Chapter 4: Putting Components Together

So far, concentrated on deriving efficient parallel algorithms for individual components of a program.

Complete programs (BIG CODES!) may incorporate multiple parallel algorithms each requiring different data structures, different partitioning/communication/mapping etc.

The complexity of this can be kept under control by modular design practices.

Key concept: encapsulate complex or changeable aspects of the design inside separate program components, or modules, with well-defined interfaces indicating they interact with their environment.

Programs are constructed by plugging together, or composing, these modules.

Modular design increases reliability and reduces costs by

✓ Making code easier to build
✓ Making code easier to change
✓ Enables greater code re-use

Goals of this chapter:

Introduce design issues that arise when developing large parallel codes

Understand principles/benefits of modular design

▪ what needs to go in modules
▪ how you compose them
▪ what are the performance issues
4.1 Review of modular design

**Basic concept:** To organise a complex system as a set of distinct components that can be developed independently and then plugged together.

Seems simple, but effectiveness depends critically on the manner in which the system is divided into components and the mechanisms used to plug them together.

**Design principles relevant to parallel programming:**

**Provide simple interfaces:** Simple interfaces reduce the no. of interactions that must considered when verifying that a system performs its intended functions. Also make it simpler to re-use (re-use = #1 software engineering cost-saver!)

**Ensure that modules hide information:** The benefits of modularity do not automatically come from sub-division of a program. The way in which the program is decomposed can make an enormous difference to to how easily the code can be implemented and modified. Each module should encapsulate information that is not necessary to the rest of the program. e.g. module may encapsulate

- Related functions that benefit from a common implementation or that are used frequently
- Functionality that is likely to change during later design or deployment
- Aspects of a problem that are particularly complex
- Code that is expected to be re-used in other programs

**Use appropriate tools:** Most modern languages support the required tools e.g. procedures, subroutines, functions, locally-scoped variables, user-defined data types, dynamic array allocation, …
4.1 Modular Design Checklist

✓ Does the design identify clearly-defined modules?

✓ Does each module have a clearly-defined purpose? (can you state it in one sentence?)

✓ Is each module’s interface sufficiently abstract that you do not need to think about the implementation of the module to understand it? i.e. does it hide it’s implementation details from other modules?

✓ Have you sub-divided modules as far as possible? (start fine-grained!)

✓ Have you verified that different modules do not replicate functionality?

✓ Have you isolated those aspects of the design that are hardware specific or complex or likely to change?

**e.g. database search:**

Modules –

input(in_file, database), search(database, search_pattern, matches), output(out_file, matches)

**Modular?**

Simple interface **BUT** if database internal structure changes, must re-write **ALL** modules **AND** each module probably replicates database access functions

Alternative modules: viewpoint - DB internal structure likely to be complex, change, common to many components => hide THIS in single database module containing e.g.

read_record(file, id, record), add_record(id, record, database), get_record(id, record, database), write_record(file, id, record)
Modularity principles apply directly to parallel programming. However, as usual, parallelism introduces extra complexities:

- Sequential modules encapsulate the functionality required by the interface and the data structures accessed by those functions.

- Parallel modules must encapsulate not only code and data but also tasks created by modules, partitioning of data structures, mapping to processors, internal communication structures …

For sequential programming, the modules can only be composed in one way: sequential = Sequence of calls to functions defined in modules

For parallel programming, 3 ways:

- **Sequential composition**: Still applicable. Common for SPMD mode.

- **Parallel composition**: different modules execute concurrently on disjoint sets of processors. Can enhance modularity, improve scalability and locality.

- **Concurrent composition**: different modules execute concurrently on the same processors. Can reduce design complexity and allow overlap of communication and computation.

Different ways of thinking about programs

Not all tools support all modes: Data parallel (e.g. HPF) => sequential only. MPI => sequential or parallel but not so much concurrent.
4.2.1 Data distribution

Data distribution was a key element for the efficiency of any particular parallel algorithm.

Data distribution can become a complex issue when a program is constructed from several components:

Optimal distributions may be different for each module.

