LLNL-CONF-404412

Automatic Parallelization Using OpenMP Based on STL Semantics

C. Liao, D. J. Quinlan, J. J. Willcock, T. Panas

June 3, 2008

Languages and Compilers for Parallel Computing (LCPC)
Edmonton, Canada
July 31, 2008 through August 2, 2008

Automatic Parallelization Using OpenMP Based on STL Semantics*

Chunhua Liao, Daniel J. Quinlan, Jeremiah J. Willcock and Thomas Panas

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
{liao6,quinlan1,willcock2,panas2}@llnl.gov

Abstract. Automatic parallelization of sequential applications using OpenMP as a target has been attracting significant attention recently because of the popularity of multicore processors and the simplicity of using OpenMP to express parallelism for shared-memory systems. However, most previous research has only focused on C and Fortran applications operating on primitive data types. C++ applications using high level abstractions such as STL containers are largely ignored due to the lack of research compilers that are readily able to recognize high level object-oriented abstractions of STL. In this paper, we use ROSE, a multiple-language source-to-source compiler infrastructure, to build a parallelizer that can recognize such high level semantics and parallelize C++ applications using certain STL containers. The idea of our work is to automatically insert OpenMP constructs using extended conventional dependence analysis and the known domain-specific semantics of high-level abstractions with optional assistance from source code annotations. In addition, the parallelizer is followed by an OpenMP translator to translate the generated OpenMP programs into multi-threaded code targeted to a popular OpenMP runtime library. Our work extends the applicability of automatic parallelization and provides another way to take advantage of multicore processors.

1 Introduction

Today's multicore processors have been forcing application developers to parallelize legacy sequential codes and/or write new parallel applications if they want to take advantage of thread-level parallelism supported by hardware. However, parallel programming is never an easy task for users, given the stunning work to deal with extra issues in parallel computing, such as dependencies, synchronization, load balancing, and race conditions. Therefore, parallelizing compilers and tools are playing increasingly important roles in allowing the full utilization of new computer systems and enhancing the productivity of users.

* This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. We thank Dr. Qing Yi for her expert advice in the design of the dependence analysis interface.

OpenMP [1] is a simple and portable parallel programming model which extends existing programming languages like C/C++ and Fortran 77/90 to include additional parallel semantics. The extensions OpenMP provides contain compiler directives, user level runtime routines and environment variables. Programmers can use OpenMP to express parallelization opportunities and strategies for applications. Moreover, the simple API provided by OpenMP has attracted parallelizing compilers and tools to use OpenMP as a target for interactive or automatic parallelization. Representative examples include the Intel C++/Fortran compiler [2] and the Parawise/CAPO tools [3].

Although numerous parallelizing compilers and tools have been presented during the past decades, most of them focus only on C and Fortran applications operating on primitive data types. On the other hand, object-oriented languages, especially C++, are being widely used for developing scientific computing applications. Those applications are often written with various high level libraries, such as the C++ Standard Template Library (STL), now part of the C++ standard. While significantly improving code reuse, high level libraries in applications often impede program analyses and, consequently, program optimizations. For example, compilers will most likely treat the references to STL vector elements as opaque function calls (member function **operator**[]). Without knowing the semantics of the abstractions, the compilers are not able to apply necessary analyses for parallelization.

In this paper, we explore compiler techniques to recognize high level abstractions in C++ applications in order to discover more opportunities for parallelizing applications that use STL. ROSE [4], a source-to-source compiler infrastructure, is used to automatically insert OpenMP constructs into a sequential input C++ code and further translate the generated OpenMP application into multithreaded code targeted to a popular OpenMP runtime library. In particular, we extend conventional data dependence analysis to help in parallelizing object-oriented applications, with optional assistance from source code annotations. Our goal is to automate the process of migrating existing sequential C++ applications to multicore machines and to assist in developing new parallel applications.

2 The ROSE Compiler Infrastructure

ROSE is a project to define a new type of compiler technology that allows even non-expert users to exploit compilation techniques to address the analysis and optimization of user-defined abstractions. It provides a mechanism to construct specialized source-to-source translators. ROSE is particularly useful in building custom program analysis and translation tools that operate on source code for C, C++, and Fortran.

