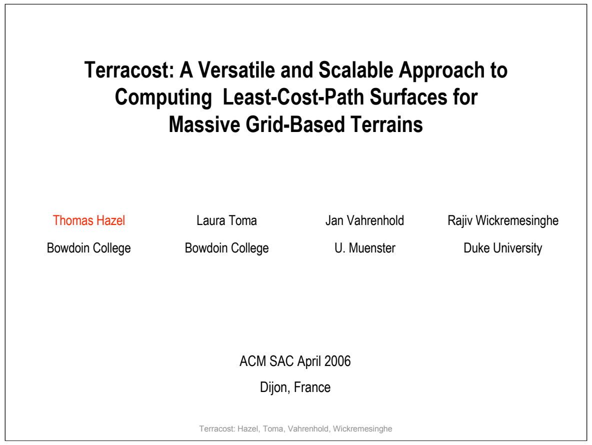


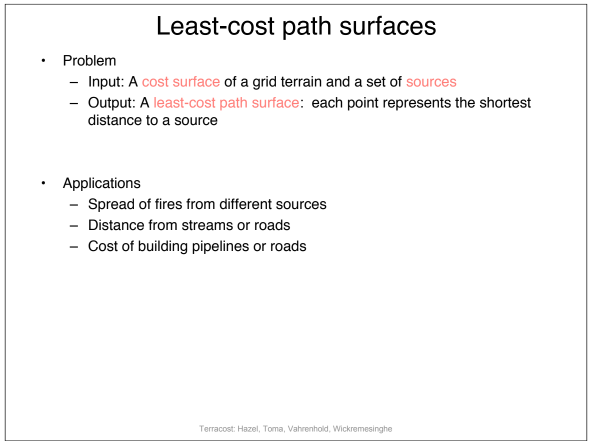
Terracost: A Versatile and Scalable Approach to Computing Least-Cost-Path Surfaces for Massive Grid-Based Terrains

Thomas Hazel, Bowdoin College
Laura Toma, Bowdoin College
Jan Vahrenhold, U. Muenster
Rajiv Wickremesinghe, Duke University

ACM SAC April 2006
Dijon, France

My name is TH and this is work that I have done with Laura Toma also from Bowdoin College, Jan V from U Muenster and R. W. from Duke University.

[Read the title to give a little time on this slide. E.g.: “I’ll be talking about a versatile…”]



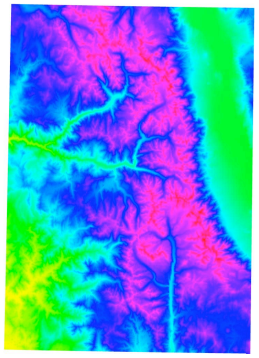
Least-cost path surfaces

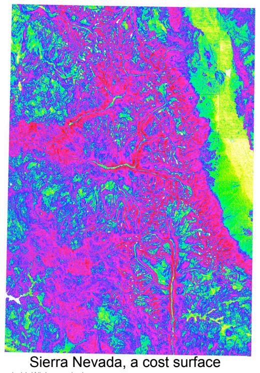
- Problem
  - Input: A cost surface of a grid terrain and a set of sources
  - Output: A least-cost path surface: each point represents the shortest distance to a source
- Applications
  - Spread of fires from different sources
  - Distance from streams or roads
  - Cost of building pipelines or roads

