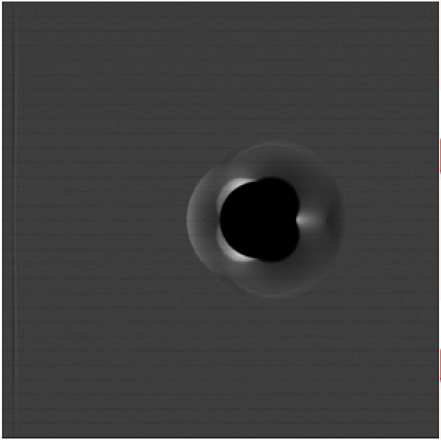
Partitioning Spatially Located Load with Rectangles: Algorithms and Simulations

Erik Saule, Erdeniz Ozgun Bas, Umit V. Catalyurek

Department of Biomedical Informatics, The Ohio State University
{esaule,erdeniz,umit}@bmi.osu.edu

Frejus 2010

A load distribution problem



Load matrix

In parallel computing, the load can be spatially located. The computation should be distributed accordingly.

Applications

- Particles in Cell (stencil).
- Sparse Matrices.
- Direct Volume Rendering.

Metrics

- **Load balance**.
- Communication.
- Stability.

Outline

The Rectangular Partitioning Problem

Definition

Let A be a $n_1 \times n_2$ matrix of non-negative values. The problem is to partition the $[1,1] \times [n_1, n_2]$ rectangle into a set S of m rectangles. The load of rectangle $r = [x,y] \times [x',y']$ is $L(r) = \sum_{x \leq i \leq x', y \leq j \leq y'} A[i][j]$. The problem is to minimize $L_{max} = \max_{r \in S} L(r)$.

Prefix Sum

Algorithms are rarely interested in the value of a particular element but rather interested in the load of a rectangle. The matrix is given as a 2D prefix sum array Pr such as $Pr[i][j] = \sum_{i' \leq i, j' \leq j} A[i'][j']$. By convention $Pr[0][j] = Pr[i][0] = 0$.

We can now compute the load of rectangle $r = [x,y] \times [x',y']$ as $L(r) = Pr[x'][y'] + Pr[x-1][y-1] - Pr[x'][y-1] - Pr[x-1][y']$.

In One Dimension

Heuristic : Direct Cut [MP97]

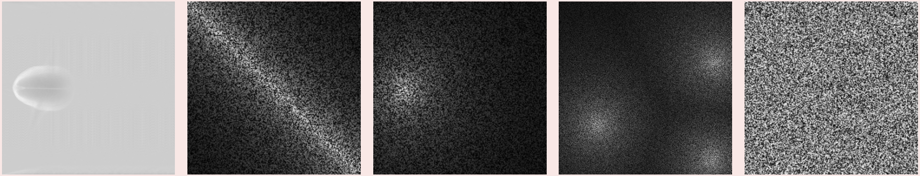
Greedily set the first interval at the first i such as $\sum_{i' \leq i} A[i'] \geq \frac{\sum_{i'} A[i']}{m}$.
Complexity: $O(m \log \frac{n}{m})$. Guarantees : $L_{max}(DC) \leq \frac{\sum_{i'} A[i']}{m} + \max_i A[i]$.

Optimal : Nicol's algorithm [Nic94] (improved by [PA04])

Use $Probe(B)$ which tries to build a solution of value less than B. It loads greedily the processors up with the largest interval of load less than B.
It exploits the property that there exists a solution so that the first interval $[1, i]$ is either the smallest such that $Probe(L([1, i]))$ is true or the largest such that $Probe(L([1, i]))$ is false.
Complexity: $O((m \log \frac{n}{m})^2)$.
Note: it works on more than load matrices, as long as the load of intervals are non-decreasing (by inclusion).

Simulation Setting

Classes (Some inspired by [MS96])



Processors

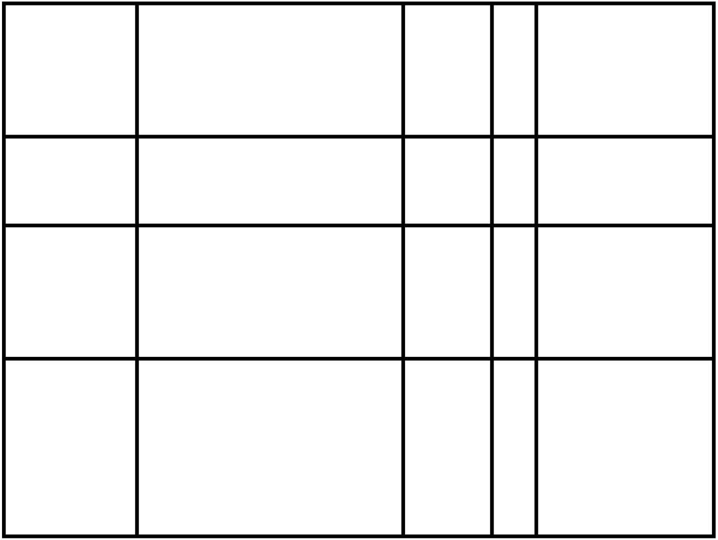
Simulation are perform with different number of processors: most squared numbers up to 10,000.

Metric

Load imbalance is the presented metric : $\frac{L_{max}}{\frac{\sum_{i,j} A[i][j]}{m}} - 1$.

