Massively Parallel Graph Algorithms with MapReduce

Alexander Reinefeld, Thorsten Schütt, Robert Maier

Zuse Institute Berlin

AEI Cluster Day, 5 April 2011
Zuse Institute Berlin

Research institute for applied mathematics and computer science

• Peter Deuflhard
  chair for scientific computing, FU Berlin

• Martin Grötschel
  chair for discrete mathematics, TU Berlin

• Alexander Reinefeld
  chair for computer science, HU Berlin
HPC Systems @ ZIB

1984
Cray 1M
160 MFlops

1987
Cray X-MP
471 MFlops

1994
Cray T3D
38 GFlops

1997-03
Cray T3E
486 GFlops

2002-08
IBM p690
2,5 TFlops

2008/09
SGI ICE, XE
150 TFlops

1.000.000-fold performance increase in 25 years
How to solve very large graph problems

Problems,
• that cannot be solved on a single computer,
• that do not fit into main memory,
• that require supercomputers or large clusters
We present two algorithms

1. Breadth-first search with MapReduce
   • for exploring complete graphs
   • for computing heuristics

2. (Heuristic) shortest-path search with MapReduce
   • for solving single problem instances
Part I

BREADTH-FIRST SEARCH
Breadth-First Search (BFS)

Duplicate detection with a Closed list
Textbook BFS Algorithm

BFS() {
    Closed = {}; 
    Open = start_node; 
    while (Open ≠ empty) {
        current = pop(Open); 
        if (current ∉ Closed) {
            Closed = Closed ∪ current; 
            foreach (node ∈ succs(current)) 
                if(node ∉ Closed) 
                    Open = Open ∪ node; 
        } 
    } 
}
Let’s increase the problem size a little bit ...

The **15-puzzle** has

\[\text{16!/2} = 10,461,394,944,000\] states

Can’t be solved with BFS

- Closed: \(10^{13}\) (~80 TB)
- Open: \(~10^9\) (~5 TB)
Need parallel computers and memory

• Data decomposition

• How to avoid cycles?
  • Communicate between processes

• MPI implementation is tedious and error prone.
We’re lazy programmers

• We use **MapReduce** [J. Dean, S. Ghemawat, OSDI 2004]
  • operates on key/value pairs
  • just two functions
    o map(key, value)
    o reduce(key, value)
  • MapReduce runtime system orchestrates parallel execution, global data shuffling, communication, synchronization, fault tolerance

• **In graphs:**
  • key = node
  • value = vertex (or vertices)
BFS with MapReduce

Process 0,

bucket of nodes

Process 1, ...

MAP
generate successors

SHUFFLE
sort nodes (global op.)

REDUCE
remove duplicates
BFS with MapReduce

void mapper(Position position, set usedMoves) {
    foreach((successor, move) ∈ successors(position))
        if(inverse(move) ⊈ usedMoves)
            emit(successor, move);
}

void reducer(Position position, set<set> usedMoves) {
    moves = 0;
    foreach(move ∈ usedMoves)
        moves = moves ∪ move;
    emit(position, moves);
}

main() {
    front = {(start_node, 0)};
    while (front ≠ 0) {
        intermediate = map(front, mapper());
        front = reduce(intermediate, reducer());
    }
}
Results on the 15-puzzle

66 hours runtime,
32 nodes SGI Altix XE250 (2 Harpertown, 64 GB)

128 threads
Problem is I/O bound
Part II

HEURISTIC SEARCH
(SHORTEST PATH SEARCH)
Heuristic Search

- shortest path
- cost-optimal path
- A*
- ...
Heuristic Search

Expand all nodes $n$ in increasing order

$$f(n) = g(n) + h(n)$$

- cost from start to $n$
- estimated cost from $n$ to a goal

$h$ must be non-overestimating
Heuristic Search

Better heuristics prune more nodes

We perform a repeated search with iteratively increased $f(n)$

$\rightarrow$ BF-IDA*
BF-IDA* with MapReduce

```java
void mapper(Position position, set usedMoves) {
    foreach((successor, move) ∈ successors(position))
        if(inverse(move) ∈ usedMoves)
            if (g+1+h(successor) ≤ thresh)  // cutoff
                emit(successor, move);
}

void reducer(Position position, set<set> usedMoves) {
    moves = 0;
    foreach(move ∈ usedMoves)
        moves = moves ∪ move;
    if (position == goal) solved = true;  // found solution?
    emit(position, moves);
}

main() {
    thresh = h(start); solved = false;  // iterative deepening
    while (!solved) {
        front = {(start, 0)}; g=0;
        while (!solved && front ≠ 0) {
            intermediate = map(front, mapper());
            front = reduce(intermediate, reducer());
            g++;
        }
        thresh++;
    }  // iterative deepening
}
```
How to solve really BIG problems

Step 1: Build a heuristic with BFS

- A heuristic is a table of estimates.
- A heuristic solves a relaxed problem.

- Example:

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  heuristic space

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  2 steps

don’t cares

Step 2: Use the heuristic with BF-IDA*

- Example:

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>8</td>
</tr>
</tbody>
</table>

  problem space

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

  2 steps

  2 is a lower bound
15-puzzle is too small for us

• We built heuristics for the 24-puzzle with BFS
  • These are the largest tables up to now (8-8-8-heuristic, 120 GB)

• We solved random 24-puzzles in a few minutes with BF-IDA*
  • On 4096 cores MPI_AllToAllv () is the bottleneck

```
  1  2  3
  4  5  6  7
  8  9 10 11
12 13 14 15

  1  2  3  4
  5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24

10^5 states  10^{13} states  10^{24} states
```
Speedup

![Graph showing speedup and ideal speedup with increasing number of cores](image)
4093 cores

~750 MRs in 18 min.

500 MRs in 55s
Execution Profile

![Diagram showing execution profile with time percentages for different core counts: 1, 7, 31, 61, 127, 253, 511, 2039, 4093. The categories include Misc., LB after Reduce, Reduce, Sort, Shuffle, LB after Map, Map.](image-url)
Summary (1)

- Two scalable search algorithms:
  - breadth-first search to build heuristics
  - heuristic search to solve problems

- Three implementations:
  - OpenMP, MPI, hybrid

- Many applications
  - bioinformatics
  - combinatorial optimization
  - navigation
  - machine learning
  - n-puzzle (worst application because of cheap operator cost!)
Summary (2)

- Writing scalable code is not easy!

![Graph showing the performance of Zero byte MPI_Alltoallv on IBM BlueGene/P](image)

Balaji et al. (2009): MPI on a Million Processors.
More Information

