Chapter 2: Designing Parallel Algorithms

Got ourselves some
 ✓ parallel machine abstraction
 ✓ parallel programming abstractions

Now need to translate a specification of a problem into an algorithm that displays concurrency, scalability and locality (will do modularity later).

- Requires creativity! ("art")
- There are no simple recipes
- Methodological design can help:
  ✓ Examine all option, evaluate all alternatives
  ✓ Reduce backtracking costs

Provide **FRAMEWORK** for doing this here. Try and develop:

- Intuition
- Experience
- Eye for design flaws that compromise efficiency/scalability

Design is basically a highly nonlinear process!

We will try and create some basic principles and some design checklists

2.1 Methodological Design

For our programming problems:

- Solution not generally unique
- Existing sequential solutions may be very misleading

Strategy:

- Consider machine independent issues (e.g. concurrency) *EARLY*
- Consider machine dependent issues *LATER*

Four stages (four main weapons of design!): P C A M

- Partitioning
- Communication
- Agglomeration
- Mapping
2.1 Methodological Design

Partitioning (domain/functional):
- Decompose computation and data into small tasks. Concentrate on recognising parallel opportunities.
- Ignore no. of processors and other practical issues.

Communication (local/global, static/dynamic, structured/unstructured, synchronous/asynchronous):
- Set up channels = communication structure required to co-ordinate task execution.

Agglomeration (reduce communication and therefore costs):
- Evaluate task-channel communication structures with regard to performance and implementation costs. Combine into larger tasks.

Mapping (load balancing, task scheduling, static or run-time):
- Assign tasks to processors. Satisfy competing goals of maximal processor utilisation but minimal communication.

P & C: concentrate on concurrency and scalability
A & M: concentrate on performance, especially due to locality

Result can be:
- Program that creates and destroys tasks dynamically
- MORE LIKELY FOR US: SPMD - static structure that assigns one task per processor for all time
  (=> agglomeration and mapping are basically the same thing)

2.2 Partitioning

• As mentioned earlier when describing task-channel model, at this stage, aim for FINE GRAIN decomposition of the problem:
  - Confucious, he say: “Fine grained sand is easy to pour ; Bricks are not so easy”
  - ⇒ fine-grained is more FLEXIBLE
  - Agglomerate LATER to change granularity
  - Take machine/practical details into account at THAT stage, not here

• Want to divide problem into pieces of computation AND the pieces of data that the computation operates on = tasks
  - Obviously want DISJOINT (non-overlapping) sets of computation/data if possible

• Two ways of doing this:
  - Domain decomposition
  - Functional decomposition
2.2.1 **Domain decomposition**

Seek to decompose the *DATA* of the problem by *GEOMETRY*

We would like *small pieces of equal size* => fine-grained and load-balanced

Then associate computations with that data => tasks

There may be different geometrical decompositions at different phases of the program (e.g. input, intermediate computation, some other computation, output etc)

*Rule of thumb*: start with the largest data structure or the one that is going to be accessed most frequently

---

**Domain decomposition example:** *(The one you all know and love -- Cartesian grids)*

- **1-D data decomposition**
  - 1 dimension cropped
  - 2 dimensions full
  - Task = plane

- **2-D data decomposition**
  - 2 dimensions cropped
  - 1 dimension full
  - Task = column/pencil

- **3-D data decomposition**
  - 3 dimensions cropped
  - 1 dimension full
  - Task = grid point (or few)
  - Aggressive!
### 2.2.2 Functional decomposition

Seek to decompose the *COMPUTATION* of the problem into disjoint tasks.

Then add data:
- If data also disjoint, good!
- If not, and LOTS of communication is required, ABORT and try domain decomposition!

**E.g. 1:** Earlier “tree search” problem:
- No obvious data decomposition
- New searches (=computation) are obvious tasks

**E.g. 2:** Complex system modelling e.g. climate model
- Divide tasks by *physics*
- Sections for atmosphere, ocean, ice, carbon dioxide, …
- Each “model” = separate task
- Works if little communication between tasks
- Tasks can be further decomposed (by domain!)

