Design and implementation of efficient tools for parallel partitioning and distribution of very large numerical problems

Cédric Chevalier

ScALAPLIX project
LaBRI and INRIA Futurs
Université Bordeaux I
351, cours de la Libération, 33405 TALENCE, FRANCE

PhD Defense, 28 Sep 2007
Numerical simulations are essential in various domains including:
- mechanics
- fluid dynamics
- computational chemistry, ...

Problems are more and more elaborate: numerical simulation cannot be run on a single workstation
→ Need of parallelized computations
→ Need to know how to distribute data across the processors: domain decomposition
Work load distribution can be modeled as a graph partitioning problem and is mandatory in:

- domain decomposition
- sparse linear algebra problems
- VLSI circuit design
- bio-informatics
- computer imaging
- scheduling and load balancing
- …
Graph Partitioning

Process which consists in dividing vertices of a graph into a given number of sets, while enforcing two typical constraints:

1. **Boundary constraint**: the size of the interface between parts should be as small as possible

2. **Balance constraint**: all sets should be evenly weighted

Additional constraints can be considered, such as:

- Connectivity of parts, especially in VLSI design
- Compactness of parts (aspect ratio), in domain decomposition
  
  ... 

**Difficulty**

Problem is **NP-Complete** [Garey, Johnson and Stockmeyer, 1976]  
⇒ We have only heuristics
Our tool **Scotch**?

**Scotch** is a graph partitioning, sparse matrix ordering and static mapping tool developed in team **SCALAPPLIX**.

**Scotch 5.0** software package provides:

1. A sequential library (**Scotch**) which contains:
   - A static mapper
   - A graph and mesh (hypergraph) partitioner
   - A sparse matrix ordering tool

2. A parallel library (**PT-Scotch**) which currently only does sparse matrix ordering: based on **MPI** and optionally **pthread**

→ object of my thesis
Our tool **Scotch**?

Scotch is a graph partitioning, sparse matrix ordering and static mapping tool developed in team SCALAPPLIX.

Scotch 5.0 software package provides:

1. A sequential library (Scotch) which contains:
   - A static mapper
   - A graph and mesh (⇒ hypergraph) partitioner
   - A sparse matrix ordering tool

2. A parallel library (PT-Scotch) which currently only does sparse matrix ordering: based on MPI and optionally *pthread*
   → object of my thesis
Graph Partitioning Problem

Goals of PT-Scotch

- Develop fully parallel graph partitioning and reordering tools, by transposing all features of Scotch in parallel
- Provide the same quality as the best sequential tools
- Be scalable in time and memory
  - Efficiency on large architectures
  - Ability to handle graphs of about a billion vertices distributed on a thousand of processors
Graph Partitioning Problem

My Work

Goal

Design and implement an efficient parallel sparse matrix ordering tool

We decided to begin with parallel sparse matrix ordering:

- Uses graph bisection, which is simpler than k-way multisection
- No dependencies between bisections as in the static mapping problem
- The only other available tool, PARMETIS [Karypis and Kumar], provides very poor quality results
When solving sparse linear systems with direct methods, non-zero terms are created during the factorization process ($A \rightarrow LL^t$, $A \rightarrow LDL^t$ or $A \rightarrow LU$).

- Fill-in depends on the order of the unknowns
  $\implies$ Need to provide fill-reducing orderings

- We do graph ordering in SCOTCH by means of Nested Dissection using a Multi-Level technique

- Metric of ordering quality: OPC, that is, the Operation Count of Cholesky factorization (overall number of additions, subtractions, multiplications and divisions)
  $\implies$ Indirect measurement of separator quality
Sparse matrix ordering

- When solving sparse linear systems with direct methods, non-zero terms are created during the factorization process \((A \rightarrow LL^t, A \rightarrow LDL^t \text{ or } A \rightarrow LU)\)
- Fill-in depends on the order of the unknowns
  - Need to provide fill-reducing orderings
- We do graph ordering in SCOTCH by means of Nested Dissection using a Multi-Level technique
- Metric of ordering quality: OPC, that is, the OPeration Count of Cholesky factorization (overall number of additions, subtractions, multiplications and divisions)
  - Indirect measurement of separator quality
Sparse matrix ordering

- When solving sparse linear systems with direct methods, non-zero terms are created during the factorization process ($A \rightarrow LL^t$, $A \rightarrow LDL^t$ or $A \rightarrow LU$).
- Fill-in depends on the order of the unknowns
  $\Rightarrow$ Need to provide fill-reducing orderings

