Speed-up of FEM simulation for granular flow via optimised Numerical Linear Algebra software

M. Köster, F. Platte, S. Turek

Institute of Applied Mathematics, University of Dortmund
Vogelpothsweg 87, 44227 Dortmund, Germany

michael.koester@mathematik.uni-dortmund.de
frank.platte@mathematik.uni-dortmund.de
stefan.turek@mathematik.uni-dortmund.de

F. Neumann, G. Rombach

Concrete Structures, Technical University of Hamburg
Denickestr. 17, 21073 Hamburg, Germany

f.neumann@tu-hamburg.de
rombach@tu-hamburg.de

Internal Report – September 2004

Abstract

In this paper we will give a short introduction to the installation and efficiency of state-of-the-art direct solver packages like CROUT and UMFPACK for linear systems, combined with highly processor optimised BLAS libraries. We will give an overview about the methods that can be used to integrate these libraries into Fortran programs compiled with the *Compaq® Visual Fortran* or the *Intel® Fortran Compiler* using Microsoft® Windows. In numerical examples arising from the numerical simulation software *SILO* we will show that these packages provide a very effective way of solving medium-size (≈ 30.000 unknowns) ill-conditioned problems as long as a highly optimised BLAS library performs the low level linear algebra operations.

1 Introduction

Most of the mathematical software for performing numerical simulation are nowadays designed to be used under UNIX or LINUX operating system environments – either due to historical reasons or due to restrictions in current Microsoft® Windows based operating systems (memory, availability of compilers). This makes it impossible to port such applications directly to Windows – unless a UNIX emulator like CYGWIN is used (cf. [10]).

The build process for libraries and Fortran programs is totally based on "makefiles" which have to be designed by the author of the library/program. In contrast the most commonly used compiler environment under Windows is the *Microsoft Visual Studio*, which provides a graphical IDE to the user to simplify application development. Both the *Compaq® Visual Fortran* and the *Intel® Fortran Compiler* base on this technique, which allows the user to build applications very comfortably without any text based "makefile".

Unfortunately both techniques are incompatible. For sure even the *Microsoft Visual Studio* allows using "makefiles", but the syntax is totally different from those used in a UNIX-like environment. On the other hand pure Fortran libraries like CROUT or UMFPACK2 can be integrated very easily into a Fortran application as they contain no special features in the build process: The library can easily be assembled and compiled inside the IDE by drag&drop. Far more critical is the use of the modern high speed solver package UMFPACK4, which is written in the programming language C. To integrate this package into a Fortran application is not such an easy task and requires exact matching compiler settings for both, the C compiler and the Fortran compiler, for every library and the project itself. In this paper we will describe which compiler settings are critical during the integration process and how they must be set to generate an application.

To maximize the efficiency of these mathematical solvers the correct settings for the compilers are only the first step. Far more critical is the use of a highly processor optimised BLAS library. To integrate such libraries into an application can be as hard as integrating C code into a Fortran application: The *Compaq® Visual Fortran* compiler for example does not contain an optimised BLAS library that can be used when parts of the program are written in C. Intel has done a much better job here with its MKL-library (cf. [14]), but their library is not freely available. On the other hand, there are very effective BLAS libraries freely available in the Internet (GOTO BLAS [12], ATLAS [9]) which can be used under Windows, too, but their integration is not trivial. Nevertheless the effort for integrating them is worthwhile: In our tests the use of the GOTO BLAS with a Pentium 4 processor using UMFPACK4 can speed up the solving process up to 700% in comparison to using the standard CROUT solver without any BLAS library!

In the following sections we will give a comprehensive overview how to compile and integrate CROUT, UMFPACK2 and UMFPACK4 into a small example application. This application is written in pure Fortran 90. It reads a matrix from a text file and starts the solver for a given linear system. In numerical examples we will show the effectiveness and necessity of the different components – compiler, solver and BLAS library – by comparing the absolute time these configurations need to solve the given system.

To get an impression of the efficiency of these components in real life applications our attention will later turn on their integration into the simulation software *SILO*. This Finite Element software has been developed to simulate the behaviour of granular material in silos during filling and during flow and to estimate the resultant wall and bottom pressures; further details are given in chapter 6.2. Currently, the solver for the upcoming linear systems is based on a CROUT implementation. At first we will replace CROUT by an implementation of

UMFPACK2 together with an optimised BLAS library. In numerical test examples we will show that already this step will boost up computation time. The replacement by UMFPACK4, which is currently under development, promises a further improvement as our numerical test examples for the solution of linear systems indicate.

General acknowledgements

In this paper we will mainly use the *Intel® Fortran Compiler 8.0* for demonstration of the different compiler options because it is the most common Fortran Compiler and it uses Microsoft's current implementation of the *Visual Studio .NET 2003* as IDE. The use of the *Compaq® Visual Fortran* compiler is only slightly different, since it makes also use of *Microsoft® Visual Studio 6.0* as IDE. Fortunately the names of the important compiler options are very similar and so it should be easy to find them. So the use of the *Compaq® Visual Fortran* is basically the same, important differences to the Intel compiler are emphasized where necessary.

Our test program is written in Fortran 90. Although it was originally developed in a UNIX environment, the porting to Windows was straightforward. In the IDE a new Fortran Console project was created and all necessary .f90-files were added to it. Afterwards the program was able to be compiled, but yet it could not be linked as some important routines were missing: The BLAS routines for basic linear algebra processing and the UMFPACK solvers. In UNIX these libraries are shipped with an installation tool which generates library files for direct use. Under Windows such installation tools are missing, and so we have to compile and integrate these libraries by hand.

2 The integration of CROUT, UMFPACK2 and a basic BLAS implementation

2.1 A CROUT implementation

The basic integration of CROUT and UMFPACK2 is rather easy because both solvers are written in pure Fortran. We start with a basic description of CROUT:

The implementation of the CROUT algorithm can be kept very shortly. Our implementation uses the SKYLINE format for storing sparse matrices[1]. In this case the algorithm itself consists of exact two subroutines:

- One subroutine which performs a factorization to compute the LU decomposition of a given matrix and
- one subroutine which uses the LU decomposition to compute a solution to the given system.

For such a short implementation it is not even necessary to build a library. For our test program we simply put these routines in a file called *CROUT.f90* as part of the test program. There are no additional compiler options to be set (cf. Figure 1).

[1] A documentation of this format can e.g. be found in the SPARSEKIT library, cf. [16].

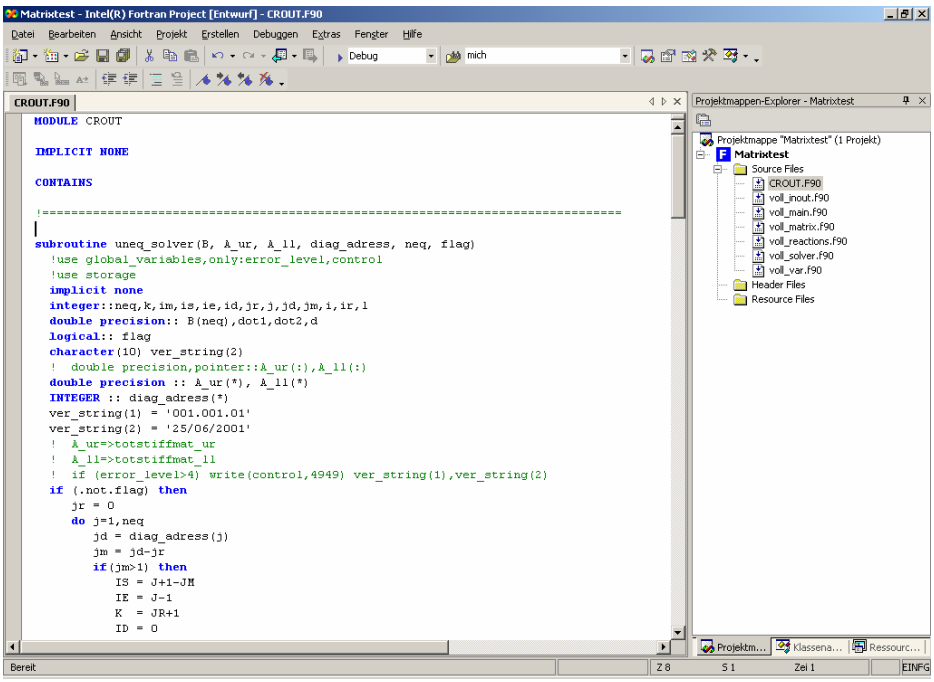


Figure 1: The SKYLINE based implementation of CROUT consists of only two Fortran subroutines and can be included directly without creating a library of it.

