Parallelisation for Beginners

Ruud van der Pas

SiliconGraphics

European High Performance Computing Team

Topics

- Concepts in parallelisation
- Parallel Architectures
- Programming Models

Fundamentals of Parallelisation

Fundamentals of Parallelisation

What is parallelisation ?

Usage of multiple processors to solve one specific task

Benefit of parallelisation ?

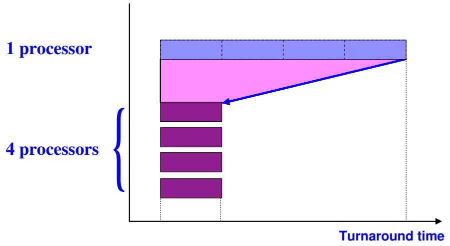
Faster turnaround time of the program

How is this achieved ?

Stay tuned .....

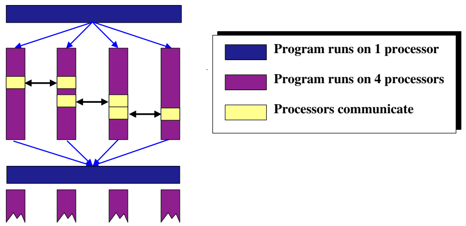
Fundamentals of Parallelisation

The objective of parallelisation:

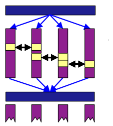


Fundamentals of Parallelisation

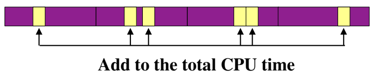
- Decrease in turnaround time –– The goal
- (Minor) Increase in total CPU time –– Side effect

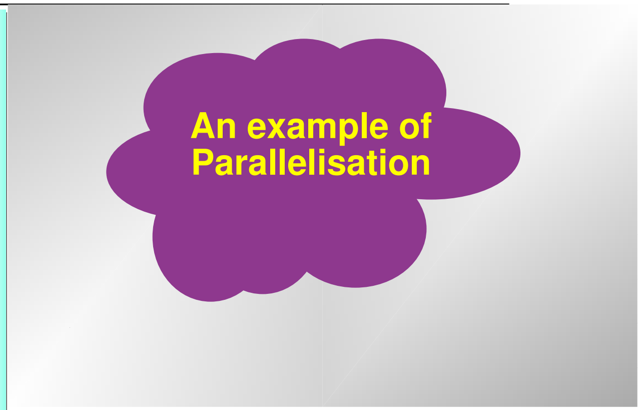


Fundamentals of Parallelisation



Efficient parallelisation minimizes communication





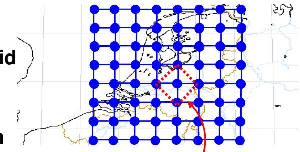
Example of Parallelisation

Weather prediction

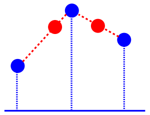
- The complex equations can not be solved with a formula
- Numerical techniques need to be applied to obtain an approximation of the solution
- This is achieved by placing a computational grid over the area of interest
- We will then get the solution at the grid points
- This calculation is repeated for different time steps and the solution at T=1 is input for the solution at T=2, and so on .....

Example of Parallelisation

Weather prediction



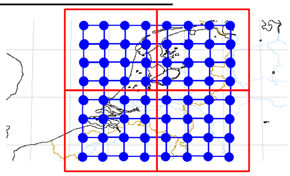
- Let's assume we use a 8x8 grid
- We then get 64 values
- Interpolation is used to obtain the solution in non-grid points:



Value depends on the neighbouring values

Example of Parallelisation

Weather prediction



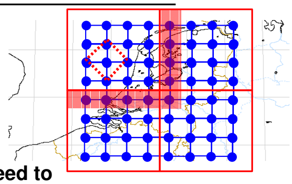
- How to get Parallelism:

  Divide the grid into pieces

- Then each piece can be solved independently i.e.
  we can compute in parallel !
- Every processor will perform the computation for it's piece