- We do graph ordering in SCOTCH by means of Nested Dissection using a Multi-Level technique.
- Metric of ordering quality: OPC, that is, the OPeration Count of Cholesky factorization (overall number of additions, subtractions, multiplications and divisions)
  $\Rightarrow$ Indirect measurement of separator quality.
Sparse matrix ordering

- When solving sparse linear systems with direct methods, non-zero terms are created during the factorization process ($A \rightarrow LL^t$, $A \rightarrow LDL^t$ or $A \rightarrow LU$)
- Fill-in depends on the order of the unknowns
  $\Rightarrow$ Need to provide fill-reducing orderings
- We do graph ordering in SCOTCH by means of Nested Dissection using a Multi-Level technique
- Metric of ordering quality: OPC, that is, the OPeration Count of Cholesky factorization (overall number of additions, subtractions, multiplications and divisions)
  $\Rightarrow$ Indirect measurement of separator quality
Outline

1. Introduction
   - Applications and graph partitioning
   - Graph Partitioning Problem
   - Sparse matrix ordering

2. Algorithms for sparse matrix ordering
   - Nested dissection
   - The Multi-Level framework
   - Parallelization steps

3. Parallelization of the coarsening phase
   - Principle
   - Synchronisation algorithms

4. Folding and duplication

5. Parallelization of the uncoarsening phase
   - Pre-constrained banding
   - Banded Fiduccia-Mattheyses
   - Banded Genetic Algorithms

6. Experiments

7. Conclusion
Sparse matrix ordering by Nested Dissection

- **Principle (George 1973)**
  - Find a vertex separator in the adjacency graph
  - Number separator vertices with the highest available indices
  - Apply recursively to both separated subgraphs

- **Interests**
  - Induces high quality block decompositions
  - Increases the concurrency of computations
  - Very suitable for parallel factorization (PASTIX solver)
A classical approach to improve partition quality [Bui and Jones, 1993]

Three steps

1. **Coarsening phase**
2. Initial partitioning
3. Uncoarsening phase

Decrease the number of vertices by merging pairs of neighbor vertices

⇒ At every step, obtain a smaller graph with the same topology
A classical approach to improve partition quality [Bui and Jones, 1993]

Three steps

1. Coarsening phase
2. Initial partitioning
3. Uncoarsening phase

Apply a global heuristic to compute a partition of the smallest graph

Typically, the size of coarsest graphs is about 100 vertices. Therefore, good initial partitions can be computed at low cost.
The Multi-Level framework

A classical approach to improve partition quality [Bui and Jones, 1993]

Three steps

1. Coarsening phase
2. Initial partitioning
3. Uncoarsening phase

Project the computed partition from the coarsest graph to finer graphs
Locally refine projected partitions with heuristics such as Kernighan-Lin or Fiduccia-Mattheyses (F.M.)
The Multi-Level framework

Inital partitioning

Coarsening

Initial partitioning

Uncoarsening

Uncoarsening
The Multi-Level framework

Sequential **Scotch**

First dissection level:
Original graph

Resulting matrix pattern:
Sequential **Scotch**

First dissection level: Coarsening phase

Resulting matrix pattern:
Sequential SCOTCH

First dissection level:
Coarsening phase

Resulting matrix pattern:
Sequential **SCOTCH**

First dissection level:
Coarsening phase

Resulting matrix pattern:
Sequential **SCOTCH**

First dissection level:
Coarsening phase

Resulting matrix pattern:
Sequential SCOTCH

First dissection level:
Initial partitioning

Resulting matrix pattern:
The Multi-Level framework

Sequential **Scotch**

First dissection level: Uncoarsening phase

Resulting matrix pattern:
Sequential **SCOTCH**

First dissection level: Uncoarsening phase

Resulting matrix pattern:
The Multi-Level framework

Sequential SCOTCH

First dissection level:
Uncoarsening phase

Resulting matrix
pattern:
The Multi-Level framework

Sequential **Scotch**

First dissection level: Uncoarsening phase

Resulting matrix pattern:
The Multi-Level framework

Sequential **SCOTCH**

First dissection level:
Uncoarsening phase

Resulting matrix pattern:
Sequential **Scotch**

Second dissection level: Uncoarsening phase

Resulting matrix pattern:
The Multi-Level framework

Sequential **Scotch**

Second dissection level: Uncoarsening phase

Resulting matrix pattern:
Sequential **Scotch**

Second dissection level:
Uncoarsening phase

Resulting matrix pattern:
Second dissection level:
Bisected graph

Resulting matrix pattern:
Sequential **Scotch**

Second dissection level:
Two subgraphs

Resulting matrix pattern:
The Multi-Level framework

Sequential SCOTCH

Second dissection level:
Processing first subgraph

Resulting matrix pattern:
Sequential **SCOTCH**

Second dissection level:
Processing first subgraph

Resulting matrix pattern:
Sequential \textsc{Scotch}

Second dissection level:
Processing second subgraph

Resulting matrix pattern:
The Multi-Level framework

Sequential **SCOTCH**

Second dissection level:
Processing second subgraph

Resulting matrix pattern:
Parallelism

3 levels of parallelism:

