A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries

Chunhua Liao, Daniel J. Quinlan, Thomas Panas and Bronis R. de Supinski

Center for Applied Scientific Computing

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344

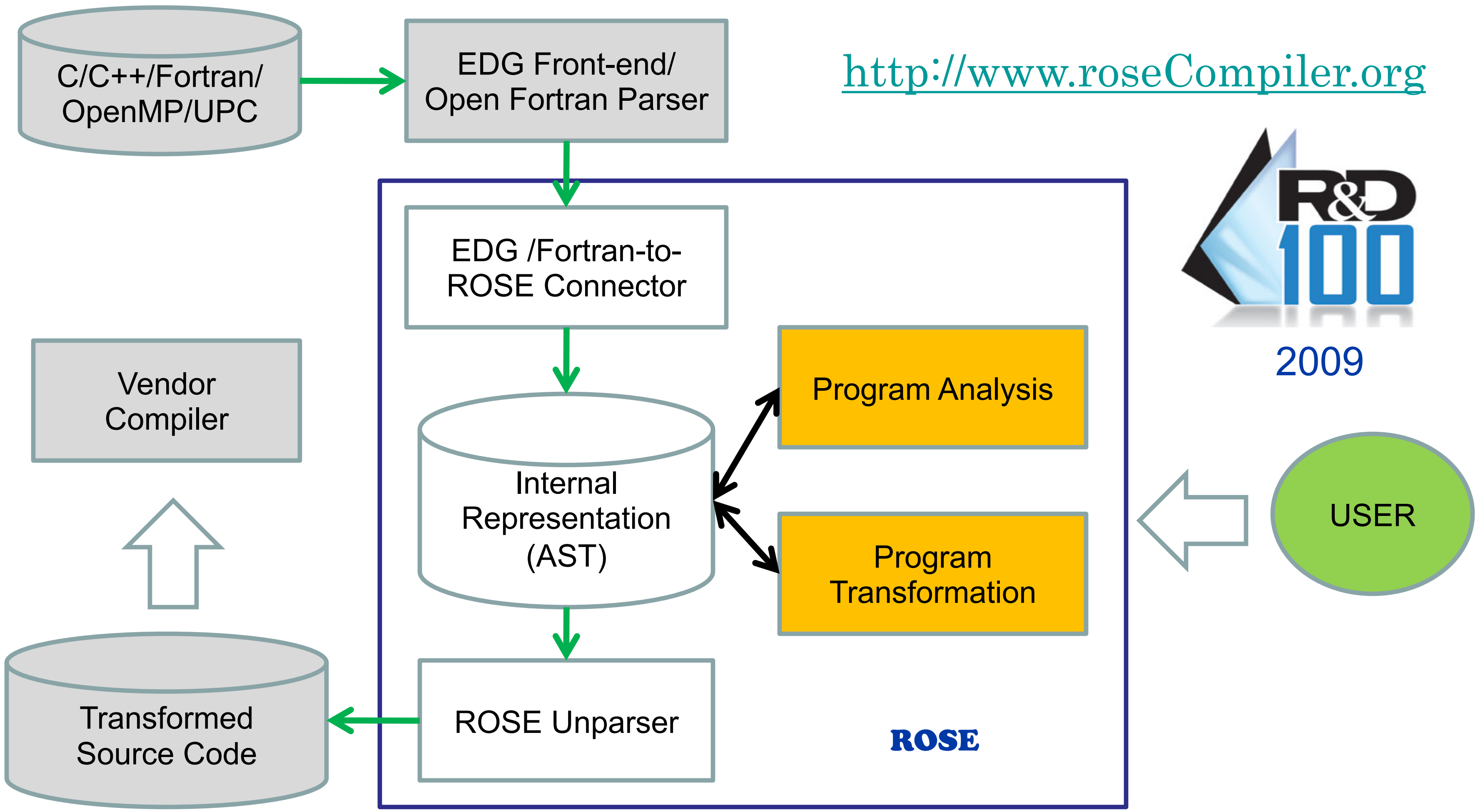
Agenda

- Motivation
- Overview of ROSE and its OpenMP support
- Parsing and representing OpenMP in ROSE
  - Facilitate static analysis for OpenMP
- Reusable translation for multiple runtime libraries
  - XOMP and preliminary results
- Future work

