1*;
`add` *func2*;
`end function`
	`inline C_lup_real(out MatReal MATL,`
	`out MatReal MATU, ...,MatReal MATA)`
	`=> lup_RRI_R(&MATL,&MATU,`
	`&MATP,MATA) ;`
	;
	`function [L,U,P] = lu(A)`
	`add C_lup_real(L,U,P,A) ;`
	`end function`

f) Assign and Cast Operators
`inline @assign(out` *cname r*,
`const` *cname i*)
`=>` *statement 1* ;
`=> ...;`
;
`inline` *cname* `@cast(`*att cname var*`)`
"*expression*"
;
	`inline @assign(out MatReal DEST,`
	`const TranspMatReal SRC)`
	`=> transpose_R_R(&DEST,&SRC) ;`
	;
	`inline int @cast(RowInt VAR)`
	`"VAR.get(1)"`
	;

Table 1. Target System Description Content.

Syntax	Example

g) Index Accesses
`inline @get|put|...()`
`=> statement 1 ;`
`=> ...;`
`;`
	`inline real @get(MatReal MAT)`
	`"get_Rii(&MAT,I1,I2)" ;`
	`inline @put(MatReal MAT,real VAL)`
	`=> set_Riir(&MAT,I1,I2,VAL) ;`
	`;`

h) MATLAB Control Statement
`inline @do(real CPT,real START,`
`const real STEP,real END)`
`=> loop code(BODY);`
`;`
	`inline @do(real I,real C0,`
	`const real S,real C1)`
	`=> for(I=C0;I<=C1;I+ =S){`
	`=> BODY`
	`=> }`
	`;`

i) Conform Statement Iterators
`inline @loop(local variablelist)`
`=> statement 1(BODY) ;`
`=> ...;`
`;`
	`inline @loop(local int I1,int I2)`
	`=> for (I1=1;I1<=DIM1;I1++)`
	`=> for (I2=1;I2<=DIM2;I2++){`
	`=> BODY`
	`=> }`
	`;`

j) Memory Management, ...
`inline @declar(cname var)`
`=> statement 1 ;`
`=> ...;`
`;`
`inline @alloc(cname var,`
`int DIM1,int DIM2)`
`=> statement 1 ;`
`=> ...;`
`;`
	`inline @declar(MatReal MAT)`
	`=> MatReal *MAT ;`
	`;`
	`inline @alloc(MatReal VAR,`
	`int DIM1,int DIM2)`
	`=> alloc_R(&VAR,DIM1,DIM2) ;`
	`;`

k) Local scan of distributed matrix
`defaccess matrix name ;`
`inline @loopinit:name(cname var)`
`=> statement 1 ;`
`;`
`inline @loop:name(parameter list)`
`=> statement 1 ;`
`;`
`inline real @get:name(cname var)`
`"(expression)" ;`
	`defaccess matrix plocal ;`
	`inline @loopinit:plocal(PMatReal MAT)`
	`=> LOCD1 = MAT.local.dim1;`
	`=> LOCD2 = MAT.local.dim2;`
	`;`
	`inline @loop:plocal(..., int LOCD1,`
	`int LOCD2)`
	`=> STAT1`
	`=> for (I2=1;I2<=LOCD2;I2++){`
	`=> for (I1=1;I1<=LOCD1;I1++) {`
	`=> BODY }`
	`=> STAT2 }`
	`;`
	`inline real @get:plocal(PMatReal MAT)`
	`"(*PTR++)" ;`

Table 2. Target System Description Content.

4 Overview of Menhir's Compilation Process

MENHIR's compilation process is divided in the following steps:

1. **lexical and syntactic analysis:** this step performs the lexical and syntactic analysis of the MATLAB M-Files.
2. **identifiers analysis:** this preliminary step of the type analysis determines the nature of the identifier. The nature of an identifier can be a script file (M-file), a function or a variable. The ambiguities are resolved statically and the compiler assumes that the nature of an identifier is constant in a program unit (MATLAB allows a identifier to reference a variable and a function or a script in the same program unit).
3. **function cloning and restructuring in canonical form:** At this step, all functions are cloned and most of the special cases of MATLAB are removed by expanding them in a canonical form. Temporary variables are added to limit code expansion for large expressions. All runtime choices are expanded at that step. For instance, the statement `A=B+C` is expanded in:

```
if (dim1(B)==1 & dim2(B)==1) then
   if (dim1(C)==1 & dim2(C)==1) then
      A = B+C % scalar+scalar
   else
      A = B+C % scalar+matrice
...
```

