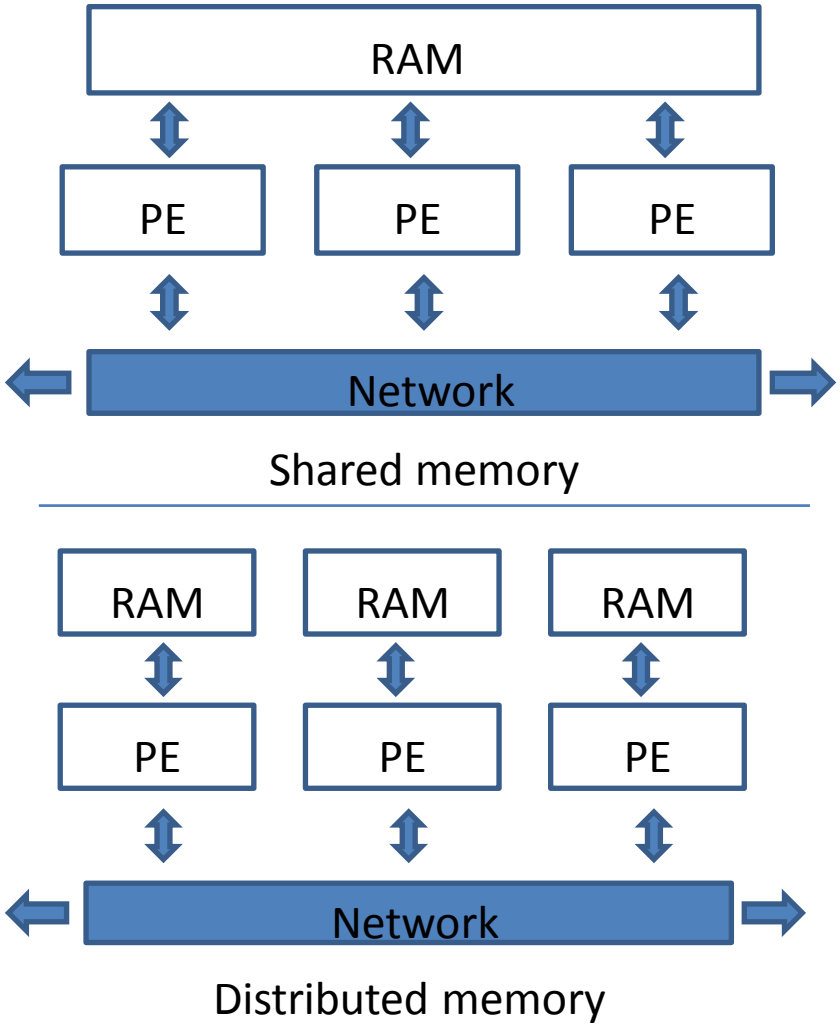
Multicore Challenge Conference 2012
UWE, Bristol

Multi/Many Core Programming Strategies

Greg Michaelson
School of Mathematical & Computer Sciences
Heriot-Watt University

Overview

- good old fashioned parallel computing based on lots of identical single CPUs is over



Overview

- Moore's Law implications have changed
  - speed of CPUs now stable at ~3.5 GHz
  - performance increases from multi- & many-core CPUs

Intel 4004 – 1971
http://en.wikipedia.org/wiki/Intel_4004

Intel Core I7 – 2008
http://en.wikipedia.org/wiki/Intel_Core_i7

Overview

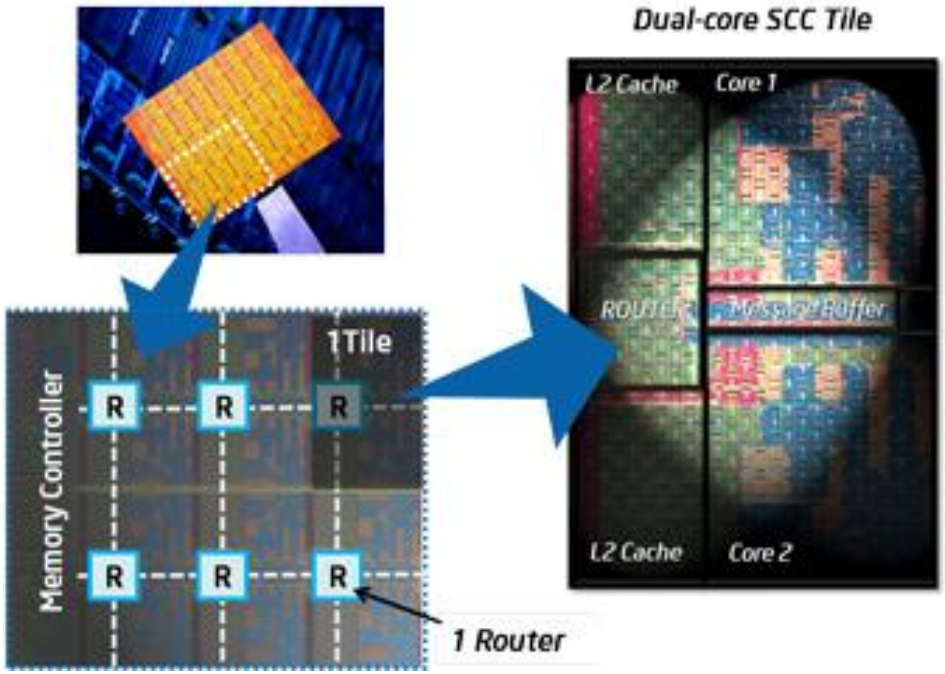
- multi-processor architectures increasingly hierarchical & heterogeneous
- message passing grids of clusters of:
  - now: shared memory multi-core