- nested dissection process, all subgraphs of the same level can be computed in parallel
- multi-level, computed on a distributed graph
- refinement on distributed graph

Need a distributed data structure that provides:

- Scalability
- Distribution of vertices
- Distribution of adjacency lists
Distributed graph in PT-Scotch

Distributed graph

Each process has:
- its own subgraph (local numbering)
- halo: neighbor vertices ("ghosts"), for overlapping communications

No adjacency data for ghost vertices
⇒ at least scalable in number of edges
Parallelization steps

Parallelization of Nested Dissection

- Straightforward, coarse-grain parallelism
- All subgraphs at the same dissection level are computed concurrently on separate groups of processors
- After a separator is computed, the two separated subgraphs are folded, that is, redistributed, on two subsets of the available processors
  - Can fold on any number of processors (not only powers of two)
  - Better data locality
  - The two subtrees are separated not only logically but also physically, which helps reducing network congestion
Outline

1. Introduction
   - Applications and graph partitioning
   - Graph Partitioning Problem
   - Sparse matrix ordering

2. Algorithms for sparse matrix ordering
   - Nested dissection
   - The Multi-Level framework
   - Parallelization steps

3. Parallelization of the coarsening phase
   - Principle
   - Synchronisation algorithms

4. Folding and duplication

5. Parallelization of the uncoarsening phase
   - Pre-constrained banding
   - Banded Fiduccia-Mattheyses
   - Banded Genetic Algorithms

6. Experiments

7. Conclusion
Coarsening phase

Principle:

1. Matching phase: make pairing between neighbor vertices
2. Construction phase: merge mated vertices into a new one and update the corresponding edges

We have to parallelize these two steps

Difficulties:

- Matching choices have to be fair to preserve graph topology when coarsen
  - Randomness in mating is important, even when using Heavy Edge Matching
- Coarsened graph has to be well distributed among processes
Matching algorithm

Matching is performed using a parallel multi-phase synchronous algorithm:

1. For each unmatched vertex $u$, randomly select an unmatched neighbor vertex $v$ to match with.
   - If $v$ is local, mark $(u, v)$ as mated.
   - Else, mark $(u, v)$ as selected.
2. Synchronise the selected vertices between processes.
3. While coarsening ratio is not small enough and some vertices can be matched, go to step 1.

Synchronisation algorithm is fundamental for keeping a good coarsening quality:

- Acceptance rate of algorithm has to be high.
- Accepting “bias” has to be low.
Algorithm

1. Communicate matching wishes to all processes
2. Analyze wishes for each process selected randomly

Authorized pairing

- The wanted vertex is free
- The wanted vertex wishes to be mated with the requesting vertex
Synchronisation algorithms

Synchronisation algorithm P

Algorithm

1. Communicate matching wishes to all processes
2. Analyze wishes for each process selected randomly

Authorized pairing

- The wanted vertex is free
- The wanted vertex wishes to be mated with the requesting vertex
Synchronisation algorithm P

Algorithm

1. Communicate matching wishes to all processes
2. Analyze wishes for each process selected randomly

Authorized pairing

- The wanted vertex is free
- The wanted vertex wishes to be mated with the requesting vertex
Synchronisation algorithms

Synchronisation algorithm L

Need a coloring of neighborhood processes graph: we use Luby coloring

Algorithm

For each color $c$ of processes
- If local process is colored by $c$
  - Send matching wishes to all neighbors
- If local process is not colored by $c$
  - Analyze wishes for each neighbor process colored by $c$ selected randomly

Authorized pairing

- The wanted vertex is free
- The wanted vertex wishes to be mated with a vertex owned by the same process as the requesting vertex
Synchronisation algorithms

Synchronisation algorithm L

Need a coloring of neighborhood processes graph: we use Luby coloring

Algorithm

For each color $c$ of processes
- If local process is colored by $c$
  - Send matching wishes to all neighbors
- If local process is not colored by $c$
  - Analyze wishes for each neighbor process colored by $c$ selected randomly

