CS205: Computing Foundations for Computational Science

HW1: Parallel Algorithm Design and Implementation

Multithreaded programming…theory and practice

We want this…

Often ends up like this…. !

Image courtesy online….

CS205, Spring 2017
Computing Foundations for Computational Science
M. Manjunathaiah (Manju)

Homework 1
Due Mon, February 27th, 2017

The objective in this homework is to gain insights into scaling characteristics of parallel algorithms. We will explore a number of concepts of parallel algorithm design from a practical perspective by implementing parallel programs and by linking the theory with the practice.

There are three aspects:

1. Analysis of parallel algorithms.
2. Implementing parallel algorithms on state-of-the-art large scale high performance computer, Harvard's Odyssey, to get practical experiences and insights into performance characteristics of parallel algorithms, parallel architectures and optimization techniques.
3. Implementing parallel algorithm to explore scalable algorithm design and strong scaling concepts. You can work in teams (of 3 to 4) for this part 3.

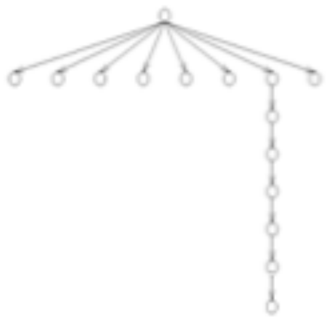
1 Guidelines

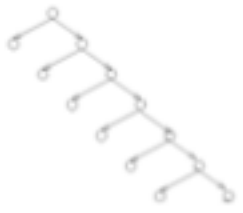
- Cython or C
  - You can implement your parallel programs in either languages using multi-threading libraries on a single node multi-core processor.
  - clone a copy of the https://github.com/harvard-cs205/cython-install-check repository, and make sure `python run tests.py` produces the expected output.
  - See Cython examples in CS205 webpage.
- Odyssey
  - Python on Odyssey
  - you will validate your parallel programs using various metrics on a single multi-core compute node with processor cores in the range $1 \leq p \leq 64$
  - For help regarding cluster usage email: rchelp@fas.harvard.edu
- HW1 submission
  - In two parts:
    - a single PDF file with code listing, test cases and performance plots to be uploaded on Canvas for parts 1 and 2.
    - a single PDF file (for each team) to be uploaded on Canvas for part 3.
- Late submissions: You can use up to two late days for any single homework assignment, with a maximum of six total late days for the semester.

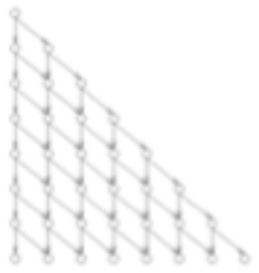
2 Analysis of Parallel Algorithms [20%]

For the computation graphs a), b) and c) shown in figure, determine the following assuming unit time for computation:

1. Maximum degree of concurreny
2. Critical path length
3. The minimum number of processors needed to obtain the maximum speed-up
4. The maximum achievable speed-up if the number of processors is limited to (i) 2 and (ii) 5


a)


b)


c)

3 Scaling: Matrix Vector Multiplication [20%]

The time optimal summation algorithm is not cost-optimal (lecture 5, slide 12-13). In this problem we will investigate this aspect from a practical perspective in two stages.

First,

- Use any cython parallel implementation for summing a list of numbers (or translate the summation algorithm from lecture ntoes).
- Modify the above to implement a cost-optimal summation parallel algorithm.
- Plot speed-up and efficiency graphs for the two versions to validate cost-optimality for problem (data) sizes $n = 2^6, 2^{10}, 2^{20}, 2^{32}$

Using the above summation program as a kernel computation:

- Implement a fast parallel program for Matrix-Vector multiplication.
- Implement a cost-optimal parallel program for Matrix-Vector multiplication.
- Derive the cost-optimality expression for the parallel Matrix-vector algorithm.
- Plot the efficiency graphs for the two versions for problem (vector) sizes $n = 2^6, 2^{10}, 2^{16}$.