Outline of the Talk

Rectilinear Partitioning



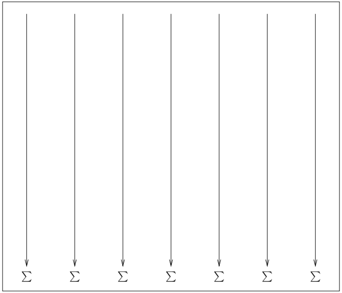
Known results on rectilinear partitioning

- NP Complete [GM96] and there is no $(2-\epsilon)$-approximation algorithm (unless $P = NP$).
- [Nic94]: a $\theta(m)$-approximation algorithm based on iterative refinement. $O(n_1 n_2)$ iterations in $O(Q(P \log \frac{n_1}{P})^2 + P(Q \log \frac{n_2}{Q})^2)$.
- [AHM01](refinement of [Nic94]): a $\theta(m^{1/4})$-approximation algorithm for squared matrices.
- [KMS97]: a 120-approximation algorithm of complexity $O(n_1 n_2)$.
- [GIK02]: 4-approximation algorithm (from rectangle stabbing) of complexity $O(\log(\sum_{i,j} A[i][j]) n_1^{10} n_2^{10})$ (heavy linear programming).
- [MS05]: $(4+\epsilon)$-approximation algorithm that runs in $O((n_1 + n_2 + PQ) P \log(n_1 n_2))$.

Nicol's Rectilinear Algorithm [Nic94]

PxQ rectilinear partitioning

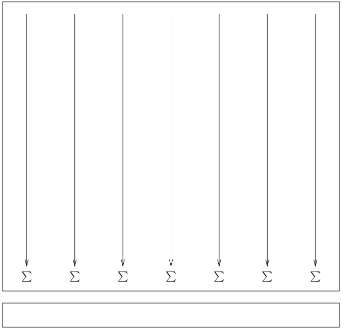
Nicol's Rectilinear Algorithm [Nic94]



PxQ rectilinear partitioning

- Sum the columns to make a 1d instance.

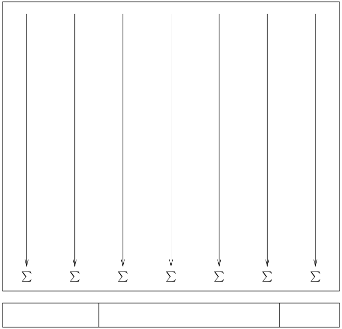
Nicol's Rectilinear Algorithm [Nic94]



PxQ rectilinear partitioning

- Sum the columns to make a 1d instance.

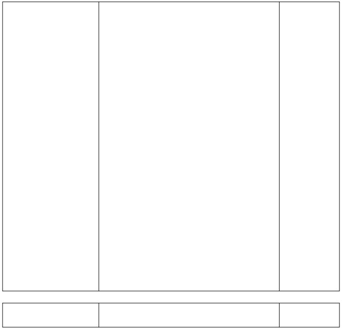
Nicol's Rectilinear Algorithm [Nic94]



PxQ rectilinear partitioning

- Sum the columns to make a 1d instance.
- Partition it in P parts.

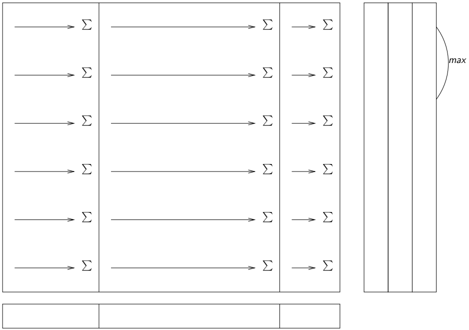
Nicol's Rectilinear Algorithm [Nic94]



PxQ rectilinear partitioning

- Sum the columns to make a 1d instance.
- Partition it in P parts.
- Get a Px1 rectilinear partitioning.

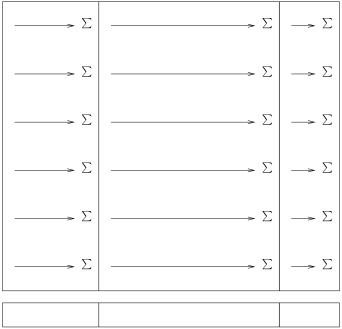
Nicol's Rectilinear Algorithm [Nic94]

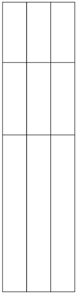


PxQ rectilinear partitioning

- Sum the columns to make a 1d instance.
- Partition it in P parts.
- Get a Px1 rectilinear partitioning.
- Sum the rows in each part.
- Build a 1d instance by taking the maximum for each interval.

Nicol's Rectilinear Algorithm [Nic94]

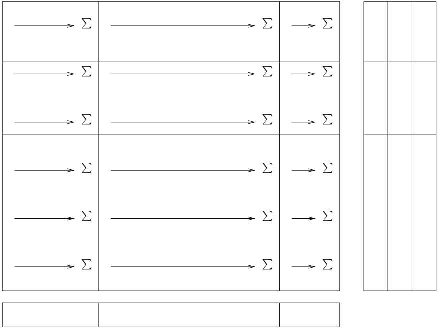




PxQ rectilinear partitioning

- Sum the columns to make a 1d instance.
- Partition it in P parts.
- Get a Px1 rectilinear partitioning.
- Sum the rows in each part.
- Build a 1d instance by taking the maximum for each interval.
- Partition it in Q.

Nicol's Rectilinear Algorithm [Nic94]



PxQ rectilinear partitioning

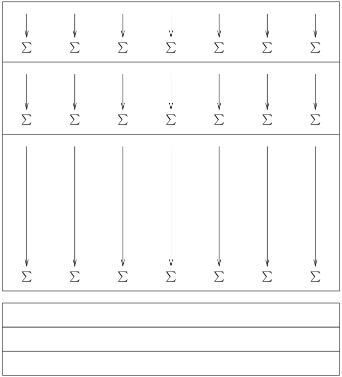
- Sum the columns to make a 1d instance.
- Partition it in P parts.
- Get a Px1 rectilinear partitioning.
- Sum the rows in each part.
- Build a 1d instance by taking the maximum for each interval.
- Partition it in Q.
- Get a PxQ rectilinear partitioning.

Nicol's Rectilinear Algorithm [Nic94]

PxQ rectilinear partitioning

- Sum the columns to make a 1d instance.
- Partition it in P parts.
- Get a Px1 rectilinear partitioning.
- Sum the rows in each part.
- Build a 1d instance by taking the maximum for each interval.
- Partition it in Q.
- Get a PxQ rectilinear partitioning.