Authorized pairing
- The wanted vertex is free
- The wanted vertex wishes to be mated with a vertex owned by the same process as the requesting vertex
Synchronisation algorithm L

Need a coloring of neighborhood processes graph: we use Luby coloring

Algorithm

For each color \( c \) of processes
- If local process is colored by \( c \)
  - Send matching wishes to all neighbors
- If local process is not colored by \( c \)
  - Analyze wishes for each neighbor process colored by \( c \) selected randomly

Authorized pairing
- The wanted vertex is free
- The wanted vertex wishes to be mated with a vertex owned by the same process as the requesting vertex
Synchronisation algorithm L

Need a coloring of neighborhood processes graph: we use Luby coloring

Algorithm

For each color $c$ of processes
- If local process is colored by $c$
  - Send matching wishes to all neighbors
- If local process is not colored by $c$
  - Analyze wishes for each neighbor process colored by $c$ selected randomly

Authorized pairing
- The wanted vertex is free
- The wanted vertex wishes to be mated with a vertex owned by the same process as the requesting vertex
Comparison between synchronisation algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>8 processes</th>
<th>OPC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\rho$</td>
<td>$t$</td>
</tr>
<tr>
<td>Originally numbered audikw1 graph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>0.55</td>
<td>81.22</td>
</tr>
<tr>
<td>L</td>
<td>0.51</td>
<td>55.48</td>
</tr>
</tbody>
</table>

Conclusions:
- Contraction ratio higher with L
- With algorithm L, quality of matching is insensitive to the initial graph distribution

But:
- Very low acceptance rate with algorithm P when graph is badly distributed
- One iteration of algorithm P is potentially less expensive than one of L
- Algorithm L is serialized when process graph is complete
Synchronisation algorithms

Comparison between synchronisation algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>8 processes</th>
<th></th>
<th></th>
<th>16 processes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ρ</td>
<td>t</td>
<td>OPC</td>
<td>ρ</td>
<td>t</td>
</tr>
<tr>
<td><strong>Originally numbered audikw1 graph</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>0.55</td>
<td>81.22</td>
<td>7.06e+12</td>
<td>0.56</td>
<td>67.72</td>
</tr>
<tr>
<td>L</td>
<td>0.51</td>
<td>55.48</td>
<td>5.77e+12</td>
<td>0.51</td>
<td>52.29</td>
</tr>
</tbody>
</table>

Conclusions:
- Contraction ratio higher with L
- With algorithm L, quality of matching is insensitive to the initial graph distribution

But:
- Very low acceptance rate with algorithm P when graph is badly distributed
- One iteration of algorithm P is potentially less expensive than one of L
- Algorithm L is serialized when process graph is complete
Synchronisation algorithms

Comparison between synchronisation algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>8 processes</th>
<th>16 processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\rho$</td>
<td>$t$</td>
</tr>
<tr>
<td>Originally numbered audikwl graph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>0.55</td>
<td>81.22</td>
</tr>
<tr>
<td>L</td>
<td>0.51</td>
<td>55.48</td>
</tr>
<tr>
<td>Randomly numbered audikwl graph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>0.77</td>
<td>97.67</td>
</tr>
<tr>
<td>L</td>
<td>0.53</td>
<td>86.26</td>
</tr>
</tbody>
</table>

Conclusions:

- Contraction ratio higher with L
- With algorithm L, quality of matching is insensitive to the initial graph distribution

But:

- Very low acceptance rate with algorithm P when graph is badly distributed
- One iteration of algorithm P is potentially less expensive than one of L
- Algorithm L is serialized when process graph is complete
Using probabilities to improve synchronisation

Idea

Favor remote requests which have the highest probability to be accepted
Using probabilities to improve synchronisation

\[ P(u) = \frac{1}{3}, \quad P(v) = \frac{6}{7}, \quad P(w) = \frac{1}{3} \]

\( u \) and \( w \) have more chance to be remotely matched

**Idea**

Favor remote requests which have the highest probability to be accepted

In practice, we treat the vertices with a probability proportional to their local degree
⇒ highly connected boundary vertices are more often free for remote matching
New matching algorithm

- **Step 1:** For each unmatched vertex $u$
  - Ignore $u$ with probability $1 - P(u)$ and go to next vertex
  - Randomly select an unmatched neighbor vertex $v$ to match with
  - If $v$ is local, mark $(u, v)$ as mated
  - Else, mark $(u, v)$ as selected