Therefore when composing modules into a program:

- Modules must be modified to use different distributions
- OR data must be explicitly redistributed between calls to different modules

Different performance characteristics for each of these approaches, different development costs, etc. *(Look at this in detail in case study later)*

Would be nice to develop data distribution neutral modules:

- Modules can deal with a variety of data distributions
- Specify the distribution as a runtime parameter (or in data structure itself)
- Can be complex! Different data distributions may require different algorithms in the module!
4.2.2 Sequential composition

Most likely to encounter this in SPMD mode.

In sequential composition, each processor runs the same program which performs a sequence of calls to different program components.

The program components communicate and synchronise but they cannot create new tasks => entire program moves sequentially from one parallel section to another.

e.g.

```
While (not done) Do
    finite_difference(local_grid, local_max)
    global_maximum(local_max, global_max)
    if (global_max < threshold) then done = true
End Do
```

This entire program would be executed by each task => SPMD

Sequential composition of 2 procedure calls and 1 conditional statement for each step:

- **Finite_difference:**
  - Advances simulation on its local part of the grid
  - Communicates if/when required
  - Returns updated values on local grid
  - Returns error estimate

- **Global_maximum:**
  - Obtain global maximum error to see if computation converged
  - Communicates

Notice: communication is hidden
**4.2.2 Sequential composition** (cont)

*Big advantages of sequential/SPMD:*

Each process has a fairly straightforward sequential reading: this is the beauty of the sequential composition/SPMD combo!

Many sequential techniques can be retained:

- *Finite_difference* can be defined in a separate “grid” module
- *Global_maximum* can be defined in a separate “reduction” module

Both encapsulate internal data and communication structures

If the data distribution is fixed, no data movement (and no communication) at module interfaces: grids can be just initialised at the beginning.

Notice that the library of parallel calls could then have the very similar interfaces as the sequential program calls (if somewhat data layout neutral) – then the parallel implementation is hidden -- good for testing, code re-use, …

Not surprisingly, sequential composition has limitations for parallel programming …
4.2.3 Parallel composition

A parallel composition specifies which program components are to execute in which part of the machine and how these components exchange data.

This can be considered a generalisation of the previous SPMD structure to where different parts of the computer execute different programs – MPMD.

Can also be considered as a special case of the concurrent composition too where some tasks are idle on certain subsets of the processors.

In principle, any parallel composition can be expressed as a sequential composition that interleaves the program components appropriately.

However, the use of parallel composition can enhance scalability and locality
e.g. task decomposition problems like atmosphere-ocean model:

If atmosphere and ocean parts can execute concurrently then map to disjoint sets of processors => increased scalability.

Furthermore, if locality increases with granularity, then this enforced increase in granularity can make better use of cache, memory, communication bandwidth …

Also reduces code replication if there is some compared to sequential composition
4.2.4 Concurrent composition

The most general form.

A concurrent composition specifies the program components that are to execute concurrently, the communication relationships between them (producer/consumer), and the mapping of the components to the processors.

Really the higher-level (granularity) version of the task/channel model.

- Has abstract simplicity.

- Components execute in a data-driven manner: they execute if their data is available. Execution order need not be specified by programmer.

- Can facilitate information hiding and the building of modular programs. This is because the interfaces consist entirely of the channels (from the task/channel model) connecting the components. All they need is data. Components contain all other details and can be built entirely independently.

- Semantics of program independent of mapping to processors => mapping decisions for tasks can be delayed until very late in the design process (a good thing, remember!).

- Disadvantage: expense of switching between tasks in a processor, i.e. cost of data-driven execution.
4.2.4 Concurrent composition (cont)

e.g. finite-difference as a concurrent composition:

Two tasks “grid” and “reduce” running on all processors:

**Grid module:**
Procedure grid (to_reduce_module, from_reduce_module)
While (not done) do
  exchange_with_neighbours(local_grid)
  compute(local_grid,local_max)
  send(to_reduce_module,local_max)
  other_computation(local_grid)
  receive(from_reduce_module,global_max)
  if (global_max < threshold) then done=true
End Do
End