2.2 An UMFPACK2 implementation

Similar to the CROUT algorithm also UMFPACK2 consists of pure Fortran code. But because the code is much larger, it is worthwhile to build a library which can later be linked to an application. The UMFPACK library (in our case we used UMFPACK 2.2.1, which is available on http://www.cise.ufl.edu/research/sparse/umfpack/old.html) is distributed as a *.tar.gz* archive and can be decompressed with a standard ZIP-compatible decompressor like WinRAR or WinZIP. Unfortunately, the standard archive does not contain two files that are necessary for a successful compilation of the whole library:

- mc13e.f
- mc21b.f

These two files are part of the *HARWELL* library for the solutions of sparse linear equations. The UMFPACK2 archive does only contain empty links to these files which might produce errors while decompressing. These errors can be ignored. The files themselves can be obtained by the internet, e.g. http://scicomp.ewha.ac.kr/netlib/harwell/.

After obtaining these files extract all *um*.f* files of the UMFPACK2 library into one directory. Copy the two *mc*.f* files from above into that directory, too. In the *Visual Studio* IDE create now a new *Fortran Static Library Project* in the folder that contains the sourcecode and add all **.f-files* to that project (cf. Figure 2). Afterwards the library can be compiled.

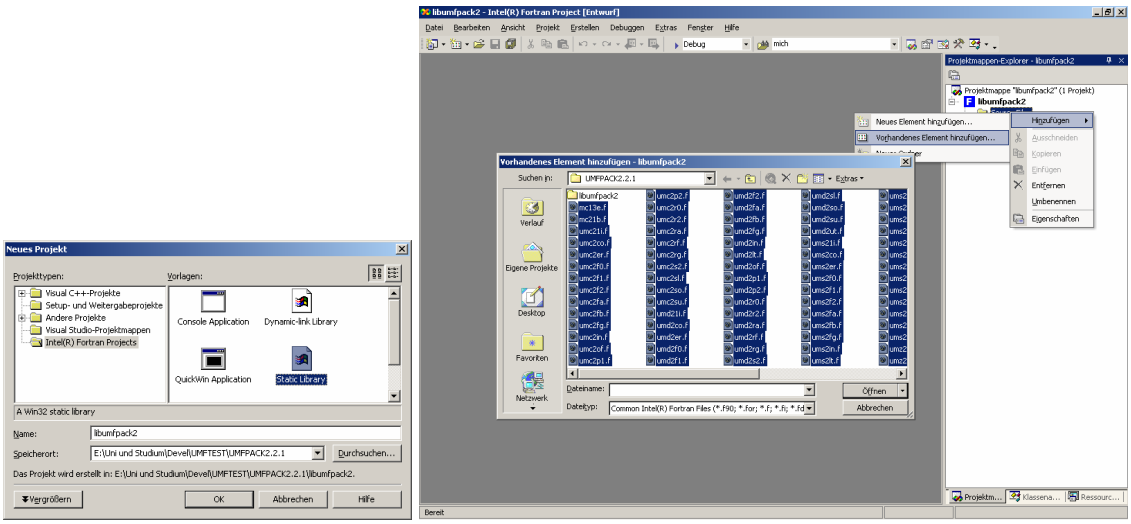


Figure 2: Creation of a new Fortran library for UMFPACK2

2.3 A basic Fortran compatible BLAS library

Before we show how to integrate a high-speed processor optimised BLAS library into an application we first build a standard Fortran compatible BLAS library. Later in our numerical tests, we will compare the results for both types of BLAS to show the enormous potential that lies in the optimised ones.

A standard implementation of the BLAS can be obtained from the internet. In our case, we choose the reference implementation available from http://www.netlib.org/blas written in Fortran 77. The *blas.tgz* archive itself only contains a set of *.f* Fortran files, so the generation of a library is similar to the creation of the UMFPACK2 library as shown in Figure 2:

1. Extract all files into a new directory, e.g. *BLAS*.
2. In the Visual Studio IDE create a new static library project in that directory, e.g. named *libblas*.
3. Add all **.f* files to the project and compile the library.

2.4 Integration of the libraries into an application

We now consider our main example program and include the libraries to it. There are two ways for integrating them

1. In the compiling process the compiler has generated **.lib* files in the appropriate directory of a library. In the properties of the main project there is a page with options for linker. Here the path/filename of each library can be added to include it to a project.
2. Use the concept of sub-projects and dependencies.

In our case, we choose the second one. The reason is the following: Later we will include UMFPACK4 and performance BLAS libraries into the project. This will require some more changes to the project configuration of *all* libraries in the project, and so using sub-projects will simplify the process of recompilation. To include both – UMFPACK2 and BLAS – to our project we add them as existing sub-projects to our workspace. Afterwards, we install dependencies of our example program to these two libraries. This will ensure that the linker automatically links both of the libraries to our project (cf. Figure 3).

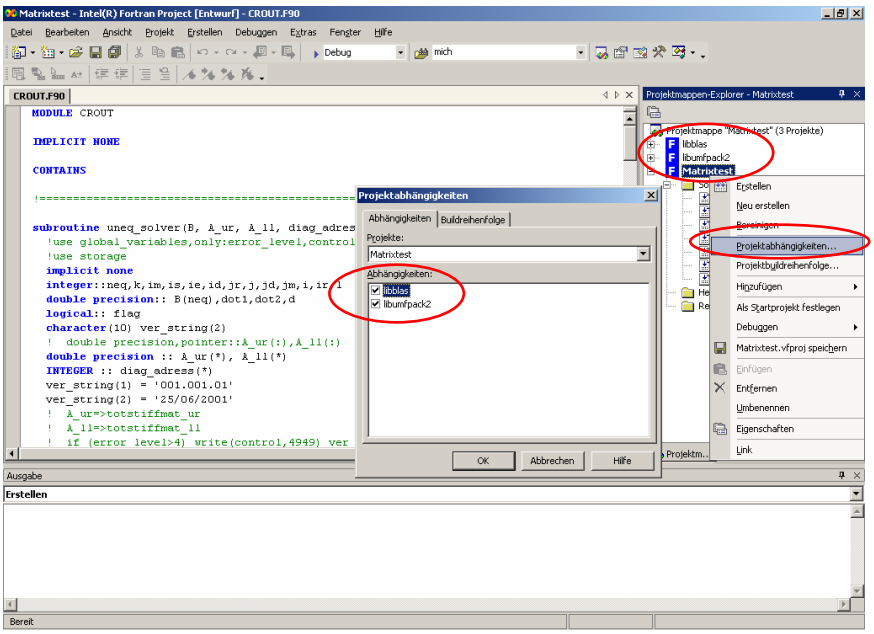


Figure 3: The BLAS and UMFPACK2 libraries are used as sub-projects. By using project dependencies both libraries are made dependent to the test application and so they are automatically linked to it.

The CROUT solver (which was already part of the main project as it is contained in only one file) as well as the UMFPACK2 solver is now available to the main project. Using the appropriate calls to the solver subroutines will start the solver and produce a solution to a given linear system.

3 Preparations for foreign libraries

The above procedures were conducted to produce a basic application for solving linear systems. Yet, there were no optimisations implemented (not even the compiler is tuned for optimisation), so the resulting application will run comparably slow. A first improvement could result in compiling the whole application in *Release* mode instead of *Debug* mode, but in our case this will be the last step. In this section we will first introduce the necessary changes in the compiler options to make it possible to use processor-optimised performance libraries and to correctly include the UMFPACK4 library. In later sections we will then describe in more detail how to include them into an application.

The main problem in including a performance BLAS and UMFPACK4 resp. is the mixture of two different compiler languages. Our main application is written in Fortran 90. UMFPACK4 is written in ANSI-C and a typical performance BLAS that can be obtained precompiled from the internet is written either in ANSI-C (ATLAS) or in assembler language with an ANSI-C interface.

To prepare their use, here we have to modify the properties of our main project in the following way (cf. Figure 4), to be more precise the way external procedures are handled by the Fortran compiler. The following parameters have to be set:

Name	Value
Calling convention	C, Reference
String Length Argument Passing	After all Arguments
Name case interpretation	Upper case
Append underscore to External Names	No

The last two parameters („Name case...“, „Append underscore...“) are compiler default for both the *Intel Fortran compiler* and the *Compaq Fortran compiler*. The calling convention has to be changed for both compilers to allow using C-code. In contrast to this, the *Intel* and the *Compaq* compiler differ in the way of handling „String length argument passing“: While the *Intel* compiler uses a default value of „After all Arguments“, the *Compaq* compiler uses a default of „After Individual String Argument“. This is incompatible with C programs and therefore has to be changed as mentioned above.