Example of Parallelisation

Weather prediction



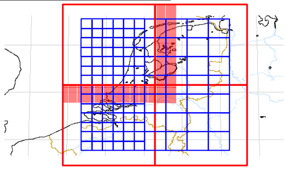
- Pieces need to know about the solution at their *boundaries*, so this data needs to be transferred
- This means that processors need to *COMMUNICATE* boundary values with each other
- After all processors are done, we need to check the solution:

  Every processor computes a physical quantity and a decision is made about the quality of the solution

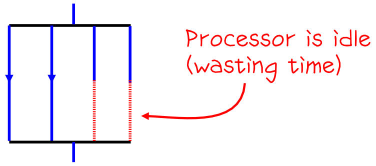
  This means they need to transmit/send a value and receive a decision on the quality

Example of Parallelisation

Weather prediction

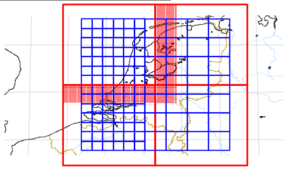


- Not all areas are of the same interest
- This means we need more grid points in certain parts
- It implies that the processors no longer do the same amount of work: this is called a load–balancing problem



Example of Parallelisation

Weather prediction



About the numbers:

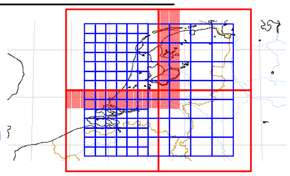
- Processors communicate on the boundaries i.e. with *at least* 4+3=7 points per piece
- So, 4*7=28 out of the 64 values need to be communicated!

  In general for an N*N piece, about 4*N*P values are transferred on a P processor system

  This is related to the BANDWIDTH ("burst transfer") of the parallel system

Example of Parallelisation

Weather prediction



More about the numbers:

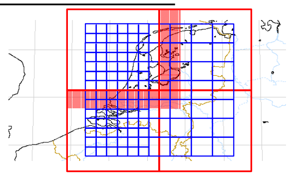
- Processors send their quality number and receive a decision back
- This is *independent of the size* of their piece

  In general 2*P single values are transferred on a P processor system

  This is related to the LATENCY ("short messages") of the parallel system

Example of Parallelisation

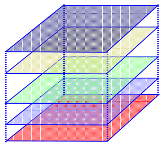
Weather prediction



About growth

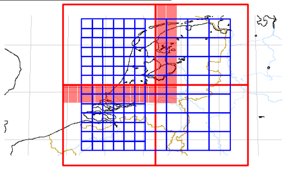
- It is not very interesting to know the weather at one specific height
- Instead, we want to know it at several heights and so want to solve a:

  3–dimensional problem instead of a 2–dimensional one:



Example of Parallelisation

Weather prediction



More about growth

- Also we want to know more than one physical quantity per point (temperature, humidity, pressure, ...)
- This easily leads to huge memory requirements and lots of data transfers (small and large) between the processors
- Example:

  500x500 points, using 100 layers, and storing 10 relevant physical values per point requires 1.86 GByte of memory

Concepts in Parallelisation

Concepts in Parallelization

Concepts in Parallel Computing:

- Speed–up and Efficiency
- Scalability and Amdahl's Law
- Latency and Bandwidth
- Interconnects
- Message Passing
- Shared Memory

Concepts in Parallelization

Speed up and Efficiency

- Both are a measure of how well a program has been parallelized
- Let T(p) be the turnaround time on p CPUs
- The Speed up S(p) and Efficiency E(p) are then defined as:

  $S(p) = T(1)/T(p) \quad (p=1,2,3,....)$

  $E(p) = S(p)/p$

Concepts in Parallelisation

Speed up and Efficiency

- For a perfect parallel speed-up we get:

  $T(p) = T(1)/p$

  $S(p) = T(1)/T(p) = T(1)/(T(1)/p) = p$

  $E(p) = S(p)/p = p/p = 100\%$

- The deviation from these optimal values tells us something about how good (or bad) the application has been parallelized

Concepts in Parallelisation

Speed up and Efficiency

Example:

