Parallel Highway-Node Routing

Manuel Holtgrewe

Institut für Theoretische Informatik, Algorithmik II
Universität Karlsruhe (TH)

January 11th, 2008

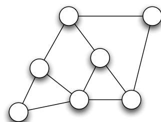
Overview

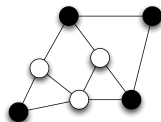
1. Preliminaries
2. Implementation
3. Experimental Results
4. Future Work

Highway-Node Routing

Overview

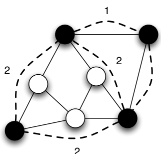
- scheme (precomputation + query algorithm) for finding shortest paths by Schultes et al.
- *dynamic* variant allows updates to edge weights
- vision: graph reflects known traffic congestions





General Idea

- given a graph $G = (V, E)$ and highway-nodes $V' \subset V$
- find *overlay graph* $G' = (V', E')$, such that $d_G(u, v) = d_{G'}(u, v)\ \forall u, v \in V'$
- construction: start a "local" DIJKSTRA search for all $u \in V'$



OpenMP

Overview

- API specification for shared memory parallel programming in C++ (and Fortran)
- functions for timing, locks, queries regarding program state
- pragmas to mark certain sections as parallel to the compiler
- runtime environment that does thread pooling, load balancing
- originally intented for scientific computation
- *join/fork* model

Parallel Loops

- for loops can be parallelized easily
- integer sequence is separated into chunks
- load balancing schemes: *static*, *dynamic*, *guided*

OpenMP (example)

```
int *a, *b, *c;

// ...

#pragma omp parallel for schedule(guided, 1)
for (int i = 0; i < 1<<20; ++i) {
  a[i] = b[i] + c[i]
}

#pragma omp parallel
{
  printf("thread # %d\n", omp_get_thread_num());
}
```

MCSTL

Overview

- essentially parallel version of the C++ STL algorithms by Singler et al.
- uses OpenMP for parallelisation
- parallel `sort`, `merge` and extensions like `multiway_merge`
- integrated as *libstdc++ parallel mode* into GCC 4.3 branch

Parallel `std::for_each`

- a *functor* is called on each element of a sequence
- provides sophisticated load balancing with *work stealing*
  - sequence is separated into p parts and placed in a dequeue for each thread
  - each thread takes k elements from its dequeue for processing
  - when done, threads try to steal half the work of other threads

MCSTL (example)

Square

```
struct f {
  void operator()(int &x) { x = x * x; }
};

int *arr = new int[10];

// ...

std::for_each(arr, arr + 10, f());
```

Implementation (1)

Graph Structure

- essentially forward star/adjacency array
- augmented with "level nodes", edges can be inserted/deleted

Step 1: Shortest Path Tree Construction

- $\forall u \in V' : \textsc{Local-Dijkstra}(u, V')$
- for all reached nodes v: mark existing (u, v) for upgrading or add it
- *parallelization*
  - DIJKSTRA searches independent, do "for all" in parallel
  - only add edge if there is enough space in edge bucket
  - otherwise, add them to a buffer and bulk-add them later

Implementation (2)

Step 2: Highway Edge Dispatching

- for all previously marked edges (u, v): "split" them if necessary, move them in edge buckets
- *parallelization*
  - edge splitting can be done in parallel
  - moving edges in buckets can be done in parallel, too

Step 3: Edge Reduction

- some edges might be superflous, remove them
- DIJKSTRA reduction search for all nodes $u \in V'$, abort if all neighbours of u have been settled or reached from another node, remove edges
- *parallelization*
  - again, DIJKSTRA searches can be performed in parallel
  - mark first, instead of removing
  - remove marked edges in second step

Experimental Results (1)

Platform

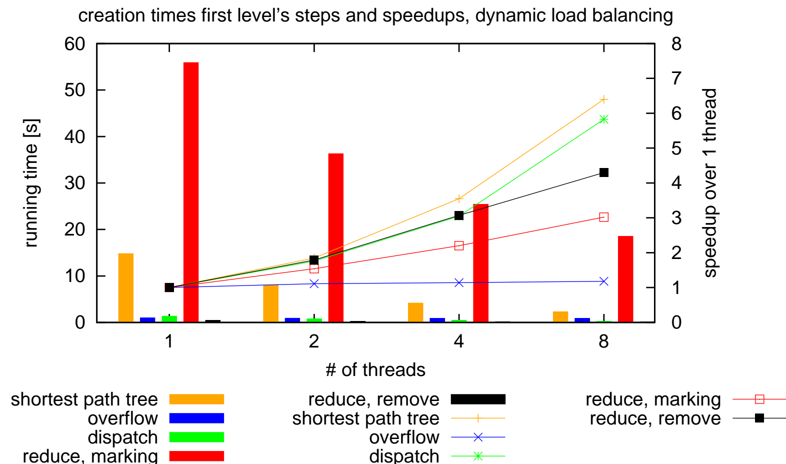
- Xeon 2.3 Ghz, 2 sockets; 4 cores, 2 $\times$ 4 MiB L2 caches each
- GCC C++ compiler, snapshot of 4.3 development branch
- Linux Kernel 2.6.18.8

