



EEE4120F

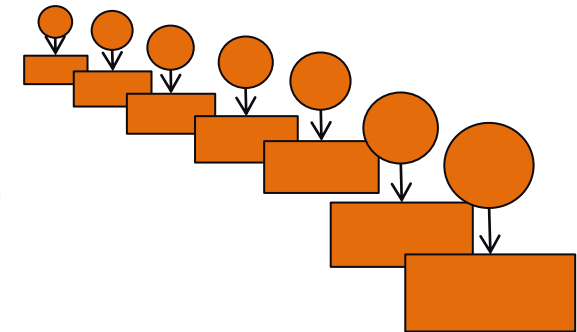
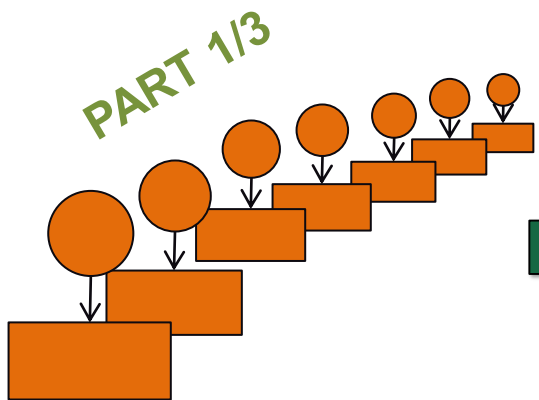


High Performance Embedded Systems

(lecture for double period if including slide 29 activity)

Lecture 9: Design of Parallel Systems

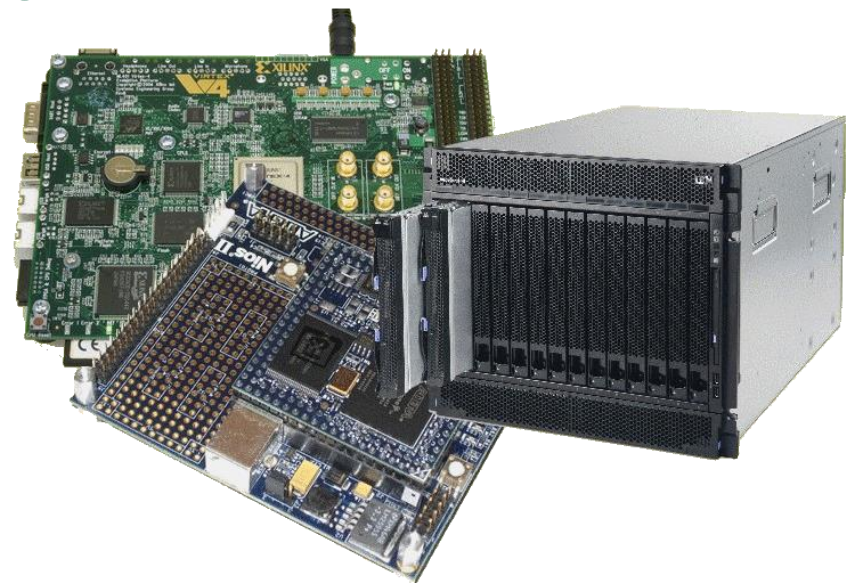
Presented by
Simon Winberg



Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

Lecture Overview

- Towards HPES systems design and programming
- Heterogeneous Computing Solutions
- Steps in parallelizing programs
 - Understand the problem
 - Partitioning
 - Granularity
 - Identify data dependencies



Toward HPES system design and programming

- We have looked into essential aspects of (single-core) processor design, and thinking about parallel systems.
- Thinking point regards parallel system design:
 - Does it make more sense to start on the design of a parallel computing solution by
Deciding the parallel platform/hardware first... or
Deciding the data and processing needs first?
- ... In this course, the direction is to start thinking about the **data and processing**, and then elaborating on methods and considering hardware constructs by which these are delivered.

... and that's why we will delve into

Steps in Designing Parallel Programs

Steps in Designing Parallel Programs

Starting the somewhat long and arduous trek on the topic of designing parallel programs



Hardcore competent HPES programmers (leading the way to greater feats)




Sequential programmers in their comfort zone.

The steps in designing parallel programs

The hardware may be done first... or later.

