CCRG OpenMP Compiler: Experiments and Improvements

Huang Chun and Yang Xuejun

National Laboratory for Parallel and Distributed Processing, P.R. China
{chunhuang73}@hotmail.com

Abstract. In this paper, we present the design and experiments of a practical OpenMP compiler for SMP, called CCRG OpenMP Compiler, with the focus on its performance comparison with commercial Intel Fortran Compiler 8.0 using SPEC OMPM2001 benchmarks. The preliminary experiments showed that CCRG OpenMP is a quite robust and efficient compiler for most of the benchmarks except *mgrid* and *wupwise*. Then, further performance analysis of *mgrid* and *wupwise* are provided through *gprof* tool and Intel optimization report respectively. Based on the performance analysis, we present the optimized static schedule implementation and inter-procedural constant propagation techniques to improve the performance of CCRG OpenMP Compiler. After optimization, all of the SPEC OMPM2001 Fortran benchmarks can be executed on SMP systems efficiently as expected.

1 Introduction

The OpenMP[1] has gained momentum in both industry and academy, and has become the de-facto standard for parallel programming on shared memory multiprocessors. The open source compilers and runtime infrastructures promote the development and acceptance of OpenMP effectively. There have been several recent attempts, such as NanosCompiler[2] and PCOMP[3] for Fortran77, Omni[4] and Intone[5] for Fortran77 and C, OdinMP[6] for C/C++, and Nanos Mercurium[7] on top of Open64 compilers. All of them are source-to-source translators that transform the code into the equivalent version with calls to the associated runtime libraries.

CCRG OpenMP Compiler[1] (CCRG, for short) aims to create a freely available, fully functional and portable set of implementations of the OpenMP Fortran specification for a variety of different platforms, such as Symmetric Multiprocessor (SMP) as well as Software Distributed Shared Memory (SDSM) system. As the above compilers, CCRG also uses the approach of the source-to-source translation and runtime support to implement OpenMP. CCRG has the following features.

[1] Both the compiler and runtime library will be available from the CCRG Fortran95 Compiler project web site at http://cf95.cosoft.org.cn

– Generate only one external subroutine for each subprogram which may specify one or several parallel regions, and the parallel regions in the subprogram are implemented using ENTRY statements. Therefore, the size of code generated by the source-to-source translator has been reduced significantly.
– Fully support Fortran90/95 programming languages except the type declaration statements whose kind-selector involves the intrinsic procedure.
– Be robust enough to enable testing with real benchmarks.
– Support multiple target processors and platforms, including Digital alpha[9], Intel Itanium and Pentium, SDSM - JIAJIA[10] and SMP.

In this paper, we present the design and experiments of CCRG for SMP, with the focus on performance comparison with commercial Intel OpenMP Compiler 8.0[11] using SPEC OMPM2001 benchmarks[12]. The preliminary experiments showed that most of the Fortran benchmarks with CCRG OpenMP are executed as fast as with Intel OpenMP Compiler on SMP, except *mgrid* and *wupwise*. Based on performance analysis for *mgrid* and *wupwise*, we present the optimized static schedule implementation and inter-procedural optimization(IPO) which improve the performance of *mgrid* and *wupwise* as desired.

In the next section we briefly outline the design of the CCRG OpenMP Compiler. Section 3 describes the experiments and performance analysis using SPEC OMPM2001 in detail. Section 4 presents the optimization techniques based on the result of section 3 and reports the performance improvements. Conclusion and future work are given in section 5.

2 CCRG OpenMP Compiler

CCRG OpenMP Compiler has fully implemented OpenMP 1.0 and part features of OpenMP 2.0 Fortran API on the POSIX thread interface. As most of the open source OpenMP compilers[2–7], it includes a source-to-source translator to transform OpenMP applications into the equivalent Fortran programs with the runtime library calls. The source-to-source translator is based on Sage++[14] and consists of two parts, a Fortran OpenMP syntax parser and a translator which converts the internal representation into the parallel execution model of the underlying machine. Sage++ is an object-oriented compiler preprocessor toolkit for building program transformation systems for Fortran 77, Fortran 90, C and C++ languages. Though many features of Fortran 90/95 are not supported in Sage++, it is not very difficult to add new elements to the system because of its well-structured architecture. In the syntax parser, the OpenMP syntax description is added for supporting OpenMP directives as well as the new Fortran90/95 languages elements, as shown in Fig.1.

