Extending Automatic Parallelization to Optimize High-Level Abstractions for Multicore

Conference Paper · June 2009

DOI: 10.1007/978-3-642-02303-3_3 · Source: DBLP

4 authors, including:

Chunhua Liao
Lawrence Livermore National Laboratory
52 PUBLICATIONS **578** CITATIONS
SEE PROFILE

Thomas Panas
Facebook
38 PUBLICATIONS **554** CITATIONS
SEE PROFILE

Extending Automatic Parallelization to Optimize High-Level Abstractions for Multicore *

Chunhua Liao[1], Daniel J. Quinlan[1], Jeremiah J. Willcock[2] and Thomas Panas[1]

[1] Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
{liao6,quinlan1,panas2}@llnl.gov
[2] Computer Science Department
Indiana University
Lindley Hall Room 215
150 S. Woodlawn Ave.
Bloomington, IN, 47404
jewillco@osl.iu.edu

Abstract. Automatic introduction of OpenMP for sequential applications has attracted significant attention recently because of the proliferation 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 and complex user-defined types, are largely ignored due to the lack of research compilers that are readily able to recognize high-level object-oriented abstractions and leverage their associated semantics. In this paper, we automatically parallelize C++ applications using ROSE, a multiple-language source-to-source compiler infrastructure which preserves the high-level abstractions and allows us to unambiguously leverage their known semantics. Several representative parallelization candidate kernels are used to explore semantic-aware parallelization strategies for high-level abstractions, combined with extended compiler analyses. Those kernels include an array-based computation loop, a loop with task-level parallelism, and a domain-specific tree traversal. Our work extends the applicability of automatic parallelization to modern applications using high-level abstractions and exposes more opportunities 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 shared-memory parallelism supported by hardware.

* This work was 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 dependence analysis implementation in ROSE.

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.

OpenMP [1] is a simple and portable parallel programming model that 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.

Although numerous parallelizing compilers [2,3] and tools [4,5] have been presented during the past decades, most of them focus only on C and/or Fortran applications operating on primitive data types. On the other hand, object-oriented languages, especially C++, are widely used to develop scientific computing applications. Those applications are often written with various standard and/or user-defined high-level abstractions, such as those in the C++ Standard Template Library (STL), now part of the C++ standard. While high-level abstractions successfully hide their implementation details and are useful to users for this purpose, they significantly impede static code analyses applied to their complex implementation. Typically, significant information about the abstractions is lost during the compiler's lowering to a simple intermediate representation (IR). Thus, compilers are often forced to make conservative assumptions for applications using such abstractions and are not able to apply many optimizations, including automatic parallelization.

In this paper, we use a source-to-source compiler infrastructure, ROSE [6], to explore compiler techniques to recognize high-level abstractions and to exploit their semantics for automatic parallelization. Our goal is to automate the process of migrating existing sequential C++ applications to multicore machines and to assist in developing new parallel applications. Specifically, our work addresses the concerns of parallelism for three target audiences: 1) users with legacy code (C/C++) using standard abstractions (STL, etc.), 2) users and library writers with domain-specific abstractions that have semantic properties that match those of the ones we make available, 3) library developers who are developing domain-specific abstractions for users and leveraging the semantics using their own semantic specifications (ones that we don't define). Our work addresses the essential requirement that modern compilers be fundamentally extensible in a way that simplifies how domain-specific abstractions can be optimized.

The remainder of this paper is organized as follows. The ROSE compiler infrastructure is introduced in the next section. Section 3 discusses high-level abstractions and parallelization. Section 4 then presents the details of a semantic-aware parallelizer using ROSE. Preliminary results of our work are given in Section 5. Section 6 discusses related work. Finally, Section 7 presents our conclusions and the future directions of this work.

2 The ROSE Compiler Infrastructure

ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale C/C++ and Fortran applications. Since it preserves the representation of high-level abstractions, no required information to recognize such abstractions is lost and the associated semantics can be reliably inferred. ROSE allows even non-expert users to exploit compiler techniques to address the analysis and transformation of abstractions.