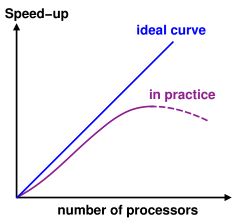
Time on 1 CPU	100	seconds (= T(1))
Time on 15 CPUs	12	seconds (= T(15))

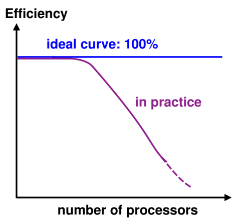
Speed up = 100/12 = 8.3

Efficiency = 8.3/15 = 56%

Concepts in Parallelisation

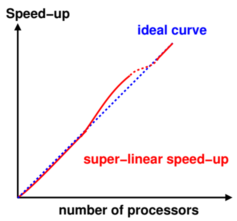
Speed up and Efficiency





Concepts in Parallelisation

Superlinear Speed up



Superlinear speed–up:

More than P times faster on P processors!

Why?

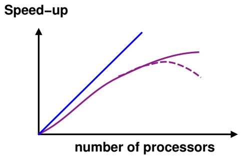
Total cache size increased with a factor of P

Don't count on this to happen ...

Concepts in Parallelisation

Scalability

- Scalability is a vague thing to start with
- Often it means that the efficiency E(p) remains constant over a large range



Concepts in Parallelisation

Scalability

Possible reasons for poor scalability:

1. **The parallel system used**
   (can't be an SGI system, buy one from us ....)
2. **The parallelized program**
   (revist your work ....)
3. **A combination of 1 and 2**
   (you're hopeless, I give up ....)

Parallel Portion

Sequential Portion

Concepts in Parallelisation

Scalability

- In general, a parallel system using slow processors will scale better
- However, in the end only the total turnaround time matters
- An example now follows

Concepts in Parallelisation

Scalability

An example. Assume:

- A program is 80% parallel: T(1) = 80 + 20
- The processors of System B are 2x faster than the processors of System A
- The additional parallel overhead on, say, 4 CPUs is 5 seconds on *both* systems

The last assumption of equal overhead is a simplification, but an example is putting faster processors on the same interconnect

Concepts in Parallelisation

Scalability

On 4 processors we obtain the following:

System A

T(1)	= 100	= 80+20	
T(4)	= 80/4+20+5	= 20+20+5 =	45
Speed-up	= 100/45	= 2.22	

Concepts in Parallelisation

Scalability

On 4 processors we obtain the following:

System A

T(1)	= 100	= 80+20	
T(4)	= 80/4+20+5	= 20+20+5 =	45
Speed-up	= 100/45	= 2.22	

!

System B

T(1)	= 50	= 40+10	
T(4)	= 40/4+10+5	= 10+10+5 =	25
Speed-up	= 50/25	= 2.00	

Concepts in Parallelisation

Scalability

- An important issue has been raised implicitly
- It even has an (incorrect) name:

Amdahl's Law

Concepts in Parallelisation

Amdahl's Law

- Although not a law, it is a very useful rule
- Basically it states the following:

The non-parallelizable fraction of a program limits performance and poses an upper limit on the scalability

Concepts in Parallelisation

Amdahl's Law

- Take our example of 80% parallel code:

  T(1) = T(parallel) + T(sequential) = 80 + 20

- On P processors we then get: T(P) = 80/P + 20
- No matter how large P is: T(P) > 20
- S(P) = T(1)/T(P) = 100/(80/P+20) < 5

  E(P) = S(P)/P < 5/P $\xrightarrow[\text{(if P increases)}]{}$ 0 !

Concepts in Parallelisation

Amdahl's Law

- In general a program that has been parallelized for "F%" will not run faster than "100/(100–F)":

  T(1) = T(parallel) + T(sequential) = F + (100–F)

- On P processors: T(P) = F/P + (100–F)
- No matter how large P is: T(P) > 100–F
- S(P) = T(1)/T(P) = 100/(F/P+100–F) < 100/(100–F)

Concepts in Parallelisation

Amdahl's Law

- We have now answered the question:

  "What is the maximum parallel speed–up I can get for a program that is parallelized for F%?"