### 2.2.3 Partitioning design checklist

- Does the partition define at least an order of magnitude more tasks than there are processors on the target machine? If not, you may not have flexibility in later design stages.
- Does the partition avoid redundant computation (and storage)? If not, algorithm may not scale well to larger problems.
- Are tasks of comparable size? If not, may be load balancing issues -- hard to allocate equal work to processors.
- Does the number of tasks scale with the problem size? Increase in problem size should ideally also increase the number of tasks rather than increase the size of individual tasks. If not, may not be able to solve really big problems even when more processors are available.
- Have you at least identified the alternatives (domain vs functional)? May help design issues down the road …
- Have you looked for some changes to the original problem that might make life a lot easier in parallel? E.g. Switch numerical methods from spectral to finite-differences! Sometimes need to examine pre-history, because things were designed for sequential programming and there are different trade-offs. Due to parallel issues, may be better to start again from scratch!
2.3 Communication

So we have *TASKS*

- Tasks should be able to run concurrently
- BUT tasks seldom completely independent

⇒ Data must be transferred between tasks for computation to proceed

Information flow => communication => *CHANNELS*

Two conceptual stages:

1. Define channels between tasks that need to communicate
2. Specify messages to be sent/received on those channels
   (May not actually do anything for 1. - just a concept!)

We need to consider:

- Sending messages incurs physical costs => we want to avoid unnecessary communication
- We want to distribute communication over tasks - don’t do all in one task - load balancing
- We need to organise to allow concurrency

---

2.3 Communication

Domain decomposed problem:

- Even simple partitions can have complex communication structures
- Tasks (data in a partition) may have many operations NOT needing communication and some sprinkled around that do
- => hard to agglomerate communication due to it’s irregularity.

Functional decomposition problem:

- Often straightforward
- Data flow is between *coarse-grained* tasks e.g. wind data from atmospheric model is used to drive ocean model.
- => Easier to agglomerate
2.3 Communication

Four basic types of communication: (our four main weapons of comm ...?)

Local/global
- Local - communication with a few other neighbouring tasks
- Global - communication with lots of other tasks

Structured/unstructured
- Structured - regular grid communication, obvious patterns
- Unstructured - arbitrary structure

Static/dynamic
- Static - communication partners do not change over time (channels fixed)
- Dynamic - comm partners determined by data at runtime

Synchronous/asynchronous
- Synchronous - senders and receivers co-ordinate
- Asynchronous - no co-operation between senders and receivers necessary (just let fly!)

Talk about these in detail ...

2.3.1 Local communication

Data required only from a small number of other tasks

Usually simple to define channels and send/recieves

Illustrate with an example - Jacobi iteration:
- Update of a grid by weighted sum of neighbouring grid points
- Which neighbours are used = stencil
- Try 5-point stencil:

\[ x_{i,j}^{(t+1)} = \frac{4x_{i,j}^{(t)} + x_{i+1,j}^{(t)} + x_{i-1,j}^{(t)} + x_{i,j+1}^{(t)} + x_{i,j-1}^{(t)}}{8} \]

Task structure?

Data decomposition
- Assign the update of a single x(i,j) for each time t=1,2,3,… to a task

Communication structure?
- Each task for an x(i,j) requires knowledge of 4 neighbours
  - x(i+1,j), x(i-1,j), x(i,j+1), x(i,j-1)
2.3.1 Local communication (cont)

Envisage channels:

Each task needs to receive a value from its four neighbours in order to update its own value.
Each task must also send its own current value to each of its four neighbours for their computation.

Task logic:

for $T=0,T-1$
  send $x(I,j)$ -> 4 neighbours
  receive from each neighbour <- $x(i+1,j),x(i-1,j),x(i,j+1),x(i,j-1)$
  compute $x(i,j,t+1)$ from these values
end for

This is also a great example of where optimal sequential and parallel computational techniques may be different:

In regular computing, would you use Jacobi iteration?