Submission:

- A one page write-up on the analysis of cost-optimality of summation and Matrix-vector multiplication algorithms.
- Code listing and performance metric graphs.

4 Strong scaling: Matrix Matrix Multiplication [60%]

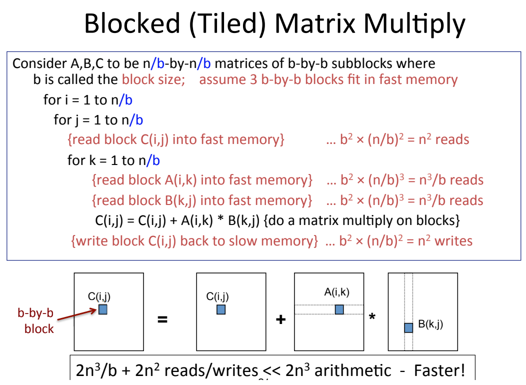
In this problem we will investigate a range of parallel algorithm design principles: from data decomposition, task granularities, locality optimization to more advanced concepts in constructing efficient parallel programs. Additionally we will start building up first intuitions on communication avoiding (or minimizations) formulations that can achieve strong scaling. In particular we will focus on data movement across memory hierarchy in a multi-core parallel system.

Parallel computers can solve a fixed problem faster and can thus achieve strong scaling. The core idea is tiling (or blocking) of the matrices so that the data sits in the fast memory (cache) and the data movement between slow and fast is minimized.

Let M be the size of the "fast" memory (cache) per processor. A tiled matrix multiplication of block size bxb then has the following relationship of data movement to computation

$$2n^3/b + 2n^2(reads/writes) \ll 2n^3(arithmetic)$$

It corresponds to implementing the following pseudo-code of the Matrix-Multiplication[1]:



Blocked (Tiled) Matrix Multiply

```
Consider A,B,C to be n/b-by-n/b matrices of b-by-b subblocks where
    b is called the block size;    assume 3 b-by-b blocks fit in fast memory
      for i = 1 to n/b
        for j = 1 to n/b
          {read block C(i,j) into fast memory}          … b² × (n/b)² = n² reads
          for k = 1 to n/b
              {read block A(i,k) into fast memory}    … b² × (n/b)³ = n³/b reads
              {read block B(k,j) into fast memory}    … b² × (n/b)³ = n³/b reads
              C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
          {write block C(i,j) back to slow memory}  … b² × (n/b)² = n² writes
```

$2n^3/b + 2n^2$ reads/writes $<<$ $2n^3$ arithmetic - Faster!

To implement and optimize the block Matrix Multiplication:

1. First benchmark a naive version (nested three loop matrix multiplication program) against the common reference for double-precision matrix multiplication *dgemm* (double-precision general matrix-matrix multiply) routine in the level-3 BLAS. Generate a performance plot of throughput (GFlop/s) for problem (vector) sizes $n = 2^6, 2^{10}, 2^{16}$ for double precision floating point real numbers.

2. Determine the size of cache (total size of all levels) M on a compute node of Odyssey and make the block size b as large as possible so that $3b^2 \leq M$.

3. Implement the blocking algorithm for square matrices (m=n=k) and benchmark for problem sizes as above.

4. Combine coarse and fine grain parallelisation: Implement a parallel blocking matrix multiplication algorithm for $p = 32\, or\, 64$. Scale this further with any additional optimization (e.g, vectorization) supported by the compiler and perform a benchmarking.

Submission: A write-up covering the following aspects uploaded as a single PDF file (per group)

1. A short one page description of the different design principles that was used to develop successive versions of the parallel program.

2. The optimizations used and the results obtained with reference to performance plots.

[1] source:Demmel

3. Code listing and test cases.

4. Performance plots of throughput (GFlop/s). The plot should indicate the peak throughput achievable on the compute node.

References:

- Block matrix multiplication: Book 1 pages 345 to 352.
- Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication