Overview of Trilinos and PT-Scotch

Markus Hopfer, Karl-Franzens University, Graz

29.03.2012
PT-Scotch

1. Mapping and Graph Partitioning
   - The Dual Recursive Bipartitioning Algorithm
   - Parallel Graph Bipartitioning Methods
   - Parallel Sparse Matrix Ordering

2. The Trilinos Library
   - Overview of the Trilinos Packages
   - Examples on using Trilinos
Overview of Trilinos and PT-Scotch

PT-Scotch

Markus Hopfer, Karl-Franzens University, Graz

Parallel Sparse Matrix Ordering
The Scotch Project

- Laboratoire Bordelais de Recherche en Informatique (LaBRI), Université Bordeaux I / ScALApplix project of INRIA Bordeaux Sud-Ouest
- Applications of graph theory to scientific computing
  - Development of the Dual Recursive Bipartitioning Mapping Algorithm
  - Graph bipartitioning heuristics
  - Computation of high-quality vertex separators for the ordering of sparse matrices
  - Hypergraph partitioning algorithms
- PT-Scotch 5.1 ⇔ parallel graph ordering/partitioning features
- Parallel static mapping will be available in the next release
### Sparse Matrix Ordering

- scientific/engineering problems $\Leftrightarrow$ sparse linear systems $\leadsto$ direct or iterative methods to solve the systems
- direct solvers
  - minimize the fill-in by re-ordering of the linear system
  - parallel graph ordering tools $\Leftrightarrow$ **PT-Scotch**

### Static Mapping

- determine best start time for processes on host machine(s) $\Leftrightarrow$ **scheduling**
- determine best distribution of processes on host machine(s) $\Leftrightarrow$ **mapping**
  - balance computational weight of processes between processors
  - reducing the cost of communication
  - keep intensively intercommunicating processes on nearby processors
- computational structure of parallel program $\Leftrightarrow$ **graph**
  - vertices $\Leftrightarrow$ processes that handle distributed data
  - edges $\Leftrightarrow$ data dependencies/communication
- static mapping $\Leftrightarrow$ computed prior to execution of the program
Parallel Static Mapping by Dual Recursive Bipartitioning

**Static Mapping**

- we want to map the parallel program on a target architecture
- \( \Rightarrow \) **parallel program** is modelled by a **graph** \( S \) (source graph)
  - vertices \( v_S \) ↔ processes \( \mapsto \) vertex number \( \omega_S(v_S) \in \mathbb{Z} \)
  - edges \( e_S \) ↔ communication between processes \( \mapsto \) edge number \( \omega_S(e_S) \in \mathbb{Z} \)
  - \( \omega_S(v_S) \) ... estimation of computation weight for process
  - \( \omega_S(e_S) \) ... estimate amount of communication between processes
- \( \Rightarrow \) **target architecture** is modelled by a **graph** \( T \) (target graph)
  - vertices \( v_T \) ↔ processors \( \mapsto \) vertex number \( \omega_T(v_T) \in \mathbb{Z} \)
  - edges \( e_T \) ↔ communication abilities \( \mapsto \) edge number \( \omega_T(e_T) \in \mathbb{Z} \)
  - \( \omega_T(v_T) \) ... estimation of computational power of corresponding processor
  - \( \omega_T(e_T) \) ... estimate costs for communication between processors
- if target is homogeneous \( \Rightarrow \) all \( \omega_T(v_T) \) are equal
Mapping of the Source Graph $S$ to the Target Graph $T$

- **vertex mapping:** $\tau_{S,T} : V(S) \rightarrow V(T)$
- **edge mapping:** $\rho_{S,T} : E(S) \rightarrow \mathcal{P}(E(T))$
  - $\mathcal{P}(E(T))$... set of all simple loopless paths which can be build from $E(T)$
- **examples:**
  - $\tau_{S,T}(v_S) = v_T$... if process $v_S$ of $S$ is mapped onto processor $v_T$ of $T$
  - $\rho_{S,T}(e_S) = \{e_1^T, e_2^T, \ldots, e_n^T\}$... if communication channel $e_S$ of $S$ is routed through communication links $e_1^T, e_2^T, \ldots, e_n^T$ of $T$
- $|\rho_{T,S}(e_S)|$... dilation of edge $e_S$, i.e. the number of necessary edges to route $e_S$ on graph $T$
Cost Function and Performance Criteria

