Noname manuscript No.
(will be inserted by the editor)

Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions

Chunhua Liao · Daniel J. Quinlan · Jeremiah J. Willcock · Thomas Panas

the date of receipt and acceptance should be inserted later

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. Modern applications using high-level abstractions, such as C++ STL containers and complex user-defined class 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 use a source-to-source compiler infrastructure, ROSE, to explore compiler techniques to recognize high-level abstractions and to exploit their semantics for automatic parallelization. Several representative parallelization candidate kernels are used to study semantic-aware parallelization strategies for high-level abstractions, combined with extended compiler analyses. Preliminary results have shown that semantics of abstractions can help extend the applicability of automatic parallelization to modern applications and expose more opportunities to take advantage of multicore processors.

Keywords automatic parallelization · high-level abstractions · semantics · ROSE · OpenMP

C. Liao · D.J. Quinlan · T. Panas
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
E-mail: {liao6,dquinlan,panas2}@llnl.gov

J.J. Willcock
Computer Science Department
Indiana University
Lindley Hall Room 215
150 S. Woodlawn Ave.
Bloomington, IN, 47404
E-mail: jewillco@osl.iu.edu

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. 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 [21] 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 [5, 14] to use OpenMP as a target for interactive or automatic parallelization.

Although numerous parallelizing compilers [6, 33] and tools [20, 35] 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, modern object-oriented languages, especially C++, are widely used to develop scientific computing applications today. 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. High-level abstractions expose user-friendly interfaces and hide low-level details, therefore the use of abstractions can substantially enhance code reuse and accelerate programming productivity. 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 implementations. 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 [26], 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 (user-defined array classes, etc.) that have semantic properties that match those of the abstractions 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 do not define).

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 explores parallelization strategies for several representative kernels. Section 4 then presents the details of a semantic-aware parallelizer built 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 [22, 24, 30] is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale C/C++, Fortran, OpenMP and UPC applications. Given its fully type-resolved abstract syntax tree (AST), ROSE faithfully preserves the representation of high-level abstractions at the source level, no required information to recognize such abstractions is lost and the associated semantics can be reliably inferred. Using a source-to-source approach, ROSE complements existing vendor compilers by providing a fundamentally extensible way to simplify the optimization of standard and user-defined abstractions, thus helping achieving high performance without losing high productivity.

Fig. 1 illustrates a typical source-to-source translator built using ROSE. The Edison Design Group (EDG) front-end [11] is used to parse C (also UPC extensions) 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) [28] developed at Los Alamos National Laboratory. Leveraging both EDG and OFP, ROSE creates 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 [7]. Also, a set of distributed symbol tables is associated with the AST 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.

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

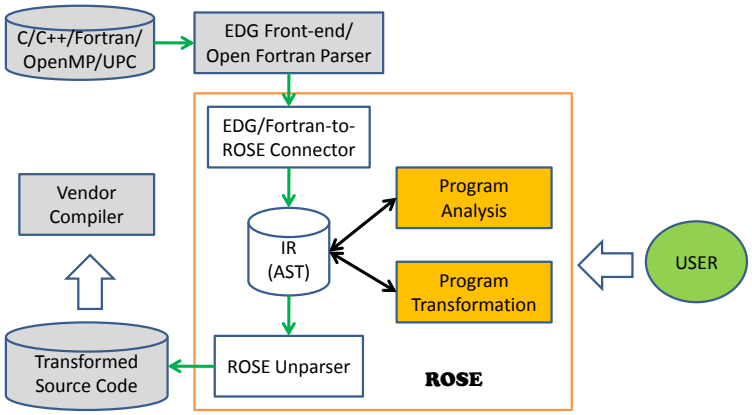

Fig. 1 A source-to-source translator built using ROSE

code analyzers and 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 [19] (separating out a portion of code as a function), and loop transformations [34] (a loop optimizer supporting aggressive loop optimizations such as fusion, fission, interchange, unrolling and blocking).

ROSE is released under a BSD-style license and is portable to Linux and Mac OS X on IA-32 and x86-64 platforms. Porting ROSE to Microsoft Windows platforms is currently a work in progress.

