INF560
Algorithmique Parallèle et Distribuée

2021/2022

Patrick CARRIBAULT

CEA, DAM, DIF, F-91297 Arpajon

Lecture Outline - OpenMP

- Tasks
  - Introduction
  - Data Flow
  - Synchronization
  - Loops
  - Data Dependencies
- Compiler Lowering
  - Compiler Structure
  - Internals
- Performance

OpenMP Tasks

- Introduction

Task Creation

- Task creation through dedicated directive

```
#pragma omp task
{
    // A
}
```

- Semantics when encountering this directive
  - Main action: task creation
  - Newly created task can be scheduled now or later
    ➔ Depends on context, clauses…
  - Task body = scope surrounded by directive
    - Block `A`

Orphaned?

- Orphaned task directive
  - Outside any OpenMP parallel region (dynamic extent)
  - Task will be executed now (no parallelism available)

- Parallel task directive
  - Inside any OpenMP parallel region (dynamic extent)
  - Task will be created now (and executed now or later)

- Typical skeleton

```
#pragma omp parallel
{
   #pragma omp single
   {
      #pragma omp task
      {
         // …
      }
   }
}
```

1 thread creates 1 task

To enable parallelism, task directive should be inside a parallel region

Task Hello World 1

```
#include <stdio.h>
#include <omp.h>

int main() {
  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp task
      {
        printf( "Hello World!\n" ) ;
      }
    }
  }
  return 0 ;
}
```

Include OpenMP header

Start parallel region (multiple threads might be launched)

Enter single construct (need to be crossed by all threads in the team)

Create one task

Task Hello World 2

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    {
      printf( "Hello ");
    }
    #pragma omp task
    {
      printf( "World!\n");
    }
  }
}
```

```
$ icc -o hello2 -iopenmp
hello2.c

$ OMP_NUM_THREADS=2 ./hello2
World!
Hello

$ OMP_NUM_THREADS=2 ./hello2
World!
Hello

$ OMP_NUM_THREADS=2 ./hello2
Hello
World!
```

Source Code

Compilation/Execution

Task Execution

- Task directive involves task creation
  - Created task might be executed later
- Task execution
  - Task might be executed by any thread of the same team (i.e., belonging to the same parallel region)
  - Threads may schedule tasks at specific points
    - *Task scheduling points*
    - E.g., during barriers
- Clause to force scheduling
  - `if([ task :] scalar-expression)`

Keyword only useful with combined construct

Task Hello World 3

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task if(0)
    {
      printf( "Hello ");
    }
    #pragma omp task
    {
      printf( "World!\n");
    }
  }
}
```

```
$ icc -o hello3 -openmp
hello3.c


$ OMP_NUM_THREADS=2 ./hello3
Hello
World!


$ OMP_NUM_THREADS=2 ./hello3
Hello
World!


$ OMP_NUM_THREADS=2 ./hello3
Hello
World!
```

Source Code

Compilation/Execution

Multiple Producers

- Previous example w/ single directive
  - Only one thread produces tasks
  - Mode called *mono-producer*
- Possibility to create tasks by different threads
  - Simply put task directive inside a parallel region
    - Outside a `single` construct
  - Mode called *multiple-producer*
- Still any thread can schedule tasks
  - Whatever the thread that created the task
  - Mode called *multiple-consumer*

Task Hello World 4

```
#pragma omp parallel
{
    #pragma omp task
    {
      printf( "Hello");
    }
}
```

```
$ icc -o hello4 -openmp hello4.c

$ OMP_NUM_THREADS=2 ./hello4
Hello
Hello

$ OMP_NUM_THREADS=8 ./hello3
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

Source Code

Compilation/Execution

Accessible Information

- Tasks can execute any piece of code

- Control/Data flow
  - Tasks can execute loops, `if` statement, function call…
  - Task can access any data that is shared or private
    - See next section for more details

- OpenMP state
  - Tasks can call OpenMP API
  - Get access to thread ID, number of threads…
  - Function `omp_get_thread_num()` will return thread ID that is currently executing the task

Task Hello World 5

```
#pragma omp parallel
{
  #pragma omp single
  {
    printf("Single %d\n",
      omp_get_thread_num());
    #pragma omp task
    {
      printf("Hello %d\n",
      omp_get_thread_num());
    }
  }
}
```

```
$ icc -o hello5 -openmp
hello5.c

$ OMP_NUM_THREADS=2 ./hello5
Single 0
Hello 0

$ OMP_NUM_THREADS=2 ./hello5
Single 0
Hello 0

$ OMP_NUM_THREADS=2 ./hello5
Single 0
Hello 1
```

Source Code

Compilation/Execution

Tasks and Workshare Constructs

- Tasks change control flow of OpenMP program
  - Threads can create tasks
  - Other threads may schedule those tasks
  - At some specific points, tasks may migrate to another threads (within the same OpenMP team)

- Be careful mixing tasks and workshare constructs
  - OpenMP defines workshare constructs
  - It includes `for`, `single`, `barrier`…
  - All threads must reach those constructs
  - Do not use within tasks!

Task and Workshare Example 1

```
#include <stdio.h>
#include <omp.h>

void f()
{
  printf( "TASK %d\n",
   omp_get_thread_num() ) ;

  #pragma omp single
  {
    printf( "[%d]Hello\n",
     omp_get_thread_num() );
  }
}
```

Deadlock because of single construct

```
int main()
{
  #pragma omp parallel
  {
    #pragma omp task
    {
      f() ;
    }
  }
  return 0 ;
}
```

Source Code (1)

Source Code (2)

Nested Tasks

- OpenMP allows creating tasks within tasks
  - Notion of *Nested Tasks*
- Syntax
  - Same as regular tasks: `#pragma omp task`
  - Same clauses (data flow, `if`…)
  - Can be within the static or dynamic scope of an existing task
- Clauses to influence task creation/scheduling
  - `final`
  - `mergeable`

Nested Example 1

```
#pragma omp parallel
{
#pragma omp single
  {
#pragma omp task
    {
      printf( "[%d]Hello 1\n",
      omp_get_thread_num() ) ;
#pragma omp task
      {
        printf( "[%d]Hello 2\n",
        omp_get_thread_num() ) ;
      }
    }
  }
}
```

```
$ icc -o nested1 -openmp
nested1.c

$ OMP_NUM_THREADS=2 ./nested1
[0]Hello 1
[1]Hello 2

$ OMP_NUM_THREADS=2 ./nested1
[1]Hello 1
[1]Hello 2

$ OMP_NUM_THREADS=2 ./nested1
[1]Hello 1
[0]Hello 2
```

Source Code

Compilation/Execution

Task Binding

- By default, tasks are said to be *tied*
  - Any thread from the team can start a new task, but once started, it has to complete it
- Influence scheduling
  - Only one thread can continue execution upon completion
- Possibility to let any thread continue task execution
  - Clause: `untied`

OpenMP Tasks

- Data Flow

Task Data Flow

- What is the data attributes of variables accessed inside a task?

- Multiple ways to determine sharing attributes
  - Clauses to specific data flow
  - Rules to determine attributes of variables declared inside the task → `private`
  - Rules to determine attributes of variables declared outside the task and accessed inside the task

- First example w/ local variable

Data Flow Example 1

```
int main() {
#pragma omp parallel
  {
    #pragma omp task
    {
      int a = 6 ;
      printf( "INSIDE "
        "a=%d, &a=%p\n",
        a, &a ) ;
    }
  }
  return 0 ;
}
```

```
$ icc -o df1 -openmp df1.c

$ OMP_NUM_THREADS=2 ./df1
INSIDE a=6, &a=0x7f78ab638da4
INSIDE a=6, &a=0x7fff7e1431d4

$ OMP_NUM_THREADS=2 ./df1
INSIDE a=6, &a=0x7ffdd5370b74
INSIDE a=6, &a=0x7f1324dc6da4

$ OMP_NUM_THREADS=2 ./df1
INSIDE a=6, &a=0x7ffe813a90c4
INSIDE a=6, &a=0x7ffe813a90c4

$ OMP_NUM_THREADS=2 ./df1
INSIDE a=6, &a=0x7ffd42efc694
INSIDE a=6, &a=0x7ffd42efc694
```

Source Code

Compilation/Execution

Data Flow Example 1 (cont)

```
int main() {
#pragma omp parallel
  {
    #pragma omp task
    {
      int a = 6 ;
      printf( "[%d]INSIDE "
       "a=%d, &a=%p\n",
       omp_get_thread_num(),
       a, &a ) ;
    }
  }
  return 0 ;
}
```

```
$ icc -o df1 -openmp df1.c

$ OMP_NUM_THREADS=2 ./df1cont
[0]INSIDE a=6, &a=0x7ffe80b6efd4
[1]INSIDE a=6, &a=0x7f81e67e7d94

$ OMP_NUM_THREADS=2 ./df1cont
[0]INSIDE a=6, &a=0x7ffd33fb0d04
[1]INSIDE a=6, &a=0x7fe4e177dd94

$ OMP_NUM_THREADS=2 ./df1cont
[0]INSIDE a=6, &a=0x7ffea7058064
[0]INSIDE a=6, &a=0x7ffea7058064

$ OMP_NUM_THREADS=2 ./df1cont
[0]INSIDE a=6, &a=0x7ffe7b428ac4
[0]INSIDE a=6, &a=0x7ffe7b428ac4
```

OK because reusing stack of same thread

Source Code

Compilation/Execution

Pre-Determined Rules

- What about variables declared outside task body?
  - What are the pre-determined rules?

- Rules that apply in specific order
  1. Predetermined by enclosing parallel region as shared ➔ `shared`
  2. Global variables ➔ `shared`
  3. Otherwise ➔ `firstprivate`

Data Flow Example 2

