HUST-USYD Summer School on Parallel Programming Practice – Lecture 6

Bing Bing Zhou (bing.zhou@sydney.edu.au)

School of Computer Science, University of Sydney

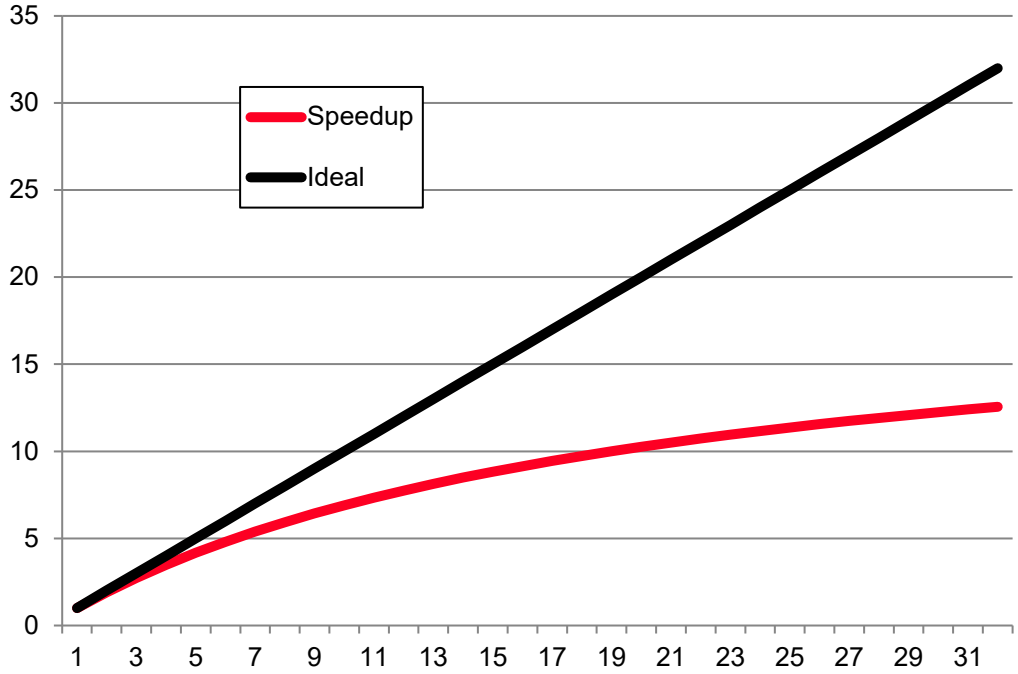
THE UNIVERSITY OF SYDNEY

Outline

- Review of Homework 3
- Task assignment and load balancing
- Graph partitioning
  - Coordinate bisection
  - inertial bisection
- Schedule clause in for directive
- Lab exercise: Matrix multiplication $C = AA^T$
- Homework 4

Ideal and Actual Speedup

- Speedup: $S = \frac{T_s}{T_p}$
- Ideally, we expect the speedup will be increased by a factor of p when using p processing elements
- In practice, the actual speedup for solving a problem is often smaller than the ideal speedup and worse as the number of processors increases



Causes of Inefficiency

- Poor single processor performance
  - Typically in the memory system
- Too much parallelism overhead
  - Synchronization, communication, redundant computation
  - Load imbalance
    - Different amounts of work across processors
      - Computation and communication
- Different speeds of computing resources
  - e.g., CPU + GPU
- To design efficient parallel algorithms, we must try to minimize all unnecessary overheads

Task Assignment

- Recall Amdahl's law: a small amount of load imbalance can significantly bound maximum speedup
- We need good strategies for task assignment to balance workload across the processes/threads
- The strategies can be classified into two categories:
  - Static task assignment
  - Dynamic task assignment (dynamic job scheduling or dynamic load balancing)