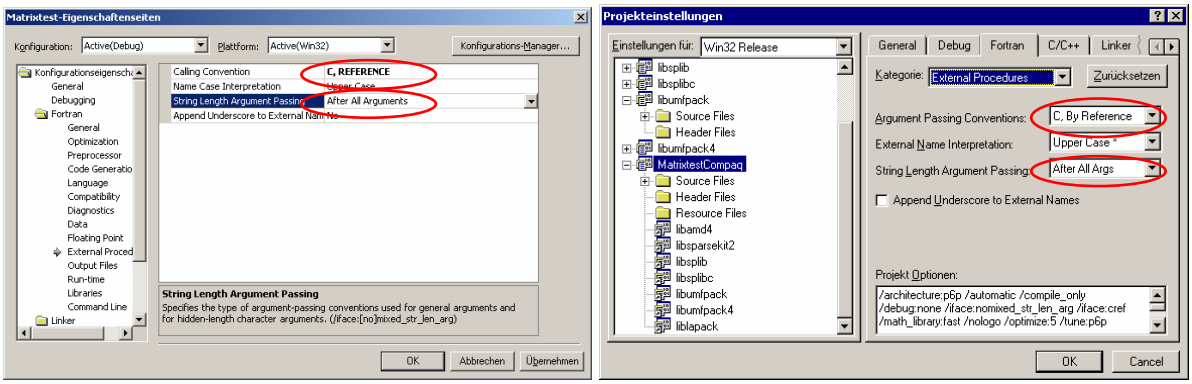


Figure 4: Defining the correct calling conventions for later use of BLAS and UMFPACK4

The main important thing of this change is the following:

> The calling convention as described above must not only be changed for the main project, but also for all Fortran libraries (the sub-projects) and all project configurations (*Release*, *Debug*) in this workspace as well!

In our case, this means that this calling convention has also be changed for the UMFPACK2 library and the BLAS library, too (as long as we use this reference BLAS). If other Fortran libraries are used by the main application (like SPARSEKIT and LAPACK, SPLIB,... resp.) the calling convention for these must be changed the same way!

4 Compiling UMFPACK 4.3

The procedure for compiling UMFPACK4 is far more complex than for UMFPACK2. After decompressing the *.tar.gz* archive UMFPACK4 appears as a set of two separate sub-libraries which both have to be compiled and to be included into a project:

- the AMD-library which performs ordering/permutation of sparse matrices and
- the main UMFPACK4-library which solves a linear system by symbolical and numerical factorization

While the creation of an AMD-library is comparably straightforward, the compilation of the UMFPACK4 library itself needs more effort. Although the library is written in ANSI-C, some of the files have to be compiled more than once with different compiler options. Furthermore, it is necessary to link F77-wrapper routines into the library (which are fortunately contained in the UMFPACK package) to make it possible to use UMFPACK4 in a Fortran application.

4.1 Compiling the AMD library

The AMD library is a standard C-library that can be compiled and linked in the usual way. To be more precise, the following steps have to be done to create it with *Visual Studio .NET* (the procedure for creating it with Visual Studio 6.0 is similar):

1) Create a new empty .NET project in the AMD library folder, e.g. named “libamd”.

2) Add the **.c* files of the *Source* subdirectory to the *Source files* folder if the *Visual Studio IDE*. Add the **.h* files of the *Source* and the *Include* folder to the *Header files* folder of the Visual Studio IDE.

3) Edit the project configuration of the *libamd* project for both *Debug* and *Release* configuration to match the following options:

   a) Project type: Static library (.lib)

   b) Use Managed Extensions for C++: No

   c) Runtime library: Single-Threaded (Single Threaded-Debug resp.)

   d) Add the path to the *Include* folder to the list of additional include folders of the C/C++ compiler. Otherwise the *amd.h* header file could not be found in the compiling process.

Afterwards the library can be compiled. The **.f* Fortran files in the *Source* folder of the library can be ignored since the library is only used by UMFPACK which is written in C.

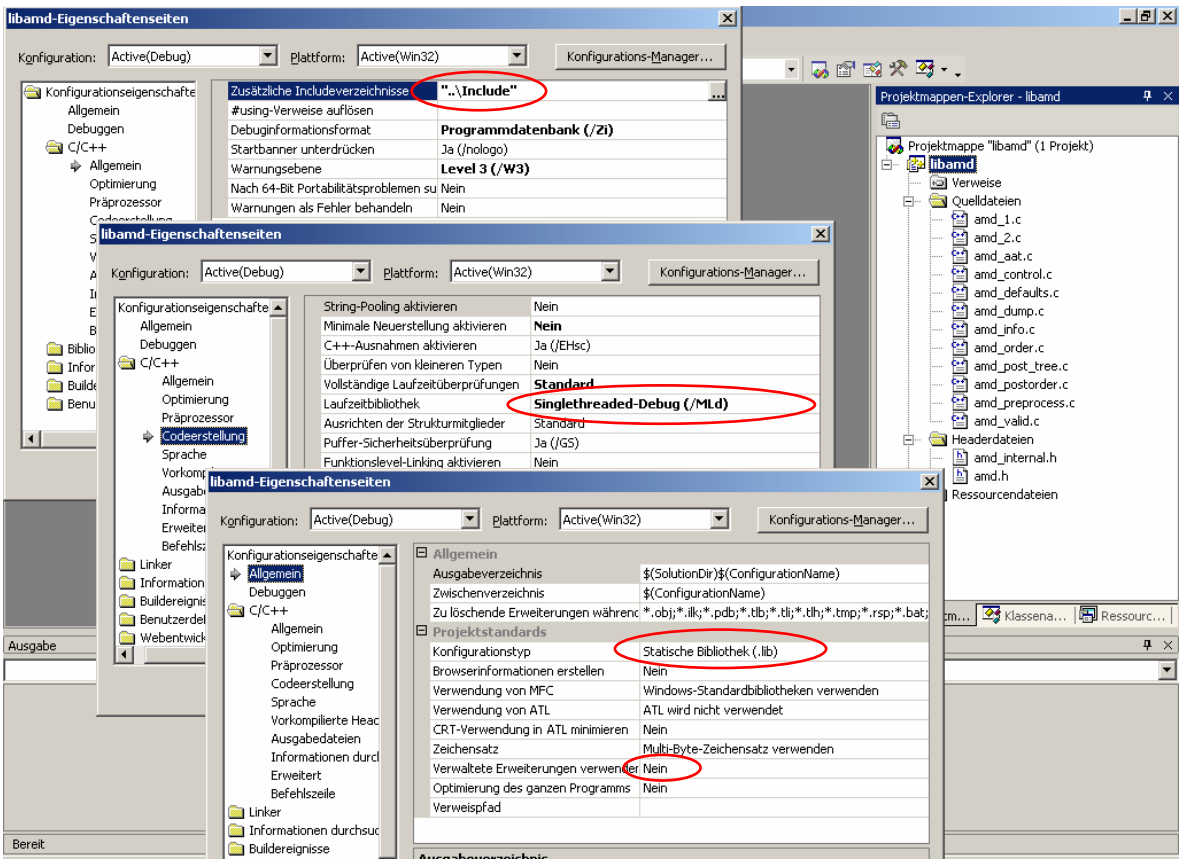


Figure 5: Preparation of the AMD library.

4.2 Compiling the UMFPACK4 library

In this section we want to describe the steps that are necessary for a successful build of the UMFPACK4 library. Because UMFPACK4 is basically written in C the first steps of creating the library are the same as for the AMD library above:

1) Create a new empty .NET project in the UMFPACK library folder, e.g. named "libumfpack4".
2) Add the **.c* files of the *Source* subdirectory to the *Source files* folder if the *Visual Studio IDE*.
3) Add the **.h* files of the *Source* and the *Include* folder to the *Header files* folder of the Visual Studio IDE.
4) Edit the project configuration of the *libamd* project for both *Debug* and *Release* configuration to match the following options:
   a) Project type: Static library (.lib)
   b) Use Managed Extensions for C++: No
   c) Runtime library: Single-Threaded (Single Threaded-Debug resp.)
   d) Add the path to the *Include* folder to the list of additional include folders of the C/C++ compiler.
   e) Add the path to the *AMD\Include* folder and the path to the *AMD\Source* folder to the list of additional include folders of the C/C++ compiler. UMFPACK4 uses AMD and therefore it needs access to the header files *amd.h* and *amd_internal.h* during compilation.