3 High-Level Abstractions and Parallelization

General purpose languages typically permit the construction of abstractions; represented by functions, data structures, etc. These permit representations of typically user-defined concepts. Modern object-oriented languages, such as C++, support more complex abstractions including classes, member functions, etc. High-level abstractions are designed to hide their complex implementation details and only conveniently expose simple user-friendly interfaces. Program-

mers today are encouraged to use high-level abstractions in order to reduce software complexity and improve programming productivity.

However, there is a common perception that using high-level abstractions often leads to inferior performance. The reason is that the exact information hiding mechanism of abstractions significantly impedes conventional compiler optimizations, including automatic parallelization, which rely on accurate static code analyses of low-level implementations.

On the other hand, abstractions are naturally associated with all kinds of standard or user-defined meanings, also called semantics. For instance, a **std**::set container means the stored objects are unique and sorted; the member function **list** :: size () has read-only semantics. Obviously, 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 and function calls, leveraging known or published semantics of the abstractions can often be more productive. As an example, the knowledge that elements in an **std**:: **list** are distinct 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 recognizing high-level abstractions and exploiting their well-defined semantics, 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 and satisfy the semantic constraints of the algorithms. 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** (initialization ; test ; increment) block) which satisfies certain 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 (also size).
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.[1] 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. The conventional automatic parallelization algorithms can be extended to handle high-level, array-like abstractions by leveraging their semantics and applying the conventional analyses and transformations extended to handle such abstractions. We take the following STL vector computation loop (shown in Fig. 2) as an example to explore a viable parallelization method of abstractions. 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 array-like types.

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

Fig. 2 A loop operating on an STL vector

The STL vector type 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

[1] We ignore the profitability analysis here as it can be treated as a relatively independent analysis and is out of the scope of this paper.

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.

In summary, based on a type-preserving IR and the knowledge about semantics associated with high-level abstractions, conventional parallelizing algorithms can be extended to conduct necessary analyses and transformations for eligible loops operating on any high-level abstractions demonstrating array-like semantics.

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 in Fig. 3 as a typical candidate for parallelization using an **omp task** directive combined with an **omp single** within an **omp parallel** region:

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

Fig. 3 A loop operating on list elements

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 applications using abstractions. 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 to identify the loop index variable which does not have an integer type 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 further discuss a specific example from a static analysis tool, namely Compass [27], which is a ROSE-based framework for writing static code analysis tools to detect software defects or bugs. Compass provides common functionalities needed for most static code analysis, including preparing necessary compiler analyses and AST traversal. In most cases, developers are only required to provide a visitor function that checks for defects or bugs based on an AST traversal and associated analysis results.

A typical Compass checker's kernel is given in Fig. 4. 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 [32]). 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. One important semantic constraint for Compass checkers is that they should not have side-effects on the input code's AST. Most functions (information retrieval functions like get_*() and type casting functions like isSg*()) used in the function body have read-only semantics.

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 makes this write access suitable to be protected using **omp critical**. Since there is no easy way to detect such semantics by existing compiler analyses, directly communicating such semantics to compilers is essential to eliminate the output dependence after adding the synchronization construct and finally make the function body thread-safe.

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

```
1  void
2  CompassAnalyses::PointerComparison::Traversal::visit(SgNode* node)
3  {
4    SgBinaryOp* bin_op = isSgBinaryOp(node);
5    if (bin_op)
6    {
7      if (isSgGreaterThanOp(node) || isSgGreaterOrEqualOp(node) ||
8          isSgLessThanOp(node) || isSgLessOrEqualOp(node))
9      {
10       SgType* lhs_type = bin_op->get_lhs_operand()->get_type();
11       SgType* rhs_type = bin_op->get_rhs_operand()->get_type();
12       if (isSgPointerType(lhs_type) || isSgPointerType(rhs_type))
13         output->addOutput(bin_op);
14     }
15   }
16 }
```

Fig. 4 A Compass checker's kernel

over the same AST nodes. The AST nodes are stored in memory pools, as in most other compilers [9]. 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 have been working on 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 (shown in Fig.5) 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 liveness analysis and dependence 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 current elements, and output dependencies caused by order-independent write accesses.
(e) Insert the corresponding OpenMP constructs if no dependencies remain.