Hector – Edinburgh Parallel Computer Centre
- 464 compute blades with…
- 4 compute nodes with…
- 2 *12-core processors.
- 44,544 cores

http://www.hector.ac.uk/abouthector/hectorbasics/

Overview

- multi-processor architectures increasingly hierarchical & heterogeneous
- message passing grids of clusters of:
  - soon: message passing many-core arrays



SCC – Intel Research
http://techresearch.intel.com/ProjectDetails.aspx?Id=1

Overview

- cores also have SIMD processors (MMX/SSE)
- non-uniform memory
  - differing degrees/levels of private & shared cache
- old programming strategies break down
  - one size no longer fits all
- need for hybrid strategies

Overview

- developing multi-processor software is still a black art
- would like:
  - low effort
  - flexibility
  - scalability
  - future proof
  - re-use

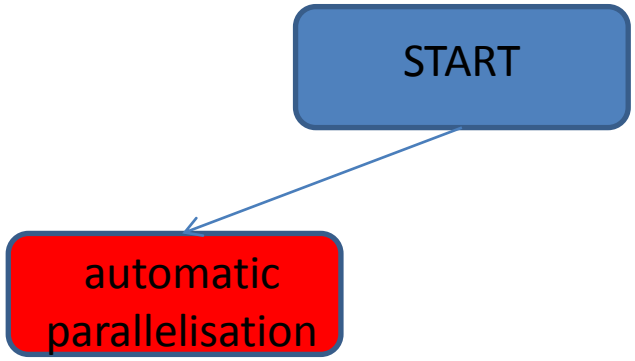
Overview

- different approaches:
  - require different effort
  - offer different degree of control over:
    - task division
    - communications
    - process placement

Methodological choices

START

Methodological choices



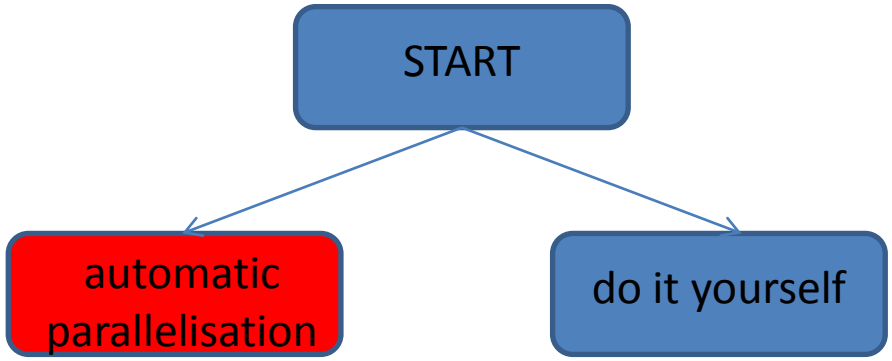
Automatic Parallelisation

- vector/array parallelisation
- implicit
  - e.g. SIMD in C with gcc
- language directives
  - Fortrans: Fortran 90; F; High Performance Fortran

