Lecture plan

- Last time: Dense Linear Algebra
- This time: Homework, Sparse Linear Algebra

Spatial binning and hashing

- Simplest version
  - One linked list per bin
  - Can include the link in a particle struct
  - Fine for this project!
- More sophisticated version
  - Hash table keyed by bin index

Partitioning strategies

Can make each processor responsible for

- A region of space
- A set of particles
- A set of interactions

Different tradeoffs between load balance and communication.

Performance

Parallel performance starts with serial performance

- Use flags $\rightarrow$ let the compiler help you!
- Can refactor memory layouts for better locality
- You will need more particles to see good speedups
- Overheads: open/close parallel sections, barriers.

Sparse vs. Dense Linear Algebra

Dense == common structures, no complicated indexing

- General dense (all entries nonzero)
- Banded (zero below/above some diagonal)
- Symmetric/Hermitian
- Standard, robust algorithms (LAPACK)

Sparse == Not stored in dense form!

- Maybe few nonzeros (e.g. compressed sparse row formats)
- May be implicit (e.g. via finite differencing)
- May be "dense", but with compact representation (e.g. via FFT)
- Most algorithms are iterative; wider variety, more subtle

Sparse Matrix-Vector Multiply (SpMV)

- Sparse matrix-(dense)vector multiplication (SpMV) used in:
  - Solving linear systems
  - Eigenvalue problems
  - Optimization algorithms
  - Machine learning, etc.
- Sparse matrix-sparse-vector (SpMSpV)
  - E.g., graph algorithms: breadth-first search, bipartite graph matching, and maximal independent sets
  - Sparse matrix-sparse matrix (SpGEMM)
  - E.g., graph algorithms
  - Common special case: A * AT
- Sparse matrix-dense matrix (SpDM 3)
  - Machine learning

Compressed Sparse Row (CSR) Storage

CSR has:

- Array of the nonzero values (val) of size nnz = number of nonzeros
- Array of the column indices for each value of size nnz
- Array of row start pointers of size n = number of rows

Other common formats (plus blocking)

- Compressed sparse column (CSC)
- Coordinate (COO): row + column index per nonzero (easy to build)

SpMV with Compressed Sparse Row (CSR)

Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j) * x(j)$

- BLAS2 not BLAS3
- No reuse in A
- Maximum reuse is y (full row) as written
- Re-use of x

```
for each row i
    for k=ptr[i] to ptr[i+1]-1 do
        y[i] = y[i] + val[k]*x[ind[k]]
```

Possible optimizations

1. Unroll the k loop: need number of non-zeros per row
2. Hoist y[i]: OK absent aliasing
3. Eliminate ind[i]: need to know non-zero pattern
4. Reuse elements of x: need good non-zero pattern

- Cache: need nonzeros in nearby rows and same-cache-line columns
- Register: need to know where these nonzeros are to save x[i]

Taking advantage of block structure in SpMV

- Bottleneck is time to get matrix from memory
  - Only 2 flops for each nonzero in matrix
  - Fetching at about 1 int (column index) + 1 float (value) for 2 flops
- Don't store each nonzero with index, instead store each nonzero row by column block with 1 column index
  - As r*c grows, storage drops by up to 2x, for all 32-bit quantities
  - Time to fetch matrix from memory decreases
- Change both data structure and algorithm
  - Need to pick r and c
  - Need to change algorithm accordingly
- Depends on the problem data and block size. Square block, r==c
  - Consider best case: dense matrix in sparse format

How do permutations affect algorithms?

A = original matrix, $A^P = A$ with permuted rows, columns

- Naïve approach: permute x, multiply $y = A^P x$, permute y
- Faster way to solve $Ax = b$:
  - Write $A^P = P^T AP$ where P is a permutation matrix
  - Solve $A^P x^P = P^T b$ for x^P , using SpMV with A^P , then let $x = Px^P$
  - Only need to permute vectors twice, not twice per iteration
- Faster way to solve $Ax = \lambda x$:
  - A and A^P have same eigenvalues, no vectors to permute!
  - $A^P x^P = \lambda x^P$ implies $Ax = \lambda x$ where $x = Px^P$