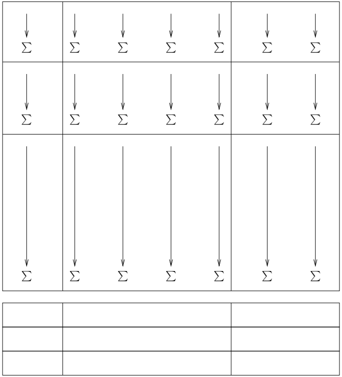
Nicol's Rectilinear Algorithm [Nic94]



PxQ rectilinear partitioning

- Sum the columns to make a 1d instance.
- Partition it in P parts.
- Get a Px1 rectilinear partitioning.
- Sum the rows in each part.
- Build a 1d instance by taking the maximum for each interval.
- Partition it in Q.
- Get a PxQ rectilinear partitioning.
- Ignore the row partition.

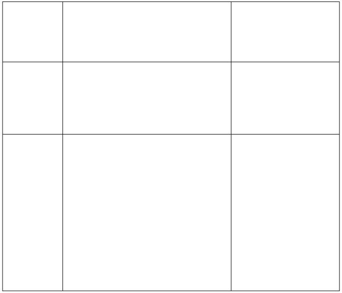
Nicol's Rectilinear Algorithm [Nic94]



PxQ rectilinear partitioning

- Sum the columns to make a 1d instance.
- Partition it in P parts.
- Get a Px1 rectilinear partitioning.
- Sum the rows in each part.
- Build a 1d instance by taking the maximum for each interval.
- Partition it in Q.
- Get a PxQ rectilinear partitioning.
- Ignore the row partition.
- Iterate if improve.

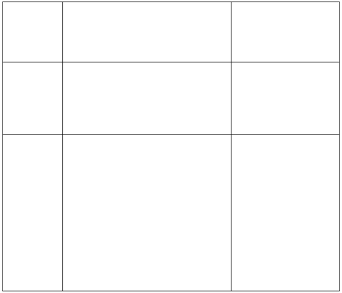
Nicol's Rectilinear Algorithm [Nic94]



PxQ rectilinear partitioning

- Sum the columns to make a 1d instance.
- Partition it in P parts.
- Get a Px1 rectilinear partitioning.
- Sum the rows in each part.
- Build a 1d instance by taking the maximum for each interval.
- Partition it in Q.
- Get a PxQ rectilinear partitioning.
- Ignore the row partition.
- Iterate if improve.

Nicol's Rectilinear Algorithm [Nic94]



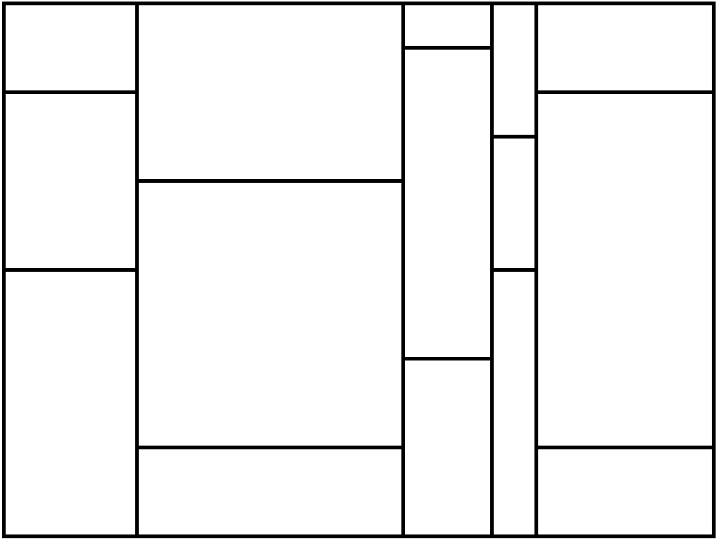
PxQ rectilinear partitioning

Complexity:

- $O(n_1 n_2)$ iterations (around 10 in practice)
- 1 iteration : $O(Q(P \log \frac{n_1}{P})^2 + P(Q \log \frac{n_2}{Q})^2)$.

Outline of the Talk

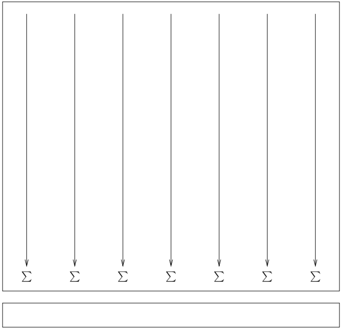
PxQ Jagged Partitioning



PxQ heuristic

PxQ Jagged Partitioning

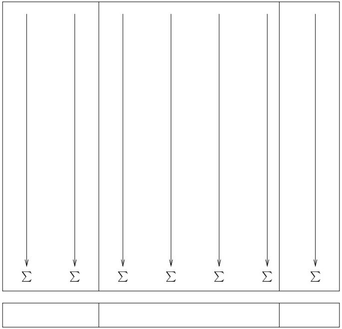
PxQ heuristic



PxQ Jagged Partitioning

- Sum on columns to generate a 1D problem.

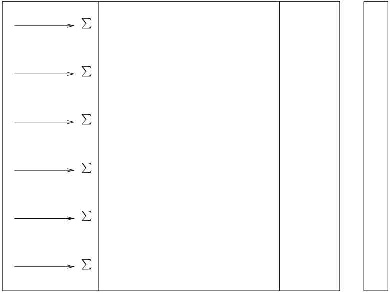
PxQ heuristic



PxQ Jagged Partitioning

- Sum on columns to generate a 1D problem.
- Partition it in P parts.

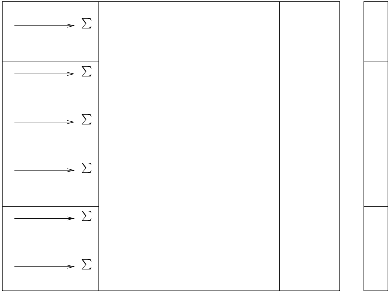
PxQ heuristic



PxQ Jagged Partitioning

- Sum on columns to generate a 1D problem.
- Partition it in P parts.
- For the first stripe, sum on rows.

