Menhir: An environment for high performance Matlab

Stéphane Chauveau and François Bodin *
IRISA-INRIA, Campus de Beaulieu, 35042 Rennes, France
E-mail: {schauvea, bodin}@irisa.fr

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 to generate 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 [6]. 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 which allows to address sequential and parallel computers. 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 [3].

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 have already been based on the MATLAB language. The first set of tools are interpreted MATLAB clones such as SCILAB [10] and OCTAVE [4]. Another class of tools proposes parallel extensions to MATLAB such as message passing in MultiMatlab [11] or a client-server mechanism in MathServer [8].

Two existing compilers MCC, distributed by MATH WORKS, and Falcon are available to generate efficient codes. The FALCON [9,5] system encompasses a compiler and an interactive environment for transforming and optimizing MATLAB programs. The Falcon project was focused on type analysis which is one of the key points in compiling MATLAB. MENHIR differs from this two systems by two aspects. Firstly MENHIR relies on a retargetable code generator which Falcon and MCC do not. Secondly, MENHIR is able to exploit parallel numerical libraries.

Ramaswamy et al. [7] have developed a compiler to exploit simultaneously the task and data parallelism. This is not a full MATLAB compiler; it accepts only a small subset of the language.

*Corresponding author.

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

Scientific Programming 7 (1999) 303–312
ISSN 1058-9244 / $8.00

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 implement 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 expressions 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 defines 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 $2^{16} - 1$ 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 1(g) while code generation for memory allocation is illustrated in Table 2(j). The `assign` and `cast` constructs are provided 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 type ($classname$) 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 1(h).

The conform statement iterators given in Table 1(i) are used by the compiler to implement conform statements, 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, see Program Code 1.

As this is illustrated on 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 1(k), is used to describe the scanning of a distributed matrix local elements.

```
...
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];
}
...
```

Program Code 1.

Table 1
Target system description content. Examples are given for target language C

What	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*] = *classname*	`default matrix,complex = MatComplex;` `default matrix,real,DIAG = DiagMatReal;`
(d) MATLAB Operators	`inline` *name* (*att classname var1,...*) => *statement 1* => ... ; `inline` *res name* (*att classname 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` *func*1;
	`add` *func*2;
	`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` *classname r*,
	`const` *classname i*)
	=> *statement 1* ;
	=> ...;
	;
	`inline` *classname* `@cast(`*att classname var*)
	"*expression*"
	;
		`inline @assign(out MatReal DEST,`
		`const TranspMatReal SRC)`
		`=> transpose_R_R(&DEST,&SRC);`
		`;`
		`inline int @cast(RowInt VAR)`
		`"VAR.get(1)"`
		`;`

(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 CPT,real START,`
		`const real STEP,real END)`
		`=> for(CPT=START;CPT<=END;`
		`=> CPT+=STEP){`
		`=> BODY`
		`=> }`
		;

Table 1
(Continued.)

What	Syntax	Example

(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, Declaration, Target Language Statements	`inline @declar(`*classname var*`)`
	`=>` *statement 1* `;`
	`=> ...;`
	`;`
	`inline @alloc(`*classname var*`,`
	`int` *DIM*`1,int` *DIM*`2)`
	`=>` *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*`(`*classname var*`)`
	`=>` *statement 1* `;`
	`;`
	`inline @loop:`*name*`(`*parameter list*`)`
	`=>` *statement 1* `;`
	`;`
	`inline real @get:`*name*`(`*classname var*`)`
	`"(expression)";`
		`defaccess matrix paralocal;`
		`inline @loopinit:paralocal(PMatReal MAT)`
		`=> LOCD1 = MAT.local.dim1;`
		`=> LOCD2 = MAT.local.dim2;`
		`;`
		`inline @loop:paralocal(..., int LOCD1, int LOCD2)`
		`=> STAT1`
		`=> for (I2=1;I2<=LOCD2;I2++){`
		`=> for (I1=1;I1<=LOCD1;I1++) {`
		`=> BODY }`
		`=> STAT2 }`
		`;`
		`inline real @get:paralocal(PMatReal MAT)`
		`"(*PTR++)";`

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 each identifier. The nature of an identifier can be a script file (M-file), a function or a variable.
(3) **function cloning and restructuring in canonical form:** At this step all functions are cloned and most of the special cases of MATLAB are reduced by expanding them in a canonical form. At this step temporary variables are introduced to limit code expansion for large expressions. All runtime choices are expanded at that step.
(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 special properties of an object then they are propagated. Once done, the type analysis is completed by removing all dead codes (i.e., runtime operator/function selections that have been solved

statically). The remaining conditionals 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 default 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 it strongly influence 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 reduces runtime type checking overheads and allows to select 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 values, a string/integer status and a set of properties defined by the classes in the MTSD.

