Designing and Building Parallel Programs

A Tutorial Presented at Supercomputing '95
San Diego
December 2nd, 1995

(c) 1995
Ian Foster **Bill Gropp**
Carl Kesselman **Charles Koelbel**

Outline

- Parallel Computers & Computing
- Designing Parallel Algorithms
- Understanding Performance
- Parallel Programming Tools
- Application Examples
- Summary and Conclusions

Outline

- **Parallel Computers & Computing**
- Designing Parallel Algorithms
- Understanding Performance
- Parallel Programming Tools
- Application Examples
- Summary and Conclusions

The Need for Higher Performance

- Traditional compute-intensive applications
  - Weather, CFD, chemistry, etc.
- New data-intensive applications
  - Video servers, data mining
- Future high performance apps will combine elements of both
  - VR, collaborative environments, CAD

DBPP4

Performance Requirements

- Climate model
  - 10^{16} ops for 10 yr simulation (10 days @ 10GF)
  - Easily generate 100 GB data
  - Want x 10^4: more resolution, duration, detail
- 3-D real-time video
  - Data sets approach 1024^3 volume elements
  - 1024^3 x 30 fps x 300 ops/el = 10^{13} ops/sec
  - I/O requirements are correspondingly high

Designing and Building Parallel Programs 5

Why Parallel Computing?

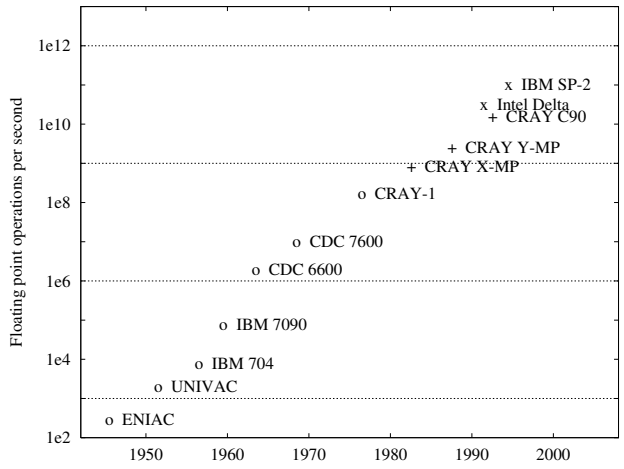
- Continuing demands for higher performance
- Physical limits on single processor performance
- High costs of internal concurrency
- Result is rise of multiprocessor architectures
  - And number of processors will continue to increase
- Networking is another contributing factor