   a simple local type analysis is applied locally to each expression to avoid useless cases. Later, after the type analysis has been performed, dead code elimination is applied.
4. **type analysis and dead code elimination:** this type analysis determines the properties of each expression and variable according to the semantic of MATLAB and the target system description. If an operator or a function returns an object with special properties, they are propagated. The type analysis is completed by removing all dead codes (i.e. runtime operator/function selection that have been solved statically). The remaining conditionals added at step 3 are left as run-time checking.
5. **data structure selection:** This step selects an implementation for each variable according to the type computed in previous step and the defaults given in the MTSD (Table 1.c).
6. **operators and functions implementation selection:** At this step operator and function implementations are selected. Data structure casts are inserted to ensure that returned results can be used as input parameters of the subsequent function calls.
7. **code issue:** This step gathers the selected implementations to issue the final code.

In the following, we present in more details steps 4 and 6.

4.1 Type Analysis

The type analysis is a major component in MENHIR as its result has a great impact on the quality of the generated code. In MATLAB, all operators and functions are polymorph and the default is to implement all objects as complex matrices. Furthermore, variable types can change during the program execution and matrices can be resized by any assignment. An accurate analysis limits runtime type checking overheads and allows selecting the appropriate library methods (for instance, operators on floats instead of complex variables).

A type in MENHIR is defined as a set of possible shapes (*scalar, row vector, column vector* and *matrix*), an interval of complex value, a string, an integer status and a set of numerical properties. A class from the MTSD defines such a type. The properties are used to select optimized implementations. To illustrate this feature consider the following MATLAB statement sequence:

```
A = triu(...);
...
v=b/A;
```

MATLAB function `triu` returns a matrix that is upper triangular. To execute efficiently specialized solver (this solver does not check at runtime if matrix `A` is upper triangular). This is achieved in MENHIR by having the following declaration in the MTSD that use the **UPPERTRI** property that is propagated by the type analysis:

```
property UPPERTRI ;
class UTMatReal : MatReal
   prop = UPPERTRI;
end ;
inline triu_Real(out UTMatReal A, MatReal B)
      => ... ;
      ;
function [A] = triu(B)
add triu_Real;
...
end function
inline @op_matdiv(MatReal RES,
   MatReal PARAM1,
   UTMatReal PARAM2)
      => ...;
      ;
```

This powerful feature is not restricted to MATLAB functions and operators. User's libraries can be exploited in a similar fashion and new properties can be created at any time.

The type propagation algorithm is interprocedural and based on an iterative data flow solver [1, 6] that is closed to the one proposed by DeRose [9]. The analysis computes a superset of the possible values for each variable and expression

at each point of the program. However, in many cases, there is not enough information to propagate in a MATLAB program to compute accurately the types describing these supersets. Via directives, MENHIR allows specifying information on objects that are used by the type analysis. The most frequent directives give information about the contents (i.e. the shape, the properties or the element types) of some variables at a given point of the program. These directives cannot be regarded as global type declarations since their scope is limited and they do not specify the implementation of the objects. A variable *declared* as a scalar in a point of program can become a matrix or a vector in another point.

The second kind of directives is used to simplify the semantic of the MATLAB language. For instance the directive

```
%$VAR var [,var] : [no] RESIZE/MASK
```

allows to declare if a variable is subject to resizing or to be acceded using a mask index[2].

4.2 Code Generation

The code generation must solve three problems:

1. selecting the data structure for each variable and expression according to the type analysis
2. selecting the implementations (the methods) for MATLAB operators and functions
3. inserting data structure cast operations

These choices are interdependent. The methods whose operand data structures match more the current parameters data structures are most suitable. Conversely, it seems interesting to adapt the data structures to the methods using them. To illustrate this problem, consider the class C_1 and C_2 and the implementations of **F**, $M_1 : C_1 \rightarrow C_2$ and $M_2 : C_2 \rightarrow C_1$, in the following MATLAB code:

```
for i=1:n
   B=F(A) ; C=F(B) ; A=F(C) ;
end
```

The figure 1 shows how the initial choice for the data structure of **A** can influence the remaining of the analysis.

In MENHIR, the choice of variable data structures is made first according to the result of the type analysis and to the defaults given in the MTSD. Although imperfect, this strategy is acceptable since only a few types are provided into the MTSD.

Then, the selection for operators/functions and the insertion of cast operations is made. This is performed by building *on the flight* a directed acyclic graph. Vertices correspond to variables, MATLAB operators/functions and data

[2] A mask variable in MATLAB is a matrix containing only 0 or 1 element values used as booleans.