Fig. 1 illustrates a typical source-to-source translator built using ROSE. The Edison Design Group (EDG) front-end [7] is used to parse C and C++ applications. EDG source files and its IR are protected under commercial or research licenses, but may be distributed freely in binary form. Language support for Fortran 2003 (and earlier versions) is based on the open source Open Fortran Parser (OFP) [8] developed at Los Alamos National Laboratory. Using both EDG and OFP, ROSE presents a common object-oriented, open-source IR for C/C++ and Fortran. The ROSE IR includes an abstract syntax tree (AST), symbol tables, a control flow graph, etc. and is based loosely on the Sage++ IR design [9]. Also, a set of distributed symbol tables is associated with the AST tree to store symbols' information within each scope. Generic and custom program analysis and transformation can be built on top of the ROSE IR. The ROSE unparser generates source code in the original source language from the transformed AST, with all original comments and C preprocessor control structures preserved. Finally, a vendor compiler is optionally called to continue the compilation of the generated (transformed) source code, generating a final executable.

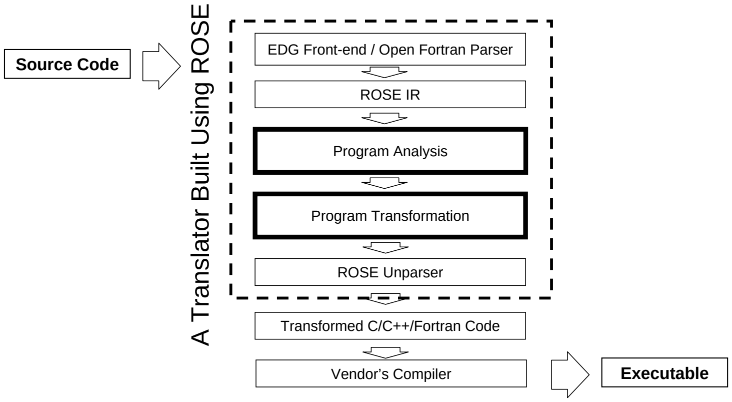

Fig. 1. A source-to-source translator built using ROSE

The ROSE AST, together with its corresponding symbol tables, fully supports type resolution, semantic analysis, and overloaded function resolution. All information in the application source code is preserved in the AST, including C preprocessor control structure, source comments, source position information, token stream (including whitespace), and C++ template information. The ROSE AST also has a rich set of interfaces for building source-to-source translators. These interfaces support efficient AST traversals, AST node queries, AST construction, copying, insertion, removal, and symbol table lookups. 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.

A number of program analyses and transformations have been developed for ROSE. They are designed to be utilized by users via calling simple function interfaces. The program analyses available include call graph analysis, control flow analysis, data flow analysis (def-use chain, reaching definition, live variables, 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).

3 High-Level Abstractions and Parallelization

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.

Knowledge of the semantics of the abstractions can be a short-cut for program analysis based on 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. As an example, the knowledge that STL vectors are contiguous in memory is critical to numerous optimization opportunities, but it might be impossible to obtain from an analysis of a specific STL implementation because of the complexity of its internal pointer handling. By exploiting well-defined semantics of high-level abstractions, compilers can significantly enhance the applicability and accuracy of existing analyses and optimizations. Such work also serves to encourage libraries to define abstractions with well-defined semantics. For instance, traditional parallelization algorithms designed for primitive data types can be extended to handle applications using high-level abstractions if the applications demonstrate similar semantic properties. The semantics of abstractions often directly indicate the side effects of function calls and such knowledge can significantly benefit parallelization which is often disabled because the inability to accurately summarize read and write accesses hidden behind call sites.

In the following subsections, we examine several typical candidates and explore parallelization strategies for applications using high-level abstractions.

3.1 An Array-Based Computation Loop

Loops operating on fixed-sized arrays are probably the most popular and representative examples for automatic parallelization using OpenMP. Typically, an array-based computation loop parallelizable by using **omp parallel for** has the following properties:

1. The loop has a canonical form (**for** (init; test; incr) block) which satisfies the requirements as defined by the OpenMP specification.
2. The loop operates on arrays using contiguous memory locations for a set of elements of the same type.
3. The elements of arrays do not overlap in memory or alias each other.
4. Random element accesses with a constant cost can be achieved by calculating offsets from an array base using subscripts.
5. The operations on the arrays do not rearrange the memory layout of elements and invalidate their accesses using subscripts across different iterations.
6. There are no loop carried data dependencies for array element accesses.