```
int main() {
  int a = 5 ;
  printf("OUTSIDE "
    "&a=%p\n", &a ) ;
#pragma omp parallel
  {
    #pragma omp task
    {
      printf( "INSIDE "
        "a=%d, &a=%p\n",
        a, &a ) ;
    }
  }
  return 0 ;
}
```

```
$ icc -o df2 -openmp df2.c

$ OMP_NUM_THREADS=2 ./df2
OUTSIDE &a=0x7fff33dabcec
INSIDE a=5, &a=0x7fff33dabcec
INSIDE a=5, &a=0x7fff33dabcec

$ OMP_NUM_THREADS=8 ./df2
OUTSIDE &a=0x7ffe7681dbec
INSIDE a=5, &a=0x7ffe7681dbec
INSIDE a=5, &a=0x7ffe7681dbec
INSIDE a=5, &a=0x7ffe7681dbec
INSIDE a=5, &a=0x7ffe7681dbec
INSIDE a=5, &a=0x7ffe7681dbec
INSIDE a=5, &a=0x7ffe7681dbec
INSIDE a=5, &a=0x7ffe7681dbec
INSIDE a=5, &a=0x7ffe7681dbec
```

Source Code

Compilation/Execution

Data Flow Example 3

```
int a = 5 ;

printf( "OUTSIDE a=%d &a=%p\n",
 a, &a ) ;

#pragma omp parallel private(a)
{
  a = omp_get_thread_num()+10;
  #pragma omp task
  {
    printf( "[%d]INSIDE a=%d,"
     "&a=%p\n",
     omp_get_thread_num(),
     a, &a ) ;
  }
}
```

```
$ icc -o df3 -openmp df3.c

$ OMP_NUM_THREADS=2 ./df3
OUTSIDE a=5 &a=0x7ffef5c76f84
[1]INSIDE a=11,&a=0x7faa51ec0d94
[0]INSIDE a=10,&a=0x7ffef5c76e14

$ OMP_NUM_THREADS=4 ./df3
OUTSIDE a=5 &a=0x7fffa022e434
[0]INSIDE a=10,&a=0x7fffa022e2c4
[2]INSIDE a=12,&a=0x7f1138ac2d94
[3]INSIDE a=13,&a=0x7f11382c1d94
[1]INSIDE a=11,&a=0x7f11392c3d94
```

Source Code

Compilation/Execution

Data-Flow Clauses

- Clauses to influence sharing attributes
- Change default behavior
  - `default(shared | none)`
- Explicit attribute for list of variables
  - `private(list)`
  - `firstprivate(list)`
  - `shared(list)`
- Advices
  - Use `default(none)`
    - Compiler will list all variables that require a sharing attribute
    - Programmer must put an attribute for each of those variables
  - Be careful to data environment between creation and execution

OpenMP Tasks

- Synchronization

Execution Scheduling

- *Recall:* task directive ➔ task creation
  - Created task can be scheduled immediately (undeferred) by the current thread or later (deferred) by any of the threads inside the team
- How to be sure that a task has been executed and completed?
  - One solution: perform a `barrier`
- Barrier: ensure that all tasks previously created are done before going on

Hello World 6 (Multi-Producer)

```
#pragma omp parallel
{
    #pragma omp task
    {
      printf( "Hello ");
    }
    #pragma omp barrier
    #pragma omp task
    {
      printf( "World!\n");
    }
}
```

```
$ icc -o hello6 -openmp hello6.c

$ OMP_NUM_THREADS=2 ./hello6
Hello
Hello
World!
World!

$ OMP_NUM_THREADS=4 ./hello6
Hello
Hello
Hello
Hello
World!
World!
World!
World!
```

Source Code

Compilation/Execution

Hello World 7 (Mono-Producer)

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    {
      printf( "Hello ");
    }
    #pragma omp barrier
    #pragma omp task
    {
     printf( "World!\n");
    }
  }
}
```

```
$ icc -o hello7 -openmp
hello7.c

hello7.c: In function 'main':

hello7.c:14:9: error: barrier
region may not be closely
nested inside of work-sharing,
'critical', 'ordered',
'master', explicit 'task' or
'taskloop' region
 #pragma omp barrier
```

Task/workshare issue.
How to synchronize tasks?

Source Code

Compilation/Execution

Taskwait

- How to wait for completion of tasks created by the current thread?
  - Directive `taskwait`

- Syntax
  - `#pragma omp taskwait`

- Current thread is blocked until its previously created tasks are not completed
  - This directive should be called only by the thread that creates the tasks to synchronize

Taskwait Example 1

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    {
      printf( "Hello ");
    }
    #pragma omp taskwait
    #pragma omp task
    {
     printf( "World!\n");
    }
  }
}
```

```
$ icc -o taskwait1 -openmp
taskwait1.c

$ OMP_NUM_THREADS=2 ./taskwait1
Hello
World!

$ OMP_NUM_THREADS=2 ./taskwait1
Hello
World!

$ OMP_NUM_THREADS=2 ./taskwait1
Hello
World!
```

Source Code

Compilation/Execution

Taskwait Example 2

```
#pragma omp parallel
{
   #pragma omp single
   {
      #pragma omp task
      {
         #pragma omp task
           printf( "Hello ");
      }
      #pragma omp taskwait
      #pragma omp task
      {
        printf( "World!\n");
      }
   }
}
```

Nested task

```
$ icc -o taskwait2 -openmp
taskwait2.c

$ OMP_NUM_THREADS=2 ./taskwait2
Hello
World!

$ OMP_NUM_THREADS=2 ./taskwait2
Hello
World!

$ OMP_NUM_THREADS=2 ./taskwait2
World!
Hello
```

Source Code

Compilation/Execution

Taskgroup

- Directive `taskwait` waits for previously-created tasks to be done, but only at the current level
  - Only tasks sharing the same parent are involved in `taskwait`
  - Nested tasks may not be already scheduled
  - It is important to note that finishing executing a task do not mean that created nested tasks will be done (no implicit `taskwait`)

- How to wait for a set tasks including nested tasks?
  - Directive `taskgroup`

- Syntax

```
#pragma omp taskgroup
{
  // ...
}
```

Taskgroup Example 1

```
#pragma omp parallel
{
   #pragma omp single
   {
      #pragma omp taskgroup
      {
        #pragma omp task
        {
          #pragma omp task
            printf( "Hello ");
        }
      }
      #pragma omp task
      {
        printf( "World!\n");
      }
   }
}
```

```
$ icc -o taskgroup1 -openmp
taskgroup1.c

$ OMP_NUM_THREADS=2 ./taskgroup1
Hello
World!

$ OMP_NUM_THREADS=2 ./taskgroup1
Hello
World!

$ OMP_NUM_THREADS=2 ./taskgroup1
Hello
World!
```

Source Code

Compilation/Execution

Locks

- How to deal with synchronization of shared-variable accesses
  - Locks
  - Atomic operations

- Regular OpenMP locks can be used within tasks
  - `void omp_init_lock(omp_lock_t *lock);`
  - `void omp_destroy_lock(omp_lock_t *lock);`
  - `void omp_set_lock(omp_lock_t *lock);`
  - `void omp_unset_lock(omp_lock_t *lock);`
  - `int omp_test_lock(omp_lock_t *lock);`

Lock Example 1

Protected region accessing a

```
#include <stdio.h>
#include <omp.h>

int a = 8 ;

int main()
{
  omp_lock_t l ;

  omp_init_lock( &l ) ;

  #pragma omp parallel
    {
```

Global shared variable

Shared lock

```
    #pragma omp task
        {
          omp_set_lock(&l);
          a++;
          omp_unset_lock(&l);
        }
    }

  printf( "Value of a = %d\n",
a ) ;
  return 0 ;
}
```

```
$ icc -o lock1 -openmp lock1.c
$ OMP_NUM_THREADS=2 ./lock1
Value of a = 10
$ OMP_NUM_THREADS=8 ./lock1
Value of a = 16
```

Source Code

Source Code/Execution

Locks

- Ownership
    - Lock owner is the current task (no issue with `untied` tasks & scheduling)

- Be careful
    - To control flow divergence
        - Be sure that there is an unlock call on every path from lock call
    - To task owner
        - Owner of a lock is a task (for regular locks and nested/recursive locks)

- In case of simple operation to protect
    - Use `atomics` construct

Atomics

- When operation is simple → possibility to use `atomics` directive

- Syntax

```
#pragma omp atomic
  operation
```

- Operation has restriction
  - Mainly manipulation of one memory cell with one operator

Example: Fibonacci

- Simple example of *taskification* with recursive algorithm: Fibonacci

- Sequential function

```
int fib( int n )
{
   if ( n <= 2 ) return n ;
   return fib(n-1)+fib(n-2) ;
}
```

Example: Fibonacci

- Parallel version with nested tasks

```
int fib( int n )
{
  int n1, n2 ;
  if ( n <= 2 ) return n ;
  #pragma omp task
    n1 = fib(n-1) ;
  #pragma omp task
    n2 = fib(n-2) ;
  #pragma omp taskwait
  return n1+n2 ;
}
```

- This function needs to be called within a parallel region

Example: Fibonacci

- Previous example may create many tasks!
- Possibility to stop recursion through a cutoff

```
int fib( int n )
{
  int n1, n2 ;
  if ( n <= 2 ) return n ;
  #pragma omp task if (n>20)
    n1 = fib(n-1) ;
  #pragma omp task if (n>20)
    n2 = fib(n-2) ;
  #pragma omp taskwait
  return n1+n2 ;
}
```

- Difficult to choose the right value for cutoff!

OpenMP Tasks

- Loops

Loops

- How to deal with loops in OpenMP?

- *For* loops
  - Rely on regular construct from OpenMP 2
  - Possibility to combine w/ parallel construct
  - Syntax: `#pragma omp for`

- *While* loops
  - Cannot rely on regular for construct
  - Possibility to use tasks for each iteration
  - Each iteration should be independent
  - Need to respect dependencies to compute the exit condition

While Loops

- Linked list traversal
  - Structure named list containing current element and pointer to next element

