TaskGraph User Manual

Peter Collingbourne

September 30, 2004

Contents

Listings

Chapter 1

Introduction

The TaskGraph library [2, 1] is a tool written in C++ for use in Multi-Stage Programming (MSP). The main focus of the library is to provide a way for a subroutine in C to be generated at runtime, by using standard C syntax. Generated C subroutines are known as *TaskGraphs*. Since the subroutine is generated at runtime, a TaskGraph can be created that is *specialised* for a particular task or set of input values. Figure 1.1 shows how a typical TaskGraph is created.

A cornerstone of the TaskGraph library is its backend *IR library*. This IR library has two main roles: to store the AST while constructing the program and to perform loop transformations and other optimisations (see Chapter 4). Before using TaskGraph you will have to decide which IR library you want to use based on your needs. The current implementation supports two compiler research projects which can be used as IR libraries: namely SUIF [3] and ROSE. The current main user-visible difference between these two libraries is their loop transformation API. If you decide to use TaskGraph's loop transformation

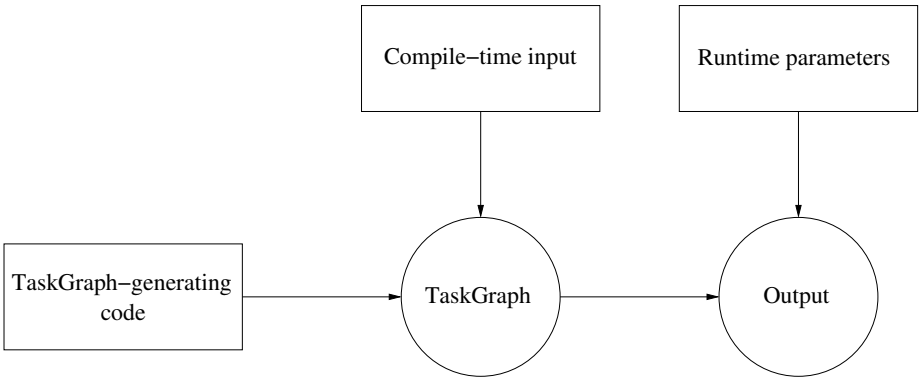

Figure 1.1: Transformation of TaskGraph code

abilities you will have to write your code to either the SUIF or ROSE interface.

1.1 License

Chapter 2

Installation

Installation should be an easy process. Unpack the distribution, which is typically named something like `2004-09-14-taskgraph.tar.bz2`, and follow the instructions found in the `README` file found in the root directory of the distribution. Listing 2.1 gives typical shell commands to issue.

2.1 Requirements

You will need to fetch and install either SUIF or ROSE in order to compile and use the TaskGraph library. SUIF is freely available for download from the Internet [1] but ROSE has not yet been publically released. If you would like to use the ROSE IR you will need access to ROSE and you must be licensed for source access to the EDG front-end (though we hope to avoid this in future).

For instructions on compiling SUIF and ROSE for use with the TaskGraph library please see the `README` file located in the `docs/suif` and `docs/rose` directories respectively.

[1] `http://suif.stanford.edu/suif/suif1/`

Listing 2.1: Shell commands to execute

```
tar xjf 2004-09-14-taskgraph.tar.bz2
cd 2004-09-14-taskgraph
more README
```

Chapter 3

Tutorial

This is a tutorial description of the TaskGraph library. It begins with a simple example and leads up to more complex ones.

3.1 Notice

All generated code and output in this manual are generated by the ROSE back end.

3.2 Creating a Program that Uses TaskGraph

The process to create a new program that uses the library is currently not very clean, but will be improved upon in the future. Make a new directory in the `examples` directory and create a `Makefile` within it – we will assume you will entitle your example `myexample`. Typical contents of the `Makefile` are shown in Listing 3.1.

Now create your program in the file `myexample.cc`. If you would like to split your program up into several source files, add them to the space-delimited list `TG_EXAMPLE_SOURCES` in the `Makefile`.

Listing 3.1: Example contents of `Makefile`

```
TG_EXAMPLE = myexample
TG_EXAMPLE_SOURCES = myexample.cc

all: tg-example

include ../Makefile.example
```

Listing 3.2: Simple Example

```
1  #include <TaskGraph>
2
3  using namespace tg;
4
5  typedef TaskGraph< Par<int>, Ret<void> > TutorialTaskGraph;
6
7  int main()
8  {
9    TutorialTaskGraph t;
10   int a = 0;
11   taskgraph ( t, tuple1 ( x ) ) {
12     x = x + 1;
13   }
14   t.compile ( tg::GCC, true );
15   printf ( "a before = %d\n", a );
16   t.execute ( a );
17   printf ( "a after = %d\n", a );
18 }
```

3.3 A Simple Example

A simple example is shown in Listing 3.2. This is possibly the most simple example of a TaskGraph. We will now look at what significant lines of this code are doing.

