C0

1 Introduction

C0 is grammatically similar to the C language and will be immediately familiar to C, C++ and Java programmers. It is a procedure-oriented language with linguistic support for massive parallelism on a modern compute cluster.

1.1 Simple example

Here we have a parallel version of vector addition it in C0.

```
#define A_SIZE 1000
standalone long a[A_SIZE];
standalone long b[A_SIZE];

void add_runner(long start, long *p, long *q, long len) {
        long i;

        for (i = start; i < start + len; i = i + 1) {
                p[i] = p[i] + q[i];
        }
        commit;
}

void start() {
        long i;

        for (i = 0; i < A_SIZE; i = i + 1) {
                a[i] = 0;
                b[i] = 1;
        }

        for (i = 0; i < A_SIZE; i = i + 100) {
                runner add_runner(i, &a[0], &b[0], 100)
                        using a[i,,i+100], b[i,,i+100];
        }
        commit;
}

void main() {
        runner start()
                using a[0,,A_SIZE], b[0,,A_SIZE];
        commit;
}
```

The above program adds two vectors of length 10000 with 100 runners, each runner adds up 100 elements. A runner is a separate execution of code which is similar to threads.

1.2 Program structure

The four key concepts in C0 are programs, types, variables and functions. A program consists of one or more source files. Each source file defines some types or functions. The program must have a function named **`main`** with no parameter or return value. The program starts from the **`main`** function.

1.3 Keywords

The following keywords are to be supported: **`default`** ,**`static`** ,**`sizeof`** ,**`register`** ,**`short`** ,**`auto`** ,**`struct`** ,**`bool`** ,**`int`** ,**`switch`** ,**`true`** ,**`const`** ,**`float`** ,**`case`** ,**`enum`** ,**`false`** ,**`extern`** ,**`volatile`**

Table 1: C0 key words

The supported key words currently.

abort	break	goto	if
continue	else	long	commit
double	commitd	return	unsigned
char	signed	runner	void
while	for	do	watching
in	abortd	standalone	

2 Types

There are several types in C0: simple types, struct types, union types, function types, void type, pointer types, array types, and array segments.

2.1 Simple types

Table 2 shows the simple types supported (Or would be supported) in C0.

Table 2: Simple types in C0

Note: The key words in yellow are not supported currently.

category	bits	type	range/precision
boolean	32	bool	true or false
signed integral	8	char	-128...127
	16	-	-32,768...32,767
	32	int	-2,147,483,648...2,147,483,647
	64	long	-9,223,372,036,854,775,808...9,223,372,036,854,775,807
unsigned integral	8	unsigned char	0...255
	16	-	0...65,535
	32	unsigned int	0...4,294,967,295
	64	unsigned long	0...18,446,744,073,709,551,615
floating point	32	float	1.5 * 10 - 45 to 3.4 * 1038, 7 - digit precision
	64	double	5.0 * 10 - 324 to 1.7 * 10308, 15 - digit precision

2.2 Struct/Union types (not supported yet)

Structure types are user defined types which contains other types (including other structure types). The struct keyword is used to define a structure type. Each element of a structure is called field. Each field in a structure has its own storage space.

```
struct Foo {
    int a;
    int *b;
};

struct {
    int (*func)(int, int);
    Foo foo;
} complex_var;
```

The union types are similar to structure types. But the field in union shares the common storage space, so at most one field contains a meaningful value at any given time.

2.3 Function types

In the program, you cannot directly define variables of function types. But you can define functions who has a function type, or define a function pointer to a specified function type. A function type describes the function prototype, including the types of parameters and the type of return value.

2.4 Void type

Void type is a special type which means "no type", it can only be used for the return type of function, which means the function does not return any value, or used for defining a pointer which can points to any kind of values.

2.5 Pointer types

A variable of pointer type stores the address of the underlying type. We can access the value stored in the memory location which the pointer points to. This operation is called dereferencing a pointer. However, a pointer whose underlying type is void type cannot be dereferenced.

2.6 Array types

An array is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array. We use array[index] to access the elements of an array. The indices of the elements of an array range from 0 to Length - 1.

2.7 Array segment

An array segment is logically same as an array (or a pointer). However, it restricts the access of elements to a specified range. The array segment is represented as array[start,,end], the start is inclusive and end is exclusive.