```
void list_traversal( struct list * l )
{
  struct list * l2 ;
  l2 = l ;
  while ( l2 != NULL ) {
    process_elt(l2) ;
    l2 = l2->next ;
  }
}
```

While Loops

```
void list_traversal( struct
list * l )
{
  #pragma omp parallel
  {
    #pragma omp single
    {
      struct list * l2 ;
      l2 = l ;
```

```
      while ( l2 != NULL ) {
        #pragma omp task
        {
          process_elt(l2) ;
        }
        l2 = l2->next ;
      }
    }
  }
}
```

Task Version

Task Version (cont.)

Taskloop

- How to create tasks with regular `for` loops?
- Possibility to traverse a for loop and create tasks
  - How to deal with multiple iterations at once?
    - Need to unroll by hand
  - How to deal with scheduling and distribution over threads?
    - Not easy
- Since OpenMP 4.5, new directive
  - `#pragma omp taskloop`

Require OpenMP 4.5 support

Taskloop Clauses

- Taskloop construct will create tasks for iterations of the following loop

- Clauses for data flow
  `shared(list)`
  `private(list)`
  `firstprivate(list)`
  `lastprivate(list)`
  `default(shared | none)`

- Clause to define if tasks should be deferred
  `if([ taskloop :] scalar-expr)`
  `final(scalar-expr)`
  `mergeable`

Taskloop Clauses

- Clauses to control priority and *tiedness* of created tasks
  `priority(priority-value)`
  `untied`

- Clause to control the processed loopnest
  `collapse(n)`

- Clause to control the size of each task and total number of tasks
  `grainsize(grain-size)`
  `num_tasks(num-tasks)`

  Clauses that may shape overall performance

- Clause to control if tasks are inside a taskgroup
  `nogroup`

OpenMP Tasks

- Data Dependencies

Task Dependencies

- Tasks created by the same parent are independent from each other
  - If deferred, they can be scheduled by any thread in any order

- How to ensure ordering
  - Available synchronization through `taskwait`, `taskgroup` and implicit/explicit `barrier`

- How to express fine-grain dependencies between tasks
  - New clause (since OpenMP 4): `depend`

Taskdep Example 1

```
int a = 0, b = 0 ;
#pragma omp parallel
 {
#pragma omp single
   {
     #pragma omp task
       a = 6 ;
     #pragma omp task
       b = 6 ;
     #pragma omp task
       printf( "a = %d\n", a );
     #pragma omp task
       printf( "b = %d\n", b );
   }
 }
```

```
$ icc -o taskdep1 -openmp
taskdep1.c

$ OMP_NUM_THREADS=2 ./taskdep1
a = 6
b = 6

$ OMP_NUM_THREADS=2 ./taskdep1
a = 0
b = 0

$ OMP_NUM_THREADS=2 ./taskdep1
a = 0
b = 6

$ OMP_NUM_THREADS=2 ./taskdep1
a = 6
b = 6
```

Source Code

Compilation/Execution

Taskdep Example 1 (cont)

```
int a = 0, b = 0 ;
#pragma omp parallel
 {
#pragma omp single
   {
     #pragma omp task
       a = 6 ;
     #pragma omp task
       b = 6 ;
     #pragma omp taskwait
     #pragma omp task
       printf( "a = %d\n", a );
     #pragma omp task
       printf( "b = %d\n", b );
   }
 }
```

```
$ icc -o taskdep1 -openmp
taskdep1.c

$ OMP_NUM_THREADS=2 ./taskdep1
a = 6
b = 6

$ OMP_NUM_THREADS=2 ./taskdep1
a = 6
b = 6

$ OMP_NUM_THREADS=2 ./taskdep1
a = 6
b = 6

$ OMP_NUM_THREADS=2 ./taskdep1
a = 6
b = 6
```

Source Code

Compilation/Execution

Taskdep Example 2

```
int a = 0, b = 0 ;
#pragma omp parallel
 {
#pragma omp single
   {
     #pragma omp task depend(out:a)
       a = 6 ;
     #pragma omp task depend(out:b)
       b = 6 ;
     #pragma omp task depend(in:a)
       printf( "a = %d\n", a );
     #pragma omp task depend(in:b)
       printf( "b = %d\n", b );
   }
 }
```

```
$ icc -o taskdep2 -openmp
taskdep2.c

$ OMP_NUM_THREADS=2 ./taskdep1
a = 6
b = 6

$ OMP_NUM_THREADS=2 ./taskdep1
a = 6
b = 6

$ OMP_NUM_THREADS=2 ./taskdep1
b = 6
a = 6
```

Source Code

Compilation/Execution

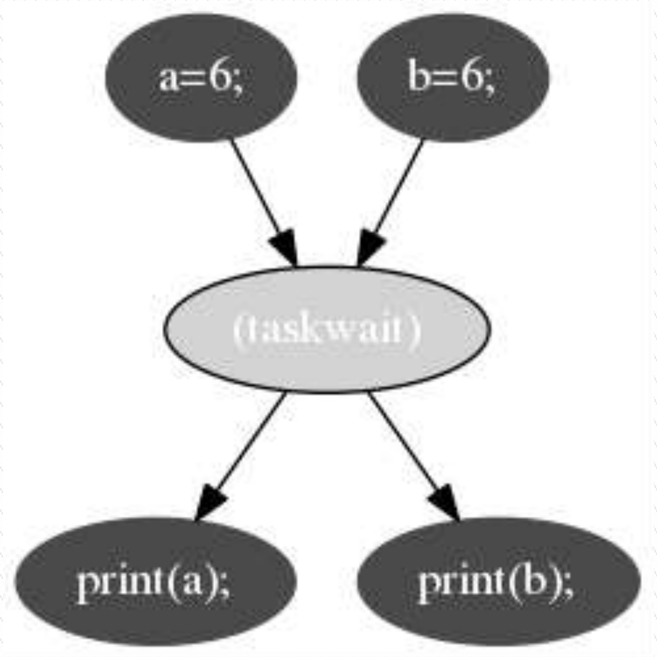
Task Dependency Graph

- Why is first example incorrect?
  - Because sibling tasks can be scheduled in any order
    - Total ordering on creation
    - No ordering on scheduling
  - Only guarantee
    - All tasks will be completed when reaching the `single` implicit barrier

- To understand advantages of data dependencies vs. `taskwait`
  - → Graph representation

Graph of Example 1

```
int a = 0, b = 0 ;
#pragma omp parallel
 {
#pragma omp single
   {
     #pragma omp task
       a = 6 ;
     #pragma omp task
       b = 6 ;
     #pragma omp taskwait
     #pragma omp task
       printf( "a = %d\n", a );
     #pragma omp task
       printf( "b = %d\n", b );
   }
 }
```

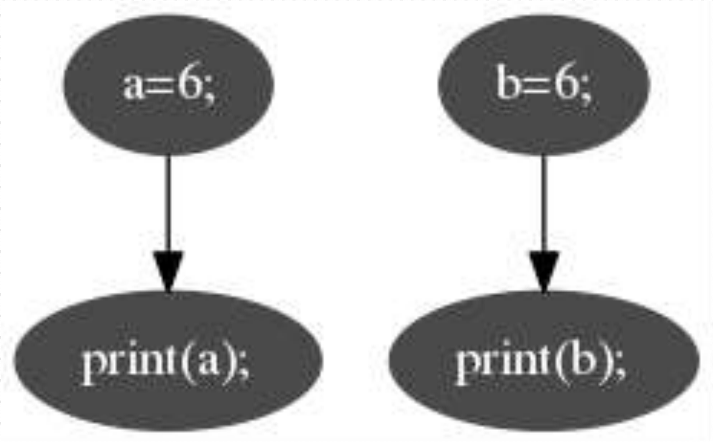


Source Code

Task Dependency Graph

Graph of Example 2