No! There are better strategies, e.g. Gauss-Seidel iteration:

Better in the sense that the iteration converges faster (less iterations)

Minor change:

If you are updating lexicographically, then $x(i-1,j)$ and $x(i,j-1)$ have already been updated to new information, so should use that (each iteration should be an improvement so use best available)
2.3.1 Local communication (cont)

Jacobi iteration can have all $N \times N$ grid updated simultaneously (since all previous times exist on whole grid before update)

Gauss-Seidel iteration cannot update all simultaneously:

Cannot update a point until the points above and to the left are finished

No. of points available for concurrent update starts at 1, goes up to $N$ and back down to 1

$\Rightarrow$ on average can $\sim N/2$ concurrently. Not so good (compared to $N^2$ for Jacobi!)

FORTUNATELY …

There are other strategies that converge faster than Jacobi and yet are more parallelisable …

---

e.g. Red-Black (checkerboard) iteration:

Don’t think lexicographically -- too sequential!

Update all the odd-numbered elements concurrently

Then update all the even numbered elements concurrently

$\sim N^2/2$ concurrency

Faster convergence than Jacobi, more parallel than Gauss-Seidel (best of both worlds!)

So, the important point …

Choice of solution strategy plays important role in determining performance of parallel program.

What may be the best choice for sequential program, may not be ideal for the parallel program.
2.3.2 Global communication

Global communication = a communication in which many tasks must participate

e.g. parallel reduction operation

Reduces N values distributed over N tasks e.g. summation

\[ S = \sum_{i=0}^{N-1} X_i \]

Assume a single manager task does the job:

Communication structure:

```
0 --\   --S --\   --\   --\ |
\ /     \ /     \ /     \ /   \|
1     2     3     4     5     6     7
```

Task logic:
Workers send their numbers to central manager
Central manager sums the numbers

Good algorithm?

No! Since central accumulator can only receive and sum one value at a time, it takes \( O(N) \) time to sum N numbers = not very parallel!

**Problems of this algorithm:**

- Algorithm is centralised: it does not distribute computation and communication evenly; the manager task is involved in every operation.
- The algorithm is sequential: it does not allow computations and communications to proceed concurrently

Must address these problems to develop good parallel algorithm for this simple function ...

Distributing computation and communication:

e.g. make a task-channel structure which is

Task \( i = \) sum my \( x(i) \) with current sum and send to the left

```
0 --\   --\   --\   --\   --\   --\ |
\ /     \ /     \ /     \ /     \ /   \|
\Sigma_0 \Sigma_1 \Sigma_2 \Sigma_3 \Sigma_4 \Sigma_5 \Sigma_6 \| \chi_i \|
```

Now have nicely distributed computations and communications (one comp, one comm per task)

BUT STILL NO CONCURRENCY! Still \( O(N) \) steps ...

(but could be good if doing lots of sums: pipeline from right, so eventually pipe full = concurrent comp and comm)
2.3.2 Global communication (cont)

To achieve concurrency ... **DIVIDE AND CONQUER!**

As always, try and divide problem into sub-problems.

In this case, assume $N=2^n$ and divide sum in half

\[
\sum_{i=0}^{2^n-1} = \sum_{i=0}^{2^{n-1}-1} + \sum_{i=2^{n-1}}^{2^n-1}
\]

These two sums can be performed concurrently.

Extrapolate this ...

Number of levels to achieve sum = Height of tree = $\log_2 N$ ($= n$)

$\Rightarrow O(\log N)$ not $O(N)$ steps to complete $\Rightarrow$ concurrency!

Still $O(N)$ communications ...

Success! **Efficient parallel algorithm!**

- Distributed the $N-1$ computation and communication operations
- Modified the order to allow concurrency
- Regular communication structure (balanced)
- Each task communicates mostly with a small set of neighbours (locality)
2.3.3 Unstructured and dynamic communication

Previous examples all static and structured => tasks communication structure forms a regular pattern (grid, tree) and does not change over time.

Can be much more complex!

- grids may take the shape of a complex object or be refined in critical regions
- Refinement may vary with time (AMR = adaptive mesh refinement)

E.g. finite-element grids in engineering

Does not necessarily prove to be a problem at early stages of design (partition, communication)

E.g. easy to still define task for each vertex in finite-element grid and to require communication at each edge

Will however cause problems at later stages of design - agglomeration and mapping

Need tasks of roughly equal computation and communication and this is difficult with such irregular objects

Agglomeration may need to be highly dynamic => costs

2.3.4 Asynchronous communication

Assumed previously, synchronous => all senders and receivers are aware of comm event and senders explicitly send to receivers.