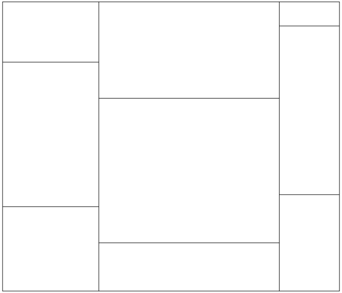
PxQ heuristic



PxQ Jagged Partitioning

- Sum on columns to generate a 1D problem.
- Partition it in P parts.
- For the first stripe, sum on rows.
- Partition it in Q parts.

PxQ heuristic



PxQ Jagged Partitioning

- Sum on columns to generate a 1D problem.
- Partition it in P parts.
- For the first stripe, sum on rows.
- Partition it in Q parts.
- Treat all stripes.

Complexity :
$O((P \log \frac{n_1}{P})^2 + P \times (Q \log \frac{n_2}{Q})^2)$.

How good is that ?

Theorem

If there are no zero in the array, the heuristic $P \times Q$-way partitioning is a $(1 + \Delta \frac{P}{n_1})(1 + \Delta \frac{Q}{n_2})$-approximation algorithm where $\Delta = \frac{\max A}{\min A}$, $P < n_1$, $Q < n_2$.

How good is that ?

Theorem

If there are no zero in the array, the heuristic $P \times Q$-way partitioning is a $(1 + \Delta\frac{P}{n_1})(1 + \Delta\frac{Q}{n_2})$-approximation algorithm where $\Delta = \frac{\max A}{\min A}$, $P < n_1$, $Q < n_2$.

Proof.

One dimension guarantee (upper bound) $L_{max}(DC) \leq \frac{\sum_{i'} A[i']}{m} + \max_i A[i]$ can be rewritten as $L_{max}(DC) \leq \frac{\sum A[i]}{m}(1 + \Delta\frac{m}{n})$.
It allows to bound the imbalance of a stripe :
$Load_{stripe} \leq \frac{\sum A[i][j]}{P}(1 + \Delta\frac{P}{n_1})$.
And finally of a processor : $L_{max} \leq (1 + \Delta\frac{P}{n_1})(1 + \Delta\frac{Q}{n_2})$. □

How good is that ?

Theorem

If there are no zero in the array, the heuristic $P \times Q$-way partitioning is a $(1+\Delta\frac{P}{n_1})(1+\Delta\frac{Q}{n_2})$-approximation algorithm where $\Delta = \frac{\max A}{\min A}$, $P < n_1$, $Q < n_2$.

Proof.

One dimension guarantee (upper bound) $L_{max}(DC) \leq \frac{\sum_{i'} A[i']}{m} + \max_i A[i]$ can be rewritten as $L_{max}(DC) \leq \frac{\sum A[i]}{m}(1+\Delta\frac{m}{n})$.
It allows to bound the imbalance of a stripe :
$Load_{stripe} \leq \frac{\sum A[i][j]}{P}(1+\Delta\frac{P}{n_1})$.
And finally of a processor : $L_{max} \leq (1+\Delta\frac{P}{n_1})(1+\Delta\frac{Q}{n_2})$. □

Theorem

The approximation ratio is minimized by $P = \sqrt{m\frac{n_1}{n_2}}$.

An optimal PxQ jagged partitioning

A Dynamic Programming Formulation

$$\begin{cases} L_{max}(n_1, P) = \min_{1 \leq k < n_1} \max L_{max}(k-1, P-1), 1D(k, n_1, Q) \\ L_{max}(0, P) = 0 \\ L_{max}(n_1, 0) = +\infty, \forall n_1 \geq 1 \end{cases}$$

- $O(n_1 m)$ L_{max} functions.
- $O(n_1^2)$ 1D functions.

For a 512x512 matrix and 1000 processors, that's 512,000+262,144 values. On 64-bit values, that's 6MB.

An optimal PxQ jagged partitioning

A Dynamic Programming Formulation

$$\begin{cases} L_{max}(n_1, P) = \min_{1 \leq k < n_1} \max L_{max}(k-1, P-1), 1D(k, n_1, Q) \\ L_{max}(0, P) = 0 \\ L_{max}(n_1, 0) = +\infty, \forall n_1 \geq 1 \end{cases}$$

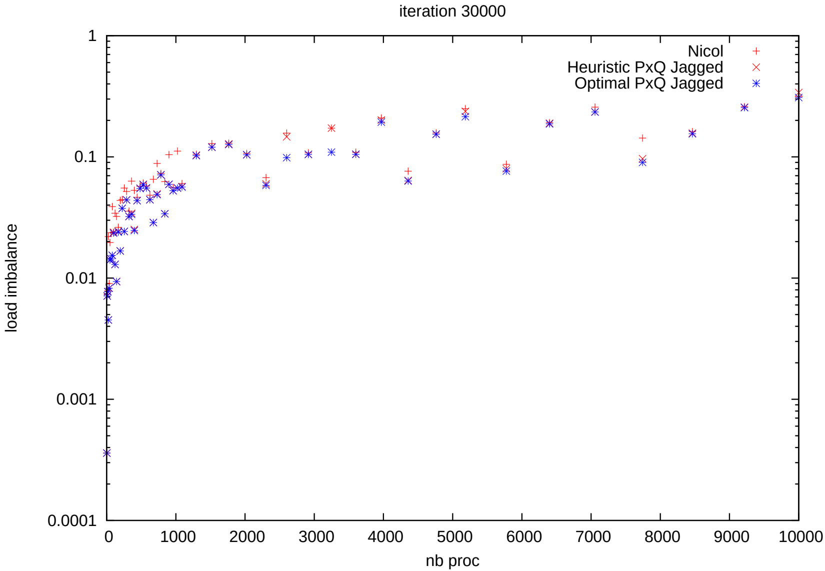
- $O(n_1 m)$ L_{max} functions.
- $O(n_1^2)$ 1D functions.