Figure 1 illustrates the main components of ROSE. The Edison Design Group (EDG) front-end [5] is used to parse C and C++ applications. EDG source code and interfaces are protected under commercial or research licenses, but may be distributed freely in binary form. Language support for Fortran 2003

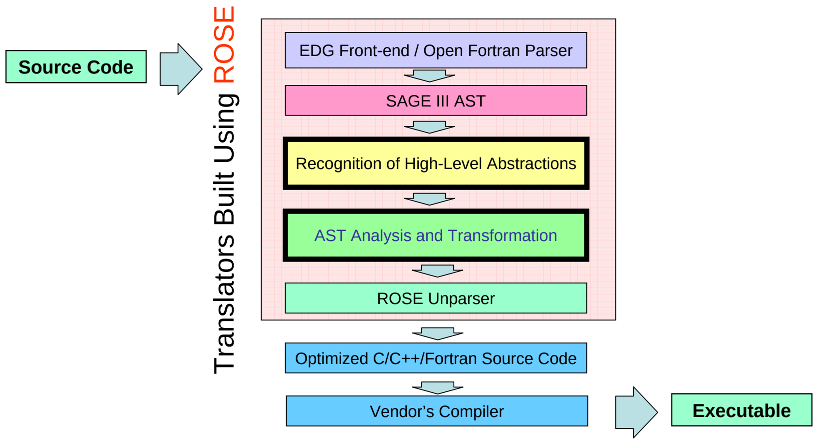
Source Code
Translators Built Using ROSE
EDG Front-end / Open Fortran Parser
SAGE III AST
Recognition of High-Level Abstractions
AST Analysis and Transformation
ROSE Unparser
Optimized C/C++/Fortran Source Code
Vendor's Compiler
Executable

Fig. 1. The ROSE compiler infrastructure

(including earlier versions back to Fortran IV) is based on the open source Open Fortran Parser [6] developed at Los Alamos National Laboratory. Internally, ROSE converts multiple intermediate representations (IR) of the front-ends into a uniform format, the ROSE Abstract Syntax Tree (AST), which is an object-oriented IR developed by the ROSE team and based loosely on the Sage++ IR design [7]. Also, a set of distributed symbol tables are associated with the AST tree to store symbols' information within each scope. Generic and custom program analysis and optimization tools can be built on top of the AST, after an optional phase recognizing high level abstractions of scientific applications. The ROSE unparser generates source code in the original source language from the optimized AST, with all original comments and C preprocessor control structures preserved.

The ROSE AST is a fully type resolved AST, with full overloaded function resolution and semantic analysis. All information in the application source code is preserved in the ROSE AST, including C preprocessor control structure, source comments, source position information, and C++ template information (e.g., template arguments). In particular, the ROSE AST is an intuitive, object-oriented IR with a rich set of interfaces for building source-to-source translators. Users can use the interfaces to quickly implement common operations needed for conducting program analyses and translations. These operations include AST traversals in various orders, efficient AST node queries, AST subtree copying, insertion, removal, and symbol table lookups. In addition, a high-level AST construction API is provided to the users to ease both top-down and bottom-up construction of AST subtrees and to hide the maintenance details of the symbol tables. Moreover, persistent attributes are introduced in the AST to easily store and evaluate arbitrary user-defined information, including AST annotations. These attributes are persistent in that they are preserved when the AST is written out to (and read in from) a binary file. AST traversals form a way to alternatively compute and manage attributes on the stack and thus permit attribute evaluation; the computation of user-defined attributes during top-down traversal (inherited attributes) and/or bottom-up (synthesized attributes) traversals. Other essential AST utilities provided for the users include AST consistancy checkers, file I/O, and AST visualization.

A number of generic program analyses and transformations have been developed for ROSE. They are designed to be utilized by users via simple function calls to interfaces. The program analyses available include call graph analysis, control flow analysis, data flow analysis (live variables, def-use chain, reaching definition, alias analysis etc.), class hierarchy analysis and dependence analysis. Representative program translations developed with ROSE are partial redundancy elimination, constant folding, inlining, outlining (separating out a portion of code as a function), and loop transformations (a loop optimizer supporting aggressive loop optimizations such as fusion, fission, interchange, unrolling and blocking).

The unparser serves as a code-generator for ROSE. It unparses the AST to generate C/C++ and Fortran source code. Users can direct the unparser to unparse all included header files or only the original source files. This control is desired when new user-defined data types are added during a transformation. The unparser can also be invoked as needed during a transformation. It converts the entire AST or a portion of it into corresponding source code. This feature helps developers to debug a transformation in multiple stages. Finally, a vendor compiler is optionally called to continue the compilation of the generated (optimized) source code; generating a final executable.

The unique features of ROSE have attracted more than three dozen users worldwide for various research, development and education purposes. For example, internal users at Lawrence Livermore National Laboratory are using it for optimizing data structure abstractions and components. External users from other national laboratories, universities, and companies have been using ROSE to support code instrumentation, static analysis, formal rewrite systems, empirical tuning, MPI verification, etc.

