Programming with less effort in C++

Measuring the programming effort with metrics

Sylvain Jubertie - Adam Ferreira Da Costa

University of Orléans

CPPCon 2015

Foreword

Who am I

- Sylvain Jubertie,
- Assistant professor, University of Orléans, France
- Researcher at the LIFO in the parallel computing team
- Teacher at the Technological Institute and CS department

Foreword: previous work

"Writing parallel/optimized codes requires a lot of effort..."

- Cache optimisation, SoA, AoS, tiling, blocking, vectorisation (auto, intrinsics), threading (pthreads, OpenMP, TBB, Cilk. . .), CUDA, OpenCL, MPI, . . .
- Evaluating the programming effort to write parallel codes.
- **Software metrics**.
- J. Legaux, S. Jubertie, F. Loulergue : *Experiments in Parallel Matrix Multiplication on Multi-Core Systems*. ICA3PP 2012, september 2012, Fukuoka, Japan

1 Software metrics & programming effort

2 Code samples

3 Automatic computation of metrics with Clang

4 Conclusion/Future work

1. Software metrics & programming effort
2. Code samples
3. Automatic computation of metrics with Clang
4. Conclusion/Future work

Software metrics

Goals

Measurements on the code to determine:

- Instruction path length
- Code coverage
- Comment density
- **Programming effort**
- ...

Programming effort metrics

Possible metrics

- Source Lines of Code
- **Halstead metrics**
- ...

Limitations

- Empirical assumptions
- Language dependency

Halstead metrics

Fundamentals

- Measurement theory
- Language independent

Code is composed of:

- operators: $+, -, \times, \ldots$
- operands: a, b, $toto$, $\ldots$

Halstead metrics: measures

Basic measures

- N_1: total number of operators
- N_2: total number of operands
- n_1: number of distinct operators
- n_2: number of distinct operands

Measures

- Program length: $N = N_1 + N_2$
- Program vocabulary: $n = n_1 + n_2$
- Volume: $V = N \times log_2 n$
- Difficulty: $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$
- Effort: $E = D \times V$

Halstead metrics: C++

Halstead metrics are independant of the considered language.
Need a definition of operators and operands for C++.

Operators

C++ operators (but not limited to), `return`, `break`, `using`, . . .

Operands

variables, constants, types, . . .

1 Software metrics & programming effort

2 Code samples

3 Automatic computation of metrics with Clang

4 Conclusion/Future work

Code samples

1. Code optimization/Parallel libraries
2. Standard C++

Code samples: matrix multiplication

```
for( int i ...) {
  for( int j ...) {
    for( int k ...) {
      m0[ i * w + j ] += m1[ i* w + k ] * m2[ k * w + j ];
    }
  }
}
```

Code samples: matrix multiplication

```
for( int i ...) {
  for( int k ...) {
    for( int j ...) {
      m0[ i * w + j ] += m1[ i* w + k ] * m2[ k * w + j ];
    }
  }
}
```

Code samples: matrix multiplication

```
for( int i ...) {
  for( int k ...) {
    float32x4_t r1 = vld1q_dup_f32( &m1[ i * w + k ] );
    for( int j ...) {
      float32x4_t r3 = vld1q_f32( &m3[ i * w + j ] );
      float32x4_t r2 = vld1q_f32( &m2[ k * w + j ] );
      r2 = vmulq_f32( r1, r2 );
      r3 = vaddq_f32( r3, r2 );
      vst1q_f32( &m3[ i * w + j ], r3 );
    }
  }
}
```

Code samples: matrix multiplication

```
#pragma omp parallel for ...
for( int i ...) {
  for( int k ...) {
    for( int j ...) {
      m0[ i * w + j ] += m1[ i* w + k ] * m2[ k * w + j ];
    }
  }
}
```

Code samples: matrix multiplication

Algorithm	Effort
sequential	9k
seq + tmp	12k
SSE	738k
blocked seq	196k
blocked seq + tmp	201k
blocked SSE	1976k

Ratio effort/performance

- Vectorisation: 10/100x more effort - speedup 2/10x.
- OpenMP: small extra effort - speedup dep. on algo., platform.
- Blocking: 10/20x more effort - speedup dep. on complexity.
- Use an existing library when possible...

Code samples: hello world

```
int main()
{
  std::cout << "Hello" << std::endl;
  std::cout << "CPPCon" << std::endl;
  std::cout << "2015!" << std::endl;
  return 0;
}
```

Measures

- Operators: `()`, `{}`, `<<`, `::`, `return`
- Operands: `int`, `main`, `std`, `cout`, `"Hello"`, ...