The parser recognizes the OpenMP directives and represents their semantics in a machine independent binary internal form. A *.dep* file is produced to store the internal representation for each OpenMP source file. The translator reads the *.dep* file and exports the normal Fortran program with calls to the runtime library. In [2–7], a subroutine is generated for each parallel region by the

```
omp_directive:
    omp_parallel
    | omp_paralleldo
    | omp_parallelsections
    | omp_parallelworkshare
    | omp_single
    | omp_master
    | ......;

omp_parallel:
  PARALLEL end_spec needkeyword omp_clause_opt keywordoff
  {
    omp_binding_rules (OMP_PARALLEL_NODE);
    $$ = get_bfnd (fi, OMP_PARALLEL_NODE, SMNULL, $4, LLNULL, LLNULL);
  }
```

Fig. 1. OpenMP Syntax Description

translators. Two separate subroutines are needed to implement the two parallel regions in Fig.2, which means that many same declare statements are included. The internal subroutine can be used to reduce the size of code generated by the source-to-source translator. Some commercial OpenMP compilers use this strategy to implement OpenMP parallel region, such as IBM XLF compiler[13]. But the special support of compilers is needed because the Fortran standard specifies some constraints on using an internal subroutine. Therefore, we use an alternative approach by using ENTRY statement to eliminate these same statements in CCRG OpenMP. If a subroutine contains one or more ENTRY statements, it defines a procedure for each ENTRY statement and permits this procedure reference to begin with a particular executable statement within the subroutine in which the ENTRY statement appears. Therefore, ENTRY name can be used to guide all the threads to execute parallel regions correctly, such as `test_$1` and `test_$2` shown in Fig.3.

```
  SUBROUTINE test(a)
    DIMENSION a(100)
!$OMP PARALLEL DO PRIVATE(K)
    DO 100 k = 1, 100
100     a(k) = 0.9
      .........
!$OMP PARALLEL NUM_THREADS(4)
      .........
!$OMP END PARALLEL
  END
```

Fig. 2. An OpenMP Example

The source-to-source translator encapsulates all parallel regions of a main program or subprogram into one external subroutine. The ENTRY procedures are generated to implement parallel regions, as shown in Fig.3. So, only one external subroutine `test_$0` is generated for the OpenMP example in Fig.2, which contains two ENTRY procedures `test_$1` and `test_$2`. The procedures

defined by ENTRY statements share the specification parts. Therefore, the code size generated by the translator is reduced largely.

```
SUBROUTINE test_$0(a)
  DIMENSION a(100)
  INTEGER lc_k
  INTEGER _omp_dolo, _omp_dohi, comp_static_more
!The first parallel region
  ENTRY test_$1 ()
    CALL comp_static_setdo (1, 100, 1, 0)
    DO WHILE (comp_static_more(_omp_dolo, _omp_dohi, 1).eq.1)
      DO 100 lc_k = _omp_dolo, _omp_dohi, 1
100     a(lc_k) = 0.9
    END DO
    CALL comp_barrier()
  RETURN
!The second parallel region
  ENTRY test_$2()
    .........
    CALL comp_barrier()
  RETURN
END

SUBROUTINE test(a)
  DIMENSION a(100)
  CALL comp_runtime_init ()
  CALL comp_parallel (test_$1, 0, 1, a)
    .........
  CALL comp_parallel (test_$2, 4, 1, a)
  CALL comp_exit ()
END
```

Fig. 3. Fortran Program using ENTRY Statement after Transformation

The CCRG OpenMP runtime library for SMP has been implemented based on the standard POSIX thread interface. The library is platform-independent except few functions, such as `comp_parallel`, `comp_barrier` and `comp_flush`. It focuses on three tasks: thread management, task schedule, and implementation of OpenMP library routines and environment variables. The "`comp_`" functions shown in Fig.3 are main functions for thread management and task schedule. `comp_runtime_init` initializes the runtime system and reads the associated environment variables. `comp_exit` terminates all the slaves in the thread pool and releases memory. Function `comp_static_setdo` and `comp_static_more` implement the static schedule in OpenMP. `comp_barrier` synchronizes all the threads in the current thread team. `comp_parallel` is the most complex function in the library, which creates slave threads when necessary and starts the slave threads in the thread pool to execute the parallel region procedures. It has following form.

```
comp_parallel (parallel_region_procedure_name,
               num_threads, num_parameter, param1, param2,...)
```

If there is no `NUM_THREADS` clause in a OpenMP parallel region directive, num_threads is `0`, as shown in the first parallel region in Fig.3. `comp_parallel` decides the number of the threads in the team according to the environment variable or library calls, or the default value which is equal to the number of physical processors of the underlying target.

