Fundamentals of Parallel Algorithms

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 368, Room 532
D-69120 Heidelberg
phone: 06221/54-8264
email: `Stefan.Lang@iwr.uni-heidelberg.de`

WS 15/16

Topics

- Foundations of parallel algorithms
- Load balancing

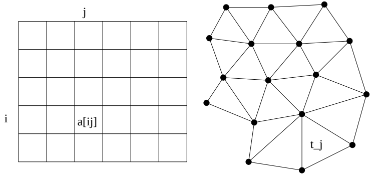
Foundations of Parallel Algorithms

Parallelisation approaches for the design of parallel algorithms:

1. Data partitioning: *Subdivide* a problem into independent subtasks. This serves for the identification of the maximal possible parallelism.
2. Agglomeration: Control of *granularity* to balance computational needs and communication.
3. Mapping: Map processes onto processors. Goal is an optimal tuning of the logical communication structure onto the machine structure.

Foundations of Parallel Algorithms: Data Partitioning

- Calculations are directly associated to specific data structures.
- For each data object certain operations have to be executed, often the same sequence of operations has to be applied onto different data. Thus a part of the data (objects) can be assigned to each process.

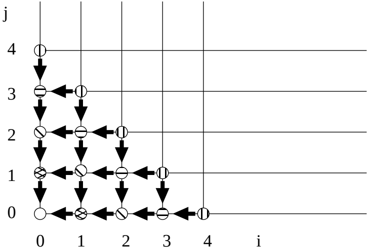


Matrix

Triangulation

- Matrix addition: For $C = A + B$ elements c_{ij} can be processed completely in parallel. In this case one would assign each process Π_{ij} the matrix elements a_{ij}, b_{ij} and c_{ij}.
- Triangulation for the numerical solution of partial differential equations: Here the calculations occur per triangle, that all can be performed at a time, each process could be assigned thus a partial set of the triangles.

Foundations of Parallel Algorithms: Data Dependencies



Data dependencies in the Gauß–Seidel method.

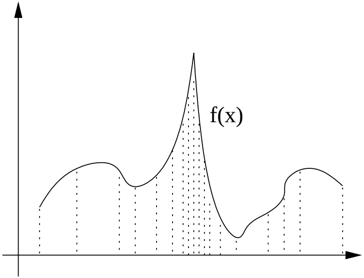
- Operations often can not be performed for all data objects synchronously.
- Example: Gauß-Seidel iteration with lexicographic numbering. Calculation at grid point (i, j) depends on the result of calculations at the grid points $(i-1, j)$ and $(i, j-1)$.
- The grid point $(0, 0)$ can be calculated without any prerequisite.
- All grid points on the diagonal $i + j = const$ can be processed in parallel.

Foundations of Parallel Algorithms: Funct. Partitioning

Functional partitioning

- For different operations on same data.
- Example compiler: This performs the steps: lexical analysis, parsing, code generation, optimization and assembling. Each step can be assigned to a separate process. („macro pipelining").

Irregular problems



- No a-priori partitioning possible.
- Calculation of the integral of a function $f(x)$ by adaptive quadrature.
- Intervall choice depends on f and is determined during the calculation.

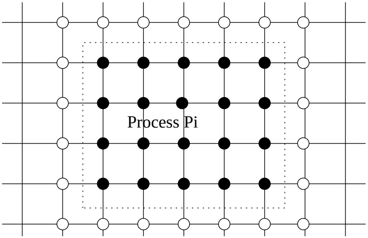
Foundations of Parallel Algorithms: Agglomeration I

- Partition step determines the maximal possible parallelism.
- Realisation (in the sense of a data object per process) is in most cases not meaningful (communication overhead).
- Agglomeration: Mapping of several subtasks to one process, thus the communication is collected for these subtasks in as less as possible messages.
- Reduction of number of messages to sent, further savings for *data locality*
- As *granularity* of a parallel algorithm we denote the relationship:

$$\text{granularity} = \frac{\text{number of messages}}{\text{computing time}}.$$

- Agglomeration reduces also the granularity.

Foundations of Parallel Algorithms: Agglomeration II



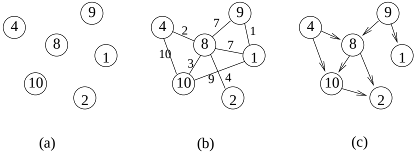
Grid based calculations

- Calculations are performed for all grid points in parallel.
- Assignment of a set of grid points to a process
- All modifications on grid points can be executed under specific circumstances simultaneously. There are thus no data dependencies.