The main steps:

1. Understand the problem
2. Partitioning (separation into main tasks)
3. Granularity
4. Communications 
5. Identify data dependencies
6. Synchronization
7. Load balancing
8. Performance analysis and tuning



Step 1: Understanding the Problem

EEE4120F

Step 1: Understanding the problem

- Make sure that you understand the problem, that is *the right problem*, before you attempt to formulate a solution (i.e. 'problem validation')
- Some problems aren't suited to parallelization – it just might not be parallelizable

Example of non-parallelizable problem:

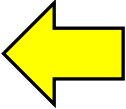
Calculating the set of Fibonacci series e.g., $\text{Fib}(10^{20})$ as quickly as possible.

$$\begin{aligned} \text{Fib}(n) &= n && \} \text{ if } n < 2 \\ &= \text{Fib}(n - 1) + \text{Fib}(n - 2) && \} \text{ otherwise} \end{aligned}$$

Step 1: Understanding the problem

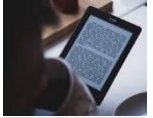
- Identify: Critical parts or 'hotspots' ←
- Determine where most of the work needs to be done. *NB:* Most scientific and technical programs accomplish the most substantial portion of the work in only a few small places.
- Focus on parallelizing hotspots. Ignore parts of the program that don't need much CPU use.
- Consider which profiling techniques and Performance Analysis Tools ('PATs') to use

Step 1: Understanding the problem

- Identify: **bottlenecks**
- Consider communication, I/O, memory and processing bottlenecks 
- Determine areas of the code that execute notably slower than others.
- Add buffers / lists to avoid waiting
- Attempt to avoid blocking calls (e.g., only block if the output buffer is full)
- Try to rearrange code to make loops faster

Step 1: Understanding the problem

- General method:
 - identify hotspots, avoid unnecessary complication, identify potential inhibitors to the parallel design (e.g., data dependencies).
 - Consider other algorithms...
 - This is an most important aspect of designing parallel applications.
 - Sometimes the obvious method can be greatly improved upon though some lateral thought, and testing on paper.



Prescribed
reading

Step 1: Identifying the Problem: where the solution may fit...

- This also brings in aspects of Lecture 1, 'the Landscape of Parallel Computing'*
- Considering the 7 questions:
 1. What are the applications?
 2. What are the common kernels?
 3. What are the hardware building blocks?
 4. How to connect them?
 5. How to describe allocations and kernels?
 6. How to program the hardware?
 7. How to measure success?

* The Landscape of Parallel Computing Research: A View from Berkeley” by Krste Asanovic, Ras Bodik, Bryan Catanzaro, *et al.*