Line 1 includes a header file containing all the declarations needed for a user of the TaskGraph library. Any program that uses TaskGraph must include this file.

Line 3 imports the `tg` namespace in which several TaskGraph symbols are defined. This will be required in every TaskGraph program unless you prefix everything with `tg::`.

Line 5 defines a TaskGraph type (known as `TutorialTaskGraph`) with one parameter of type `int`, and of return type `void`.

Line 9 declares a TaskGraph of type `TutorialTaskGraph`, with name `t`. At this point the TaskGraph has not yet had any code added to it – attempting to run it will cause an error.

Line 11 is very important. It begins the definition of this TaskGraph. That is to say, it prepares the TaskGraph's AST to have nodes added to it. It also defines the temporary names of the parameters of the TaskGraph, so that they can be used within the TaskGraph code, much like a function definition in C.

Line 12 is another important line. It adds the instruction node "`x = x + 1`" to this TaskGraph. That is to say, it adds an instruction to the TaskGraph that increments the value of the first parameter passed to the TaskGraph.

Line 13 closes the definition of the TaskGraph, so that it can be compiled.

Listing 3.3: Simple Example Generated Code

```
extern void *taskGraph_0(void **params)
{
  int *(((x))) = ((int *)(params[0]));
  *(((x))) = *(((x))) + 1;
}
```

Listing 3.4: Simple Example Output

```
a before = 0
a after = 1
```

Line 14 compiles the TaskGraph so that it can be executed. The two parameters `tg::GCC` and `true` respectively tell the TaskGraph library to compile with the GNU C compiler, and to keep the generated sources (which will have a name such as "`TaskGraph-Source-5G8JkP`") in the `/tmp` directory so that they can be inspected.

Line 16 executes the TaskGraph with parameter `a`. Note that this parameter is type safe [4], so `a` must be of type `int`.

Listing 3.3 shows source code that is generated by the ROSE backend, and Listing 3.4 shows the output of the program.

Notice that the value of `a` has incremented by one during the running of the TaskGraph. This demonstrates that parameters to a TaskGraph are passed by *reference* not by value.

Important Note The block of code that consists of the definition of the TaskGraph (i.e. that delimited by `taskgraph(...) { ... }` is NOT part of the static definition of the TaskGraph. It is instead a piece of code that runs in order to create instruction nodes within the TaskGraph. This is a very important point to remember and may cause unexpected effects. For example, if one placed the code `printf("Hello world!");` in the `taskgraph` block it would not represent a `printf` call in the TaskGraph, instead it would cause the message to be printed while the program is constructed. The same goes for any statements that assign values to variables not declared in the TaskGraph (i.e. their values will be set during construction). Similarly, any statement which reads a value from a non-TaskGraph variable in the `taskgraph` will not be affected by changes to the variable after its construction. This is because the value of the variable is statically placed in the abstract syntax tree at construction time.

Listing 3.5: Specialising Example

```
1  #include <TaskGraph>
2
3  using namespace tg;
4
5  typedef TaskGraph< Par<int>, Ret<int> > TutorialTaskGraph;
6
7  int main(int argc, char **argv)
8  {
9    TutorialTaskGraph t;
10   int m = atoi ( argv[1] );
11   int c = atoi ( argv[2] );
12
13   taskgraph ( t, tuple1 ( x ) ) {
14     tReturn ( m*x + c );
15   }
16
17   t.compile ( tg::GCC, true );
18
19   for (int a = 1; a <= 10; a++ ) {
20     printf ( "a = %d, f(a) = %d\n", a, t.execute ( a ) );
21   }
22 }
```

3.4 A Specialising Example

The next example we will show is one that specialises the TaskGraph generated using parameters obtained at runtime. Consider the example given in Listing 3.5:

The most significant line of note is line 14. The variables `m` and `c` are retrieved from the command line and compiled into the expression "`m*x + c`" whereas `x` is retained as a reference to the first parameter of the TaskGraph. The `tReturn` in line 14 adds a "`return x`" instruction to the AST where `x` is the parameter to `tReturn`. When `t.execute` is called it returns the value returned by the TaskGraph. So the `for` loop on lines 19-21 displays the value of $mx + c$ for values of x between 1 and 10 inclusive.