2.8 Standalone

standalone is a special keyword of C0. It is for global variables which are in the Shared Region (SR). When a global variable is standalone, it always monopolizes one or many memory pages. The main purpose of using standalone is to reduce commit conflict among different global variables, since the basic unit of memory space management is a page.

```
standalone long a;
standalone long b;
```

For the above example, if runner A modify a and runner B modify b, there will be no commit conflict.

3 Expressions and Statements

3.1 Expressions

Expressions are constructed from operands and operators. The operators of an expression indicate which operations to apply to the operands. Examples of operators include +, -, *, /. Examples of operands include literals, fields, local variables, and expressions.

When an expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated. For example, the expression x + y * z is evaluated as x + (y * z) because the * operator has higher precedence than the + operator.

Table 3 summarizes C0 operators, listing the operator categories in order of precedence from highest to lowest. Operators in the same category have equal precedence.

3.2 Statements

The actions of a program are expressed using statements. A block permits multiple statements to be written in contexts where a single statement is allowed. A block consists of a list of statements written between the delimiters and . Declaration statements are used to declare local variables and constants. Expression statements are used to evaluate expressions. Expressions that can be used as statements include method invocations, assignments using = and the compound assignment operators, and increment and decrement operations using the ++ (Not supported yet) and – (Not supported yet) operators. Selection statements are used to select one of a number of possible statements for execution based on the value of some expressions. In this group are the if and switch statements. Iteration statements are used to repeatedly execute an embedded statement. In this group are the while, do, and for statements. Jump statements are used to transfer control. In this group are the break, continue, goto, and return statements.

4 Task and depending task

Defining a task is just the same as defining a function. Actually any function satisfying the necessary con-

Table 3: Operators in C0

Note: The key words in yellow are not supported currently.

Category	Expression	Description
Primary	x.m	Field access
	x(...)	Method invocation
	x[...]	Array or array segment access
	x++	Post-increment
	x−	Post-decrement
	x->y	Pointer
Unary	*x	Dereference
	&x	Referencing the address
	+x	Identity
	-x	Negation
	!x	Logical negation
	x	Bitwise negation
	++x	Pre-increment
	−x	Pre-decrement
	(T)x	Explicitly convert x to type T
Multiplicative	x * y	Multiplication
	x / y	Division
	x % y	Remainder
Additive	x + y	Addition
	x - y	Subtraction
Shift	x « y	Shift left
	x » y	Shift right
Relational	x < y	Less than
	x > y	Greater than
	x <= y	Less than or equal
	x >= y	Greater than or equal
Equality	x == y	Equal
	x != y	Not equal
Bitwise AND	x & y	Integer bitwise AND
Bitwise XOR	x ^ y	Integer bitwise XOR
Bitwise OR	x | y	Integer bitwise OR
Logical AND	x && y	Boolean logical AND
Logical OR	x || y	Boolean logical OR
Conditional	x ? y : z	Evaluates y if x is true, z if x is false
Assignment	x = y	Assignment
	x op= y	Compound assignment; supported operators are *= /= %= += -= «= »= &= ^= |=

straints (will be mentioned later) can be started as a task. A same function can either be directly invoked or be started as a new task.

The function that can become a task must have the prototype with the following constraints

- It has no return type (with return type void)
- The parameters can only be either 1) simple types, or 3) array segments, or 3) structure types whose fields meet the constraints of 1) or 3).

The above constraints ensure that the input parameters to a new task will not reference external memory locations not in the range of the parameters. The use of array segments constraints the use of pointers so the runtime can create the snapshots efficiently.

4.1 Creating task instances

The syntax of creating a task is the same as invoking a function, plus the keyword ***runner***. Note that the task will only start to execute after current task exits.