Although the library can now theoretically be compiled there are some more important changes necessary to make sure that it can be linked to a Fortran application properly (again the changes have to be done for the *Debug* as well as the *Release* configuration!):

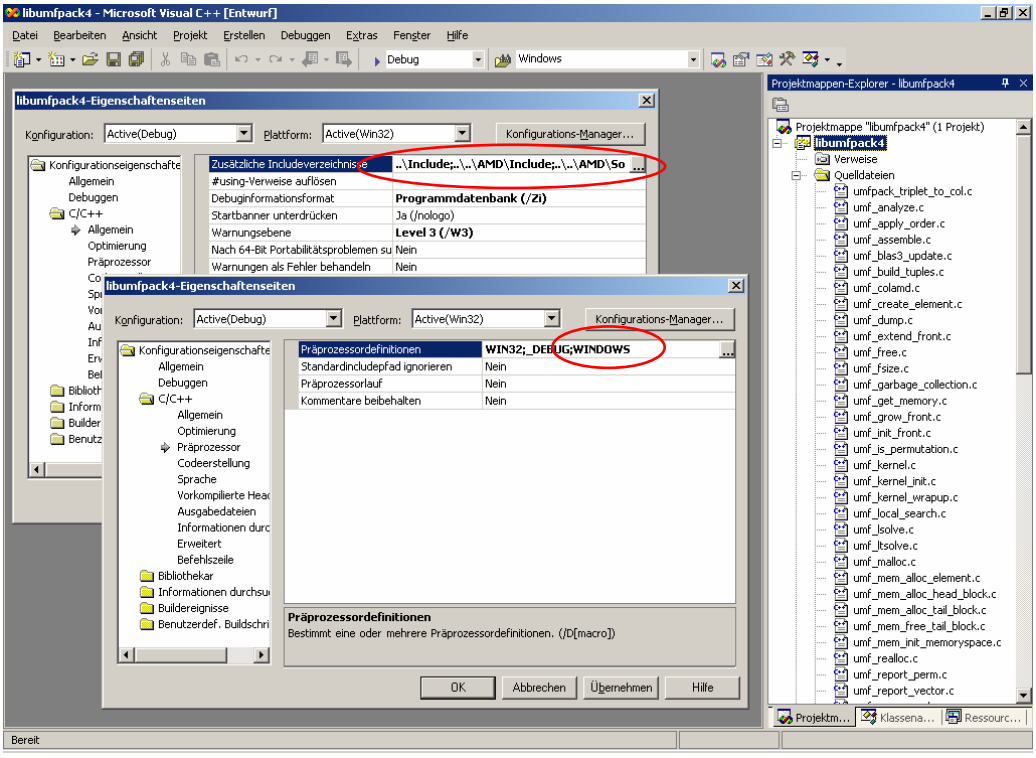


Figure 6: Preparation of UMFPACK4: Additional include paths and preprocessor definitions.

5) Add a preprocessor definition *WINDOWS* to the list of preprocessor definitions of the project.

6) Add the file *umf4_f77wrapper.c* from the *DEMO* folder to the list of source files in the Visual Studio IDE. This is the wrapper interface which connects Fortran and C.

7) Our main Fortran application requires upper case function names without any underscore. Unfortunately, the standard wrapper interface does not support this, so the file *umf4_f77wrapper.c* has to be edited manually: All procedure names "umf4xxxx" in this file have to converted to uppercase.

8) UMFPACK4 uses BLAS. The standard BLAS library we have compiled earlier is a Fortran library, so all procedure names are uppercase. UMFPACK4 in contrast expects lowercase characters, therefore we have to tell UMFPACK4 to use uppercase procedure names for BLAS subroutines. Open the file *umf_config.h* and search for the define *BLAS_NO_UNDERSCORE*. It appears multiple times in this file. Depending on this define there are the names of the *BLAS* functions declared that are to be used by UMFPACK4. Find those definitions (BLAS_GEMM_ROUTINE, BLAS_TRSM_ROUTINE, ...) and convert the names of the *BLAS* functions to uppercase (cf. Figure 8).

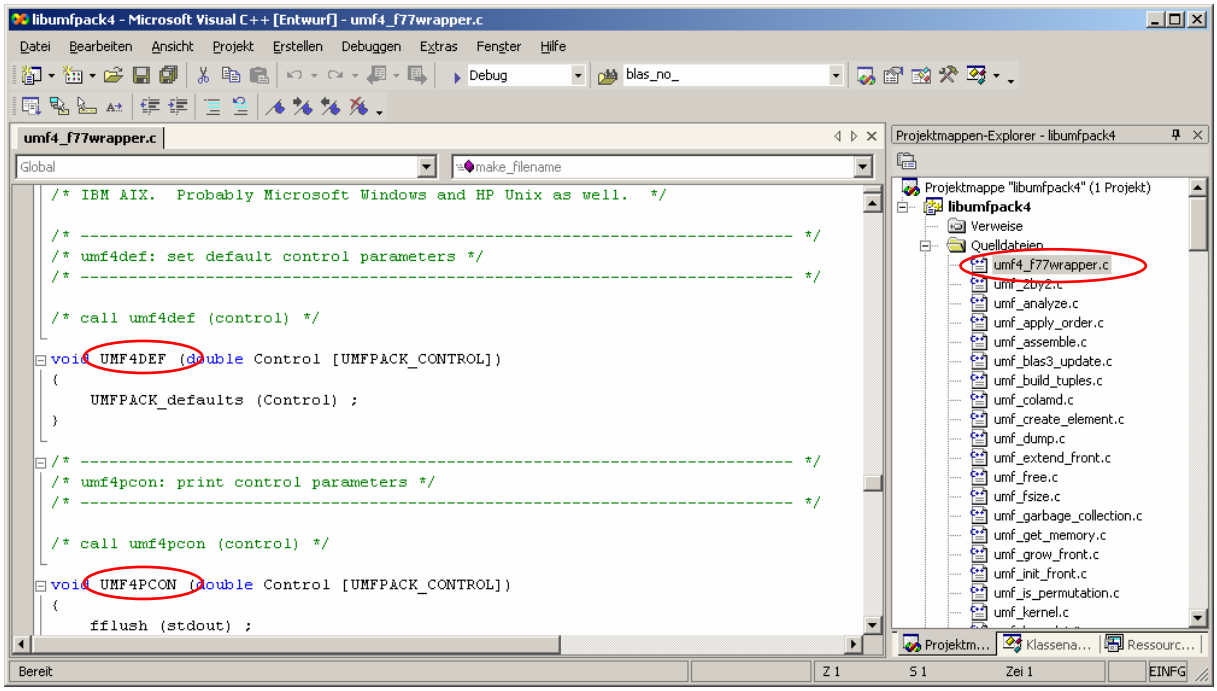


Figure 7: Preparing the Fortran wrapper. Convert all procedure names in the wrapper file to uppercase manually.

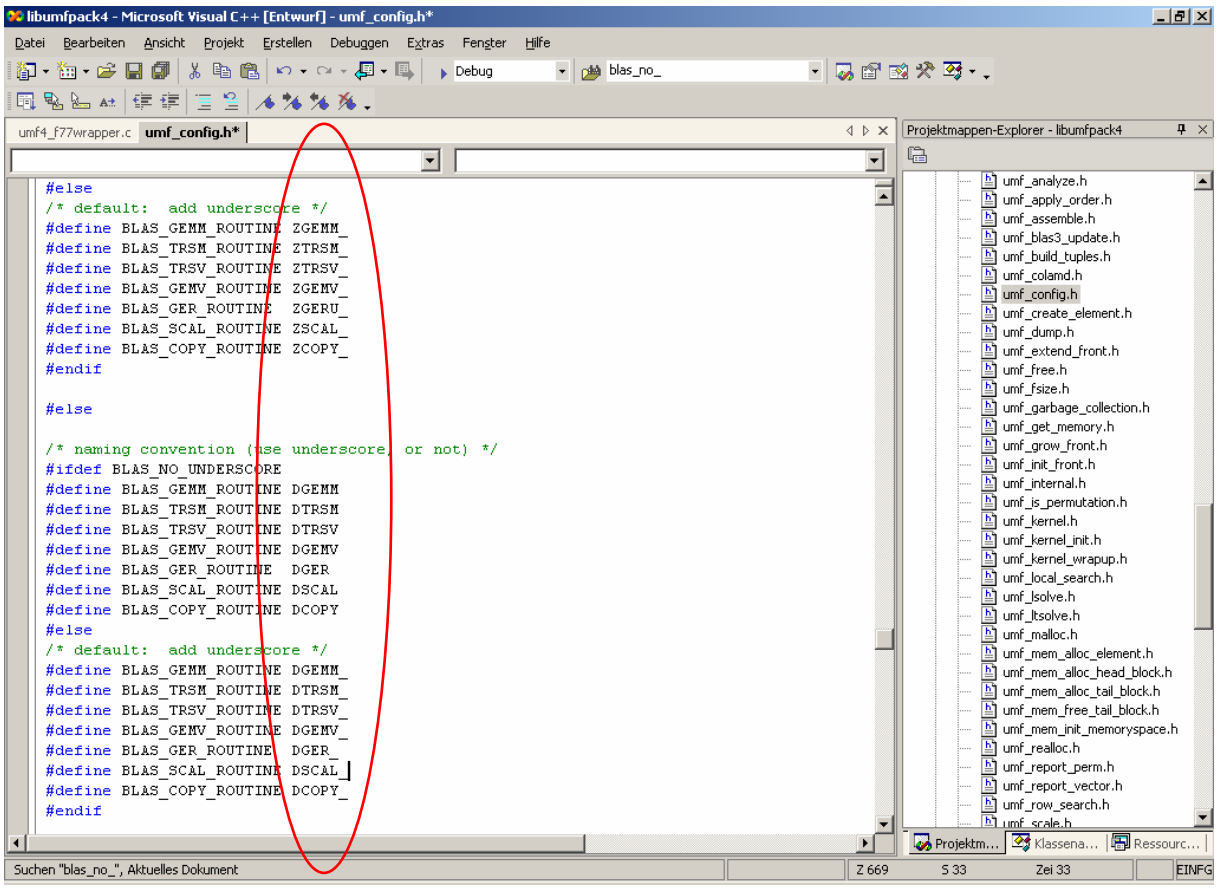


Figure 8: Prepare UMFPACK4 to use Fortran BLAS functions. Convert names of BLAS subroutines to uppercase.

