Real-time Grid Computing:
Monte-Carlo Methods in Parallel Tree Searching

Hermann Heßling

6. 2. 2012
Outline

1. Real-time Computing
2. GriScha: Chess in the Grid - by Throwing the Dice
3. Parallel Tree Searching
4. Global Optimization - by Throwing the Dice
   4.1 Genetic Algorithms
5. Summary and Outlook
1. Real-time Computing

- Some characteristics
  - Computing system reacts within constraints on the response time
  - Deadlines must be fulfilled independently of the load of the system
  - Hard real-time systems: no latency tolerated
  - Soft real-time systems: some latency tolerated

- Preemptive, static-priority schedule
  - Priority assignment to all tasks of a system
  - Jobs of higher priority preempt jobs of lower priority

- Theorem [Eisenbrand, Rothvoß, 2008]
  - Given a preemptive, static-priority schedule. The response time computation is NP-hard.
  - Proof: real-time scheduling is linked to algorithmic number theory using integer relations.
2. Grischka

- Gatekeeper
  - allocates worker nodes in advance
  - submits a pilot job to worker nodes
- Worker node (WN)
  - runs a pilot job
  - the firewall protecting a WN must allow 1 outbound connection
- Pilot job
  - contains chess evaluation software (JAVA) - beginners level
  - initiates a TCP connection (using SIMON) to the MasterNode
  - receives requests from MasterNode (over the same TCP connection)
- MasterNode
  - calculates the first legal moves (3-4 ply)
  - distributes chess positions to the worker nodes
  - evaluates results from the worker nodes and selects a move
2. GríScha

Working Group
- Michael Kurth B.Sc. (2011 - )
- Marco Strutz M.Sc. (2009 - )
- HH

Former members
- Laurence Bortfeld B.Sc. (2011)
- Daniel Heim M.Sc. (2009 – 2011)
- Christoph Neumann M.Sc. (2009 - 2011)
- Francesco Tietke B.Sc. (2010)
2.1 Tree Searching

- Many optimization problems can be solved by tree searching
  - game tree: nodes = possible moves

- Minimax algorithm with alternate moves
  - determine moves up to a certain depth
  - initial step: assign a value to each leaf node
    - heuristic evaluation function
  - iteration step: assign value to parent nodes
    - choose optimal value from children nodes
  - final step: root node chooses “best” move
    - assumption: high value = high chance of winning
2.1 Monte-Carlo Tree Search (MCTS)

- MCTS method: randomized exploration of a game tree
  - Advantage: even applicable if a good evaluation function is not known

---

Exploitation
- concentrate on best moves

Exploration
- consider also bad moves

Moderate increase of the tree

... until end of game

2.2 Grischa: Monte-Carlo Engine

- Idea: exploring next moves by throwing the dice
  - **Preparation**
    - Determine the number $N$ of moves of root node
    - Assign a value $f(n)$ to each node $n$ using the evaluation function $f$
  - **Monte-Carlo (MC) selection**
    - **Probability measure** for each node $n$
      
      $p_n = \frac{f(n) - (f_{\text{min}} - f_*)}{\sum_{n=1}^{N} f(n) - N(f_{\text{min}} - f_*)}$
      
      $f_* = f_{\text{min}}$ if $f_{\text{min}} > 0$, otherwise $f_* = 1$
      - positive and normalized: $0 < p_n \leq 1$ $P_N = 1$ ($P_n := \sum_{i=1}^{n} p_i$, $P_0 := 0$)
    - **Random node selection.** Let $p$ be a random number from a uniform distribution over the unit interval. Select node $n$ if $P_{n-1} < p \leq P_n$
    - The node selection is similar to the MC **importance sampling** method
      - A “good” move $n$ ($\sim$ high value of $f(n)$) is explored quite likely
      - Even “worst” moves are selected with non-vanishing probability as $f_* > 0$
2.2 G.riScha: Monte-Carlo Engine

- **Simulation**
  - Repeat "Preparation" and "MC selection" until max. depth
    - previously selected node = root node
  - In case of "min-player" change signs:
    - replace $f_{\text{min}}$ by $f_{\text{max}}$
    - $f_*$: replace $f_{\text{min}} > 0$ by $f_{\text{max}} < 0$

- **Update**
  - Modify value of parent nodes according to minimax algorithm
  - Which parts of the node history should be taken into account?