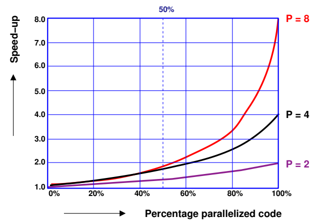
- Answer:

  S(P) = 100/(F/P+100–F) (F=0%,....,100%)

  Amdahl's Law

Concepts in Parallelisation

Amdahl's Law



Concepts in Parallelisation

Amdahl's Law

It is now quite easy to *estimate the fraction F* of parallelizable code

We use:

$S(P) := T(1)/T(P) = 100/(F/P+100-F)$

S(P) can be measured. P is known and so we can solve for F:

Concepts in Parallelisation

Amdahl's Law

$$F = \frac{P}{P-1} * \frac{(S(P)-1)}{S(P)} * 100\ \%$$

An example:

T(1) = 100 *measured* 1 processor elapsed time
T(2) = 60 *measured* 2 processor elapsed time

Parallel fraction:

$F = (2/1)*((1.67-1)/1.67)*100 = 80\%$

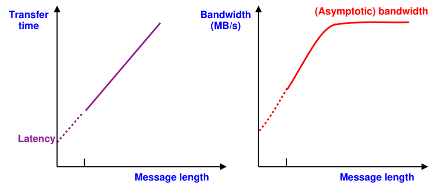
Concepts in Parallelisation

Amdahl's Law

- In practice, things are not as bad as they may now seem, but ....
- It is clear that a *significant portion* needs to be parallelized, but ....
- A significant portion of *what* ?
- ***The runtime*** (not necessarily much code!)
- We will return to this issue later on

Concepts in Parallelisation

Latency and Bandwidth



Latency Characteristic for short messages
Bandwidth Characteristic for long messages

Concepts in Parallelisation

Interconnects

- With parallel processing, processors often need to communicate with each other
- Therefore some sort of connection has to exist between them: "the interconnect"
- This can be almost anything, ranging from a ordinary network (like Ethernet) to a dedicated interprocessor communication device

Concepts in Parallelisation

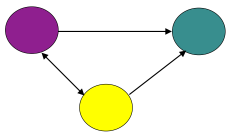
Interconnects

- The interconnect can have a big impact on:
  - The parallel performance
  - Implementation of the parallelization
  - Ease of use
- Of course, the application itself will influence the relative importance of these factors

Concepts in Parallelisation

Message Passing

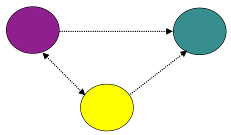
- Communication is performed by explicitly sending and receiving messages
- Also synchronization is controlled explicitly



Concepts in Parallelisation

Shared Memory

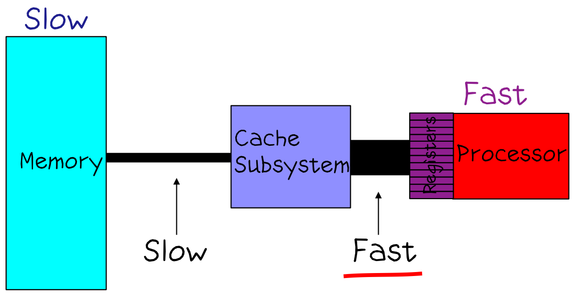
- Communication is performed transparently
- Also synchronization is done transparently



Parallel Architectures

Caches

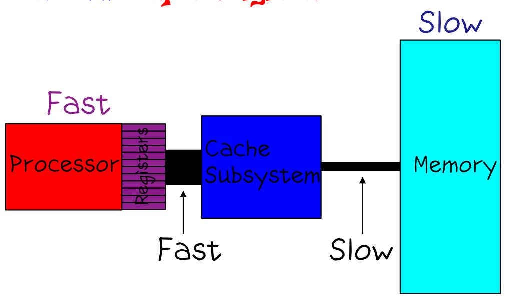
Caches play an important role in performance