Foundations of Parallel Algorithms: Agglomeration III

- A process owns N grid points and has thus to execute $O(N)$ computing operations.
- Only for grid points at the boundary of the partition a communication is needed.
- Therefore it is only for in total $4\sqrt{N}$ grid points communication necessary.
- Relationship of communcation needs towards computing needs is thus in the order of $O(N^{-1/2})$
- Increase of the number of grid points per processor shrinks the needs for communication relatively to the calculations to an arbitrary small size (*surface-to-volume effect*).

How has the agglomeration to be performed?



(a) (b) (c)

1. Uncoupled calculations
2. Coupled calculations
3. Coupled calculations with time dependency

Foundations of Parallel Algorithms: Partitioning (a)

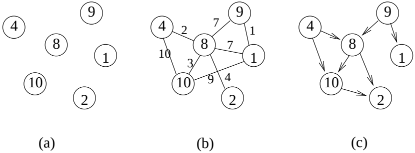
(a) Uncoupled calculations

- Calculation consists of subproblems, that can be calculated completely independent of each other.
- Each subproblem may need different computing time.
- Representable as set of nodes with weights. Weights are a measure for the required computing time.
- Agglomeration is trivial. One assigns the nodes one by one (e.g. ordered by its size or randomly) each to the process, that has the least work (this is the sum of all its node weights).
- Agglomeration gets more complicated if the number of nodes is only known during the calculation (as with the adaptive quadrature) and/or the node weigths are not known a-priori (as e.g. at depth first search).

→ Solution by dynamic load balancing

  - central: a process makes the load balancing
  - decentral: a process gets work of others, that have to much

Foundations of Parallel Algorithms: Partitioning (b)



(b) Coupled calculations

- Standard model for static, data local calculations.
- Calculation is described by an undirected graph.
- First a calculation per node is required, whose computing time is modelled by the node weight. Then each node exchanges data with its neighbor nodes.
- Count of data to be sent is proportional to the associated edge weight.
- Uniform graph with constant weights: trivial agglomeration.

Foundations of Parallel Algorithms: Partitioning

- General graph $G = (V, E)$ for P processors: node set V is to partititon such, that

$$\bigcup_{i=1}^{P} V_i = V, \;\; V_i \cap V_j = \emptyset, \;\; \sum_{v \in V_i} g(v) = \sum_{v \in V} g(v)/|V|$$

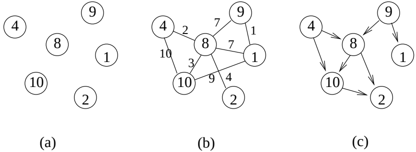
and the separator costs

$$\sum_{(v,w) \in S} g(v,w) \to \min, \; S = \{(v,w) \in E | v \in V_i, w \in V_j, i \neq j\}$$

are minimal.

- This problem is denoted as *graph partitioning problem*.
- $\mathcal{NP}$–complete, already in the case of constant weights and $P = 2$ (graph bisection) $\mathcal{NP}$–complete.
- There exist good heuristics, that generate in linear time (concerning number of nodes, $O(1)$ neighbors) a (reasonably) well partitioning.

Foundations of Parallel Algorithms: Partitioning (c)



(a) (b) (c)

(c) Coupled calculations with temporal dependency

- Model is a directed graph.
- A node can only be calculated, if all nodes of ingoing edges are calculated.
- When a node is calculated, the result is passed further over the outgoing edges. The computing time corresponds to the node weights, the communication time correlates to the edge weights.
- In general very difficult to solve problem.
- Theoretically not „more difficult than graph partitioning“ (also $\mathcal{NP}$–complete)
- Practically no simple and good heuristics are known.
- For special problems, e.g. adaptive multigrid methods, one can however find good heuristics.

Foundations of Parallel Algorithms: Mapping

Mapping: Map the processes on to processors

- Set of processes Π form a undireced graph $G_\Pi = (\Pi, K)$: Two processes are connected with each other, if they communicate together (edge weights could model the extent of the communication).
- Likewise the set of processores P with the communication network forms a graph $G_P = (P, N)$: Hypercube, array.
- Be $|\Pi| = |P|$ and the following question has to be asked:
  Which process shall be executed on which processor?
- In general we want to perform the mapping in such a way, that processes that communicate with each other are mapped onto neighboring or near processors.
- This optimization problem is denoted as *graph assignment problem*
- This is again $\mathcal{NP}$–complete (unfortunately).
- Contemporary multi processors possess very powerful communication networks: In cut-through networks the transmission time of a message is practically distant independent.
- The problem of optimal process placement is thus not preferential any more.
- A good process placement is nevertheless important if many processes communicate *synchronously* with each other.