Static Task Assignment

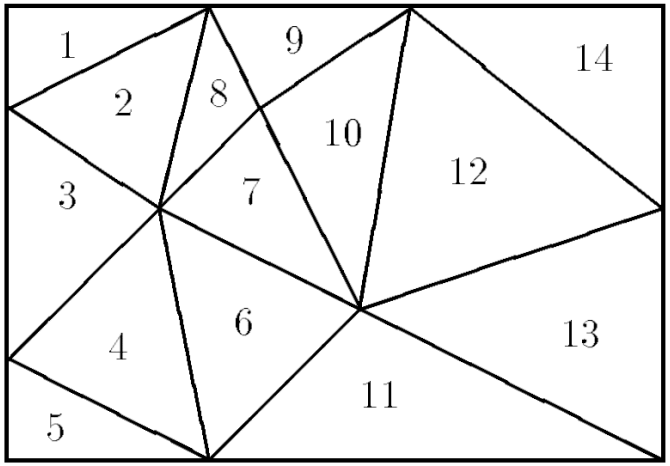
- In static assignment the assignment of tasks to processes/threads is pre-determined
  - Simple and zero runtime overhead
- Static assignment is applicable when the amount of tasks are predictable
- The quality of task assignment is directly related to the quality of task partitioning
  - Some problems, e.g., dense matrix computation, regular meshes, can be easily decomposed into a number of tasks of equal size with regular data dependency pattern – task assignment relatively easy
  - Some other problems, e.g., sparse matrices, unstructured meshes, and graphs, are more complicated, need to consider how to balance workload, minimize communication overhead and redundant computation

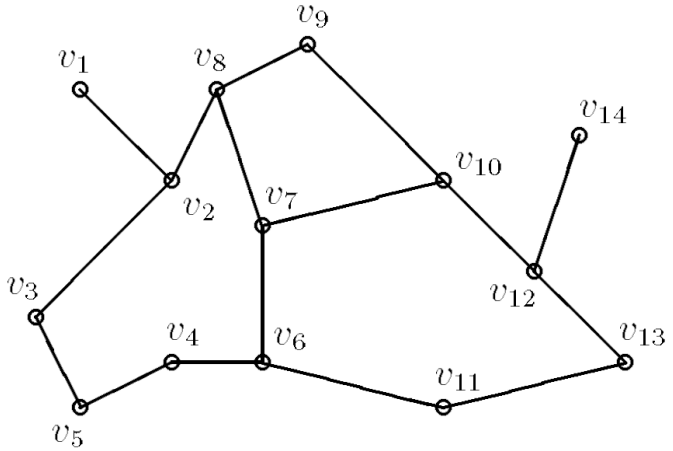
Graph Partitioning

- Graph partitioning is to reduce a graph into a number of smaller graphs
- It has a wide range of applications in practice – many problem can be reformulated as a graph problem
  - e.g., scientific computing, VLSI circuit design, computer graphics, and web graph analysis
- We are interested in large graph computation/analysis using parallel machines
  - thus graph partitioning is an essential pre-processing step

Graph Partitioning

- A lot of the methods for solving partial differential equations (PDEs) are grid-oriented, e.g.,
  - The 2D space is divided into a number of triangles
  - Each triangle, represented by a vertex in the corresponding dependency graph (or mesh), has a state
  - The neighbouring triangles, represented by edges connecting the vertices in the mesh
    - State update of a vertex involves the states of its neighbouring vertices





Definition of Graph Partitioning

- Given a graph $G = (N, E, W_N, W_E)$
  - N: vertices, W_N: vertex weights, E: edges, W_E: edge weights
  - e.g., $N = \{\text{tasks}\}$, $W_N = \{\text{task costs}\}$, Edge (j, k) in E means task j sends $W_E(j, k)$ words to task k
- Choose a partition $N = N_1 \cup N_2 \cup \cdots \cup N_p$ such that
  - The sum of node weights in N_i is about the same
  - The sum of all edge weights of the nodes connecting all different pairs N_i and N_j is minimized
  - e.g., balance workload (have the same amount of node weights), while minimizing communication (minimize the edge cut set)

Graph Bisection

- In practice we often have $N = 2^k$
  - Bisect the graph recursively k times
- Question: how many possible partitions are there for a graph bisection?

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

- Thus graph bisection is in general an NP (nondeterministic polynomial) problem
- We need good heuristics

Partitioning with Geometric Information

- There are many techniques for graph partitioning
- If additional information is available, it may help to solve the problem
  - e.g., the graph is derived from some structure in 2D or 3D space (e.g., in the solution of PDEs), the geometric coordinates can be attached to the vertices and therefore the geometric layout of the graph is known