```
int a = 0, b = 0 ;
#pragma omp parallel
 {
#pragma omp single
   {
     #pragma omp task depend(out:a)
       a = 6 ;
     #pragma omp task depend(out:b)
       b = 6 ;
     #pragma omp task depend(in:a)
       printf( "a = %d\n", a );
     #pragma omp task depend(in:b)
       printf( "b = %d\n", b );
   }
 }
```



Source Code

Task Dependency Graph

Dependency Clause

- Clause to express dependencies
  - `#pragma omp task depend`

- Apply on data dependencies w/ following syntax
  `depend(dependence-type : list)`
  - Type: `in`, `out`, `inout`

- Summary:
  - Variables read inside the task: `in`
  - Variables written inside the task: `out`
  - Variables read/written inside the task: `inout`

- Be careful: only available through sibling tasks!
  - Only tasks with the same parent can be ordered through `depend` clause

Nested Task Dependency

- Dependencies are only valid for sibling tasks
  - I.e., tasks created by the same parent task
  - Task creation ordering allow expressing data dependency
- What about nested tasks?
  - No data dependencies between one task and the children of another sibling task
- Solution if required such dependencies
  - Put the depend clauses at the top tasks (i.e., tasks that are created first)
  - Put `taskwait` clause within those tasks (at each level) to ensure that execution of children tasks are done before finishing tasks with dependencies

Compiler Lowering

- Introduction

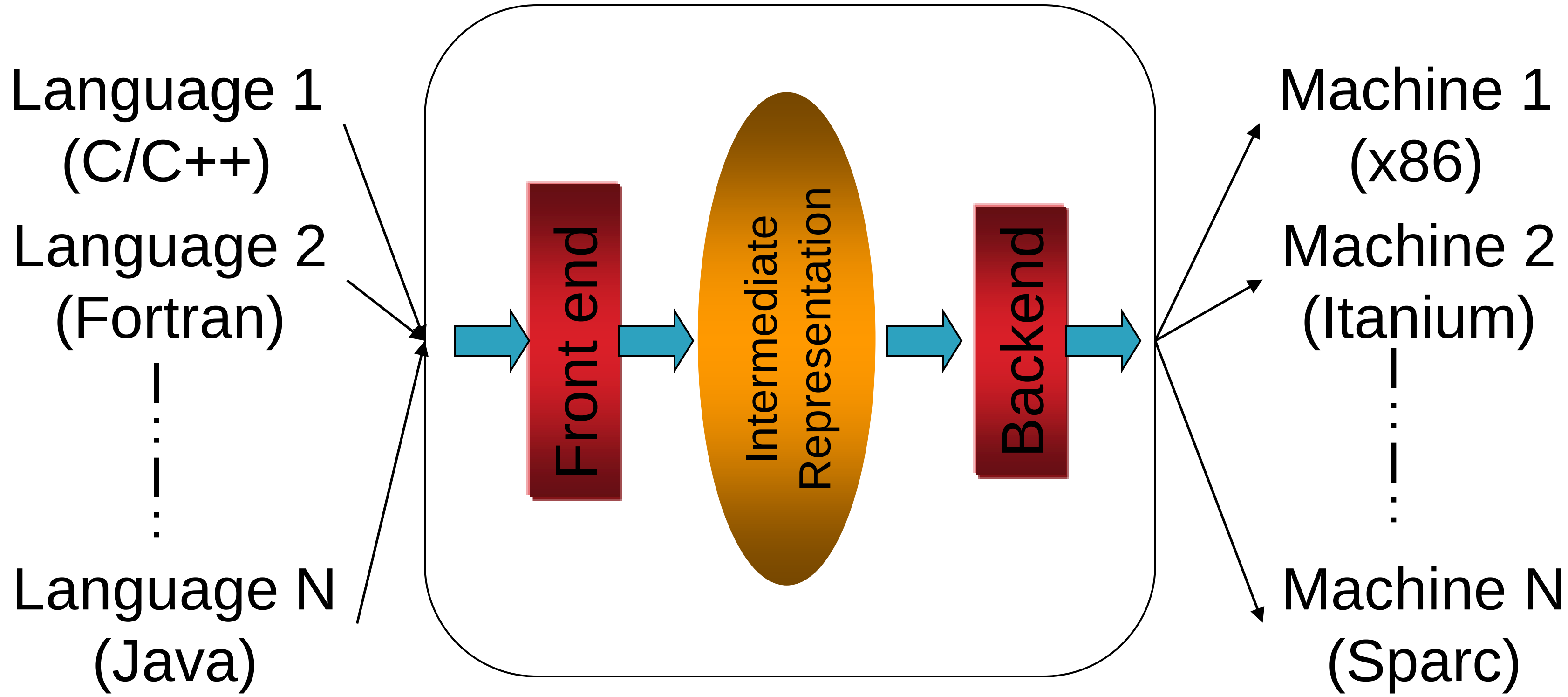
Compiler Lowering

- To understand how OpenMP works (and therefore have some hints on performance)
  - → necessary to know how OpenMP directives are transformed

- Two parts involved
  - Compiler
  - Library (usually, library is shipped with corresponding compiler, but not always true)

- Example with GNU GCC

Regular Compiler



Ideal View

GNU Compiler Chain

- GCC: GNU Compiler Collection
  - Used to be GNU C Compiler
- Set of tools and libraries for compilation
  - Multiple languages
  - Multiple target architectures
  - ➔ Compiler generator!
- Available under GPL license
  - http://gcc.gnu.org
- Main compiler used during Labs!

GCC History

- 0.9: March 1987
- GCC 1.0: May 1987
- GCC 3.0: June 2001
  - JAVA support
- GCC 4.0: April 2005
  - SSA form + software pipelining
  - Intermediate representation GIMPLE
- GCC 4.2.0: May 2007
  - OpenMP for C/C++ and FORTRAN
- GCC 4.5.0: April 2010
  - Link optimizations (LTO)
- GCC 4.6.0: March 2011
  - Optimizations + support of CAF and GO
- GCC 4.7.0: March 2012
  - OpenMP 3.1
  - Standard C++11
- GCC 4.8.0: March 2013
  - Support of FORTRAN 2003 and C++11
- GCC 4.9.0: April 2014
  - OpenMP 4.0
  - Experimental support for C++14
  - Support AVX-512
- GCC 5.1: April 2015
  - Default to C11
  - Intel Xeon Phi support
  - C++11
- GCC 6.1: April 2016
  - C++14
  - OpenACC 2.0a
  - OpenMP 4.5
- New major version each Spring
  - GCC 10.1: May 2020
  - GCC 11.1: April 2021
- Current version → GCC 11.2

Main Features

- Supported languages
  - C, C++
  - Objective-C, Objective-C++
  - JAVA,
  - Fortran
  - ADA

- Supported processors (ISA)
  - ARM, IA-32 (x86), x86-64, IA-64, MIPS, SPARC…

- Plugin system to add/modify compilation passes

- How many lines of codes in GCC?

GCC SLOC

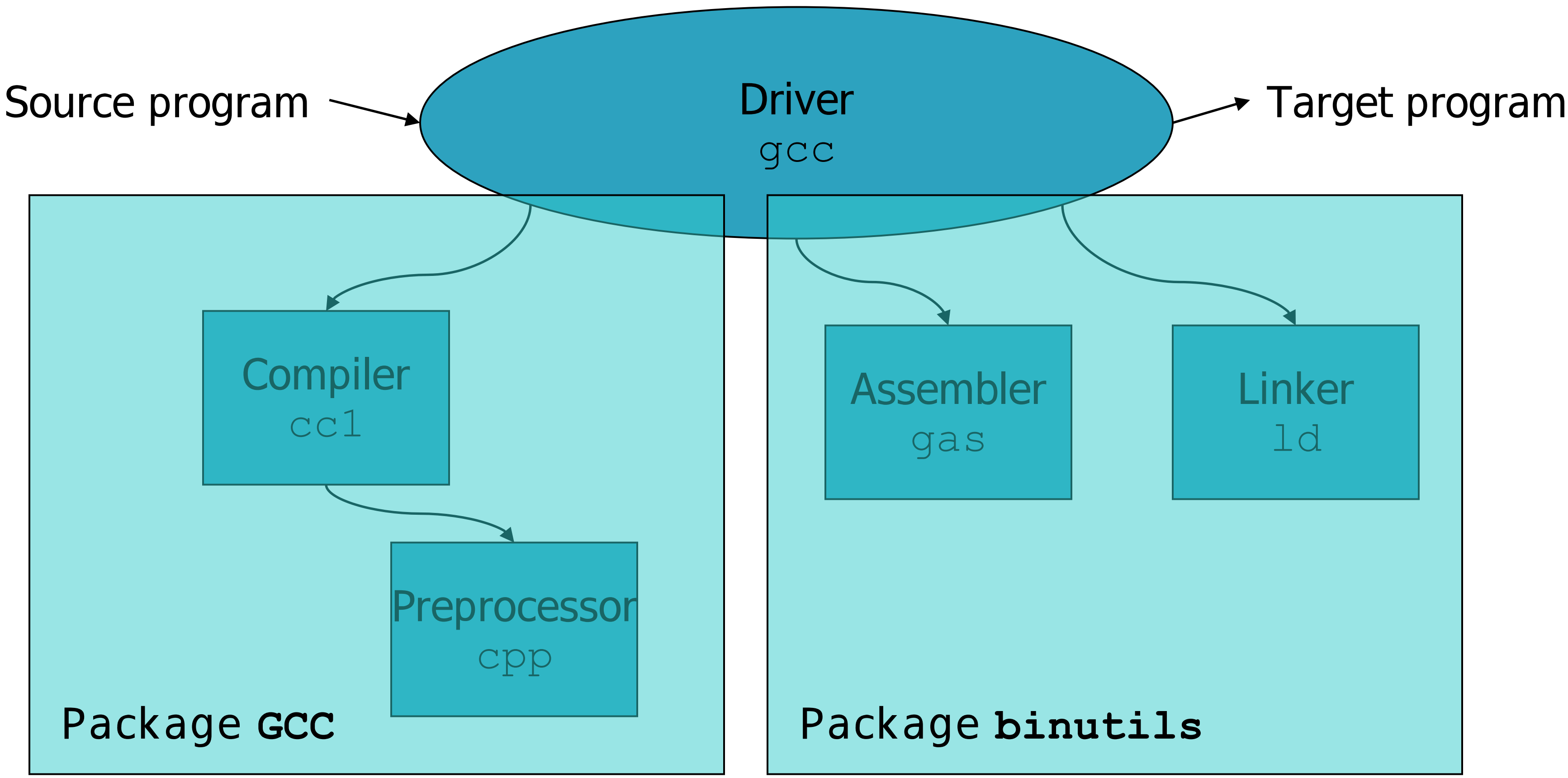
Count		GCC 4.3.0	GCC 4.4.2	GCC 4.5.0
Lines	Main source	2,029,115	2,187,216	2,320,963
	Libraries	1,546,826	1,633,558	1,671,501
	Subdirectories	3,527	3,794	4,055
Files	Number of files	57,660	62,301	77,782
	C source files	15,477	18,225	20,024
	Header files	9,646	9,213	9,389
	C++ files	3,708	4,232	4,801
	Machine description	186	206	229

(Line counts estimated by David A. Wheeler's sloccount program)

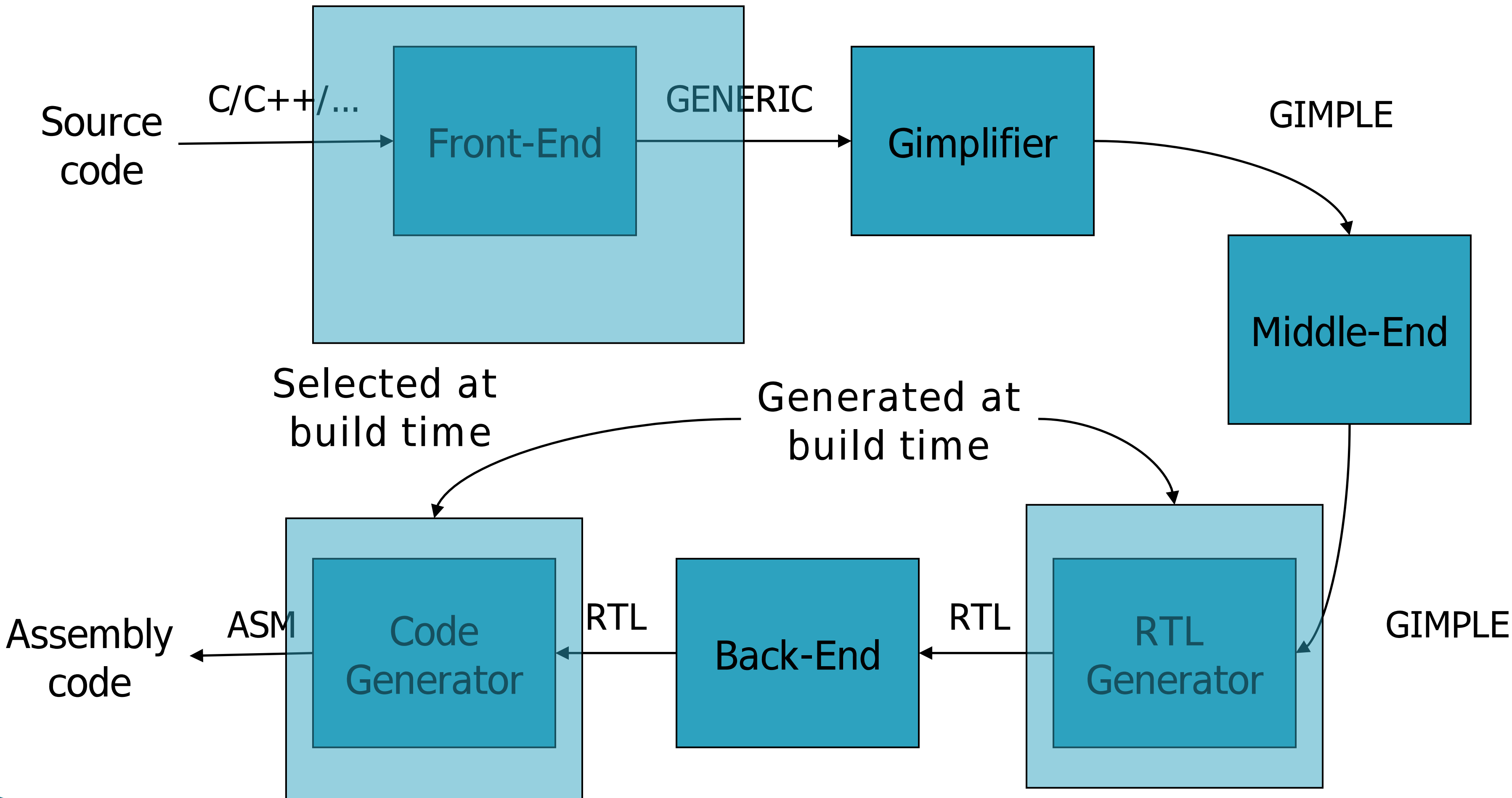
GCC 4.6.2

Language	Files	Code	Comment	Comment %	Blank	Total
c	18624	2106311	445288	17.5%	419325	2970924
cpp	22206	989098	230376	18.9%	215739	1435213
java	6342	681938	645505	48.6%	169046	1496489
ada	4616	680251	316021	31.7%	234551	1230823
autoconf	91	405517	509	0.1%	62919	468945
html	457	168378	5669	3.3%	38146	212193
make	98	121136	3658	2.9%	15555	140349
fortranfixed	2989	100688	1950	1.9%	13894	116532
shell	148	48032	10451	17.9%	6586	65069
assembler	208	46750	10227	17.9%	7854	64831
xml	75	36178	282	0.8%	3827	40287
objective_c	869	28049	5023	15.2%	8124	41196
fortranfree	831	13996	3204	18.6%	1728	18928
tex	2	11060	5776	34.3%	1433	18269
scheme	6	11023	1010	8.4%	1205	13238
automake	67	9442	1039	9.9%	1457	11938
perl	28	4445	1316	22.8%	837	6598
ocaml	6	2814	576	17.0%	378	3768
xslt	20	2805	436	13.5%	563	3804
awk	11	1740	396	18.5%	257	2393
python	10	1725	322	15.7%	383	2430
css	24	1589	143	8.3%	332	2064
pascal	4	1044	141	11.9%	218	1403
csharp	9	879	506	36.5%	230	1615
dcl	2	402	84	17.3%	13	499
tcl	1	392	113	22.4%	72	577
javascript	4	341	87	20.3%	35	463
haskell	49	153	0	0.0%	17	170
bat	3	7	0	0.0%	0	7
matlab	1	5	0	0.0%	0	5
Total	57801	5476188	1690108	23.6%	1204724	8371020

GCC Architecture



GCC Architecture



GCC Transformations

- GCC proposes 203 different transformation passes
- Total number of passes applies during compilation is 239
  - Some transformations are applied multiple times during the compilation process
- Notion of *pass manager* to control the application of transformations on intermediate representations
  - For more details, see files `${SOURCE}/gcc/passes.c` and `${SOURCE}/gcc/passes.def`

GIMPLE Transformations

Pass Group	Number of passes
Lowering	12
Interprocedural optimizations	49
Intraprocedural optimizations	42
Loop optimizations	27
Remaining intraprocedural optimizations	23
Generating RTL	01
Total	154

RTL Transformations

Pass Group	Number of passes
Intraprocedural Optimizations	21
Loop optimizations	7
Machine Dependent Optimizations	54
Assembly Emission and Finishing	03
Total	85

Preprocessor

- CPP: Manage compiler directives
- Directive prefix
  - `#keyword`
- Directive examples
  - `#ifdef`
  - `#include`
  - `#warning`
  - `#error`