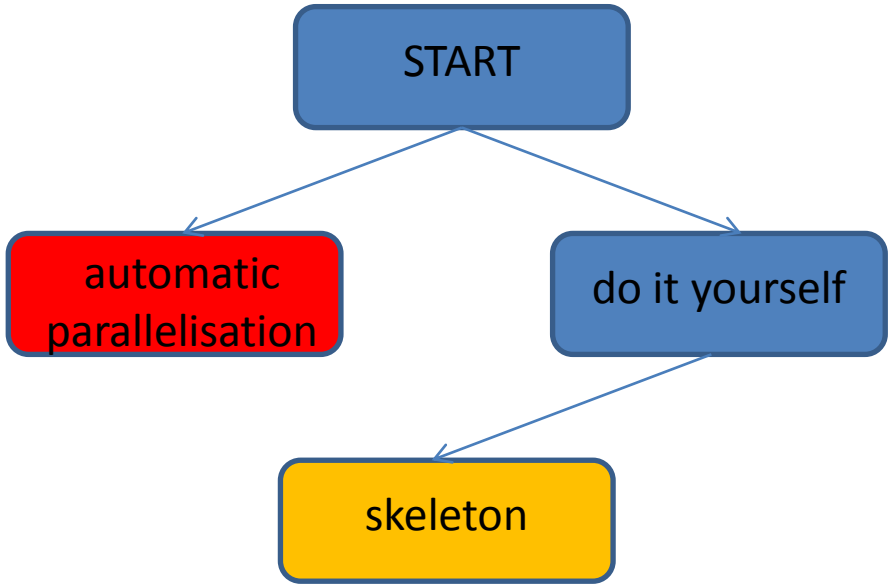
Automatic Parallelisation

- low effort
  - no communications
  - no/minimal task division
- poor flexibility/scalability
  - good for regular problems
  - good on uniform architectures

Methodological choices

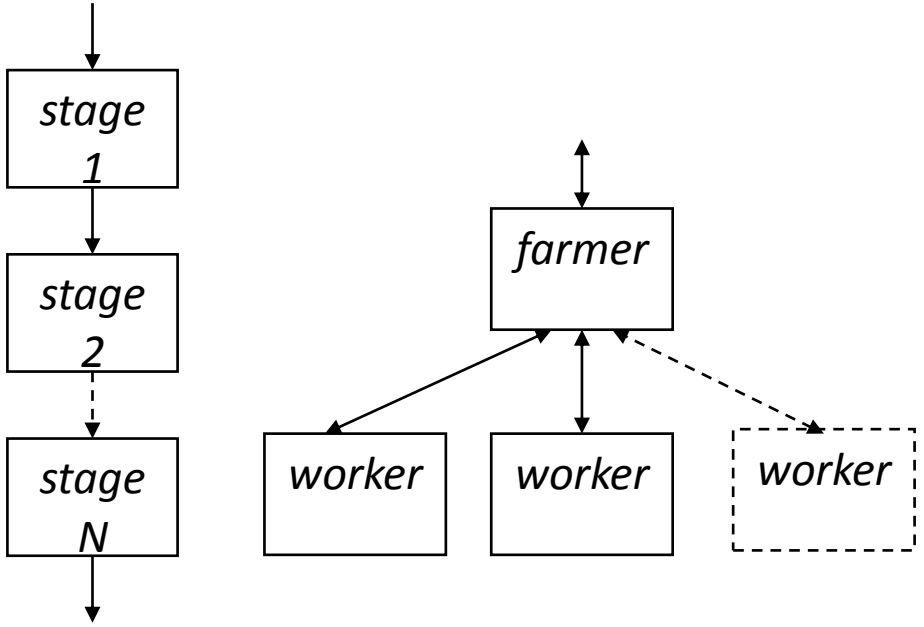


Methodological choices



Algorithmic skeletons

- capture common patterns of data & control parallelism
  - e.g. pipeline; farm; divide & conquer
- skeleton libraries for C/Java