**Reduce module:**
Procedure reduce (to_grid_module, from_grid_module)
While (not done) do
  receive(from_grid_module, local_max)
  compute(global_max,local_max)
  send(to_grid_module,global_max)
  if (global_max < threshold) then done=true
End Do
End

**Notice:**
Tasks “grid” and “reduce” running concurrently, in “data-driven” mode.
Overlap of computation with communication
4.2.5 Design rules

- Design modules to handle multiple data distributions.
- Incorporate data distribution info into data structures (rather than interfaces).
- Use sequential composition for SPMD (e.g. with MPI).
- Consider sequential composition when program components cannot execute concurrently or must share a lot of data.
- Consider concurrent composition if program elements can execute concurrently, and/or communication costs are high and computation/communication overlap is possible.
- Consider parallel composition if memory is a problem or if locality is very important (intra-component communication costs are higher than inter-component costs).
4.2.6 Performance analysis

*Brief synopsis: then we will do case studies*

Could be easy

e.g. for **sequential composition** of two components that do not require a data redistribution, could simply be the sum of the performance models of the components.

However, performance analysis of composed programs can of course be complex.

*Some thoughts:*

*Idle time*

-- can be significantly reduced in concurrent composition

-- parallel composition can increase idle time if not well load-balanced.

*Communication time*

-- increases for **parallel composition** (moving between disjoint sets)

-- increases for **sequential composition** if must redistribute data

*Locality*

-- Parallel composition tends to increase granularity ⇒ increased locality
4.4 Case study: Convolution

Need to do a **convolution on two sequences of images**.

- Convolution = nasty operation in real space
- Convolution = easy operation in Fourier space!

⇒ FFT both images, multiply both images pointwise (=convolution), FFT back

Three obvious components for composition: **FFT, pointwise multiplication, inverse FFT**

**STEP 1: How to parallelise the components:**

**STEP 1A: Pointwise multiplication**

Easy. Zero communication as long as both arrays have same data distribution

**STEP 1B: FFTs** (forward and inverse basically the same)

Image = 2D array ⇒ FFT is a 2D FFT. What is a 2D FFT?

⇒ 1D FFT on rows first, then 1D FFT on columns

⇒ 1D FFT is a butterfly operation

**Two possible ways of parallelising:**

(a) Very fine-grained: exploit concurrency WITHIN a 1D FFT. Complex: lots of communication.

(b) Coarse-grained: do complete 1D FFTs concurrently. No internal communication only between row and column calls. Better. Choose this.
4.4 Case study: Convolution (cont)

Two possible coarse-grained FFT algorithms: transpose and pipeline

**Transpose:**
Row FFTs first, partitioned $N_x/P$

**Transpose**
Column FFTs second, partitioned $N_y/P$

**Pipeline:**
Concurrently do:
Row FFTs for one image
Column FFTs for previous image

**QUESTION:** Type of composition here?
Transpose is a sequential composition within this component, whereas pipeline is a parallel composition.
4.4 Case study: Convolution (cont)

**Communication costs** for an N x N image:

**Transpose:**

Each processor sends one message of size one Pth of \((N/P \times N)\) to all \((P-1)\) other processors =>

\[
T_{\text{transpose}} = P(P - 1)(t_s + t_w \frac{N^2}{P^2})
\]

**Pipeline:**

Two sets of processors of size \(P/2\)

Each processor in first set must communicate with each processor in second set

⇒ no. of messages = \(P/2 \times P/2 = P^2/4\)

Size of message = one \((P/2)\)th of 1 image/ \((P/2) = N^2/(P/2)^2\) =>

\[
T_{\text{pipeline}} = \frac{P^2}{4} (t_s + t_w \frac{N^2}{P^2}) = t_s \frac{P^2}{4} + t_w N^2
\]

Pipeline sends significantly fewer messages => should be better if \(P, t_s\) large or \(N, t_w\) small?