Future software must be concurrent & scalable

Designing and Building Parallel Programs 6

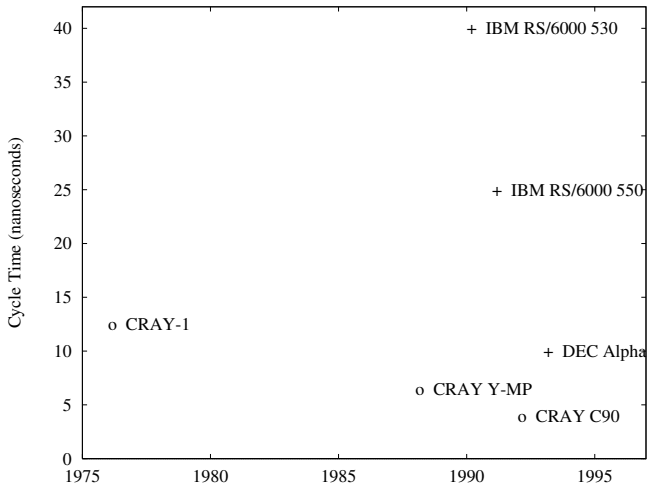
DBPP6

Supercomputer Peak Performance



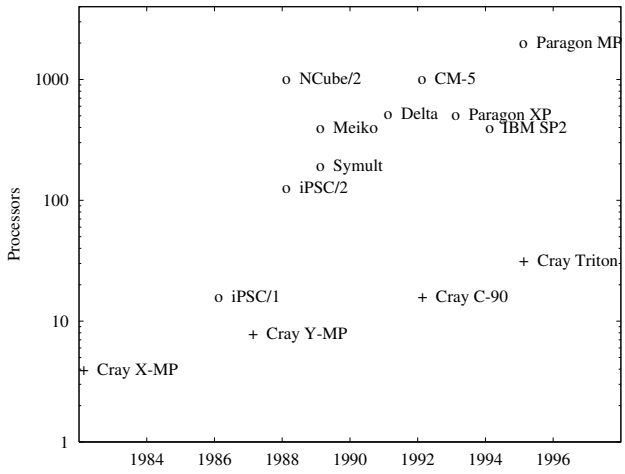
DBPP7

Cycle Time Trends



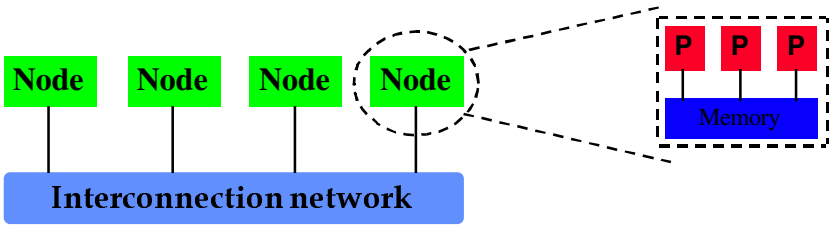
DBPP8

Processor Count Trends



DBPP10

The Multicomputer: an Idealized Parallel Computer



Multicomputer Architecture

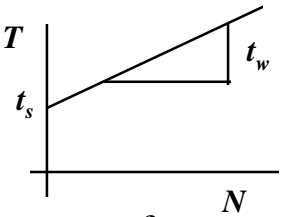
- Multicomputer = nodes + network
- Node = processor(s) + local memory
- Access to **local** memory is **cheap**
  - 10s of cycles; does not involve network
  - Conventional memory reference
- Access to **remote** memory is **expensive**
  - 100s or 1000s of cycles; uses network
  - Use I/O-like mechanisms (e.g., send/receive)

Multicomputer Cost Model

- Cost of remote memory access/communication

$$T = t_s + N\,t_w$$

  - t_s = per-message cost ("latency")
  - t_w = per-word cost
  - N = message size in words

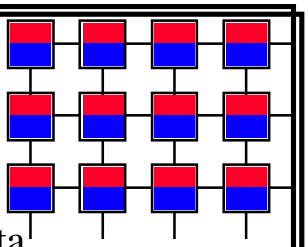


- Hence **locality** is an important property of good parallel algorithms

How do Real Parallel Computers Fit the Model?

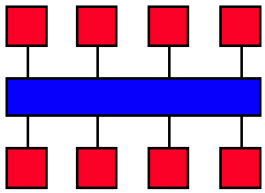
- Major architecture types
  - Distributed memory MIMD
  - Shared memory MIMD
  - Workstation clusters and "metacomputers"
  - SIMD
- First three converging; fourth disappearing
- Model fits first three reasonably well

Distributed Memory MIMD Multiprocessor



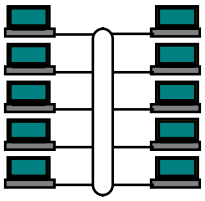
- Multiple Instruction/Multiple Data
- Processors with local memory connected by high-speed interconnection network
- Typically high bandwidth, medium latency
- Hardware support for remote memoryaccess
- Model breaks down when topology matters
- Examples: Intel Paragon, Cray T3D, IBM SP

Shared Memory MIMD Multiprocessor



- Processors access shared memory via bus
- Low latency, high bandwidth
- Bus contention limits scalability
- Search for scalability introduces locality
  - Cache (a form of local memory)
  - Multistage architectures (some memory closer)
- Examples: Cray C90, SGI, Sun

Workstation Clusters



- Workstations connected by network
- Cost effective
- High latency, low to moderate bandwidth
- Often lack integrated software environment
- Model breaks down if connectivity limited
- Examples: Ethernet, ATM crossbar, Myrinet

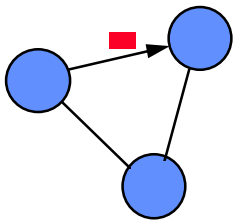
SIMD

- Single Instruction, Multiple Data
- Each processor executes same program, operates on different data
- Excellent for some problems, but limited applicability
- Multicomputer model not appropriate
- Example: Maspar MP-2

DBPP12

A Simple Parallel Programming Model

- A parallel computation is a set of **tasks**
- Each task has local data, can be connected to other tasks by **channels**
- A task can
  - Compute using local data
  - Send to/receive from other tasks
  - Create new tasks, or terminate itself
- A receiving task blocks until data available