For a 512x512 matrix and 1000 processors, that's 512,000+262,144 values. On 64-bit values, that's 6MB.

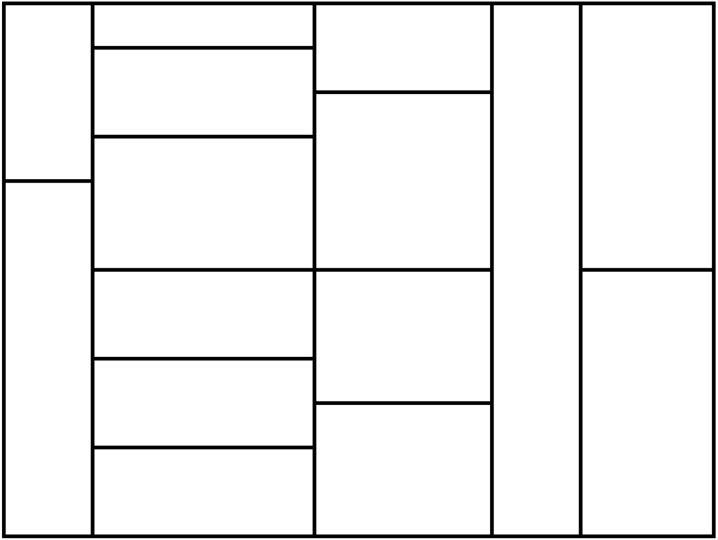
Not all values need to be stored

- Binary search on k.
- Lower bound/Upper bound on L_{max} and $1D$.
- Tree pruning.

Performance of PxQ jagged Partitioning



m-way Jagged Partitioning



m-way jagged partitioning heuristic

Algorithm

Cut in P stripes. Distribute processors in each stripe proportionally to the stripe's load : $alloc_j = \left\lceil \frac{\sum_{i,j} A[i][j]}{load_j}(m - P) \right\rceil$.

m-way jagged partitioning heuristic

Algorithm

Cut in P stripes. Distribute processors in each stripe proportionally to the stripe's load : $alloc_j = \left\lceil \frac{\sum_{i,j} A[i][j]}{load_j}(m-P) \right\rceil$.

Theorem

If there are no zero in A, the approximation ratio of the described algorithm is $\frac{m}{m-P}(1+\frac{\Delta}{n_2}) + \frac{m\Delta}{Pn_2}(1+\frac{\Delta P}{n_1})$.

Proof.

Same kind of proof than for heuristic PxQ jagged partitioning. □

Recall that the guarantee of heuristic PxQ jagged partitioning was: $(1+\Delta\frac{P}{n_1})(1+\Delta\frac{Q}{n_2})$. m-way is better for large m values.

An optimal m-way partitioning

A Dynamic Programming Formulation

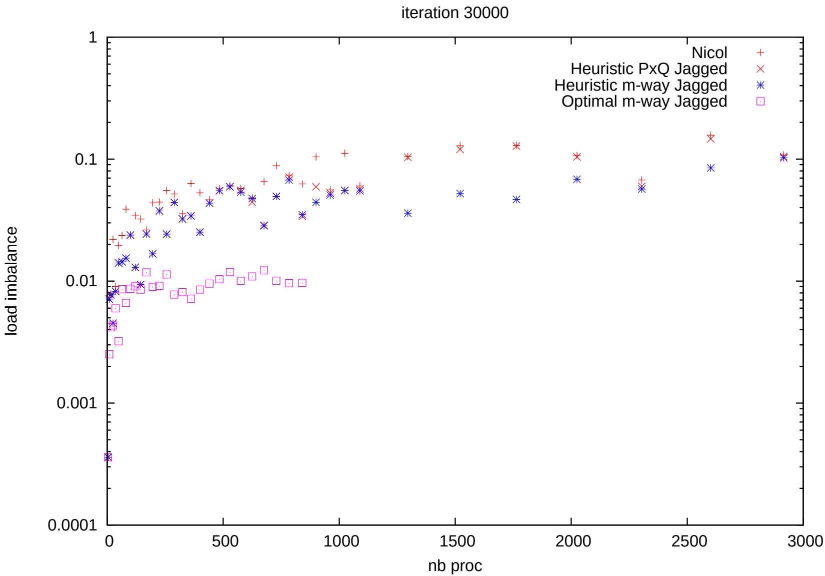
$$\begin{cases} L_{max}(n_1, m) = \min_{1 \leq k < n_1, 1 \leq x \leq m} \max L_{max}(k-1, m-x), 1D(k, n_1, x) \\ L_{max}(0, m) = 0 \\ L_{max}(n_1, 0) = +\infty, \forall n_1 \geq 1 \end{cases}$$

- $O(n_1 m)$ L_{max} functions.
- $O(n_1^2 m)$ 1D functions.

The same kind of optimizations apply.

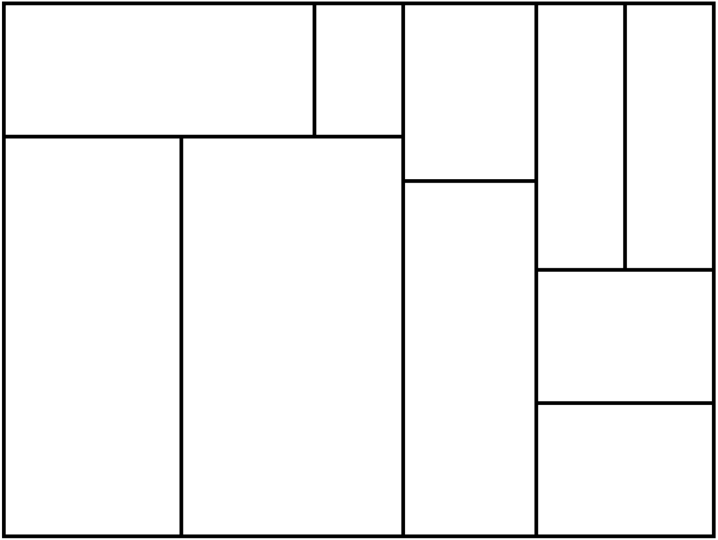
For a 512x512 matrix on 1,000 processors. That's 512,000 + 262,144,000 values, if they are 64-bits, about 2GB (and takes 30 minutes).