Listing 3.6 shows the code generated for $m = 2$ and $c = 3$ and Listing 3.7 shows the output of the program.

Listing 3.6: Specialising Example Generated Code

```
extern void *taskGraph_0(void **params)
{
  int *(((x))) = ((int *)(params[0]));
  return (void *)(2 * *(((x))) + 3);
}
```

Listing 3.7: Specialising Example Output

```
a = 1, f(a) = 5
a = 2, f(a) = 7
a = 3, f(a) = 9
a = 4, f(a) = 11
a = 5, f(a) = 13
a = 6, f(a) = 15
a = 7, f(a) = 17
a = 8, f(a) = 19
a = 9, f(a) = 21
a = 10, f(a) = 23
```

3.5 An Optimising Example

The example in Listing 3.8 displays some of the "loop processing" functionality available to users of the TaskGraph library. The code uses `ifdef` statements to shield itself from the different loop transformation APIs, so the example should work in both SUIF and ROSE.

Listing 3.8: Optimising Example

```
1  #include <TaskGraph>
2
3  using namespace tg;
4
5  #define MATRIXSIZE 1024
6  #define TILESIZE 64
7
8  typedef float MATRIX[MATRIXSIZE][MATRIXSIZE];
9
10 void initMatrix ( float m[][MATRIXSIZE], float val )
11 {
12   for ( int x = 0 ; x < MATRIXSIZE ; x++ )
13   {
14     for ( int y = 0 ; y < MATRIXSIZE ; y++ )
15     {
```

```
16        m[x][y] = val;
17      }
18    }
19  }
20
21  typedef TaskGraph< Par< MATRIX, MATRIX, MATRIX >,
22                      Ret< void > > MMTaskGraph;
23
24  MATRIX a, b, c;
25
26  int main()
27  {
28    initMatrix ( a, 1.0f );
29    initMatrix ( b, 1.0f );
30    initMatrix ( c, 0.0f );
31
32    MMTaskGraph t;
33    taskgraph ( t, tuple3 ( x, y, z ) ) {
34      tVar ( int, i );
35      tVar ( int, j );
36      tVar ( int, k );
37
38      tFor ( i, 0, MATRIXSIZE - 1 ) {
39        tFor ( k, 0, MATRIXSIZE - 1 ) {
40          tFor ( j, 0, MATRIXSIZE - 1 ) {
41            z[i][j] += x[i][k] * y[k][j];
42          }
43        }
44      }
45    }
46
47    t.print();
48
49  #ifdef USE_SUIF1_IR
50    TileSettings tile ( LoopIdentifier ( 1, 1 ), 2, TILESIZE );
51    t.applyOptimisation( "tile", &tile );
52  #endif
53
54  #ifdef USE_ROSE_IR
55    t.blockInner( TILESIZE );
56  #endif
57
58    t.print();
59
60    t.compile ( tg::GCC, true );
61    t.execute ( a, b, c );
```

62

63 }

You may recognise this as being standard matrix multiply code, and indeed it is a simplified version of the full matrix multiply example found in `examples/matrixmult` in the TaskGraph distribution. We will now explain a few of the extra features shown in this example.

The `tVar` lines on lines 34-36 define three general-purpose variables (not parameters) that will be used at *runtime* (i.e. when the TaskGraph is running). We will henceforth refer to these as *runtime variables*. In this instance they will be used as `for` loop control variables.

The `tFor` lines (38-40) create new `for` loop structures in the AST of the TaskGraph. They do not cause the code within them to be executed in a `for` loop at construction time but they cause `for` loops to enclose any statements added to the AST while the `tFor` statement is still in scope. The parameters to the `tFor` construction are: a variable (this needs to be a runtime variable, rather than a 'real' C variable), and two expressions representing the lower and upper bounds of the loop. Both of these bounds are *inclusive*.

The `t.print()` statements on lines 47 and 58 print out the current contents of the TaskGraph, as formatted C code. This can be very useful while debugging in order to check what happens to the code at each stage of optimisation. In this instance it allows us to view the 'before' and 'after' states of the generated program.

Lines 50-51 demonstrate the technique of loop tiling in SUIF. For TaskGraph's SUIF interface, each optimisation is specified using a name and a `Settings` object. In the case of tiling, the `TileSettings` object is used. The constructor for this object takes the following arguments: a `LoopIdentifier` specifying the most outer loop to tile, the number of loops to tile and the tile size. In this case the loop specified is the second loop (the `k` loop) and the number of loops to tile is 2, so both the `k` and `j` loops are tiled.

