Parallel Computing

Design of Parallel Algorithms

Thorsten Grahs, 15.06.2015
Designing parallel Algorithms

PCAM approach

Designing and Building Parallel Programs
Ian Forster
http://www.mcs.anl.gov/itf/dbpp/

- Partitioning
- Communication
- Agglomeration
- Mapping

PCAM approach
Initial situation

- A sequential programme consist of sequence of elementary steps to solve a problem.
- A parallel programme has to take into account several additional aspects:
  - Decomposition into subtasks
  - Data allocation
  - Communication
  - Merging/combining similar tasks
  - Scheduling/mapping to processors

Goal: Methodological approach
Methodology

Realisations of parallel Algorithms

- For each aspect there are several implementation options based on different parallel computer architectures
- These different approaches may lead to different run time realisations for the same task

Methodology
Used as a guide for designing/outlining parallel algorithms

Foster’s Methodology
PCAM-Method
- Partitioning
- Communication
- Agglomeration
- Mapping
PCAM methods

- **Partitioning**
  - The computation that is to be performed and the data operated are decomposed into small tasks.

- **Communication**
  - Determine required communication to coordinate tasks.
  - Appropriate communication structures are defined.

- **Agglomeration**
  - Task and communication structures are evaluated with respect to performance and implementation costs.
  - If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.

- **Mapping**
  - Each task is assigned to a processor with the goals
    - maximising processor utilisation
    - minimising communication costs.
Partitioning | Goals

- The partitioning stage of a design is intended to expose opportunities for parallel execution.
- Focus is on defining a large number of small tasks i.e. fine-grained decomposition of a problem.
- Fine-grained decomposition provides the greatest flexibility in terms of potential parallel algorithms.
- Decomposition of computation and data into task without regarding the available processors
  - ⇒ inherent parallelism, scalability
- Determination of the maximum available parallelism.
- Avoiding the duplication of data / calculations.
A good partition divides into small pieces both the computation associated with a problem and the data on which this computation operates.

Programmers most commonly first focus on partitioning the data associated with a problem, called domain decomposition.

The alternative approach, termed functional decomposition, first decomposes the computation.

These are complementary techniques.

Seek to avoid replicating computation and data (may change this later in process).
Domain decomposition

- First focus on the data associated with a problem, i.e.
  - partition data; ideally divide data into small pieces of approximately equal size.

- Next partition computation,
  - This could be done by associating each operation with the data on which it operates.

- Focus first on the largest data structure or on the data structure that is accessed most frequently.

- Sometimes called data parallelism
Example | 3D cube of data

- 1-D decomposition: split cube into slice (each slice is 2-D, coarse granularity).
- 2-D decomposition: split cube into columns (each column is 1-D)
- 3-D decomposition: split cube into individual data elements. (fine granularity)
Functional decomposition

- Initial focus is on the computation that is to be performed rather than on the data.
- Divide computation into disjoint tasks.
- Examine data requirements of tasks:
  1. Requirements may be disjoint, in which case the partition is complete.
  2. Requirements may overlap significantly, in which case considerable communication will be required to avoid replication of data.
  3. Second case is a sign that a domain decomposition approach should be considered instead.
- Sometimes called control parallelism
Functional decomposition is valuable as a different way of thinking about problems and should be considered when exploring possible parallel algorithms.

A focus on the computations that are to be performed can sometimes reveal structure in a problem, and hence opportunities for optimization, that would not be obvious from a study of data alone.

Functional decomposition is an important program structuring technique; can reduce the complexity of the overall design.
Example

Matrix multiplication

\[ A \cdot B = C \]

A, B, C \( n \times n \) matrices

- Domain decomposition
  - based on output matrix
  - \( \Rightarrow n^2 \) tasks

- Functional decomposition
  - based on arithmetic operations
  - \( \Rightarrow n^3 \) Multip., \( n^2(n-1) \) Additions
Check list partitioning

- Partitioning defines at least an order of magnitude more tasks than there are processors
  \[ \Rightarrow \text{ Yields good flexibility} \]
- Partitioning avoids
  - redundant calculations
  - memory consumptions
  \[ \Rightarrow \text{ good scalability to deal with large problems.} \]
- tasks are of comparable size
  \[ \Rightarrow \text{ good balancing expected} \]
- Increases the number (not size) of tasks with the problem size?
  \[ \Rightarrow \text{ good scalability} \]
- Identified several alternative partitions
  \[ \Rightarrow \text{ flexibility} \]
Communication | Goals