3 Domain-Specific Abstractions

General purpose languages typically permit the construction of abstractions; represented by functions, data structures, etc. These permit high level representations of typically user-defined concepts. C++, as an object-oriented language, supports more complex abstractions and encourages the use of classes, member functions, templates, etc. ROSE uses a high level IR which permits the high fidelity representation of such abstractions without loss of precision. As a result, program analysis can see all the details of the use of the high-level abstractions typically lost in a lower level IR.

Knowledge of the semantics of the abstractions can be a short-cut for program analysis focused on the analysis of the implementation of an abstraction. In the case of complex abstractions with semantics hidden behind the use of pointers, leveraging known or published semantics of the abstractions can often be more productive. Since ROSE does not lower the high level representation to lose details of the use of specific abstractions in applications, the context of their use can be combined with their known semantics to provide fundamentally more information than could be known from static analysis alone. Leveraging this information is what permits higher levels of optimization for applications using such knowledge about high-level abstractions.

As an example, the knowledge that STL vectors are contiguous in memory is critical to numerous optimization opportunities of the STL vector container abstraction. Yet the specification of a vector abstraction would not require such implementation-specific detail. That it is specified in the standard is knowledge that can be leveraged, but might be impossible to obtain from an analysis of a specific STL implementation because of the complexity of its internal pointer handling. The efforts to optimize the abstraction are counter productive to the discovery via static analysis. Worse yet, a required conservative approach would have to prove such a property. In contrast our optimizations, can leverage the fact from the specification, and assume a compliant implementation.

4 Automatic Parallelization

In this section, we describe the parallelization methods we use for building a parallelizer to automatically generate OpenMP directives for C++ applications using high level object-oriented data types, including STL containers. We focus on loops because they are often the most time-consuming part in scientific applications.

We propose the following algorithm to parallelize loops for an input application. The loops may contain variables of either primitive data types or STL container types, or both.

1. Traverse AST and build a loop hierarchy.
2. Identify candidate loops based on their forms and computation requirements.
3. For each selected loop, do the following tasks:
   (a) Perform dependence analysis on variable references, including scalars and arrays of both primitive types and STL types.
   (b) Compute liveness information for variables referenced.
   (c) Judge the safety of parallelization and decide on variable classifications.
   (d) Annotate the loop with parallelization information if it is parallelizable.
   (e) Insert OpenMP directives based on the parallelization annotations.

The parallelizer starts by building a loop hierarchy at points in the AST in order to facilitate the traversal of loops under investigation. A loop candidate selection phase immediately follows to avoid unnecessary, expensive program analyses for automatic parallelization. The selection phase uses two main criteria to see if a loop is worth consideration: whether it is a canonical for loop and whether it has enough computation work to justify possible parallelization efforts. We define a canonical *for loop* as a loop with a format of either **for** (i = lb; i < ub; i += stride) or **for** (i = ub; i > lb; i −= stride). Statically estimating computation cost [8] to justify parallelization with OpenMP, namely profitability analysis, can be a very complicated process. A simple estimation method is adopted in our algorithm since we only need profitability analysis to preselect the candidate loops. The method is based on a cost table mapping computational AST nodes and their estimated execution cost. A loop traversal accumulates the cost of all nodes per loop iteration and multiplies the cost with the iteration count to get the final estimation. An alternative profitability analysis could be profile-guided: leveraging the persistent attribute mechanism, ROSE is able to annotate the AST with arbitrary performance information collected by external performance tools like HPCToolkit [9]. The parallelizer can directly retrieve the computation requirements of loops from performance information obtained from past executions. A configurable number is provided as a threshold to decide whether the loop warrants parallelization.

The essential work of the parallelizer relies on dependence analysis and liveness analysis. Both analyses are actually conducted at the function level so loops within the same function can share the results. The dependence analysis gives dependence information for variable referenced (excluding loop induction variables) in the loop. But this information alone is not enough for aggressive parallelization using OpenMP because some of the loop-carried dependences can be eliminated by using data-sharing clauses provided by OpenMP. Therefore, we also conduct variable classification based liveness analysis to exclude certain loop-carried data dependences that would otherwise prohibit safe parallelization. With the final dependence and variable classification results, the parallelizer attaches parallelization information as persistent attributes to the loop without any loop-carried dependences

The final work of the parallelizer is to generate the actual OpenMP directives for loops guided by the associated persistent attributes for the parallel information of each loop. We give more details of the parallelizer in the following subsections.

4.1 Dependence Analysis

Dependence analysis is the basis for the parallelizer to decide whether a loop is parallelizable. ROSE conducts dependence analysis and builds dependence graphs in the loop optimizer, which implements algorithms proposed in [10, 11] to effectively transform both perfectly nested loops and non-perfectly nested loops. An extended direction matrix (EDM) dependence representation is used to cover non-common loop nests that surround only one of the two statements in order to handle non-perfectly nested loops. For array accesses within loops, a Gaussian elimination algorithm is used to solve a set of linear integer equations of loop induction variables.