- Repeat "Simulation" and "Update" until max. time
2.2 Gischa: Monte-Carlo Engine

- A game tree is growing fast
  - chess: ~ 30-40 nodes per ply (middle game)
  - apply dynamical techniques to avoid memory overflow
    - prune subtrees with bad moves
    - save subtrees with promising moves

- Game tree stored in an associative memory
  - Nodes store a lot of information
    - How to identify subtrees efficiently and effectively?
  - An associative structure provides a link between key and value
    - both key and value may be objects
      - Vector: values are indexed by an integer
        - e.g. key = integer, value = real number: \( v = (v_1, v_2, \ldots, v_N) \)
      - AssociativeVector: values are indexed by any type of object
        - e.g. key, value = string: \( p = (p_{\text{wKing}}, p_{\text{bKing}}, p_{\text{wQueen}}, p_{\text{bQueen}} \ldots) \)
2.2 GriScha: Monte-Carlo Engine

- Evaluation of dynamical tree pruning
  - GriScha: pruning layers
    - the key of a node corresponds to its ply layer in the tree
      - other tree structures are not stored (parent node, children nodes)
    - nodes are removed ply by ply from the bottom of the tree until enough memory is available
  - simple, fast
  - nodes storing promising moves may be lost
2.2 GriScha: Monte-Carlo Engine

- Evaluation of dynamical tree pruning

  - **GriScha: pruning subtrees**
    - specify a maximum number of storable nodes (e.g. \( N_{\text{max}} = 100,000 \))
    - characteristics stored in every node
      - \( t_{MC} \): time since last MC update
        - old subtrees (> ~1 s) are assumed to contain only bad moves
      - \( n_{ch} \): number of children nodes
        - pruning subtrees with many children is efficient in memory freeing
      - \( f_{MC} \): value of last MC update
      - \( f_{eval} \): value of positional evaluation function
    - a family of weights is assigned to each node (\( b_0, b_1, b_2, b_3 = 0, 1 \))
      \[
      w_{b_3,b_2,b_1,b_0} = t_{MC}^{b_0} (n_{ch}/10)^{b_1} (\delta_{0,b_2+b_3} + b_2 f_{MC} - b_3 f_{eval})
      \]
      e.g. \( w_{1,0,0,1} \equiv w_9 = t_{MC} (-f_{eval}) \) (\( \delta_{n,m} = \text{Kronecker delta} \))
    - nodes with the highest weights are removed until \( \#\text{nodes} < N_{\text{max}} \)
2.3 Evaluation of Dynamical Tree Pruning

![Bar Graph]

Winning rate of white (11 games/mode)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Winning Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>40%</td>
</tr>
<tr>
<td>1</td>
<td>60%</td>
</tr>
<tr>
<td>2</td>
<td>60%</td>
</tr>
<tr>
<td>3</td>
<td>60%</td>
</tr>
<tr>
<td>4</td>
<td>60%</td>
</tr>
<tr>
<td>5</td>
<td>60%</td>
</tr>
<tr>
<td>6</td>
<td>60%</td>
</tr>
<tr>
<td>7</td>
<td>60%</td>
</tr>
<tr>
<td>8</td>
<td>60%</td>
</tr>
<tr>
<td>9</td>
<td>60%</td>
</tr>
<tr>
<td>10</td>
<td>60%</td>
</tr>
<tr>
<td>11</td>
<td>60%</td>
</tr>
<tr>
<td>12</td>
<td>60%</td>
</tr>
<tr>
<td>13</td>
<td>60%</td>
</tr>
<tr>
<td>14</td>
<td>60%</td>
</tr>
<tr>
<td>15</td>
<td>60%</td>
</tr>
</tbody>
</table>

White = GriScha: pruning subtrees in mode \((b_3, b_2, b_1, b_0) = 0, 1, 2, \ldots, 15\)