Coordinate Bisection

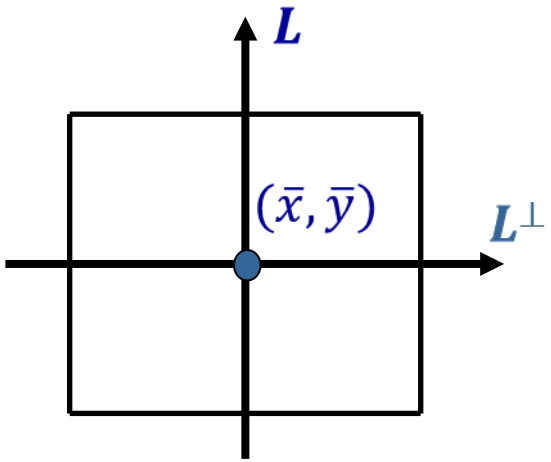
- Coordinate bisection is the simplest coordinate-based partitioning method
- We may simply draw a hyperplane orthogonal, say, to the x-coordinate which divides the vertices into two equal parts
- For recursive bisection we alternately bisect x-, y- and z-coordinates
  - This leads to a better aspect ratio and hopefully a lower edge cut size

Coordinate Bisection

- In a 2D space choose a line L perpendicular to the x-axis with the centre of mass $(\bar{x}, \bar{y})$ on the line, where

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \text{ and } \bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$$

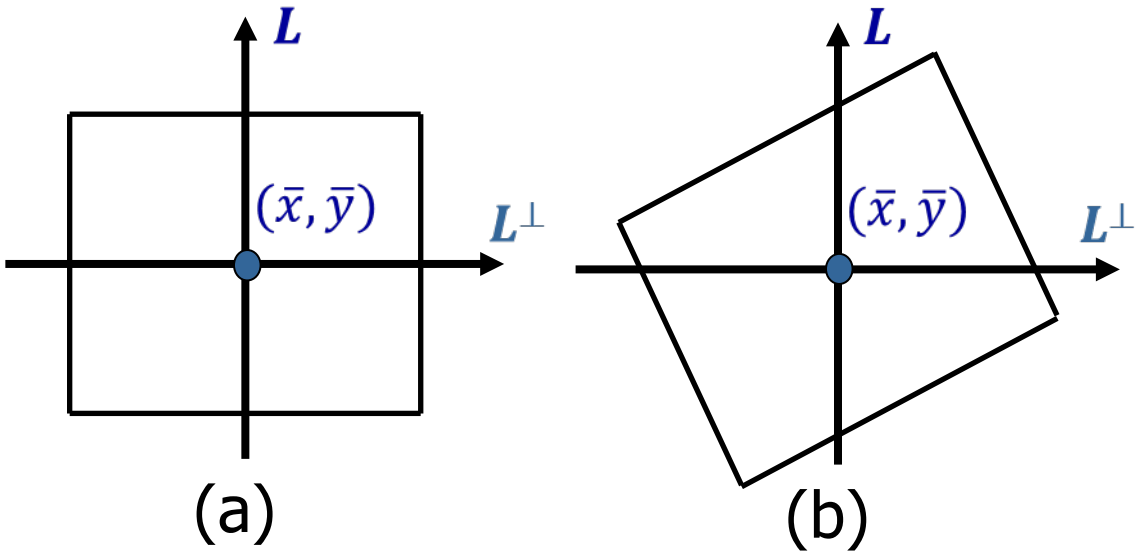
- Then place all the mesh points on the left to N_1 and those on the right to N_2



(a)

Coordinate Bisection

- Disadvantage – coordinate dependent
  - The same structure may partitioned quite differently in different coordinate systems
  - e.g., the partition in (b) is not good as the edge cut set is not minimized in this simple example



Inertial Bisection

- The basic idea of inertial bisection is that we choose a hyperplane in such a way that the sum of squares of vertices distance to that hyperplane is minimized
  - Rotation of coordinates makes the sum of squares of distance from meshes to line L be minimized
- We want to find a line L by solving a minimization problem

$$min \sum_{i=1}^{N} d_i^2$$

where d_i is the distance of mesh point (x_i, y_i) to line L

Rotation of Coordinates

- In the (x, y) system let point P have a polar coordinates (r, α) and then $(r, \alpha - \theta)$ in the (x', y') system
- We have