```
$ cat hello.c
#include <stdio.h>
int main() {
 printf( "Hello World\n" ) ;
 return 0 ;
}

$ gcc –o hello.i –E hello.c

$ wc –l hello.i
843 hello.i
```

- Size of code can be very large after applying *preprocessing*
- Warning: `#pragma` directives is not processed by preprocessor

Front-end

- Read input file
  - C, C++, Fortran, Java, C#, …

- Check code validity
  - Lexical analysis
  - Syntax analysis
  - Semantics analysis

- Each frontend is located in different subdirectory
  - C, ObjectiveC → ${SOURCE}/gcc/c/, ${SOURCE}/gcc/c-family/
  - C++ → ${SOURCE}/gcc/cp/, ${SOURCE}/gcc/c-family/
  - Fortran → ${SOURCE}/gcc/fortran/

- Output: code represented in GENERIC IR
  - Except for C/C++ which generated GIMPLE IR

GENERIC

- Tree-based intermediate representation
- Independent from source language
- Process to create GENERIC representation
  1. AST generation by parser
  2. Removal of language-dependent construct
  3. Emission of GENERIC tree
- Available nodes from tree representation defined in `$(SOURCE)/gcc/tree.def`
  - Notion of *tree codes*

Middle-end

- High-level optimization
  - Independent from target architecture
- Granularity
  - Loop-wise optimization
  - Intraprocedural optimization
  - Interprocedural optimization
- Transformation ordering → GCC *pass manager*
- Intermediate representation: GIMPLE
  - In addition to other IRs (i.e., CFG)

GIMPLE

- High-level intermediate representation
  - Introduced in GCC 4.4
  - Tree-based representation
  - Node with semantics

- Simplified sub-set of GENERIC
  - 3-adress coding
  - Flat control flow
  - Transformation GENERIC → GIMPLE
    - `gimplify_function_tree()` in file `gimplify.c`

- Available GIMPLE levels
  - *High GIMPLE*
  - *Low GIMPLE*

GIMPLE – C Example

- Simple example
  - C language
  - Only one function (main)

- Compilation with intermediate file dump
  - `gcc -fdump-tree-all test.c`
  - Generation of GIMPLE IR during pass manager

```
int main() {
  int x = 10 ;
  if (x) {
      int y = 5 ;
      x = x*y+15 ;
  }
}
```

GIMPLE – C Example

File test.c:

```
int main() {
   int x = 10 ;
   if (x) {
         int y = 5 ;
         x = x*y+15 ;
   }
}
```

File test.c.004t.gimple:

```
main() {
   int D.2720;
   int x;
   x = 10 ;
   if (x!=0) goto <D.2718>;
   else goto <D.2719>;
   <D.2718>:
   {
         int y;
         y=5;
         D.2720 = x*y;
         x = D.2720+15
   }
   <D.2719>:
}
```

- Temp variable declaration
  - `D.2720`
- 3-address form
  - `D.2720 = x*y`
- Control flow with `goto`

GIMPLE – C Example

- GIMPLE code
  - `gcc -fdump-tree-all-raw test.c`

