

Parallel Hardware Architectures and Parallel Programming Models

Rolf Rabenseifner
rabenseifner@hlrs.de

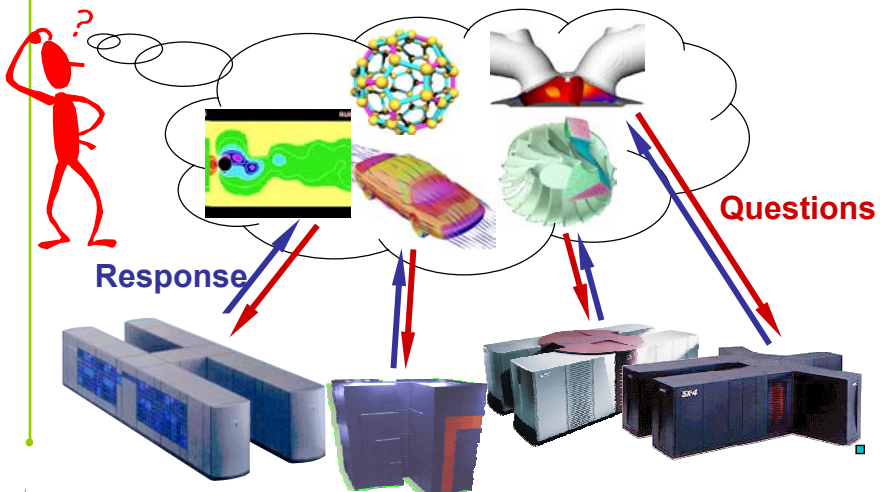
University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de



Hardware Architectures & Parallel Programming Models
Slide 1
Höchstleistungsrechenzentrum Stuttgart

H L R I S

Motivation

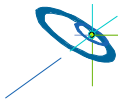


Hardware Architectures & Parallel Programming Models
Slide 2
Höchstleistungsrechenzentrum Stuttgart

H L R I S

Outline

- Parallel hardware architectures
- Parallel programming models
- Which parallel programming model is the best for my application?



Concepts

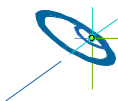
Parallel Processing concepts:

- Pipelining -> vector computing
 - Functional Parallelism -> modern processor technology
 - Combined instructions -> e.g. multiply-add as one instruction
 - Multithreading
 - Array-Processing
 - Multiprocessors (strongly coupled) -> Shared memory
 - Multicomputers (weakly coupled) -> Distributed memory
- } Hybrid architectures

Memory access concepts:

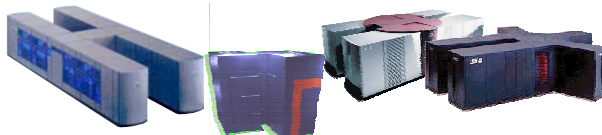
- Cache based
- Vector access via several memory banks
- Pre-load, pre-fetch

—> MFLOP/s performance **and** MB/s or Mword/s memory bandwidth



Major Parallel Hardware Architectures

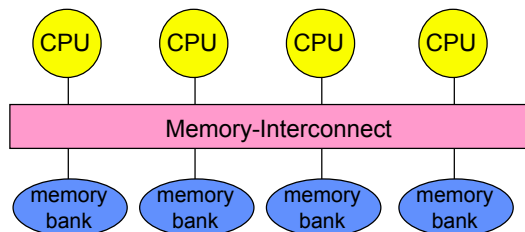
- Shared Memory
 - SMP = symmetric multiprocessing
- Distributed Memory
 - DMP = distributed memory parallel
- Hierarchical memory systems
 - combining both concepts



Hardware Architectures & Parallel Programming Models
Slide 5 Höchstleistungsrechenzentrum Stuttgart