3 Experiments

To evaluate CCRG OpenMP Compiler, SPEC OMPM2001[12] Fortran benchmarks are compiled and executed. The host platform for the experiments is a HP server rx2600 with four Itanuim2 processors (1.5GHz) and Linux IA-1 2.4.18-e.12smp.

3.1 Result

The backend compiler of CCRG can be any compilers executed over the target machines, including commercial compilers(Intel, PGI, etc.) and GNU compiler. To compare CCRG with commercial Intel compiler exactly, Intel Fortran Compiler 8.0 is used as the backend compiler of CCRG. Fig.4 and Fig.5 show the Base Ratios of SPEC OMPM2001 Fortran benchmarks of CCRG and Intel OpenMP Compiler 8.0[2] with four OpenMP threads . "-O3" and "-O3 -ipo" options are used respectively. "-ipo" option enables inter-procedural optimization(IPO) across files.

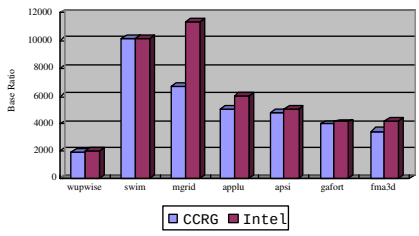

Fig. 4. Base Ratios of CCRG and Intel without IPO

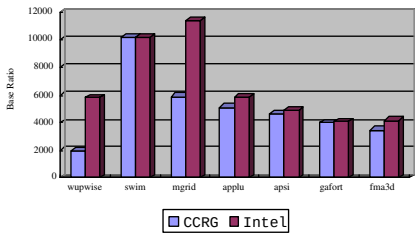

Fig. 5. Base Ratios of CCRG and Intel with IPO

The performance data in Fig.4 and Fig.5 suggest that CCRG indeed makes good use of the multiprocessing capabilities offered by the underlying platform as Intel OpenMP Compiler with two exceptions: *mgrid* and *wupwise*. The Base Ratio of *mgrid* with CCRG is only half of that with Intel whether inter-procedure optimization option is used or not. When "-ipo" option is used, the performance of *wupwise* with Intel can be improved greatly, while CCRG seems to block some further optimization.

[2] 318.galgel can not execute correctly using Intel OpenMP Compiler 8.0 on our server.

3.2 Performance Analysis

For most of the SPEC OMPM2001 Fortran benchmarks, CCRG OpenMP Compiler results in almost exactly the same Base Ratios as Intel OpenMP Compiler. But the performance of *mgrid* and *wupwise* with CCRG are much worse than that with Intel OpenMP Compiler. In this section, we analyze and explain why *mgrid* and *wupwise* perform poorly in detail. HP server rx2600 with 2 Itanuim2 processors (1.0GHz) is used for performance analysis here.

mgrid Fig.6 shows the execution time of the top six pocedures[3] in *mgrid* with "TRAIN" input sets. "`_p1`" and "`_p2`" denote the procedures generated for the first and second parallel regions in one procedure respectively. For example, the column of `resid_p1` in Fig.6 denotes the execution time of the first parallel region of `resid`. The procedures `resid_p1`, `psinv_p1`, `rprj3_p1`, `interp_p1` and `interp_p2` cause the different execution time between CCRG and Intel.

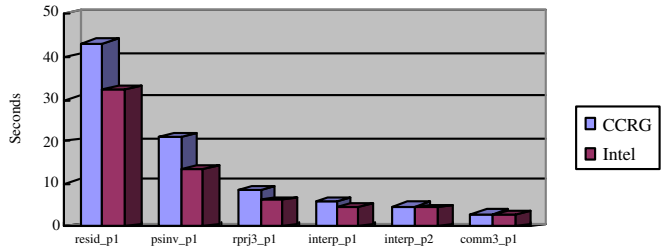

Fig. 6. Execution Time of Top 6 Procedures in *mgrid*

`resid_p1`, `psinv_p1`, `rprj3_p1`, `interp_p1` and `interp_p2` are the procedures generated for the simple `PARALLEL DO` constructs. In CCRG, the structure of the procedure is same as that shown in Fig.3. Most of the execution time of the procedures are spent to execute the nested DO-loop. Comparing with Intel OpenMP Compiler, CCRG introduces an additional loop level to implement the schedule types in OpenMP.

```
DO WHILE (comp_static_more(_omp_dolo,_omp_dohi,_omp_doin).eq.1)
  .........
END DO
```

