Accelerate Your Science: An Introduction to High Performance Computing

Lecture 2: Programming languages as a means for expressing algorithms

Guido Juckeland
Visiting Scholar
Technische Universität Dresden, Germany

July 7, 2014
Solving problems using algorithms

• “In mathematics and computer science, an algorithm is a step-by-step procedure for calculations.” (Wikipedia)

• Most large problems can be broken down into step by step instructions for solving them:
Divide and conquer

- Decompose the problem into smaller easily solvable subtasks

Program BuildWall

Repeat while (height < ok)
  start_new_row
  repeat while (row not full)
    lay_one_brick
  end(while)
end(while)
Components of an algorithm

• Statement
  – Assign value of a function/evaluated expression to a variable
• Conditions
  – Execute the following code block only if the expression is true
  – Can also have else branch
• Loops
  – Iterate until a condition is true/false
  – Can have fixed iteration count
• Function calls
  – Modularize reused parts as functions
Programming languages for expressing algorithms

• Interpreted languages
  – Runtime system reads program and executes it
  – Portable
  – Usually lacking performance
  – Shell scripts, matlab, perl, python, ruby,…

• Compiled languages
  – Program is compiled to machine code
  – Only portable within same machine type
  – Better performance through better adaption
  – Fortran, C, C++,…
Handling data

• Need to store the somewhere to process it (i.e. run algorithms on it)
• Data layout and algorithm need to match
  – Brick wall example:
    • Needs to have sufficient number of bricks in input available
    • Needs to determine end of row correctly
Typical data forms

- **Variable**
  - Can keep a single value of a certain type (e.g. an integer)
  - `int force=0;`

- **Array**
  - Can keep a predefined number of values of the same type (e.g. vector coordinates)
  - `float pos[3];`

- **Structure**
  - Can keep a set of (different variables)
  - `struct {
    float x, y, z;
    double mass;
  } particle_type;`
Data types

- Integers of various sizes (also used for booleans)
  - char, short int, int, long, long long
- Floating point numbers
  - float, double
- Strings as array of characters
  - char[size]
- No type at all
  - void
- Pointers to any type (will hold a memory address to a chunk of data of the type)
  - float *numbers
Other stuff you will find

- Comments (!! Use them !!)
  - // one line comment
  - /* multi line comment, goes all the way to */
- Code Blocks (a.k.a. scopes)
  - {
    // stuff that is grouped together
    // also allows local variables
    ...
  }
- Defines (are substituted by the compiler)
  - #define MAX 1000
Statements

• Evaluate an expression and assign the result to a variable
• Examples
  - \( c = c + 1; \)  // also available as short form \( c++ \);
  - \( a[2] = (b \times c) / 5; \)
  - \( \text{myvar} = \sin(a); \)
  - \( \text{result.mass} = \text{myfunctions}(a, b, c, 15); \)
Conditions

• Execute following instruction/code block only if condition is met
• Examples
  – if ( a == b) b = 0;   // note the two “=”!
  – if ( a % 2 == 0) {
    a /= 2;            // same as a = a / 2;
    do_other_stuff(a);
  }
  – if ( b != a ) b = a;
    else {
      do_lots_of_stuff( a, b );
    }
Loops

- Condition check at beginning
- Repeats loop body as long as condition is true
- Example:
  
  ```
  while (a % 2 == 0)
     a >>= 1; // same as a /= 2;
  ```

- Watch out that something in the loop actually changes a!
- Condition check at end
- Leaves loop when condition is false
- Example:
  ```
  do {
      lots_of_work();
  } while (!quitting_time());
  ```
Counted loops

• When the loop trip count is predetermined
• Example:
  
  ```c
  for ( i=0; i<MAX; i++ ) a[i]=0;
  ```

• Actually the same as:
  
  ```c
  i=0;
  while (i<MAX){
    a[i]=0;
    i++;
  }
  ```
Functions

- Used to provide reusable code blocks
- Example:
  ```c
  void myfunction(int *a, int *b, int *c, int n) {
      int i;
      for(i=0; i<n; i++)
          c[i] = a[i] + b[i];
  }

  double squared(double x) {
      return x*x;
  }
  ```
Doing things in Fortran

• Algorithmic concepts still the same
• Expression in a program different (syntax, semantics)
• http://en.wikipedia.org/wiki/Fortran_language_features

• Main differences:
  – No end of statement mark needed (“;” in C)
  – You can live without this pointer stuff
Walking through an example

- Determine whether an integer $N$ is a prime number
- Requires test if $N$ is divisible by integers between 2 and $N/2$

```c
int is_prime(int num)
{
    if (i <= 1) return 0;
    if (i==2) return 1;
    if (i % 2 == 0) return 0;
    for(int i = 3; i < num / 2; i+= 2)
    {
        if (num % i == 0)
            return 0;
    }
    return 1;
}
```
Goals for parallelization

• Shorten compute time
• Solve larger problems
• More precise solutions

• In sum: Require more computation speed
Good starting point

- Very large problem (spatial)
- Complex problem (multiple components)
Implementation steps

• Identify parallelism in the problem
• Design an algorithm that exploits the parallelism
• Implement the algorithm
• Performance tuning
Challenges

- Identify parallelism
- Determine dependencies
- Performance is a fragile thing
  - Overhead for parallelization
  - Load imbalances
  - Insufficient data reuse
  - Insufficient resources (e.g. memory bandwidth)
Problem distribution

- Distribution into tasks
  - Determine concurrent activity
  - Put enough work into tasks (overhead)

- Data distribution
  - Same task for different parts of the overall data
  - Determine local data per task

- Group and order tasks
- Determine common data
- Consider and judge alternatives
Process

Dependence Analysis

- Group Tasks
- Order Tasks
- Data Sharing

Decomposition
- Task Decomposition
- Data Decomposition

Design Evaluation

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE408, University of Illinois, Urbana-Champaign
Example: Problem distribution

- C = A * B of SIZE \cdot SIZE
  - One task computes one element of C
  - All tasks are independent and can be executed concurrently
Grouping of tasks

- Group of tasks compute a segment of P
  - Every input element will be used multiple times (better usage of memory bandwidth)
  - Threads in the group synchronize their work (overhead)
Selecting a “good” algorithm

- **Things to consider**
  - Iterations/ steps until a solution is found
  - Parallelism in the algorithm
  - Computational intensity (compute operations per memory access)

- **Good strategies**
  - Tiling of data
  - Gather instead of scatter
  - Double buffering
  - Cutoffs