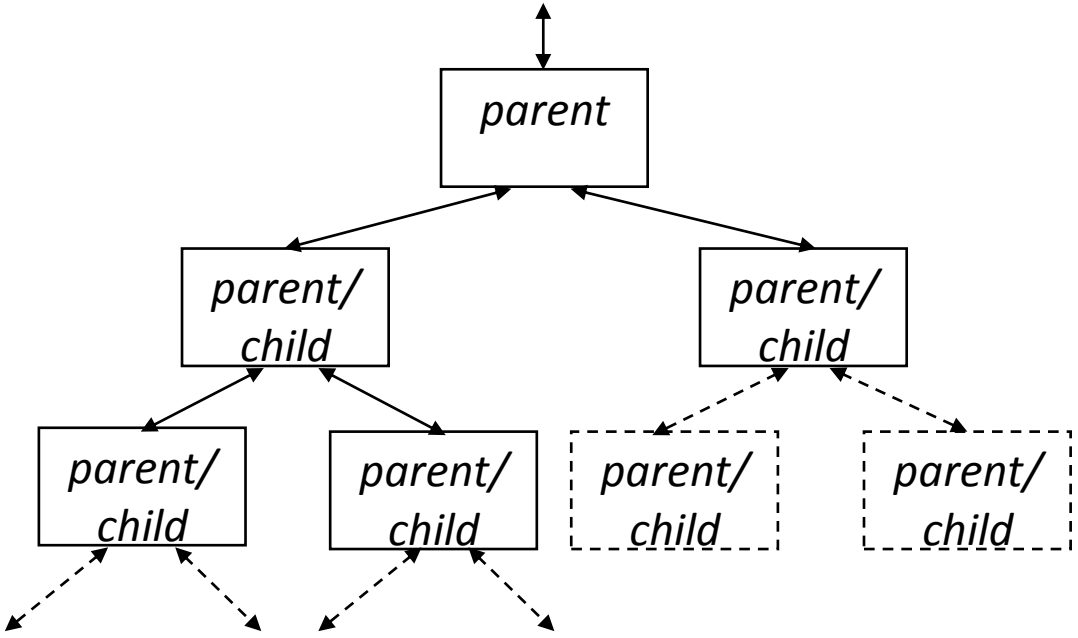


pipeline

process farm

Algorithmic skeletons

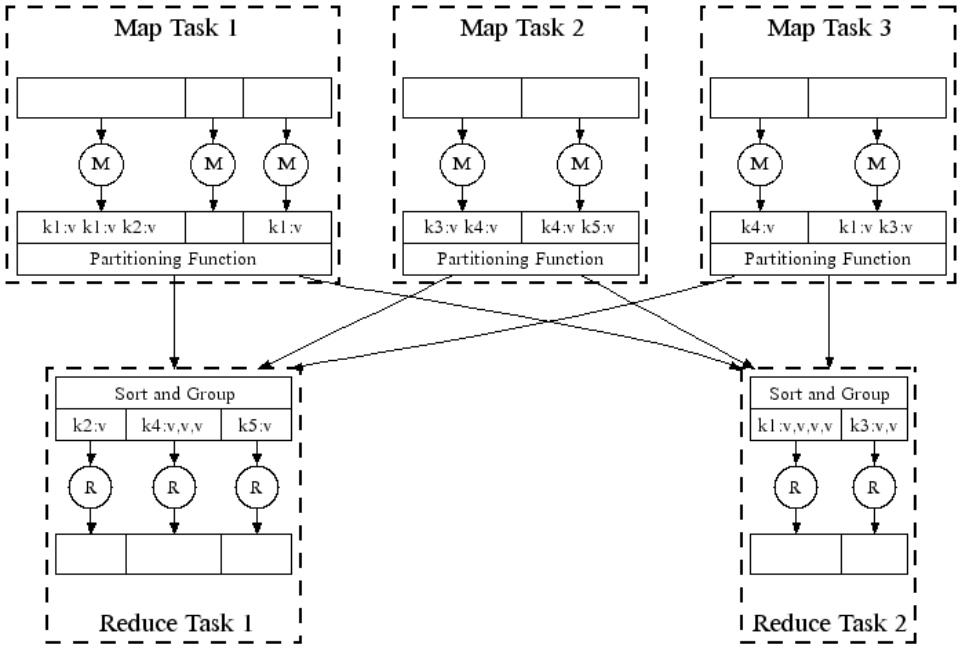
- capture common patterns of data & control parallelism
  - e.g. pipeline; farm; divide & conquer
- skeleton libraries for C/Java



divide & conquer

Algorithmic skeletons

- industrial frameworks
- e.g. Google Map-Reduce
- Apache Hadoop



Google Map-Reduce

http://labs.google.com/papers/mapreduce-osdi04-slides/index-auto-0008.html

Algorithmic skeletons

- industrial frameworks
- e.g. Microsoft Dryad

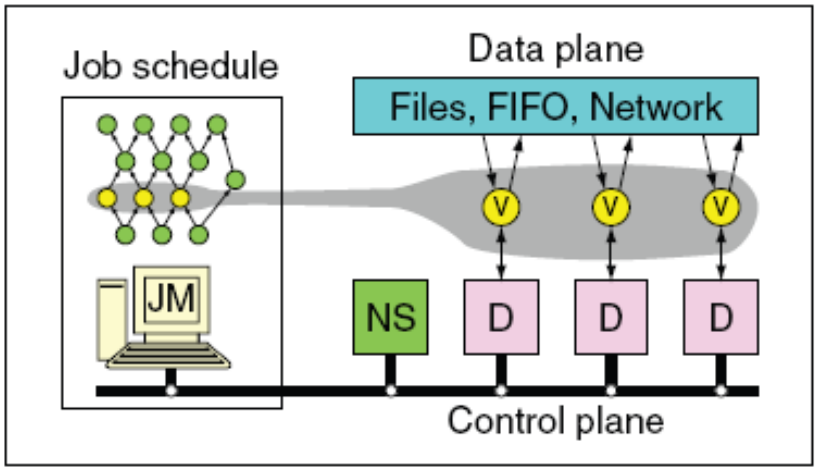


Figure 1: The Dryad system organization. The job manager (JM) consults the name server (NS) to discover the list of available computers. It maintains the job graph and schedules running vertices (V) as computers become available using the daemon (D) as a proxy. Vertices exchange data through files, TCP pipes, or shared-memory channels. The shaded bar indicates the vertices in the job that are currently running.