Figure 2 gives an example dependence graph dump for an input code, in which a statement is surrounded by two loops ($commonlevel = 2$). Two true dependence relations exist, caused by two pairs of array references and carried

```
  for (i=1;i<n;i++)
    for (j=1;j<m;j++)
      a[i][j]=a[i][j−1]+a[i−1][j];
/*
dep SgExprStatement @3−−> SgExprStatement @3
    2*2 TRUE_DEP; commonlevel = 2 CarryLevel = 1
    SgPntrArrRefExp:a[i])[j] @3:14−>SgPntrArrRefExp:a[i][j − 1] @3:19
     == 0; * 0;
     * 0; == −1;

dep SgExprStatement @3−−> SgExprStatement @3
    2*2 TRUE_DEP; commonlevel = 2 CarryLevel = 0
    SgPntrArrRefExp:a[i][j]@3:14−>SgPntrArrRefExp:a[i − 1][j]@3:31
    == −1; * 0;
    * 0 ; == 0;
*/
```

Fig. 2. An example output of ROSE's dependence graph

in both loop levels ($CarryLevel = 0$ and $CarryLevel = 1$). The extended direction matrices give the dependence directions(one of $=$, $\leq$,$\geq$, and $*$) and alignment factors. The details of the dependence analysis and corresponding graph can be found in [10,11]. It is clear from the dependence analysis that the example code in Fig. 2 cannot be parallelized because of loop-carried dependences in both loop levels.

Extending the existing dependence analysis in ROSE to handle STL containers is straightforward because full C++ type information is preserved in AST. The key is to tell if some function calls are semantically equivalent to a subscripted element access of an array-like object. For simple cases like processing std::vectors of integers, checking the qualified type and function names associated with a function call in the AST will be sufficient to find an eligible STL container reference that can be treated as a subscripted array element access in the parallelizer. Dependence analysis can then also be easily carried out. However, given the complexity of user applications and limitations of static compiler analysis, additional information is often used to help the parallelizer to recognize array access semantics buried in high level user-defined abstractions and to enable the dependence analysis.

We leverage the array abstraction annotations [12] previously done in ROSE to aid in recognizing complex array-like containers and their corresponding element access member functions. In addition, side effects of function calls and alias information are also represented using source code annotations to facilitate conducting dependence analysis on loops with function calls and pointer references. Two types of annotations are used: one for class declarations and the other for member functions and regular functions. We use class annotations such as is-array and inheritable to indicate a user-defined STL container has array-like semantics and such semantics can be inherited. Function annotations express if a member function is an element access function and give lists of variables that may be modified, read, or aliased inside a function. A future side-effect analysis could eliminate some or all of this step.

4.2 OpenMP Variable Classification

OpenMP loop variable classification is used by the parallelizer to figure out which data-sharing attribute clause should be used for variables referenced inside a possible parallelizable loop. It is also used to exclude certain loop-carried dependences which would otherwise prevent possible parallelization.

Our variable classification is based on the classic liveness analysis [13], which decides if a variable may be used in the future at a certain position in the code. Table 1 shows the categories of data-sharing attributes for variables based on their live-in (before the execution of a loop) and live-out (after the execution of a loop) analysis results. For instance, a private variable inside a loop is neither live-in nor live-out of the loop, which means the variable is immediately killed (redefined) inside the loop and then used inside the loop somehow, but is never going to be used anywhere after the loop. All loop index variables are also classified as OpenMP private variables to a avoid possible race condition. On the other hand, shared variables are live at both the beginning and the end of the loop. **Firstprivate** and **lastprivate** variables are live at either only the beginning or only the end of the loop, respectively.

Table 1. OpenMP variable classification based on liveness analysis

Data-sharing attribute	**Live-in**	**Live-out**
shared	Yes	Yes
private	No	No
firstprivate	Yes	No
lastprivate	No	Yes

Reduction variables are handled specially to maximize the opportunities for parallelization. A typical reduction operation inside a loop, such as sum = sum + a[i], causes a loop-carried output dependence, a loop-carried anti-dependence, and a loop independent anti-dependence. We use an idiom recognition analysis to capture such typical operations and exclude the associated loop-carried dependences when deciding if a loop is parallelizable.

4.3 Representing Parallelization Information

The persistent AST attribute mechanism in ROSE is used to represent parallelization information associated with OpenMP directives.