Step 2: Partitioning

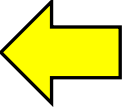
EEE4120F

Step 2: Partitioning

partitioning

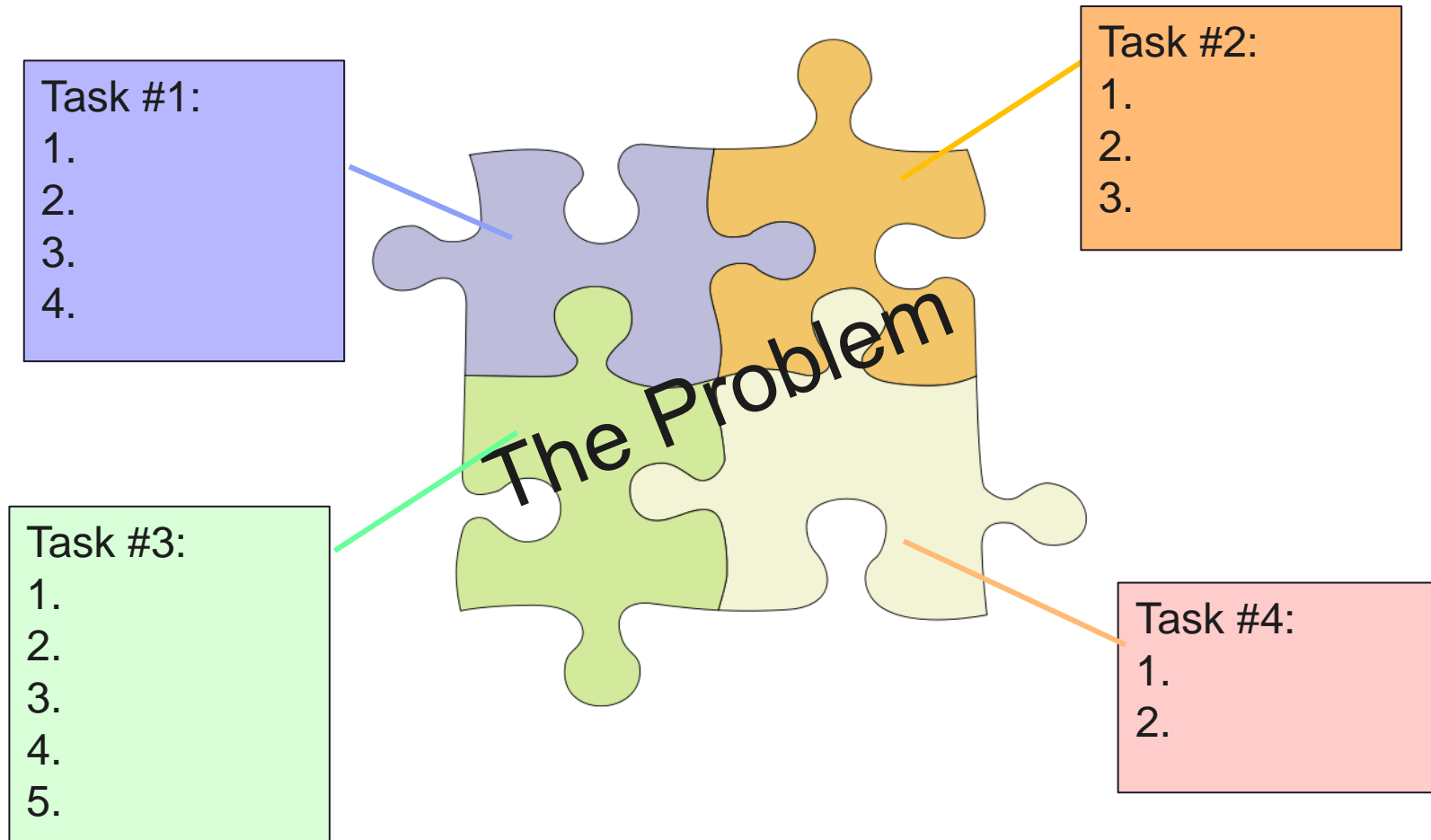


The
Problem

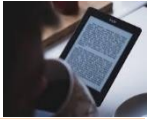
- This step involves breaking the problem into separate chunks of work, which can then be implemented as multiple distributed tasks. 
- Two typical methods to partition computation among parallel tasks:
 1. Functional decomposition or
 2. Domain decomposition

Functional decomposition

Decomposing the problem into tasks to be done.



Functional decomposition is suited to problems that can be split into different tasks



Prescribed
reading

Functional decomposition

- Example applications
 - Environment modelling
 - Simulating reality
 - Signal processing, e.g.:
 - Pipelined filters: $\text{in} \rightarrow P1 \rightarrow P2 \rightarrow P3 \rightarrow \text{out}$
 - Here P1 is filled first, its result is sent to P2 while simultaneously P1 starts working on a block of new input, and so on.
 - Climate modelling (e.g., simultaneously running simulations for atmosphere, land, and sea)