The type analysis proceeds as follow. First it expands all choices that can happened at run-time and then, using the type analysis removes, when possible, previously introduced tests:

1. Modifying the MATLAB abstract syntax trees considers two cases:
   (a) The checking of matrices' ranks and dimension sizes is added. Fig. 1 shows the transformation to take into account the possible rank of the variables for the MATLAB expression `(A+B)`. In the body of the conditionals the variables ranks are known.
   (b) Inlining of the possible implementation offered by the MTSD is performed using the SELECT construct. The SELECT statement is similar to a classical SWITCH statement but allows to have more than one case to be verified. The SELECT construct is illustrated in Fig. 2. This figure shows the computation of the maximum of two scalar values. If the values are integers then both methods can be used indifferently. Because, the compiler assumes that cases' order reflect the implementation's cost (the first ones the cheapest) it chooses the integer maximum. In a non-retargetable compiler, the effect of the SELECT construct would be hardwired in the type analyzer and the code generator.

```
if (dim(A)==1)
  if (dim(B)==1)
    % scalar case
    D=max(A(1×1),B(1×1))(1×1)*C ;
  else
    D=max(A(1×1),B(p×q))(p×q)*C ;
  end
else
  if (dim(B)==1)
I1 : D=max(A(m×n),B(1×1))(m×n)*C ;
  else
    if (dim1(A)==dim1(B) et dim2(A)==dim2(B))
      D=max(A(m×n),B(p×q))(m×n)*C ;
    else
      error(...)
    end
  end
end
```

Fig. 1. Expansion of the expression `max(A,B)`.

2. The second step propagates the variable's types. This information allows to remove dead code and tests that might occur at run-time. Tests that have not be removed at compile time are left for run-time selection of the proper methods for implementing the operators. The type propagation algorithm is interprocedural and based on an iterative data flow solver [1] which is closed to the one proposed by DeRose [9].

The information on element's value type (integer, float or complex) does not have the same status as the rank and sizes of the dimensions of matrices. Indeed, it is always possible, in MATLAB, to use complex computation instead of integer or float ones. We use this possibility to reduce the code expansion. If the type analysis finds out that a variable is either an integer or a complex value, then only the complex computation is used. This may, in practice, decrease the efficiency of the generated code.

To select optimized functions, the MTDS class properties are also propagated by the type analysis. 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 instruction `v=b/A`, we want the generated code to call a specific solver that does not check at runtime if matrix `A` is upper triangular. This is achieved in MENHIR by having the following declarations in the MTSD:

```
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)
      => ...;
      ;
```

```
SELECT
   CAS (A and B are integers)
     C = max_int(A,B) ;
   CAS (A and B are reals)
     C = max_float(A,B) ;
   DEFAUT
     error(...)
END
```

Fig. 2. SELECT statement.

This powerful feature is not restricted to MATLAB functions and operators. User's libraries can be exploited in a similar fashion.

4.2. Directives to improve the type analysis

However, in many cases, there is not enough information to propagate in a MATLAB program to compute accurately the objects' types. Via directives, MENHIR allows to specify information on objects that are used by the type analysis. For instance the directive `%$VAR x : no MASK` declares that the variable `x` is not accessed using a mask index. (A mask variable in MATLAB is a matrix containing only 0 or 1 element values used as booleans.) Others directives available in MENHIR allow to indicate to the compiler the shape and element types of variables.