After these steps the UMFPACK4 library can already be compiled and linked into our test program, although it is not complete yet: In its current condition the library supports real number but not complex ones. If complex numbers are required, the author suggests that some files of the library are compiled more than once with different compiler options (more precisely: different preprocessor-*defines*). Unfortunately the Visual Studio IDE does not support compiling one file more than once directly, so we have to do some additional changes to the code:

Simply create some additional files umf_multicompile_1.c, ..., umf_multicompile_8.c to the sourcecode. Every file has to contain only 2-3 lines making some defines and including another file. The content of these files can be seen in table 1 (cf. Figure 9).

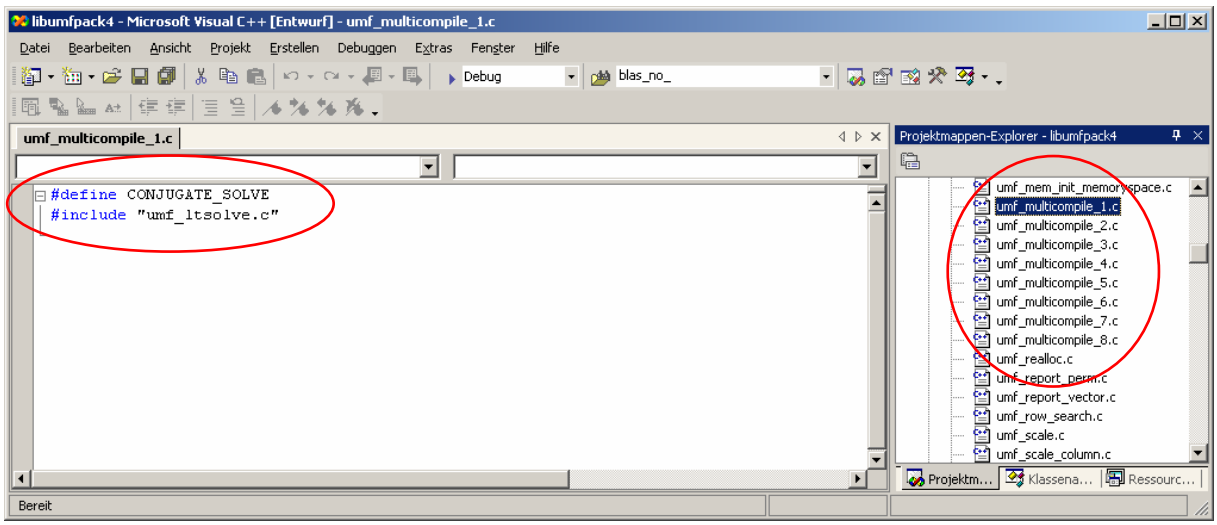


Figure 9: To support complex numbers some files must be compiles more than once.

Filename	Content
umf_multicompile_1.c	`#define CONJUGATE_SOLVE` `#include "umf_ltsolve.c"`
umf_multicompile_2.c	`#define CONJUGATE_SOLVE` `#include "umf_utsolve.c"`
umf_multicompile_3.c	`#define DO_MAP` `#include "umf_triplet.c"`

umf_multicompile_4.c	`#define DO_MAP`
	`#define DO_VALUES`
	`#include "umf_triplet.c"`

umf_multicompile_5.c	`#define DO_VALUES` `#include "umf_triplet.c"`
umf_multicompile_6.c	`#define FIXQ` `#include "umf_assemble.c"`
umf_multicompile_7.c	`#define DROP` `#include "umf_store_lu.c"`
umf_multicompile_8.c	`#define WSOLVE` `#include "umfpack_solve.c"`

table 1: Additional files for the UMFPACK4 library to support complex numbers.

After these changes the library can be compiled and linked. There will be a lot of warning messages like “unknown pragma”, “DGEMM undefined” (these are BLAS routines that are yet unknown) or warning about conversion of numbers. These messages can be ignored.

To integrate the UMFPACK4 library into a project we again suggest to add *libumfpack4* and *libamd* as subprojects and install dependencies like we did with BLAS and UMFPACK2 in section 2.4 (cf. Figure 10).

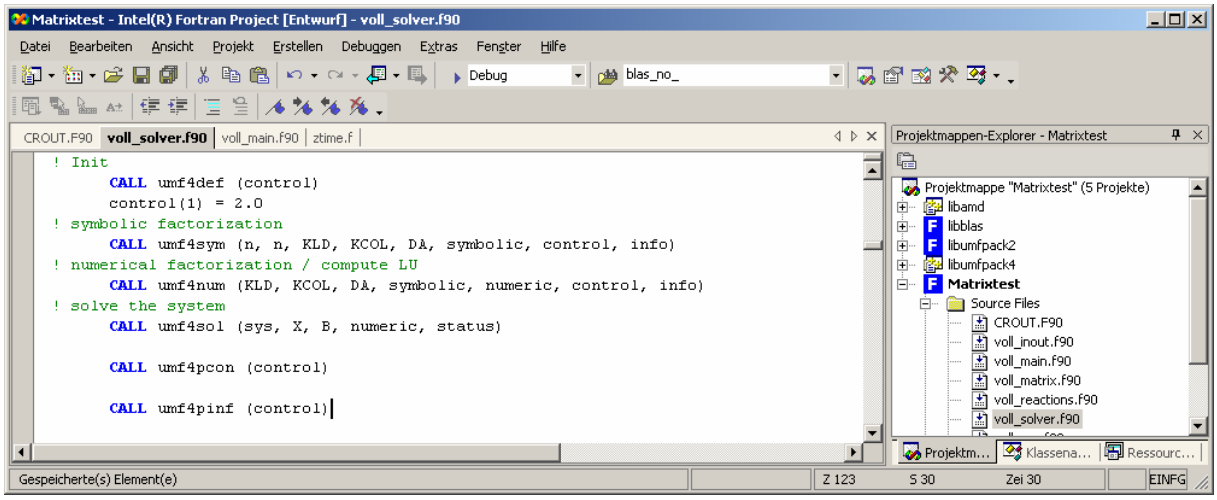


Figure 10: *libamd* and *libumfpack4* are included as depending subprojects to the main application. The libraries can be used in Fortran with the help of the wrapper functions that were defined in *umf4_f77wrapper.c*.

5 The key to performance: optimised BLAS libraries

In this section we will introduce some hardware-optimised BLAS libraries and the necessary steps to include them in a typical Fortran test application. Such libraries are typically written in C or directly in assembler language with a C calling convention because it is simply not possible to do by-hand-optimisation in Fortran. Therefore any Fortran application that wants to use such libraries has to be prepared to work together with C-like code. In the last sections we already described the steps that are necessary to prepare an application for this: The calling convention has to be "C, Reference" and the name case interpretation has to be "After all Arguments". Our test application is already prepared for this, but it still uses our standard *BLAS* routines.

To switch an application from standard-*BLAS* to hardware-optimised *BLAS*, three steps have to be performed (cf. Figure 11):

1. Download/generate a hardware-optimised *BLAS* as static library (*.lib*-file). Often such libraries can be downloaded from the internet for many processors, but sometimes (like in the case of ATLAS) they have to be generated manually. The resulting *.lib* file should be copied to the project folder.
2. If the standard-*BLAS* is used as a subproject (like in our test-application), deactivate the dependency of the main application to this *BLAS*.
3. In the properties of the *Visual Studio* linker the *.lib*-file of that library has to be added to the list of *additional dependencies* / *additional libraries*. If the library is not located in the same directory as the application, the path to the *.lib* file has to be specified in the "Additional Library Directories" option of the linker.