Figure 3 gives an excerpt of the C++ type we use to store OpenMP information in the AST. The information of an OpenMP directive is represented by an instance of OmpAttribute, which is a subclass of the persistent attribute AstAttribute. OmpAttribute contains all possible information for an OpenMP directive, including the directive type (**omp parallel**, **omp for**, **omp section**, **omp master**, etc.), a list of referenced variables (stored as OmpSymbols for variables in clauses such as private, shared, and reduction and so on), and others (hasNowait, schedule type and chunk

```
class OmpAttribute:public AstAttribute
{
public:
  omp_construct_enum omp_type;
  Rose_STL_Container < OmpSymbol * >var_list;
  bool hasLastprivate;
  bool hasFirstprivate;
  bool hasReduction;
  bool hasNowait;
  bool hasOrdered;
  bool isOrphaned;
  omp_construct_enum sched_type;
  SgNode *chunk_size;
//...
}
```

Fig. 3. Representing OpenMP semantics using an AST persistent attribute.

size, etc.). All variables appearing in a directive are kept in a list of OmpSymbols, which associate the variables and the corresponding OpenMP semantics such as that variable x is a **threadprivate** variable. It is possible to have one variable with two OmpSymbols in the list because a variable may appear in two OpenMP clauses, such as **firstprivate** and **lastprivate**.

OmpAttributes are designed to be attached to a structured block enclosed by an OpenMP directive. In addition to supporting automatic parallelization, attributes are also used to store information from an OpenMP directive parser processing C/C++ pragmas or Fortran comments. As a result, the later OpenMP translation has a uniform view of the AST and associated parallelization information for either manually coded or automatically generated OpenMP programs.

5 OpenMP Translation

The automatically generated OpenMP code can be either directly output to users or further translated into multithreaded code by ROSE. An OpenMP translator has been implemented in ROSE to enable streamlined processing from automatic OpenMP insertion to OpenMP translation. It also acts as a standalone OpenMP implementation when the input source applications already have user-introduced OpenMP directives.

As shown in Fig. 4, the ROSE OpenMP translator consists of two phases: a top-down AST processing phase (OmpFrontend) to collect and propagate OpenMP directive information down through the AST and a bottom-up AST processing phase (OmpMidend) to conduct the actual transformations for each type of OpenMP construct. We provide our own OmpFrontend to parse OpenMP directives because neither the EDG front-end nor Open Fortran Parser supports OpenMP. With the clear interface to represent OpenMP using OmpAttribute, OmpFrontend can be easily replaced when the actual language frontends become able to parse OpenMP. Also, OmpMidend is able to translate the AST annotated with OmpAttributes generated from either OmpFrontend or the parallelizer. The Omni OpenMP compiler's runtime library [14] is used to provide low level library calls

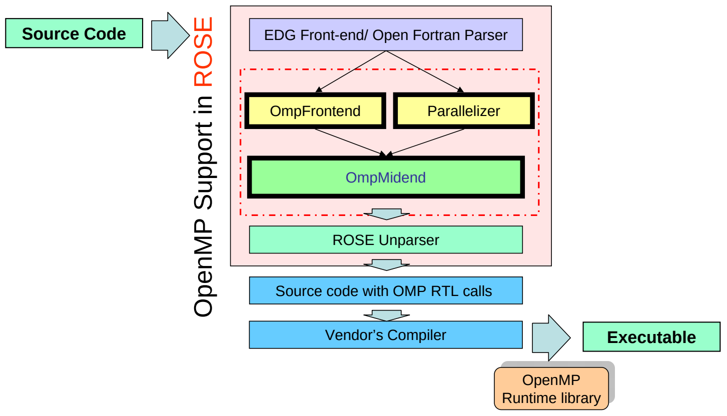

Fig. 4. The ROSE OpenMP translator

generated by the ROSE OpenMP translator; the translation framework can be changed to work with other runtime systems as necessary.

OmpMidend is divided into two phases: a preprocessing phase and a translation phase. The preprocessing phase is responsible for loop normalization and directive processing. Loop normalization converts all parallelizable loops into canonical forms to simplify the loop rewriting needed for OpenMP translation and the connection between translation and the runtime support (especially the loop scheduling support). Combined OpenMP directives such as **omp parallel for** and **omp parallel sections** are split into separated directives in order to reuse the individual translators for each types of OpenMP directives in the translation phase as much as possible. Some implicit OpenMP constructs, such as the first **omp section**, are made explicit. Semantic checks for OpenMP constructs can also be done in this phase.