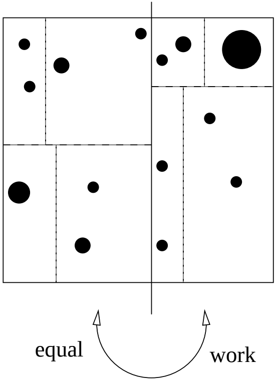
Foundations of Parallel Algorithms: Load Balancing

Situation 1: Static distribution of uncoupled problems

- Task: Partitioning of accumulated work on to the different processors.
- This corresponds to the agglomeration step at which we have again combined the subproblems, that are processable in parallel.
- The measure for the work is hereby known.
- **Bin Packing**
  - Initially all processors are empty.
  - Nodes, that are available either in arbitrary sequence or sorted (e.g. correspoinding to their size), are packed after each other on the processor that has currently the least work.
  - This also functions dynamically, if new work is generated in the calculation.

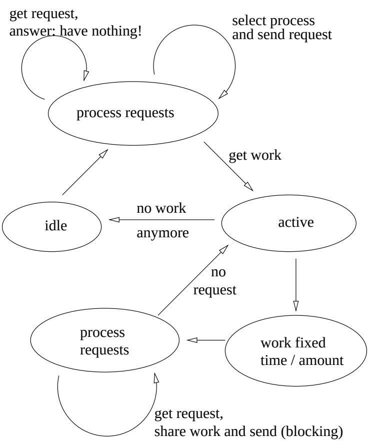
Foundations of Parallel Algorithms: Load Balancing

- **Recursive Bisection**
  - Each node is assigned a position in space.
  - The space is divided orthogonal to the coordinate system such, that in each part is about the same amount of work.
  - This approach is then recursively applied onto the generated subvolumes with alternating coordinate direction.



Foundations of Parallel Algorithms: Load Balancing

Situation 2: Dynamic distribution of uncoupled problems



activity/state diagram

- The measure for the work is not known.
- Process is either active (performs works) or idle (without work).
- During work division the following questions have to be considered:
  - What do I want to pass away? For travelling-salesman problem e.g. preferably nodes from the stack, that reside far low.
  - How much do I want to pass away? E.g. half of the work (half split).
- Beside the work sharing further communication can take place (Forming of the global minimum for branch-and-bound during travelling-salesman).
- Furthermore the problem of termination recognition exists.
  When are all processes idle?

Foundations of Parallel Algorithms: Load Balancing

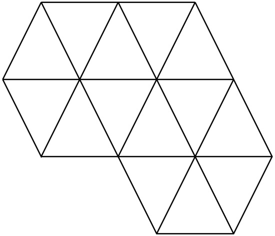
Which idle process shall be addresses next?
Different selection strategies:

- **Master/Slave (Worker) principle**
  A process distributes work. It knowns, who is active resp. free and forwards the request. It controls (since it knowns, who is idle) also the termination problem. Disadvantage this method does not scale. Alternatively: hierarchical structure of masters.
- **Asynchrones Round Robin**
  The process Π_i has a variable target_i. It sends its requsts to Π_{target_i} and sets then $\text{target}_i = (\text{target}_i + 1)\%P$.
- **Globales Round Robin**
  There is only a single global variable target. Advantage: no synchronous request onto the same process. Disadvantage: access on the global variable (what e.g. a server process can do).
- **Random Polling**
  Each chooses randomly a process with same probabilty ($\rightarrow$ parallel random generator, watch out at least for the distribution of seeds). This approach provides a uniform distribution of the request and needs no global resource.

Foundations of Parallel Algorithms: Load Balancing

Graph partitioning

Consider a Finite-Element mesh:



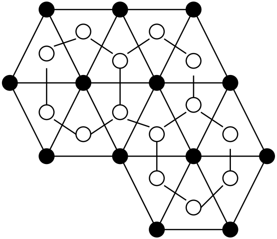
It consists of triangles $T = \{t_1, \ldots, t_N\}$ with

$$\bar{t_i} \cap \bar{t_j} = \begin{cases} \emptyset \\ \text{a node} \\ \text{an edge} \end{cases}$$

- In the method of Finite-Elements the work is associated to the nodes.
- Alternatively one can also put a node in each triangle and connect these with edges over the triangle neighbors. Now the evolving dual graph can be considered.