Summary of Other Sequential Performance Optimizations

- Optimizations for SpMV
  - Register blocking (RB): up to 4x over Compressed Sparse Row (CSR)
  - Variable block splitting: 2.1x over CSR, 1.8x over RB
  - Diagonals: 2x over CSR
  - Reordering to create dense structure + splitting: 2x over CSR
  - Symmetry: 2.8x over CSR, 2.6x over RB
  - Cache blocking: 2.8x over CSR
  - Multiple vectors (SpMM): 7x over CSR
  - And combinations...
- Sparse triangular solve
  - Hybrid sparse/dense data structure: 1.8x over CSR

Row parallelism in SpMV

$y = A * x$, where A is a sparse matrix...

- In iterative solvers, y is often used to compute next x
- Row parallelism
  - Random access to x
  - No inter-thread dependences, so no races / locks
- Load balancing: Divide number of nonzeros about evenly, not number of rows
- Compare to column parallelism:
  - Both random access read and write to y
  - 2x bandwidth and need to synchronize
  - But combined row and column gives more potential parallelism

Summary of Multicore Optimizations

- NUMA - Non-Uniform Memory Access: pin submatrices to memories close to cores assigned to them
- Prefetch: values, indices, and/or vectors (use exhaustive search on prefetch distance)
- Matrix Compression: not just register blocking, 32 or 16-bit indices, Block Coordinate format for submatrices
- Cache-blocking: 2D partition of matrix, so needed parts of x,y fit in cache

What about memory traffic?

After maximizing memory bandwidth, the only hope is to minimize memory traffic.

- Compression: exploit
  - register blocking
  - other formats
  - smaller indices
- Use a traffic minimization *heuristic* rather than search
- Benefit is matrix-dependent.
- Register blocking enables efficient software prefetching (one per cache line)

Parallelism in Distributed SpMV

$y = A * x$, where A is a sparse matrix

- Row parallelism (y and A partitioned)
  - Replicate x across processors
  - Or exchange only necessary elements: Are nonzeros clustered, e.g., near diagonal?
- Column parallelism (x and A partitioned)
  - Make temporary `delta_y = [0,...]` on all processors;
  - Update that; and add-reduce across processors
- 2D parallelism for large p and when nonzeros uniform
  - Divide processors into $p1 \times p2$ (e.g., square grid)
  - Hybrid of Row and Column parallelism
  - Bad load balance for clustered nonzeros

Sparse Matrix Multiply

- Sparse $\times$ Dense Matmul? Dense result.
- 2D/2.5D/3D only optimal for dense-dense / sparse-sparse matmul
- 100x Improvement for the right algorithm

Matrix Reordering via Graph Partitioning

- "Ideal" matrix structure for parallelism: block diagonal
  - p (number of processors) blocks, can all be computed locally.
  - If no non-zeros outside these blocks, no communication needed!
- Can we reorder the rows/columns to get close to this? Most nonzeros in diagonal blocks, few outside

Graph partitioning

Given:

- Graph $G = (V, E)$
- Possibly weights $(W_V\ ,\ W_E\ )$
- Possibly coordinates for vertices (e.g. for meshes)

We want to partition G into k pieces such that

- Node weights are balanced across partitions.
- Weight of cut edges is minimized

$k = 2$ is a special case (bisection)

Cost

How many partitionings are there? If n is even,

$$\binom{n}{n/2} = \frac{n!}{((n/2)!)^2} \approx 2^n \sqrt{2/(\pi n)}$$

Finding the optimal one is NP-complete. We need heuristics!