Motivation

- Build a source-to-source research compiler for ourselves and others
  - Support OpenMP 3.0
  - Support C/C++/Fortran
  - OpenMP representation for quickly building analyses
  - Enable runtime research with minimum changes to compiler, and vice versa

ROSE: making compiler technologies accessible



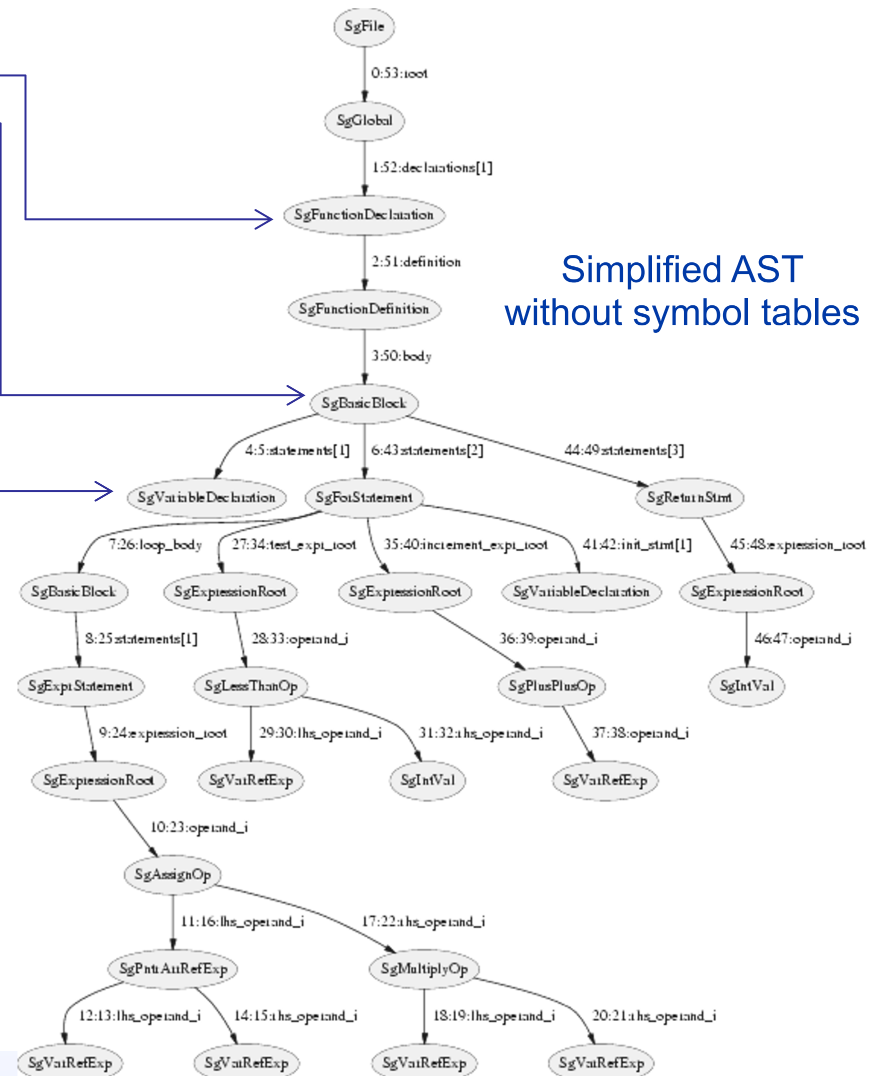
ROSE IR = AST + symbol tables + interface

```
int main()
{

 int a[10];

  for(int i=0;i<10;i++)
    a[i]=i*i;

  return 0;
}
```