This additional loop is a while loop whose control condition is a logical expression containing a function call. It encloses the original loops and becomes the most outer loop. Therefore, the performance of the whole procedure degrades significantly.

[3] In Intel OpenMP Compiler, the procedure should be the "T-region".

wupwise Unlike *mgrid*, the performance of *wupwise* is affected by IPO largely. Table 1 shows the execution time of the top four procedures in *wupwise* with "TRAIN" input sets. The time of subroutine `zgemm` with CCRG is 82.10 seconds, while the time with Intel is only 7.91 seconds.

Table 1. Execution Time of Top 4 procedures in *wupwise*

	CCRG		Intel	
	Subroutine	Execution time(sec)	Subroutine	Execution time(sec)
1	zgemm	82.10	dlaran	9.77
2	gammul	10.77	zaxpy	8.57
3	zaxpy	7.74	zgemm	7.91
4	dlaran	7.35	lsame	1.87

The Intel compiler provides the extensive support for inter-procedural analysis and optimization, such as points-to analysis and mod/ref analysis required by many other optimizations. However, only the equivalent version transformed by the source-to-source translator of CCRG can be seen by the backend compiler Intel Fortran Compiler. Because the parallel region procedures are called by function `comp_parallel` as actual parameters, the source-to-source translator dose not keep the information about the caller-callee relationship between the original procedures. So, the backend compiler can not process the further inter-procedural analysis and optimization.

```
High Level Optimizer Report for: zgemm_
Block, Unroll, Jam Report:
(loop line numbers, unroll factors and type of transformation)
Loop at line 2194 unrolled with remainder by 6
Loop at line 2177 unrolled with remainder by 6
Loop at line 2158 unrolled with remainder by 6
.........
                        (a) For Program Transformed by CCRG
```

```
High Level Optimizer Report for: zgemm_
Block, Unroll, Jam Report:
(loop line numbers, unroll factors and type of transformation)
Loop at line 2377 completely unrolled by 3
Loop at line 2379 completely unrolled by 3
Loop at line 2360 completely unrolled by 3
.........
                                  (b) For Source OpenMP Program
```

Fig. 7. Optimization Report Generated by Intel Fortran Compiler 8.0

Though `zgemm` is only called in `su3mul` with several constants which are used to control the loops of `zgemm`, these constants have not been propagated to `zgemm` in CCRG. From the above two optimization reports, it is obvious that inter-procedural constant propagation has been applied to `zgemm` when

using Intel OpenMP Compiler. Many loops in `zgemm` are completely unrolled according to the value of actual parameter.

4 Optimization

Section 3.1 describes the key factors which influence on the performance of *mgrid* and *wupwise* programs. In this section, the optimized static schedule implementation and inter-procedural optimization are presented to CCRG OpenMP Compiler for the improvement of performance of these programs.

4.1 Optimized Static Schedule

In the following three cases, function `comp_static_more` is `.TRUE.` only once for each thread when executing a parallel region procedure in CCRG.

- Absence of the `SCHEDULE` clause.
- Static schedule without chunk, i,e. `SCHEDULE(STATIC)` is specified.
- Static schedule, and both chunk size and number of iteration are known during compile time, and `(chunk size` $\times$ `number of threads)` $\leq$ `number of iteration`.

Therefore, the parallel region subroutine code in Fig.3 can be replaced with the codes in Fig.8. `comp_static_once` is called only once to implement the `PARALLEL DO` directive in Fig.2.

```
SUBROUTINE test_$0(a)
  DIMENSION a(100)
  INTEGER lc_k
  ENTRY test_$1 ()
    CALL comp_static_setdo(1,100,1,0)
    CALL comp_static_once(_omp_dolo,_omp_dohi,1)
    DO 100 lc_k=_omp_dolo,_omp_dohi,1
100   a(lc_k)=0.9
    CALL comp_barrier()
  RETURN
END
```

Fig. 8. Implementation of STATIC Schedule without Chunk Size

After optimization, the execution time of all procedures in Fig.6 has been reduced significantly. The middle columns in Fig.9 are the execution time using the optimized static schedule implementation. Obviously, the performance of whole *mgrid* has been improved largely too.