Conventional parallelization algorithms rely on a set of transformations and analyses in order to judge the safety of parallelization. For example, loop normalization is conducted to produce a canonical form, if possible. Alias analysis is used to tell if there are aliased elements. A set of data dependence tests based on array subscripts are used to determine if different loop iterations are independent. Automatic parallelization can be extended to handle high-level abstractions by leveraging their semantics and applying the conventional analyses and transformations. We take the following STL vector computation loop as an example to explore a viable parallelization method. The method is generic so that it can be applied to other high-level abstractions with similar semantics, including the STL deque or user-defined types.

```
std::vector <int> v1(100);
for (int i = 0; i < 100; i++)
  v1[i] = v1[i] + i;
```

The STL vector has many semantics (e.g., iterator invalidation rules) which can be taken advantage of by automatic parallelization. As a sequential container with contiguous storage for its elements, it supports random element access via both iterators and member functions (**operator**[] and at()). Although a vector can be reallocated or resized during its lifetime, it is quite common to have computation phases in which the vector participates in computations as if it was a fixed-sized primitive array. Within these phases, the arguments of random element access functions can be directly treated as array subscripts and passed to relevant parallelization analysis, especially array dependence analysis. The elements of the vector have to be verified to be alias-free and non-overlapping, either by compiler analyses or user annotations. Even for a loop using random access iterators, an extended loop normalization phase can convert the loop into a canonical form that is friendly to parallelization. For example, **for**(**vector**<T>::iterator i = v.begin(); i != v.end(); i++) can be transformed to size_t n = v.size(); **for** (size_t i = 0; i < n; i++). Dereferences of the iterator within the loop body can be replaced with equivalent element access function calls. In this case, all variable accesses like (*i) and i[n] are replaced with v[i](or v.at(i)) and v[i + n] (or v.at(i + n)) respectively according to the semantics defined in the language standard.

3.2 A Loop with Task-Level Parallelism

OpenMP 3.0 allows programmers to explicitly create tasks, which enable more parallelization opportunities, especially for algorithms applying independent tasks on non-random accessible data sets, or those using pointer chasing, recursion and so on. It is worthwhile to study how the semantics of high-level abstractions can facilitate parallelization targeting task level parallelism.

An example using the STL list is shown below as a typical candidate for parallelization using an **omp task** directive combined with an **omp single** within an **omp parallel** region:

```
for (std::list<myType>::iterator i = my_list.begin(); i != my_list.end(); i++)
  process(*i);
```

In order to parallelize the loop, a parallelization algorithm has to recognize the following program properties (a conservative case of parallelizable loops):

1. Whether the container supports random access, thus enabling the use of **omp for**; **omp task** is allowed in either case.
2. The elements in the container do not alias or overlap.
3. At most one element accessed via the loop index variable, we refer it as the *current* element, is written within each iteration (no loop carried output dependence among the elements).
4. The loop body does not read elements other than the current element if there is at least one write access to the current element (no loop carried true dependence or antidependence among the elements).
5. There are no other loop carried dependencies caused by variable references other than accessing the elements in the container.

A parallelization algorithm can significantly benefit from the known semantics of standard and user-defined high-level abstractions when dealing with a target mentioned above. It is essential that individual iterations of the loop be independent, substantial analysis is required to verify this. For instance, STL lists do not support random access. Knowing the usage of iterators will help identifying the loop index variable of non-integer types and is critical to recognize the reference to the current element by iterator dereferencing. Element accesses using other than dereferencing the index iterator, such as front() and back() can be conservatively treated as accesses to non-current elements. Many standard and custom functions have well-defined side effects on both function parameters and/or global variables. Therefore compilers can skip costly side effect analysis for those functions, such as size() and empty() for STL containers. Domain-specific knowledge can even be used to ensure the uniqueness of elements within a container to be processed as an alternative to conventional alias and pointer analysis. For example, a list of C function definitions returned by a ROSE AST query function has unique and non-overlapping elements.