Example (quick sort):

```
#define A_SIZE 100

long a[A_SIZE];

long partition(long *v, long length) {
        // implementation is omitted
        return 0;
}

void qsort(long *v, long start, long length) {
        long ipivot;

        if (length < 2)
                commit;

        ipivot = partition(&v[start], length);

        runner qsort(v, start, ipivot - start)
                using v[start,,ipivot];
        runner qsort(v, ipivot + 1, length - ipivot - 1)
                using v[ipivot + 1 ,, start + length - 1];

        commit;
}

void start() {
        long i;
        long rand;

        rand = 3141592621;

        for (i = 0; i < A_SIZE; i = i + 1) {
                // pseudo rand generator
                rand = rand * rand;
                a[i] = rand;
        }

        runner qsort(&a[0], 0, A_SIZE)
                using a[0 ,, A_SIZE];
        commit;
}
```

```
void main() {
        runner start()
                using a[0 ,, A_SIZE];
        commit;
}
```

4.2 Depending tasks

Depending tasks are tasks with additional startup conditions. Specifically, it will start after the parent task commits successfully and the specified memory location has been modified since the creation of the depending task.

Defining a depending task is similar to define a normal task. To create a depending task, we also use task keywords, with additional parameters to specify the memory location to watch. The depending task will get executed if the content of the memory has changed. The parameter can either be the pointer to a simple type or structure type, or an array segment.

```
long a;
long b[10];

void func() {
        a = 2;
        b[3] = 5;
        commit;
}

void watcher_func() {
        a = 0;
        b[3] = 0;
        commitd;
}

void main() {
        runner func()
                using a, b[0,, 10];
        runner watcher_func()
                watching a, b[0,, 10];
        commit;
}
```

Difference between "commit" and "commitd", "abort" and "abortd". "commitd" and "abortd" are for depending tasks only. Normal tasks only use "commit" and "abort".

For depending tasks:

1. When you create a depending task, you create a depending task type which can be considered as a "task template".
2. The depending task type watches the commits to the watched memory ranges.
3. When a depending task type is activated, a depending task instance, which can be considered as a normal task, is created. That is, the depending task type keeps watching the memory range changes after it is activated.
4. If there is a depending task instance in the task queue waiting to be scheduled, a commit to the depending task's depending ranges will not activate the depending task and create an instance any more.
5. The "commit" and "abort" have effect on the depending task instance only.
6. The "commitd" and "abortd" will commit/abort and delete the depending task type upon success.

This way, it is ensured that after each commits to the watched ranges by a depending task, at least one depending task instance is created. Hence, no changes are missed by the depending task before it deletes itself by "commitd" or "abortd", and many changes may be "aggregatively" checked by a depending task instance for avoiding creating too many depending task instances that will abort and improving the efficiency.

4.3 Creating tasks in another space

The addressable memory in i0 contains many “spaces”. Each space has a space specifier and the offset ranges for all spaces are the same. By default, the task statement creates tasks in the same space as the parent task. The space can be specified by the in clause of the task statement.

For example, to create a qsort task in space SPACE1:

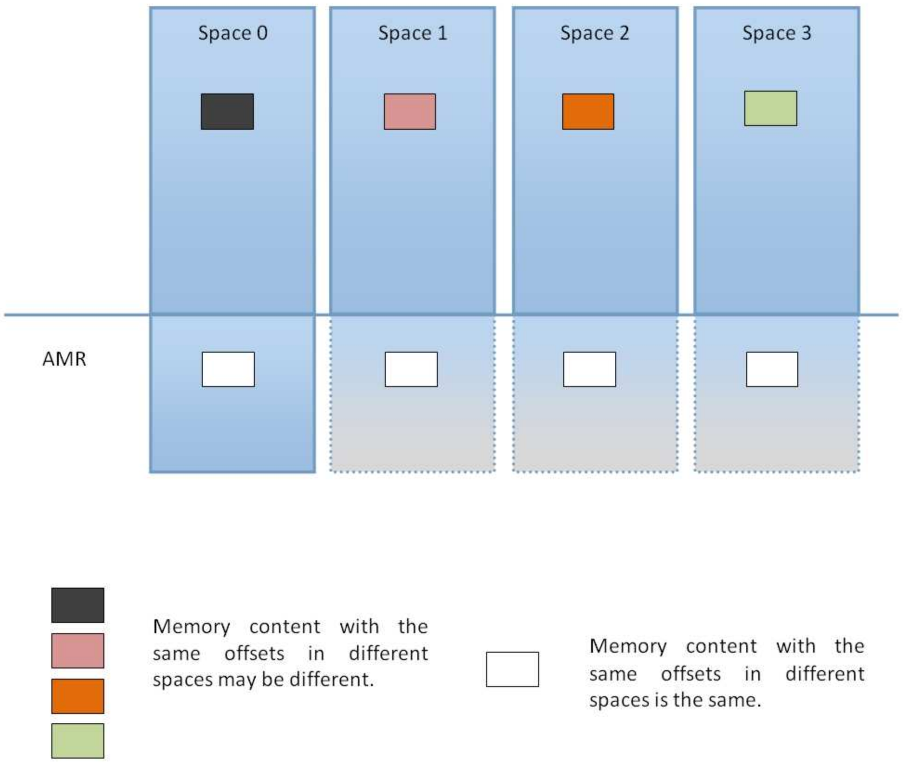
```
#include "libi0/stddef.h"

long space;

// To create a qsort task in SPACE1

space = SPACE1;
runner qsort(0, 100)
    using v[0,,100]
in space
;
```