```
File test.c.004t.gimple:

main() {
   int D.2720;
   int x;
   x = 10 ;
   if (x!=0) goto <D.2718>;
   else goto <D.2719>;
   <D.2718>:
   {
         int y;
         y=5;
         D.2720 = x*y;
         x = D.2720+15
   }
   <D.2719>:
}
```

```
File test.c.004t.gimple:

main()
gimple_bind <
   int D.2720;
   int x;
   gimple_assign<integer_cst,x,10,NULL>
   gimple_cond <ne_expr, x, 0, <D.2718>,
   <D.2719> >
   gimple_label <<D.2718>>
   gimple_bind <
         int y;
         gimple_assign<integer_cst, y,
   5, NULL>
          gimple_assign<mult_expr,
   D.2720, x, y>
         gimple_assign<plus_expr,x,
   D.2720,15>
   >
   gimple_label<<D.2719>>
>
```

GIMPLE – C Example

File test.c.004t.gimple:

```
main() {
   int D.2720;
   int x;
   x = 10 ;
   if (x!=0) goto <D.2718>;
   else goto <D.2719>;
   <D.2718>:
   {
         int y;
         y=5;
         D.2720 = x*y;
         x = D.2720+15
   }
   <D.2719>:
}
```

File test.c.011t.cfg

```
main() {
   int y;
   int x;
   int D.2720;

<bb2>:
   x=10;
   if (x!=0) goto <bb 3>;
   else goto <bb 4>;

<bb 3>:
   y=5;
   D.2720 = x*y;
   x=D.2720+15;

<bb 4>:
   return ;
}
```

GIMPLE - *tree code*

- All *tree code* de GCC (152) are defined in `$(SOURCE)/gcc/tree.def`
- Binary Operator
  - MAX EXPR
- Comparison
  - EQ EXPR, LT EXPR
- Constants
  - INTEGER CST, STRING CST
- Declaration
  - FUNCTION DECL, LABEL DECL , VAR DECL
- Expression
  - PLUS EXPR, ADDR EXPR
- Reference
  - COMPONENT REF, ARRAY RANGE REF
- Statement
  - GIMPLE MODIFY STMT, RETURN EXPR, COND EXPR, INIT EXPR
- Type
  - BOOLEAN TYPE, INTEGER TYPE
- Unary
  - ABS EXPR, NEGATE EXPR

Back-end

- Main goals
  - Optimization based on target architecture
  - Assembly code generation

- IR named RTL
  - *Register Transfer Language*

- Based on machine description
  - Notion de *machine description*

RTL

- Building blocks : RTL object
  - Expressions
  - Integers
  - Wide integers
  - Strings
  - Vectors

- Each expression has a code
  - List of available codes is defined in `rtl.def`
  - Macro to access code of an expression: `GET_CODE(x)`

RTL

- Example: assignment
  - `DEF_RTL_EXPR(SET, "set", "ee", RTX_EXTRA)`
- Two operands
  1. Destination (register, memory…)
  2. Value
- Macro
  1. Internal name (upper case)
  2. ASCII form (lower case)
  3. Print forma (documentation in `rtl.c`)
     1. ‘e’ defines a pointer to an expression

RTL

- Example : expression `b = a + 3 ;`
  - `a` is located inside register `reg:SI 60`

Previous/
Current/
Next instruction

RTL code

```
(insn 7 6 8 test.c:2 (set
    (reg:SI 59)
    (plus:SI (reg:SI 60)
      (const_int 3 [0x3])))
-1 (nil))
```

Sub-expression
(addition)

Assembly

- Core compiler generates an assembly ASCII file at the end of the compiler chain
  - Output of back-end part

- Assembly tool
  - Translation from ASCII to binary
  - Straightforward translation

- GCC relies on OS assembly tool: `GAS`
  - From package `binutils`

Linker

- Final step to generate a binary file
- Collects objects file to create final executable
  - Binary that can be run
  - Library
- Update symbols for dynamic calls
- Finalize few optimizations (e.g., *Thread Local Storage* or TLS)

Compiler Lowering

- OpenMP Transformations

OpenMP Transformations

- How to transform a parallel region?

1. Data flow
   - Which variables are alive inside the parallel region?
2. Temporary structure
   - Store copies of variables needed for data flow
3. Extraction of parallel region inside new function (*outlining*)
4. Data flow restoration
   - Copy data from the temporary structure
5. Call to internal library to start parallel region

- Steps are done automatically by compiler

- Manual application?

Data Flow

- Study of data flow in function `vectAdd`
- Variables `a`, `b`, `c`, `N` and `i` are accessed inside parallel region
  - Variable `i` is declared private in parallel region
  - Each thread should have its own copy
- All variable are alive before and after parallel region (function scope)

```
void vecAdd( double * a,
  double * b,
  double * c,
  int N ) {
  int i ;
#pragma omp parallel for\
  private(i)
  for (i=0; i<N; i++) {
    c[i] = a[i] + b[i] ;
  }
}
```

Temporary Structure

- Structure creation
  - One field per variable inside parallel region
  - Variables with *private* status are not taken into account
- Structure should be filled with correct values
- In our example:
  - Transfer of `a`
  - Transfer of `b`
  - Transfer of `c`
  - Transfer of `N`

```
struct _s {
    double * _a ;
    double * _b ;
    double * _c ;
    int _N ;
  };

void vecAdd( double * a,
  double * b,
  double * c,
  int N ) {
  int i;
  struct _s s ;
  s._a = a ;
  s._b = b ;
  s._c = c ;
  s._N = N ;
#pragma omp parallel for\ private(i)
   for (i=0; i<N; i++) {
     c[i] = a[i] + b[i] ;
   }
}
```

Parallel Region Extraction

- Step 3 consists in extracting parallel region
  - Function *outlining*
  - Opposite from *inlining*

- Algorithm
  - Selection of parallel region block (set of statements)
  - In our example: loop with its body
  - Creation of new function named `omp1` (symbol should be unique)
  - Arguments
    - Temporary structure containing data values

Parallel Region Extraction

```
struct _s {
   double * _a ;
   double * _b ;
   double * _c ;
   int _N ;
 } ;


void vecAdd( double * a,
  double * b,
  double * c,
  int N ) {
  int i;
  struct _s s ;
  s._a = a ;
  s._b = b ;
  s._c = c ;
  s._N = N ;
  GOMP_start_parallel(omp1, &s);
}
```

```
void omp1( struct _s * s) {
  int i;
  double * a ;
  double * b ;
  double * c ;
  int N ;
  int min ;
  int max ;

  // Compute iteration
  // bounds
  min = … ;
  max = … ;

  for (i=min; i<max; i++) {
     c[i] = a[i] + b[i] ;
   }
}
```

Data Flow Restoration

- Call to internal OpenMP function to start parallel region
- In practice, threads are created once
- Input argument of function
  - Temporary structure
- Need to add code to assign variable with correct value (based on temporary structure)

Data Flow Restoration

```
void vecAdd( double * a,
  double * b,
  double * c,
  int N ) {
  int i;
  struct {
    double * _a ;
    double * _b ;
    double * _c ;
    int _N ;
  } s ;
  s._a = a ;
  s._b = b ;
  s._c = c ;
  s._N = N ;
  GOMP_start_parallel(omp1, &s);
}
```

```
void omp1( struct _s * s) {
  int i;
  double * a ;  double * b ;
  double * c ;  int N ;
  int min ;
  int max ;

  // Compute iteration
  // bounds
  min = … ;
  max = … ;

  a = s->_a ;
  b = s->_b ;
  c = s->_c ;
  N = s->_N ;

  for (i=min; i<max; i++) {
    c[i] = a[i] + b[i] ;
  }
```

How runtime can manage threads?

API POSIX

- Programming interface for thread creation and management
  - Fork/join model

- Thread creation

```
pthread_create( pthread_t * thread,
    pthread_attr_t * attribut,
    void *(*routine) (void *),
    void * argument )
```

  - New execution flow starts by calling the target routine with specified argument

- Wait for thread to finish

```
pthread_join( pthread_t thread, void ** resultat )
```

- End of current thread

```
pthread_exit( void * resultat ).
```

API POSIX

- Send a signal to a thread

  `pthread_kill( pthread_t thread, int nu_du_signal )`

- Stop current execution flow to let another thread be scheduled

  `sched_yield()` ou `pthread_yield()`

- Thread ID

  `pthread_self()`

POSIX Example

```c
#include <pthread.h>

int NB_THREADS;
void* run(void* arg){
    long rank;
    rank = (long)arg;
    printf("Hello, I'am %ld (%p)\n",rank,pthread_self());
    return arg;
}
int main(int argc, char** argv){
    pthread_t* pids;
    long i;
    NB_THREADS=atoi(argv[1]);
    pids =
   (pthread_t*)malloc(NB_THREADS*sizeof(pthread_t));
    for(i = 0; i < NB_THREADS; i++){
        pthread_create(&(pids[i]),NULL,run,(void*)i);
    }
    for(i = 0; i < NB_THREADS; i++){
        void* res;
        pthread_join(pids[i],&res);
        fprintf(stderr,"Thread %ld Joined\n",(long)res);
        assert(res == (void*)i);
    }
return 0;
}
```

```
$ gcc -o test test.c -pthread

$ ./test 4
Hello, I'am 0 (0xb785db70)
Thread 0 Joined
Hello, I'am 1 (0xb705cb70)
Thread 1 Joined
Hello, I'am 2 (0xb685bb70)
Thread 2 Joined
Hello, I'am 3 (0xb605ab70)
Thread 3 Joined

