Design and compilation of a C-like front end language for GPRM

Ross Meikleham

27th March 2015

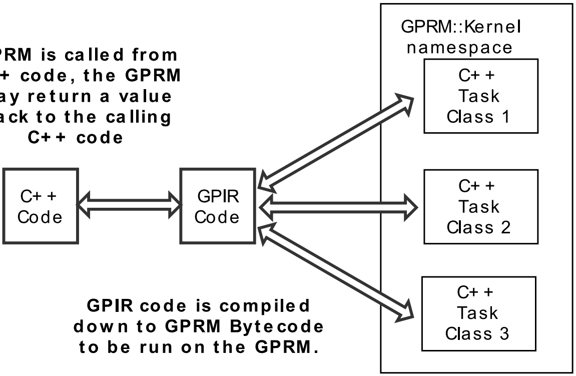
Introduction

- In the past most code written with serial computation in mind
- Rate of increase of processor speeds declining, focus on multiple processor cores
- Would be useful to be able to reuse existing serial code to take advantage of multiple cores without rewriting it.

GPRM

- The Glasgow Parallel Reduction Machine(GPRM) is a framework for parallel programming using a task based approach.
- Seperate code into task code and communication code.
- Task code is written as C++ classes, can use already written serial C++ code
- Communication code written in a language called GPIR

GPRM Structure



GPRM is called from C++ code, the GPRM may return a value back to the calling C++ code

GPIR code is compiled down to GPRM Bytecode to be run on the GPRM.

GPIR code manages the composition of Kernel Class methods and threads to run the methods on.

Task Kernels must reside within the GPRM::Kernel Namespace, these are written in C++.

Kernel methods may return values back to the GPRM

GPIR

- Glasgow Parallel Intermediate Representation language
- Purely functional S-expression like language like lisp or scheme.
- Evaluated in parallel by default with optional sequential evaluation
- Controls the composition of tasks.

Problems with GPIR

- Language isn't "consistent" with the rest of the framework.
- Requires programmers to learn multiple languages which are vastly different.
- Still requires programmers to manually enter threads for each individual task to run on

GPC

- GPC (Glasgow Parallel C). Near subset of C++ with restrictions.
- Layer ontop of GPIR, can take advantage of parallel evaluation semantics of GPIR.
- Introduces two new keywords “seq” and “par” to denote sequential or parallel evaluation of code blocks.

GPC Features

- Purely functional, no multiple assignments
- Support for C++ objects, basic arithmetic operations, functions, types, restricted for-loops.
- Serial semantics.
- Jumping to labels is an expensive operation in the GPIR so inlining as much as possible to the point that execution path can be determined at compile time is useful.
- However kernel calls for C++ can return values, for a turing complete language this is essentially the halting problem.

Kernel type system

- Resulting return values from calls to C++ Kernel Object methods placed in the `GPRM::Kernel` namespace.
- Kernel types can be passed into kernel calls, but not used as “GPRM” types, this prevents them being used for conditional statements, caught by type checker.
- Essentially keeps GPC purely functional while the “side effect” code is restricted to the kernels.

Benefits of being a C++ subset

- Being a subset of C++, we can define `seq` and `par` as no-ops, compiler ignores preprocessor directives.
- Can be compiled using a C++ compiler, due to serial semantics behaviour should be the same as parallel version.
- Can use already available C++ tools to test.

GPC Implementation

- Language compiler is implemented in Haskell
- Uses parsec library to parse/lex source
- Type and Scope checking then run over the AST
- Then interprets the AST to generate GPIR code

Interpreter

- Runs over AST
- Assigns threads to tasks
- Performs branch elimination on if statements.
- Constant propogation and partial evaluation of expressions
- Function inlining for speed and for loop unrolling
- Generates GPIR AST, can be “pretty printed” to GPIR source

MergeSort Example

```
GPRM::Kernel::Mergesort MS;

int overall_size = 8;
int cutoff = 1;

void Sort(int size, int c, int index) {
    int new_size = size/2;
    if (size <= cutoff) {
        MS.serial_ms(c%index, size);
    } else {
        MS.merge_two(c%index, size,
            Sort(new_size, 2*c, 2*index),
            Sort(new_size, 2*c + 1, 2*index)
        );
    }
}

void GPRM_MergeSort() {
    seq {
        MS.array(overall_size);
        Sort(overall_size, 1, 1);
    }
}
```

MergeSort Example

```
;GPRM_MergeSort.yml

(seq
    '(MS.array[0] '8)
    '(MS.merge_two[0] '0 '8
        (MS.merge_two[1] '0 '4
            (MS.merge_two[2] '0 '2
                (MS.serial_ms[3] '0 '1)
                (MS.serial_ms[4] '1 '1))
            (MS.merge_two[5] '1 '2
                (MS.serial_ms[6] '2 '1)
                (MS.serial_ms[7] '3 '1)))
        (MS.merge_two[0] '1 '4
            (MS.merge_two[1] '2 '2
                (MS.serial_ms[2] '4 '1)
                (MS.serial_ms[3] '5 '1))
            (MS.merge_two[4] '3 '2
                (MS.serial_ms[5] '6 '1)
                (MS.serial_ms[6] '7 '1)))))
```

Summary

- Designed a purely functional language which is familiar to programmers which describes the composition of serial C++ tasks to be run in parallel
- Built the compiler for this language.
- Questions?