Asynchronous => tasks that produce data (producers) are not able to determine when other tasks require data (consumers); consumers must request data from producers.

Commonly occurs when tasks periodically read/write a shared data structure.

Assume data structure too large (or too frequently accessed) to be encapsulated in a single task.

Require it to be distributed and yet supporting asynchronous read/writes on its components.

Possible ways to do this:

1. Data structure distributed amongst computational tasks. Each task performs its own computation and requests data located in other tasks. Also periodically halts computational task and POLLS for pending requests on it's own data. (Convoluted programming, polling expensive => trade: expense of frequent vs. rapid response to poll events)

2. Distributed data structure encapsulated in a SEPARATE set of tasks, responsible only for read/write requests (More modular, loose locality - there is no local data => extra comm)

3. In a shared memory paradigm: no problem! All done automatically. However, must watch for order of operations (synchronisation), cache coherence etc
2.3.5 Communication design checklist

✓ Do all tasks perform about the same number of communication operations? Unbalanced communication requirements suggests non-scalable. Revisit design to distribute more equitably. e.g. if there exists a frequently-accessed data structure encapsulated in one task, consider distributing it (if large) or replicating it (if small).

✓ Does each task only communicate with a small number of neighbours? If each task communicates with many, see if it is possible to reformulate global communication into local (e.g. like “pairwise interactions” exercise or “divide and conquer”).

✓ Are communication operations able to proceed concurrently? If not, algorithm is likely inefficient and non-scalable. Try “divide and conquer” again to uncover concurrency?

✓ Can the computation proceed concurrently or is it blocked by the communication (e.g. Gauss-Seidel)? If not, losing efficiency and scalability again. Can you re-order the computation and communication and “unblock” it? Notice, this might require a complete revision of the algorithm in use (e.g. change Gauss-Seidel to red-black).

2.4 Agglomeration

We have

✓ partitioned computation into a set of tasks
✓ introduced communication to provide the required data for the tasks

Implementation is currently abstract - not specialised for efficient execution on any particular parallel machine.

In fact, may be highly inefficient if, e.g., there are many more tasks than processors if machine is not designed for efficient execution of small tasks.

Therefore, we begin a 3rd stage where we move from the abstract towards the more concrete.

We will revisit decisions made about partitioning and communication with a view to obtaining an algorithm that will execute efficiently on some class of parallel machine.

In particular:

✓ Consider if it is useful to combine, or AGGLOMERATE, tasks identified in the partitioning phase so as to provide a smaller number of tasks each of greater size.
✓ Determine whether it is worthwhile to replicate some data and/or computation.
2.4 Agglomeration (cont)

Agglomeration may reduce the number of tasks, but we may still end up with more tasks than processors. In this case, our design remains somewhat abstract still, and the MAPPING of tasks to processors is still required.

Alternatively, at this stage we may aim for one task per processor if we are intending to work in SPMD mode in which case the mapping is largely done at the same time.

So here we will concentrate on the GRANULARITY issue ...

There are THREE (somewhat) CONFLICTING GOALS:

- Reduce communication costs by increasing computation and communication granularity
- Retain flexibility with respect to scalability and mapping decisions
- Reduce software engineering costs

We will now discuss these three issues ...

2.4.1 Increasing granularity

In partitioning phase, focused on defining as many tasks as possible. Useful because it forces us to consider wide range of possibilities for parallel execution.

However, defining a large number of fine-grained tasks is not necessarily efficient.

Generally have to stop computation to communicate. We would rather be computing than communicating! Therefore, try and reduce amount of time spent communicating.

How can we reduce communication time/costs?

- Send less data: cost is proportional to data length
- Send same amount of data, less messages: also a fixed startup cost incurred sending messages
- We may also want to reduce the task creation costs.

If the number of communication partners is small, we can often reduce both the number of communication operations and the total communication data volume by increasing the granularity of our tasks => agglomerating several tasks into one
2.4.1 Increasing granularity (cont)

Examples of agglomerating tasks:

(a) 

(b) 

(c) 

(d)