Microsoft Dryad

www.wikibench.eu/CloudCP2011/wp-content/.../Isaacs-keynote.ppsx

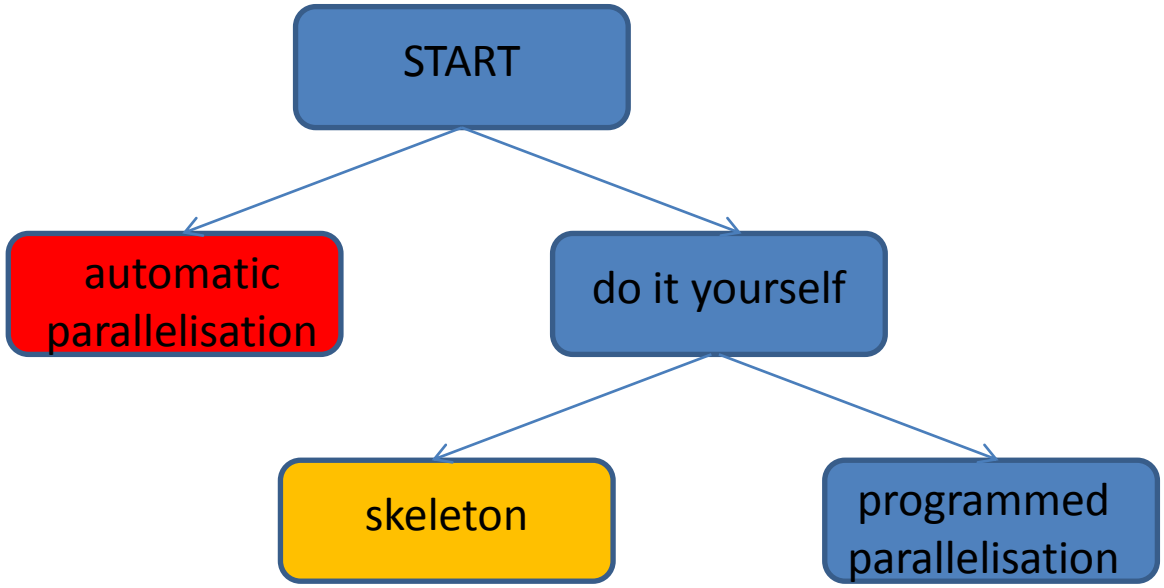
Algorithmic skeletons

- can choose appropriate skeleton for problem class
- medium effort to use skeleton library/industrial framework
  - must fit problem to skeleton
- high effort to develop own skeletons
  - must make communication & task division explicit