The least-cost path surface computation takes as input…
And outputs …

This is a useful part of many applications in GIS. For example…

Grid terrains

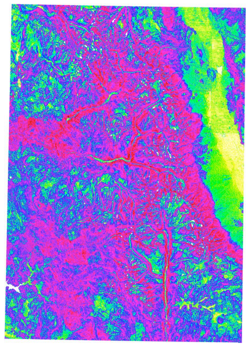

Sierra Nevada, 30m resolution

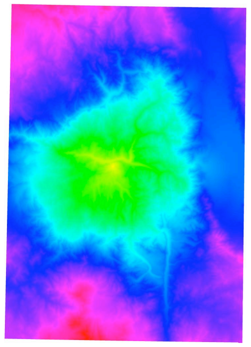

Sierra Nevada, a cost surface

Terracost: Hazel, Toma, Vahrenhold, Wickremesinghe

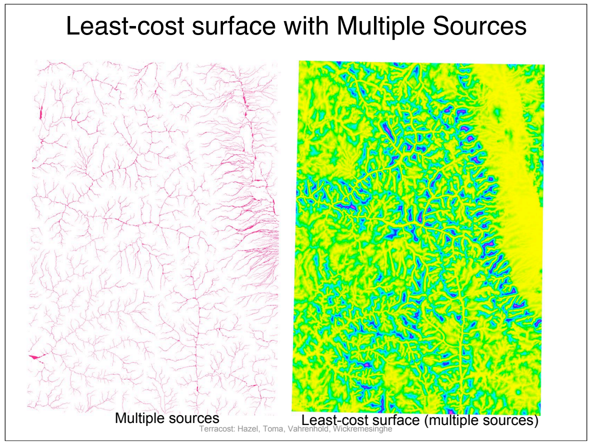
[on the right: example cost surface]

Least-Cost Surface with one source

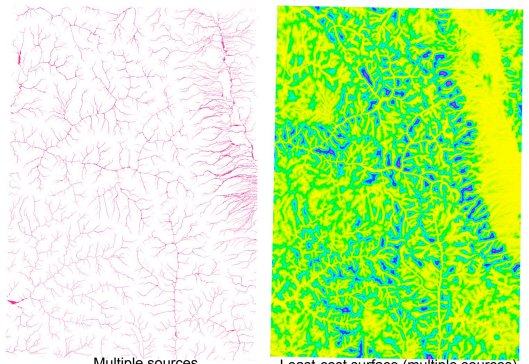

Cost surface


Least-cost path surface (1 source)

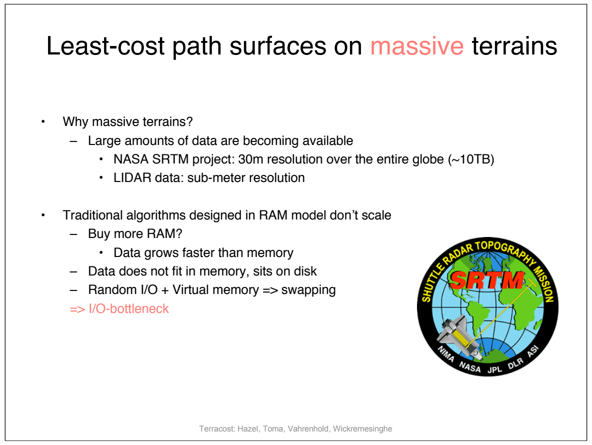
Terracost: Hazel, Toma, Vahrenhold, Wickremesinghe



Least-cost surface with Multiple Sources



[sometimes makes sense to have a large number of sources. For example the river network, if we want to find the distance to flowing water]

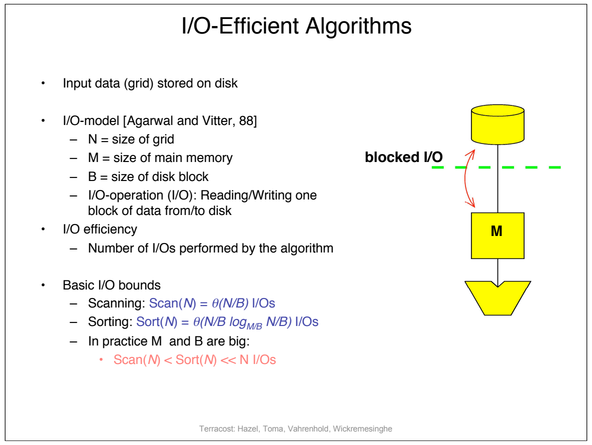


Least-cost path surfaces on massive terrains

- Why massive terrains?
  - Large amounts of data are becoming available
    - NASA SRTM project: 30m resolution over the entire globe (~10TB)
    - LIDAR data: sub-meter resolution
- Traditional algorithms designed in RAM model don't scale
  - Buy more RAM?
    - Data grows faster than memory
  - Data does not fit in memory, sits on disk
  - Random I/O + Virtual memory => swapping
  - => I/O-bottleneck

Note to self: People tend to think “why not buy more memory”? Data grows faster than memory.