- **Step 2:** synchronise the selected vertices between processes

- **Step 3:** While coarsening ratio is not small enough and some vertices can be matched, go to step 1
### Synchronisation algorithms

#### Impact of selection

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>8 processes</th>
<th>16 processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ρ</td>
<td>t</td>
</tr>
<tr>
<td>Originally numbered audikw1 graph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>0.55</td>
<td>81.22</td>
</tr>
<tr>
<td>SP</td>
<td>0.52</td>
<td>52.89</td>
</tr>
</tbody>
</table>

- Significantly speeds-up algorithm P
- Improves matching quality by increasing remote matching rate
### Impact of selection

| Algorithm | 8 processes | | 16 processes | |
|-----------|-------------|-------------|-------------|
|           | $\rho$ | $t$ | OPC | $\rho$ | $t$ | OPC |
| Originally numbered \textit{audikw1} graph |
| P   | 0.55 | 81.22 | 7.06e+12 | 0.56 | 67.72 | 6.75e+12 |
| SP  | 0.52 | 52.89 | 5.78e+12 | 0.52 | 48.34 | 7.33e+12 |
| Randomly numbered \textit{audikw1} graph |
| P   | 0.77 | 97.67 | 6.11e+12 | 0.90 | 74.29 | 6.71e+12 |
| SP  | 0.65 | 82.18 | 5.67e+12 | 0.83 | 66.79 | 6.66e+12 |

- Significantly speeds-up algorithm P
- Improves matching quality by increasing remote matching rate
Synchronisation algorithms

Coupling SP and L

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>8 processes</th>
<th>16 processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\rho$</td>
<td>$\tau$</td>
</tr>
<tr>
<td>L</td>
<td>0.51</td>
<td>55.48</td>
</tr>
<tr>
<td>SP+L</td>
<td>0.51</td>
<td>55.32</td>
</tr>
</tbody>
</table>

- Improves matching quality
- Improves matching speed
Synchronisation algorithms

Coupling SP and L

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>8 processes</th>
<th>16 processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ρ</td>
<td>t</td>
</tr>
<tr>
<td>Originally numbered audikw1 graph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>0.51</td>
<td>55.48</td>
</tr>
<tr>
<td>SP+L</td>
<td>0.51</td>
<td>55.32</td>
</tr>
<tr>
<td>Randomly numbered audikw1 graph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>0.53</td>
<td>86.26</td>
</tr>
<tr>
<td>SP+L</td>
<td>0.53</td>
<td>78.99</td>
</tr>
</tbody>
</table>

- Improves matching quality
- Improves matching speed
Coarsening algorithms

Coarsening summary

- Randomized tie-breaking at first when many neighbors are available (SP), then deterministic process (L) to complete the matching.
- Insensitive, in term of quality, to the initial graph distribution.
- Designed to balance coarsened vertices evenly across processors: at the end of the matching phase, all exchanges between each pair of processes are balanced.
- Local pairs are numbered before remote ones: computation/communication overlap during coarser graph construction.

![Diagram showing coarsening process with labeled processors P0 and P1.](image-url)
Coarsening summary

- Randomized tie-breaking at first when many neighbors are available (SP), then deterministic process (L) to complete the matching
- Insensitive, in term of quality, to the initial graph distribution
- Designed to balance coarsened vertices evenly across processors: at the end of the matching phase, all exchanges between each pair of processes are balanced
- Local pairs are numbered before remote ones: computation/communication overlap during coarser graph construction
Coarsening summary

- Randomized tie-breaking at first when many neighbors are available (SP), then deterministic process (L) to complete the matching
- Insensitive, in term of quality, to the initial graph distribution
- Designed to balance coarsened vertices evenly across processors: at the end of the matching phase, all exchanges between each pair of processes are balanced
- Local pairs are numbered before remote ones: computation/communication overlap during coarser graph construction
Outline

1. Introduction
   - Applications and graph partitioning
   - Graph Partitioning Problem
   - Sparse matrix ordering

2. Algorithms for sparse matrix ordering
   - Nested dissection
   - The Multi-Level framework
   - Parallelization steps

3. Parallelization of the coarsening phase
   - Principle
   - Synchronisation algorithms

4. Folding and duplication

5. Parallelization of the uncoarsening phase
   - Pre-constrained banding
   - Banded Fiduccia-Mattheyses
   - Banded Genetic Algorithms

6. Experiments

7. Conclusion
The fold-dup feature

Coarsened graphs may be folded and duplicated onto two halves of the processors to reduce communication and increase quality.
The fold-dup feature