To achieve maximum portability, an outlining translation strategy is adopted in the translation phase of OmpMidend, in which a parallel region is outlined to a separate function and that function is passed to an OpenMP runtime function to spawn new threads. Fig. 5 gives the code snippet translating a parallel region. It starts with retrieving the OmpAttribute attached to the structured block enclosed by a parallel region. Then a function skeleton is generated for an outlined function. Various variable declarations for shared, private and other OpenMP variables are inserted into the outlined function's body, followed by a block of statements copied from the original structured block. The next step is to replace variable references inside the copied block with new references to the newly declared variables. Finally, the completed outlined function is inserted to the AST and the structure block is replaced with the runtime function call spawning threads.

Translations for other OpenMP directives involve similar operations on the AST tree including attribute retrieval, AST traversal, building, copying, in-

```
void OmpMidend::transParallelRegion(SgNode* block)
{
  AstAttribute* astattribute=block->getAttribute("OmpAttribute");
  OmpAttribute * ompattribute=
     dynamic_cast<OmpAttribute* > (astattribute);
  // ...
  //Build an outlined function
  SgFunctionDeclaration * func = buildDefiningFunctionDeclaration \
        (func_name, func_return_type,parameterList,globalscope);
  SgBasicBlock * body = func->get_definition()->get_body();
  addSharedVarDeclarations(ompattribute, body);
  addPrivateVarDeclarations(ompattribute, body);
  addThreadprivateDeclarations(ompattribute,body);
  // copy the statements inside the original block
  SgBasicBlock *bBlock2 = buildBasicBlock();
  appendStatement(bBlock2,body);
  deepCopy(block, bBlock2);
  variableSubstituting(ompattribute, bBlock2);
  if(ompattribute->hasReduction)
    addReductionCalls(ompattribute,body);
  insertOutlinedFunction(block, outFuncDecl);

  //Replace the original block with a runtime library call
  SgBasicBlock *rtlCall=
      generateParallelRTLcall(block, func, ompattribute);
  replaceBlock(block, rtlCall);
}
```

Fig. 5. Translating a parallel region

sertion, and replacement. The details of the translations are similar to other OpenMP implementations [14, 15] and are out of the scope of this paper. The rich interfaces provided by ROSE largely ease the work of manipulating the AST by transparently taking care of many low level side effects occurring during code transformation, including those associated with symbol tables, parent and scope edges, and preprocessing information.

6 Preliminary Results

As this work is an ongoing project, we present some preliminary results in this section. Several computation kernels in C and C++ were chosen to test the automatic parallelization and the OpenMP translation. Figure 6 gives speedup of a C Jacobi program (4-point stencil using array copying in each iteration) and a C++ vector 2-norm distance calculation ($\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$) program. We ran the experiments on a dual processor Dell Precision T5400 workstation. Each processor (Intel Xeon X5460) has four cores running at 3.16 GHz, for a total of eight cores. GCC 4.1.2 was used as the backend compiler without using any optimization flags. The vector calculation used 100 million elements and the Jacobi iteration used a 500x500 double precision array. As shown in Fig. 6, our algorithm is able to recognize the parallelizable loops using either C primitive arrays or STL vectors. The OpenMP translation can also generate correct multithreaded code and achieve scalable performance.

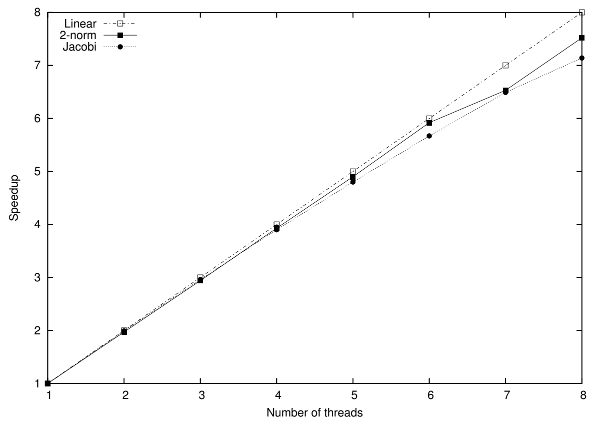

Fig. 6. Speedup of two example programs

We have implemented OpenMP 2.5 for C and C++ using ROSE. Fortran support is also being added. The implemented OpenMP constructs include **omp parallel**, **omp for**, **omp sections**, data environment clauses (**private**, **firstprivate**, **lastprivate**, **threadprivate**, **reduction**, **copyin**), **omp master**, **omp atomic**, **omp critical**, **omp flush**, etc. Mainstream OpenMP benchmarks such as the OpenMP validation suite [16], EPCC microbenchmark [17] and NAS parallel benchmarks [18] have been used to verify the correctness and robustness of the ROSE OpenMP translator.