Line 55 shows loop tiling in ROSE (note that internally, ROSE refers to loop tiling as *blocking*, which is why the method is called `blockInner`). As you can see the interface is much cleaner; you simply specify the tile size. This is because the ROSE library uses a technique known as *profitability analysis* in order to traverse the entire TaskGraph and find suitable loops to tile. This is one of the main reasons why the decision was made to provide different APIs for the SUIF and ROSE loop transformations.

Listing 3.9 shows the unoptimised matrix multiply code output by this program, and Listing 3.10 shows the code after it has been optimised by ROSE.

Listing 3.9: Optimising Example Unoptimised Output

```
extern void *taskGraph_0(void **params)
{
  int k;
  int j;
  int i;
  float (*(((z))))[1024] = ((float (*)[1024])(params[2]));
  float (*(((y))))[1024] = ((float (*)[1024])(params[1]));
  float (*(((x))))[1024] = ((float (*)[1024])(params[0]));
  for (i = 0; i <= 1023; i += 1) {
    for (k = 0; k <= 1023; k += 1) {
      for (j = 0; j <= 1023; j += 1) {
        (((z)))[i][j] = (((z)))[i][j] + (((x)))[i][k] * (((y)))[k][j];
      }
    }
  }
}
```

Listing 3.10: Optimising Example Optimised Output

```
Finished building EDG AST, now build the SAGE AST ...
/* AST Fixes started. */
/* AST Fixes reset pointers */
/* AST Fixes finished */

inline static int min(int a,int b)
{
  return a < b?a:b;
}


extern void *taskGraph_0(void **params)
{
  int _var_1;
  int _var_0;
  int k;
  int j;
  int i;
  float (*(((z))))[1024] = ((float (*)[1024])(params[2]));
  float (*(((y))))[1024] = ((float (*)[1024])(params[1]));
  float (*(((x))))[1024] = ((float (*)[1024])(params[0]));
  for (_var_1 = 0; _var_1 <= 1023; _var_1 += 64) {
    for (_var_0 = 0; _var_0 <= 1023; _var_0 += 64) {
      for (i = 0; i <= 1023; i += 1) {
        for (k = _var_1; k <= min(1023,_var_1 + 63); k += 1) {
          for (j = _var_0; j <= min(1023,_var_0 + 63); j += 1) {
            (((z)))[i][j] = (((z)))[i][j] + (((x)))[i][k] * (((y)))[k][j];
          }
        }
      }
    }
  }
}
```

Chapter 4

Loop Transformations

Here we give a brief description of the loop transformations available to a user of the TaskGraph library. For instructions on how to invoke each transformation, please refer to the Reference section in Chapter 8.

4.1 Fusion

The TaskGraph is traversed for loops that are adjacent to each other and perform the same functionality when their inner instructions are concatenated or 'fused' together. This functionality requires that the two loops have the same number of iterations (the number needs not be constant). For example, the loops in Listing 4.1 can be fused.

4.2 Unrolling

Loop unrolling is the replacement of a loop with the explicit instructions within that loop for each iteration of the loop, with the value of the loop control variable at that iteration substituted in. This technique relies on the number of iterations being fixed and known at compile time, so the optimiser can unroll the loop to the required number of repetitions.

Listing 4.1: Fusable Loops

```
for (int i = 1; i <= 10; i++) {
        x[2 * i] = x[2 * i + 1] + 2;
}

for (int i = 1; i <= 10; i++) {
        x[2 * i] = x[2 * i] + i;
}
```

Listing 4.2: Loop that can be Split in Half

```
for (int i = 0; i <= 1000; i++) {
        x[i] = x[1000-i] * 2;
}
```

4.3 Fission/Distribution

Loop distribution is a kind of partial unrolling. It can be used to separate out one iteration of a loop (for example the first iteration, which may contain conditional initialization code which would not have to be tested for in subsequent loops) or to split a loop in two or more pieces so that each piece can be executed in parallel. Listing 4.2 shows a loop that can be split in half.

4.4 Skewing

Loop skewing is a technique that can sometimes lead to better loop performance or parallelism. For example consider the loop in Listing **??**. It does not immediately appear to be parallel. However we can apply the *skewing matrix*

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

to it and we will get the loop in Listing **??**. Now we can see that the inner loop is parallelisable.

4.5 Interchange

Loop interchange is a technique that improves performance by improving *cache locality*. That is to say, it restructures a loop to improve the chances that a piece of data that is accessed in the loop will be in the processor's level 1 or 2 cache, and thus lead to faster memory access times. The canonical example of this is matrix multiplication. Consider the loop in Listing 4.3. It turns out that better performance can be achieved by interchanging the j and k loops, as in Listing 4.4.