Note to self: if anybody asks: for a raw dataset of 1 billion elements, need about 20B per element. That is 20GB of RAM. And that's just for a not-so-large dataset.


blocked I/O
M

I/O-Efficient Algorithms

- Input data (grid) stored on disk
- I/O-model [Agarwal and Vitter, 88]
  - N = size of grid
  - M = size of main memory
  - B = size of disk block
  - I/O-operation (I/O): Reading/Writing one block of data from/to disk
- I/O efficiency
  - Number of I/Os performed by the algorithm
- Basic I/O bounds
  - Scanning: $\mathrm{Scan}(N) = \theta(N/B)$ I/Os
  - Sorting: $\mathrm{Sort}(N) = \theta(N/B \log_{M/B} N/B)$ I/Os
  - In practice M and B are big:
    - $\mathrm{Scan}(N) < \mathrm{Sort}(N) \ll N$ I/Os

In order to address the I/O-bottleneck that we see when dealing with massive data, we need to change the model we use to design algorithms.

The standard model used for developing I/O-eff algo was prop by A&V. It assume the machine has a main mem of size M, and an infinite disk, The data , in this case the grid, is assumed to reside on disk; computation can be done only on data present in main memory. the data is moved between main memory and disk in blocks of size B..

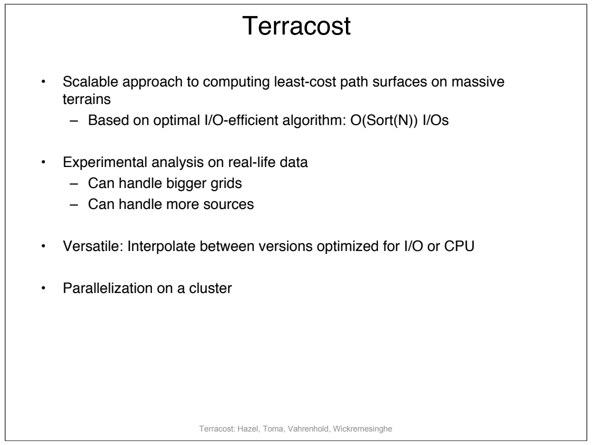
In this model we count I/O-operations. One operation is reading or writing one block of data.

There are two major bounds that are used in this model.

Scanning is the number of I/Os required to load a problem of size N into memory from disk.

Sorting is the number of I/Os required to sort N items on disk.

basically the log factor is v small and in practice it is very close to N/B. Because of big block size scan (N) < sort(N) <<<< N I/Os



Terracost

- Scalable approach to computing least-cost path surfaces on massive terrains
  - Based on optimal I/O-efficient algorithm: O(Sort(N)) I/Os
- Experimental analysis on real-life data
  - Can handle bigger grids
  - Can handle more sources
- Versatile: Interpolate between versions optimized for I/O or CPU
- Parallelization on a cluster

Here is an overview of our results.

We developed Terracost which is a scalable approach to computing least-cost path surfaces on massive terrains.

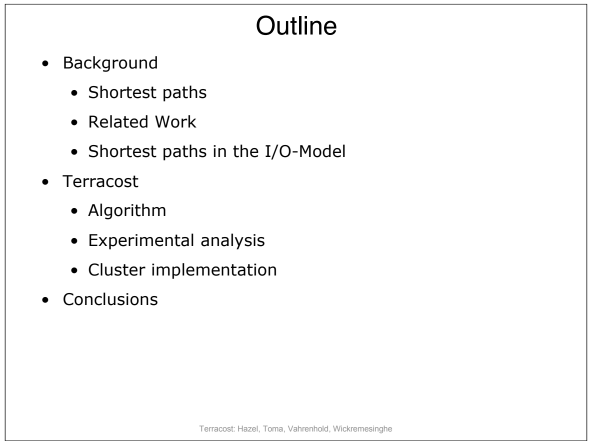
Terracost is based on an optimal I/O-efficient algorithm. This algorithm has a theoretical bound of sort N I/Os [for the SSSP problem]

(note: don't mention SSSP, we don't know what that is yet)

We present an experimental analysis on real-life data and show that terracost erforms well in practice and obtains a significant speed-up compared to existing algorithms.