Performance of *m*-way

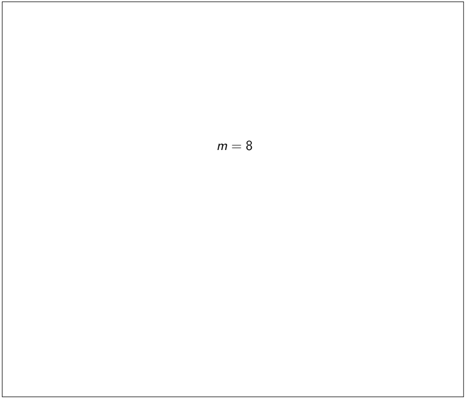


Outline of the Talk

Hierarchical Bisection Partitioning



Recursive Bisection [BB87]

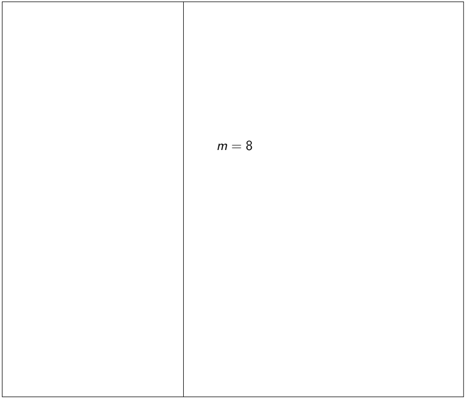


Algorithm

- m processors to partition a rectangle.

Complexity: $O(m \log \max n_1, n_2)$.

Recursive Bisection [BB87]

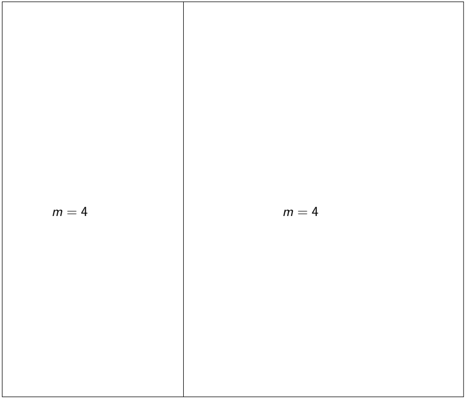


Algorithm

- m processors to partition a rectangle.
- Cut to balance the load evenly.

Complexity: $O(m \log \max n_1, n_2)$.

Recursive Bisection [BB87]

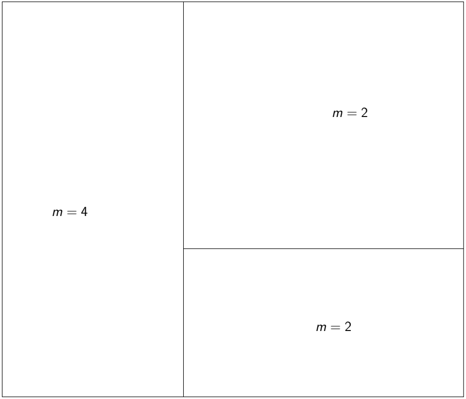


Algorithm

- m processors to partition a rectangle.
- Cut to balance the load evenly.
- Allocate half the processors to each side.

Complexity: $O(m \log \max n_1, n_2)$.

Recursive Bisection [BB87]

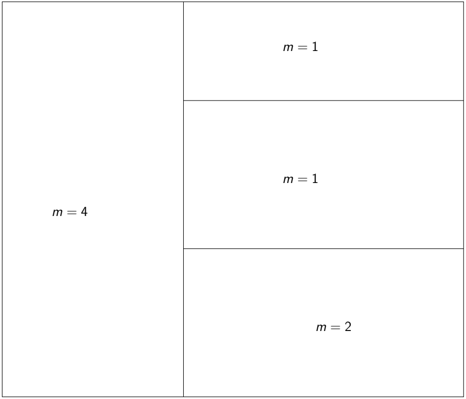


Algorithm

- m processors to partition a rectangle.
- Cut to balance the load evenly.
- Allocate half the processors to each side.

Complexity: $O(m \log \max n_1, n_2)$.

Recursive Bisection [BB87]

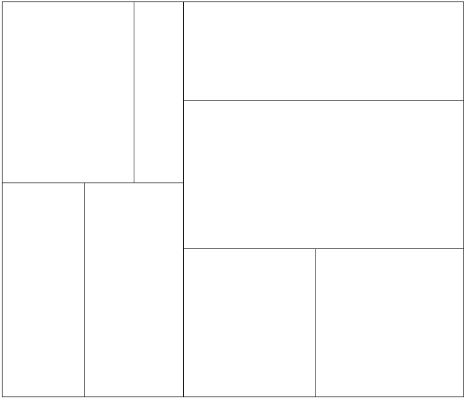


Algorithm

- m processors to partition a rectangle.
- Cut to balance the load evenly.
- Allocate half the processors to each side.
- Cut the dimension that balances the load best.

Complexity: $O(m \log \max n_1, n_2)$.