4.6 Tiling/Blocking

Loop tiling is also used to improve cache locality. It does this by structuring a loop so that it iterates through 'tiles' rather than strips of the data set as it is processed. Again the classical example of loop tiling is matrix multiplication. Unoptimised code is shown in Listing 4.3 and an optimised version with interchanging performed and the two inner loops tiled is shown in Listing 4.5.

Listing 4.3: Unoptimised Matrix Multiply

```
for (int i = 0; i <= 511; i++) {
  for (int j = 0; j <= 511; i++) {
    for (int k = 0; k <= 511; i++) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

Listing 4.4: Interchanged Matrix Multiply

```
for (int i = 0; i <= 511; i++) {
  for (int k = 0; k <= 511; i++) {
    for (int j = 0; j <= 511; i++) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

Listing 4.5: Tiled Matrix Multiply

```
for (i = 0; i <= 511; i++) {
  for (k_tile = 0; k_tile <= 511; k_tile += 64) {
    for (j_tile = 0; j_tile <= 511; j_tile += 64) {
      for (k = k_tile; k <= min(511, 63 + k_tile); k++) {
        for (j = max(0, j_tile); j <= min(511, 63 + j_tile); j++) {
          c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
      }
    }
  }
}
```

Chapter 5

User-defined Functions

We describe the mechanism for defining and using user-defined functions within a TaskGraph.

5.1 About User-Defined Functions

A user-defined function will allow you to use a standard C function from a piece of generated TaskGraph code. The TaskFunction facility in TaskGraph provides the ability to do this. The functionality works by defining a function with a special name (usually of the form tMyfunction, where myfunction is the name of the function you would like to call). When this function is called during the construction of your TaskGraph, it creates an expression that represents a call to your function (or if your function returns void it simply creates a statement representing a call to your function).

5.2 Predefined Functions

5.2.1 tPrintf

The tPrintf function is TaskGraph's built in TaskGraph wrapper for the printf function. It takes a variadic number of arguments and does not return anything. It allows you to print messages from your TaskGraph code (possibly for debugging purposes).

5.2.2 tSqrt

tSqrt is TaskGraph's wrapper for the sqrt function. It takes one float argument and returns a float. It allows you to compute square roots in your TaskGraph code.

5.3 Creating Functions

We describe how to create your own user-defined function that can be usable from a TaskGraph. First define a function that creates the function-calling statement or expression. This is not strictly necessary but it helps readability. Here is the general template for such a function:

```
TaskExpression tMyfunction(TaskExpression &a1,
                           TaskExpression &a2, ...) {
  static TaskFunctionX<r, t1, t2, ...> f("myfunction");
  return func.call(a1, a2, ...);
}
```

Replace the following parts of the template:
`tMyfunction` – replace with the name of your function. If your function is a void function you can replace the `TaskExpression` before this with `void`
`"myfunction"` – replace with the name of your function in quotes
`a1`, `a2`, ... – repeat as many times as your function has arguments
`TaskFunctionX` – replace with `TaskFunction` followed by the number of arguments of your function, e.g. `TaskFunction3`
`r` – replace with the return type of your function
`t1`, `t2`, ... – replace with the argument types of your function
`return` – replace this with nothing if your function returns `void`

For the TaskFunction functionality to work you must ensure that your program is linked with the `-Wl,--export-dynamic` option. If you are creating a new program in the `examples` directory, ensure that `TG_EXAMPLE_LDFLAGS` is defined to contain this flag. The procedure for user-defined functions may become easier to use in the future.

Chapter 6

Examples

We now describe briefly each example provided with the TaskGraph distribution.

6.1 Image Filter

A common task in image processing applications is to apply a mask to an image. This example can statically compile mask filter values into a specialised version of the code, leading to a much faster implementation (up to 4 times faster). This example can be found in the `examples/imgfilt` directory, in the TaskGraph distribution.

6.2 JIT Virtual Machine Interpreter

This is an interesting example of a virtual machine interpreter that takes bytecode and generates specialised C code to execute the bytecode instructions, similar to a Just In Time (JIT) compiler. This particular example contains instructions for calculating numerical values, in this case 10 factorial (10!). The bytecode interpreter contains C++ and TaskGraph implementations, and typically the TaskGraph implementation outperforms the C implementation by a factor of 2. This example can be found in the `examples/interpreter` directory, in the TaskGraph distribution.