$ ./test 4
Hello, I'am 2 (0xb6860b70)
Hello, I'am 3 (0xb605fb70)
Hello, I'am 1 (0xb7061b70)
Hello, I'am 0 (0xb7862b70)
Thread 0 Joined
Thread 1 Joined
Thread 2 Joined
Thread 3 Joined
```

POSIX Example

```
#include <pthread.h>

int NB_THREADS;
void* run(void* arg){
  long rank;
  rank = (long)arg;
  printf("Address of rank (%p) and NB_THREADS (%p)\n",
    &rank,&NB_THREADS );
  return arg;
}
int main(int argc, char** argv){
  pthread_t* pids;
  long i;
  NB_THREADS=atoi(argv[1]);
  pids =
   (pthread_t*)malloc(NB_THREADS*sizeof(pthread_t));
  for(i = 0; i < NB_THREADS; i++){
      pthread_create(&(pids[i]),NULL,run,(void*)i);
  }
  for(i = 0; i < NB_THREADS; i++){
      void* res;
      pthread_join(pids[i],&res);
      assert(res == (void*)i);
  }
return 0;
}
```

```
$ gcc -o test2 test2.c -pthread
$ ./test2 4
Address of rank (0xb788e38c) and
NB_THREADS (0x804a06c)
Address of rank (0xb708d38c) and
NB_THREADS (0x804a06c)
Address of rank (0xb688c38c) and
NB_THREADS (0x804a06c)
Address of rank (0xb608b38c) and
NB_THREADS (0x804a06c)
```

Barrier Transformation

- How an explicit barrier is lowered?

```
#include <omp.h>
#include <stdio.h>

int main()
{
  #pragma omp barrier
}
```

```
;; Function main (main,
funcdef_no=0, decl_uid=2298,
symbol_order=0)

main ()
{

<bb 2>:
__builtin_GOMP_barrier ();
 return;

}
```

Function call

Master Transformation

- How a master construct is lowered?

```
#include <omp.h>
#include <stdio.h>

int main()
{
  #pragma omp master
    printf( "Hello\n" )
;
}
```

```
main ()
{
  int D.2302;

  <bb 2>:
  D.2302 =
__builtin_omp_get_thread_num ();
  if (D.2302 == 0)
    goto <bb 3>;
  else
    goto <bb 4>;

  <bb 3>:
  __builtin_puts (&"Hello"[0]);

  <bb 4>:
  return;

}
```

If statement

Single Transformation

- How a single construct is lowered?

```
#include <omp.h>
#include <stdio.h>

int main()
{

  #pragma omp single
    printf( "Hello\n" ) ;

}
```

```
main ()
{
  _Bool D.2304;

  <bb 2>:
  D.2304 =
     __builtin_GOMP_single_start ();
  if (D.2304 == 1)
    goto <bb 3>;
  else
    goto <bb 4>;

  <bb 3>:
  __builtin_puts (&"Hello"[0]);

  <bb 4>:
  __builtin_GOMP_barrier ();
  return;

}
```

If statement + barrier

Single Transformation

- How to handle the nowait clause?

```
#include <omp.h>
#include <stdio.h>

int main()
{

  #pragma omp single \
       nowait
    printf( "Hello\n" )
;

}
```

```
main ()
{
  _Bool D.2304;

  <bb 2>:
  D.2304 =
__builtin_GOMP_single_start ();
  if (D.2304 == 1)
    goto <bb 3>;
  else
    goto <bb 4>;

  <bb 3>:
  __builtin_puts (&"Hello"[0]);

  <bb 4>:
  return;

}
```

If statement + barrier

Loop Transformation

- How to transform a simple loop with a static schedule?

```
#include <omp.h>

void g( int i ) ;

void f(int N)
{
int i ;

#pragma omp for schedule(static)
for ( i = 0 ; i < N ; i++ )
       g(i) ;

}
```

Loop Transformation

```
f (int N)
{
  // Local declarations

  <bb 2>:
  N.0 = N;
  D.2307 = __builtin_omp_get_num_threads ();
  D.2308 = __builtin_omp_get_thread_num ();
  q.1 = N.0 / D.2307;
  tt.2 = N.0 % D.2307;
  if (D.2308 < tt.2)
    goto <bb 3>;
  else
    goto <bb 4>;

  <bb 3>:
  tt.2 = 0;
  q.1 = q.1 + 1;

<bb 4>:
  D.2311 = q.1 * D.2308;
  D.2312 = D.2311 + tt.2;
  D.2313 = D.2312 + q.1;
  if (D.2312 >= D.2313)
    goto <bb 7>;
  else
    goto <bb 5>;

  <bb 5>:
  i = D.2312;

  <bb 6>:
  g (i);
  i = i + 1;
  if (i < D.2313)
    goto <bb 6>;
  else
    goto <bb 7>;

  <bb 7>:
  __builtin_GOMP_barrier ();
  return;

}
```

Local computation

Loop Transformation

- How to transform a simple loop with a dyanmic schedule?

```
#include <omp.h>

void g( int i ) ;

void f(int N)
{
int i ;

#pragma omp for \
      schedule(dynamic)
for ( i = 0 ; i < N ; i++ )
       g(i) ;

}
```

Loop Transformation

```
f (int N)
{
// Local declarations

  <bb 2>:
  N.0 = N;
  D.2309 = (long int) N.0;
  D.2310 =
__builtin_GOMP_loop_dynamic_start (0,
D.2309, 1, 1, &.istart0.1, &.iend0.2);
  if (D.2310 != 0)
    goto <bb 3>;
  else
    goto <bb 6>;

  <bb 3>:
  .istart0.3 = .istart0.1;
  i = (int) .istart0.3;
  .iend0.4 = .iend0.2;
  D.2313 = (int) .iend0.4;

 <bb 4>:
  g (i);
  i = i + 1;
  if (i < D.2313)
    goto <bb 4>;
  else
    goto <bb 5>;

  <bb 5>:
  D.2314 = __builtin_GOMP_loop_dynamic_next
(&.istart0.1, &.iend0.2);
  if (D.2314 != 0)
    goto <bb 3>;
  else
    goto <bb 6>;

  <bb 6>:
  __builtin_GOMP_loop_end ();
  return;

}
```

Runtime calls to get chunks

Task Transformation

```
int main()
{
  #pragma omp parallel
  {
    #pragma omp task
    {
      printf( « Hello « \n ) ;
    }
  }
  return 0 ;
}
```

Additional function outlining

```
main._omp_fn.1 (void * .omp_data_i) {
  <bb 4>
  printf ("Hello\n");
  return;
}

main._omp_fn.0 (void * .omp_data_i) {
  <bb 10>
  __builtin_GOMP_task (main._omp_fn.1, 0B, 0B, 0,
    1, 1, 0, 0B, 0);
  return;
}