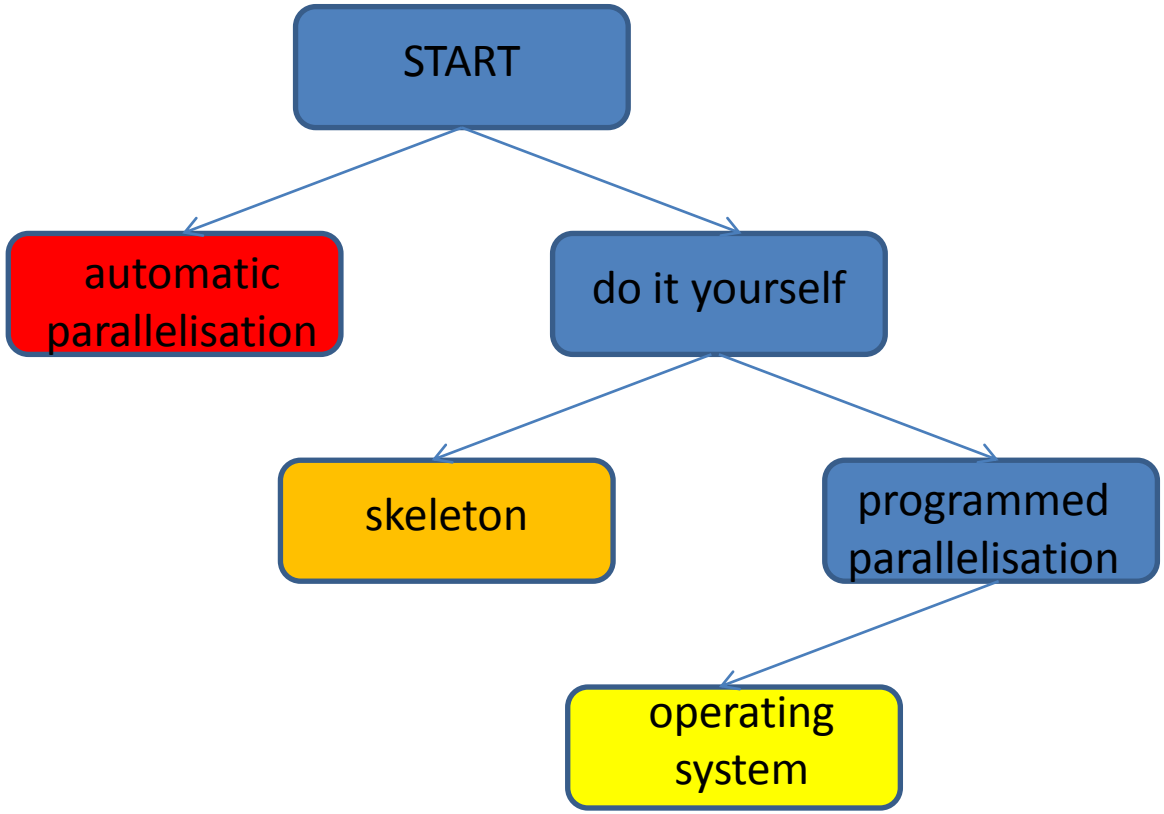
Algorithmic skeletons

- can hand tune for:
  - problem
  - irregularity
  - scalability
  - process placement
- strong potential re-use of components

Methodological choices



Methodological choices



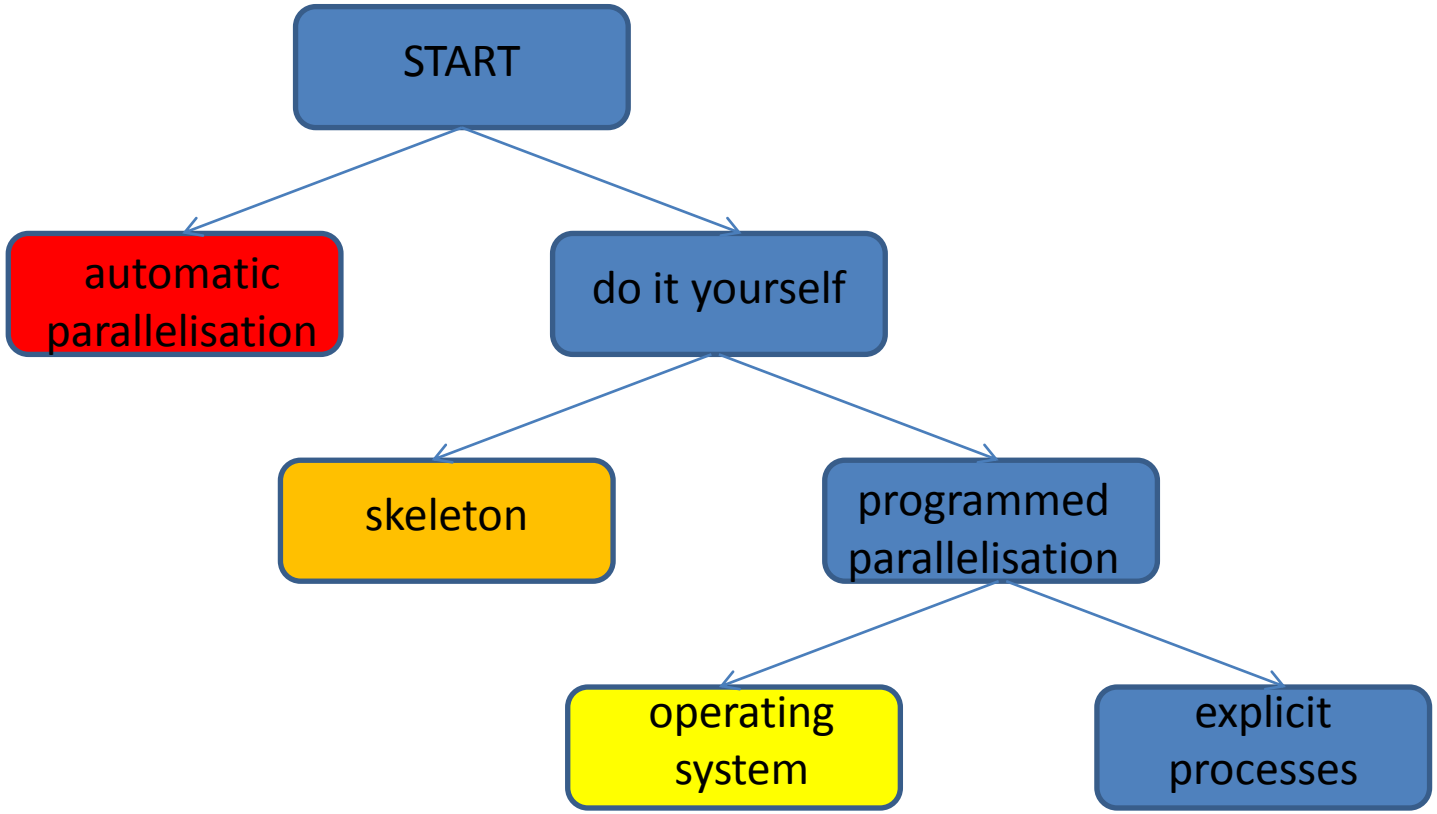
Operating system

- independent programs
  - realised as threads
- communication via pipes/sockets
- bolted together with shell scripts

