THE SCOUT PROGRAMMING LANGUAGE

Language Reference

Los Alamos
NATIONAL LABORATORY
EST. 1943

Version 0.X – December, 2010

CONTENTS

List of Figures

LIST OF TABLES

SOURCE CODE LISTINGS

PREFACE

scout *verb* [intrans.] – *to explore or examine so as to gather information.*

Welcome

The goal of this manual is to provide you with a quick reference to the syntax, semantics and features of the experimental Scout language. We assume that the reader is an experienced programmer with a basic understanding of parallel computing. Although we do our best to keep the language documentation up to date with the feature set supported by the open-source versions of Scout, readers are encouraged to look over the release notes provided with each version for important details.

About Source Code Listings

The source code listings in this manual uses both font changes and *syntax aware* coloring that helps the reader to identify parts of the language. Table 0.1 provides a key to the font and colors using in the listings.

TYPE	COLOR	FONT
KEYWORDS	`keyword`	`keyword`
BUILT-INS	`built-ins`	`built-in`
COMMENTS	*comments*	*comments*
STRINGS	*"string"*	*"string"*

Table 0.1: Source code listing colors and fonts.

Support

If you have questions, or encounter problems, please feel free to send an email to the Scout support team via email: scout-support@lanl.gov.

Open Source Effort

This version of Scout is an open-source software effort established by Los Alamos National Laboratory's Applied Computer Science Group. If you are interested in joining the development team please visit our web site for more details:

`http://www.need-a-scout-url-here.com`.

1 Introduction

Scout is an experimental programming language that combines sequential and parallel general-purpose constructs with data analysis- and visualization-centric features. The language can be seen as both subset and extension to the C programming language, but it has also been influenced by many other languages and is fundamentally a higher-level language than C. In this manual we assume the reader is familiar with parallel programming and the basics of the C programming language. In this chapter we review the features that Scout shares with C. Following chapters will introduce Scout's abstract computational data structures and the parallel, data analysis and visualization specific constructs.

1.1 Scout and the C Programming Language

Scout implements a subset of the C programming language. The most obvious difference between Scout and C is the lack of support for pointers and Scout's explicit parallel computing constructs. In addition, many of the system-level features available to C (via the C standard library) are not supported. As we will show in later chapters, Scout builds upon the fundamental types of C to include higher-level abstractions – such as computational meshes and the associated fields stored on a mesh. From this perspective, Scout is a higher-level language than C.

Like C, Scout provides fundamental control-flow constructs for decision making and looping. In addition, functions and block-structured variable declarations are supported using C's syntax and semantics. Scout extends the fundamental data types available in C to include vector types of two, three, and four components. Scout's syntax for vector types follows closely with those used by the OpenGL Shading Language and NVIDIA's C for CUDA. Section 1.4 below covers these topics in more detail.

1.2 Getting Started

The best way to learn any programming language is to get your hands dirty. Following in the footsteps of C, Listing 1.1 presents a version of the classic *hello world* program. For the most part this code looks identical to C/C++. A key difference is the use of the built-in `print` function vs. a more traditional `printf` call from the standard C library. In addition, note that it is not necessary to include any header files in this program; built-in language-specific functions are automatically declared for you. For details on Scout's available set of built-in functions see Chapter **??**. In addition, Chapter **??** provides the details of writing your own functions.

```
/* A simple hello world example... */
int main() {
  print("hello, world!\n");
  return 0;
}
```

Listing 1.1: Hello world example.

Scout's compiler follows the characteristics of traditional (Unix) command line interfaces. In the following example, the hello world program from Listing **??** is compiled and

1.3 Compiling a Scout Program

Scout's compiler follows the characteristics of traditional (Unix) command line interfaces. In the following example, the hello world program from Listing **??** is compiled and executed from the command prompt:

```
$ scc hello.sc                    compiler produces default executable "a.out"...
$ a.out
hello, world!
$ _
```

See the on-line `scc` man page, or Appendix **??** for more information on the available command line options.

1.4 Data Types

Those familiar with programming in C or C++ should find Scout's syntax and fundamental data types to be very familiar. However, the lack of pointers, the higher-level data types and the supporting language constructs differ in terms of both semantics and the typical low-level nature of C. In this section we quickly review Scout's additional composite data types.

Vectors

As briefly mentioned in Section 1.1, Scout provides support for two-, three-, and four-component vector types. The "base" type for vectors includes signed and unsigned values of each of the fundamental types (`bool`, `char`, `short`, `int`, `long`, `float`, `double`). The syntax for defining a vector uses these types followed by the number of components in the vector.

```
float2  u; // two-component single-precision vector.
int3    v; // three-component integer vector.
double4 w; // four-component double-precision vector.
```

Meshes

Computational science data structures are frequently based on the concept of a mesh. Scout directly follows this philosophy by introducing mesh-centric data types. Scout supports uniform, rectilinear, structured, and unstructured mesh types. In addition, the language provides constructs for defining the values stored on the various locations of the mesh (e.g. cell centers, vertices, edges). Complete details of Scout's mesh data types are covered in Chapter 2.

```
// Two-dimensional uniform mesh with values stored at cell
// centers and cell verticies.
uniform mesh myMesh[512,512] {
  cells:    float temperature;
            float presure;
  vertices: float3 velocity;
};

// Structured mesh defined from input file.
structured mesh myMesh("mesh-def.smd") {
  cells:    float temperature, density;
};
```

```
  uniform mesh myMesh[16] {
    cells: float a, b;
  };

  forall cells c of myMesh { // 'c' is the active cell.
    c.a = c.b = 0.0f;
}
```

Listing 1.2: A `forall` loop construct.