6.3 Matrix Multiplication

This example shows some of the loop processing techniques that are present in the TaskGraph library (or to be more precise, the ROSE/SUIF libraries). It generates a standard matrix multiply implementation using TaskGraph and uses the technique of loop blocking in order to partition it into 64x64 blocks, which leads

to better performance. This example can be found in the `examples/interpreter` directory, in the TaskGraph distribution.

6.4 Iterative Matrix Multiplication

This example is similar to the matrixmult example. It uses the TaskGraph library's TaskIterative technique in order to try several different tiling sizes. There are two TaskGraph implementations of the algorithm here: a hand-tiled implementation and one that relies on the tiling/blocking abilities of the ROSE/SUIF library to do the actual tiling. This example can be found in the `examples/mmiter` directory, in the TaskGraph distribution.

6.5 Ray Tracer

An interesting example - a ray tracer in TaskGraph. This example can specialise a ray tracer to a specific scene and thus render a scene very quickly (up to 6 times as fast as the regular implementation). This example can be found in the `examples/raytracer` directory, in the TaskGraph distribution.

6.6 Zero Finder

This is an example of a program that finds zero solutions in a specific polynomial. The example given is the polynomial $(x-10)(x-9)(x-8)(x-7)(x-6)(x-5)(x-4)(x-3)(x-2)(x-1)$, which is specified as the list `p` in the code. The specialised implementation runs up to 3 times as fast as the standard implementation. This example can be found in the `examples/zeros` directory, in the TaskGraph distribution.

6.7 Other Examples

Some other examples are included and they can be found in the `examples` directory of the TaskGraph distribution: `fft`, `mathintprt`, `morton-gaussseidel`, `mortonmm`, `mortontiling`, `multistage`, `simple`.

Chapter 7

Implementation Notes

Here we give some details on how the TaskGraph library works.

7.1 Macros

Macros are used extensively in the TaskGraph library for the `taskgraph`, `tFor` and other block statements. Any block statements in the TaskGraph library are actually macros that expand to a `for` loop. In the `for` loop a special `BlockEnder` object is created on the stack. When it is created this allows for special actions to be taken in the constructor. For example the `taskgraph` macro initialises the specified TaskGraph's internal data structures and sets it to be the *current TaskGraph*. This means that any new statements are added to it. The `tFor` macro appends a `for` loop to the current TaskGraph and pushes it onto the *statement body stack*. The statement body stack is a structure internal to the TaskGraph that keeps a stack of blocks (such as `for` loops, `if` blocks etc.). Any statements that are added to the TaskGraph are actually appended to the block at the top of the stack. The `for` loop is arranged so that its body only executes once, so any statements within the `taskgraph`, `tFor` or other block statements are executed only once. When the `for` loop finishes the `BlockEnder` goes out of scope and its destructor is called. This allows us to perform cleanup or other activities when the block finishes. For example a `tFor` block would, when closed, pop the `for` loop it represents from the statement body stack, so any statements executed after the `for` loop closes will be added to the 'parent' of the `for` loop.

7.2 Operator Overloading

Operator overloading is used to implement the feature of the TaskGraph library that allows statements that look like ordinary statements (for example, `a = a + 1`) to be converted into statements that append themselves to the current TaskGraph.

7.2.1 Variables and Expressions

A variable in TaskGraph is represented by the `TaskScalarVariable` class, for example a parameter declared by `tParameter` or a variable declared by `tVar`. An expression (such as `(1 + a) / 2`) is represented by a `TaskExpression`. `TaskExpression`s actually represent expression *trees* and a `TaskExpression` is recursively defined as:

```
TaskExpression ::=
   TaskScalarVariable |
   char | short | int | long | float | double |
   TaskExpression + TaskExpression |
   TaskExpression - TaskExpression |
   TaskExpression * TaskExpression |
   TaskExpression / TaskExpression |
   TaskExpression | TaskExpression |
   TaskExpression & TaskExpression |
   TaskExpression << TaskExpression |
   TaskExpression >> TaskExpression |
   TaskExpression == TaskExpression |
   TaskExpression != TaskExpression |
   TaskExpression < TaskExpression |
   TaskExpression > TaskExpression |
   TaskExpression <= TaskExpression |
   TaskExpression >= TaskExpression |
   TaskExpression || TaskExpression |
   TaskExpression && TaskExpression |
   -TaskExpression |
   !TaskExpression |
   ~TaskExpression
```

Note that `TaskScalarVariable`s and standard C numerical types (`int`, `float`, `double`, etc.) are implicitly converted to `TaskExpression`s when they are used in a `TaskExpression` tree. Also a notable omission from this list is the ternary operator (`?:`). This is because it is impossible to overload the ternary operator in C++.