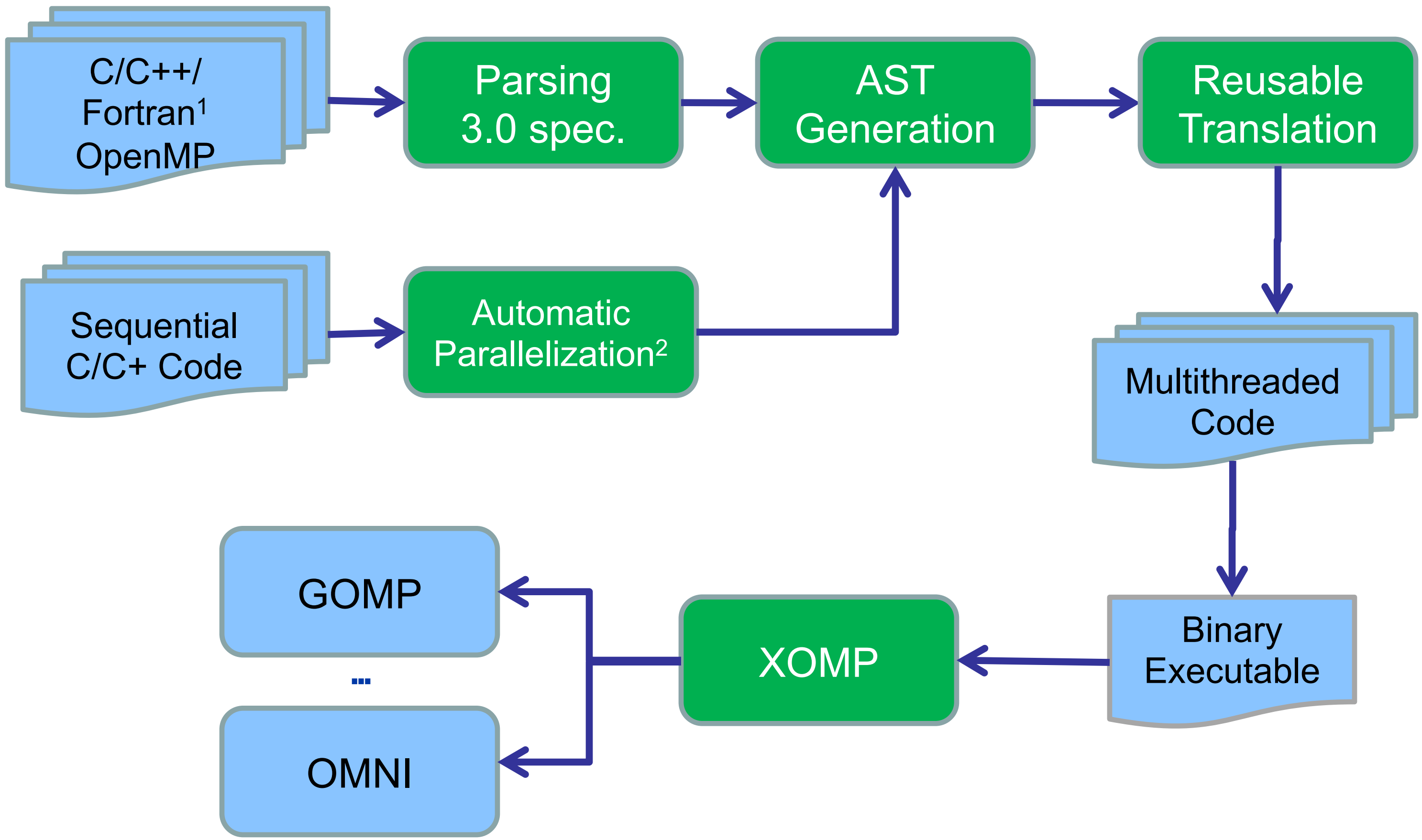
Simplified AST
without symbol tables

ROSE AST(abstract syntax tree)
- High level: preserve all source level details
- Easy to use: interface functions
- Extensible: user defined persistent attributes

Analyses and Optimizations in ROSE

- Analyses
  - Call graph and control flow graph
  - Def/use, liveness analysis
  - Side effect, pointer/alias analysis
  - Dependence analysis
- Transformations and Optimizations
  - Constant folding, partial redundancy elimination
  - Inlining and outlining
  - Loop optimizations(unrolling, blocking, interchanging)
  - Auto parallelization …

OpenMP Support in ROSE: Overview



1 Support for OpenMP Fortran is still ongoing work
2 Covered in a previous paper

Parsing and representing OpenMP

Neither EDG nor OFP processes OpenMP constructs
Two OpenMP 3.0 directive parsers were developed in ROSE:
- for C/C++ and Fortran