For instance, let us consider the following function that computes, by a conjugate gradient algorithm, the vector `x` such that `A*x=b`.

```
function [x,nb_iter]=grad(A,x,b,tol)
   r=b-A*x ;
   v=r ; c=norm(r)^2 ;
   for k=1:500
      res = norm(v) ;
      if (res<tol) , break , end
      z=A*v ; t=c/(v'*z) ;
      x=x+t*v ; r=r-t*z ;
      d=norm(r)^2 ; v=r+(d/c)*v ;
      c=d  ;
   end
   res = norm(v) ;
   nb_iter=k ;
```

The function accepts parameters of various shapes although it has been written for a matrix `A`, two column vectors `x` and `b` and a scalar `tol`. We can expect that the compiler is able to extract this information from the calling code. If not (or if the code is compiled to be linked with a C or Fortran program), it generates many useless cases. The user can help by in-

serting some directives to indicate to the compiler that the input parameters are matrices and columns different from scalars; We use the directives Smatrix (i.e., 2D and dimensions'sizes > 1) and Scolumn (1D with a size > 1).

```
function [x,nb_iter]=grad(A,x,b,tol)
%$PRAGMA A : Smatrix ;
%$PRAGMA b : Scolumn ;
%$PRAGMA x : Scolumn ;
r=b-A*x ;
... \\
```

4.3. 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 for MATLAB operators and functions,
(3) inserting data structure cast operations (i.e., data structure conversion).

The three problems are inter-dependent as choosing variables' implementation first influence the implementation of the functions and operators and vice versa. The cast insertion depends on the matches between the data structures and the operators implementation chosen.

Data structure choice, for a variable is performed first according to the uses of the variable and according to a default choice declared in the MTSD. For instance, declaring `matrix(real) = MatReal` indicates to use the MTSD class `MatReal` as the default implementation for the matrices of reals.

When generating the code for functions calls and MATLAB expressions not only the implementation is needed but also it is necessary to choose the type of accesses (i.e., a matrix/vector as a whole, elementary with row scanning, local, distributed, etc.) and the storage nature (i.e., if the code contains expressions that are not associated to memory addresses, if the variable is modified or not). These data are gathered in tuples (MTSD *class, storage, access, type*) call *code generation contexts*. For instance, in the MATLAB statement `A=B+1`, the nature of the subexpression `A` is *var* (for a modified variable); `B` is *const* (a non modified variable) and `B+1` is *expr* (an expression not stored in a variable). The compiler can associate the nature *var* to `B` if it can prove that `B` is not used after the statement. This allows the use of optimized codes that might modify their first parameters. For instance, if `B` is a matrix, its value can be incremented and its memory is associated to the matrix `A` using an efficient exchange of pointers.

Choosing the implementation of the functions and operators is performed in two steps:

(1) Possible choices are expanded according to the chosen variable implementation. For instance, several codes can be declared for the $*$ operator, see Program Code 2.
(2) "Cast" operations are inserted to convert "contexts" between statements. These casts are needed to ensure that the operators and functions parameters fulfill the requirements of their implementations. For instance, the following casts a real matrix of size 1×1 into a real scalar.

```
inline real @cast(const MatReal MAT)
                        "MAT.data[0]" ;
```

Assignments can also be inserted to transform expressions into variables (from the nature *expr* to the nature *const*):

```
% Scalar implementation of *
inline int @op_mult(expr int A,expr int B) "(A*B)" ;
inline @op_mult(out int C,expr int A,expr int B)
  => C=(A*B)" ;
inline real @op_mult(expr real A,expr real B) "(A*B)" ;
% Optimized implementation of * for diagonal matrices
inline @op_matmult(out DiagMatReal RES, const DiagMa-
tReal PARAM1,
      const DiagMatReal PARAM2,local IND)
   => alloc_Diag(RES,PARAM1.size) ;
   => for (IND=0;IND<PARAM1.size;IND++)
   => RES.diag[IND]=PARAM1.diag[IND]*PARAM2.diag[IND];
   ;
```

Program Code 2.