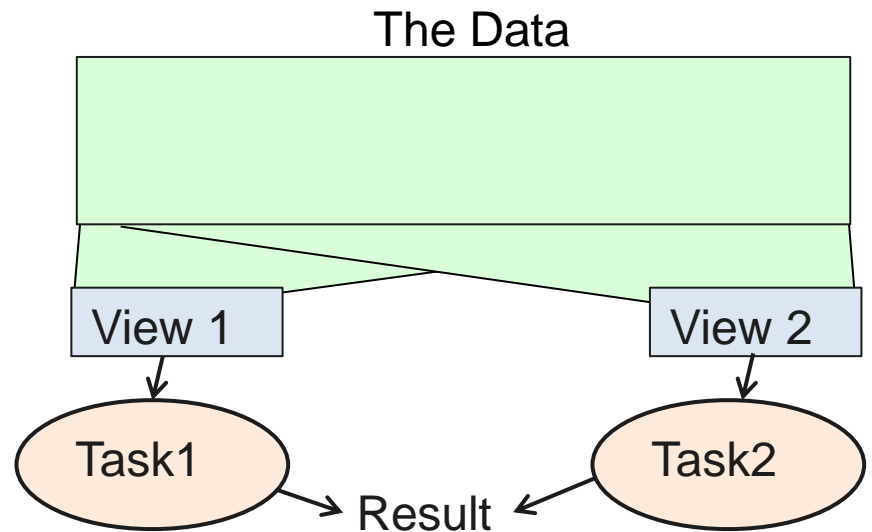
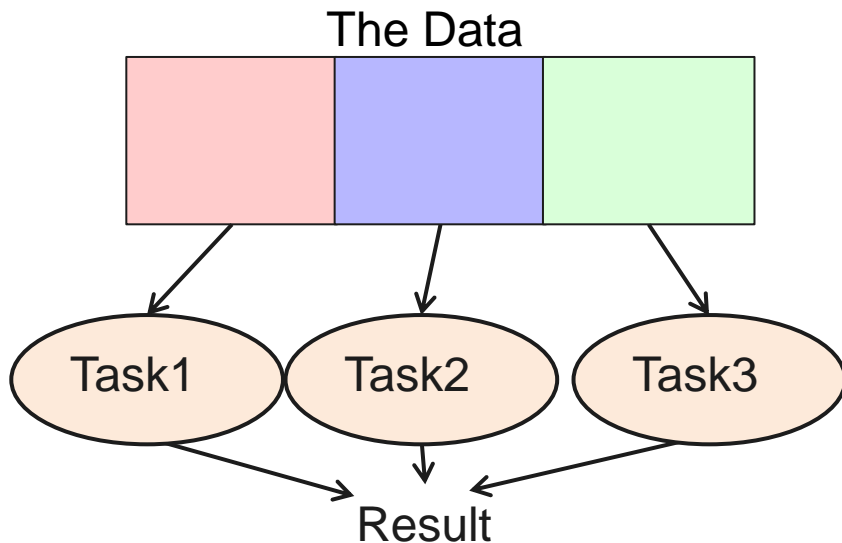
Domain decomposition

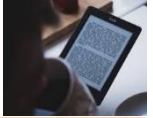
→ Involves separating the data, or taking different 'views' of the same data. e.g.

View 1 : looking at frequencies (e.g. FFTs)

View 2 : looking at amplitudes

Each parallel task works on its own portion of the data, or does something different with the same data





Prescribed
reading

Domain decomposition

- Good to use for problems where:
 - Data is static (e.g., factoring; matrix calculations)
 - Dynamic data structures linked to a single entity (where entity can be made into subsets) (e.g., multi-body problems)
 - Domain is fixed, but computation within certain regions of the domain is dynamic (e.g., fluid vortices models)

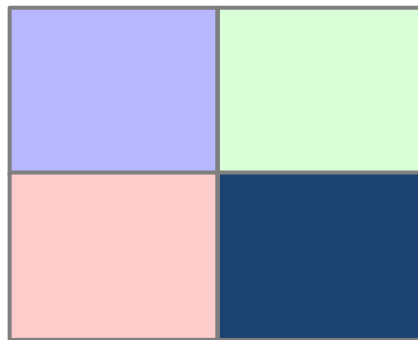
Domain decomposition

(terms introduced in Lecture 7)

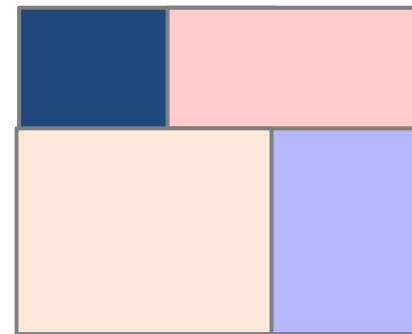
Many possible ways to divide things up. If you want to look at it visually...



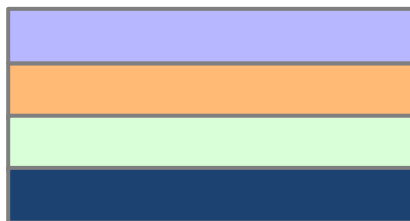
continuous



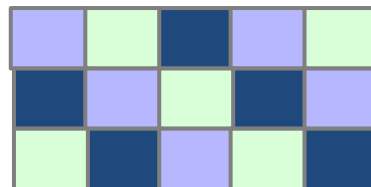
Blocked (same-size partitions)