7 Related Work

Numerous research compilers have been developed to support automatic parallelization. For example, the Vienna Fortran compiler (VFC) [19] is a source-to-source parallelization system for an optimized version of High Performance Fortran. PARADIGM [20] focuses on parallelization on distributed-memory multicomputers. The Polaris compiler [21] is mainly used for improving loop-level automatic parallelization [22]. The SUIF compiler [23] was designed to be a parallelizing and optimizing compiler supporting multiple languages. However, to the best of our knowledge current research parallelizing compilers only support Fortran and/or C applications. Commercial parallelizing compilers like the Intel C++/Fortran compiler [2] also use OpenMP internally as a target for automatic parallelization. Our work in ROSE aims to complete existing compilers by providing a multiple-language, source-to-source, parallelizing compiler infrastructure.

Several papers in the literature present parallelization based on C++ Standard Template Library (STL). The Parallel Standard Template Library (PSTL) [24] uses parallel iterators and provides some parallel containers and algorithms. The Standard Template Adaptive Parallel Library (STAPL) [25] is a superset of the C++ STL. It supports both automatic parallelization and user specified parallelization policies with several major components, including parallel containers (pContainers) and algorithms (pAlgorithms), randomly accessible data ranges (pRange), a distributor for data distribution, a scheduler to enforce data dependence, and an executor. GCC 4.3's runtime library (libstdc++) provides an experimental parallel mode, which implements an OpenMP version of many C++ standard library algorithms [26, 27]. However, all library-based parallelization methods require users to make sure that their applications are parallelizable. Our work tries to extend conventional compiler analysis to automatically ensure the safety of parallelization with the help of optional source code annotations.

A number of research compilers support OpenMP. Most of them adopt a similar source-to-source translation approach using outlining. For example, Omni [14] is a popular source-to-source translator supporting C/Fortran 77 with a portable OpenMP runtime library based on POSIX and Solaris threads. OdinMP/CCp [28] is another source-to-source translator with only C language support. PCOMP [29] contains an OpenMP parallelizer and a translator for Fortran 77. OpenUH [15] is one of the few research compilers with backends. It is based on the Open64 [30] compiler and also has limited source-to-source translation capability. ROSE is a unique source-to-source compiler infrastructure because it handles C++ applications with OpenMP.

Some previous research [31, 32] has initially explored OpenMP implementation in ROSE and the parallelization opportunities using the high-level semantics of A++/P++ libraries and user-defined C++ containers. This paper presents a more robust OpenMP implementation in ROSE, considers more generic C++ applications using the Standard Template Library, and incorporates dependence analysis to further broaden the applicable scenarios.

8 Conclusions and Future Work

In this paper, we have presented automatic parallelization for C++ applications using the ROSE source-to-source compiler infrastructure. Our work uses OpenMP as a target for parallelization and considers STL container types in C++ applications in addition to conventional primitive data types as in C or Fortran. A full-featured OpenMP translation has also been provided to enable wider OpenMP development and research using ROSE. The unique features available in ROSE, such as the rich AST manipulation interfaces and abundant generic compiler analyses, have provided excellent support for this work. Our work extends the applicability of automatic parallelization and provides another way to exploit multicore processors.

Importantly, the explicit use of OpenMP can be at conflict with loop optimizations, because OpenMP translation happens in a preliminary step before program analyses that would permit significant loop restructuring (in later phases of the compilation). We expect that the automated insertion of OpenMP directives is strategic since a source-to-source approach permits us to delay the OpenMP specific translation until after the source-to-source loop optimization. In conventional OpenMP compilers the OpenMP translation precedes the later phases of compilation where the program analysis is done and loop transformations are implemented. Conventional loop fusion is more feasible when applied to the loop structure before translation using OpenMP. Loop optimizations combined with OpenMP is thus fundamentally more aggressive and we expect less likely to occur in explicitly annotated OpenMP code. We anticipate this level of optimization is important for multicore optimizations because the loop optimizations are required to support reuse of variables for cache optimization. Without the loop optimizations ahead of the OpenMP translation, the OpenMP-translated loops would be translated to forms that would make their fusion significantly more complex. We expect that future work will layout the advantages and disadvantages of being able to stage source-to-source loop optimization ahead of the OpenMP translation.