Properties

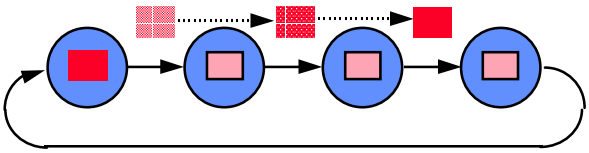
- Concurrency is enhanced by creating multiple tasks
- Scalability: More tasks than nodes
- Locality: Access local data when possible
- A task (with local data and subtasks) is a unit for modular design
- Mapping to nodes effects performance only

Parallel Algorithm Example

- Compute interactions between N objects

$$F_i = \sum_{j=0}^{N-1} I_{ij}$$

- *N* tasks, one per object, connected in a ring
- Objects circulate, tasks compute interactions



Summary: Parallel Computers and Computation

- Parallelism is motivated by future performance needs, and will become ubiqitious
- An idealized machine model: **multicomputer**
- A simple parallel programming model
  - **Tasks** and **channels**
- Four important attributes of parallel software
  - **Concurrency**, **scalability**, **locality**, **modularity**

Outline

- Parallel Computers & Computing
- **Designing Parallel Algorithms**
- Understanding Performance
- Parallel Programming Tools
- Application Examples
- Summary and Conclusions

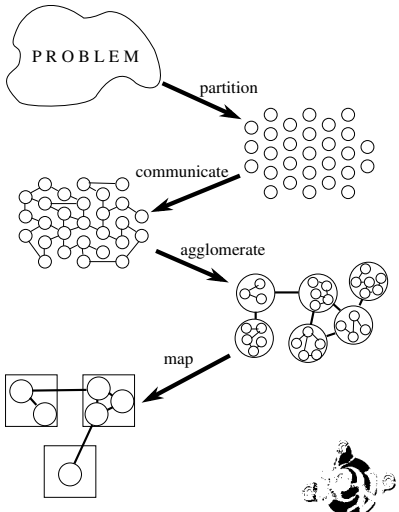
Parallel Algorithm Design

- Goal: "Develop an efficient (parallel) solution to a programming problem"
  - Identify sensible parallel algorithms
  - Evaluate performance and complexity
- We present
  - A systematic design methodology
  - Some basic design techniques
  - Illustrative examples

DBPP29

A Design Methodology

- Partition
  - Define tasks
- Communication
  - Identify requirements
- Agglomeration
  - Enhance locality
- Mapping
  - Place tasks

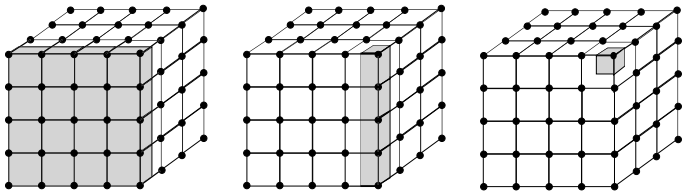


Partitioning

- Goal: identify opportunities for concurrent execution (define tasks: computation+data)
- Focus on *data operated on* by algorithm ...
  - Then distribute computation appropriately
  - Domain decomposition
- ... or on the *operations performed*
  - Then distribute data appropriately
  - Functional decomposition

Domain Decomposition

- Example: Finite difference atmosphere model (data structure = 3-D grid)



- 3-D decomp provides greatest flexibility

$$Xi = (X_{i-1} + 2*X_i + X_{i+1})/4$$

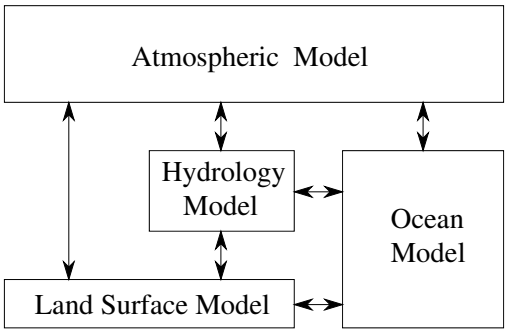
Functional Decomposition

- Example: naive search
- No obvious data structures to decompose
- Create new task for each search call

```
procedure search(A)
begin
  if (solution(A)) then
    report solution and score
  else
    foreach child A(i) of A
      search(A(i))
    endfor
  end
end
```

Functional Decomposition (2)