Foundations of Parallel Algorithms: Load Balancing

Graph and according dual graph



The partitioning of the graph to processors leads to the graph partitioning problem. Therefore we define the following notation:

$$
\begin{array}{ll}
G = (V, E) & \text{(Graph or dual graph)} \\
E \subseteq V \times V & \text{symmetric (undirected)}
\end{array}
$$

The weighting functions

$$
\begin{array}{ll}
w : V \longrightarrow \mathbb{N} & \text{(computing demand)} \\
w : E \longrightarrow \mathbb{N} & \text{(communication)}
\end{array}
$$

Foundations of Parallel Algorithms: Load Balancing

The total work

$$W = \sum_{v \in V} w(v)$$

Furthermore be k the number of partitions to be constituted, where holds $k \in \mathbb{N}$ and $k \geq 2$. We seek now the partition mapping

$$\pi : V \longrightarrow \{0, \ldots, k-1\}$$

and the associated edge separator

$$X_\pi := \{(v, v') \in E \mid \pi(v) \neq \pi(v')\} \subseteq E$$

The graph partitioning problem consists of finding the mapping π such that the cost function (communication costs)

$$\sum_{e \in X_\pi} w(e) \to \min$$

gets minimal under the restriction (equal work balancing)

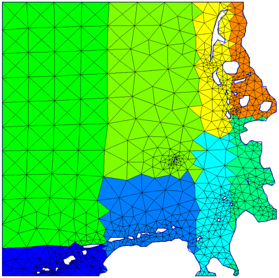
$$\sum_{v, \pi(v) = i} w(v) \leq \delta \frac{W}{k} \quad \text{for all } i \in \{0, \ldots, k-1\}$$

where δ determines the tolerated imbalance (δ = 1.1 10% deviation).

Foundations of Parallel Algorithms: Load Balancing

- Calculation costs dominate communication costs. Otherwise the partitioning might eventually be not necessary because of the high communication costs. This is albeit only a model for the run time!
- For binary partitioning one speaks about the graph bisection problem. By recursive bisection 2^d-way partitionings can be generated.
- Problematically the graph partitioning is NP-complete for k $\geq$ 2.
- Optimal solution thus would dominate the original calculation as parallel overhead and is not acceptable.

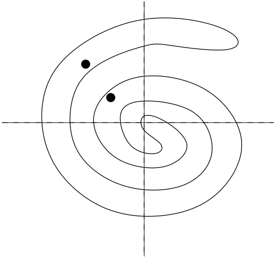
$\rightarrow$ Necessity of fast heuristics.



Foundations of Parallel Algorithms: Load Balancing

Recursive coordinate bisection(RCB)

- One needs the positions of the nodes in space (for Finite-Element applications these exist).
- Up to now we have seen the methods under the name **recursive bisection**.
- This time the problem is coupled. Hence it is important, that the space, in whose coordinates the bisection is performed, coincides with the space, in which the nodes reside.
- In the picture this is not the case. Two nodes may be near each other in space, a coordinate bisection does not make sense since the points here are not coupled, thus a processor has no advantage of storing both.



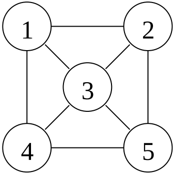
Counter example for application of RCB

Foundations of Parallel Algorithms: Load Balancing

Rekursive Spectral Bisektion(RSB)

Here the positions of nodes in space are not needed. One constitutes first the Laplacian matrix A(G) for the given graph G. This is defined in the following way:

$$A(G) = \{a_{ij}\}_{i,j=1}^{|V|} \quad \text{with} \quad a_{ij} = \begin{cases} \text{degree}(v_i) & i = j \\ -1 & (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$



	1	2	3	4	5
1	3	−1	−1	−1	0
2	−1	3	−1	0	−1
3	−1	−1	4	−1	−1
4	−1	0	−1	3	−1
5	0	−1	−1	−1	3

graph and related Laplacian matrix

Foundations of Parallel Algorithms: Load Balancing

Then solve the eigen value problem

$$Ae = \lambda e$$

The smallest eigen value λ_1 equals zero, since with $e_1 = (1, \dots, 1)^T$ applies $Ae_1 = 0 \cdot e_1$. The second smallest eigen value λ_2 however is not zero, if the graph is connected.