- Conceptualize a need for communication between two tasks as a channel linking the tasks, on which one task can send messages and from which the other can receive.
- Identifying necessary communication between task in order to allow computation to proceed
- Identifying channel structure, i.e. sender and receiver
- Specify the message/data that are to be send
- Avoid unnecessary communication
- Distribute communication operations over many tasks
- Organize communication operations in a way that permits concurrent execution.
Communication in domain decomposition

- Communication requirements can be difficult to determine in domain decomposition problems.
  - First partition data structures into disjoint subsets and then associate with each datum those operations that operate solely on that datum.
  - Some operations that require data from several tasks usually remain.
  - Organizing the resulting communication in an efficient manner can be challenging.

- Communication requirements in parallel algorithms obtained by functional decomposition are often straightforward:
  - They correspond to the data flow between tasks.
Patterns of communication

Foster categorizes communication patterns along four loosely orthogonal axes

1. local ⇔ global
2. structured ⇔ unstructured
3. static ⇔ dynamic
4. synchronous ⇔ asynchronous
Communication patterns I

**local ⇔ global**

- **local**: each task communicates with a small set of other tasks (its “neighbours”)
- **global**: each task to communicate with many (nearly all) tasks.

**structured ⇔ unstructured**

- **structured**: a task and its neighbours form a regular structure, such as a tree or grid
- **unstructured**: communication networks may be arbitrary graphs
Communication patterns II

static ⇔ dynamic

- **static**: the identity of communication partners does not change over time
- **dynamic**: Communication structures may be determined by data computed at runtime and may be highly variable.

synchronous ⇔ asynchronous

- **synchronous**: sender and receiver execute in a coordinated fashion, with sender/receiver pairs cooperating in data transfer operation
- **asynchronous**: may require that a receiver obtain data without the cooperation of the sender.
Local communication

Local communication structure

A *local communication structure* is obtained when an operation requires data from a small number of other tasks.

- Easy to define channels that link
  - consumer task (needs the data) with
  - producer tasks (have the data).

- **Example**
  Finite differences Laplacian stencil
Finite differences is a method used to solve certain differential equation problems. A multidimensional grid is repeatedly updated by replacing the value at each point with a weighted average of the values at a small, fixed number of neighbouring points. Set of values required to update a single grid point is called that grid point’s stencil. For example,

\[ U_{i,j}^{n+1} = \frac{U_{i-1,j}^n + U_{i,j+1}^n - 4U_{i,j}^n + U_{i+1,j}^n + U_{i,j+1}^n}{h^2} \]

uses a five-point stencil to update each element \( U_{i,j} \) of a two-dimensional grid.

The variable \( n \) indicates the time step \( t^n = n \cdot \Delta t \) and \( i \) and \( j \) denote the grid locations.
The communications pattern for a particular node of the Laplace stencil.

The communications channels shown by the arrows in the diagram on the right.

Assume that the domain decomposition results in a distinct task for each point in the two-dimensional grid.

The task allocated to $U_{i,j}$ must compute the sequence $U_{i,j}^1, U_{i,j}^2, U_{i,j}^3, \ldots$
Local communication | Laplace stencil

- Computation requires in turn the four corresponding sequences which are produced by the four neighbouring tasks:

\[
\begin{align*}
U_{i-1,j}^1, U_{i-1,j}^2, U_{i-1,j}^3, \ldots & \quad U_i^{j-1}, U_i^{j-1}, U_i^{j-1} \\
U_i^{j+1}, U_i^{j+1}, U_i^{j+1}, \ldots & \quad U_i^{j+1}, U_i^{j+1}, U_i^{j+1}
\end{align*}
\]

- Define channels linking each task that requires a value with the task that generates that value.
- Each task then executes the following logic:

\[
\text{for } t = 0 \text{ to } T - 1 \\
\text{send } U_{i,j}^t \text{ to each neighbour} \\
\text{receive } U_{i-1,j}^n, U_{i+1,j}^n, U_{i,j-1}^n, U_{i,j+1}^n \text{ from neighbours} \\
\text{compute } U_{i,j}^{n+1} \\
\text{endfor}
\]
Global communication