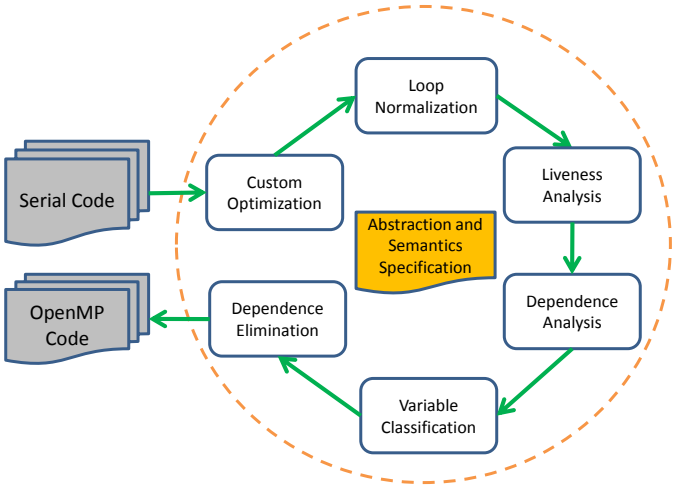

Fig. 5 A semantic-aware parallelizer built using ROSE

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 special conditions (explained in the step (d) of the algorithm). 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 on the ordering of write accesses to shared variables.

The custom transformation for optimizing the Compass checkers is straightforward 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 [5], 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 [34] 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 loops 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. 6. 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.

```
1  class std::vector<MyType> {
2    //elements are alias-free and non-overlapping
3    alias none; overlap none;
4    //semantic-preserving functions as a fixed-sized array
5    is_fixed_sized_array {
6      length(i) = {this.size()};
7      element(i) = {this.operator[](i); this.at(i);};
8   };
9  };
10
11 void my_processing(SgNode* func_def) {
12   //side effects of a function
13   read{func_def}; modify {func_def};
14 }
15
16 std::list<SgFunctionDef*> findCFunctionDefinition(SgNode* root){
17   read {root};  modify {result};
18   //return a unique set
19   return unique;
20 }
21
22 void Compass::OutputObject::addOutput(SgNode* node){
23   //order-independent side effects
24   read {node};
25   modify {Compass::OutputObject::outputList<order_independent>};
26 }
```

Fig. 6 A semantics specification file

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 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. ROSE's high level AST makes this work easy. For example, a reference to an STL vector element (v1[i]) in ROSE's AST is represented as a node of a function call expression (SgFunctionCallExp) with two children: a dot expression (SgDotExp) and an expression list of function parameter expressions (SgExprsListExp). The dot expression in turn has two children: a variable reference expression (SgVarRefExp) and a member function reference expression (SgMemberFunctionRefExp). An internal function, is_array(), is used to resolve

the type of the object (SgVarRefExp of the dot expression) 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 a member function reference expression (SgMemberFunctionRefExp) which is equal to an array element access and obtain its subscripts from the function's parameter list (SgExprsListExp). 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 [2].

5 Preliminary Results

As this work is an ongoing project (the current implementation is released with the ROSE distribution downloadable from our website [26]), 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. As shown in Table 1, they include a C version Jacobi iteration converted from [29] 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 120 million elements, a web server simulation kernel processing independent HTTP requests from a queue storing 1000 requests implemented using stl :: **list** (kernel is shown in Fig. 3), and a Compass checker (shown in Fig. 4 for MISRA Rule 5-0-18 [32]) applied on a ROSE source file (Cxx_Grammar.C) with approximately 293K lines of code (LOC).