	OmpAttribute	SgOmp* Nodes
Implementation	AST persistent attribute	Dedicated statement-like AST node
Cost	Minimum	Medium
Scope Information	Lost	Kept
Location	Flexible (pragmas, loops)	Fixed in AST
Manipulation	Special handling	Generic handling via AST interface functions
Use	Output of parsers and auto parallelization	Formal AST for OpenMP

Using ROSE to build OpenMP analysis tools

```
#pragma omp parallel
 {
   omp_lock_t lck;
   omp_set_lock(&lck);
   printf("Thread = %d\n",  omp_get_thread_num());
   omp_unset_lock(&lck);
 }
```

A mistake when using OpenMP locks

- A lock: shared among threads to be effective
- A lock declared within a parallel region is private !

AST representatio

```c
#pragma omp parallel
 {
   omp_lock_t lck;
   omp_set_lock(&lck);
   printf(…);
   omp_unset_lock(&lck);
 }
```

Find private OpenMP locks using ROSE

```
void OmpPrivateLock::visit(SgNode* node){
  //1. Find an OpenMP lock routine by names
  SgFunctionCallExp * func_call = isSgFunctionCallExp(node);
  if (!func_call) return;
  std::string f_name = func_call->get_name();
  if (f_name != "omp_unset_lock" && f_name != "omp_set_lock"
     && f_name != "omp_test_lock")    return;

  //2. Grab the only routine parameter as the use of a lock
  std::vector<SgVarRefExp*> exp_vec =
     SageInterface::querySubTree<SgVarRefExp>(func_call, V_SgVarRefExp);

  //3. Get the parallel region of the lock
  SgOmpParallelStatement* lock_region =
   SageInterface::getEnclosingNode<SgOmpParallelStatement>(exp_vec[0]);

  if (lock_region)
  {
  //4. Check if the lock declaration is also inside the same region
    SgVariableDeclaration* lock_decl = exp_vec[0]->get_declaration();
   if (SageInterface::isAncestor(lock_region, lock_decl))
     cerr<<"Found a private lock within a parallel region"<<endl;
  } }
```

Agenda

- Motivation
- Overview of ROSE and its OpenMP support
- Parsing and representing OpenMP in ROSE
  - Facilitate static analysis for OpenMP
- Reusable translation for multiple runtime libraries
  - XOMP and preliminary results
- Future work

Tight coupling of translation and runtime

```c
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif

int main(void)
{
int i=0;
/*a parallel region */
#pragma omp parallel private(i)
{
#ifdef _OPENMP
  i=omp_get_thread_num();
#endif
  printf("Hello,world! I am thread
%d\n",i);
}
return 0;
}
```

Code outlining

Variable handling

Replacing the original code with a RTL call

```c
#include <stdio.h>
#include <omp.h>
#include "ompcLib.h"
/* Outlined function */
static void __ompc_func_1(void **__ompc_args)
{
  int _p_i; /*private variable*/
  _p_i = omp_get_thread_num();
   printf("Hello,world! I am thread %d\n",_p_i);
}
 int main(int argc,char **argv)
{
  int status = 0;
  _ompc_init(argc,argv); /*runtime system init*/
  int i = 0;
  _ompc_do_parallel(__ompc_func_1,0); /*RTL call*/
  _ompc_terminate(status); /*runtime system cleanup*/
  return 0;
}
```

Programmer → **compiler** → **Runtime library** → **OS threads**

How different are runtime libraries?