Experiments show that folding-with-duplication is more efficient when performed on the coarsest graphs:

- Due to memory overcost, it cannot be applied to all levels
  - Every level requires $O(|V|)$ space when always duplicating
  - Without duplication, level $k$ requires only $O\left(\frac{|V|}{2^k}\right)$ space
- Improves performances in time by increasing data locality
- Improves quality by allowing multiple computations of initial partitions

Timing results on a 16-CPU/node SMP cluster for $\texttt{brgm}$ ($|V| = 3.7e+06$, $|E| = 1.51e+08$):

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>No F-D</td>
<td>260.31</td>
</tr>
<tr>
<td>F-D</td>
<td>276.91</td>
</tr>
</tbody>
</table>
Outline

1. Introduction
   - Applications and graph partitioning
   - Graph Partitioning Problem
   - Sparse matrix ordering

2. Algorithms for sparse matrix ordering
   - Nested dissection
   - The Multi-Level framework
   - Parallelization steps

3. Parallelization of the coarsening phase
   - Principle
   - Synchronisation algorithms

4. Folding and duplication

5. Parallelization of the uncoarsening phase
   - Pre-constrained banding
   - Banded Fiduccia-Mattheyses
   - Banded Genetic Algorithms

6. Experiments

7. Conclusion
The uncoarsening phase

Goals

1. Project partition found to coarser graph on the finer graph (*global* aspect)
2. Improve quality of projection by taking account of the finest definition of topology (*local* optimization)
   → separator displacement

- Projection step is easy to parallelize: only one change in parallel, we must select partition to project back when using folding and duplication
- Refinement algorithms are much harder to parallelize
Problems with refinement algorithms

- Local optimization algorithms, because of their local nature:
  - Are often trapped in local optima of their cost functions
  - Are intrinsically sequential and therefore do not parallelize well

- Global optimization algorithms are too expensive for large graphs

Need to define a framework within which:

- Sequential local optimization algorithms are still usable on large distributed graphs
- Global algorithms can be run inexpensively
Problems with refinement algorithms

- Local optimization algorithms, because of their local nature:
  - Are often trapped in local optima of their cost functions
  - Are intrinsically sequential and therefore do not parallelize well

- Global optimization algorithms are too expensive for large graphs

Need to define a framework within which:

- Sequential local optimization algorithms are still usable on large distributed graphs
- Global algorithms can be run inexpensively
Separator does not move very far during refinement

**Explanation**

Provided that the coarser graph is topologically close to its finer graph, the projection of the partition computed on the coarser graph is close to the one of the finer, and only further local improvements have to be carried out.

\[
\text{Niveau } \lambda - 1 : u \quad v \quad x \quad y \\
\text{Niveau } \lambda : \{u,v\} \quad \{x,y\}
\]
Introducing band graphs

Experimentally evidenced by looking at the distribution of finer separator vertices after local F.M. refinement

<table>
<thead>
<tr>
<th>Distance from coarser separator</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>≥ 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>80%</td>
<td>17%</td>
<td>1.5%</td>
<td>&lt; 0.5%</td>
<td>&lt; 0.1%</td>
</tr>
</tbody>
</table>

Idea [Chevalier and Pellegrini, 2006]

Only consider for refinement vertices that are likely to move

⇒ Extract a priori a band subgraph around the coarser separator and apply optimization heuristics only to it

⇒ Dramatic reduction in problem size:

for a $10^9$ vertices 3D graph, separator is about $10^6$ vertices
Using band graphs

Extracting band graphs allows us:

- To go on using classical sequential refinement algorithms, on smaller graphs
  \[\Longrightarrow\] Guarantees no loss of quality over existing sequential software
- To use parallel algorithms which are much too expensive to be run on large graphs
- To use new approaches which exploit the fact that the band constrains the refinement process
Very little overcost in time: $\sim 3\%$ of increase in time compared to plain F.M.

While the default bipartitioning strategy of SCOTCH 4.0 uses two runs of multi-level bisection and keeps the best, we can achieve equivalent quality with only one run of multi-level banded F.M.