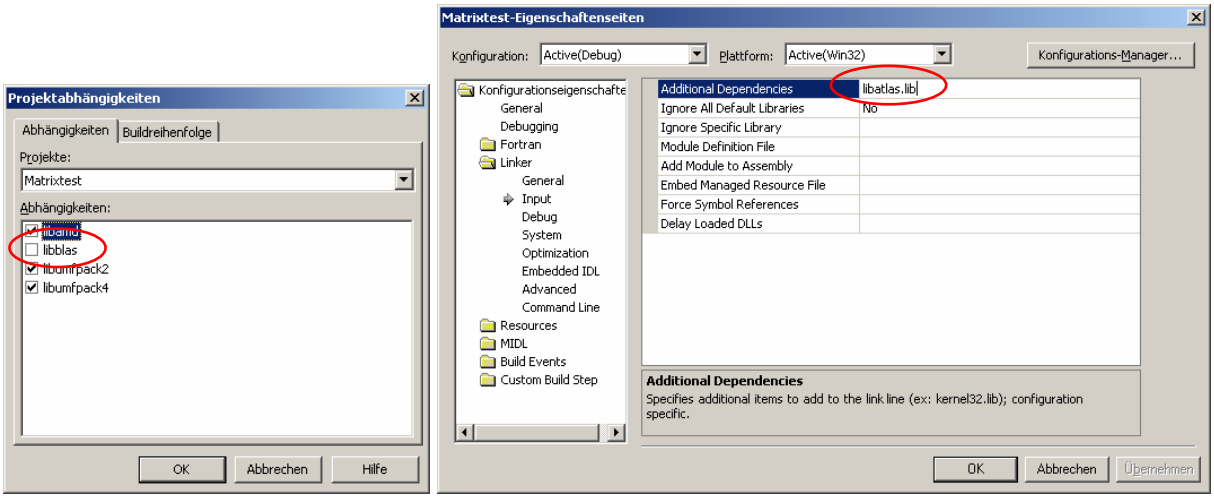


Figure 11: Switch to a hardware-optimised BLAS: Deactivate the dependency to standard-BLAS, add the name of the new BLAS library to the list of additional libraries.

Remember to do this especially for the *Release* configuration of the project. Afterwards it is enough to perform a re-link of the project.

We will now introduce some hardware-optimised *BLAS* libraries. Most of them can be downloaded from the internet free of charge.

5.1 Compaq Extended Math Library (CXML)

The Compaq CXML library is the internal math library of the Compaq Fortran Compiler. This library can be used in pure Fortran programs very well. Unfortunately, this library does not support C calling convention, what makes it impossible to use it in conjunction with UMFPACK4. Therefore we skip further investigations on this library.

5.2 Intel Math-Kernel-Library (MKL)

The MKL library of Intel is a library specially designed for Intel Pentium® compatible processors. It can be used with the Intel Fortran Compiler and supports both, standard Fortran and C calling convention. The library can be obtained as a test version directly from the Intel home page. After installation the library can be linked to any application by

- specifying the path to the library in the property page of the linker (e.g. "C:\Programme\Intel\MKL70\ia32\lib")
- specifying the name of the library that is using C calling convention in the list of additional libraries as described above. For example in version 8.0 of the Intel compiler the appropriate *.lib*-file is called *mkl_c.lib*.

5.3 GOTO-BLAS

This BLAS library is written by Kazushige Goto and is available at http://www.cs.utexas.edu/users/flame/goto/. It is mainly optimised for Intel Pentium®-based processors, but the latest development is an AMD Opteron® compatible BLAS. The library itself is shipped as a ZIP file containing two files, a *.lib*/*.dll* pair:

- *libgoto_xxxx.dll*
  This is the main library file containing the mathematical routines.
- *libgoto_xxxx.lib*
  This is a wrapper file for the *.dll* to use it like a static library.

To use this library both files have to be copied into the application directory. The *.lib* file has to be specified to the list of additional libraries as above. When the application is started the *.dll* file is loaded into memory by the wrapper routines, therefore this file must be available on runtime.

Remark: One additional step has to be done before linking. The author suggests to specify the parameter "/stack:0x10000,0x10000" to the linker. In the case of the *Visual Studio* this means that the *Stack reserve size* and the *Stack commit size* parameters of the linker in the properties of the project both have to be set to *65536* (cf. Figure 12).

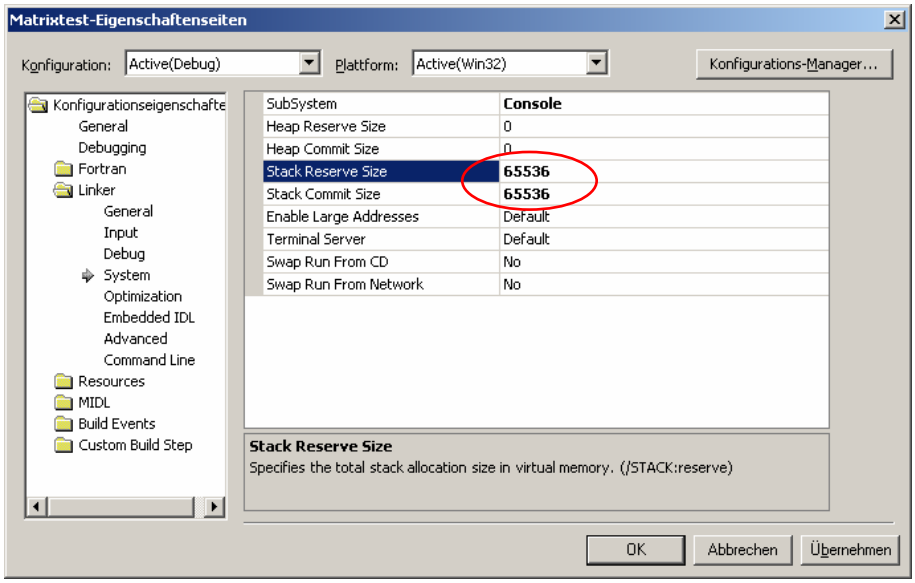


Figure 12: When using the GOTO BLAS library the stack parameters have to be set properly.

5.4 ATLAS-BLAS

The ATLAS BLAS is a self-optimising BLAS library written in ANSI-C. This allows it to adapt it to processors where a hardware-optimised BLAS is not available. For example the GOTO BLAS library is optimised mainly for Pentium-based processors – although it can also be used for Athlon-based processors this does not develop full performance. ATLAS is an alternative: It can be compiled directly for the target processor and develops sub-optimal performance, but a number of precompiled ATLAS libraries are available, too. We will give a short introduction here how to compile it for a Windows-based system. A more detailed description can be found via the ATLAS homepage (http://math-atlas.sourceforge.net/errata.html).

To compile the ATLAS BLAS it is necessary to use the GCC compiler and the MINGW package of the UNIX emulator CYGWIN (http://www.cygwin.com). The current version of ATLAS can be obtained at http://math-atlas.sourceforge.net/ as a *.tar.gz* file. After decompression with *gunzip*/*tar* one has to execute the *make* command in the ATLAS subdirectory where the file *Makefile* resides in (cf. Figure 13).

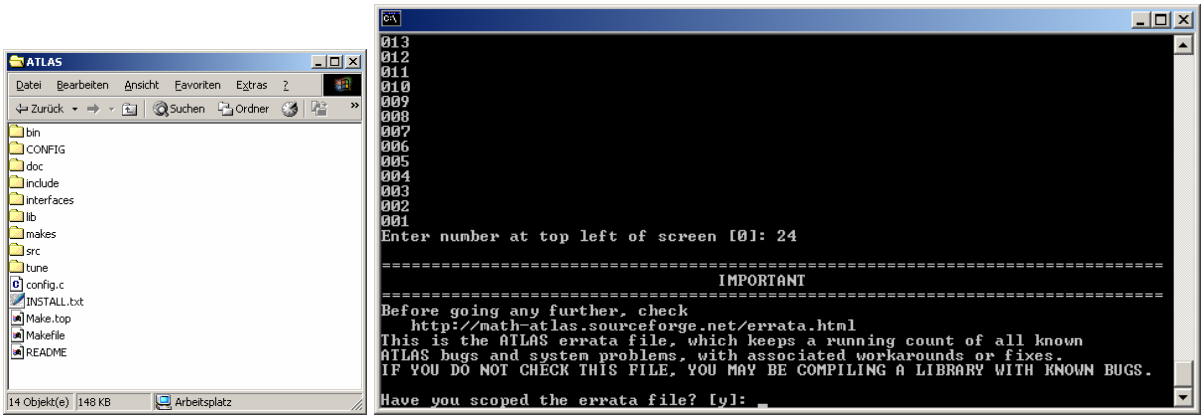


Figure 13: ATLAS sourcecode after decompression. Execution of the *Make* command in this folder starts the build process of this library.

The installation script asks a number of questions before starting the GCC compiler, and the installation process itself needs a couple of minutes. A UNIX-like *.a*-file will be generated containing the library. Unfortunately, this library is not compatible to any Microsoft specific compiler/linker that is used by *Visual Studio*, so it has to be converted to a *.lib*-file.

On http://www.kevinsheppard.com/research/matlabatlas/matlab_atlas.aspx a detailed description can be found how to do this. There is a *bash*-script available for this purpose. This script will basically perform two steps:

- Using the MINGW package of CYGWIN the *.a*-file will be converted to a *.dll* dynamic link library file that does not need CYGWIN anymore.
- A *.lib* wrapper file will be generated with the same name as the *.dll* file.

Before the script can be used, it must be slightly modified to work the following way:

- The names of the procedures that the library should export have to be converted to uppercase.
- The path to the Microsoft command line linker *lib.exe* has to be specified in the script correctly.