- In contrast to local communication, a global comm. operation is one in which many tasks must participate.
- May result in too many communications or may restrict opportunities for concurrent execution.
- Consider a parallel reduction operation, that is, an operation that reduces N values distributed over N tasks using a commutative associative operator such as addition: \( S = \sum_i X_i \)
- If a single master task requires the result S we can define a communication structure that allows each task to communicate its value to the manager independently.
- Because the manager can receive and add only one number at a time, this approach takes O(N) time to sum N numbers. (not a very good parallel algorithm!)
Global communication

- This illustrates two general problems that can hinder efficient parallel execution in algorithms based on a purely local view of communication:
  1. The algorithm is centralized: it does not distribute computation and communication. A single task (in this case, the manager task) must participate in every operation.
  2. The algorithm is sequential: it does not allow multiple computation and communication operations to proceed concurrently.
- We must address both these problems to develop a good parallel algorithm.
- Foster’s example comes from finite elements, where the finite element mesh is composed of triangles and the number of edges incident to a vertex is not constant.
- Channel structure representing communication partners can irregular, data-dependent and can change over time.
- Unstructured communication complicates the tasks of agglomeration and mapping.
- It is often non-trivial to determine an agglomeration strategy that both creates tasks of approximately equal size and minimizes communication requirements by creating the least number of intertask edges.
Asynchronous communication

- In the case of an asynchronous communication pattern, we have a mismatch of:
  - data channels or
  - information about necessary data
- Tasks that possess data (producers) are not able to determine when other tasks (consumers) may require data.
- The consumers in the communication must explicitly request data from producer side.
Check list Communication

- Do all tasks perform about the same number of communication operations?
  ⇒ Influences scalability
- Does each task communicate only with a small number of neighbours?
- Are communication operations able to proceed concurrently?
  If not, your algorithm is likely to be inefficient and non-scalable.
- Is the computation associated with different tasks able to proceed concurrently?
Agglomeration

Goals

- Move from an abstract level toward the concrete one.
- Revisit decisions made in the partitioning and communication phases (to obtain an efficient algorithm)
- Minimise the communication costs
- Combination of strongly interacting tasks
- Increasing task
- Improving scalability

Methods

- Replication of computation
- Overlapping communication and computation
At this point we’ve broken down our problem enough that we understand the individual tasks and the necessary communication between tasks.

Now make it practical and as efficient as possible.

Naturally two question arises:
1. is it useful to combine, or agglomerate, tasks to reduce the number of tasks?
2. is it worthwhile to replicate data and/or computation?

The number of tasks yielded by the agglomeration phase, although reduced, may still be greater than the number of processors. Resolution is deferred to the mapping phase.
Agglomeration | Conflicting goals

Three sometimes-conflicting goals guide decisions concerning agglomeration and replication:
1. reducing communication costs by increasing computation and communication granularity,
2. retaining flexibility with respect to scalability and mapping decisions, and
3. reducing software engineering costs.

Increasing granularity

- A large number of fine-grained tasks does not necessarily produce an efficient parallel algorithm.
- Communication costs and task creation costs are overhead that can be reduced by increasing granularity.
Examples of Agglomeration

Matrix multiplication

- Use sub-matrices or sub-blocks instead of single matrix elements
- Ratio of communication/computation decreases
  \[\Rightarrow\] improves scalability

For regular multidimensional structures such as grids, cubes etc., agglomeration is possible in several ways.

- Dimension reduction
- Block separation
Examples | Fine grained approach

- Fine-grained partition of $8 \times 8$ grid.
- Partitioned into 64 tasks.
- Each task responsible for a single point.
- $64 \times 4 = 256$ communications are required, 4 per task.
- Total of 256 data values transferred

Outgoing messages are dark shaded and incoming messages are light shaded.
Examples | Coarse grained approach

- Coarse-grained part. of $8 \times 8$ grid
- Partitioned into 4 tasks.
- Each task resp. for 16 point.
- $4 \times 4 = 16$ communications are required.
- Total of $16 \times 4 = 64$ data values transferred

Outgoing messages are dark shaded and incoming messages are light shaded.
Surface-to-Volume effects