Code samples: hello world

```
using namespace std;

int main()
{
  cout << "Hello" << endl;
  cout << "CPPCon" << endl;
  cout << "2015!" << endl;
  return 0;
}
```

Effort

Similar difficulty, less volume, less effort!

Code samples: using C++11 features

```
std::vector< int >::const_iterator cit = v.cbegin();

auto cit = v.cbegin(); // C++11
```

auto

Less difficulty, less volume, less effort!

Code samples: using C++11 features

```
struct Functor {
  void operator()(...) {
    ...
  }
};

[](...) { ... };
```

Lambdas

Less difficulty, less volume, less effort!

Code samples: using C++11 features

```
vector< int > v;
v.push_back( 1 );
v.push_back( 2 );
...

vector< int > v{ 1, ... };
```

Initializer lists

No comment!

Code samples: using C++11 features

```
class Toto {
...
  Toto( Toto && toto ) {...} // move constructor
  Toto & operator=( Toto && toto ) {...} // operator=
...
};
```

Move semantics

More effort, but maybe more performance...

Code samples: using C++11 features

Other interesting features:

- ranged based loops
- variadic templates
- threads (compared to pthreads)
- futures
- ...

1 Software metrics & programming effort

2 Code samples

3 Automatic computation of metrics with Clang

4 Conclusion/Future work

Metrics calculator: libTooling

- Generation, traversal, transformation of the AST
- Clang Tools: `clang-check`, `clang-format`, `clang-modernize`
- Possible tools: static analysis, automatic unused variables removal, comments generator, **C++ Core Guidelines checker!**, ...

Metrics calculator: dumping the AST

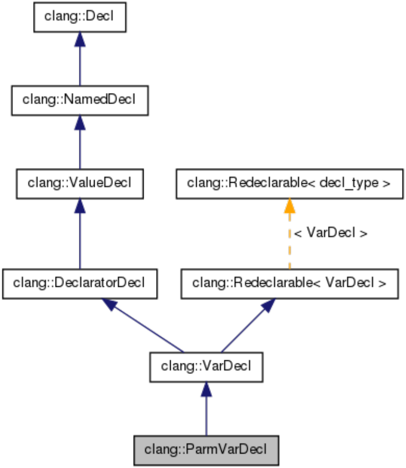
`clang-check` tool

```
clang-check -ast-dump test.cpp --
```

AST nodes

- Declarations
- Statements
- Expressions

Metrics calculator: libTooling concepts



Metrics calculator: hello world

```
class MyASTVisitor :
    public clang::RecursiveASTVisitor< MyASTVisitor >
{
  ...
  bool VisitDecl( clang::Decl * p_decl )
  {
    p_decl->dump(); // print info
    return true;    // continue...
  }
  ...
};
```

Metrics calculator

`metrics-calculator`

- 1st version developed by Adam Ferreira.
- 2nd version still experimental
- `https://github.com/sjubertie/metrics-calculator`

1. Software metrics & programming effort
2. Code samples
3. Automatic computation of metrics with Clang
4. Conclusion/Future work

Conclusion

- An empirical approach... but verified in practice.
- Measuring effort ratio betwen codes seems relevant.
- Do not take into account the user level (learning effort).
- Most of the C++11 new fonctionnalities help reducing the programming effort.
- Introduction of new keywords/annotations/... → more programming/learning effort.
- Need to remove/simplify things? (typedef → using, . . .)

Conclusion: Feedback from students

120 students from the technological institute (2nd year) and the CS departement (3rd-5th year), University of Orléans.

- adoption of `auto` is fast.
- `decltype` is more difficult to use.
- lambdas are easier to write/use than functors
- `using` is preferred to `typedef`: the syntax = makes it more intuitive...
- ranged based loops rather than iterator based loops but sometimes need of a counter.
- mixing auto, decltype, template parameters, constexpr, ... is complex: also compiler error messages do not help...
- move semantics...

Conclusion: Clang libtooling

- Powerfull tool to manipulate the AST.
- Sometimes complex to use (learning effort!): type hierarchy, traversal, excluding header files.
- Need to combine with the lexer to verify token presence:

```
class B : A {...}; // implicit private
```

- Use `clang-modernize` tool to convert existing code to C++11.

Future work

- Measuring the code produced by templates (metaprogramming), ratio between code written and code generated by the compiler.
- Clang libtooling offers different ASTs with or without instantiated templates.
- Taking the lenght of variable/type identifiers into account.
- Evaluate my students accordingly to the effort required to write their codes...
- **C++ Core Guidelines checker!**
- ...

Questions?

sylvain.jubertie@gmail.com