HLRS

Multiprocessor - shared memory



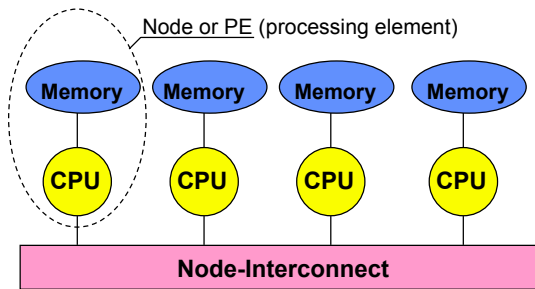
- All CPUs are connected to all memory banks with same speed
- **Uniform Memory Access (UMA)**
- **Symmetric Multi-Processing (SMP)**
- Network types, e.g.
 - Crossbar → independent access from each CPU
 - BUS → one CPU can *block* the memory access of the other CPUs



Hardware Architectures & Parallel Programming Models
Slide 6 Höchstleistungsrechenzentrum Stuttgart

HLRS

Multicomputer - distributed memory

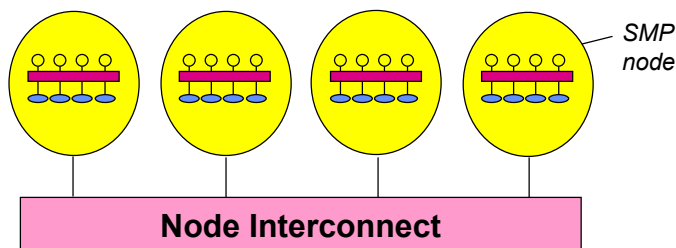


- Nodes are coupled by a node-interconnect
- Each CPU: – Fast access to its own memory
– but slower access to other CPU's memories
- **Non-Uniform memory Access (NUMA)**
- Different network types, e.g. BUS, torus, crossbar



Hybrid architectures


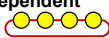
- Most modern high-performance computing (HPC) systems are clusters of SMP nodes



- SMP (symmetric multi-processing) inside of each node
- DMP (distributed memory parallelization) on the node interconnect



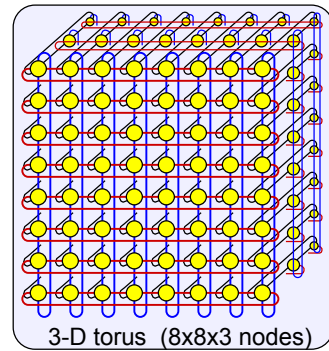
Interconnects

- Memory interconnect
 - bus
 - cross-bar
- Node interconnect
 - bus based networks
 - 
 - multi-link networks, e.g.,
 - ring with independent connections 
 - 2-D or 3-D torus
 - each processor is connected by a link with 4 or 6 neighbors
 - hierarchical networks
 - multi-level cross-bars
 - cross-bar (single level)
 - full interconnect