Code	**Description**	**Language**	**Data Structure**	**Data Size**
Jacobi	Stencil computation	C	2-D array	500x500
2-norm	Distance calculation	C++	stl::vector	120 million
WebServer	Web server simulation	C++	stl::list	1K requests
Compass	Static code analysis	C++	User classes	293K LOC

Table 1 Sequential test kernels

With the help of the semantics specification file, the ROSE parallelizer could successfully parallelize all the test kernels using either primitive data types or complex abstraction types. Both the OpenMP loop construct (**omp for**) and task construct (**omp task**) could be introduced properly (as shown in

```
#define MSIZE 500
double u[MSIZE][MSIZE], f[MSIZE][MSIZE], uold[MSIZE][MSIZE];
/* .. code omitted ... */

#pragma omp parallel for private(j,resid) reduction(+:error)
for (i=1;i<(n-1);i++)
  for (j=1;j<(m-1);j++)
  {
    resid = (ax*(uold[i-1][j] + uold[i+1][j]) \
          + ay*(uold[i][j-1] + uold[i][j+1]) \
          + b * uold[i][j] - f[i][j])/b;

    u[i][j] = uold[i][j] - omega * resid;
    error = error + resid*resid ;
  }
```

Fig. 7 Parallelized Jacobi kernel

```
 std::vector <double> v1 (SIZE), v2 (SIZE);
 //vector initialization code omitted here ...
#pragma omp parallel for private(i) reduction(+:result)
  for (i=0;i<SIZE;i++)
    result=result + (v1.at(i)-v2.at(i))*(v1.at(i)-v2.at(i));
 result = sqrt (result);
```

Fig. 8 Parallelized vector 2-norm distance calculation kernel

```
// code omitted here ...
std::list<httpRequest> request_queue;
std::list<httpRequest>::iterator i;

#pragma omp parallel
{
#pragma omp single
  for (i = request_queue.begin(); i!=request_queue.end();i++)
  {
#pragma omp task
    process (*i);
  }
}
```

Fig. 9 Parallelized web server simulation kernel

Fig. 7– Fig. 10). The generated OpenMP versions were further compiled using our own OpenMP translator, which is a ROSE-based OpenMP 3.0 implementation targeting the GCC OpenMP runtime library (GOMP) [1]. Internally, our OpenMP translator invokes the ROSE outliner [19] to outline code portions to generate parallel tasks. GCC 4.4.1 was used as the backend compiler with an optimization option *-O3*. 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.

```
1  //A super class provides parallel traversal on memory pool
2  //This is manually prepared.
3  class parallelTraversal
4  {
5    public:
6      void traverse()
7      {
8  #pragma omp parallel for private(j) shared (memoryPool)
9        for (j=0;j<memoryPool.size();j++)
10         visit(memoryPool[j]);
11     }
12   protected:
13     virtual void visit(SgNode* node);
14 // code omitted here..
15 };
16
17 // It was made thread-safe after automatically inserting 'omp critical'.
18 // The original super class was replaced with parallelTraversal.
19 void
20 CompassAnalysis::PointerComparison::parallelTraversal::visit(SgNode* node)
21 {
22   SgBinaryOp* bin_op = isSgBinaryOp(node);
23   if (bin_op)
24   {
25     if (isSgGreaterThanOp(node)||isSgGreaterOrEqualOp(node)||
26         isSgLessThanOp(node)||isSgLessOrEqualOp(node))
27     {
28       SgType* lhs_type = bin_op->get_lhs_operand()->get_type();
29       SgType* rhs_type = bin_op->get_rhs_operand()->get_type();
30       if (isSgPointerType(lhs_type)||isSgPointerType(rhs_type))
31       {
32 #pragma omp critical
33         outputList.push_back(bin_op);
34       }
35     }
36   }
37 }
```

Fig. 10 Parallelized compass checker

Fig. 11 gives the speedups of all the test kernels after domain-specific optimization (optional) and parallelization compared to their original sequential executions. The results demonstrate that our semantic-aware parallelization algorithm is able to capture the parallelization opportunities associated with both primitive data types and high-level abstractions. In particular, the optimization of replacing the tree traversal with a loop iteration for the Compass checker directly contributed to a performance improvement of 35% of the single-thread execution compared to the original sequential execution. Automatic parallelization helped most tests to achieve linear or near-linear speedup, except for the 2-norm calculation. The critical section within the checker's parallel region made a linear speedup impossible when 7 and 8 threads were used. This is also true for Jacobi in which a reduction operation exists. One possi-

ble reason for the non-linear speedup of the 2-norm vector calculation is that its computation requirement is not large enough to scale up for more than 4 threads (It only took 0.3 seconds to finish the execution when 4 threads were used). Performance degradation happened when 5 or more threads were used. However, 120 million was already the biggest vector size we could use without causing GCC to send the **std**::bad_alloc exception. More dramatic performance improvements for the Compass checker can be obtained if only the relevant memory pools are searched but this step is not yet automated in our implementation.