Note:
The R10000 processor has non-blocking caches to overlap computation and memory access

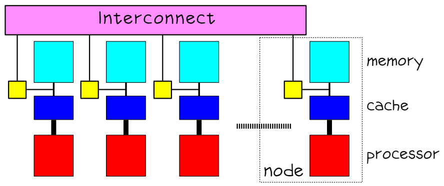
Parallel Architectures

A modern computer system



Parallel Architectures

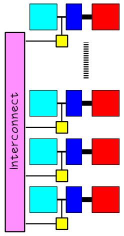
A distributed memory parallel computer



Parallel Architectures

A distributed memory parallel computer

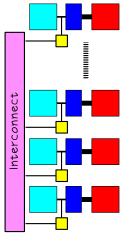
- The interconnect can be:
  - A standard network (Ethernet, FDDI, ..)
  - A dedicated data–transfer system
- A node can be:
  - A complete workstation
  - A processor+memory+cache only



Parallel Architectures

A distributed memory parallel computer

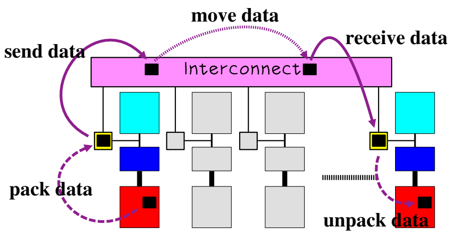
- The memory of each node is:
  - Independent
  - Unaware of other node's memories
- Data transfer:
  - Under full programmer control
  - Synchronous or a–synchronous



Parallel Architectures

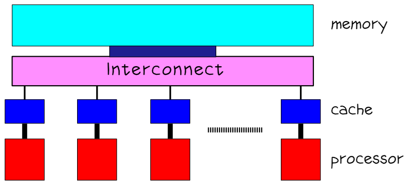
A distributed memory parallel computer

The system is programmed using message passing



Parallel Architectures

A shared memory parallel computer



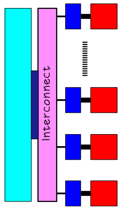
Parallel Architectures

A shared memory parallel computer

- The interconnect can be:

  A bus or a crossbar

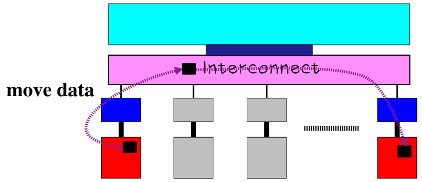
- There is no notion of a "node"
- The memory of the system is shared
- Data transfer:
  - Transparent to the programmer
  - Synchronous or a-synchronous



Parallel Architectures

A shared memory parallel computer

Data movement is transparent to the programmer



Parallel Architectures

Symmetric Multi Processor

- This is also abbreviated as "SMP"
- With SMP it is meant that all processors are equal i.e. there is no master-slave situation
- It is often used as a synonym for a shared memory multi-procesor system

Parallel Architectures

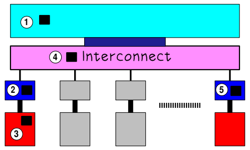
Cache Coherency

- Globally speaking, "coherency" is the (hidden) mechanism that ensures that data has a consistent value throughout the entire system
- With "cache coherency" this is meant to refer to data where cache memory is part of the memory hierarchy

Parallel Architectures

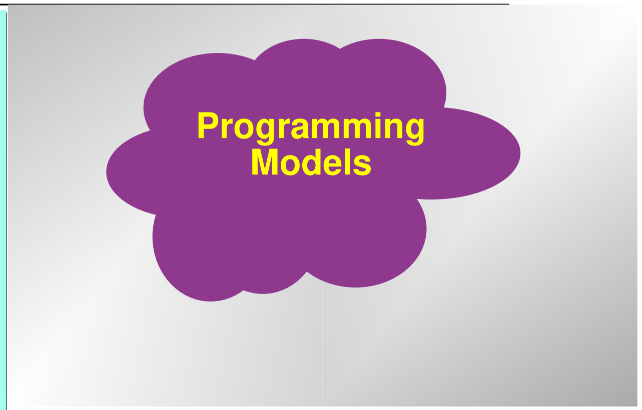
Cache Coherency

The same data can be:

