SMPL - Simple Parallel Language

Language Reference Manual

Andrei Papancea, Ajay S S Reddy Challa, Devashi Tandon
{alp2200, ac3647, dt2450} @columbia.edu

October 19, 2013

Contents

1. Introduction

This document describes Simple Parallel Language (SMPL) which is an approach to parallel programming that is based on a subset of the C programming language. Its major strength is that it has a clearer and simpler syntax for doing parallel programming than programs written using OpenMP or POSIX threads.

With the addition of only four keywords to a subset of the C language, SMPL gives the programmer the ability to parallelize his code with minimal effort.

2. Lexical Conventions

SMPL introduces four parallel keywords. SMPL supports four types of tokens: identifiers, keywords, functions, and separators. Tabs, and newlines are ignored, while multiple spaces are contrived to only one space, which serves as a token separator.

2.1 Identifiers

An identifier consists of a letter followed by one or more alphanumeric and/or underscore characters. The identifiers are case-sensitive.

2.2 Keywords

The following set of keywords are reserved names and cannot be used as identifiers:

- ➔ `int`
- ➔ `float`
- ➔ `boolean`
- ➔ `char`
- ➔ `string`
- ➔ `void`
- ➔ `true`
- ➔ `false`
- ➔ `null`
- ➔ `if`
- ➔ `else`
- ➔ `for`
- ➔ `while`
- ➔ `return`
- ➔ `break`
- ➔ `main`
- ➔ `pfor`
- ➔ `spawn`
- ➔ `lock`
- ➔ `barrier`

2.3 Comments

Comments are represented in a block as follows:

```
/* single line comment */

/* multi
   line
   comment */
```

SMPL does not support comment nesting.

3. Types

3.1 Fundamental Types

In SMPL the following fundamental data types are supported:

- ➔ `int`: integer variable
- ➔ `double`: floating-point variable
- ➔ `boolean`: boolean variable (`true`, `false`)
- ➔ `char`: single ASCII character variable
- ➔ `string`: multi ASCII character variable

In addition, SMPL uses `void` to symbolize that a function has not return value.

3.2 Derived Types

In SMPL the following derived data types are supported:

- ➔ **arrays:** sequence of fundamental types (an array must have elements of the same type)
- ➔ **functions:** sequence of code having its own execution context

Refer to section 5.3 and 5.4 on how to use these types.

3.3 Storage Classes

In SMPL the scope of a fundamental type can be either of the following two storage classes:

- ➔ **local:** scope is limited to the function
- ➔ **global:** accessible in any part of the program

The scope is derived from where it is defined. If the variable is defined outside the function the scope is global, whereas if it is defined within a function the scope is local.

4. Expressions

4.1 Operators

The table below lists the operators supported by SMPL:

Operator	Description
`a[i]`	element `i` of array `a`
`*`	multiplication
`/`	division
`%`	modulus
`+`	addition
`-`	subtraction
`<`	less than comparison
`>`	greater than comparison
`<=`	less than or equal to comparison
`>=`	greater than or equal to comparison
`==`	equality comparison
`!=`	inequality comparison
`!`	boolean NOT operator
`&&`	boolean AND operator
`||`	boolean OR operator
`=`	assignment
`,`	argument separator
`;`	statement separator

4.2 Precedence

The table below illustrates the precedence for all the operators supported by SMPL:

Precedence	Expressions/Operators
highest	`f(arg,arg,...)` `a[i]`
	`!b`
	`n * o` `n / o` `i % j`
	`n + o` `n - o`
	`n < o` `n > o` `n <= o` `n >= o`
	`r == r` `r != r`
	`b && c`
	`b || c`
	`l = r`
lowest	`r1 , r2`

The table above illustrates how expressions are evaluated. Therefore these rules define implicit grouping, so the expression `1+2*3` is equivalent to `1+(2*3)`. If a different grouping is desired, such as `(1+2)*3`, then it needs to be explicitly specified using a pair of parentheses.