After execution of the script, the ATLAS library has the same form as the GOTO library above: a *.lib*/*.dll*-pair, where the *.lib*-file can be included into the application and the *.dll*-file is loaded on execution time. Therefore the *.dll*-file has to be placed again in the same directory as the application in order to be available in runtime mode.

6 Benchmark computations

6.1 Solver benchmarks

In this section we will present some benchmark tests with the solvers and BLAS libraries described above. As a benchmark system we choose a Dual Pentium-4 system[2] running on 2600 MHz with 1024 MB RAM. Furthermore we choose two linear systems – one with 3563 unknowns and one with 12000 unknowns. Both of these problems arise from a real-life simulation with the numerical simulation tool *SILO*. We compare the three different solvers

[2] Although the system is a Dual P4 system, our test application is single threaded and so it uses only one processor.

(CROUT, UMFPACK2 and UMFPACK4), the different BLAS libraries (Standard, MKL, GOTO, ATLAS) and the different compilers (Intel, Compaq).

To show which compiler do the better job in optimisation we choose the maximum available optimisation techniques that are possible for the systems above. In detail this means:

Compaq Visual Fortran 6.0 Compiler:

- Generate Code for: Blended[3]
- Optimisation level: Maximum optimisations
- Math-Library: fast

Intel Fortran Compiler 8.0

- Optimisation: Maximize Speed
- Optimise for: Intel P4
- Use Processor Extensions: Intel P4
- Require Processor Ext.: Intel P4

Microsoft C/C++ Compiler (.NET)

- Optimisation: Maximize Speed
- Optimise for: Intel P4
- Size or speed: Optimise for Speed
- Global optimisation: Yes
- SSE-Extensions: Use SIMD-Extensions 2 (SSE2)

The results for different solvers with different configurations are shown in table 2. It depicts only the result for the solvers, measured in seconds – the time for generating the system (i.e. reading it from hard disc) is of course not included. Every result was computed three times, then the mean value was formed. Because the CROUT does not use any BLAS library at all, the corresponding cells in the table are marked with"—" as these computations would give no new results.

[3] This optimisation is valid for all Pentium-based processors. Configuring the optimisation directly to a Pentium III platform (the maximum possible choice for *Compaq Fortran*) produced a binary, which was about 300-400% slower on our Pentium IV when using standard BLAS routines with UMFPACK2. This behaviour could not be explained.

BLAS library	CROUT		UMFPACK2		UMFPACK4	
	Intel	Compaq	Intel	Compaq	Intel	Compaq
Standard/None	3,8	4,3	2,5	3,1	1,4	1,7
ATLAS	—	—	2,4	2,4	1,3	1,3
MKL	—	—	1,5	1,6	0,9	0,9
GOTO	—	—	1,6	1,6	0,9	0,9

table 2: Different solvers with different BLAS libraries and different compilers, test system with 3563 unknowns

BLAS library	CROUT		UMFPACK2		UMFPACK4	
	Intel	Compaq	Intel	Compaq	Intel	Compaq
Standard/None	64,8	72,1	46,3	59,7	23,2	28,0
ATLAS	—	—	42,5	42,4	16,1	16,1
MKL	—	—	25,0	24,9	9,1	9,3
GOTO	—	—	25,5	25,3	9,2	9,4

table 3: Different solvers with different BLAS libraries and different compilers, test system with 12000 unknowns

It can clearly be seen that the type of the BLAS library is the key component for getting performance. Every BLAS library is superior than the standard BLAS. The most efficient BLAS libraries are obviously the GOTO BLAS library and Intel's MKL-library, both results are nearly identical. The ATLAS library is slower, but it is still an alternative to standard BLAS. ATLAS has even the advantage that it can be compiled for 32 bit Athlon-based processors in contrast to GOTO BLAS and MKL respectively, which only exists for Pentium-based 32 bit processors.

Concerning the solvers UMFPACK4 is obviously the best and so the effort of compiling it under Windows is completely worthwhile. If UMFPACK4 is not available, UMFPACK2 is still a good alternative to the standard CROUT solver.

The type of compiler is only important if no performance BLAS library is used. In this case the *Intel Fortran compiler* is the best choice, as it supports all the Intel processor extensions that are available on the processor we used. When one of the performance BLAS libraries is activated, the differences in speed between the binaries of the both compilers become negligible. But obviously no compiler can compete in its optimisations to the benefits of a well optimised BLAS library.

6.2 Benchmark in a real-life application

The benchmarks in section 6.1 are stand-alone tests. In the following the performance of the various solvers are tested in the Finite Element software named *SILO*. This programme has been developed to simulate the behaviour of granular material in silos and to estimate the relevant loads on the structure. It should be noted, that the failure rate of real silos is significant higher than in other structures.

The following important aspects have to be considered when using a Finite Element Program to simulate granular flow:

- *Material model*
  It is obvious that the reliability of any FE-analysis depends on the assumptions and simplifications of the numerical and mechanical model. Here the constitutive model for the bulk material is of greatest importance. It should be out of discussion that simple limit state models like e.g. the Mohr Coulomb failure criteria can certainly not model the complex behaviour of granular bulk materials during at rest conditions and during flow. Constitutive models as published by Lade (elastic-plastic), Kolymbas (hypoplastic) or v. Wolffersdorff (hypoplastic) for example have to be used in conjunction with a dynamic, viscous part (cf. [8]).
- *Filling procedure*
  The silo has to be filled numerically layer by layer as in reality. This is of great importance in case of bins with inclined walls.
- *Dynamic analysis of granular flow*
  Usually the greatest pressures in silos don't occur during at rest conditions but during emptying of the bin. Therefore one must model the discharge phase. This requires a dynamic mechanical model and code.
- *Boundary conditions*
  The interaction between the granular media and the silo structure has to be modeled. Interface elements and accurate material models (e.g. friction) are required.
- *Numerical algorithms*
  The numerical algorithms must be fast and robust. Correct convergence criterions must be defined.

To take into account all these specific requirements, a special non-linear Finite Element Program *SILO* based on a continuum approach has been developed to simulate the behavior of non-cohesive granular material in silos during at rest conditions and during flow. Details and background information as well as results for 2-dimensional and axisymmetric conditions can be found in several publications (e.g. [4], [5]). The software package including pre- and post processors is written in standard Fortran 77.

This FE-program has been improved in the last years to 3 dimensions to model un-symmetric conditions, like e.g. an eccentric discharged bin. For this investigations a 20-node-isoparametric volume element (bulk element) was implemented in the program (cf. Figure 14).

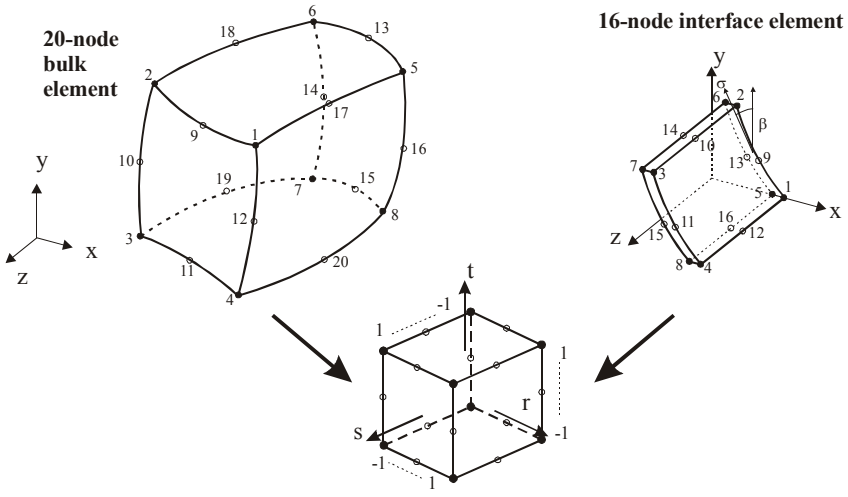
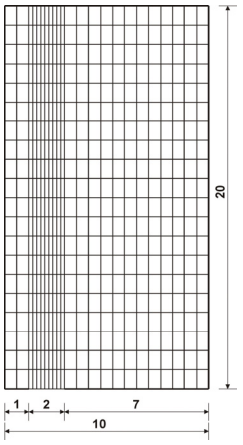


Figure 14: 20-node-bulk and 16-node interface element

The stresses in the granular media stored in bins are significantly dependant on the interaction between the bulk material and the silo walls. Several interface algorithms like e.g. point to surface elements, spring elements, contact elements had been studied. The often used point to surface elements have caused great numerical problems. It should be noted that the bulk material is always in contact with the silo walls. Thus complicated contact algorithms are not required. The best results were obtained with a 16-node-Interface-element (Figure 14). The shape functions are linear in thickness direction as 4 nodes are omitted. Several comparison analysis had been conducted with different linear and non-linear contact models. Fortunately a linear friction model (Mohr-Coulomb) is sufficient enough in most relevant cases.