Recursive Bisection [BB87]

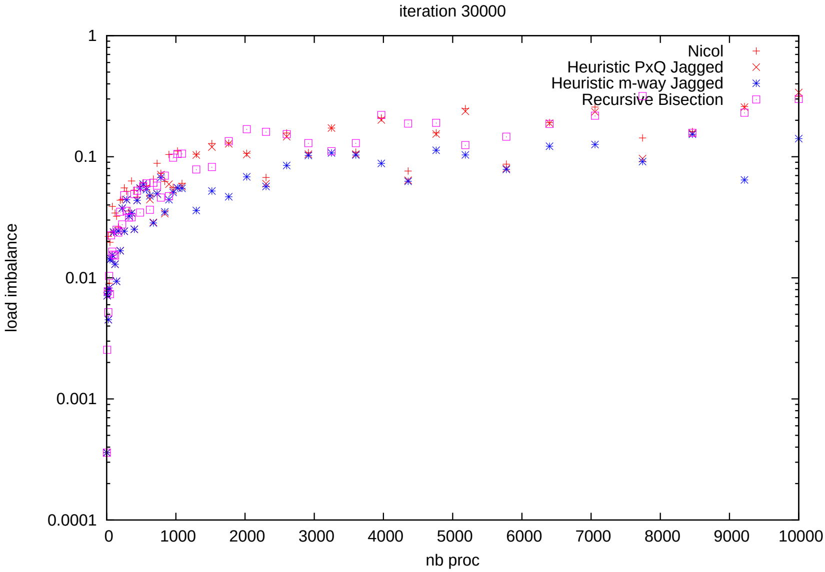


Algorithm

- m processors to partition a rectangle.
- Cut to balance the load evenly.
- Allocate half the processors to each side.
- Cut the dimension that balances the load best.

Complexity: $O(m \log \max n_1, n_2)$.

Performance of Recursive Bisection



An Optimal Hierarchical Bisection Algorithm

A Dynamic Programming Formulation

$$\begin{cases} L_{max}(x_1, x_2, y_1, y_2, m) = \min_j \min \\ \quad (\min_x \max L_{max}(x_1, x, y_1, y_2, j), L_{max}(x+1, x_2, y_1, y_2, m-j)) \\ \quad , (\min_y \max L_{max}(x_1, x_2, y_1, y, j), L_{max}(x_1, x_2, y+1, y_2, m-j)) \end{cases}$$

- $O(n_1^2 n_2^2 m)$ L_{max} functions.

For a 512x512 matrix and 1000 processors, that's 68,719,476,736,000 values. On 64-bit values, that's 544TB.

An Optimal Hierarchical Bisection Algorithm

A Dynamic Programming Formulation

$$\begin{cases} L_{max}(x_1, x_2, y_1, y_2, m) = \min_j \min \\ \quad (\min_x \max L_{max}(x_1, x, y_1, y_2, j), L_{max}(x+1, x_2, y_1, y_2, m-j)) \\ \quad , (\min_y \max L_{max}(x_1, x_2, y_1, y, j), L_{max}(x_1, x_2, y+1, y_2, m-j)) \end{cases}$$

- $O(n_1^2 n_2^2 m)$ L_{max} functions.

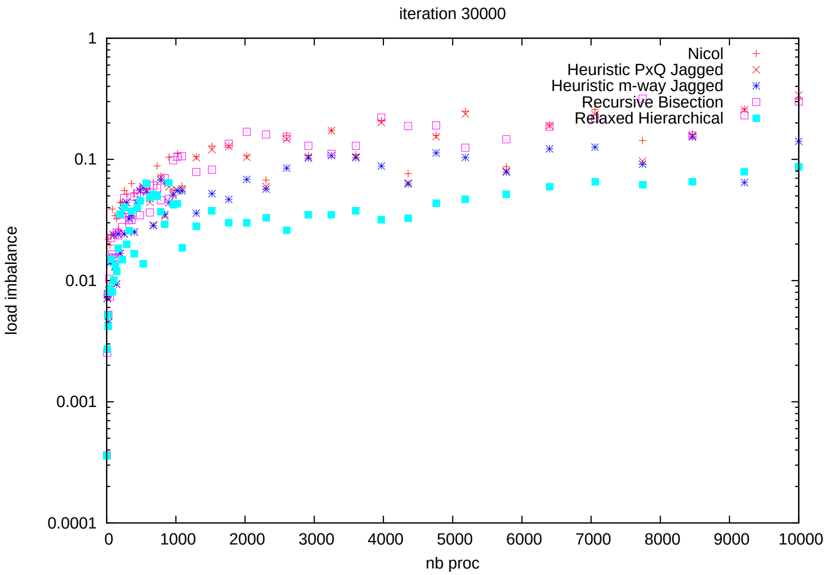
For a 512x512 matrix and 1000 processors, that's 68,719,476,736,000 values. On 64-bit values, that's 544TB.

The Relaxed Hierarchical Heuristic

Build the solution according to

$$\begin{cases} L_{max}(x_1, x_2, y_1, y_2, m) = \min_j \min \\ \quad (\min_x \max \frac{L(x_1,x,y_1,y_2)}{j}, \frac{L(x+1,x_2,y_1,y_2)}{m-j}) \\ \quad , (\min_y \max \frac{L(x_1,x_2,y_1,y)}{j}, \frac{L(x_1,x_2,y+1,y_2)}{m-j}) \end{cases}$$

Performance of Relaxed Hierarchical

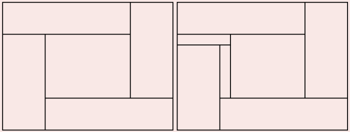


Outline of the Talk

More General ?

Recursively Defined Partitioning

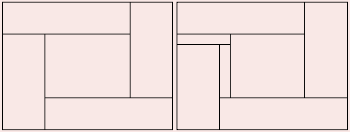
Most of them are polynomial by Dynamic Programming



More General ?