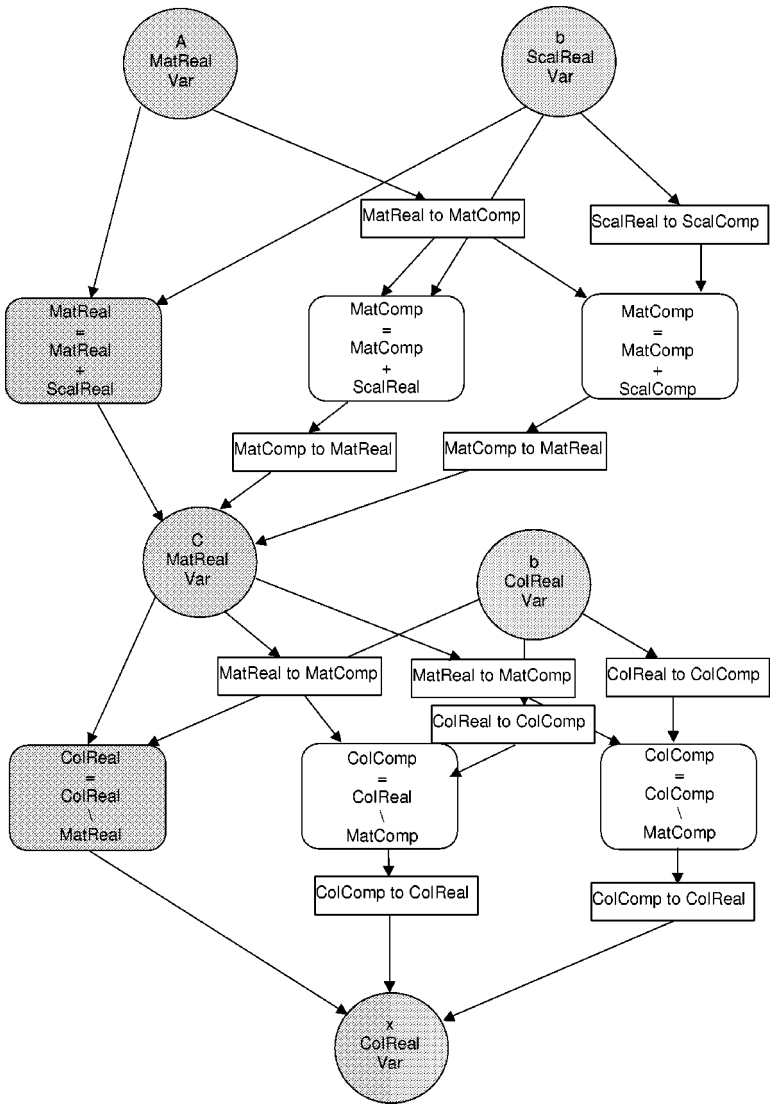

Fig. 3. Code generation graph for MATLAB code `C=A+b; x=C/d;`. `MatComp`, `MatReal`, `ColReal`, `ColComp`, and `ScalReal`, respectively denote complex matrices, float matrices, float column vectors, complex column vectors, and float scalars data structures.

```
% Scalar implementation of *
inline  @assign( int DEST,expr
                 int SRC)
   => DEST=SRC ;
```

These two steps are implemented by building *on the flight* a directed acyclic graph. Vertices correspond to variables, MATLAB operators/functions and data structure conversion subroutine. An example of such a graph is shown Fig. 3. Code generation consists then in selecting a set of paths that covers all program statements and variables (indicated in gray in Fig. 3). For more details the reader can refer to [2].

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 while the parallel codes are obtained using ScaLapack. The parallel codes are SPMD and the same data distribution (block-cyclic) is used for all matrices. Only a small number of directives (20 for all benchmarks) were inserted in the MATLAB codes. Because MENHIR relies on parallel libraries and conform op-

Table 2
MATLAB program code sizes and corresponding generated C code sizes (in lines).

Benchmark	Matlab code size	C code size
gauss	70	200
chol	180	620
pfl	200	9000
INT 1	140	4700
INT 4	200	3100

erations but not, yet, on MATLAB loops, program that does not exhibit coarse grain matrix operations were 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 sizes of the MATLAB programs and the generated codes are given in Table 2. Cloning, performed at step 3 of the compilation process, can substantially generate large target codes as shown for the program *pfl*.

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

Fig. 5 gives the execution times. 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 $n \times m$ are the execution times on the Onyx parallel computer ($n \times m$ is the size of the logical processor grid). As it can be seen, MENHIR performs better than 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. MCC uses the integer arithmetic provided by the architecture. This is more efficient but some overflow errors can appear.

In general, the code produced by MENHIR is close, for most of the benchmarks to the "hand coded" versions of the algorithms. However, these "hand coded" versions were not aggressively tuned.