- Example: Earth system model. Functional decomposition enhances modularity



Communication

- Identify communication requirements
  - If computation in one task requires data located in another, communication is needed
- Example: finite difference computation

$$Xi = (X_{i-1} + 2*X_i + X_{i+1})/4$$

Partition creates one task per point

- Must communicate with each neighbor

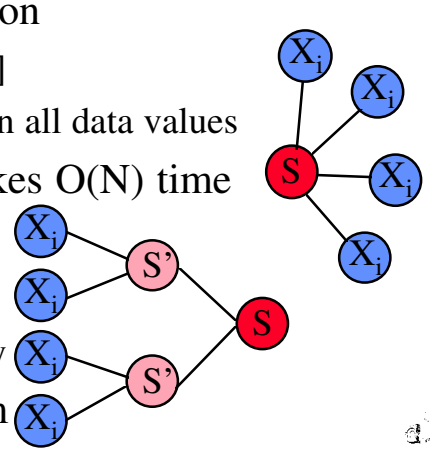
Communication Structures

- **Local** (vs. global)
  - If number of "neighbors" small
- **Structured** (vs. unstructured)
  - If tasks form a regular structure (tree, mesh)
- **Static** (vs. dynamic)
  - If structure unchanging over time
- **Synchronous** (vs. asynchronous)
  - If both sender and receiver participate

DBPP36

Global Communication

- Example: summation
  - S = sum $_{i=0:N-1}$ [X_i]
  - Value S depends on all data values
- Simple solution takes O(N) time
- Design principles
  - Distribute work
  - Allow concurrency
- O(log N) algorithm

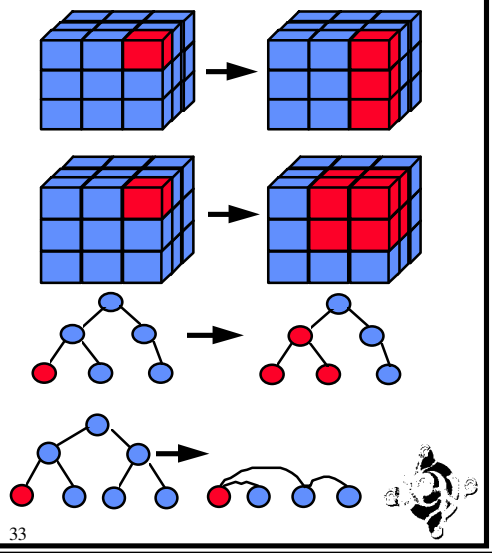


Agglomeration

- Once tasks + communication determined, "agglomerate" small tasks into larger tasks
- Motivations
  - To reduce communication costs
  - If tasks cannot execute concurrently
  - To reduce software engineering costs
- Caveats
  - May involve replicating computation or data

Agglomeration Strategies

- Reduce dimension
- Increase grain
- Coalesce subtrees
- Combine nodes



Mapping

- Place tasks on processors, to
  - Maximize concurrency } *note potential*
  - Minimize communication } *conflict*
- Techniques:
  - Regular problems: agglomerate to *P* tasks
  - Irregular problems: use **static load balancing**
  - If irregular in time: **dynamic load balancing**

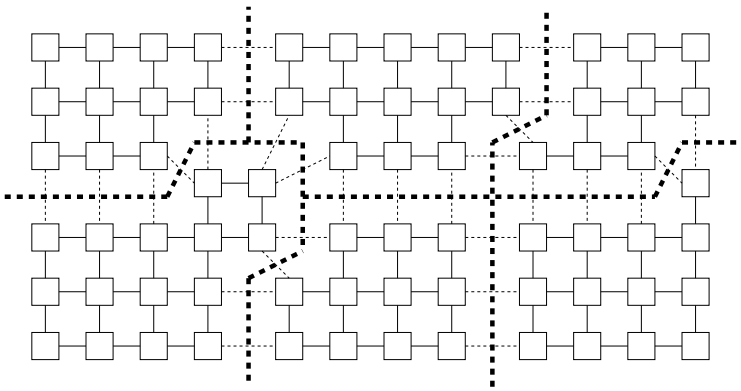
DBPP52

Load Balancing Techniques

- Static load balancing
  - Recursive bisection
  - Local algorithms
  - Probabilistic algorithms
  - Cyclic mappings
- Dynamic load balancing [task scheduling]
  - Manager/worker: centralized & hierarchical
  - Decentralized schemes