We have implemented this algorithm such that a user can chose to make the application run more I/O-efficiently at the cost of CPU-efficiency or more CPU-efficiently at the cost of i/O-efficiency.

A very nice effect of algorithm is that we can run it in parallel on a cluster. (Note to self: while I agree that it is indeed straightforward, we should try to sell it a bit better)

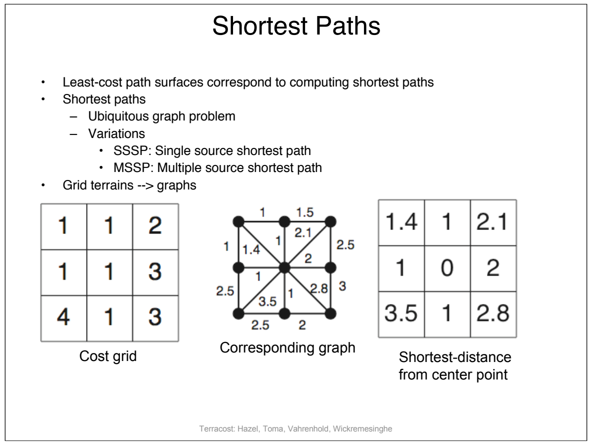


Outline

- Background
  - Shortest paths
  - Related Work
  - Shortest paths in the I/O-Model
- Terracost
  - Algorithm
  - Experimental analysis
  - Cluster implementation
- Conclusions

The outline for the rest of the talk is the following: first I'll describe some background on computing least-cost surfaces, and related work in internal and external memory. Then I will talk about the approach we use in Terracost, describe the design, and the experimental analysis and present the results. Finally I will present some highlights on the cluster implrementation and conclude.

Note: if you are short on time, cut this slide, just show it and read the lines in 10 seconds total.

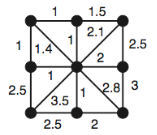


Shortest Paths

- Least-cost path surfaces correspond to computing shortest paths
- Shortest paths
  - Ubiquitous graph problem
  - Variations
    - SSSP: Single source shortest path
    - MSSP: Multiple source shortest path
- Grid terrains --> graphs

1	1	2
1	1	3
4	1	3

Cost grid



Corresponding graph

1.4	1	2.1
1	0	2
3.5	1	2.8

Shortest-distance from center point

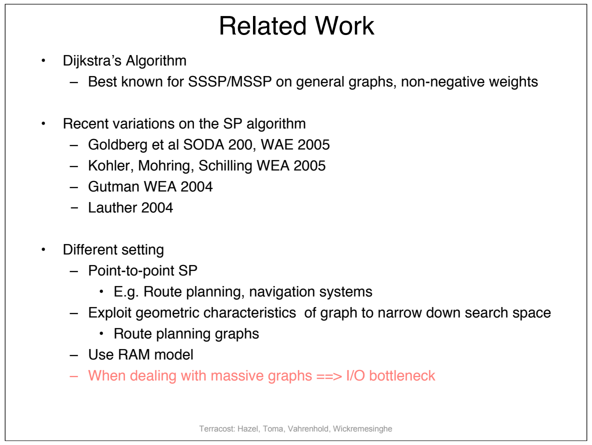
The problem of computing least-cost path surfaces on terrains is related to computing shortest paths in a graph that represent the terrain.

The shortest path problem is a ubiquitous problem in graph theory; two variations are computing SSSP and MSSP.

The connection between grid terrain and graphs is as follows:

[Define a grid terrain] A grid terrain is essentially a matrix of values that represent uniform samples from the terrain. The figure on the left shows a cost grid which we use as input for Terracost. Each point in the grid has a value associated with it, which is the cost of traversing that cell. Each grid celll is adjacent to 8 neighbors, and we model that by connecting each cell by edges with its 8 neighbors according to the manner shown in the middle figure. Each vertex is connected to its eight immediate neighbors, and for any the weight of each edge is the average of the cost of its two incident vertices multiplied by a scaling factor. The scaling factor in this case is 1 for all NSEW, edges and root 2 for all of the diagonal edges.