$\implies$ Overall gain in execution time

$\implies$ No loss of quality but, on the contrary, an improvement

**Explanation**

Better accounting for the global topology of graphs by limiting the ability to be trapped in purely local optima whom may be coarsening artefacts
Multi-centralized banded Fiduccia-Mattheyses

1. Extract band graph in parallel
2. Each process owns his centralized band graph
3. Each process runs F.M. on his centralized band graph
4. The best partition is broadcasted to every process and kept to go to next level
Consequences:
- **PT-SCOTCH** exploits processors in a multi-sequential way for refinement
- Quality of refinement increases when number of processes increases
- Definitely not scalable but useful before scalable optimization algorithms are available
Banded Genetic Algorithms

Genetic Algorithms (G.A.) [Holland, 1970s]

- Meta-heuristics which can be applied to solve various optimization problems
- Easy to parallelize
- Known to be very expensive in time, as problem size increases
- An ideal candidate to operate on band graphs
Genetic Algorithms (G.A.) [Holland, 1970s]

- Meta-heuristics which can be applied to solve various optimization problems
- Easy to parallelize
- Known to be very expensive in time, as problem size increases
- An ideal candidate to operate on band graphs
To use G.A.s, we must be able to compare the quality of two partitions

Two criteria:

- Separator size (number of separator vertices)
- Separator shape (number of edges adjacent to separator vertices)
To use G.A.s, we must be able to compare the quality of two partitions

Two criteria:

- Separator size (number of separator vertices)
- Separator shape (number of edges adjacent to separator vertices)
Mono-chromosome individuals: an uni-dimensional array which associates a number between 0 and 2 to any subgraph vertex index.

- Reproduction operator is a classical two-points cross-over operator.
- Selection operator is a mix between roulette and elitism.
Our G.A. implementation

- Mono-chromosome individuals: an uni-dimensional array which associates a number between 0 and 2 to any subgraph vertex index
- Reproduction operator is a classical two-points cross-over operator
- Selection operator is a mix between roulette and elitism

A Two points crossing over:

\[
\begin{array}{c}
\text{Red} \\
\downarrow \\
\downarrow \\
\text{Green} \\
\uparrow \\
\uparrow \\
\text{Red}
\end{array}
\]

\[
\begin{array}{c}
\text{Green} \\
\downarrow \\
\downarrow \\
\text{Red} \\
\uparrow \\
\uparrow \\
\text{Green}
\end{array}
\]

\[
\begin{array}{c}
\text{Red} \\
\uparrow \\
\downarrow \\
\text{Green} \\
\text{Red}
\end{array}
\]

\[
\begin{array}{c}
\text{Green} \\
\uparrow \\
\downarrow \\
\text{Red} \\
\text{Green}
\end{array}
\]

\[
\begin{array}{c}
\text{Red} \quad + \\
\downarrow \quad \uparrow \\
\downarrow \quad \uparrow \\
\text{Green} \quad = \\
\text{Red}
\end{array}
\]

\[
\begin{array}{c}
\text{Green} \quad = \\
\uparrow \quad \downarrow \\
\uparrow \quad \downarrow \\
\text{Red} \quad + \\
\text{Green}
\end{array}
\]
Our G.A. implementation

- Mono-chromosome individuals: an uni-dimensional array which associates a number between 0 and 2 to any subgraph vertex index
- Reproduction operator is a classical two-points cross-over operator
- Selection operator is a mix between roulette and elitism

- Elitism: Best individuals are kept in the next generation
- Roulette: At each turn, two individuals are selected to be parents of the next generation. The probability to be chosen is proportional to the quality of the partition
Multi-deme model (Goldberg, 1997):

- Several isolated populations (islands or demes): one population per processor in our case.
- Occasionally, a few *champions* are allowed to migrate between demes: Very few inter-processor communications, thus model is easy to parallelize.

Increases the opportunity to converge to the global minimum of the partition cost function.
Banded Genetic Algorithms

Genetic Algorithms in parallel

- Multi-deme model (Goldberg, 1997):
  - Several isolated populations (islands or demes):
    one population per processor in our case
  - Occasionally, a few *champions* are allowed to migrate between demes:
    Very few inter-processor communications, thus model is easy to parallelize

- Increases the opportunity to converge to the global minimum of the partition cost function
Tests carried out with a threaded version of SCOTCH and a multi-deme algorithm without splitting chromosomes on several processors:

- Achieve good quality
- Are more expensive than banded F.M. but more parallelizable

⇒ We can extend this multi-sequential method in parallel without increasing too much the volume of communications, simply by splitting individual chromosomes across several processors
Outline

1. Introduction
   - Applications and graph partitioning
   - Graph Partitioning Problem
   - Sparse matrix ordering

2. Algorithms for sparse matrix ordering
   - Nested dissection
   - The Multi-Level framework
   - Parallelization steps