Simulations of the discharge processes are extremely time consuming. The size of the element mesh and the analysed discharge time are mostly limited by the time for the computation. Most effort is required for solving the linearised global equilibrium equations as will be shown in the following. It should be noted that the global stiffness matrix is non-symmetric due to the constitutive material models used in the program.Various solvers based on the Gauss algorithms or iterative procedures had been tested.



bulk material:	wheat
constitutive model:	Kolymbas
kind of filling:	layer by layer
height:	20 m
width:	10 m (H/D = 2)
width of opening:	2 m
distance of the outlet to corner:	1 m
eccentricity of the outlet:	3 m
time to open the outlet:	0,2 s
time to discharge:	1 s
wall friction coefficient μ_m:	0,25

Figure 15: silo geometry to determine the time performance

Time composition of the various parts of the program

Several calculations were made to compare the time performance of the program *SILO*. All simulations based on the 2-dimensional geometry (plain strain) shown in Figure 15. The number of elements has been increased, from 520 to 1170, gradually.

The results of this comparison calculations are shown in table 4. It can be clearly seen that the triangular decomposition takes the most time (table 4, Figure 16). This part of the numerical analysis needs between 72 % and 87 % of the whole simulation time. When more elements are used the triangular decomposition is going to increase its fraction.

discharging calculations						
number of nodes/elements (steps = 420)	**1754/520**	**2174/650**	**2594/780**	**3014/910**	**3434/1040**	**3854/1170**
start	0,24	0,25	0,24	0,26	0,25	0,27
input (nodes, coordinates)	0,05	0,06	0,07	0,10	0,09	0,09
input (elements)	0,12	0,14	0,17	0,20	0,21	0,32
calculation of the load vector	0,00	0,00	0,00	0,00	0,01	0,01
calculation of the load increments	75,37	95,98	113,47	133,99	151,34	169,45
bild-up the stiffness matrix	163,3	211,84	254,69	309,35	353,66	404,73
triangular decomposition of the stiffness matrix	**1444,75**	**2414,28**	**3692,91**	**5434,34**	**7507,46**	**10036,50**
back substitution	56,28	85,49	118,62	160,16	201,68	252,24
calculation of the inner loads	65,45	88,10	108,22	132,33	151,13	174,07
frictional forces	0,02	0,06	0,06	0,03	0,06	0,04
residuum and mass terms	171,88	229,86	283,97	345,17	397,27	456,57
output	6,62	8,18	9,67	11,53	12,86	14,38
restart output	0,23	0,34	0,39	0,47	0,52	0,60
whole time of the calculation	**1984,62**	**3134,93**	**4582,81**	**6528,36**	**8777,01**	**11509,72**

table 4: time composition in [s]

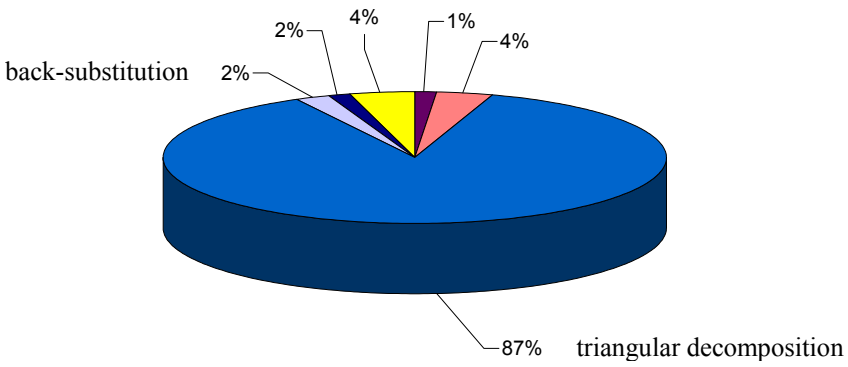


Figure 16: graphical presentation of the time composition for 3854 nodes

Therefore the enhancement of the triangular decomposition should be the main objective. Different solver were tested, et al. CROUT, UMFPACK2. The results are given in table 5.

kind of solver number of equations	CROUT	UMFPACK2	time difference	acceleration (UMFPACK 2)
3563	9,4 s	4,5 s	4,9 s	**52 %**
5747	36,5 s	14,4 s	22,1 s	**60,6 %**

table 5: Time for triangular decomposition: Crout versus Umfpack 2

The fastest acceleration of UMFPACK2 versus CROUT of the stand alone analysis (see section 6.1) is about 65-70 %. The acceleration in *SILO* (a real-life application) is 61 % for 5747 nodes. The analysis with UMFPACK4 also for larger problem sizes is currently undertaken.

7 Conclusion

In this report we demonstrated how different highly optimised mathematical libraries for the solution of linear systems can be successfully compiled and linked to a Fortran program in a Windows environment. Our numerical tests have shown that the replacement of a standard CROUT solver by the UMFPACK4 solver can speed up the solution of a linear system to a factor of 2-3. With a further replacement of the underlying BLAS library that performs basic linear algebra operations by a processor optimised variant one can again obtain a a factor of 2-3, so the combination of both can result in a speed-up of almost factor 10 for the solution of linear systems. Furthermore the ratio of speed-up will grow with the problem size: In our tests a *smaller* problem size of 3563 unknowns indicate a speed-up of about 400% whereas with a larger problem size the speed-up was even higher.

The software package *SILO* which was used to generate our test problems now heavily depends on the solution of linear systems: Up to 89% of the computational time is used only for the solution of linear systems with the CROUT solver (cf. [1], Figure 16: 87% for triangular decomposition, 2% for back-substitution). Thus replacing the solver components of the *SILO* simulation tool with a proper UMFPACK/BLAS combination promises a large potential of improvement. As our numerical examples have shown, the replacement of the solver of the linear problem has direct influence on the performance of the whole solving process, and the performance of this component is directly passed to the whole solver without any considerable loss. So this represents the main objective of the current development.

Apart from accelerating the solver itself, another point of interest for future development will be the porting of *SILO* and the solver packages to an Opteron®-based 64 bit computer system. Most of the simulations with *SILO* are at the moment performed with AMD Athlon®- or Intel Pentium®-based computers of 1-2 GHz. Numerical tests with the FEATFLOW package have shown (cf. FEAT-Indices in [13]), that the Opteron-based processor with about 2.2 GHz is faster than an Intel P4-Xeon based computer running with about 3 GHz, and even much faster than a standard 32 bit Intel-P4-based computer. Highly optimised BLAS routines are already available for Opteron-based computers (cf. [12]), but unfortunately at the moment only for LINUX-based operating systems. So another aim will be to port *SILO* to LINUX to exploit the power reserves provided by these kind of systems, too.

8 References

[1] Eibl, J., Landahl, H.: Zur Frage des Silodrucks, Beton- und Stahlbetonbau, 104{110, 1982

[2] Eibl J., Rombach G. et. al.: Rechnerische Erfassung der Silodrücke – Algorithmen. in: Silobauwerke und ihre spezifischen Beanspruchungen (Deutsche Forschungsgemeinschaft Hrsg.). Weinheim, 2000

[3] Eibl, J., Ruckenbrod, C., Braun, A.: Rechnerische Erfassung der Silodrücke - Algorithmen, Arbeits- und Ergebnisbericht des SFB 219, Universität Karlsruhe, 1993

[4] Rombach, G.; Numerical Simulation of Granular Flow in Silos; Proc. 20th Int. Finite Element Congress -- FEM 91, Baden-Baden, November 1991

[5] Rombach, G.: Schüttguteinwirkungen auf Silozellen - Exzentrische Entleerung, Eibl, J., Hilsdorf, H.K. (Hrsg.), Schriftenreihe d. Inst. f. Massivbau u. Baustofftechnologie, Univ. Karlsruhe, Heft 14, 1991

[6] Rombach, G.; Eibl, J.; Granular Flow of Materials in Silos - Numerical Results; Bulk Solids Handling, Volume 15, No. 1, 1995, S. 65-70

[7] Rombach, G.; Turek, S.; Pöschel, T.; DFG Sonderbrogramm: Verhalten Granularer Medien; Arbeitsbericht 2002

[8] Weidner, J.: Vergleich von Stoffgesetzen granularer Schüttgüter zur Silodruckermittlung, Dissertation, Universität Karlsruhe, 1990

[9] ATLAS-Library, http://math-atlas.sourceforge.net

[10] CYGWIN, http://www.cygwin.com

[11] Harwell-Library, http://scicomp.ewha.ac.kr/netlib/harwell/

[12] High-Performance BLAS by Kazushige Goto, http://www.cs.utexas.edu/users/kgoto/

[13] Homepage of the FEATFLOW project, http://www.featflow.de

[14] Intel Homepage, http://www.intel.com/

[15] Kevin K. Sheppard, http://www.kevinsheppard.com/research/matlabatlas/matlab_atlas.aspx

[16] SPARSEKIT matrix computation library, http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html

[17] Standard-BLAS-implementation, http://www.netlib.org/blas

[18] UMFPACK4-library, http://www.cise.ufl.edu/research/sparse/umfpack/index.html