- communication cost function:

\[ f_c(\tau_S, \tau, \rho_S, \tau) \overset{\text{def}}{=} \sum_{e_S \in E(S)} \omega_S(e_S)|\rho_S, \tau(e_S)| \]

- goal: minimize cost function, while keeping good load balance

- average load per computational power unit:

\[ \mu_{\text{map}} \overset{\text{def}}{=} \frac{\sum_{v_S \in V(S)} \omega_S(v_S)}{\sum_{v_T \in V(T)} \omega_T(v_T)} \]

- load imbalance ratio:

\[ \delta_{\text{map}} \overset{\text{def}}{=} \frac{\sum_{v_T \in V(T)} \left| \left( \frac{1}{\omega_T(v_T)} \sum_{v_S \in V(S)} \omega_S(v_S) \right) - \mu_{\text{map}} \right|}{\sum_{v_S \in V(S)} \omega_S(v_S)} \]

- maximum load imbalance ratio \( \delta^{\text{max}}_{\text{map}} \) provided by user as input
Cost Function and Performance Criteria

- average edge expansion/dilation:
  \[ \mu_{\text{exp}} \overset{\text{def}}{=} \frac{f_C}{\sum_{e_S \in E(S)} \omega_S(e_S)} \quad \mu_{\text{dil}} \overset{\text{def}}{=} \frac{\sum_{e_S \in E(S)} |\rho_{S,T}(e_S)|}{|E(S)|} \]

- \[ \delta_{\text{exp}} \overset{\text{def}}{=} \frac{\mu_{\text{exp}}}{\mu_{\text{dil}}} < 1 \] if mapper succeeds in putting heavily intercommunicating processes closer together
The Dual Recursive Bipartitioning Algorithm

- recursively allocate subsets of processes to subsets of processors
  1. take all processors of target (domain) & take all processes to map
  2. bipartition a yet unprocessed domain into two disjoint subdomains
  3. call a graph bipartitioning algorithm to split the subset of processes associated with the domain across the two subdomains

- sketch of the algorithm:

```c
mapping (D, P);
Set_of_Processors D;
Set_of_Processes P;
{
    Set_of_Processors D0, D1;
    Set_of_Processes P0, P1;
    if (P == 0) return; /* If nothing to do. */
    if (D == 1) {
        result (D, P); /* P is mapped onto it. */
        return;
    }
    (D0, D1) = processor_bipartition (D);
    (P0, P1) = process_bipartition (P, D0, D1);
    mapping (D0, P0); /* Perform recursion. */
    mapping (D1, P1);
}
```
Partial Cost Function

- cost function for bipartitioning of a subgraph $S'$ of $S$

$$f_C'(\tau_S, \rho_S, \tau) \overset{\text{def}}{=} \sum_{\{v, v'\} \in E(S)} \omega_S(\{v, v'\}) |\rho_S, \tau(\{v, v'\})|$$

- $\{v, v'\}$ ... cocycles, i.e. connections between $S$ and $S'$
- taking into account results of previous bipartitionings to avoid local choices that might be globally disadvantageous
Parallel Graph Bipartitioning Methods

- **Dual Parallel Recursive Mapping Algorithm** uses graph bipartitioning methods as black boxes.
- Allows to run any type of graph bipartitioning method which is conformal with cost function:
  - Kernighan-Lin-type algorithms
  - simulated annealing
  - quadratic assignment
  - genetic algorithms
  - etc.
- Define **mapping strategies**:
  - Different methods for different domains
  - Apply different methods in sequence
- **Currently** implemented graph bipartitioning methods:
  - **Band method**: meta-algorithm which supports other partitioning algorithms to perform better
  - **Diffusion**: global optimization method
  - **Multi-Level**: in combination with banded diffusion method to refine the projected partitions at every level
Multi-level Framework for Graph Partitioning

Figure: Multi-level framework for computing a bipartition of a graph.
Figure: Partition of graph **bump** into 8 parts:
(upper) Scotch 4.0 with un-banded Fiduccia-Mattheyses refinement
(lower) Scotch 5.0 with multi-level banded diffusion scheme
The Jug(s) of the Danaides