Otherwise, transpose has lower data transfer amounts so better if \(N\) large \((N^2 > P; \text{notice max } N = P\) for these algorithms).
4.4 Case study: Convolution (cont)

**STEP 2: Composing the components:**

2A. Sequential composition

For each image

\[ A_1 = \text{FFT}(A, \text{DIR}=+1) \]
\[ B_1 = \text{FFT}(B) \]
\[ C_1 = A_1 \times B_1 \]
\[ C = \text{FFT}(C_1, \text{DIR}=-1) \]

Endfor

Communication costs are 3* single FFT cost

\[ T_{\text{sequential}} = 3T_{\text{transpose}} \]

2B. Parallel composition

Do the 2 forward FFTs and the inverse FFT concurrently (*BUT compose the multiplication sequentially with the FFTs since computational cost FFT = N^2 \log(N) whereas the multiplication = N^2*)

⇒ P/3 processors available for each of the FFT operations

Must shift data from 2 forward FFT processors to inverse FFT processor

\[ T_{\text{extra, parallel}} = 2 \frac{P}{3} (t_s + t_w N^2) \]

⇒ Total comm time

\[ T_{\text{parallel}} = T_{\text{transpose / pipeline}} (P = P / 3) + 2 \frac{P}{3} (t_s + t_w N^2) \]
4.4 Case study: Convolution (cont)

\[
T_{\text{sequential/transpose}} = 3(t_s P(P-1) + t_w N^2 \frac{(P-1)}{P})
\]

\[
T_{\text{parallel/transpose}} = t_s \left(3 \frac{P}{3} \left(\frac{P}{3} - 1\right) + 2 \frac{P}{3}\right) + t_w N^2 \left(3 \frac{\left(\frac{P}{3} - 1\right)}{3} + 2\right)
\]

\[
T_{\text{parallel/pipeline}} = t_s \left(3 \frac{P}{4} + 2 \frac{P}{3}\right) + t_w N^2 (3 + 2)
\]

Yikes!
### 4.4 Case study: Convolution (cont)

<table>
<thead>
<tr>
<th>Composition</th>
<th>2D FFT</th>
<th>Messages</th>
<th>Data volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Transpose</td>
<td>FFT: $3P(P-1) \sim 3P^2$</td>
<td>FFT: $3N^2(P-1)/P \sim 3N^2$</td>
</tr>
<tr>
<td>Parallel</td>
<td>Transpose</td>
<td>FFT: $3\times P/3(P/3-1) \sim P^2/3$</td>
<td>Par Comp: $2P/3$</td>
</tr>
<tr>
<td>Parallel</td>
<td>Pipeline</td>
<td>FFT: $3\times(P/3)^2/4 \sim P^2/12$</td>
<td>Par Comp: $2P/3$</td>
</tr>
</tbody>
</table>

Parallel composition ⇒ fewer messages but more data movement ⇒ better on smaller problems or larger P

**Conclusions:**

Even simple problem can have design trade-offs

Maybe best design would adjust algorithm according to N and P!
4.5 Case study: Matrix multiplication

A, B = NxN matrices: matrix multiplication =>

N operation per element of $C_{ij}$ =>

$O(N^3)$ operations total

Maybe desire a library routine that will work for

✓ Row partitioning of A, B, C across processors
✓ Column
✓ Row and column

Basic question:

✓ Different algorithms for different partitioning?
✓ OR convert incoming data to one standard distribution on entry?

Examine algorithms for the various distributions …
1D column-wise partitioning

**Tasks:** Processor \(P_j\) is responsible for \(C_{i,j}\) -- the \(j^{th}\) column (black in diagram)

**Channels:** \(N^2/P\) data are required from all other (\(P-1\)) processors (grey) =>

\[
T_{\text{comm}_{-1D}} = (P - 1)(t_s + t_w \frac{N^2}{P}) \sim t_s P + t_w N^2
\]

Each task does \(N^3/P\) of the work => \(T_{\text{comp}_{1D}} = t_c N^3/P\)

Note that if \(N \sim P\) then \(T_{\text{comp}_{1D}} \sim T_{\text{comm}_{1D}} \sim N^2\)

i.e. one piece (word) of data transferred per computation (one mult and one add)

⇒ Not very efficient!