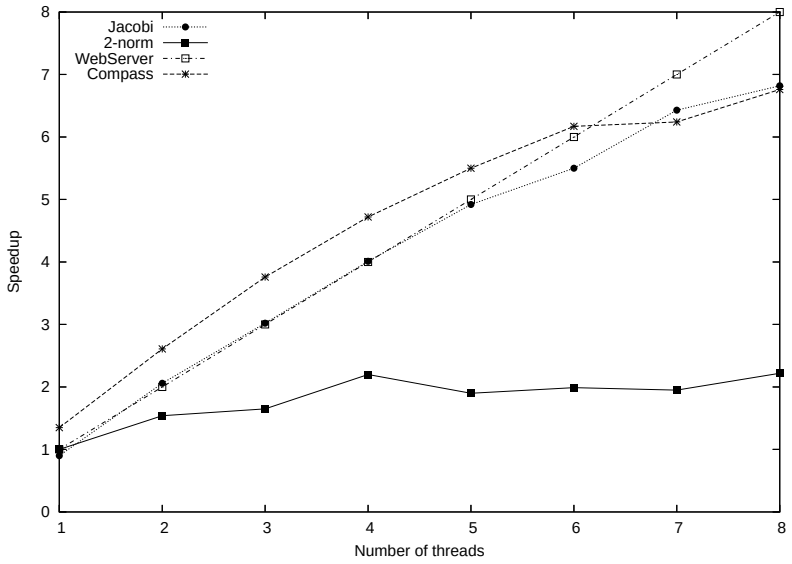

Fig. 11 Speedup of the example programs after parallelization

6 Related Work

Automatic parallelization of sequential code using compilers and tools has been pursued by researchers for several decades [10,18,35]. We only mention a few of parallelizing compilers and tools for brevity. The Vienna Fortran compiler (VFC) [4] is a source-to-source parallelization system for an optimized version of High Performance Fortran. The Polaris compiler [6] is mainly used for improving loop-level automatic parallelization. The SUIF compiler [33] was designed to be a parallelizing and optimizing compiler supporting multiple languages. Sophisticated interactive environments have also been created to integrate user knowledge (e.g. SUIF Explorer [20], Polaris [6] and Parawise [14]). However, to the best of our knowledge, current parallelizing compilers and tools

largely focus on Fortran and/or C applications. Commercial parallelizing compilers like the Intel C++/Fortran compiler [5] also use OpenMP internally as a target for automatic parallelization. Our work in ROSE aims to complement existing work 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) [13] uses parallel iterators and provides some parallel containers and algorithms. The Standard Template Adaptive Parallel Library (STAPL) [3] 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 [31]. Kambadur et al. [15] 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 for high-level abstractions. The Telescoping language project [8, 16] was aimed to develop a framework for automatically generating custom optimizing compilers for domain-specific languages and libraries. ROSE, on the other hand, uses a more pragmatic approach to allow average programmers to write customized analyses and optimizations for abstractions. Kulkarni et. ali [17] explored the use of abstractions and semantics, including un-ordered set, commutativity, inverse and so on for parallelization. They focused on language and runtime support for optimistic parallelization while we aim to extend the classic compiler-based parallelization. STLlint [12] performs static checking for STL usage based on symbolic execution. Yi and Quinlan [34] developed a set of sophisticated semantic annotations to enable conventional sequential loop optimizations on user-defined array classes. Quinlan et al. [23, 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 algo-

rithm 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-aware 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.

Acknowledgements 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.