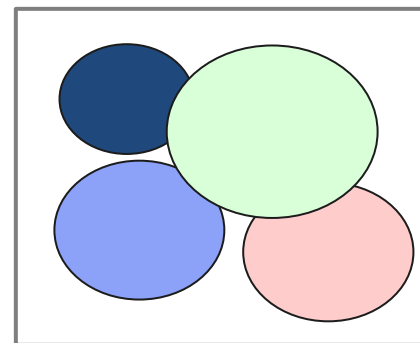
Blocked



Interlaced



Interleaved or cyclic



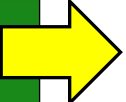
Other methods can be done

Step 3: Granularity

EEE4120F

Granularity of the Problem
vs.
Granularity of the Parallelism

Step 3: Granularity of the Problem

- 
- Granularity of the problem
 - How big or small are the parts that the problem has been decomposed into?
 - How interrelated are the sub-tasks

Fine Grained:

Each calculation highly dependent on other parts of the problem space. Requires **a great deal of communication**.

Coarse Grained:

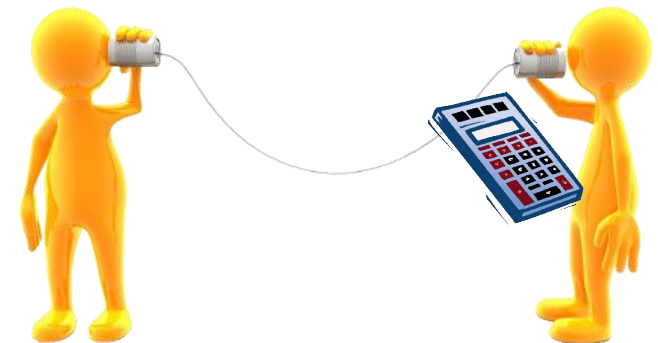
Each calculation largely independent of other parts of the problem space. Requires **only a little communication**.

 : marks key point(s)

See also explanation on <https://en.wikipedia.org/wiki/Granularity>

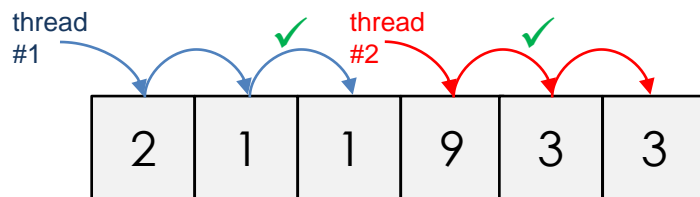
The Ratio of Computation : Communication

- This ratio can help to decide is a problem is a fine- or coarse-grained problem.
 - $1 : 1 =$ Each intermediate result needs a communication operation
 - $100 : 1 =$ 100 computations (or intermediate results) require only one communication operation
 - $1 : 100 =$ Each computation needs 100 communication operations



Decomposition and Problem Granularity

- Coarse grained:
 - Breaking problems into larger pieces
 - Usually, low level of task inter-dependence and fewer interrelations (e.g., can separate into parts whose elements are unrelated to other parts)
 - These solutions are generally easier to parallelize than fine-grained, and
 - Usually, parallelization of these problems provides significant benefits.



e.g. sequence search to find a number followed by its square root.

e.g. Computation:Communication = 100:1
(note: we're assuming sqrt will take much computation)

Decomposition and Granularity

- Fine-grained problem:
 - Problem broken into pieces that have high levels of inter-dependence. Or difficult to partition the data because each result is dependent on much of the available data.



e.g.: having to look at relations between neighboring dust particles to determine how a dust cloud behaves.

e.g. Computation:Communication = 1:100

*let's next consider:
granularity of problem vs
granularity of parallelism...*

Example of fine-grained problem, coarse-grained parallelism

- Fine-grained problem with coarse-grained parallelism:
 - Result(s) of the computation are highly dependent on much of the data... but can be done as many parallel operations.
- Example
 - Dot-product of two vectors : result is a single scalar value – so overall obtaining result is fine-grained problem as it depends on all the other values and their ordering, but can be sub-divided into many smaller tasks that provide partial results, so has coarse-grained parallelism.

$$\mathbf{a} \cdot \mathbf{b} = a_1 \times b_1 + a_2 \times b_2 + \dots a_n \times b_n$$

2	1	1	9	3	3
2	1	1	9	3	3

The result, a single value in this case, is fine-grained, depending on all the available data (but the computation is fairly coarse-grained, can be split-up into sub-tasks of $a_i \times b_i$ which are then summed)

Example of Coarse-grained

- Course-grained

- Result(s) of the computation dependent on only a small portion of the data

- Example

- Increment each element in a vector.

$$\mathbf{a} + 1 = [a_1 + 1, + a_2 + 1, \dots a_n + 1]$$

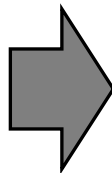
Each result, which happens to be n results in this case, depends on only one item of data.