The figure on the right shows the output of the SP problem on grids where each point represents the least-cost (shortest distance) to get to the middle square (therefore it has a cost of 0).



Related Work

- Dijkstra's Algorithm
  - Best known for SSSP/MSSP on general graphs, non-negative weights
- Recent variations on the SP algorithm
  - Goldberg et al SODA 200, WAE 2005
  - Kohler, Mohring, Schilling WEA 2005
  - Gutman WEA 2004
  - Lauther 2004
- Different setting
  - Point-to-point SP
    - E.g. Route planning, navigation systems
  - Exploit geometric characteristics of graph to narrow down search space
    - Route planning graphs
  - Use RAM model
  - When dealing with massive graphs ==> I/O bottleneck

Terracost: Hazel, Toma, Vahrenhold, Wickremesinghe

The best known algorithm for computing SSSP on general graphs with non-negative weights is Dijkstra's algorithm.

A variety of theoretical and practical approaches to computing shortest paths have been proposed.

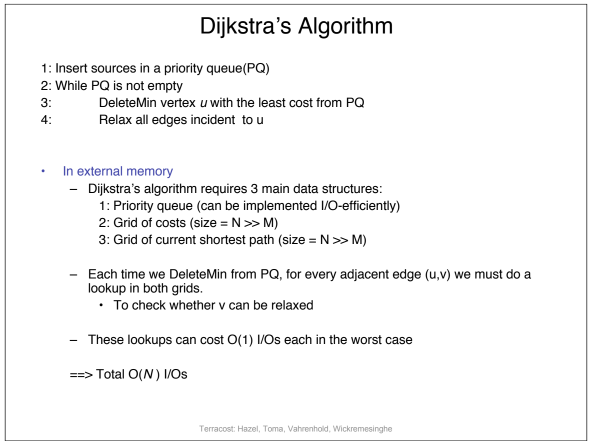
Some of the most recent results are listed below: […]

The setting that we consider is different for a couple pf reasons.:

- these algorithms are aimed at optimizing P2P shortest distance queries, whereas we are concerned with finding the SP to ALL points.

- the algorithms exploit geographic characteristics of the graph to narrow down the search space; in our case there is no association between geographic distance and the cost of traversing an edge.

- Most importantly, Dijkstra and the other algorithms are designed using the RAM model and assume the graph and all data fit in memory. They don't take into account the I/O-bottleneck that occurs when all of the problem data cannot fit into main memory.



Dijkstra's Algorithm

```
1: Insert sources in a priority queue(PQ)
2: While PQ is not empty
3:          DeleteMin vertex u with the least cost from PQ
4:          Relax all edges incident to u
```

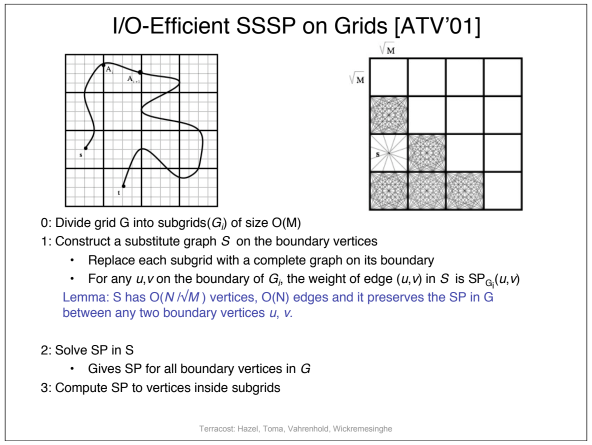
- In external memory
  - Dijkstra's algorithm requires 3 main data structures:
    - 1: Priority queue (can be implemented I/O-efficiently)
    - 2: Grid of costs (size = N >> M)
    - 3: Grid of current shortest path (size = N >> M)
  - Each time we DeleteMin from PQ, for every adjacent edge (u,v) we must do a lookup in both grids.
    - To check whether v can be relaxed
  - These lookups can cost O(1) I/Os each in the worst case

  ==> Total O(*N*) I/Os

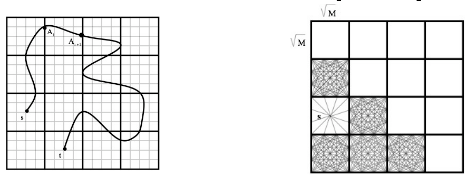
Terracost: Hazel, Toma, Vahrenhold, Wickremesinghe

[Describe Dijkstra very briefly; can cut this if pressed for time. Just say Dijkstra algorithm is as mentioned the best known algo for SSSP in internal memory.]