3.3 A Domain-Specific Tree Traversal

We discuss a specific example from a static analysis tool, namely Compass [10], which is a ROSE-based framework for writing static code analysis tools to detect software defects or bugs. A typical Compass checker's kernel is given in Fig. 2. It is a visitor function to detect any error-prone usage of relational comparison, including $<$, $>$, $\leq$, and $\geq$, on pointers (MISRA Rule 5-0-18 [11]). A recursive tree traversal function walks an input code's AST and invokes the visitor function on each node. Once a potential defect is found, the AST node is stored in a list (output) for later display. Most functions (information retrieval functions like get_*() and type casting functions like isSg*()) used in the function body have read-only semantics.

```
 1  void CompassAnalyses::PointerComparison::Traversal::visit(SgNode* node)
 2  {
 3    SgBinaryOp* bin_op = isSgBinaryOp(node);
 4    if (bin_op)
 5    {
 6      if (isSgGreaterThanOp(node) || isSgGreaterOrEqualOp(node) ||
 7          isSgLessThanOp(node) || isSgLessOrEqualOp(node))
 8      {
 9        SgType* lhs_type = bin_op->get_lhs_operand()->get_type();
10        SgType* rhs_type = bin_op->get_rhs_operand()->get_type();
11        if (isSgPointerType(lhs_type) || isSgPointerType(rhs_type))
12          output->addOutput(bin_op);
13      }
14    }
15  }
```

Fig. 2. A Compass checker's kernel

Even with ideal side effect analysis and alias analysis, a conventional parallelization algorithm will still have trouble in recognizing the kernel as an independent task. The reason is that the write access (line 12) to the shared list will cause an output dependence among different threads, which prevents possible parallelization. However, the kernel's semantics imply that the order of the write accesses does not matter, which make this write access suitable to be protected using **omp critical**. Communicating such semantics to compilers is essential to eliminate the output dependence after adding the synchronization construct.

Another piece of semantic knowledge will enable an even more dramatic optimization. The AST traversal used by Compass checkers does not care about the order of nodes being visited. So it is semantically equal to a loop over the same AST nodes. The AST nodes are stored in memory pools, as in most other compilers [12]. The memory pools in ROSE are implemented as arrays of each type of IR node stored consecutively. Converting a recursive tree traversal into a loop over the memory pools is often beneficial due to better cache locality and less function call overhead. The loop is also more friendly to most analyses and optimizations than the original recursive function call; and importantly to this paper, can be automatically parallelized. In a more aggressive optimization, the types of IR nodes analyzed by the checker can be identified and only the relevant memory pools will be searched.

4 A Semantic-Aware Parallelizer

We design a parallelizer using ROSE to automatically parallelize target loops and functions by introducing either **omp for** or **omp task**, and other required OpenMP directives and clauses. It is designed to handle both conventional loops operating on primitive arrays and modern applications using high-level abstractions. The parallelizer uses the following algorithm:

1. Preparation and Preprocessing
   (a) Read a specification file for known abstractions and semantics.
   (b) Apply optional custom transformations based on input code semantics, such as converting tree traversals to loop iterations on memory pools.
   (c) Normalize loops, including those using iterators.
   (d) Find candidate array computation loops with canonical forms (for **omp for**) or loops and functions operating on individual elements (for **omp task**).
2. For each candidate:
   (a) Skip the target if there are function calls without known semantics or side effects.
   (b) Call dependence analysis and liveness analysis.
   (c) Classify OpenMP variables (autoscoping), recognize references to the current element, and find order-independent write accesses.
   (d) Eliminate dependencies associated with autoscoped variables, those involving only the current elements, and output dependencies caused by order-independent write accesses.
   (e) Insert the corresponding OpenMP constructs if no dependencies remain.

The key idea of the algorithm is to capture dependencies within a target and eliminate them later on as much as possible based on various rules. Parallelization is safe if there are no remaining dependencies. Semantics of abstractions are used in almost each step to facilitate the transformations and analyses, including recognizing function calls as variable references, identifying the current element being accessed, and ensuring if there are constraints for the ordering of write accesses to shared variables.

The custom transformation for optimizing the Compass checkers is trivial to implement in ROSE since the Compass checkers are derived from an AST traversal class to implement its capability of AST traversal. ROSE already provides AST traversal classes using either recursive tree traversal or loops over memory pools. Changing the checkers' superclass will effectively change the traversal method. Similar to other work [3], our variable classification is largely based on the classic live variable analysis and idiom recognition analysis to identify variables that could be classified as **private**, **firstprivate**, **lastprivate**, and **reduction**.