7.2.2 Statements

Many statements in the TaskGraph library are also implicitly created by operator overloading. Whenever a `TaskScalarVariable` is assigned a value (using `=`) it is implicitly replaced by a call to the TaskGraph library that appends the assignment statement to the current TaskGraph. This call will also return a `TaskExpression` representing the variable that was just set, so statements such as `a = b = c` can be constructed (however this will be represented in the TaskGraph as `b = c; a = b`, which is basically the same thing). Whenever an

operation is executed on a `TaskScalarVariable`, such as the post-increment operator `++`, this also appends a statement executing the operation.

Chapter 8

Reference

We now give a reference style documentation to the various features available in the TaskGraph library.

8.1 TaskGraph

Synopsis

`TaskGraph< Par<`$\texttt{T}_1$`, `$\texttt{T}_2$`, `$\texttt{T}_3$`, ..., `$\texttt{T}_N$`>, Ret<R> >`

Description

The general class type of any TaskGraph. Objects of this type represents a TaskGraph with parameter types $\texttt{T}_1$, $\texttt{T}_2$, $\texttt{T}_3$, ... and return type `R`. Newly created TaskGraphs cannot be compiled or used; they must have statements added to them using the `taskgraph` construct.

Restrictions

none

8.2 TaskGraph::compile

Synopsis

```
T.compile(C, K);
```

Description

Compiles the TaskGraph `T`. Before calling this method, the TaskGraph must be constructed using the `taskgraph` construct. The parameter `C` indicates the compiler to use. Possible values for `C` include:

- `tg::GCC` – Compile using the GNU C compiler (gcc).
- `tg::ICC` – Compile using the Intel C compiler (icc).

To use either of these compilers you must ensure that their paths were correctly configured at compile time. Consult the `README` file in the root of the TaskGraph distribution for more information.

If the boolean parameter `K` is true, the TaskGraph library will keep the source files it generates in the `/tmp` directory after compilation. If `K` is false, the source files will be deleted after compilation.

Restrictions

must be called after constructing the TaskGraph

Example code

```
typedef TaskGraph< ... > TG;
...
TG T;
taskgraph ( T, ... ) {
  ...
}
T.compile ( tg::GCC, true );
```

8.3 TaskGraph::print

Synopsis

```
T.print();
```

Description

Prints a C representation of the TaskGraph `T`. The TaskGraph must have previously been created using the `taskgraph` construct.

Restrictions

must be called after constructing the TaskGraph

Example code

```
typedef TaskGraph< ... > TG;
...
TG T;
taskgraph ( T, ... ) {
  ...
```

```
}
T.print();
```

8.4 TaskGraph::execute

Synopsis

`T.execute ( P1, P2, P3, ... );`

Description

Executes a TaskGraph `T`. The TaskGraph must have previously been compiled. Every parameter P_k supplied to this method must be an lvalue of type T_k as defined by `T` (this restriction is due to the fact that parameters are passed by reference so that they can be modified), and the number of parameters supplied to this function must be equal to the number of parameters `N` as defined by `T`.

The return value of the method is that defined by a `tReturn` call in the TaskGraph and is of type `R`, the return type as defined by `T`.

Restrictions

must be called after compiling the TaskGraph

Example code

```
typedef TaskGraph < Par<int, float, double>,
                    Ret<short> > TG;
...
TG T;
int i;
float f;
double d;
short s;
...
T.compile ( tg::GCC, true );
s = T.execute ( i, f, d );
```

8.5 taskgraph

Synopsis

`taskgraph ( G, tuple`N`(P1, P2, P3, ..., PN) ) {`

```
...
}
```

Description

Prepares the TaskGraph `G` to start having nodes attached to it. Any nodes created within the `taskgraph` block are added to the function represented by the `TaskGraph`. The parameter tuple indicates the temporary names of the parameters within the `taskgraph` block. Each parameter name $\texttt{P}_r$ represents the rth parameter to the TaskGraph and will be of type $\texttt{T}_r$ as defined in `G`.

Restrictions

must not appear within another `taskgraph` block

Example code

```
typedef TaskGraph < Par<int, float, double>,
                    Ret<void> > TG;
...
TG t;
taskgraph ( t, param3 ( x, y, z ) ) {
  ...
}
```

Translation

```
void *taskGraph_0(void **params) {
  int *x = params[0];
  float *y = params[1];
  double *z = params[2];
  ...
}
```

8.6 tIf

Synopsis

```
tIf ( E ) {
  ...
}
```

Description

All statements created within the block are placed in an `if` block in the generated code. The condition on the block is the expression `E`.