`SCHEDULE` clause is not specified in most of OpenMP programs, we can just use `comp_static_once` instead of `comp_static_more` with introducing an additional outer loop.

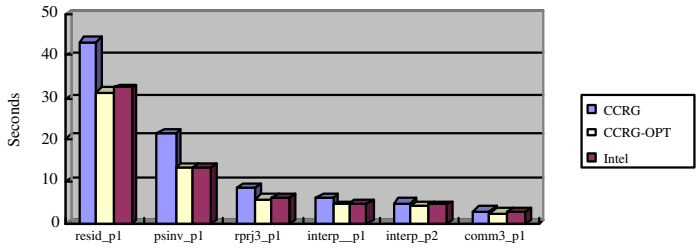

Fig. 9. Execution time of Main Procedures after Optimization

4.2 Inter-Procedural Constant Propagation

Because CCRG uses the source-to-source approach, some inter-procedural optimizations may no longer be applicable for some OpenMP programs. For example, in *wupwise*, subroutine `zgemm` is called only once in subroutine `su3mul` where the third, fourth and fifth actual parameters are integer constants. But these constants have not been propagated to `zgemm`.

This is a native problem of source-to-source OpenMP compilers. We present an approach to solving it by adding inter-procedural optimization in source-to-source translator. Inter-procedural optimization contains two-pass compilation, as shown in Fig.10.

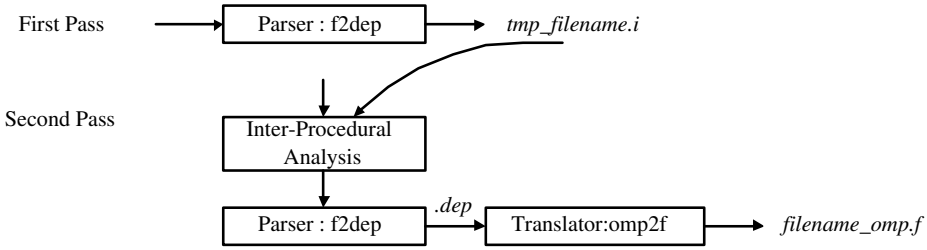

Fig. 10. Inter-Procedural Optimization

In the first pass, the parser scans all the project files to record the information about procedure calls, such as procedure name, formal parameters, procedure name and actual parameters called by the procedures in the files. Temporary file *tmp_filename.i* is generated for each file in the project. For example, *tmp_su3mul.i* for *su3mul.f* in *wupwise* contains the information as follows.

```
{SUBROUTINE "SU3MUL"
  (FORMAL ("U" COMPLEX*16 DIMENSION(2 3 *))
          ("TRANSU" CHARACTER*1 SCALAR)
          ("X" COMPLEX*16 DIMENSION(1 *))
          ("RESULT" COMPLEX*16 DIMENSION(1 *)))
  (SUBROUTINE "ZGEMM"
    (ACTUAL (TRANSU, 'NO TRANSPOSE',3,4,3, ONE,U,3,X,3,ZERO,RESULT,3)
}
```

In the second pass, the parser reads and analyzes all of the temporary files firstly. If the callers always use the same integer constant as certain actual parameter to call a procedure, the parser inserts an assignment statement before the first executable statement in the callee. The constant is assigned to the parameter in the assignment statement(see Fig.11).

```
  SUBROUTINE ZGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
$                    BETA, C, LDC )
  ! Variables Declaration Statements.........
  ! Assignment to Formal parameters
    M = 3
    N = 4
    K = 3
  !Other Executable Statements
  END
```

Fig. 11. Inserts the Assignment Statements in the Callee

Therefore, the constant propagation is implemented through assignments to formal parameters. That is, the source-to-source translator only provides the initial values of the formal parameters after inter-procedural analysis, the backend compiler utilizes the information to make further optimization. After optimization, the execution time of *wupwise* has been reduced significantly as shown in Fig.12, whose middle columns are the execution time after inter-procedural optimization.