However, if the input graph does not fit in internal memory, Dijkstra's algorithm performs one I/O per edge. This is because for every vertex that is extracted from the PQ, for each one of its neighbor vertices v, it needs to access the cost grid and the current distance grid to see if v can be relaxed. If these two grids sit on disk, In the worst case this is one I/O per edge, or O(N) I/Os in total for a grid of N cells.



I/O-Efficient SSSP on Grids [ATV'01]



0: Divide grid G into subgrids(G_i) of size O(M)

1: Construct a substitute graph S on the boundary vertices

- Replace each subgrid with a complete graph on its boundary
- For any u,v on the boundary of G_i, the weight of edge (u,v) in S is $SP_{G_i}(u,v)$

Lemma: S has $O(N/\sqrt{M})$ vertices, O(N) edges and it preserves the SP in G between any two boundary vertices u, v.

2: Solve SP in S

- Gives SP for all boundary vertices in G

3: Compute SP to vertices inside subgrids

Terracost: Hazel, Toma, Vahrenhold, Wickremesinghe

The main idea of our base algorithm, the algorithm by ArgeTomaVitter, is to divide the grids into sub-grids, or tiles, that fit in memory and reduce the SSSP problem to SSSP on a smaller substitute graph defined only on the boundaries of the tiles.

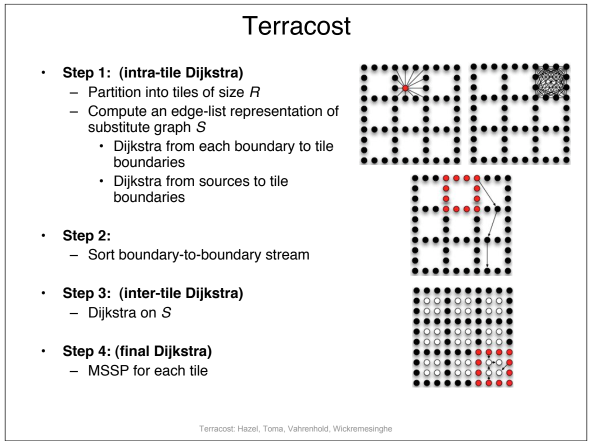
The motivation for this is the following observation: csd two arbitrary vertices, s, t in the grid, and the SP between them, and let A_i, A_{i+1} be the intersection of the path with a tile boundary. Then the piece of the SP inside a tile must be the SP between those two points inside the tile.

This is exploited as follows: we replace each tile with a with a complete graph on its boundary vertices. The weight of an edge between 2 bnd vertices is the length of the shortest path between the bnd vertices WITHIN that tile.

In this manner we define a a \emph{reduced} version of the grid, S --- it has fewer vertices; [Note to self: for simplicity consider the source vertex is a boundary vertex]. The claim is that the SP between two bnd vertices is the same in S as in the original grid.

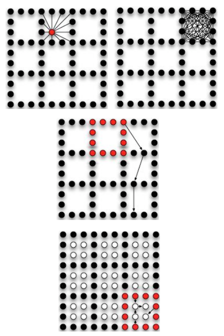
[putting everything together:]

the SSSP algorithm consists of three steps: (1) Compute a substitute graph S defined on the boundary vertices with the property that the lengths of the shortest paths in S are the same as in the grid.