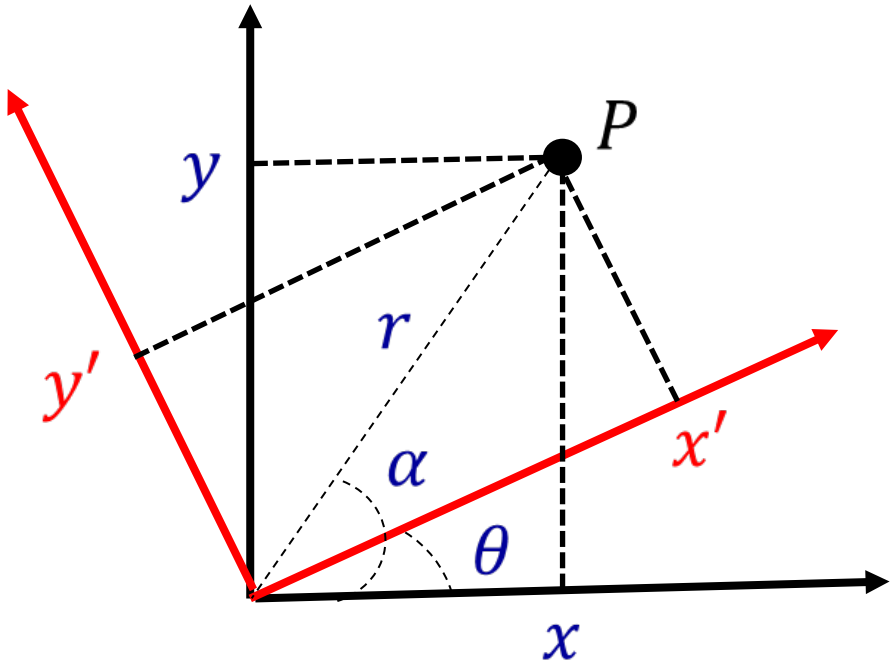
  $x = r\cos\alpha$ and $y = r\sin\alpha$
- and
  - $x' = r\cos(\alpha - \theta)$

    $= r\cos\alpha\cos\theta + r\sin\alpha\sin\theta$

    $= x\cos\theta + y\sin\theta$
  - $y' = r\sin(\alpha - \theta)$

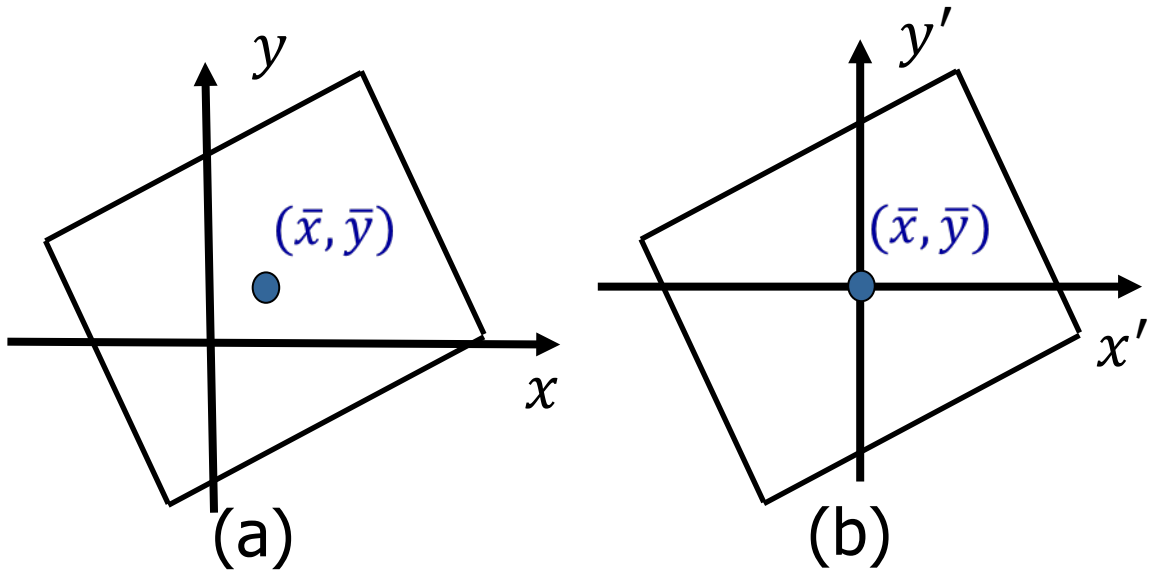
    $= r\sin\alpha\cos\theta - r\cos\alpha\sin\theta$

    $= -x\sin\theta + y\cos\theta$



Inertial Bisection

- First we shift the origin to the centre of mass $(\bar{x}, \bar{y})$
  - $(x_i, y_i) \rightarrow (x_i - \bar{x}, y_i - \bar{y}) = (x_i', y_i')$