- nodes ⇔ leaking barrels of \( \infty \) capacity
- graph edges ⇔ pipes of section equal to their weight
- two sorts of liquids (annihilate each other)

```plaintext
while (number of passes to do) {
  reset contents of new array to 0;
  old[s0] ← old[s0]−\(|V|\)/2;
  old[s1] ← old[s1]+\(|V|\)/2;
  for (all vertices v in graph) {
    c ← old[v];
    if (|c|>weight[v]) {
      c ← c−weight[v]*sign(c);
      σ ← \( \sum_{e=(v,v')}\)weight[e];
      for (all edges e=(v,v')) {
        f ← c*weight[e]/σ;
        new[v'] ← new[v'] + f;
      }
    }
  }
  swap old and new arrays;
}
```

/* Refill source barrels */

/* Get contents of barrel */
/* If not all contents have leaked */
/* Compute what will remain */
/* Sum weights of all adjacent edges */
/* For all edges adjacent to v */
/* Fraction to be spread to v' */
/* Accumulate spreaded contributions */
The Trilinos Software Package
<table>
<thead>
<tr>
<th>Trilinos Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Epetra</strong></td>
</tr>
<tr>
<td>basic classes for distributed matrices and vectors, linear operators and linear problems, time/FLOP measuring, etc.</td>
</tr>
<tr>
<td>each Trilinos package accepts Epetra objects</td>
</tr>
<tr>
<td><strong>Triutils</strong></td>
</tr>
<tr>
<td>collection of utilities for software development</td>
</tr>
<tr>
<td>command line parser, matrix generator</td>
</tr>
<tr>
<td><strong>AztecOO</strong></td>
</tr>
<tr>
<td>linear solver package based on preconditioned Krylov methods</td>
</tr>
<tr>
<td>object-oriented version of Aztec</td>
</tr>
<tr>
<td><strong>Belos</strong></td>
</tr>
<tr>
<td>provides next-generation iterative linear solvers and a powerful linear solver framework</td>
</tr>
<tr>
<td><strong>IFPACK</strong></td>
</tr>
<tr>
<td>performs various incomplete factorizations</td>
</tr>
<tr>
<td>used with AztecOO</td>
</tr>
<tr>
<td><strong>Isorropia and Zoltan</strong></td>
</tr>
<tr>
<td>partitioning and load balancing</td>
</tr>
<tr>
<td><strong>Teuchos</strong></td>
</tr>
<tr>
<td>classes for advanced code development</td>
</tr>
</tbody>
</table>
Trilinos Packages

- **ML**
  - algebraic multilevel and domain decomposition preconditioner package
  - scalable preconditioning capabilities for a variety of problems

- **Amesos**
  - provides common interface to certain sparse linear solvers (generally available outside the Trilinos framework)

- **Anazasi**
  - provides a common interface to parallel eigenvalue and eigenvector solvers for both symmetric and non-symmetric linear problems

- **NOX**
  - collection of non-linear solvers
  - designed to be easily integrated into an application or combined with linear solvers

- **Tpetra**
  - next-generation, templated version of Petra
  - taking advantage of the newer advanced features of C++

- **Didasko**
  - contains all the examples from the tutorial
Example: Epetra Objects

```c
#include "Epetra_ConfigDefs.h"
#ifdef HAVE_MPI
#include "mpi.h"
#include "Epetra_MpiComm.h"
#else
#include "Epetra_SerialComm.h"
#endif

// .. other include file and others ...

int main( int argc, char *argv []) {

// .. some declarations here ...

#ifdef HAVE_MPI
    MPI_Init (&argc, &argv);
    Epetra_MpiComm Comm(MPI_COMM_WORLD);
#else
    Epetra_SerialComm Comm;
#endif

// .. other code follows ...

#ifdef HAVE_MPI
    MPI_Finalize();
#endif

    return 0;
}
```
Example: Solving a Linear System $A x = b$ with AztecOO

- create an Epetra object for the linear system

  ```
  Epetra_LinearProblem problem(&A, &x, &b);
  ```

- create an AztecOO object

  ```
  AztecOO solver(Problem);
  ```

- define option & parameter vector to override default by desired values

  ```
  int options[AZ_OPTIONS_SIZE];
  double params[AZ_PARAMS_SIZE];
  AZ_defaults(options, params);

  solver.setAllAztecOptions(options);
  solver.setAllAztecParams(params);
  ```