The bisection now is performed by means of the components of the eigen vector e_2, one builds for $c \in \mathbb{R}$ the two index sets

$$I_0 = \{i \in \{1, \dots, |V|\} \mid (e_2)_i \leq c\}$$
$$I_1 = \{i \in \{1, \dots, |V|\} \mid (e_2)_i > c\}$$

and the partition mapping

$$\pi(v) = \begin{cases} 0 & \text{if } v = v_i \wedge i \in I_0 \\ 1 & \text{if } v = v_i \wedge i \in I_1 \end{cases}$$

Tough one chooses c such that the work is equally distributed (median).

Foundations of Parallel Algorithms: Load Balancing

Kerninghan/Lin

- Iteration method, that improves a given partitioning under consideration of the cost functional.
- We restrict ourselves to bisection (k-way extension is possible) and assume the node weights as 1.
- Furthermore the count of nodes shall be even.
- The method of Kerninghan/Lin is mostly used in combination with other methods.

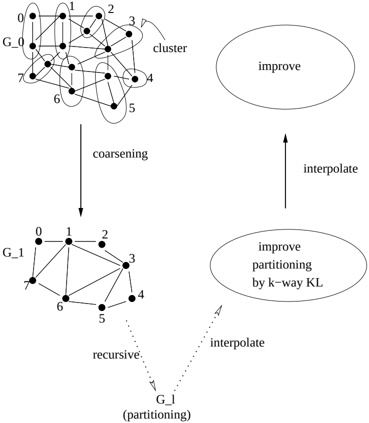
Foundations of Parallel Algorithms: Load Balancing KL

```
i = 0; $| V | = 2n$;
// Generate $\Pi_0$ such, that equal distribution is given.
while (1) { // iteration step
    $V_0 = \{v \mid \pi(v) = 0\}$;
    $V_1 = \{v \mid \pi(v) = 1\}$;
    $V_0{}' = V_1{}' = \emptyset$;
    $\bar{V_0} = V_0$;
    $\bar{V_1} = V_1$;
    for ( i = 1 ; i $\leq$ n ; i++ )
    {
        // choose $v_i \in V_0 \setminus V_0{}'$ und $w_i \in V_1 \setminus V_1{}'$ such, that
        $\sum_{e \in (\bar{V_0} \times \bar{V_1}) \cap E} w(e) - \sum_{e \in (V_0{}'' \times V_1{}'') \cap E} w(e) \rightarrow max$
        // where
        $V_0{}'' = \bar{V_0} \setminus \{v_i\} \cup \{w_i\}$
        $V_1{}'' = \bar{V_1} \setminus \{w_i\} \cup \{v_i\}$
        // set
        $\bar{V_0} = \bar{V_0} \setminus \{v_i\} \cup \{w_i\}$;
        $\bar{V_1} = \bar{V_1} \setminus \{w_i\} \cup \{v_i\}$;
    } // for
    // rem.: max can be negative, thus worsening of the separator costs
    // result at this point: sequence of pairs $\{(v_1, w_1), \ldots, (v_n, w_n)\}$.
    // $V_0, V_1$ have not been altered.
    // Now choose a partial sequence up to m $\leq$ n, that performs a
    // maximal improvement of the costs („hill climbing “)
    $V_0 = V_0 \setminus \{v_1, \ldots, v_m\} \cup \{w_1, \ldots, w_m\}$;
    $V_1 = V_1 \setminus \{w_1, \ldots, w_m\} \cup \{v_1, \ldots, v_m\}$;
    if ( m == 0 ) break; // end
} // while
```

Foundations of Parallel Algorithms: Load Balancing

Multilevel k-way partitioning

1. Agglomeration of nodes of the starting graph G^0 (e.g. random or by means of heavy edge weight) in clusters.
2. These clusters define the nodes in a coarsened graph G^1.
3. By recursion this behaviour leads to a graph G^l.



1. G^l is now partitioned (e.g. RSB/KL).
2. After that the partition function is interpolated onto the finer graph G^{l-1}.
3. This interpolated partitioning function can now be improved again via KL recursively and then be interpolated onto the next finer graph.
4. In this way we continue recursively up to the starting graph.
5. The implementation is then possible in $\mathcal{O}$(n) steps. The method provides qualitatively profound partitions.

Foundations of Parallel Algorithms: Load Balancing

Further problems

Further problems for the partitioning are

- **Dynamic repartitioning:** The graph partitioning shall be changed with the least possible local rebalancing to regain work balance.
- **Constraint partitioning:** From other algorithmic parts additonal data dependencies exist.
- **Parallelisation of partitioning methods:** Is necessary for large data sets (Here finished software exists)