References

1. GOMP – An OpenMP implementation for GCC. `http://gcc.gnu.org/projects/gomp` (2008)
2. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann (2001)
3. 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), pp. 193–208 (2001)
4. Benkner, S.: VFC: The Vienna Fortran Compiler. Scientific Programming **7**(1), 67–81 (1999)
5. 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)
6. 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), 78–82 (1996). DOI http://dx.doi.org/10.1109/2.546612
7. 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)
8. Broom, B., Cooper, K., Dongarra, J., Fowler, R., Gannon, D., Johnsson, L., Kennedy, K., Mellor-Crummey, J., Torczon, L.: Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. Journal of Parallel and Distributed Computing (2000)
9. Cooper, K., Torczon, L.: Engineering a Compiler. Morgan Kaufmann (2003)
10. Cooper, K.D., Hall, M.W., Hood, R.T., Kennedy, K., McKinley, K.S., Mellor-Crummey, J.M., Torczon, L., Warren, S.K.: The ParaScope parallel programming environment. Proceedings of the IEEE **81**(2), 244–263 (1993)

11. Edison Design Group: C++ Front End. `http://www.edg.com`
12. Gregor, D., Schupp, S.: STLlint: lifting static checking from languages to libraries. Softw. Pract. Exper. **36**(3), 225–254 (2006). DOI http://dx.doi.org/10.1002/spe.v36:3
13. 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), pp. 124–131. ACM Press, New York (1997)
14. 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, pp. 67–82 (2004)
15. Kambadur, P., Gregor, D., Lumsdaine, A.: OpenMP extensions for generic libraries. In: International Workshop on OpenMP (IWOMP) (2008)
16. Kennedy, K., Broom, B., Chauhan, A., Fowler, R., Garvin, J., Koelbel, C., McCosh, C., Mellor-Crummey, J.: Telescoping languages: A system for automatic generation of domain languages. Proceedings of the IEEE **93**(2), 387–408 (2005). DOI 10.1109/JPROC.2004.840447
17. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. Commun. ACM **52**(9), 89–97 (2009). DOI http://doi.acm.org/10.1145/1562164.1562188
18. Lamport, L.: The parallel execution of do loops. Commun. ACM **17**(2), 83–93 (1974). DOI http://doi.acm.org/10.1145/360827.360844
19. Liao, C., Quinlan, D.J., Vuduc, R., Panas, T.: Effective source-to-source outlining to support whole program empirical optimization. In: The 22th International Workshop on Languages and Compilers for Parallel Computing (LCPC). Newark, Delaware, USA (2009)
20. 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, pp. 37–48. ACM Press, New York, NY, USA (1999). DOI http://doi.acm.org/10.1145/301104.301108
21. OpenMP Architecture Review Board: The OpenMP specification for parallel programming. `http://www.openmp.org` (2009)
22. Quinlan, D.: ROSE: Compiler support for object-oriented frameworks. In: In Proceedings of Conference on Parallel Compilers (CPC) (2000)
23. 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), *LNCS*, vol. 2716, pp. 13–25. Springer-Verlag (2003)
24. Quinlan, D.J., Schordan, M., Miller, B., Kowarschik, M.: Parallel object-oriented framework optimization: Research articles. Concurr. Comput.: Pract. Exper. **16**(2-3), 293–302 (2004). DOI http://dx.doi.org/10.1002/cpe.v16:2/3
25. 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, vol. 2958, pp. 524–538 (2003)
26. Quinlan, D.J., et al.: ROSE compiler project. `http://www.rosecompiler.org/`
27. Quinlan, D.J., et al.: Compass user manual. `http://www.rosecompiler.org/compass.pdf` (2008)
28. Rasmussen, C., et al.: Open Fortran Parser. `http://fortran-parser.sourceforge.net/`
29. Robicheaux, J., Shah, S.: `http://www.openmp.org/samples/jacobi.f` (1998)
30. Schordan, M., Quinlan, D.: A source-to-source architecture for user-defined optimizations. In: JMLC'03: Joint Modular Languages Conference with EuroPar'03, *Lecture Notes in Computer Science*, vol. 2789, pp. 214–223. Springer Verlag (2003)
31. 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, pp. 15–22. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1370082.1370089
32. The Motor Industry Software Reliability Association: MISRA C++: 2008 Guidelines for the use of the C++ language in critical systems (2008)

33. 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), 31–37 (1994)
34. 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)
35. Zima, H.P., Bast, H., Gerndt, M.: Superb: A tool for semi-automatic MIMD and SIMD parallelization. Proceedings of Parallel Computing **6**(1), 1–18 (1988)