Black = GriScha: pruning layers
2.4 ELO Rating System

- ELO ratings are probability predictions
- The probability to win a game is related to the ELO numbers of the players by
  \[ w = \frac{1}{10^{-\frac{\Delta E}{400}} + 1} \]
  \[ \Delta E = \text{difference in the ELO rating} = E_{\text{player}} - E_{\text{opponent}} \]
- A player having 200 ELO points more than the opponent, is winning \( \sim \frac{3}{4} \) of the matches (in the mean):
  \[ \Delta E = 200 \quad \Rightarrow \quad w = 0.76 \]
- A FIDE tournament category covers 25 ELO points
- For comparison: Fermi-Dirac distribution for a system of identical fermions
  \[ \langle N_i \rangle = \frac{1}{e^{(E_i - \mu)/kT} + 1} \]
3. Parallel Tree Searching

- Classification of algorithms
  - scalability (as a function of the number of processors $p$)
    - search speed ($\sim$ number of evaluated tree nodes per second)
    - in games: playing strength
  - speedup $T(1)/T(p)$
    - $\sim 1/\text{search time}$ ($\sim 1/\text{time to evaluate a tree up to horizon nodes}$)
    - Amdahl’s law: $T(p) = T_{\text{seq}} + \frac{T_{\text{par}}}{p} + T_{\text{overhead}}$
      $\Rightarrow$ too many processors are contraproductive if $T_{\text{overhead}} \sim p$
3. Parallel Tree Searching

- Distributed chess algorithms
  - Young Brothers Wait Concept (YBWC) [Feldmann et al., 1991]
    - Master-Slave
  - Dynamic Tree Splitting [Hyatt, 1994]
    - Peer-to-Peer
    - Synchronization problem: “what is more important: depth or score?”

- Results on special hardware
  - ZUGZWANG [Feldmann et al., 1992]
    - YBWC
    - 1024 processors: speedup = 344
    - Finalist, World Computer Chess Championship (WCCC) Madrid, 1992
  - Parallelized game tree search on SIMD machines [Hopp, Sanders, 1995]
    - YBWC, multiple masters
    - Workload dynamically balanced
    - 16,000 processors: speedup = 5850
3.1 Parallel Monte-Carlo Tree Search

- Scalability of a parallelized MCTS method [Chaslot et al., 2010]
  - fast tree parallelization
    - tree stored on the master, slaves perform MC calculations
    - sensitive to Amdahl’s law
  - slow tree parallelization
    - each node stores the tree, slow synchronization (~3 Hz) of subtrees
- Monte-Carlo program MoGo for the game of Go
  - parallelization of MoGo: good efficiency for
    - multi-core machines
    - message-passing machines
  - scalability is slowing down if the number of simulations is increased
  - for certain (difficult) problems “parallelization is not the solution”
  - intermediate communication necessary for optimal performance
3.2 Knowledge-based Tree Search

- Abstract Library for Parallel Search (ALPS) [Xu, 2007]
  - Improving scalability
    - three-tier architecture: master-hub-worker
      - provides improvement for large-scale parallel computing
    - knowledge management system
      - information generated during the tree search stored in shared knowledge pools
      - strong impact on scalability
    - dynamic load balancing schemes
      - all worker nodes have work to do AND the work is useful
      - within a cluster AND between clusters
    - mechanism for adjusting granularity
      - handling data-intensive applications
      - reduce memory usage
  - ALPS offers a framework for developing applications
4. Global Optimization

- **GriScha** depends on many parameters (selection)

  **[Tree]**
  - `max_nodes` = 100,000  
    # max. number of nodes
  - `pruning_factor` = 5  
    # remove 1/pruning_factor of the nodes

  **[Node]**
  - `time_modiﬁer` = 1  
    # time since last update [ms]
  - `child_modiﬁer` = 1  
    # number of children nodes
  - `quality_modiﬁer` = 1  
    # MC value of last update
  - `value_modiﬁer` = 1  
    # value of evaluation function

  **[MonteCarlo]**
  - `depth` = 10  
    # horizon: max. number of ply
  - `fstar` = 1  
    # probability for “bad” moves
  - `(bo, b1, b2, b3)` = (1, 1, 1, 1)  
    # subtree pruning: weight parameters

- How do they influence the strength of GriScha?
From:
Thomas Weise,
(Global Optimization Algorithms,
(2nd Ed., 2009)
Basic cycle of evolutionary algorithms

From:
Thomas Weise,
*Global Optimization Algorithms*,
(2nd Ed, 2009)
4.1 Genetic Algorithms

From:
Thomas Weise,
*Global Optimization Algorithms*,
(2nd Ed., 2009)
4.1 Genetic Algorithms