Decomposition and Granularity

- Embarrassingly Parallel:
 - So coarse that there's no or very little interrelation between parts/sub-processes

NB: also called “embarrassingly fine-grained parallel” – about to hear why...

2	3	2	1	4	0
1	2	3	4	3	2
0	1	2	3	4	3
1	0	3	2	4	1
2	2	0	3	4	1



4	6	4	2	8	0
2	4	6	8	6	4
0	2	4	6	8	6
2	0	6	4	8	2
4	4	0	6	8	2

e.g. double all the entries in a matrix.

e.g. Computation:Communication = 1:0

Granularity of Parallelism

- Granularity of parallelism:
 - How small and plentiful are the tasks (computing parts or threads) the program is broken into.
 - It is more about the implementation.
 - Is not answering “how interrelated are the computations”
 - Is about: how small are the pieces of computing

Granularity of Problem



Granularity of Parallelism





Supplementary
reading

Homework task

- Which of the following are more fine-grained problems, and which are more coarse-grained problems?
 - Matrix multiply
 - FFTs
 - Decryption code breaking
 - (deterministic) Finite state machine validation / termination checking
 - Map navigation (e.g., shortest path)
 - Population modelling

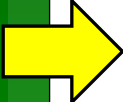


*In dog-think: I'm not
budging from here
until I'm done.
Woof!*

Step 5: Identify Data Dependencies

EEE4120F

Data dependencies

- 
- A **dependency** exists between statements of a program when the order of executing the statements affects the results of the program.
 - A **data dependency** is caused by different tasks accessing the same variables (i.e., memory addresses).
 - Dependencies are a major inhibition to developing parallel programs.

Data dependencies

- Common approaches to working around data dependencies:
 - For distributed memory architectures: tend to use **synchronization points** (periods when sets of shared data is communicated between tasks).
 - Shared memory architectures: make use of read/write **synchronize operations** (no sending of data, just temporarily block other tasks from reading/writing a variable).

Common data dependencies

- **Loop carried data dependence:**
dependence between statements in different iterations
- **Loop independent data dependence:**
dependence between statements in the same iteration
- **Lexically forward dependence:**
source precedes the target lexically
- **Lexically backward dependence:**
opposite from above
- **Right-hand side of an assignment precede the left-hand side**

Part 1/3 Done



1. Understand the problem
2. Partitioning
3. Granularity
4.
5. Identify data dependencies

Next two slides discusses solution to last week's microcoding homework assignment.

Over to you to work on OpenCL prac or have an early start on Prac3



Supplementary
reading

Further Reading / Refs

Suggestions for further reading, slides partially based / general principles elaborated in:

- https://computing.llnl.gov/tutorials/parallel_comp/
- http://www2.physics.uiowa.edu/~ghowes/teach/ihpc12/lec/ihpc12Lec_DesignHPC12.pdf

Some resources related to systems thinking:

- <https://www.youtube.com/watch?v=lhbLNBqhQkc>
- <https://www.youtube.com/watch?v=AP7hMdnNrH4>

Some resources related to critical analysis and critical thinking:

An easy introduction to critical analysis:

- <http://www.deakin.edu.au/current-students/study-support/study-skills/handouts/critical-analysis.php>

An online quiz and learning tool for understanding critical thinking:

- <http://unilearning.uow.edu.au/critical/1a.html>

Disclaimers and copyright/licensing details

I have tried to follow the correct practices concerning copyright and licensing of material, particularly image sources that have been used in this presentation. I have put much effort into trying to make this material open access so that it can be of benefit to others in their teaching and learning practice. Any mistakes or omissions with regards to these issues I will correct when notified. To the best of my understanding the material in these slides can be shared according to the Creative Commons “Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)” license, and that is why I selected that license to apply to this presentation (it’s not because I particulate want my slides referenced but more to acknowledge the sources and generosity of others who have provided free material such as the images I have used).

Image sources:

Clipart sources – public domain CC0 (<http://pixabay.com/>)

PxFuel – CC0 (<https://www.pxfuel.com/>)

Pixabay

commons.wikimedia.org

Images from flickr