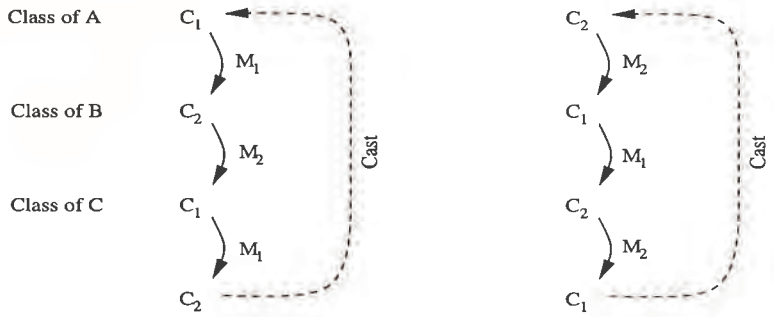

Fig. 1. Selection of the implementation according to the initial data structure of **A**

structure conversion subroutine. An example of such a graph is shown figure 2. Code generation consists in selecting a set of paths that covers all program statements and variables (indicated in gray in Figure 2).

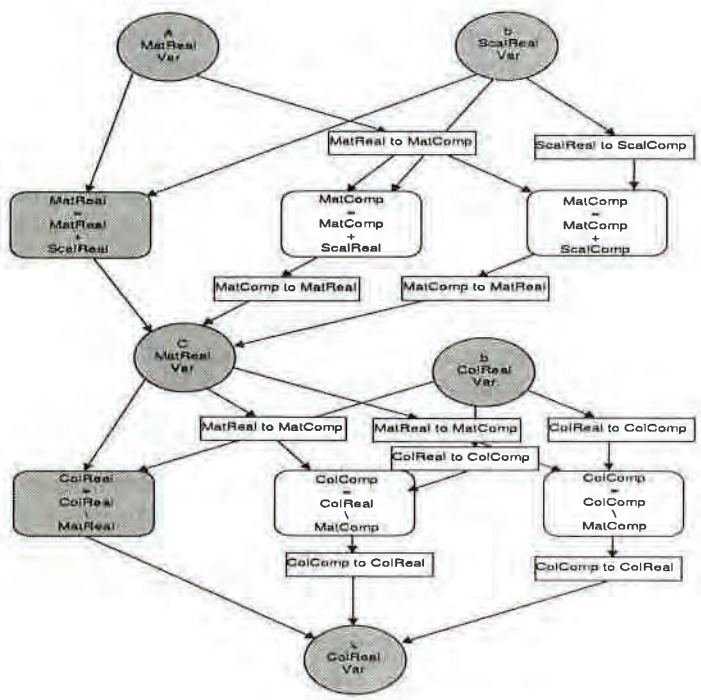

Fig. 2. Code generation graph for MATLAB code `C=A+b; x=C/d;`. `MatComp`, `MatReal`, `ColReal`, `ColComp`, and `ScalReal`, respectively denote complex matrices, float matrices, float column vectors, complex column vectors, and float scalars data structures.

A score is associated to each cast and each method according to the shapes of the data structures to which they are applied. Then, the algorithm looks for a path with a lowest score on a subset of the graph. The costly part of the algorithm is not the shortest path computation itself but the creation of the subgraphs that represents the data structures casts. Each of them depends on the type analysis result for the local value and can contain several nodes. In many cases, the construction of the graph is avoided by choosing as preset cast (i.e. applied to generic types). For more details the reader can refer to [3].

The last stage consists in assembling the codes of the selected methods for the expressions, the instructions and the casts. The syntax of the target language, except for the literals and the identifiers, is entirely described by the **MTSD**.

5 Preliminary Performances

In this section, we present preliminary performance obtained using MENHIR on a single processor Sun workstation and on a parallel computer Onyx SGI with 4 processors R10000. The sequential codes are generated using a target system description based on the Lapack library [2] while the parallel codes are obtained using ScaLapack [5]. The parallel codes are SPMD and the same data distribution (block-cyclic) is used for all matrices. Only a very few directives (20 for all benchmarks) were inserted in the MATLAB codes. Because MENHIR relies on parallel libraries and conform operations but not on MATLAB loops, program that does not exhibit coarse grain matrix operations was not run in parallel. This is the case of the benchmarks *gauss*, a MATLAB program computing a gaussian elimination written in a Fortran style, *chol*, an incomplete cholesky factorization, *pfl*, an iterative method for eigenspace decomposition by spectral dichotomy, and *INT1*, *INT4* two differential equation integration algorithms.