Runtime Support	GOMP	Omni
omp barrier	void GOMP_barrier (void)	void _ompc_barrier(void);
omp critical	GOMP_critical_name_start(void **data) GOMP_critical_name_end(void **data)	_ompc_enter_critical(void **data); _ompc_exit_critical(void **data);
omp single	int GOMP_single_start();	int _ompc_do_single();
omp parallel	void GOMP_parallel_start (void (*func) (void *), void *data, unsigned num_threads); void GOMP_parallel_end (void);	void _ompc_do_parallel(void (*func)(void **),void *args);
Initialization & Termination	None (Implicit)	_ompc_init(); _ompc_termination();
default loop scheduling	None (compiler generates all necessary code)	void _ompc_default_sched(int *lb, int *ub, int *step);
threadprivate	None (compiler inserts __thread)	void * _ompc_get_thdprv(void ***thdprv_p,int size,void *datap);

XOMP: a common translation-runtime layer

Rule ID	libA v.s. libB	XOMP interface	Compiler translation
Rule 1	funcA() and funcB(): similar functionality , but may differ by names / parameters	A common function with a union set of parameters for each	Targets XOMP_funcX()
Rule 2.1	libA has an extra funcA() : due to special need	XOMP_funcA() { if (libA) funcA(); else NOP; }	Targets XOMP_funcA()

Rule 2.2	funcA()'s functionality: suitable for runtime support	XOMP_funcA()
		{
		copy of funcA() body here
		}
			Targets XOMP_funcA()

Rule 2.3	funcA() 's functionality: suitable for compiler translation	No XOMP function	self-contained w/o runtime support
Rule 3	Support for feature X is too different to be merged	XOMP_funcA() XOMP_funcB()	Custom translation for each

-80% translations can be reused

XOMP rules applied to GOMP and Omni

XOMP	GOMP	Omni
void XOMP_barrier (void)	void GOMP_barrier (void)	void _ompc_barrier(void);
XOMP_critical_start (void** data) XOMP_critical_end (void** data)	GOMP_critical_name_start(void **data) GOMP_critical_name_end(void **data)	_ompc_enter_critical(void **data); _ompc_exit_critical(void **data);
int XOMP_single()	int GOMP_single_start();	int _ompc_do_single();
void XOMP_parallel_start (void (*func) (void *), void *data, unsigned numThread); void XOMP_parallel_end (void);	void GOMP_parallel_start (void (*func) (void *), void *data, unsigned num_threads); void GOMP_parallel_end (void);	void _ompc_do_parallel(void (*func)(void **),void *args);
void XOMP_init (int argc, char ** argv); void XOMP_terminate (int exitcode);	None (Implicit)	_ompc_init(); _ompc_termination();
None (Rule 2.3)	None (compiler generates all necessary code)	void _ompc_default_sched(int *lb, int *ub, int *step);
None (Rule 3) Void XOMP_get_thdprv(void ** t_p, int size, void* data);	None (compiler inserts __thread)	void * _ompc_get_thdprv(void ***thdprv_p,int size,void *datap);

Translation algorithm

- Top-down AST traversal
  - Make implicit data-sharing attribute explicit
    - e.g. firstprivate for certain variables within omp task
- Bottom-up traversal: translate OpenMP nodes
  - *Variable handling*: private, firstprivate, lastprivate, reduction
  - For omp parallel or omp task, call *the ROSE outliner* to
    - generate outlined function, and
    - replace the original code with XOMP call
  - For loop construct
    - Normalize the loop
    - Distribute iteration chunks to threads with help from XOMP calls

The ROSE outliner

- Outlining: Form a function from a code segment and replace the code segment with a call to the function
  - Used for kernel extraction (empirical tuning), task generation (OpenMP implementation), code refactoring, and so on
- The ROSE outliner*
  - The only freely available, standalone, source-to-source outlining tool supporting C/C++/Fortran
  - Variable cloning: reducing pointer dereferencing and reserve performance characteristics
  - Separate outlined functions into independent compilable files with all dependent declarations

***C. Liao**, D. Quinlan, R. Vuduc and T. Panas, **Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization**, *The 22nd International Workshop on Languages and Compilers for Parallel Computing*, Newark, Delaware, USA. October 8-10, 2009