We give more details of the parallelizer and its handling of high-level abstractions in the following subsections.

4.1 Recognizing High-Level Abstractions and Semantics

ROSE uses a high-level AST which permits the high fidelity representation of both standard and user-defined abstractions in their original source code forms without loss of precision. As a result, program analyses have access to the details of high-level abstraction usage typically lost in a lower level IR. The context of those abstractions can be combined with their known semantics to provide fundamentally more information than could be known from static analysis alone.

Although semantics of standard types and operations can be directly integrated into ROSE to facilitate parallelization, a versatile interface is still favorable to accommodate semantics of user-defined types and functions. As a prototype implementation, we extend the annotation syntax proposed by [13] to manually prepare the specification file representing the knowledge of known types and semantics. A future version of the file will be expressed in C++ syntax to facilitate handling.

The original annotation syntax was designed to allow conventional serial loop optimizations to be applied on user-defined array classes. As a result, it only contains annotation formats for array classes to indicate if the classes are arrays (array) and their corresponding member access functions for array size (**length**()) and elements(**element**()). It also allows users to explicitly indicate read (**read**), written (**modify**), and aliased (**alias**) variables for class operations or functions to complement compiler analysis. We have extended the syntax to accept C++ templates in addition to classes. In particular, **is_fixed_sized_array** is used instead of array to make it clear that a class or template has a set of operations which conform to the semantics of a fixed size array, not just any array. Although standard or user-defined high level array abstractions may support some size changing operations such as resize(), those non-conforming operations are not included in the specification file and will be treated as unknown function calls. The semantic-aware parallelizer will safely skip the loop containing such function calls as shown in our algorithm. New semantic keywords have also been introduced to express knowledge critical to parallelization, such as **overlap**, **unique**, and **order_independent**.

An example specification file is given in Fig. 3. It contains a list of qualified names for classes or instantiated class templates with array-like semantics, and their member functions for element access, size query, and other operations preserving the relevant semantics. We also specify side effects of known functions, uniqueness of returned data sets, order-independent write accesses, and so on.

4.2 Dependence Analysis

We generate dependence relations for both eligible loop bodies and function bodies to explore the parallelization opportunities. We compute all dependence relations between every two statements s_1 and s_2, including the case when s_1 is equal to s_2, within the target loop body or function body. Each dependence relation is marked as local or thread-carried (either loop-carried and task-carried).

The foundation of the analysis is the variable reference collection phase, in which all variable references from both statements are collected and categorized

```
class std::vector<MyType> {
  alias none; overlap none; //elements are alias−free and non−overlapping
  is_fixed_sized_array { //semantic−preserving functions as a fixed−sized array
    length(i) = {this.size()};
    element(i) = {this.operator[](i); this.at(i);};
 };
};
void my_processing(SgNode∗ func_def) {
  read{func_def}; modify {func_def}; //side effects of a function
}
std::list<SgFunctionDef∗> findCFunctionDefinition(SgNode∗ root){
  read {root}; modify {result};
  return unique; //return a unique set
}
void Compass::OutputObject::addOutput(SgNode∗ node){
  //order−independent side effects
  read {node}; modify {Compass::OutputObject::outputList<order_independent>};
}
```

Fig. 3. A semantics specification file

into read and write variable sets. In addition to traditional scalar and array references, each member function call returning a C++ reference type is checked against the known high-level abstractions and semantics to see if it is semantically equivalent to a subscripted element access of an array-like object. An internal function, is_array(), is used to resolve the type of the object implementing the member function call and compare it to the list of known array types as given in the specification file. If the resolved type turns out to be an instantiated template type, its original template declaration is used for the type comparison instead. Consequently, is_element_access() is applied to the function call to check for an array element access and obtain its subscripts. Read and write variable sets of other known functions are also recognized and the affected variables are collected.

After that, a dependence relation is generated for each pair of references, r_1 from s_1's referenced variable set and r_2 from s_2's, if at least one of the references is a write access and both of them refer to the same memory location based on their qualified variable names or the alias information in the specification file. For array accesses within canonical loops, a Gaussian elimination algorithm is used to solve a set of linear integer equations of loop induction variables. The details of the array dependence analysis can be found in [14].