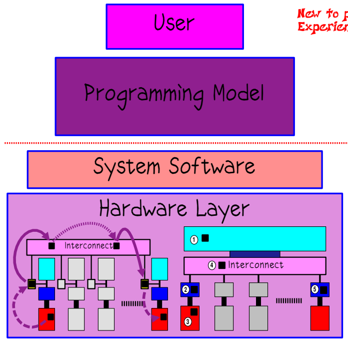


1. In main memory and
2. In the cache of CPU 1 and
3. In a register of CPU 1 and
4. On it's way to CPU 2 and
5. In the cache of CPU 2

With cache coherency it is ensured that a read request always returns the most recent value (wherever it may come from)



Programming Models



New to parallel processing
Experienced in parallel processing

Transparent
Shared Memory
Message Passing
Hybrid

Operating System
Compilers

Distributed Memory
Shared Memory

Programming Models

Message Passing

- The programmer is fully responsible to implement the parallelization
- This means taking care of:
  - Process management
  - Sending and receiving messages
  - Synchronization

Programming Models

Message Passing

Code fragment example

Put in by the programmer

```
cc = pvm_spawn ("hello_other", (char**)0, 0, "", 1, &tid);

if (cc == 1) {
   cc = pvm_recv (-1, -1);
   pvm_bufinfo (cc, (int*)0, (int*)0, &tid);
   pvm_upkstr (buf);
   printf("from t%x: %s\n", tid, buf);

} else
   printf("can't start hello_other\n");

pvm_exit();
```


Programming Models

Message Passing

– Nowadays, portable packages exist to implement the message passing:

- Message Passing Interface (MPI)

  Designed to be a de–facto standard

- Parallel Virtual Machine (PVM)

  Used to be the most widely used package

- PARMACS

  Loosing ground

Programming Models

Message Passing

– MPI and PVM come for free

– Both are available through the public domain

– In addition to this, vendors may provide tuned versions

– Such versions can be expected to exploit the features of a specific architecture

– Often also official vendor support is given

Programming Models

Message Passing

– Effort has been put into automating the process of writing message passing codes

– Two of such projects are:

1. Vienna Fortran – Research Project
2. High Performance Fortran (HPF) – Standard?

– Both extend the Fortran language and rely on the same principles

Programming Models

Message Passing

- The user specifies:
  - the data–layout
  - processor topology
- The compiler then generates the message passing code
- Although this is a nice initiative, it still forces the programmer to consider many details

Programming Models

Shared Memory

- This is really a bad synonym for "automatic parallelization by a compiler"
- It is always implicitly assumed to be used on shared memory systems, hence the name
- The goal is to use a compiler to parallelize a program without user–intervention
- Current state of compiler technology is advanced, but the goal hasn't been reached

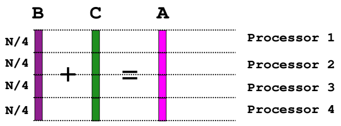
Programming Models

Shared Memory

A trivial example:

```
DO I = 1, N
   A(I) = B(I) + C(I)
END DO
```

The compiler will do the following (on 4 CPUs):



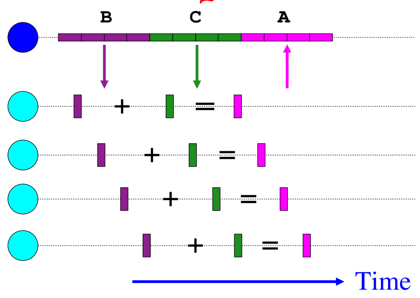
Programming Models

Distributed Memory

- As can be seen in this example, a shared memory architecture is a great help
- We will now symbolically outline a distributed memory implementation
- For sake of simplicity, we will assume that:
  - There is one (logical) master processor that sends and collects the data
  - There are 4 slave processors that do the work