Translation for parallel regions and tasks

```
#include "libxomp.h"
struct OUT__1__1527___data {  int i; } ;
struct OUT__2__1527___data { int i; };

static void OUT__1__1527__(void *__out_argv)
{
  int i = (int )(((struct OUT__1__1527___data *)__out_argv) -> i);
  int _p_i = i;
  process((item[_p_i]));
}

static void OUT__2__1527__(void *__out_argv)
{
  int i = (int )(((struct OUT__2__1527___data *)__out_argv) -> i);
  int _p_i = i;
  for (_p_i = 0; _p_i < 5000; _p_i++) {
    struct OUT__1__1527___data __out_argv1__1527__;
    __out_argv1__1527__.i = _p_i;
    /* void XOMP_task (void (*fn) (void *), void *data, void (*cpyfn) (void *, void *),
     * long arg_size, long arg_align, bool if_clause, bool untied )*/
    XOMP_task(OUT__1__1527__,&__out_argv1__1527__,0,4,4,1,0);
  }
}
```

```
#pragma omp parallel
#pragma omp single
 {
     int i;
#pragma omp task untied
     {
       for (i = 0; i < N; i++)
       {
 #pragma omp task if(1)
         process (item[i]);
       }
     }
   }
```

Translation for parallel regions and tasks (cont.)

```
//…. Continued from the previous slide
static void OUT__3__1527__(void *__out_argv)
{
  if (XOMP_single()) {
    int i;
    struct OUT__2__1527___data __out_argv2__1527__;
    __out_argv2__1527__.i = i;
    XOMP_task(OUT__2__1527__,&__out_argv2__1527__,0,4,4,1,1);
  }
  XOMP_barrier();
}}

int main(int argc,int argv)
{
int status = 0;
  XOMP_init(argc,argv);
  /* void XOMP_parallel_start (
   * void (*func) (void *), void *data, unsigned num_threads )*/
  XOMP_parallel_start(OUT__3__1527__,0,0);
  XOMP_parallel_end();

  XOMP_terminate(status);
  return 0;
}
```

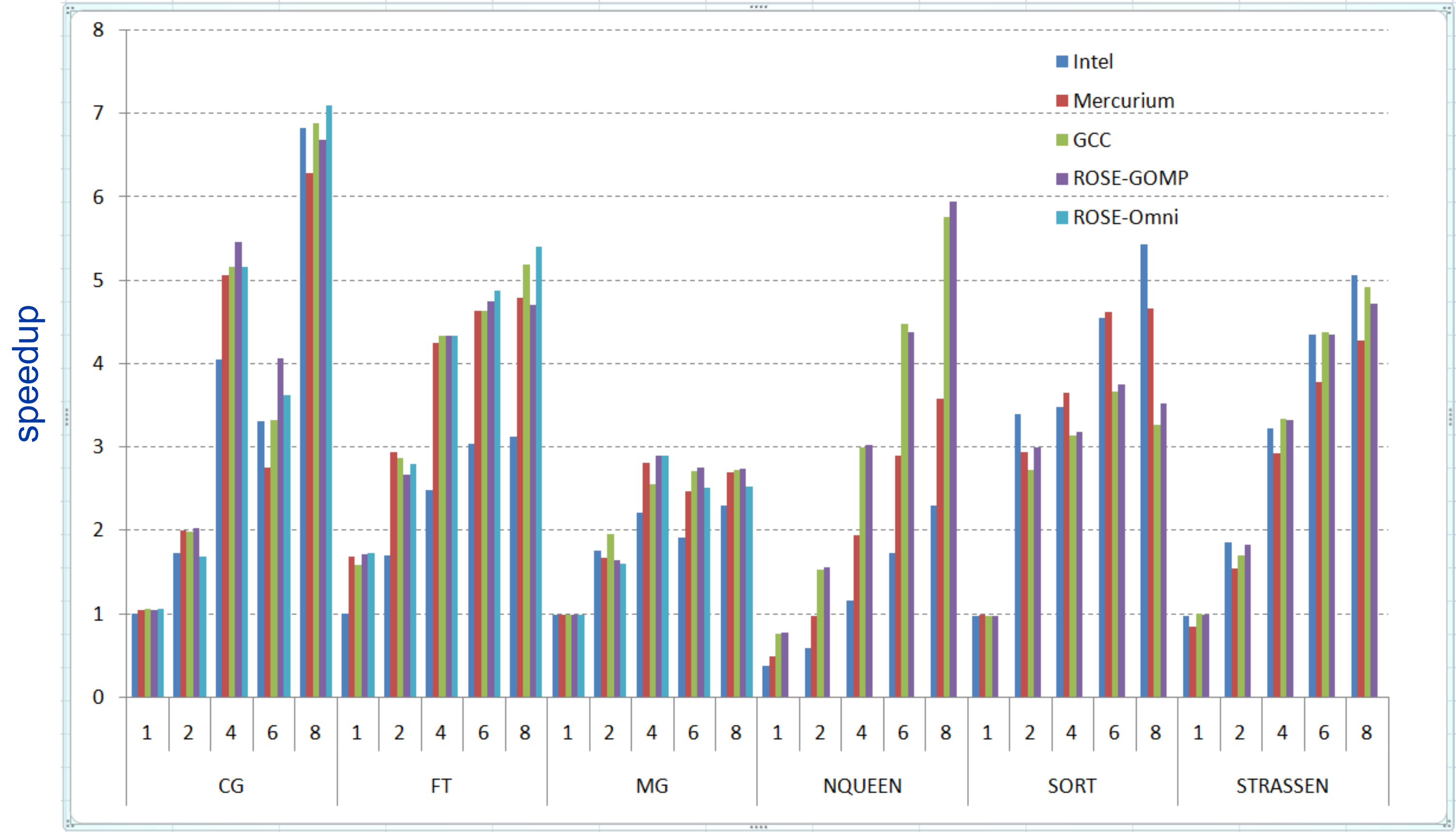
```
int main (int argc, int argv)
{
#pragma omp parallel
#pragma omp single
 {
     int i;
#pragma omp task untied
     {
       for (i = 0; i < N; i++)
       {
 #pragma omp task if(1)
          process (item[i]);
       }
     }
   }
}
```