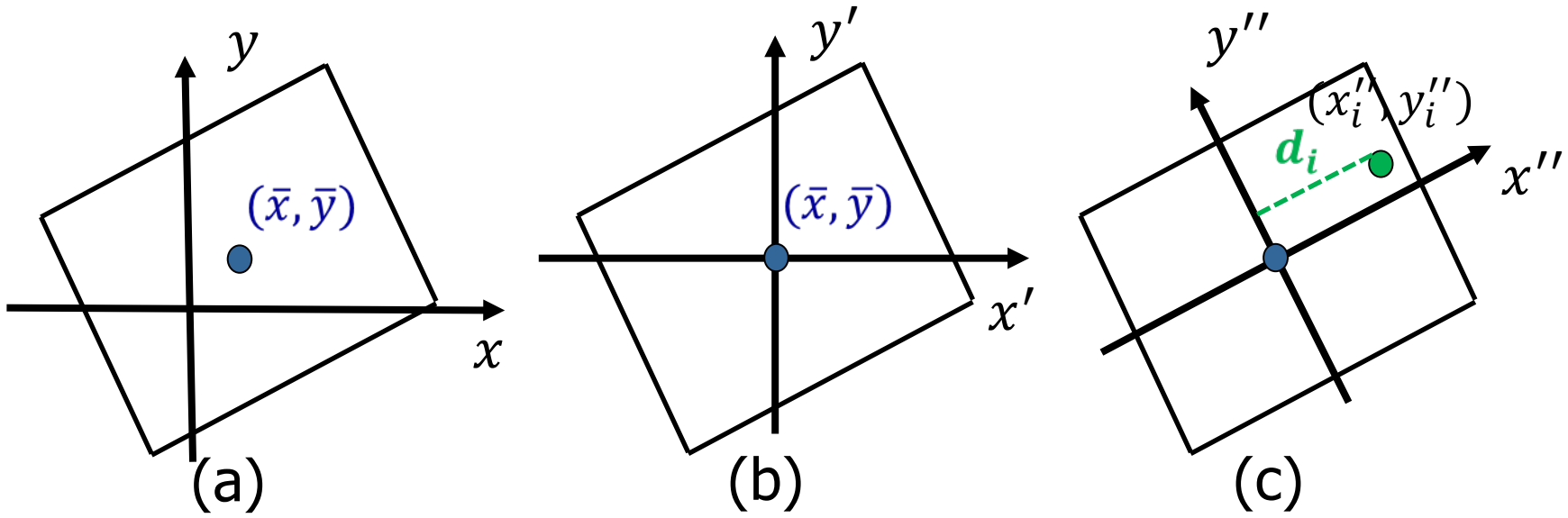
Inertial Bisection

- Next rotate (x', y') coordinates θ degrees:

$$x'' = x'cos\theta + y'sin\theta \text{ and } y'' = -x'sin\theta + y'cos\theta$$

- From (c) we have $d_i = x_i''$
- Letting $cos\theta = v$ and $sin\theta = w$, with simple math we have

$$\sum d_i^2 = v^2 \sum (x_i - \bar{x})^2 + 2vw \sum (x_i - \bar{x})(y_i - \bar{y}) + w^2 \sum (y_i - \bar{y})^2$$