Recursively Defined Partitioning

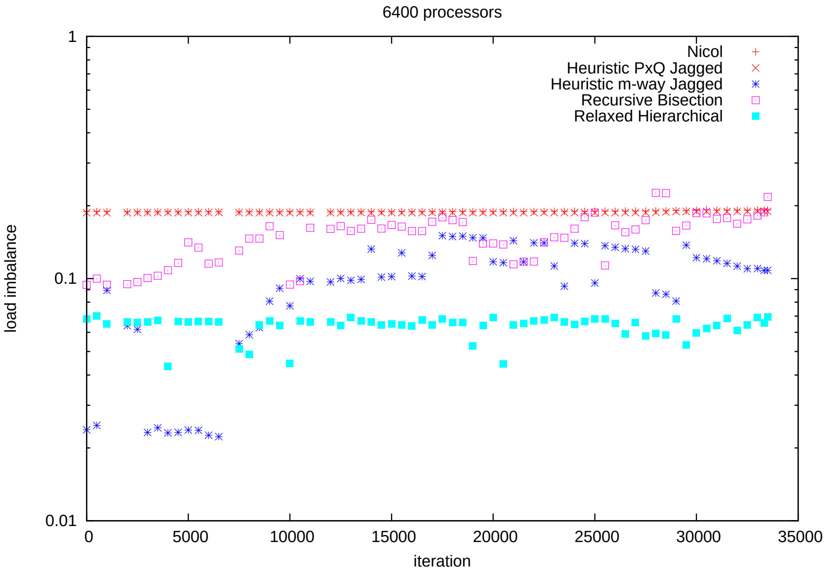
Most of them are polynomial by Dynamic Programming



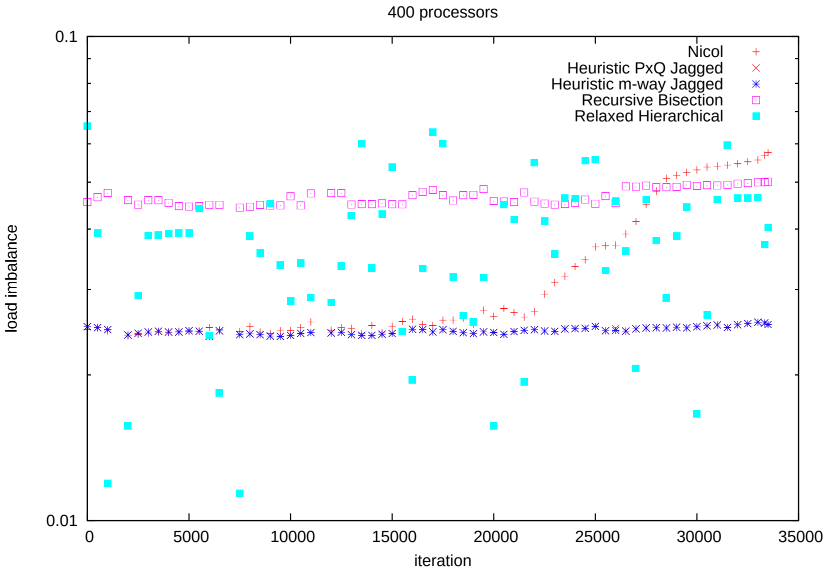
Arbitrary Rectangles

NP-Complete with a $\frac{5}{4}$ non-approximability result [KMP98].
There is a known 2-approximation of complexity $O(n_1 n_2 + m \log n_1 n_2)$ which heavily relies on linear programming [Pal06].

Performance Over the Execution

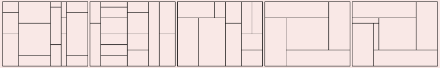


Relaxed Hierarchical Might Be Unstable



Conclusion and Perspective

Conclusion

- Proposed new classes of partitioning.
- Proved that most recursively defined classes are polynomial:
.
- Proposed two new well-founded heuristics which outperform state-of-the-art algorithm.
- Theoretically analyzed two heuristics.

Perspective

- Better *m*-way jagged partitioning algorithm.
- Integration into real physic simulation codes.
- Include communication models.

Thank you

Collaborators

Thanks to H. Karimabadi, A. Majumdar, Y.A. Omelchenko and K.B. Quest, collaborators of the Petaapps NSF OCI-0904802 grant, for providing the particle-in-cell dataset.

More information

contact : esaule@bmi.osu.edu
visit: http://bmi.osu.edu/hpc/

Research at HPC lab is funded by

Bengt Aspvall, Magnús M. Halldórsson, and Fredrick Manne.
Approximations for the general block distribution of a matrix.
Theor. Comput. Sci., 262(1-2):145–160, 2001.

Marsha Berger and Shahid Bokhari.
A partitioning strategy for nonuniform problems on multiprocessors.
IEEE Transaction on Computers, C36(5):570–580, 1987.

Daya Ram Gaur, Toshihide Ibaraki, and Ramesh Krishnamurti.
Constant ratio approximation algorithms for the rectangle stabbing problem and the rectilinear partitioning problem.
J. Algorithms, 43(1):138–152, 2002.

Michelangelo Grigni and Fredrik Manne.
On the complexity of the generalized block distribution.
In *IRREGULAR '96: Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 319–326, London, UK, 1996. Springer-Verlag.

S. Khanna, S. Muthukrishnan, and M. Paterson.

On approximating rectangle tiling and packaging.
In *proceedings of the 19th SODA*, pages 384–393, 1998.

Sanjeev Khanna, S. Muthukrishnan, and Steven Skiena.
Efficient array partitioning.
In *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 616–626, London, UK, 1997. Springer-Verlag.

Serge Miguet and Jean-Marc Pierson.
Heuristics for 1d rectilinear partitioning as a low cost and high quality answer to dynamic load balancing.
In *HPCN Europe '97: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 550–564, London, UK, 1997. Springer-Verlag.

Fredrik Manne and Tor Sørevik.
Partitioning an array onto a mesh of processors.

In *PARA '96: Proceedings of the Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization*, pages 467–477, London, UK, 1996. Springer-Verlag.

S. Muthukrishnan and Torsten Suel.
Approximation algorithms for array partitioning problems.
Journal of Algorithms, 54:85–104, 2005.

David Nicol.
Rectilinear partitioning of irregular data parallel computations.
Journal of Parallel and Distributed Computing, 23:119–134, 1994.

Ali Pinar and Cevdet Aykanat.
Fast optimal load balancing algorithms for 1d partitioning.
Journal of Parallel and Distributed Computing, 64:974–996, 2004.

K. Paluch.
A new approximation algorithm for multidimensional rectangle tiling.
In *Proceedings of ISAAC*, 2006.