The size of the MATLAB programs and the generated code is given in Table 3. Cloning, performed at step 3 of the compilation process, can substantially generate large target codes as shown for the program *pfl*. MENHIR is more efficient than MCC, the MathWorks MATLAB compiler, except for *chol*. The main loop of *chol* is a **WHILE** statement that computes a lot of integer expressions. Some directives specify that the handled variables remain in the integer range $[-2^{31}, 2^{31}-1]$ but they have no effect on intermediate calculations, which must be done with the float arithmetic. MCC gives better performance since it uses the integer arithmetic. However, this heuristic does not always produce the expected results.

gauss	70	200
chol	180	620
pfl	200	9000
INT 1	140	4700
INT 4	200	3100

Table 3. MATLAB program code sizes and corresponding generated C code sizes.

Two programs illustrate the parallel code generation: a conjugate gradient algorithm with preconditioning and a Jacobi computation.

Figure 4 gives the execution times for the conjugate gradient. On the x axis, MATLAB 4.2 and 5 correspond to the MATLAB interpreters execution time, MCC is the execution time of the code produced by the MathWorks compiler, MENHIR is the time for the sequential code generated by our compiler and ONYX$n \times m$ and PARAGON$n \times m$ are the execution times on the Onyx and the Paragon parallel computers. As it can be seen, MENHIR performs better that the interpreter and MCC except in one case, *chol*. In this case MENHIR's code contains float to integer conversions that slow down the algorithm. In general, the code produced by MENHIR is closed, for most of the benchmarks of the "hand-coded" versions of the algorithms. However, these "hand coded" versions were not aggressively optimized.

Parallel codes reach good speedups thanks to the ScaLapack library but also to the type analysis[3]. Figure 5 gives the execution times for two versions of the same version of Jacobi compiled with different MTSD. In the second version, good performance is obtained when the MTSD propagates the information that the preconditioning matrix M, shown Figures 3, is diagonal. Then, the run-time checking of this property disappears and a large amount of memory space and communication is saved.

```
...
m=size(A,1) ;
n=size(A,2) ;
[ M, N ] = split( A , b, 1.0, 1 ) ;
for iter = 1:max_it,
   x_1 = x;
   y = N*x + b ;
   x = M /y ;
   error = norm(x-x_1 )/norm( x );
   if ( error <= tol ), break, end
end
...
```

Fig. 3. Main loop of the Jacobi benchmark.

[3] We don't have MCC or MATLAB on the Onyx computer, so the performance numbers in sequential are not given for this architecture.

6 Conclusion

In this paper we briefly present MENHIR, a **retargetable** compiler for MATLAB. The strength of MENHIR is its original target system description that allows generating code that exploits optimized sequential and parallel libraries. Performance shows that the generated code is in most of the cases more efficient than the one obtained by the Mathworks compiler MCC on sequential workstation. Future work will focus on exploiting more aggressively parallelism by also considering MATLAB loops.