References

1. : The OpenMP specification for parallel programming. `http://www.openmp.org` (2008)
2. Bik, A., Girkar, M., Grey, P., Tian, X.: Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. Intel Technology Journal **5** (2001)
3. Johnson, S.P., Evans, E., Jin, H., Ierotheou, C.S.: The ParaWise Expert Assistant — widening accessibility to efficient and scalable tool generated OpenMP code. In: WOMPAT. (2004) 67–82
4. : The ROSE compiler project. `http://www.rosecompiler.org/` (2008)
5. : Edison design group. `http://www.edg.com`
6. : Open fortran parser. `http://fortran-parser.sourceforge.net/`
7. Bodin, F., Beckman, P., Gannon, D., Gotwals, J., Narayana, S., Srinivas, S., Winnicka, B.: Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In: Proceedings of the Second Annual Object-Oriented Numerics Conference. (1994)
8. Liao, C., Chapman, B.: Invited paper: A compile-time cost model for OpenMP. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International (March 2007) 1–8
9. : HPCToolkit. `http://www.hipersoft.rice.edu/hpctoolkit/` (2008)
10. Yi, Q., Kennedy, K.: Improving memory hierarchy performance through combined loop interchange and multi-level fusion. Int. J. High Perform. Comput. Appl. **18**(2) (2004) 237–253
11. Yi, Q., Kennedy, K., Adve, V.: Transforming complex loop nests for locality. J. Supercomput. **27**(3) (2004) 219–264
12. Yi, Q., Quinlan, D.: Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In: The 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC). (2004)
13. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1986)
14. Sato, M., Satoh, S., Kusano, K., Tanaka, Y.: Design of OpenMP compiler for an SMP cluster. In: the 1st European Workshop on OpenMP (EWOMP'99). (September 1999) 32–39
15. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: an optimizing, portable OpenMP compiler. Concurrency and Computation: Practice and Experience **19**(18) (2007) 2317–2332

16. Müller, M.S., Niethammer, C., Chapman, B., Wen, Y., Liu, Z.: Validating OpenMP 2.5 for Fortran and C/C++. In: Sixth European Workshop on OpenMP, KTH Royal Institute of Technology, Stockholm, Sweden (October 2004)
17. Bull, J.: Measuring synchronization and scheduling overheads in OpenMP. In: the European Workshop of OpenMP (EWOMP'99), Lund, Sweden (September 1999)
18. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center (1999)
19. Benkner, S.: VFC: The Vienna Fortran Compiler. Scientific Programming **7**(1) (1999) 67–81
20. Banerjee, P., Chandy, J.A., Gupta, M., IV, E.H., Holm, J.G., Lain, A., Ramaswamy, D.J.P.S., Su, E.: The PARADIGM compiler for distributed-memory multicomputers. Computer **28**(10) (1995) 37–47
21. Padua, D., Eigenmann, R., Hoeflinger, J., Petersen, P., Tu, P., Weatherford, S., Faigin, K.: Polaris: A new-generation parallelizing compiler for MPP's. Technical Report 1306, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. and Dev. (june 1993)
22. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., Tu, P.: Advanced program restructuring for high-performance computers with Polaris. IEEE Computer **29**(12) (1996) 78–82
23. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.A.M., Tjiang, S.W., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: SUIF: An infrastructure for research on parallelizing and optimizing compilers. SIGPLAN Notices **29**(12) (1994) 31–37
24. Johnson, E., Gannon, D., Beckman, P.: HPC++: Experiments with the Parallel Standard Template Library. In: Proceedings of the 11th International Conference on Supercomputing (ICS-97), New York, ACM Press (July 1997) 124–131
25. An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N.M., Rauchwerger, L.: STAPL: An adaptive, generic parallel C++ library. In: Languages and Compilers for Parallel Computing (LCPC). (2001) 193–208
26. Putze, F., Sanders, P., Singler, J.: MCSTL: the Multi-Core Standard Template Library. In: PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2007) 144–145
27. Singler, J., Konsik, B.: The GNU libstdc++ parallel mode: software engineering considerations. In: IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering, New York, NY, USA, ACM (2008) 15–22
28. Brunschen, C., Brorsson, M.: OdinMP/CCp — a portable implementation of OpenMP for C. Concurrency—Practice and Experience **12**(12) (2000) 1193–1203
29. Min, S.J., Kim, S.W., Voss, M., Lee, S.I., Eigenmann, R.: Portable compilers for OpenMP. In: WOMPAT '01: Proceedings of the International Workshop on OpenMP Applications and Tools, London, UK, Springer-Verlag (2001) 11–19
30. : Open64, the open research compiler. `http://www.open64.net` (2008)
31. Quinlan, D., Schordan, M., Yi, Q., de Supinski, B.: A C++ infrastructure for automatic introduction and translation of OpenMP directives. In: Proceedings of the Worshop on OpenMP Applications and Tools (WOMPAT). Volume 2716 of LNCS., Springer-Verlag (June 2003) 13–25
32. Quinlan, D.J., Schordan, M., Yi, Q., de Supinski, B.R.: Semantic-driven parallelization of loops operating on user-defined containers. In: Workshop on Languages and Compilers for Parallel Computing. Volume 2958. (2003) 524–538