main () {
  int D.2067;
  <bb 2> [0.00%]:
  __builtin_GOMP_parallel (main._omp_fn.0, 0B, 0,
0);
  D.2067 = 0;
<L0> [0.00%]:
  return D.2067;
}
```

Compilers & Runtime

- Compilers transform source code and add function calls
  - Function bodies are located inside a runtime library (shipped with the compiler package)
- What are the differences between compilers?
  - May choose different internal data structures and algorithms
- Main differences between GNU and INTEL task implementations
  - GNU creates one list for the whole team (centralized list)
    - Sorting tasks inside this list help scheduling
  - INTEL creates one list per thread
    - Work stealing between threads for scheduling

Performance

Performance

- Parallel programming may lead to poor performance
  - Not easy to debug and just make it work!
  - Overhead might appear almost everywhere!

- Performance can be shaped by
  - Underlying architecture (number of cores, frequency…)
  - OpenMP implementation
  - Compiler code generation
  - How parallelism is expressed and exploited

- Profiling application may help…

Profiling

- Getting the right application profil is important to focus on parts of interest
- OpenMP proposes 2 functions:

1. `double omp_get_wtime(void);`
   Time in seconds
2. `double omp_get_wtick(void);`
   Counter accuracy

- Can be used as MPI profiling functions
- Values are coherent on the same core
  - May depend on frequency and/or system time

Speed-up

- Profiling helps getting parallel time spent inside some part of the code

- Speed-up can be estimated on parallel applications by comparing sequential time and parallel time

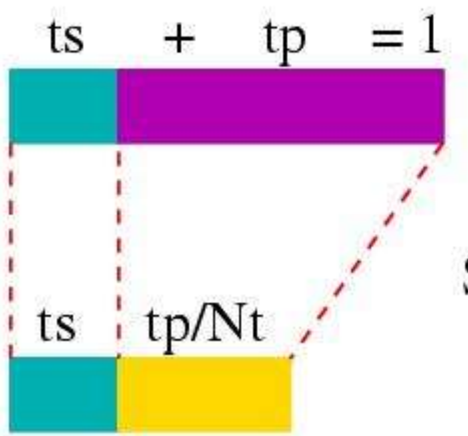
- Speed-up S(n)
  - Depends on the number of threads/tasks
  - Related to sequential time and parallel time
  - Ideally: S(x)=x

- But ideal speedup may not be possible
  - Some parts of code may not be parallelized
    - For example, outside parallel regions
  - Some parts of code is serialized
    - For example, barriers

Amdahl's Law

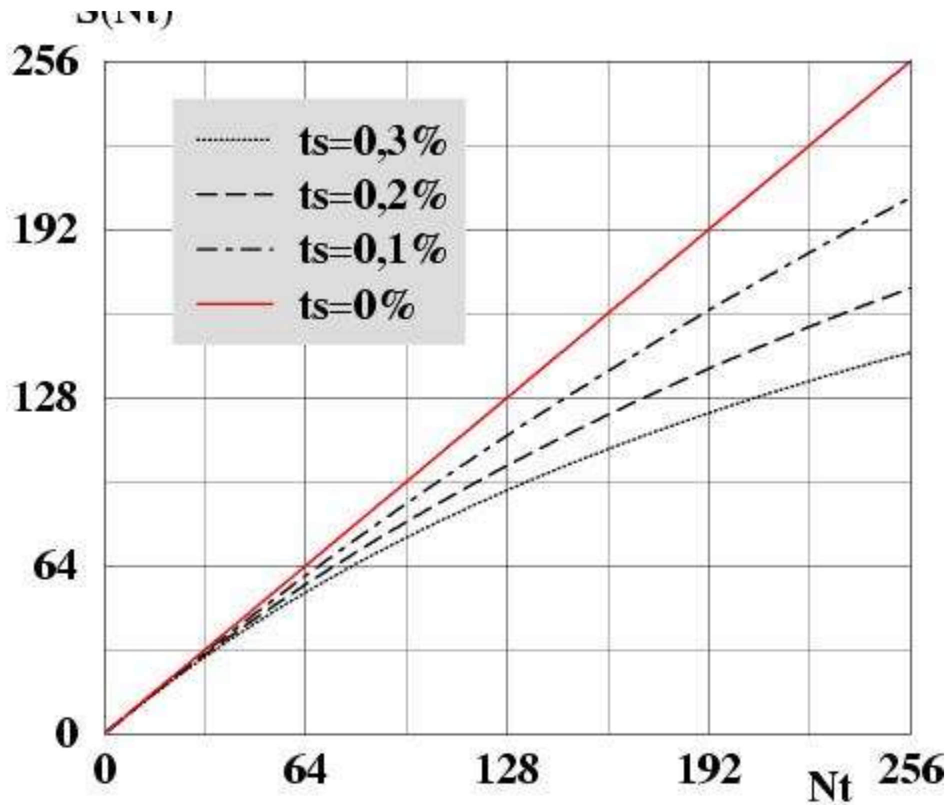
- Consider the following parameters
  - ts: part of code that is serialized or sequential
  - tp: part of code that scales perfectly

- Amdahl's law:



$$S(Nt) = \frac{1}{ts + \frac{tp}{Nt}}$$

Amdahl's Law



- If only 0,3% of code is serial (ts)
  - ➔ Barely possible to get more than 50% parallel efficiency!

Best Practices

- Parallel Regions

- Fork/join model involves penalty
  - → See Amdahl's law

- Solution
  - Coarse-grain parallelism
  - Merge parallel region when possible

Best Practices

- Workshare

- Load imbalance
  - → Test multiple schedule including `dynamic`
  - Be careful to chunk size

- Coarse parallelism in loop nest
  - → Workshare the outermost loop

Best Practices

- Synchronizations

- Use the right synchronization
  - Atomic, locks or critical?

- Avoid useless synchronization
  - Rely on `nowait` clause when possible
  - Remove redundant barriers

Best Practices

- Memory Accesses
- NUMA accesses can be costly
  - Perform allocation (first touch policy) the same way the compute part will work
- Sharing may appear on cache line
  - False sharing
  - Avoid by padding or changing data layout

Best Practices

- Binding

- By default, threads may change core
  - Based on scheduler policy

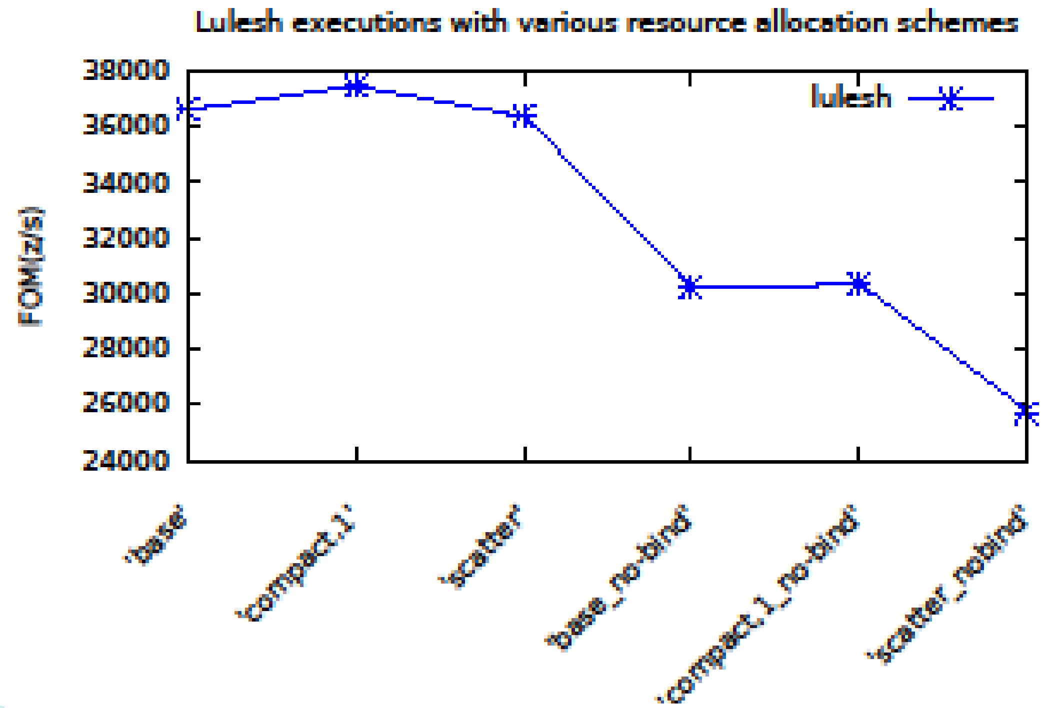
- Thread pinning may increase performance
  - Keep data locality (NUMA and cache)

- Notion of affinity

Placement Experiments

- Experiments of running LULESH benchmark on 4 nodes w/ dual-socket 16-core Haswell (w/ hyperthreading)
  - W/ and w/out KMP_AFFINITY (OpenMP)
  - W/ and w/ binding for MPI (from SLURM)

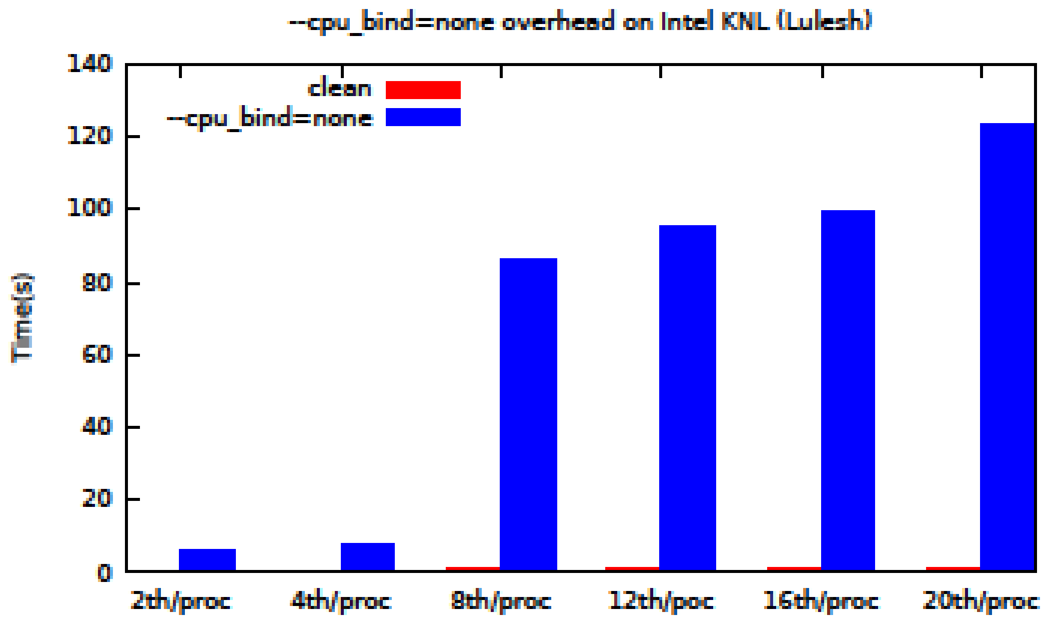
- Results in Figure of Merit (higher is better)



Placement Experiments

- Experiments of running LULESH benchmark on Intel Xeon Phi KNL
  - W/ and w/ binding for MPI (from SLURM)

- Results in time (lower is better)



Task Parallelism

- Task advantages for parallelism
  - Tasks can express a large amount of parallelism
  - Tasks are asynchronous which may lead to good performance

- Main parallelism pitfalls
  - To enable parallelism tasks should be created within a parallel region
  - If orphaned, a task directive will executed right away
  - Be careful of mono-producer vs. multi-producer mode → don't create useless tasks
  - Be careful about data flow (especially `firstprivate` variable which will capture values during task creation and not during task scheduling)

Task Deadlocks

- Task can be created by one thread and scheduled by another thread
  - If untied, it can be scheduled by different threads during its lifetime
- Thus there are no guarantee about parallel task execution
  - Except during a `barrier` or a `taskwait` construct
- Avoid any workshare constructs or barriers within a task body
  - Inside the static body and the dynamic extent

Task Granularity

- Creating and scheduling a task might take some time
  - As seen previously with compiler lowering (function calls, data packaging…)
  -
- Be careful about the granularity
  - Need some amount of work for leveraging tasks
- In case of very small tasks, use cutoff
  - Through `if` clause and/or `final` clause

Data Dependencies

- Data dependencies can expressed very fine relationship between tasks
- But it is valid only between tasks that share the same parent
  - Sibling tasks

- Be careful of data dependencies in following situations
  - Nested tasks
  - Multiple producers