Restrictions

must appear within a `taskgraph` block

Example code

```
tVar ( bool, x );
tVar ( int, y );
...
tIf ( x ) {
  y++;
}
```

Translation

```
int y;
bool x;
...
if (x) {
  y++;
}
```

8.7 tElse

Synopsis

```
tIf ( ... ) {
  ...
} tElse {
  ...
}
```

Description

Statements created within the block are attached to the `else` block after the preceding `tIf` block.

Restrictions

- must appear within a `taskgraph` block
- must appear after a `tIf` block

Example code

```
tVar ( bool, x );
tVar ( int, y );
...
tIf ( x ) {
  y++;
} tElse {
```

```
  y--;
}
```

Translation

```
int y;
bool x;
...
if (x) {
  y++;
} else {
  y--;
}
```

8.8 tFor

Synopsis

```
tFor ( V, A, Z ) {
  ...
}
```

Description

Statements within the block are placed in a FORTRAN-style `for` loop, where the variable `V` is the loop control variable and the expressions `A` and `Z` are the inclusive lower and upper bounds of the loop.

Restrictions

must appear within a `taskgraph` block

Example code

```
tVar ( int, x );
tVar ( int, y );
tVar ( int, z );
...
tFor ( x, 1, 100 ) {
  tFor ( y, 1, x ) {
    z++;
  }
}
```

Translation

```
int z;
int y;
int x;
...
for ( x = 1; x <= 100; x += 1 ) {
  for ( y = 1; y <= x; y += 1 ) {
    z++;
  }
}
```

8.9 tForStep

Synopsis

```
tFor ( V, A, Z, S ) {
  ...
}
```

Description

Similar to a `tFor`, except the loop increment is `S`, rather than being 1.

Restrictions

must appear within a `taskgraph` block

Example code

```
tVar ( int, x );
tVar ( int, y );
...
tForStep ( x, 1, 100, 2 ) {
  y++;
}
```

Translation

```
int y;
int x;
...
for ( x = 1; x <= 100; x += 2 ) {
  y++;
}
```

8.10 tWhile

Synopsis

```
tWhile ( C ) {
  ...
}
```

Description

Statements within the block are placed in a `while` loop. The while loop condition is the expression `C`.

Restrictions

must appear within a `taskgraph` block

Example code

```
tVar ( int , i );
...
i = 100;
tWhile ( i > 0 ) {
  i--;
}
```

Translation

```
int i;
...
i = 100;
while (i > 0) {
  i--;
}
```

8.11 tReturn

Synopsis

```
tReturn ( V );
```

Description

Adds a statement to the current block that returns the value represented by the expression `V`.

Restrictions

must appear within a `taskgraph` block

Example code

```
tVar ( int , i );
...
tReturn ( i );
```

Translation

```
int i;
...
return i;
```

8.12 tContinue

Synopsis

`tContinue;`

Description

Adds a `continue` statement to the current block.

Restrictions

must appear within a `tWhile` or `tFor` block

Example code

```
tContinue;
```

Translation

```
continue;
```

8.13 tBreak

Synopsis

`tBreak;`

Description

Adds a `break` statement to the current block.

Restrictions

must appear within a `tWhile` or `tFor` block

Example code

```
tBreak;
```

Translation

```
break;
```

8.14 tPrintf

Synopsis

`tPrintf(S, P1, P2, P3, ...);`

Description

Adds a `printf` statement to the current block, in order to print out a message. The parameters to `printf` are `S`, P_1, P_2, P_3, ...

Restrictions

must appear within a `taskgraph` block

Example code

```
tVar ( int , i );
tVar ( int , j );
...
tPrintf ( "i = %d, i*2 = %d, i*j = %d", i, i*2, i*j );
```

Translation

```
int j;
int i;
...
printf ("i = %d, i*2 = %d, i*j = %d", i, i*2, i*j);
```

Bibliography

[1] Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul H. J. Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In *proceedings of the 2003 Dagstuhl Workshop on Domain-Specific Program Generation*, 2003.

[2] Alastair J. Houghton. Run-time specialisation using a C++ meta-language. Master's thesis, Imperial College London, 2000.

[3] Michael Mellor. Run-time specialisation and optimisation using a C++ meta-language and SUIF. Master's thesis, Imperial College London, 2003.

[4] Kostantinos Spyropoulos. Run-time metaprogramming in C++ and its applications. Master's thesis, Imperial College London, 2004.