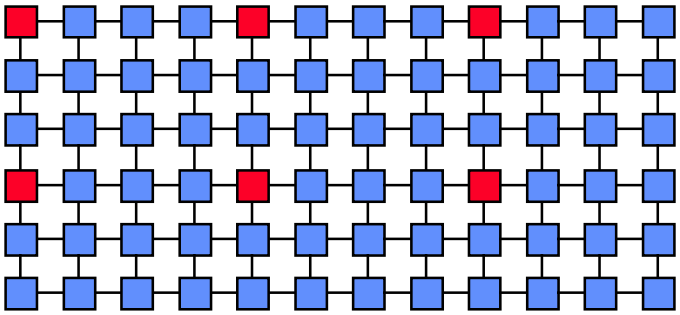
DBPP53

Example: Grid Problem



DBPP54

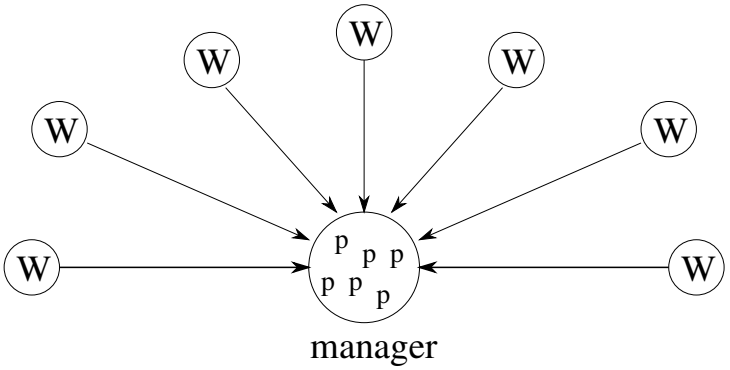
Cyclic Mapping



Can address static and dynamic load imbalances;
a form of probabilistic mapping

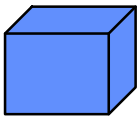
DBPP55

Manager/Worker Structure



Workers request tasks from a central manager

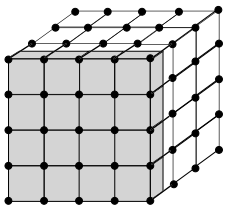
Example: Atmosphere Model



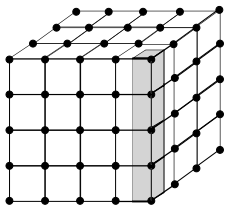
- Used to simulate atmospheric processes
- Represent atmosphere state by 3-D grid
  - Periodic in two horizontal dimensions
  - $N_x \cdot N_y \cdot N_z$: e.g., N_y=50-500, N_x=2N_y, N_z=15-30
- Computation includes
  - Atmospheric dynamics: finite difference
  - "Physics" (radiation etc.) in vertical only

Atmosphere Model: Partition

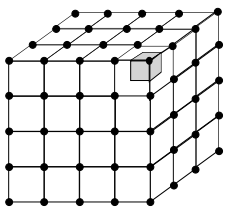
- Use domain decomposition
  - Because model operates on large, regular grid
  - Can decompose in 1, 2, or 3 dimensions
  - 3-D decomposition offers greatest flexibility



1-D



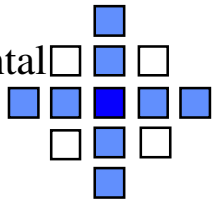
2-D



3-D

Atmosphere Model: Communication

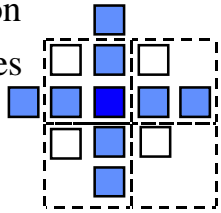
- Finite difference stencil in horizontal
  - Local, regular, structured



- Radiation calculations in vertical
  - Global, regular, structured $TCS_k = \prod_{i=1}^{k}(1-\mathrm{dd}_i)TCS_1$
- Diagnostic sums $Mass = \sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\sum_{k=0}^{Z-1} X_{ijk}$
  - Global, regular, structured

Atmosphere Model: Agglomeration

- In vertical
  - Performance: avoid communication
  - Modularity: Reuse physics modules
- In horizontal
  - Clump so that 4 points per task
  - Efficiency: communicate with 4 neighbors only



- Resulting algorithm "reasonably scalable"
  - $(N_x \cdot N_y)/4$: at least 1250 tasks

Atmosphere Model: Mapping

- Technique depends on load distribution

1) Agglomerate to one task per processor
   - Appropriate if little load imbalance
2) Extend (1) to incorporate cyclic mapping
   - Works well for diurnal cycle imbalance [DBPP63]
3) Use dynamic, local load balancing
   - Works well for unpredictable, local imbalances

Summary

- Goal: explore design space, identify optima
- Four step process
  - Partition a problem into many small tasks
  - Determine communication requirements
  - Use agglomeration to improve performance
  - Map tasks to processors
- Small set of standard structures & techniques