Terracost

- **Step 1: (intra-tile Dijkstra)**
  - Partition into tiles of size R
  - Compute an edge-list representation of substitute graph S
    - Dijkstra from each boundary to tile boundaries
    - Dijkstra from sources to tile boundaries
- **Step 2:**
  - Sort boundary-to-boundary stream
- **Step 3: (inter-tile Dijkstra)**
  - Dijkstra on S
- **Step 4: (final Dijkstra)**
  - MSSP for each tile



Terracost: Hazel, Toma, Vahrenhold, Wickremesinghe

Our implementation of Terracost follows the generic algorithm and is broken down into four steps.

The first step breaks the grid up into tiles of size R, where R is a parameter that the user can specify. Then we compute the substitute graph for each tile. That is, we run Dijkstra from each bnd vertex to its tile boundaries, and If there are sources in the tile we compute the SP from those sources to the boundary points.

In step 2 we sort the boundary-to-boundary shortest paths (which represent the edge list representation of the substitute graph).

We have separated this into its own step because it can take a significant amount of time.

Step 3 takes the boundary points for any tile which contained a source in step 1 and uses those points as sources for MSSP on the substitute graph.

At the end of step 3 we know the least cost path to all the points in the substitute graph which are also all of the boundary points in the original graph.

Finally in step 4 we use the boundary points for each tile as sources and run MSSP to completion on each tile so that we now have the

Experimental Analysis

Dataset	Grid Size (million elements)	MB (Grid Only)
Kaweah	1.6	6
Puerto Rico	5.9	24
Hawaii	28.2	112
Sierra Nevada	9.5	38
Cumberlands	67	268
Lower New England	77.8	312
Midwest USA	280	1100

Experimental Platform

- Apple Power Macintosh G5
- Dual 2.5 GHz processors
- 512 KB L2 cache
- 1 GB RAM

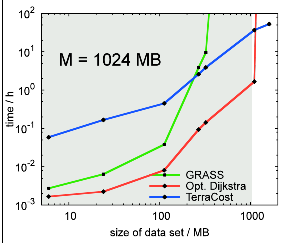
Compare Terracost with r.cost in GRASS

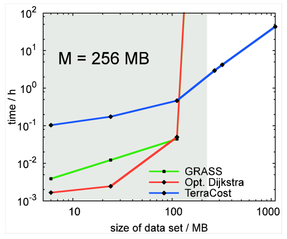
- r.cost has same functionality
- GRASS users have complained it is very slow for large terrains

[in case anyone asks: need space for the computation in addition to the grid]

Experimental Analysis

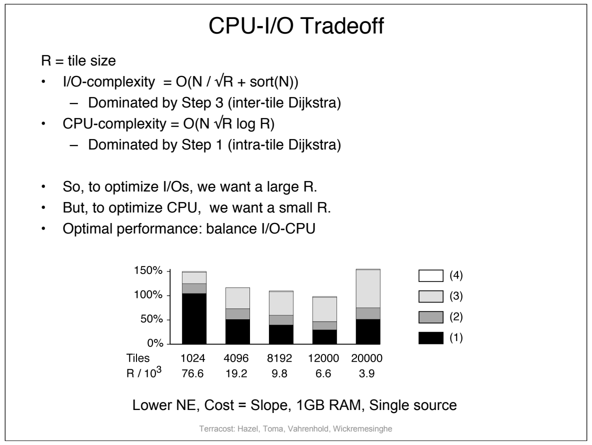
- GRASS: r.cost
- Opt Dijkstra: internal memory version of Terracost (num tiles = 1)
- Terracost: I/O-efficient version of Terracost





[this is the main slide of the talk. Don’t speed. Take a deep breath].

[always read out the graph axes: “time” “data set size”]

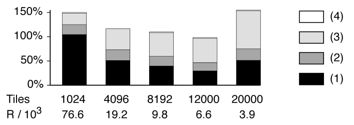


CPU-I/O Tradeoff

R = tile size

- I/O-complexity = O(N / √R + sort(N))
  - Dominated by Step 3 (inter-tile Dijkstra)
- CPU-complexity = O(N √R log R)
  - Dominated by Step 1 (intra-tile Dijkstra)