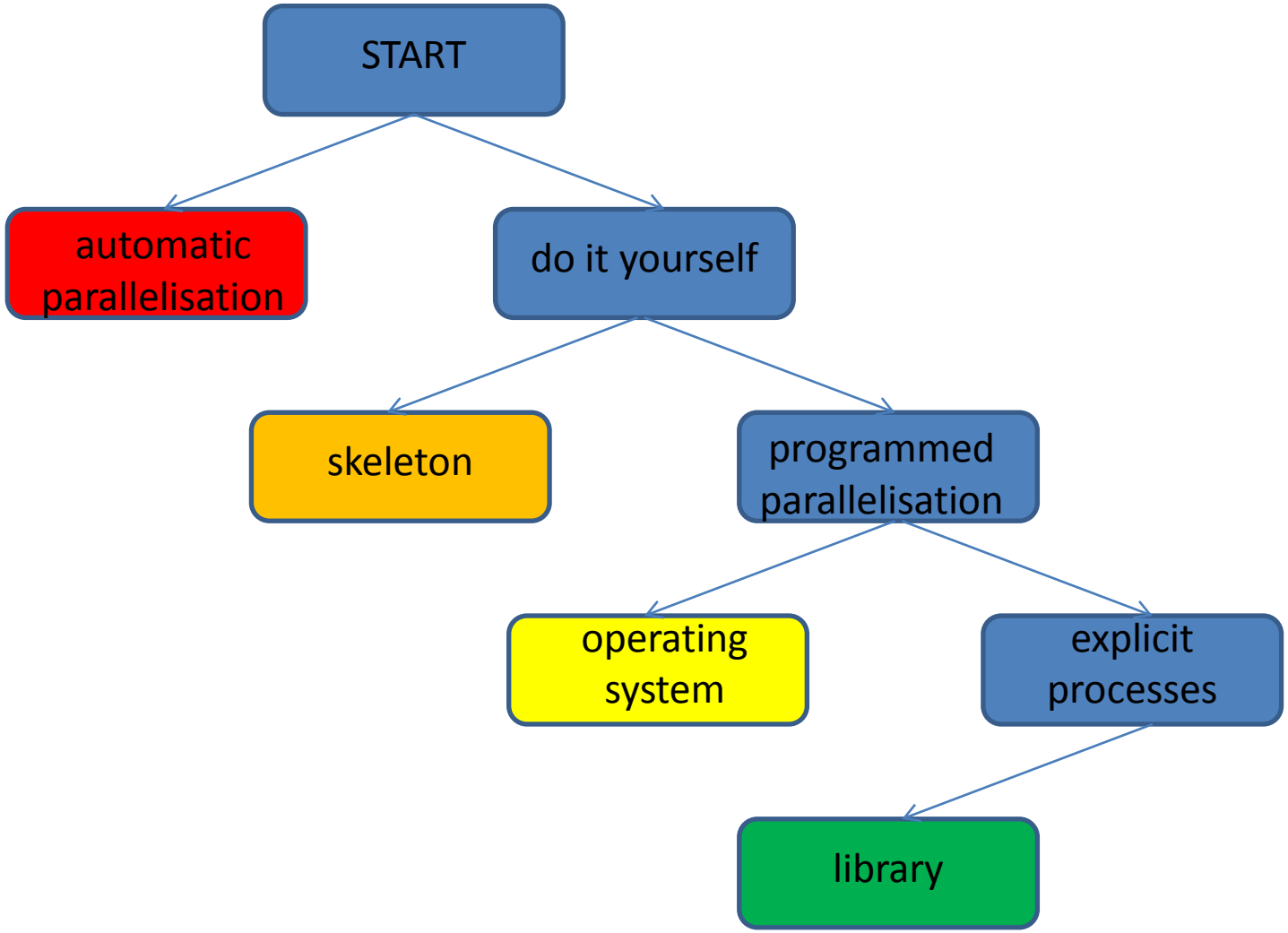
Operating system

- low effort
- highly dependent on underlying operating system for:
  - communication
  - scheduling
  - process placement
- unpredictable performance

Methodological choices



Methodological choices



Library

- shared memory
  - OpenMP
    - platform & architecture independent
  - Posix Threads
    - Unix/Linux specific/architecture independent
  - Intel Threading Building Blocks
    - platform/architecture independent

Library

- distributed memory
  - MPI & PVM
- specialised hardware
  - SIMD on MMX/SSE
  - CUDA & OpenCL for GPU arrays