Parallel codes reach good speedups thanks to the ScaLapack library but also to the type analysis. Indeed,

```
...
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. 4. Main loop of the Jacobi benchmark.

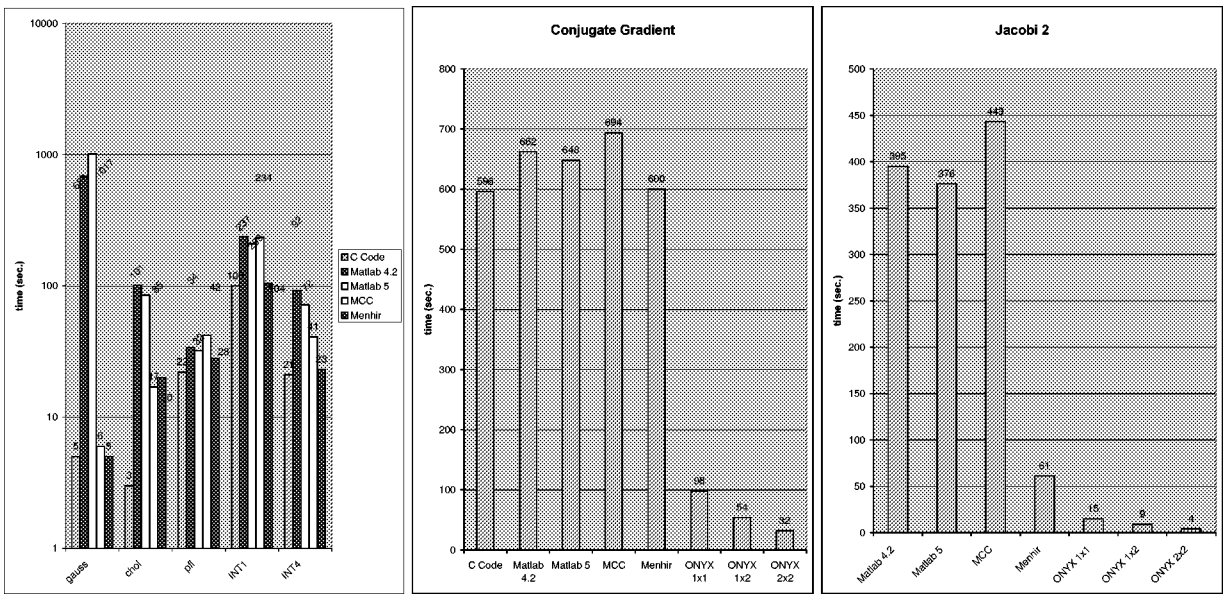

Fig. 5. Execution time in seconds of the sequential and parallel generated C codes.

in the case of Jacobi, good performance is obtained by propagating the information that the preconditioning matrix M, shown Fig. 4, is diagonal, avoiding the run-time checking of this property.

6. Conclusion

In this paper we briefly presented MENHIR, a compiler for MATLAB. The strength of MENHIR is its original target system description that allows to generate 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.

References

[1] A. Aho, R. Sethi and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addisson-Wesley, 1985.

[2] S. Chauveau, MENHIR. Un environnement pour l'exécution efficace des codes Matlab (in French), PhD thesis, Université de Rennes 1, February 1998.

[3] 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.)

[4] J.W. Eaton, ftp.che.utexas.edu/pub/octave.

[5] B.A. Marsolf, Techniches for the interactive development of numerical linear algebra libraries for scientific computation, PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[6] The Mathworks, Inc., *MATLAB, High-Performance Numeric Computation and Visualization Software. Reference Guide*, August 1992.

[7] S. Ramaswamy, E.W. Hodges and P. Banerjee, Polaris: A new generation parallelizing compiler for MPP, year = 1994 compiling MATLAB programs to ScaLAPACK: Exploiting task and data parallelism, in: *IEEE Symposium on Parallel and Distributed processing*, 1996, pp. 613–619.

[8] M. Rezny, MATHSERVER: A client-server approach to scientific computation, Department of Mathematics, The University of Queensland, Australia.

[9] L. DeRose, Compiler techniques for MATLAB programs, PhD thesis, University of Illinois at Urbana-Champain, 1996.

[10] Scilab, http://www-rocq.inria.fr/scilab.

[11] A.E. Trefethen, V.S. Menon, C.-C. Chang, G.J. Czajkowski, C. Myers and L.N. Trefethen, MultiMATLAB: MATLAB on multiple processors, Technical Report 239, Cornel Theory Center, 1996.