5. Syntax

5.1 Statements

A statement can have any of the following formats:

- ➔ expression;
- ➔ { statement-list }
- ➔ `if` (expression) statement `else` statement
- ➔ `while` (expression) statement
- ➔ `for` (expression ; expression ; expression) statement
- ➔ `pfor` (literal ; expression ; expression ; expression) statement
- ➔ `lock` statement
- ➔ `spawn` statement
- ➔ `barrier`;
- ➔ `break`;
- ➔ `return` expression;

5.2 Variable Declaration

The syntax for variable declaration in SMPL has the following format:

```
type identifier; /* the identifier will be initialized to a default value */
type identifier = value; /* the identifier will be initialized to the specified value */
```

The default value for different data types is as follows:

Data Type	**Default Value**
`int`	`0`
`double`	`0.0`
`boolean`	`false`
`char`	`0` (null character)
`string`	`""` (empty string)

5.3 Array Declaration

The syntax for array declaration in SMPL has the following format:

```
type identifier[size];
type identifier[size] = {v1,v2,...,vsize};
```

The indexing of the array starts at 0 (zero).

5.4 Function Declaration

The syntax for function declaration in SMPL has the following format:

```
return-type identifier(arg-type arg1, arg-type arg2, ...){
     declaration-list
     statement-list
}
```

The `return-type` of a function can be any of the fundamental types, including `void`. The `arg-type` can be any of the fundamental types, excluding `void`.

5.5 Parallel Constructs

5.5.1 spawn

The `spawn` statement creates a thread for the given statement. Its syntax looks as follows:

```
spawn function call;
```

5.5.2 barrier

The `barrier` statement prevents execution of code following it until all the threads `spawn`ed prior to it finish executing. Its syntax looks as follows:

```
barrier;
```

Here is a code example of when you would need `barrier`:

```
spawn thread1();
spawn thread2();

print("Done!");
```

Note that in the example above, `thread1` and/or `thread2` could finish after the printing is done, which is probably an undesired result. Subsequently, using a `barrier` above the `print` statement, would prevent the program execution to reach the latter before all the `spawn`ed threads have completed. The fix would look like this:

```
spawn thread1();
spawn thread2();

barrier;

print("Done!");
```

5.5.3 pfor

The `pfor` statement defines a `for` loop that splits up the work in its body into multiple threads. Its syntax has the following format:

```
pfor(k; expression; expression; expression; expression)
     statement
```

In the example above, `k` represents the number of threads that `pfor` will break its body into. For instance, if `k=4` then the work of the statements in the body of `pfor` will be broken into 4 separate threads. The `pfor` construct uses an implicit `barrier` in order to ensure that all the threads have finished before proceeding to the next statement.

Note that you should not `spawn` other threads inside `pfor` since that would entail `spawn`ing threads inside other threads and create unexpected concurrency problems.

5.5.4 lock

The `lock` statement prevents other threads from accessing or modifying the contents of the statement that it precedes until the latter's computation finishes. Its syntax looks as follows:

`lock statement`

Once the computation finishes, the lock will be automatically removed.

Here is a code example of when you would need a `lock`:

```
int sum=0;

pfor(4;int i=0;i<1000000;i++){
     sum += i;
}

print(sum);
```

Note that in the example above, each of the four threads that pfor creates will attempt to modify sum at the same time, creating concurrency problems and therefore outputting a different result every time the program runs. Subsequently, using a `lock` around the `sum += i` statement, would prevent a thread from modifying the variable while another is already trying to change it. The fix would look like this:

```
int sum=0;

pfor(4;int i=0;i<1000000;i++){
     lock {
          sum += i;
     }
}

print(sum);
```

6. Miscellaneous

We will implement the following language features if time permits:

1. nested comments
2. static variables
3. type casting
4. `continue` statement
5. comma-separated variable declaration (ex. `int a, b, c;`)

7. References

We have modeled our Language Reference Manual after the following two documents:

1. http://www.cs.columbia.edu/~sedwards/classes/2013/w4115-fall/clrm.pdf
2. http://www.cs.columbia.edu/~sedwards/classes/2013/w4115-summer2/lrms/CLIP.pdf