Dynamic* Load Balancing

# of threads	1	2		4		8	
level 1	73.9	46.7	*2.1*	31.3	*2.4*	22.6	*3.3*
level 2	28.3	15.3	*1.8*	8.7	*3.2*	5.2	*5.4*
level 3	6.7	3.7	*1.8*	2.3	*2.9*	1.6	*4.2*
level 4	2.6	1.6	*1.6*	1.3	*2.1*	0.9	*2.8*
levels 5-8	$\leq$ 1 s each						

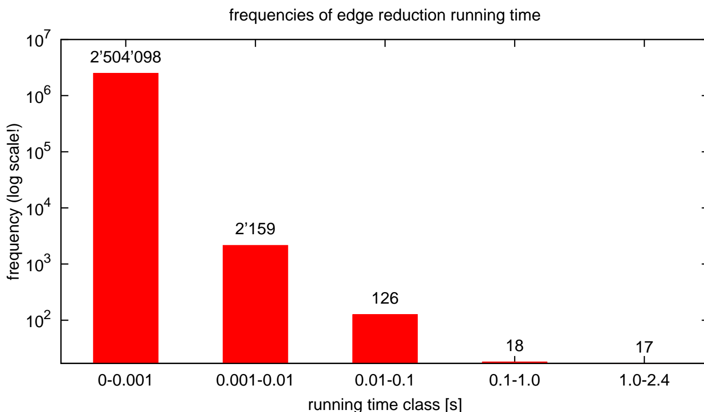
Experimental Results (2)

- creation of first level shows bad speedup, dominates running time
- consider creation time of first level



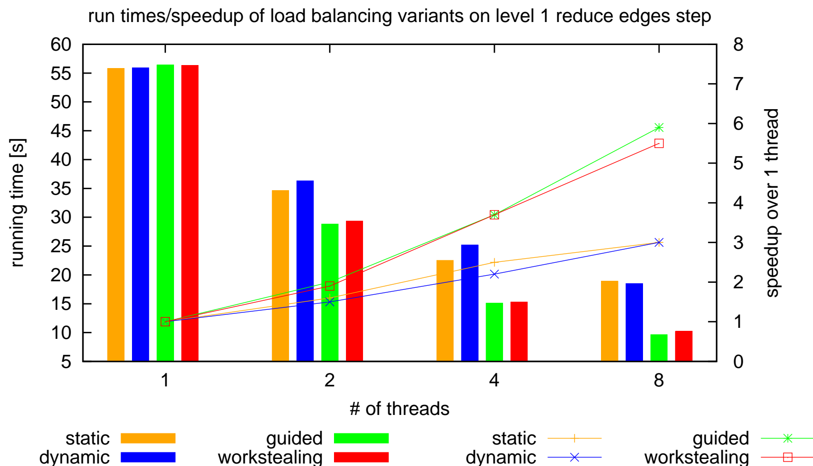
Experimental Results (3)

- load balancing problems in reduction step (long-ferry connections)
- observation: the "hard" nodes are clustered and – by chance – they have high indices

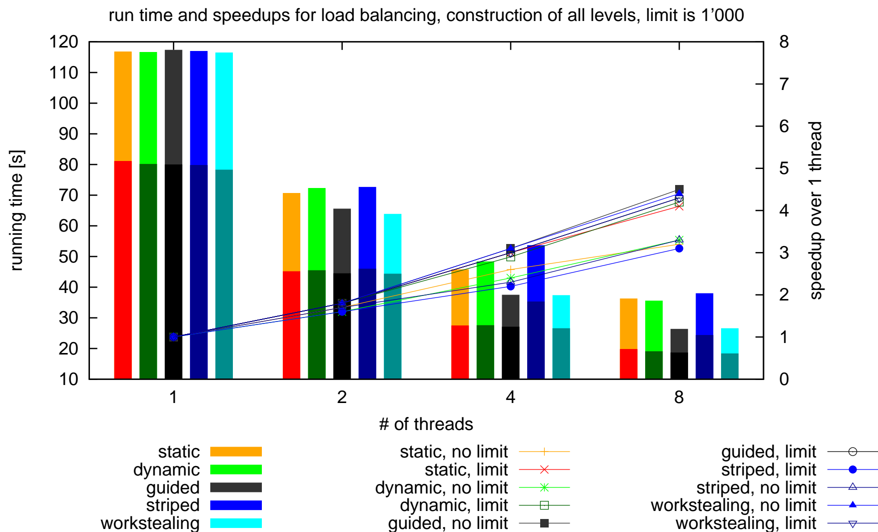


Experimental Results (4)

- load balancing problems in reduction step (long-ferry connections)
- use more sophisticated load balancing



Overall Results



Investigating Memory Bandwidth Limitation

- suboptimal speedup with 8 threads despite sophisticated load balancing
- consider memory bandwidth limitations
- run indepedenten instances of the program at the same time, each has a copy of all data structures of its own

description	longest running time (s)
one thread	109.3
2 threads, different processors	115.3
2 threads, 1 socket, 2 L2 caches	118.4
2 threads, 1 socket, 1 L2 cache	119.9
4 threads, different L2 caches	133.8
7 threads	173.0

Future Work

On Functionality

- allow updating edge weights *dynamically*

On Parallelization

- experiments on NUMA machine, adjustment to non-uniform memory access costs
- adaption on distributed shared memory (DSM) systems
- implementation using message passing
- evaluate combination of work stealing and “striped” load balancing

The End

References

- See paper.

Thank You

- Any questions?