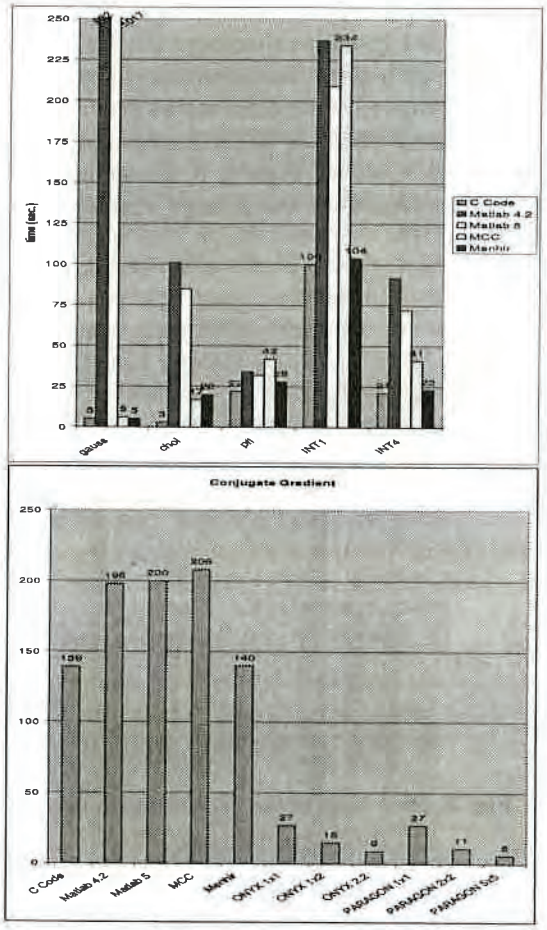

Fig. 4. Execution time in seconds of the sequential and parallel generated C codes.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addisson-Wesley Publishing Compagny, 1985.
2. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, second edition, 1995.
3. Stéphane Chauveau. *MENHIR. un environnement pour l'exécution efficace des codes Matlab (in French).* PhD thesis, Université de Rennes 1, February 1998.
4. Shirish Chinchalkar. IPSC-MATLAB reference manual. Technical Report 92-106, Cornell Theory Center, September 1992.
5. J. Choi, J. DEmmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. Technical Report CS-95-283, Computer Science Dept., University of Tennesse, Knoxville, 1995. (LAPACK Working Note 95).
6. John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 31(1):158–171, January 1976.

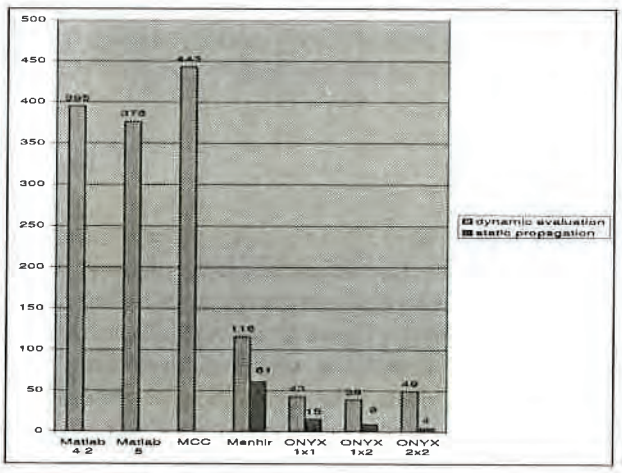

Fig. 5. Execution time in seconds for the jacobi code with and without the static propagation of the preconditioning matrix diagonal nature.

7. Bret Andrew Marsolf. Techniches for the interactive development of numerical linear algebra libraries for scientific computation. Master's thesis, University of Illinois at Urbana-Champaign, 1997.
8. M. Rezny. MATHSERVER: A client-server approach to scientific computation. Department of Mathematics, The University of Queensland, Australia.
9. Luis De Rose. compiler techniques for MATLAB programs. Master's thesis, University of Illinois at Urbana-Champain, 1996.

10. INC THE MATHWORKS. *MATLAB, High-Performance Numeric Computation and Visualization Software. Reference Guide*, August 1992.
11. INC THE MATHWORKS. *MATLAB, High-Performance Numeric Computation and Visualization Software. User's Guide*, August 1992.
12. Anne E. Trefethen, Vijay S. Menon, Chi-Chao Chang, Grzegorz J. Czajkowski, Chirs Myers, and Lloyd N. Trefethen. MultiMATLAB: MATLAB on multiple processors. Technical Report 239, Cornel Theory Center, 1996.
13. Keren Yaron. MATCOM, a Matlab to C++ Translator, March 1996.

Lecture Notes in Computer Science

This series reports new developments in computer science research and teaching, quickly, informally, and at a high level. The timeliness of a manuscript is more important than its form, which may be unfinished or tentative. The type of material considered for publication includes

- drafts of original papers or monographs,
- technical reports of high quality and broad interest,
- advanced-level lectures,
- reports of meetings, provided they are of exceptional interest and focused on a single topic.

Publication of Lecture Notes is intended as a service to the computer science community in that the publisher Springer-Verlag offers global distribution of documents which would otherwise have a restricted readership. Once published and copyrighted they can be cited in the scientific literature.

Manuscripts

Lecture Notes are printed by photo-offset from the master copy delivered in camera-ready form. Manuscripts should consist of no fewer than 100 and preferably no more than 500 pages of text. Authors of monographs and editors of proceedings volumes receive 50 free copies of their book. Manuscripts should be printed with a laser or other high-resolution printer onto white paper of reasonable quality. To ensure that the final photo-reduced pages are easily readable, please use one of the following formats:

Front size (points)	Printing area (cm)	Printing area (inches)	Final size %
10	12.2 x 19.3	4.8 x 7.6	100
12	15.3 x 24.2	6.0 x 9.5	80

On request the publisher will supply a leaflet with more detailed technical instructions or a TEX macro package for the preparation of manuscripts.

Manuscripts should be sent to one of the series editors or directly to:

Springer-Verlag, Computer Science Editorial III, Tiergartenstr.17,
D-69121 Heidelberg, Germany

ISBN 3-540-65172-1

9 783540 651727

ISSN 0302-9743

http://www.springer.de