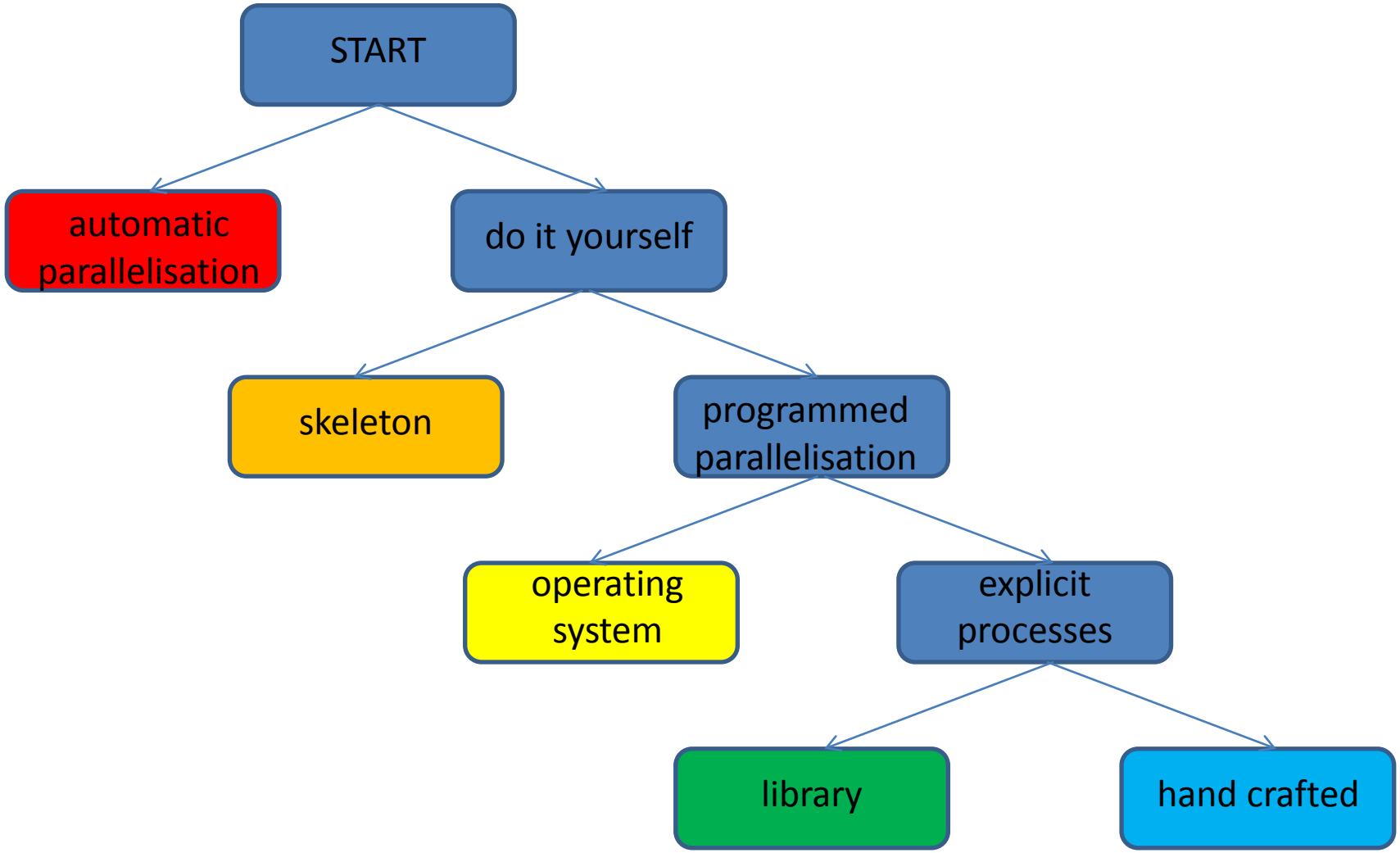
Library

- now common to use:
  - MPI for inter-cluster
  - OpenMP for intra-cluster
- medium to high effort
  - explicit communication & task division
- can shape algorithm to architecture
- best for irregular problem/architecture

Library

- often end up re-inventing some standard algorithmic skeleton
- good potential for reuse of:
  - structure
  - components

Methodological choices



Hand crafted

- very low level
- shared memory
  - critical regions via semaphores
- distributed memory
  - communication over RS232; USB

Hand crafted

- very high effort
- highly problem/architecture specific
- best for embedded systems

Questions...

- is my problem suitable for parallelisation?
- how do I know how my problem scales?
- if I parallelise my problem, how do I tell how much communication overhead will be incurred?
- how do I assess the benefits of shared versus distributed memory?

Questions...

- can I do better with smarter solutions on my existing technology?
- where can I get help with deciding how to proceed?
- have other people already come up with solutions that might work for me?

Future

- UK has major research strengths in multi-processor architectures, parallel languages/compilers, skeletons etc
- groups don't talk much to each other or to practitioners e.g. in eScience
- need to build inclusive UK community
- opportunities through
  - EPSRC multi-core priority for ITC
  - TSB ICT KTN for multi-core