- search space $\mathcal{G}$ elements (vectors) are manipulated: mutation, crossing
- problem space $\mathcal{X}$ elements (tensors, trees, ...) are solution candidates
- population $\mathcal{P} \subset \mathcal{G} \times \mathcal{X}$ $p = (p.g, p.x)$
- problem: suitable mutation and crossing operators to find global optimum
### 4.1 Example

- Finding a global extremum is hard, in general
- Also genetic algorithms have to be applied with care
- Consider the function

\[
f(x_1, x_2, \ldots, x_k) = \varphi(x_1) - (x_2^2 + \cdots + x_k^2)
\]

\[
\varphi(x) = -x^2(x^2 - 16x + 128)
\]

It has a global maximum

\[
f(16, 0, \ldots, 0) = 32768
\]

and a local maximum

\[
f(-4, 0, \ldots, 0) = 768
\]
4.1 Example

- Initial population
  - genotype: \( k \) random integers
    \[ g = [n_1, \ldots, n_k] \]
    \[ n_i = \text{RandomInt}[-7, 3] \]
    \[ i = 1, \ldots, k \]
  - genotype = phenotype: \( x = g \)
    - here, global maximum not included in the initial domain of phenotypes
  - create population of \( N \) individuals
    \( g_1, g_2, \ldots, g_N \)
    - \( N \) should not be too small.
      Otherwise, a spike-like global maximum may not be found.
      (This is not a problem for the simple example considered here.)
4.1 Example

- **Evaluation**
  - calculate function value for each individual
  \[ f_1 = f(g_1), \ldots, f_N = f(g_N) \]

- **Elite selection**
  - find \( N_E \) individuals with the largest values \( f_i \)

- **Reproduction**
  - create \( N - N_E \) new random individuals out of elite population
  - crossover: \( g, g' \rightarrow g_{\text{cross}} \)
    \[
g = [n_1, \ldots, n_l, n_{l+1}, \ldots, n_k] \
g' = [n'_1, \ldots, n'_l, n'_{l+1}, \ldots, n'_k] \
l = \text{RandomInt}[1, k] \
g_{\text{cross}} = [n_1, \ldots, n_l, n'_{l+1}, \ldots, n'_k]
\]
  - mutation: \( g \rightarrow g_{\text{mut}} \)
    \[
    \nu_i = \text{RandomInt}[-2, 2] \\
g_r = [\nu_1, \ldots, \nu_k] \\
g_{\text{mut}} = g + g_r
    \]

\[ \varphi(x) = -x^2(x^2 - 16x - 128) \]
4.1 Example

- $N = 6$, $N_E = 2$, $k = 5$
  - mutation probability: 0.25
- Normally, under the specified conditions, only the local maximum (-4,0,0,0,0) is found
- Here, a lucky run (due to mutation a bad initial population is "improved")

Initial population

Iteration 1

Iteration 2

Iteration 28

$$f(-2, 0, -3, -1, 1) = 357$$

$$f(2, -4, -5, -2, 1) = 578$$

$$f(5, 0, -5, 1, -3) = 4540$$

$$f(16, -2, -5, 0, -1) = 32738$$

$\varphi(x) = -x^2(x^2 - 16x - 128)$
- Python framework for genetic algorithms
  - Crossover methods
    - single point
    - two point
    - ...
  - Mutation methods
    - integer/real range
    - adding gaussian distributed random values to a gene
    - ...
  - graphical plotting tool
- Added features
  - Checkpoints for dumps
  - Extended MySQL DB adapter
    - import of interrupted evolution runs
- MPI communication interface
  - parallelized genetic algorithms
  - Master node distributes data to slaves
4. Summary and Outlook

- GriScha is extended by a Monte-Carlo chess engine
  - Game tree updated dynamically by pruning subtrees
- For optimizing GriScha-specific parameters, a genetic algorithm framework is currently being integrated into GriScha
  - Framework is parallelizable (master/slaves)
  - Domains of definition can be specified individually for every phenotype variable $x_i$ in the initial population
    - Special care is taken when using mutation operations
    - Results should depend only weakly on details of crossing operations
- Extending parallelization capabilities of GriScha
  - Dynamical load balancing
  - Distributed knowledge pools
- GriScha’s ELO number
  - Collective communication in huge communities feasible?
  - How many beginners throwing the dice make a master?