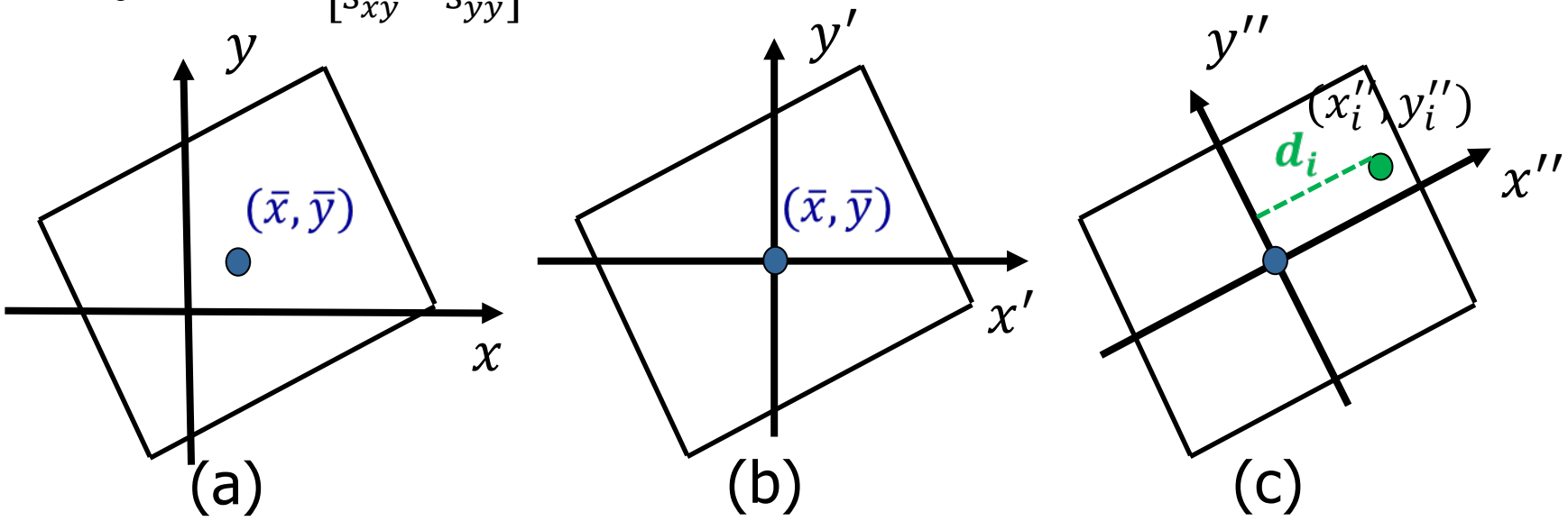


Inertial Bisection

- Let $u^T = [v, w]$, $s_{xx} = \sum(x_i - \bar{x})^2$, $s_{xy} = \sum(x_i - \bar{x})(y_i - \bar{y})$ and $s_{yy} = \sum(y_i - \bar{y})^2$
- We have

$$\sum d_i^2 = v^2 s_{xx} + 2vws_{xy} + w^2 s_{yy} = u^T \begin{bmatrix} s_{xx} & s_{xy} \\ s_{xy} & s_{yy} \end{bmatrix} u$$

- Since we want to minimize $\sum d_i^2$, then choose u as the eigenvector of the smallest eigenvalue of $\begin{bmatrix} s_{xx} & s_{xy} \\ s_{xy} & s_{yy} \end{bmatrix}$



Partitioning without Geometric Information

- The coordinate-based methods, such as inertial bisection, require geometric information about the mesh points
- In many practical problems, however, vertices in the graph don't have geometric information
  - e.g., the model of WWW, vertices are web pages
- We need different algorithms
- There are also a number of methods to deal with the problem

Time vs. Quality

- There often is a trade off between the execution time and the quality of the solution
  - Some algorithms run quite fast but find only a solution of medium quality
  - Some others take a long time but deliver excellent solutions
  - Even others can be tuned between both extremes
- The choice of time vs. quality depends on the intended application
  - For VLSI-design we prefer the quality as even a slightly better solution can save real money
  - For sparse matrix computation we are only interested in the total time
    - The execution time for the graph partitioning has to be less than the time saved by parallel matrix computation
  - Thus there is no single best solution for all situations

Parallel for Directive schedule clause

- The default partitioning for loop iterations is block partitioning
  - The iterations are partitioned into a number of groups and each group contains a set of consecutive iterations of equal size, e.g., (0..7), (8..15), (16..22), (23..29) for N=30 and nthreads=4
- In many cases it is desirable to partition iterations with a different block size, or even to partition iterations cyclically (i.e., cyclic partitioning)
- Solution: use schedule clause

Parallel for Directive schedule clause

- Schedule clause describes how iterations of the loop are divided among the threads in the team
- schedule(static[, chunk])
  - Deal-out blocks of iterations of size "chunk" to each thread
  - Static scheduling is done at compile-time
  - The chunk size is pre-determined and predictable by the programmer – least work at runtime
  - E.g.,
    - schedule(static) – block partitioning
    - schedule(static, 1) – cyclic partitioning
- schedule(dynamic[, chunk])
  - Each thread grabs "chunk" iterations off a queue until all iterations have been handled
  - Dynamic scheduling uses complex scheduling logic at run-time
  - Unpredictable, highly variable work per iteration – most work at runtime

Lab Exercise: Matrix Multiplication $C = AA^T$

- Write an OpenMP program to compute $C = AA^T$, assuming A is of size $n \times m$
- Note the resulting matrix C is a symmetric matrix

$$C^T = (AA^T)^T = AA^T = C$$

- We have $c_{ij} = c_{ji}$
- To save the amount of operations we only need to compute those elements in the upper triangle, i.e., c_{ij} for $i \leq j$
- Write a sequential program and then add OpenMP directives to parallelize the program
  - You need to consider load balancing

Homework 4

- Optimize your OpenMP program for matrix multiplication $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{A}^{\boldsymbol{T}}$ by adding loop unrolling with unrolling factor $= 4$

THE UNIVERSITY OF SYDNEY