The AMR as discussed in Section 5.1 is used for data communication across spaces since a task can only run inside of one space. The changes to memory ranges in AMR will activates all depending tasks in any space that watch the ranges.

5 Runtime Environment

5.1 Memory Layout

Each task runs in one space of the many spaces supported. The aliased memory region (AMR) is accessible from all spaces and can be used to share data across spaces. Fig. 1 illustrates the spaces.



Space 0
Space 1
Space 2
Space 3
AMR
Memory content with the same offsets in different spaces may be different.
Memory content with the same offsets in different spaces is the same.

Figure 1: Spaces

The memory layout from the view inside of one space is as in Table 5.1.

Table 4: Memory layout

Higher address		L0 memory type (indicated in colors): Heap Stack
	[Task j] Additional Heap range of (array segments)	
		* The locations and sizes of additional heap ranges may be overlapped with other stack/heap ranges, depending on the location of array segments passed in the startup parameters.
	[Task 0] Stack (grows to lower address)	
	...	
	[Task i] Stack (grows to lower address)	
	...	
	[Task i*] Additional Heap range of array segments	
	[Shared] Runtime Heap (grows to higher address)	
	.bss (Global variables without initial value)	
	.data (Global variables with initial values)	
	[Shared] .rodata (Read-only data)	
	[Shared] .text (Code)	
	L0 Internal range	
Lower address		

5.2 Program loading

A c0 program will be compiled into a binary in the ELF format. At the start of the L0, the program loader will perform the following operations.

- Load the ELF binary from the disk
- Parse the ELF headers
- For each section of ELF. (We only use the following sections: ".text", ".data", ".rodata", ".bss".)
  - Allocate the virtual memory range
  - Copy/map the data block into the memory; note that the length of data block might be less than the memory range. Fill the rest of the space with zeros.
- Create a snapshot, includes:
  - Heap: all the memory ranges of the ELF sections in memory
  - Initial dynamic heap with fixed size (e.g.1GB?, but we don't need to allocate memory pages now)
  - Fixed size (e.g. 64KB?) stack
- Start a new task with the entry point and the created snapshot.

Update History

- Originally written by Xiang Gao.
- May. 8, 2013. Add space for the task statement. - zma
- Feb. 5, 2014. Revise this document. - Weiwei Jia
- Feb. 18, 2014. Revise this document. - Zhiqiang ma, Weiwei Jia
- Feb. 24, 2014. Corrected several typos and improved writings of several places. - Zhiqiang Ma
- Feb. 26, 2014. Add standalone-related stuff for keywords and a section to introduce it. - zma
- Feb. 28, 2014. Revisions from lingu; add a figure illustrating the spaces. - zma
- Feb. 28, 2014. Mark that ‘double‘ are supported; ‘int‘ is buggy yet. - zma
- Mar. 20, 2014. Mark bitwise AND, OR and XOR (&, |, ^) that is supported. Fix several strange names. - zma
- Apr. 3, 2014. Translate C0 doc into Latex version. - Weiwei Jia.
- Apr. 6, 2014. Resize the figure 1, 2 and table 4 and make them closer to corresponding descriptions.
- Apr. 29, 2014. Add the “differene between commit/abort and commitd/abortd”. - zma
- May. 16, 2014. Refined writing. - zma