3. Parallelization of the coarsening phase
   - Principle
   - Synchronisation algorithms

4. Folding and duplication

5. Parallelization of the uncoarsening phase
   - Pre-constrained banding
   - Banded Fiduccia-Mattheyses
   - Banded Genetic Algorithms

6. Experiments

7. Conclusion
**Experimental results (1)**

Our largest test graphs to date:

<table>
<thead>
<tr>
<th>Graph</th>
<th>Size ($\times 10^3$)</th>
<th>$\bar{\delta}$</th>
<th>OPC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>V</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>audikw1</td>
<td>944</td>
<td>38354</td>
<td>81.28</td>
<td>5.48e+12</td>
</tr>
<tr>
<td>coupole8000</td>
<td>1768</td>
<td>41657</td>
<td>47.12</td>
<td>7.66e+12</td>
</tr>
<tr>
<td>brgm</td>
<td>3699</td>
<td>151940</td>
<td>82.14</td>
<td>2.70e+13</td>
</tr>
<tr>
<td>cage15</td>
<td>5154</td>
<td>47022</td>
<td>18.24</td>
<td>4.06e+16</td>
</tr>
<tr>
<td>qimonda07</td>
<td>8613</td>
<td>29143</td>
<td>6.76</td>
<td>8.92e+10</td>
</tr>
<tr>
<td>23millions</td>
<td>23114</td>
<td>175686</td>
<td>7.6</td>
<td>1.29e+14</td>
</tr>
</tbody>
</table>
Results on audikw1 (|V| = 0.94e+6, |E| = 38e+6)
Results on coupole8000 ($|V| = 1.76 \times 10^6$, $|E| = 41 \times 10^6$)
Results with multi-sequential band FM

<table>
<thead>
<tr>
<th>Test case</th>
<th>Number of processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>qimonda07 (seq : 8.92e+10)</td>
<td>–</td>
</tr>
<tr>
<td>OPC t(s)</td>
<td>–</td>
</tr>
<tr>
<td>brgm (seq : 2.70+13)</td>
<td>2.70e13</td>
</tr>
<tr>
<td>OPC t(s)</td>
<td>276.90</td>
</tr>
<tr>
<td>23millions (seq : 1.29e+14)</td>
<td>1.45e14</td>
</tr>
<tr>
<td>OPC t(s)</td>
<td>671.60</td>
</tr>
</tbody>
</table>

For all of these graphs, PARMETIS crashes.
Runs done on a Power5 SMP computer, with 8 dual core CPU per node.
Outline

1. Introduction
   - Applications and graph partitioning
   - Graph Partitioning Problem
   - Sparse matrix ordering
2. Algorithms for sparse matrix ordering
   - Nested dissection
   - The Multi-Level framework
   - Parallelization steps
3. Parallelization of the coarsening phase
   - Principle
   - Synchronisation algorithms
4. Folding and duplication
5. Parallelization of the uncoarsening phase
   - Pre-constrained banding
   - Banded Fiduccia-Mattheyses
   - Banded Genetic Algorithms
6. Experiments
7. Conclusion
Conclusion

- We have implemented the parallel ordering feature of PT-Scotch
- We can achieve the same quality than sequential tools and still remain scalable (save for multi-sequential refinement)

To do this we have developed and implemented several new algorithms which can be reused in the context of graph partitioning:
- Pre-constrained banding, which has also improved the sequential tool
- Multi-level parallelization
Conclusion (2)

For the sparse matrix reordering tool:

- The time taken by our algorithm is about twice the one of PARMETIS, but seems to be scalable
  → Multi-sequential F.M. not scalable, visible on smaller graphs
- Quality is clearly better than the one of PARMETIS
  - Does not decrease on average when number of processors increases
  - Can even increase along with the number of processors
Conclusion

Work available as part of PT-SCOTCH in SCOTCH 5.0

Distributed on CeCiLL-C licence:
http://gforge.inria.fr/projects/scotch/
Perspectives

- Parallelize band refinement algorithms, for cases when band graphs will no longer fit in memory:
  - Fully parallel Genetic Algorithm which spreads individual chromosomes across processors
  - Parallel diffusion algorithm
- Basing on these existing parallel routines, design efficient and scalable k-way graph partitioning algorithms
Comparison between synchronisation algorithms (1)

Remote matching rates during the first loop of synchronisation algorithms:

Algorithm P:

Algorithm L:

→ Very poor performance of algorithm P when average degree increases
Moreover:

- for algorithm P, performance decreases when number of processes increases
- for algorithm L, remote match rate increases when number of processes increases
Refinement problem in parallel