1.5 Parallel Computations

Building upon the mesh data types Scout provides support for parallel computations over the elements/attributes of the mesh (cells, vertices, etc.). The majority of these constructs use an explicit parallel form that is mixed with the main body of the code that is, by definition, executed sequentially. Listing 1.2 shows an example parallel `forall` construct. In this case we are looping over all the cells of the mesh `myMesh` and setting the values of the cell attributes (`a` and `b`) to 0.0. Note that we can use the specified cell placeholder "`c`" to directly access the attributes of the currently active cell using a C-like structure member access notation. If there are no clashes within scope, the use of the of explicit cell deferencing may be dropped within the body of the loop (e.g. `c.a = 0.0;` can be replaced with `a = 0.0f;`).

Additional levels of parallelism can be introduced by nesting parallel constructs. Specifically, the parallel operations over the cells of a mesh can be combined with with a set of operations over the components of each individual cell. Listing 1.4 shows such a nesting.

```
forall cells c of myMesh { // 'c' is the active cell.
  forall vertices v of c { // 'v' is the active vertex.
    c.a += v.a * v.a;
  }
  c.a = sqrt(c.a);
}
```

Listing 1.3: Nested `forall` loop construct over mesh components.

```
forall cells c of myMesh { // 'c' is the active cell.
  forall n => sten(c) { // forall neighbords 'n' of cell 'c'.
    ...
  }
  ...
}
```

Listing 1.4: Nested `forall` loop construct with selection of neighboring cells.

```
// Stream only a subset of cells of myMesh into the loop
// body for processing.
forall cells c of myMesh(where c.a > 0.0) {
  c.a = c.b / c.a;
}
```

Listing 1.5: Streaming `forall` loop construct.

```
// Stream only a subset of cells of myMesh into the loop
// body for processing.
forall cells of myMesh(select(a, b)) {
  a = b / a;
}
```

Listing 1.6: Streaming `forall` loop construct.

1.6 Queries

1.7 Visualization Constructs

Scout provides a set of different language constructs for directly programming visualization operations on various data types. In comparison to other techniques, this approach presents this functionality as first-class language features instead of the more common function-based, application programming interfaces.

```
// Stream only a subset of cells of myMesh into the loop
// body for processing.
expr e = where cells c of myMesh (a < 0.5);

forall cells of myMesh(apply e) {
  a = b / a;
}
```

Listing 1.7: Streaming `forall` loop construct.

```
float maxval = max(myMesh.cells.temperature);
float range  = maxval - min(myMesh.cells.temperature);
renderall cells c of myMesh {
  // 'color' is a built-in for visualization constructs.
  color = hsv(240.0 - (maxval-c.temperature)/range*240.0, 1.0, 1.0);
}
```

Listing 1.8: A simple visualization construct.

2 Higher-Level Data Types

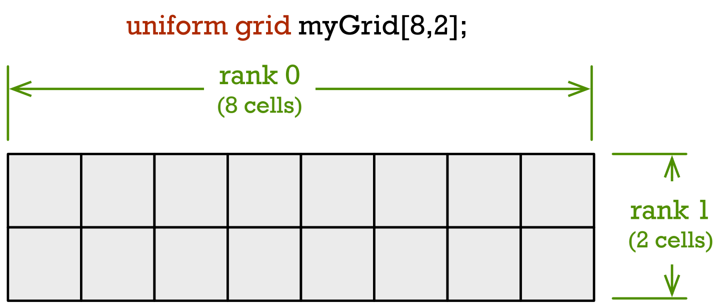

Figure 2.1: Example of a uniform grid definition.

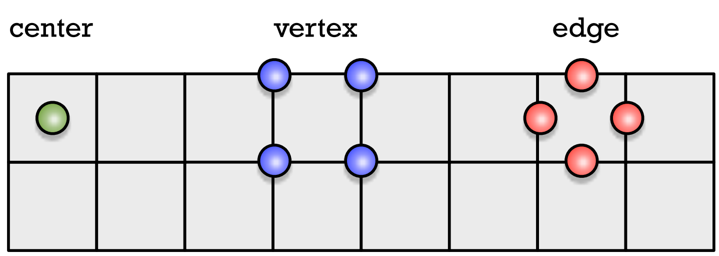

Figure 2.2: Field placement options within a grid.

3 Parallel Constructs

3.1 Data-parallel Operations

`forall` Loops

```
/* Compute the sum of two cell fields. */
int main() {

  uniform grid uGrid[100,100] {
    cells: float a, b, sum;
  }

  forall cells c in uGrid {
    c.sum = c.a + c.b;
  }
  return 0;
}
```

Listing 3.1: A simple `forall` loop.

4 Visualization Constructs

A key feature of the Scout language is the incorporation of visualization and rendering operations directly within the syntax and semantics of the language as first-class constructs.

```
// Example visualization construct for rendering grid cells.
uniform grid myGrid[16] {
  cells: float a, b;
};

renderall cells c of myGrid {
  where (a < 0.5)
    color = rgb(1.0, 0.0, 0.0);
  else
    color = rgb(0.0, 0.0, 1.0);
}
```

Listing 4.1: A `renderall` loop construct.

```
uniform grid myGrid[16] {
  cells: float a, b;
};

  // Compute two isosurfaces (values 0.5 and 1.0) and store them
  // in an unstructured grid.
unstructured grid mySurface = isosurface(myGrid.a, {0.5, 1.0});

renderall faces f of mySurface { // "solid" surface
  ...
}

renderall edges e of mySurface {  // wireframe
  ...
}

renderall cells c of myGrid {  // how do we volume render?
  // If the selected grid is three-dimensional
  // do we just assume this is a volume rendering
  // construct?
  ...
}
```

Listing 4.2: A `renderall` loop for viewing an isosurface.