Lecture 7X: Practices with Principles of Parallel Algorithm Design

Concurrent and Multicore Programming
CSE 436/536

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
Short Review and Today’s Class

• Parallel Algorithms
  1. Tasks and Decomposition
  2. Processes and Mapping
  3. Minimizing Interaction Overheads

• Practice on data decomposition with working examples
  – BLAS and linear algebra
  – AXPY, Matrix vector multiplication, matrix matrix multiplication

• Practice on running examples, and collect and report performance results
  – See examples
Review of Last Class Contents

Decomposing a large problem into multiple smaller one (tasks)

• Recursive Decomposition

• Data Decomposition
Recursive Decomposition: Quicksort

At each level and for each vector

1. Select a pivot
2. Partition set around pivot
3. Recursively sort each subvector

Each vector can be sorted concurrently (i.e., each sorting represents an independent subtask).

```
quicksort(A, lo, hi)
if lo < hi
    p = pivot_partition(A, lo, hi)
    quicksort(A, lo, p-1)
    quicksort(A, p+1, hi)
```
Recursive Decomposition: Min

Finding the minimum in a vector using divide-and-conquer

**Procedure SERIAL_MIN (A, n)**

```plaintext
min = A[0];
for i := 1 to n - 1 do
  if (A[i] < min) min := A[i];
return min;
```

**Procedure RECURSIVE_MIN (A, n)**

```plaintext
if (n = 1) then min := A[0];
else
  lmin := RECURSIVE_MIN (A, n/2);
  rmin := RECURSIVE_MIN (&(A[n/2]), n - n/2);
  if (lmin < rmin) then min := lmin;
  else min := rmin;
return min;
```

Applicable to other associative operations, e.g. sum, AND ...
Output Data Decomposition

- Each element of the output can be computed independently of others
  - simply as a function of the input.
- A natural problem decomposition

**Example:**
*dense matrix-vector multiply*
Output Data Decomposition: Matrix Multiplication

multiplying two $n \times n$ matrices $A$ and $B$ to yield matrix $C$

\[
\begin{pmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{pmatrix}
\cdot
\begin{pmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{pmatrix}
\]

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$
Background:
Dense linear algebra and BLAS
Motifs

The Motifs (formerly “Dwarfs”) from “The Berkeley View” (Asanovic et al.) form key computational patterns
Dense linear algebra

• Software library solving linear system

• BLAS (Basic Linear Algebra Subprogram)
  – Vector, matrix vector, matrix matrix

• Linear Systems: $Ax=b$

• Least Squares: choose $x$ to minimize $||Ax-b||_2$
  – Overdetermined or underdetermined
  – Unconstrained, constrained, weighted

• Eigenvalues and vectors of Symmetric Matrices
  • Standard ($Ax = \lambda x$), Generalized ($Ax=\lambda Bx$)

• Eigenvalues and vectors of Unsymmetric matrices
  • Eigenvalues, Schur form, eigenvectors, invariant subspaces
  • Standard, Generalized

• Singular Values and vectors (SVD)
  • Standard, Generalized

• Different matrix structures
  – Real, complex; Symmetric, Hermitian, positive definite; dense, triangular, banded ...

• Level of detail
  – Simple Driver
  – Expert Drivers with error bounds, extra-precision, other options
  – Lower level routines (“apply certain kind of orthogonal transformation”, matmul...)
BLAS (Basic Linear Algebra Subprogram)

• **BLAS 1**, 1973-1977
  - 15 operations (mostly) on vectors (1-d array)
    • “AXPY” ( \( y = \alpha \cdot x + y \) ), dot product, scale ( \( x = \alpha \cdot x \) )
  - Up to 4 versions of each (S/D/C/Z), 46 routines, 3300 LOC
  - Why BLAS 1? They do \( O(n^1) \) ops on \( O(n^1) \) data
  - AXPY ( \( y = \alpha \cdot x + y \) )
    • 2n flops on 3n read/writes
    • Computational intensity = \( \frac{2n}{3n} = 2/3 \)
BLAS 2

- **BLAS 2**, 1984-1986
  - 25 operations (mostly) on matrix/vector pairs
  - "GEMV": $y = \alpha \cdot A \cdot x + \beta \cdot x$, "GER": $A = A + \alpha \cdot x \cdot y^T$, $x = T^{-1} \cdot x$
  - Up to 4 versions of each (S/D/C/Z), 66 routines, 18K LOC

- **Why BLAS 2?** They do $O(n^2)$ ops on $O(n^2)$ data
  - Computational intensity still just $\sim (2n^2)/(n^2) = 2$

\[
\begin{bmatrix}
\alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\
\alpha_{21} & \alpha_{22} & \cdots & \alpha_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
\alpha_{m1} & \alpha_{m2} & \cdots & \alpha_{mn}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_n
\end{bmatrix}
= 
\begin{bmatrix}
\alpha_{11}x_1 + \alpha_{12}x_2 + \cdots + \alpha_{1n}x_n \\
\alpha_{21}x_1 + \alpha_{22}x_2 + \cdots + \alpha_{2n}x_n \\
\vdots \\
\alpha_{mn}x_1 + \alpha_{m2}x_2 + \cdots + \alpha_{mn}x_n
\end{bmatrix}
\]
**BLAS 3**

