Computational Optimization
ISE 407

Lecture 4

Dr. Ted Ralphs

Reading for this Lecture

- See Web site

Programming Languages

- The purpose of all programming languages is to allow a human to give instructions to a computer.
- There are two basic steps.
  - The human describes what is to be done in the programming language (*source code*).
  - Through (possibly multiple) translation steps (compilation), this original source code is translated into "equivalent" machine code.
- "Equivalent" here means that the machine code *should* run according to the specifications of the original program.
- Note, however, that getting a program to execute as you expect is much harder than you might think!
- Naturally, each translation step may introduce errors due both to incorrect and/or ambiguous specifications.
- Much of the error is introduced by the imprecision of machine arithmetic.
- Furthermore, the translation is not unique and some translations will be "better" than others.

What Does "Better" Mean?

- Assuming that the specifications are correct and unambiguous, so that the program runs correctly, "better" usually means *using few resources*.
- What resources matter depends on the goal of your computation.
  - Minimize computation *time*.
  - Minimize *memory* usage.
  - Minimize *energy consumption*.
- In some cases, it is not possible to compute the answer to the problem we are trying to solve precisely, so "better" could also mean "more precise."
- Making code "better" is a combination of possible improvements to the source code and possible improvements to the translation procedure.
- We will take the compilation process as a given and concern ourselves with how to write better source code.

What Does "Better" Source Code Mean?

- With respect to source code, there are many more dimensions of "better."
  - How easy it is for a human to read (and understand) the code.
  - How easy it is to maintain the code.
  - How easy it is for a compiler to "understand" the code, i.e., generate efficient machine code from it.
- These may be conflicting objectives $\Rightarrow$ it may be necessary to sacrifice some efficiency in order to make code readable.
- Other tradeoffs are evident in the choice of programming language or development environment.

The Evolution of Programming Languages

- Over time, languages have evolved further and further away from the hardware and towards human thought.
  - First generation: Machine language
  - Second generation: Assembly language; can be written and read by a human and requires compilation, but still linked to specific hardware platform.
  - Third generation: General-purpose, multi-platform, but still relatively "close to the metal" (Fortran, C, Python)
  - Fourth generation: Language with a specific purpose (Matlab, AMPL)
  - Fifth generation?: No programming, just state the problem and let the computer write the code.
- In general, later generation languages are also "higher level," but this is not entirely accurate.
- There are a wide variety of third-generation languages, some of which are quite high level (Python).

The Right Tool for the Job

- Programming languages vary on many axes.
  - Low level versus high level
  - General purpose vs specific purpose
  - Strongly/statically typed versus weakly typed
  - Paradigm A versus paradigm B
  - Memory management
  - AOT versus JIT versus interpreted
- There are tradeoffs along all of these axes.
- Selecting the right tool for the job can be challenging in itself.

Some Programming Paradigms

- Imperative/Procedural:
  - Most natural and common paradigm.
  - First do this, then do that.
  - Closest to machine language.
- Functional programming:
  - Evaluate an expression and use the resulting value for something.
  - Directly reflects the mathematics of functions.
- Logic Programming:
  - Prove/disprove a statement based on known logical propositions.
  - Most closely reflects the kinds of questions we typically ask in optimization.
- Object-oriented programming:
  - Data and data types are central.
  - Procedures are only defined in the context of a specific data type.
  - Most directly captures the notion of data structures and algorithm in theoretical computer science.
  - Emphasis on maintainability, re-usability, extensibility.

What's the Right Paradigm?

- In reality, the paradigms are not at all distinct.
- It's not really possible to say unequivocally what paradigm a given language follows.
- Solving an optimization problems is generally equivalent to determining the validity of a statement in logic (prove a bound on a function value).
- Alternatively, solving an optimization problems is also equivalent to the evaluation of a certain function (e.g., the value function).
- Mostly, however, we program in either procedural or object-oriented languages.