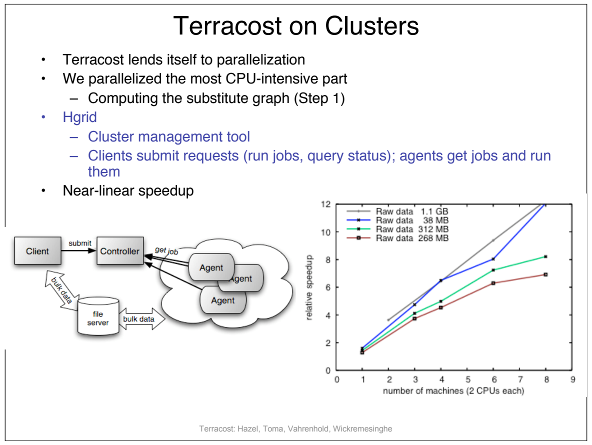
- So, to optimize I/Os, we want a large R.
- But, to optimize CPU, we want a small R.
- Optimal performance: balance I/O-CPU



Lower NE, Cost = Slope, 1GB RAM, Single source

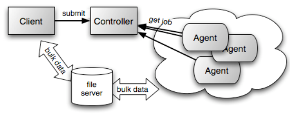
[here you pretty much read the slide.]

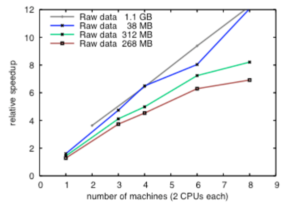
The figure shows the running time of terracost for the same dataset (lower new england) with various tile sizes. The y axis shows runtime relative to optimal tile size, which is somewhere around 12000 tiles. We see how the proportion between the 4 steps of the algorithms changes. Step 1 is dominant for large tille size (I.e. few tiles), and it gets smaller as the tile size decreases. Step 3, on the other hand, is small for large R and it gets dominant for small R (I.e. many tiles).



Terracost on Clusters

- Terracost lends itself to parallelization
- We parallelized the most CPU-intensive part
  - Computing the substitute graph (Step 1)
- Hgrid
  - Cluster management tool
  - Clients submit requests (run jobs, query status); agents get jobs and run them
- Near-linear speedup



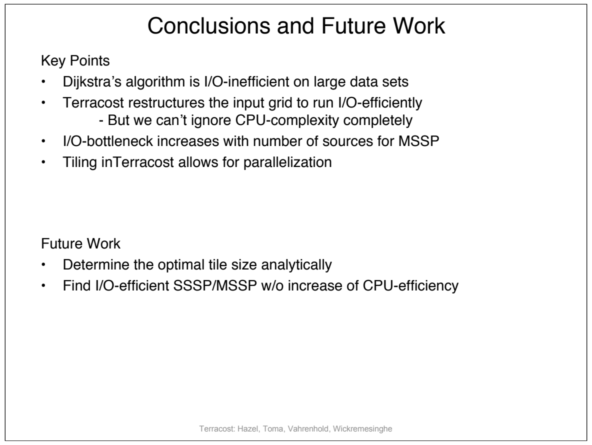


Terracost naturally lends itself to parallelization. In particular we parallelized the most CPU-intensive step, which is INTRA-tile dijkstra (step 1). For this we developed a cluster management tool called Hgrid.

Hgrid is conceptually very similar to Apple's Xgrid, it is implementd in Perl. It is designed to be easy to setup.

The figure illustrates the components of Hgrid. Clients submit requests (requests to run jobs or query status). Agents get jobs and run them.

The figure shows the speedup of step1 for various datasets, as machines are added to a cluster (Each machine has 2 processors). We see that terracost obtains almost a linear speedup.



Conclusions and Future Work

Key Points

- Dijkstra's algorithm is I/O-inefficient on large data sets
- Terracost restructures the input grid to run I/O-efficiently
  - But we can't ignore CPU-complexity completely
- I/O-bottleneck increases with number of sources for MSSP
- Tiling inTerracost allows for parallelization

Future Work

- Determine the optimal tile size analytically
- Find I/O-efficient SSSP/MSSP w/o increase of CPU-efficiency

[You'll probably run out of time. Its ok to just put this slide up and say that you are out of time, but you'll b happy to discuss future work after the talk.]

Thank you for your attention.

Thank you.