Translation for C++ OpenMP code

```
#include "libxomp.h"
struct OUT__1__1527___data {  void *this__ptr___p; };
static void OUT__1__1527__(void *__out_argv);
static void *xomp_critical_user_;
class A {
  private:     int i;
  public:  friend void ::OUT__1__1527__(void *__out_argv);
    void pararun()
    {
      class A *this__ptr__ = this;
      struct OUT__1__1527___data __out_argv1__1527__;
      __out_argv1__1527__.this__ptr___p = (void *)this__ptr__;
      XOMP_parallel_start(OUT__1__1527__,&__out_argv1__1527__,0);
      XOMP_parallel_end();
    }
};
static void OUT__1__1527__(void *__out_argv)
{
   class A *this__ptr__ =  (class A *)(((struct OUT__1__1527___data *)__out_argv) -> this__ptr___p);
   XOMP_critical_start(&xomp_critical_user_);
    std::cout<<"i= "<<( *this__ptr__).i<<std::endl;
   XOMP_critical_end(&xomp_critical_user_);
}
```

```
class A {
  private:    int i;
  public:    void pararun()
    {
#pragma omp parallel
      {
#pragma omp critical
          cout<<"i= "<< i <<endl;
      }
    }
}
```

Preliminary results



Platform: Dell Precision T5400, 3.16GHz quad-core Xeon X5460 dual processor, 8GB
Benchmarks: NAS parallel benchmark suite v 2.3, Barcelona OpenMP task suite v 1.0
Compilers: ROSE, Omni 1.6, GCC 4.4.1, Mercurium 1.3.3 compiler with Nanos 4.1.4 runtime. Intel compiler 11.1.059

Conclusions

- ROSE
  - Metatool to build custom tools
  - High level AST with easy interface functions
  - Support C/C++/Fortran, OpenMP and UPC
- OpenMP support in ROSE
  - Analysis: dedicated AST nodes reusing existing interface functions
  - Transformation: reusable for multiple libraries
  - Optimization: semantic-aware parallelization*

C. Liao, D. Quinlan, J. Willcock and T. Panas, **Extending Automatic Parallelization to Optimize High-Level Abstractions for Multicore**, In Proceedings of the 5th international Workshop on OpenMP (Dresden, Germany, June 03 - 05, 2009).

Future work

- Better OpenMP implementation
  - Fortran support
  - More 3.0 features: e.g. loop collapse
- Empirical tuning for optimal compilation/execution
  - E.g. omp tasks' cut-off depth, tied vs. untied, task aggregation granularity, etc.
- Static analysis tools for OpenMP
- External collaborations using ROSE for OpenMP