Partitioning with coordinates

- Many partitioning problems from "nice" meshes
  - Planar meshes (maybe with regularity condition)
  - Nice enough: Can partition with edge cuts (Tarjan, Lipton Planar Separation Theorem)
  - Edges link nearby vertices
- Get useful information from vertex density
- Can initially ignore edges (but can use them in later refinement)
- Don't always have natural coordinates
  - Example: the web graph
  - Can sometimes add coordinates (metric embedding)
  - So use edge information for geometry

Breadth-first search

- Pick a start vertex v_0 (Might start from several different vertices)
- Use BFS to label nodes by distance from v_0
- Could use a different order to minimize edge cuts locally (Karypis, Kumar)
- Partition by distance from v_0

Kernighan-Lin Bisection

- Usually converges in a few (2-6) sweeps. Each sweep is $O(N^3)$.
- Can be improved to $O(|E|)$

```
While no vertices marked
    Choose (a, b) with greatest gain
    Update D(v) for all unmarked v as if
    (a, b) were swapped
    Mark a and b (but don't swap)
Find j such that swaps 1,...,j yield maximal gain
Apply swaps 1,...,j
```

Partitioning for sparse matvec (SpMV)

Consider :

- Edge cuts $\neq$ communication volume
- Haven't looked at minimizing maximum communication volume
- Look at communication volume: what about latencies?

Sparsity and partitioning

Matrices to graphs

- $A_{ij} \neq 0$ means there is an edge between i and j
- Ignore self-loops and weights for the moment
- Symmetric matrices correspond to undirected graphs

Want to partition sparse graphs so that

- Subgraphs are same size (load balance)
- Cut size is minimal (minimize communication)

Multilevel Partitioning

If we want to partition $G(N, E)$, but it is too big to do efficiently, what can we do?

1. Replace $G(N, E)$ by a coarse approximation $G_c(N_c, E_c)$, and partition G_c instead
2. Use partition of G_c to get a rough partitioning of G, and then iteratively improve it

What if G_c still too big? Apply same idea recursively!

Multilevel Partitioning - High Level Algorithm

$(N+, N-) =$ Multilevel_Partition(N, E)
... recursive partitioning routine returns $N+$ and $N-$ where $N = N+ \cup N-$
if $|N|$ is small:
Return $(N+, N-)$

1. Partition $G = (N, E)$ directly to get $N = N+ \cup N-$; Return $(N+, N-)$
2. *Coarsen* G to get an approximation $G_c = (N_c, E_c)$
3. $(N_c+, N_c-) =$ Multilevel_Partition(N_c, E_c)
4. *Expand* (N_c+, N_c-) to a partition $(N+, N-)$ of N
5. *Improve* the partition $(N+, N-)$

Coarsen, Expand, Improve

- Coarsen: Use *Maximal Matching* to group nodes together
- Expand: Convert coarse partition into fine partition
- Improve: Improve Eigenvalues

Maximal Matching

- Definition: A *matching* of a graph $G(N, E)$ is a subset E_m of E such that no two edges in E_m share an endpoint
- Definition: A *maximal matching* of a graph $G(N, E)$ is a matching E_m to which no more edges can be added and remain a matching

Coarsening using a maximal matching

Construct a maximal matching E_m of $G(N, E)$
`for all edges` $e = (j, k)$ `in` E_m`:`
`-- collapse matched nodes into a single one`
    `Put node` $n(e)$ `in` N_c
    $W(n(e)) = W(j) + W(k)$ `-- update node/edge weights`
`-- add unmatched nodes`
`for all nodes n in N not incident on an edge in` E_m`:`
    `Put n in` N_c `-- do not change W(n)`
`-- Connect two nodes in` N_c `if nodes inside them are connected in` E
`for all edges` $e = (j, k)$ `in` E_m`:`
    `for each other edge` $e' = (j, r)$ `or` (k, r) `in` E
        `Put edge` $ee = (n(e), n(r))$ `in` E_c `-- W(ee) = W(e')`
If there are multiple edges connecting two nodes in N_c, collapse them, adding edge weights

Nested Dissection

Idea: Think of block tree structures.

- Eliminate block trees from bottom up.
- Can recursively partition at leaves.
- Notice graph partitioning appears again!
- And again we want small separators

Sparse Matrix Conclusions

- Tuning for modern processors is hard
- Sparse matrices: tuning harder
- SpMV: benefits lower due to low Computational Intensity (you need to read the matrix)
- Usual low level tuning (prefetch, etc.) have some benefit
- Compressing the matrix can be a big win
- Reordering (including graph partitioning) improves locality
- After tuning SpMV *should be* memory bandwidth limited
- Optimizing at a high level (across iterations) can improve reuse, but it does affect numerics