⇒ require \(N \gg P\) so \(T_{\text{comp}_{1D}} \gg T_{\text{comm}_{1D}}\)
4.5 Case study: Matrix multiplication (cont)

**2D partitioning**

**Tasks:** Processor \( P_{i,j} \) is responsible for \( C_{i,j} \) (black in the diagram)

**Channels:**

Requires all the data from \( i^{th} \) row of processors of \( A \) => from \( P_{i,*} \)

Requires all the data from \( j^{th} \) column of processors of \( B \) => from \( P_{*,j} \)

Amount of data per processor = \( \frac{N^2}{P} \) (again)

No. of processors in a row/col = \( \sqrt{P} \)

\[ \Rightarrow \text{Total data volume moved} = 2*(\sqrt{P}-1)\frac{N^2}{P} \sim \frac{N^2}{\sqrt{P}} \]

Note: this is considerably less than the \( \sim N^2 \) of the 1D algorithm (\( P>2 \)).
4.5 Case study: Matrix multiplication (cont)

Strategy for moving the sub-matrices around?

B1 = Blocal

For each \( j=0,\sqrt{P}-2 \) do

In each row \( I \),

\((i+j)\mod(p)\) th task broadcasts

\( A1=A\text{local} \) to the other tasks in row

Accumulate \( A1*B1 \)

Send \( B1 \) to upward neighbour

Endfor
Cost of moving the sub-matrices around?

Sqrt(P)-1 steps

For A1: Broadcast N^2/P data to sqrt(P)-1 tasks

For B1: Nearest-neighbour communication of N^2/P data

⇒ Per task:

\[ T_{comm\_2D} = (\sqrt{P} - 1)(\log(P) + 1)(t_s + t_w \frac{N^2}{P}) \]

Since

\[ T_{comm\_1D} \sim t_s P + t_w N^2 \]

\[ \sqrt{P} \log(P) < P \]

\[ \frac{\log(P)}{\sqrt{P}} < 1 \]

⇒ more efficient communication than 1D!
4.5 Case study: Matrix multiplication (cont)

Should our library routine therefore convert 1D partitioned arrays into 2D partitions?

What is the cost of re-organisation?

Original task must send data to $\sqrt{P}$-1 other processors

Amount of data = $N/P$ (original width) \* $N/\sqrt{P}$ (chunk to send) = $N^2/P\sqrt{P}$

$$T_{\text{re-org}} = (\sqrt{P} - 1)(t_s + t_w \frac{N^2}{P\sqrt{P}}) \sim \sqrt{P} t_s + t_w \frac{N^2}{P}$$

$\Rightarrow$ Total cost for re-org + 2D version:

$$T_{1D\rightarrow2D} = 3(\sqrt{P} t_s + t_w \frac{N^2}{P}) + (\sqrt{P} \frac{\log(P)}{2} t_s + t_w N^2 \frac{\log(P)}{2\sqrt{P}})$$

3 arrays to be re-org

$$T_{1D\rightarrow2D} = (3\sqrt{P} + \sqrt{P} \frac{\log(P)}{2}) t_s + t_w (3\frac{N^2}{P} + N^2 \frac{\log(P)}{2\sqrt{P}})$$
4.5 Case study: Matrix multiplication (cont)

This will be more efficient than remaining with 1D when

\[ T_{1D} > T_{1D\rightarrow 2D} \]

\[ Pt_s + t_w N^2 > (3\sqrt{P} + \sqrt{P} \frac{\log(P)}{2}) t_s + t_w (3 \frac{N^2}{P} + N^2 \frac{\log(P)}{2\sqrt{P}}) \]

Turns out this holds for all but very small P

⇒ Should always convert!

⇒ Who’d have guessed?

(of course you can do better than this: different “systolic” data distribution ⇒ all nearest neighbours comms ⇒ reduce costs by factor of log(P)/4!)
4.6 Summary

Modular design is fundamental to good software engineering

- Simple interfaces
- Information hiding
- Try and abstract data distribution details from the interface
- Distinguish between sequential, parallel and concurrent composition
  - Sequential -- simple but inflexible
  - Parallel -- improves locality and scalability
  - Concurrent -- most general
- Performance models possible but care must be taken with module interfaces where there may be communication costs, and with overlapping computation and communication etc.