- **BLAS 3, 1987-1988**
  - 9 operations (mostly) on matrix/matrix pairs
  - “GEMM”: \( C = \alpha \cdot A \cdot B + \beta \cdot C, \quad C = \alpha \cdot A \cdot AT + \beta \cdot C, \quad B = T^{-1} \cdot B \)
  - Up to 4 versions of each (S/D/C/Z), 30 routines, 10K LOC
  - Why BLAS 3? They do \( O(n^3) \) ops on \( O(n^2) \) data
    - Computational intensity \( (2n^3)/(4n^2) = n/2 \) – big at last!
    - Good for machines with caches, deep mem hierarchy

\[
A[M][K] \times B[K][N] = C[M][N]
\]
Practice:
AXPY, Matrix Vector, and Matrix Multiplication
BLAS 1: AXPY

- $y = \alpha \cdot x + y$
  - $x$ and $y$ are vectors of size $N$
    - In C, $x[N]$, $y[N]$
  - $\alpha$ is scalar

- Decomposition is simple
  - Terms: partition, distribution, the same
  - Evenly divide $N$ by `num_tasks`
    - Handle corner cases, non divisible of $N$ by `num_tasks`

\[ i_{\text{start}} = 0 \quad i_{\text{start}} = 3 \quad i_{\text{start}} = 6 \]
BLAS 2: Matrix Vector Multiplication

- $y = A \cdot x$
  - $A[M][N], x[N], y[N]$
- Row-wise decomposition

\[
\begin{array}{ccc}
  A & x & y \\
  \times & = & \\
  \text{Mt} & \text{i\_start} & \\
\end{array}
\]
BLAS 3: Dense Matrix Multiplication

\[ A[M][K] \times B[k][N] = C[M][N] \]

- Base
- Base_1: column major order of access
- row1D_dist
- column1D_dist
- rowcol2D_dist

- Decomposition is to calculate Mt and Nt
BLAS 3: Dense Matrix Multiplication

- Row-based 1-D

\[ Mt = \frac{N}{\text{num\_tasks}} \]
\[ i\_start = \text{tid} \times Mt; \]
\[ Nt = N \]
\[ j\_start = 0 \]
BLAS 3: Dense Matrix Multiplication

- Column-based 1-D

\[ \begin{align*}
    i_{\text{start}} &= 0 \\
    M_t &= N \\
    j_{\text{start}} &= \text{tid} \times N_t
\end{align*} \]

\[ A \times X = B = C \]

\[ T_0 \quad T_1 \quad T_2 \quad T_3 \]

\[ N_t \]
BLAS 3: Dense Matrix Multiplication

- Row/Column-based 2-D

- If you do nested parallelism
  - export OMP_NESTED=true
Submatrix Multiplication

• Work with any of the three decomposition

```c
/* compute submatrix multiplication, A[start:length] notation
 * A[i_start:Mt][N] x B[N][j_start:Nt] = C[i_start:Mt][j_start:Nt]
 */

void matmul_base_sub(int i_start, int j_start, int Mt, int Nt, int N,
                      REAL A[][N], REAL B[][N], REAL C[][N]) {
    int i, j, k;
    for (i = i_start; i < Mt+i_start; i++) {
        for (j = j_start; j < Nt + j_start; j++) {
            C[i][j] = 0;
            for (k = 0; k < N; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    }
}
Background: C multidimensional array
Vector/Matrix and Array in C

- C has row-major storage for multiple dimensional array
  - A[2,2] is followed by A[2,3]

- 3-dimensional array
  - B[3][100][100]

- Think it as recursive definition
  - A[4][10][32]
Column Major

Fortran is column major

Row-major order  Column-major order
Array Layout: Why We Care?

1. Makes a big difference for access speed
   • For performance, set up code to go in row major order in C
     – Caching: each read from memory will bring other adjacent elements to the cache line
   • (Bad) Example: 4 vs 16 accesses
     – matmul_base_1

```plaintext
for i = 1 to n
    for j = 1 to n
        A[j][i] = value
```
2. Affect decomposition and data movement

- Decomposition may create submatrices that are in non-contiguous memory locations, e.g. A3 and B1
- Submatrices in contiguous memory location of 2-D row major matrix
  - A single-row submatrix, e.g. A2
  - A submatrix formed with adjacent rows with full column length, e.g. A1
Array Layout: Why We Care?

2. Affect decomposition and submatrix
- Row or column wise distribution of 2-D row-major array
- # of data movement to exchange data between T0 and T1
  - Row-wise: one memory copy by each
  - Column-wise: 16 copies each
Array and pointers in C

• In C, an array is a pointer + dimensionality
  – They are literally the same in binary, i.e. pointer to the first element, referenced as base address

• Cast and assignment from array to pointer, int A[M][N]
  • A, &A[0][0], and A[0] have the same value, i.e. the pointer to the first element of the array

• Cast a pointer to an array
  – int *ap; int (*A)[N] = (int(*)[N])ap; A[i][j] ....

• Address calculation for array references
  – Address of A[i][j] = A + (i*N+j)*sizeof (int)