5 Preliminary Results

As this work is an ongoing project (the current implementation is released with the ROSE distribution downloadable from our website [6]), we present some preliminary results in this section. Several sequential kernels in C and C++ were chosen to test our automatic parallelization algorithm on both primitive types and high-level abstractions. They include a C version Jacobi iteration converted from [15] operating on a 500×500 double precision array, a C++ vector 2-norm distance calculation ($\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$) on 100 million elements, and a Compass checker (shown in Fig. 2 for MISRA Rule 5-0-18 [11]) applied on a ROSE source file (Cxx_Grammar.C) with approximately 300K lines of code. The generated OpenMP versions were compiled using our own OpenMP translator, which is a ROSE-based OpenMP 2.5 implementation targeting the Omni OpenMP runtime library [16]; thus, we do not have performance results for task parallelism (we are currently working on an OpenMP 3.0 implementation and we will also use other OpenMP 3.0 compilers as the backend compiler in the future). GCC 4.1.2 was used as the backend compiler with optimization disabled; optimization is not relevant because we are only showing that our algorithm can extract parallelism from high-level abstractions. We ran the experiments on a Dell Precision T5400 workstation with two sockets, each a 3.16 GHz quad-core Intel Xeon X5460 processor, and 8 GB memory.

Fig. 4 gives speedup of all the three test kernels after domain-specific optimization (optional) and parallelization compared to their original sequential executions. The results proved the efficiency of the semantic-driven optimization of replacing the tree traversal with a loop iteration for the Compass checker: a performance improvement of 35% of the one thread execution compared to the original sequential execution. Our algorithm was also able to capture the parallelization opportunities associated with both primitive data types and high-level abstractions. All tests showed near-linear speedup except for the Compass checker. The critical section within the checker's parallel region made a linear speedup impossible when 7 and 8 threads were used. More dramatic performance improvements can be obtained if only the relevant memory pools are searched but this step is not yet automated in our implementation.

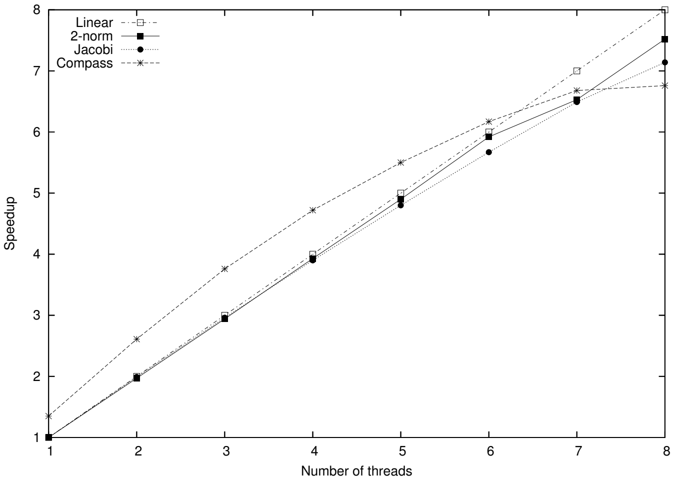

Fig. 4. Speedup of the three example programs after parallelization

6 Related Work

Numerous research compilers have been developed to support automatic parallelization. We only mention a few of them due to the page limit. For example, the Vienna Fortran compiler (VFC) [17] is a source-to-source parallelization system for an optimized version of High Performance Fortran. The Polaris compiler [2] is mainly used for improving loop-level automatic parallelization. The SUIF compiler [18] was designed to be a parallelizing and optimizing compiler supporting multiple languages. However, to the best of our knowledge current research parallelizing compilers largely focus on Fortran and/or C applications. Commercial parallelizing compilers like the Intel C++/Fortran compiler [3] also use OpenMP internally as a target for automatic parallelization. Our work in ROSE aims to complement existing compilers by providing a source-to-source, extensible parallelizing compiler infrastructure targeting modern object-oriented applications using both standard and user-defined high-level abstractions.

Several papers in the literature present parallelization efforts for C++ Standard Template Library (STL) or generic libraries. The Parallel Standard Template Library (PSTL) [19] uses parallel iterators and provides some parallel containers and algorithms. The Standard Template Adaptive Parallel Library (STAPL) [20] is a superset of the C++ STL. It supports both automatic parallelization and user specified parallelization policies with several major components for containers, algorithms, random access range, data distribution, scheduling and execution. GCC 4.3's runtime library (libstdc++) provides an experimental parallel mode, which implements an OpenMP version of many C++ standard library algorithms [21]. Kambadur et al. [22] proposes a set of language extensions to better support C++ iterators and function objects in generic libraries. However, all library-based parallelization methods require users to make sure that their applications are parallelizable. Our work automatically ensures the safety of parallelization based on semantics of high-level abstractions and compiler analyses.

Some previous research has explored code analyses and optimizations based on high-level semantics. STLlint [23] performs static checking for STL usage based on symbolic execution. Yi and Quinlan [13] developed a set of sophisticated semantic annotations to enable conventional sequential loop optimizations on user-defined array classes. Quinlan et al. [24, 25] presented the parallelization opportunities solely using the high-level semantics of A++/P++ libraries and user-defined C++ containers without using dependence analysis. This paper combines both standard and user-defined semantics with compiler analyses to further broaden the applicable scenarios of automatic parallelization. We also consider the new OpenMP 3.0 features and domain-specific optimizations.

7 Conclusions and Future Work

In this paper, we have explored the impact of high-level abstractions on automatic parallelization of C++ applications and designed a parallelization algorithm to take advantage of the capability of the ROSE source-to-source compiler infrastructure and the known semantics of both standard and user-defined abstractions. Though only three representative cases have been examined, our approach is very generic so that additional STL or user-defined semantics which are important to parallelization can be discovered and incorporated into our implementation. Our work demonstrates that semantic-driven parallelization is a very feasible and powerful approach to capture more parallelization opportunities than conventional parallelization methods for multicore architectures. Our approach can also be seamlessly integrated with conventional analysis-driven parallelization algorithms as a significant complement or enhancement.

In the future, we will apply our method on large-scale C++ applications to recognize and classify more semantics which can be critical to parallelization. We are planning to extend our work to support applications using more complex and dynamic control flows such as pointer chasing and use more OpenMP construct types. Further work also includes investigating the impact of polymorphism used in C++ applications, exploring the interaction between the automatic parallelization and conventional loop transformations, and leveraging semantics for better OpenMP optimizations as well as correctness analyses.

References

1. OpenMP Architecture Review Board: The OpenMP specification for parallel programming. `http://www.openmp.org` (2008)
2. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., Tu, P.: Parallel programming with Polaris. Computer **29**(12) (1996) 78–82
3. 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)
4. 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
5. Liao, S.W., Diwan, A., Robert P. Bosch, J., Ghuloum, A., Lam, M.S.: SUIF Explorer: an interactive and interprocedural parallelizer. In: PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, New York, NY, USA, ACM Press (1999) 37–48
6. Quinlan, D.J., et al.: ROSE compiler project. `http://www.rosecompiler.org/`
7. Edison Design Group: C++ Front End. `http://www.edg.com`
8. Rasmussen, C., et al.: Open Fortran Parser. `http://fortran-parser.sourceforge.net/`
9. Bodin, F., et al.: 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)
10. Quinlan, D.J., et al.: Compass user manual. `http://www.rosecompiler.org/compass.pdf` (2008)
11. The Motor Industry Software Reliability Association: MISRA C++: 2008 Guidelines for the use of the C++ language in critical systems. (2008)
12. Cooper, K., Torczon, L.: Engineering a Compiler. Morgan Kaufmann (2003)

13. 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)
14. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann (2001)
15. Robicheaux, J., Shah, S. `http://www.openmp.org/samples/jacobi.f` (1998)
16. 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
17. Benkner, S.: VFC: The Vienna Fortran Compiler. Scientific Programming **7**(1) (1999) 67–81
18. 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
19. 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
20. 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
21. 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
22. Kambadur, P., Gregor, D., Lumsdaine, A.: OpenMP extensions for generic libraries. In: International Workshop on OpenMP (IWOMP). (2008)
23. Gregor, D., Schupp, S.: STLlint: lifting static checking from languages to libraries. Softw. Pract. Exper. **36**(3) (2006) 225–254
24. 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
25. 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