High Level versus Low Level

- Generally speaking, high level languages try to reflect the structure of human thought (to the extent possible).
- Low level languages reflect how the hardware works.
- It is usually easier for a human to read and write a high level language, but easier to generate efficient machine code from a low-level language.
- Low level languages make it much more apparent how the code will be translated and executed and it is easier to directly affect the end result.
- In high level languages, it is often easy to write inefficient code without realizing it if the implementation of the language is not understood.
- Special-purpose languages can sometimes be both high level and efficient by exploiting the structure inherent in the application.

Typing

- One of the major differences between programming languages is how they handle types.
- Type checking ensures that the program doesn't perform operations that are undefined for a given type.
- More importantly, knowledge of the types of different data elements allows better optimization of the generated machine code.
- Machine language is an untyped language, as it regards all data as just strings of bits.
- Most other languages have some kind of typing
  - Strong/weak
  - Static/dynamic
  - Explicit/implicit
  - Safe/unsafe

Memory Management

- Another big difference with programming languages is the degree to which the user is responsible for managing memory.
- C famously allows users direct access to and responsibility for memory allocation and management.
- This allows the user tremendous flexibility in economizing and improving efficiency, but imposes a huge implementational burden.
- Many high level languages manage memory automatically, but this comes at the cost.

Other Considerations

- Programming efficiency
  - Maintainability
  - Expressiveness
  - Readability
- Simplicity
  - Generality
  - Orthogonality
  - Uniformity
- Extensibility
- Intended purpose
- Tools
  - Debugging
  - Documentation
  - Libraries
  - IDEs

Comparing C, Julia, Python, and Matlab

- To illustrate the differences between languages, we exhibit naive methods for multiplying two matrices and printing the result.
- Note the differences in terseness, the ability to type and the exposure of implementational details.
- Which language would make it easier to improve efficiency?

Comparing Languages: C

```c
#include <stdio.h>
#include <memory.h>
int main(int argc, char **argv){
    int A[2][3] = {{1, 2, 3},
                   {4, 5, 6}};
    int B[3][4] = {{1,  2,  3,  4},
                   {5,  6,  7,  8},
                   {9, 10, 11, 12}};
    int C[2][4], i = 0, j = 0, k = 0;
    memset (C, 0, sizeof(int[2][4]));
    for (i = 0; i < 2; ++i){
        for (j = 0; j < 4; ++j){
            for (k = 0; k < 3; ++k){
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    print_matrix(C);
}
```

- We hard-coded the matrices here so memory allocation is on the stack.
- This simplifies things substantially.

Comparing Languages: C (cont.)

```c
void print_matrix(int **C, int num_rows, int num_cols){
    int i = 0, j = 0;
    printf("Result is");
    for (i = 0; i < num_rows; i++){
        printf("\n");
        for (j = 0; j < num_cols; j++){
            printf("%4d", C[i][j]);
        }
    }
    printf("\n");
    return;
}
```

Comparing Languages: Python

```python
def matmult(A, B):
    B_T = list(zip(*B))
    return [[sum(x * y for (x, y) in zip(r, c)) for c in B_T]
            for r in A]

A = [[ 1,  2,  3],
     [ 4,  5,  6]]

B = [[ 1,  2,  3,  4],
     [ 5,  6,  7,  8],
     [ 9, 10, 11, 12]]

C = matmult(A, B)
```

Comparing Languages: Julia

```
function matmult_untyped(A, B, C)
    fill!(C, 0)
    N = size(A, 1)
    for i = 1:N, j = 1:N, k = 1:N
        C[i, j] += A[i, k] * B[k, j]
    end
    return(C)
end

function matmult(A::Array{T, 2}, B::Array{T, 2},
                         C::Array{T, 2}) where T <: Number
    fill!(C, 0)
    N = size(A, 1)
    for i = 1:N, j = 1:N, k = 1:N
        C[i, j] += A[i, k] * B[k, j]
    end
    return(C)
end

A = rand(100, 100); B = rand(100, 100); C = similar(A)
matmult(A, B, C)
```

Comparing Languages: Matlab

```
1  function matmult()
2      A = [ 1,  2,  3;
3            4,  5,  6];
4      B = [ 1,  2,  3;
5            4,  5,  6;
6            7,  8,  9;
7           10, 11, 12];
8      C = A * B
9      disp(C)
```