- alternatively: set options explicitly → Jacobi preconditioner

  ```
  solver.setAztecOption(AZ_precond, AZ_Jacobi);
  ```

- solve the linear system with a maximum of 1550 iterations and a residual error norm threshold of $10^{-9}$

  ```
  solver.iterate(1550, 1E-9);
  ```
Example: Solving a Linear System $A x = b$ with AztecOO

- create an Epetra object for the linear system
  
  ```cpp
  Epetra_LinearProblem Problem(&A, &x, &b);
  ```

- create an AztecOO object
  
  ```cpp
  AztecOO Solver(Problem);
  ```

- define option & parameter vector to override default by desired values
  
  ```cpp
  int options[AZ_OPTIONS_SIZE];
  double params[AZ_PARAMS_SIZE];
  AZ_defaults(options, params);

  Solver.SetAllAztecOptions(options);
  Solver.SetAllAztecParams(params);
  ```

- alternatively: set options explicitly $\rightarrow$ Jacobi preconditioner
  
  ```cpp
  Solver.SetAztecOption(AZ_precond, AZ_Jacobi);
  ```

- solve the linear system with a maximum of 1550 iterations and a residual error norm threshold of $10^{-9}$
  
  ```cpp
  Solver.Iterate(1550, 1E-9);
  ```
Example: Solving a Linear System $A x = b$ with AztecOO

- create an Epetra object for the linear system
  
  ```cpp
  Epetra_LinearProblem Problem(&A, &x, &b);
  ```

- create an AztecOO object
  
  ```cpp
  AztecOO Solver(Problem);
  ```

- define option & parameter vector to override default by desired values
  
  ```cpp
  int options[AZ_OPTIONS_SIZE];
  double params[AZ_PARAMS_SIZE];
  AZ_defaults(options, params);

  Solver.SetAllAztecOptions(options);
  Solver.SetAllAztecParams(params);
  ```

- alternatively: set options explicitly → Jacobi preconditioner
  
  ```cpp
  Solver.SetAztecOption(AZ_precond, AZ_Jacobi);
  ```

- solve the linear system with a maximum of 1550 iterations and a residual error norm threshold of $10^{-9}$
  
  ```cpp
  Solver.Iterate(1550, 1E-9);
  ```
Example: Solving a Linear System $A x = b$ with AztecOO

- create an Epetra object for the linear system
  
  ```
  Epetra_LinearProblem Problem(&A, &x, &b);
  ```

- create an AztecOO object
  
  ```
  AztecOO Solver(Problem);
  ```

- define option & parameter vector to override default by desired values
  
  ```
  int options[AZ_OPTIONS_SIZE];
  double params[AZ_PARAMS_SIZE];
  AZ_defaults(options, params);

  Solver.SetAllAztecOptions(options);
  Solver.SetAllAztecParams(params);
  ```

- alternatively: set options explicitly → Jacobi preconditioner
  
  ```
  Solver.SetAztecOption(AZ_precond, AZ_Jacobi);
  ```

- solve the linear system with a maximum of 1550 iterations and a residual error norm threshold of $10^{-9}$
  
  ```
  Solver.Iterate(1550, 1E-9);
  ```
Example: Solving a Linear System $A x = b$ with AztecOO

1. create an Epetra object for the linear system
   
   ```
   Epetra_LinearProblem Problem(&A, &x, &b);
   ```

2. create an AztecOO object
   
   ```
   AztecOO Solver(Problem);
   ```

3. define option & parameter vector to override default by desired values
   
   ```
   int options[AZ_OPTIONS_SIZE];
   double params[AZ_PARAMS_SIZE];
   AZ_defaults(options, params);

   Solver.SetAllAztecOptions(options);
   Solver.SetAllAztecParams(params);
   ```

4. alternatively: set options explicitly → Jacobi preconditioner
   
   ```
   Solver.SetAztecOption(AZ_precond, AZ_Jacobi);
   ```

5. solve the linear system with a maximum of 1550 iterations and a residual error norm threshold of $10^{-9}$
   
   ```
   Solver.Iterate(1550, 1E-9);
   ```
Thank you for your attention!