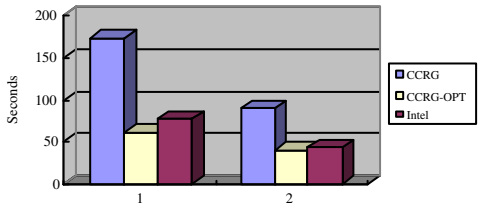

Fig. 12. Execution Time of *wupwise* after Optimization

Complete inter-procedure constant propagation needs to be supported by other optimizations, such as source-level data flow analysis and constant propagation within a procedure. At present only integer constant actual parameters can be propagated cross procedures.

5 Conclusion and Future Work

The CCRG OpenMP Compiler is a mature source-to-source compiler for OpenMP. All SPEC OMPM2001 Fortran benchmarks have been compiled and executed on

SMP system efficiently. CCRG supports OpenMP Fortran90/95 programming, and all of the Fortran benchmarks achieve the comparable Base Ratios as Intel OpenMP Compiler.

In the paper, we show our experience for performance improvement of CCRG. We analyze two benchmarks *mgrid* and *wupwise* whose execution time with CCRG were much longer than that with Intel. Two optimization techniques, namely, optimized static schedule and inter-procedural constant propagation, are presented to resolve the performance problems in these two programs. After optimization, the performances of *mgrid* and *wupwise* are improved significantly.

In the future, we plan to design and implement more source-to-source optimization strategies and complete inter-procedural optimization framework. This framework will be applicable for not only IPO but also for profile-guided optimization aimed at OpenMP. In addition, the performance of CCRG will be evaluated on the large SMP systems.

Acknowledgements

This work was supported by National 863 Hi-Tech Programme of China under grant No. 2002AA1Z2101 and 2004AA1Z2210.

References

1. The OpenMP Forum. OpenMP Fortran Application Program Interface, Version 2.0, November 2000. See http://www.OpenMP.org.
2. M. Gonzalez, E. Ayguade, J. Labarta, X. Martorell, N. Navarro and J. Oliver. NanosCompiler: A Research Platform for OpenMP Extensions. In Proc. of the 1st European Workshop on OpenMP (EWOMP'99). Lund, Sweden, October, 1999.
3. Seung Jai Min, Seon Wook Kim, Michael Voss, Sang Ik Lee, and Rudolf Eigenmann. Portable Compilers for OpenMP. In Proc. of the 2nd Workshop on OpenMP Applications and Tools (WOMPAT'01), Lecture Notes in Computer Science, 2104 pages 11-19, July 2001.
4. Omni OpenMP Compiler Project. http://phase.hpcc.jp/Omni/
5. INTONE: Innovative Tools for Non Experts, IST/FET project (IST-1999-20252). http://www.cepba.upc.es/intone/
6. C. Brunschen and M. Brorsson. OdinMP/CCp - A Portable Implementation of OpenMP for C. In Proc. of the 1st European Workshop on OpenMP(EWOMP'99). Lund, Sweden, October 1999.
7. J. Balart, A. Duran, M. Gonz'alez, X. Martorell, E. Ayguade and J. Labarta, Nanos Mercurium: a Research Compiler for OpenMP, In Proc. of the 6th European Workshop on OpenMP (EWOMP'04), Stockholm, Sweden. October, 2004.
8. Open64 Compiler and Tools. http://sourceforge.net/projects/open64
9. Huang Chun and Yang Xuejun. Performance Analysis and improvement of OpenMP on Software Distributed Shared Memory System. In Proc. of the 5th European Workshop on OpenMP (EWOMP'03). Aachen, Germany. September, 2003.

10. Weiwu Hu, Weisong Shi and Zhimin Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In Proc. of the High Performance Computing and Networking (HPCN'99), Lecture Notes in Computer Science 1593, pp. 463-472, Springer, Amsterdam, Netherlands. April, 1999.
11. Intel Corporation. Intel Fortran Compilers for Linux Application Development, 2003. http://www.intel.com/software/products/compilers/linux.
12. V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W.B. Jones and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT'01), Lecture Notes in Computer Science 2104, pages 1-10, July 2001.
13. Charles Grassl. Shared Memory Programming: Pthreads and OpenMP. http://www.csit.fsu.edu/ burkardt/fsu/7.OpenMP.pdf. October, 2003.
14. Sage++ Users Guide. http://www.extreme.indiana.edu/sage/.