- This reduction in communication costs is due to a surface-to-volume effect.
- The communication requirements of a task are proportional to the surface of the sub-domain on which it operates, while the computation requirements are proportional to the sub-domain’s volume.
- In a two-dimensional problem, the *surface* scales with the problem size while the *volume* scales as the problem size squared. The communication/computation ratio decreases as task size increases.
- From the viewpoint of efficiency it is usually best to increase granularity by agglomerating tasks in all dimensions rather than reducing the dimension of the decomposition.
Dimension-/Block reduction

**Dimension reduction**

\[ n^3 \] tasks
with
volume: 1
surface: 6

\[ \rightarrow \]

\[ n^2 \] tasks
with
volume: \( n \)
surface: \( 4n+2 \)

**Block reduction**

\[ n^3 \] tasks
with
volume: 1
surface: 6

\[ \rightarrow \]

\[ (n/k)^3 \] tasks:
with
volume: \( k^3 \)
surface: \( 6k^2 \)
Check list Agglomeration

- Has agglomeration reduced communication costs by increasing locality?
- If agglomeration has replicated computation, have you verified that the benefits outweigh its costs?
- Does the number of tasks still scale with problem size?
- Can the number of tasks be reduced still further, without introducing load imbalances, increasing software engineering costs, or reducing scalability?
- Has agglomeration yielded tasks with similar computation and communication costs? The larger the tasks created by agglomeration, the more important it is that they have similar costs.
Check list Agglomeration (cont.)

- If agglomeration eliminated opportunities for concurrent execution, have you verified that there is sufficient concurrency for current and future target computers?
- Can the number of tasks be reduced still further, without introducing load imbalances, increasing software engineering costs, or reducing scalability?
- If you are parallelize an existing sequential program, have you considered the cost of the modifications required to the sequential code?
At this point we have a set of tasks and we need to assign them to processors on the available machine.

The mapping problem does not arise on uniprocessor or on shared-memory computers that provide automatic task scheduling.

General-purpose mapping mechanisms have yet to be developed for scalable parallel computers. Our goal in developing mapping algorithms is normally to minimize total execution time.

The general-case mapping problem is NP-complete.
Mapping | Goals

- Minimising total execution time
- Assignment of processors to parallel tasks

Strategies

- Place tasks that are able to execute concurrently (independently) on different processors, so as to enhance concurrency.
- Place tasks that communicate frequently on the same processor, so as to increase locality.

Mapping methods

- static distribution of tasks
- dynamic load balancing (task scheduling)
Mapping methods

**static distribution of tasks**
- all information available before computation starts
- use off-line algorithms to prepare before execution time
- run as pre-processor, can be serial, can be slow and expensive.

**dynamic load balancing (task scheduling)**
- information not known until runtime
- work changes during computation (e.g. adaptive methods), or locality of objects change (e.g. particles move)
- use on-line algorithms to make decisions mid-execution; must run side-by-side with application, should be parallel, fast, scalable. Incremental algorithm preferred (small changes in input result in small changes in partitions)
Mapping methods

- When domain decomposition is used there is often a fixed number of equal-sized tasks and structured communication.
- If, instead, there are variable amounts of work per task and/or unstructured communication patterns, we might use load balancing algorithms that seek to identify efficient agglomeration and mapping strategies.
- When the number of tasks or the amount of computation/communication per task changes dynamically during program execution we might use dynamic load-balancing strategy in which a balancing algorithm is executed periodically to determine new agglomeration & mapping.
- If functional decomposition is used we can use task-scheduling algorithms which allocate tasks to processors that are idle or that are likely to become idle.
Load balancing

Several ways (i.e. approaches/algorithms) to map your data:

- Recursive Bisection
- Local Algorithms
- Probabilistic Methods
- Cyclic Methods

We examine the recursive bisection algorithm
Recursive bisection

- Partition a domain into sub-domains of approximately equal computational cost while attempting to minimize the number of channels crossing task boundaries.
- First cut in one dimension to yield two sub-domains.
- Cuts are then made recursively in the new sub-domains until we have as many sub-domains as we require tasks.
Mapping checklist

- If considering an SPMD design for a complex problem, have you also considered an algorithm based on dynamic task creation and deletion?
- If considering a design based on dynamic task creation and deletion, have you also considered an SPMD algorithm?
- If using a centralized load-balancing scheme, have you verified that the manager will not become a bottleneck?
- If using a dynamic load-balancing scheme, have you evaluated the relative costs of different strategies?
- If using probabilistic or cyclic methods, do you have a large enough number of tasks to ensure reasonable load balance?