Programming Models

Distributed Memory



Programming Models

Distributed Memory

```
       MASTER                          SLAVE

spawn slaves

(split B and C)

foreach slave
   send (B,C)      ------------>    receive (B,C)
end for
                                    A = B + C
foreach slave
   receive (A)     <------------    send (A)
end for

(merge A)
```


Programming Models

Shared Memory

- This is not the perfect world either .....
- The status of compiler technology is such that in certain cases it is needed to guide the parallelization
- This can often be accomplished by using "compiler directives" or pragma's
- These take the form of language comment and so preserve portability

Programming Models

Shared Memory

- A compiler directive/pragma could look like this:

```
c$ parallelize_this_loop
      do i = 1, N
         call suba(i)
      end do

#pragma parallelize_this_loop
      for (i=0; i<N; i++)
         suba(&i);
```

- In general there is more flexibility than this ...

Programming Models

Shared Memory

- Typically, compilers parallelize at the loop level
- To ensure correct results (!), an (extensive) data dependency analysis is required
- Two major compiler issues:
  1. Get the right answer (ALWAYS)
     choose a conservative approach
  2. Not all information is known at compile time
     assume worst case, because of 1

Programming Models

Message Passing or Shared Memory ?

- It sometimes looks as if some sort of a religious war is going on about this issue
- It is really a matter of preference and goals
- Some observations, but no recommendation

Programming Models

Message Passing	Shared Memory
Takes more time to implement	Easier to implement
More details to worry about	System takes care of many details
Increases number of source lines	Little increase in source lines
More cumbersome to maintain	Little additional maintenance cost
Complex to debug and tune	Relatively easy to debug and tune
Increase of memory usage	Very efficient memory usage
May be best for optimal scalability	May be harder to achieve scalability
Parallelism is obviously visible	Parallelism may depend on compiler

Programming Models

The Hybrid Programming Model

Two level parallelism:

- Coarse grained parallelization using message passing

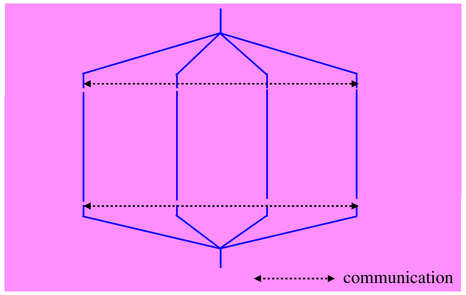
combined with

- Fine grained parallelization using shared memory (and the compilers to do this)

Coarse grained

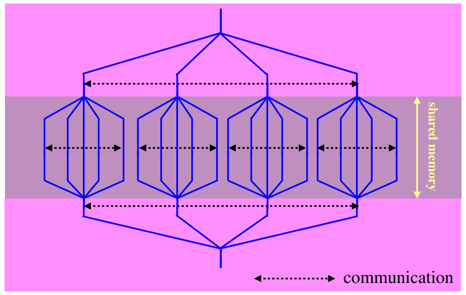
Programming Models

One level Parallel Programming Model



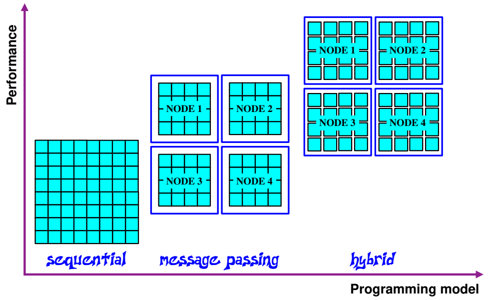
Programming Models

The Hybrid Parallel Programming Model



Programming Models

Example of the Hybrid Programming Model



Programming Models

What can you do ..... TODAY?

Hey Ruud,

This is all nice, but I don't have a budget for a big parallel system

Programming Models

What can you do ..... TODAY?

Even on a single CPU SGI workstation you can:

- **Develop and execute any parallel code**
  - **This includes the hybrid model**
- **Debug your code**
- **The only thing you can't achieve is a decrease in turnaround time (but we can sell you a parallel system to solve that problem .....)**