cheap,
but poor
inter-
connect

scalable
network
costs,
high accumulated
bandwidth

not scalable! — $n*(n-1)/2$ links

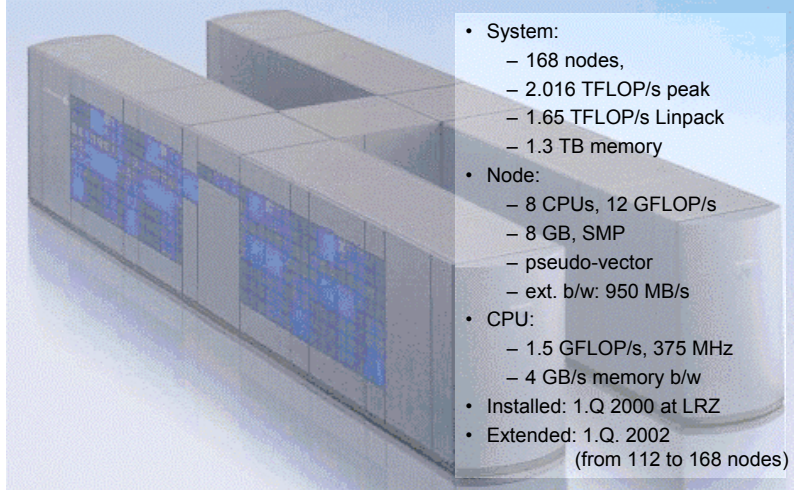


Other Architectures

- ccNUMA (cache coherent non-uniform memory access)
 - a distributed (hybrid) architecture
 - looks like one big SMP
 - programmable like one big SMP
 - but cluster of several small SMPs in reality
 - cache coherent
 - programming:
 - global access with same load/store instruction as local
 - parallelization, e.g., with OpenMP
- ccNUMA with >500 CPUs and multi-level network
 - parallelization, e.g., with Multi Level Parallelism (MLP)
- DMP with RDMA (remote direct memory access)
 - programming:
 - global memory access with special instructions, but without OS
 - e.g. Co-array Fortran, UPC (Universal Parallel C), shmem
- MTA (multi-threaded architecture)



Hitachi SR 8000-F1/112 (Rank 5 in TOP 500 / June 2000)



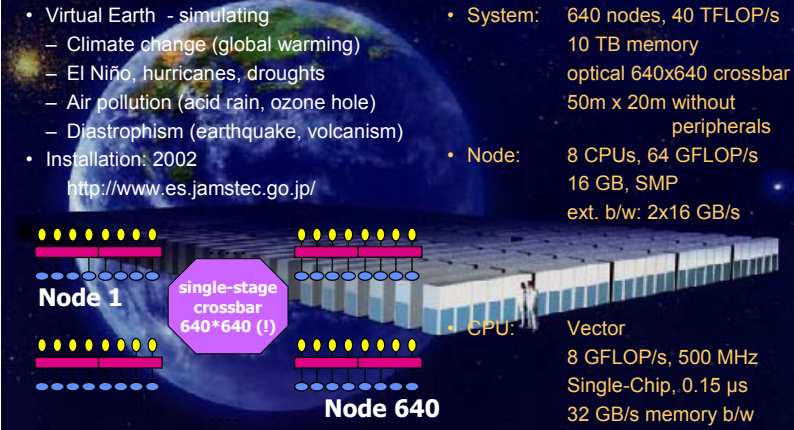
- System:
 - 168 nodes,
 - 2.016 TFLOP/s peak
 - 1.65 TFLOP/s Linpack
 - 1.3 TB memory
- Node:
 - 8 CPUs, 12 GFLOP/s
 - 8 GB, SMP
 - pseudo-vector
 - ext. b/w: 950 MB/s
- CPU:
 - 1.5 GFLOP/s, 375 MHz
 - 4 GB/s memory b/w
- Installed: 1.Q. 2000 at LRZ
- Extended: 1.Q. 2002
(from 112 to 168 nodes)



Hardware Architectures & Parallel Programming Models
Slide 11 Höchstleistungsrechenzentrum Stuttgart

H L R I S

Earth Simulator Project ESRDC / GS 40 (NEC)



- Virtual Earth - simulating
 - Climate change (global warming)
 - El Niño, hurricanes, droughts
 - Air pollution (acid rain, ozone hole)
 - Diastrophism (earthquake, volcanism)
- Installation: 2002
<http://www.es.jamstec.go.jp/>
- System: 640 nodes, 40 TFLOP/s
10 TB memory
optical 640x640 crossbar
50m x 20m without peripherals
- Node: 8 CPUs, 64 GFLOP/s
16 GB, SMP
ext. b/w: 2x16 GB/s
- CPU: Vector
8 GFLOP/s, 500 MHz
Single-Chip, 0.15 μ s
32 GB/s memory b/w

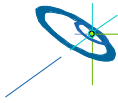


Hardware Architectures & Parallel Programming Models
Slide 12 Höchstleistungsrechenzentrum Stuttgart

H L R I S

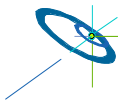
Outline

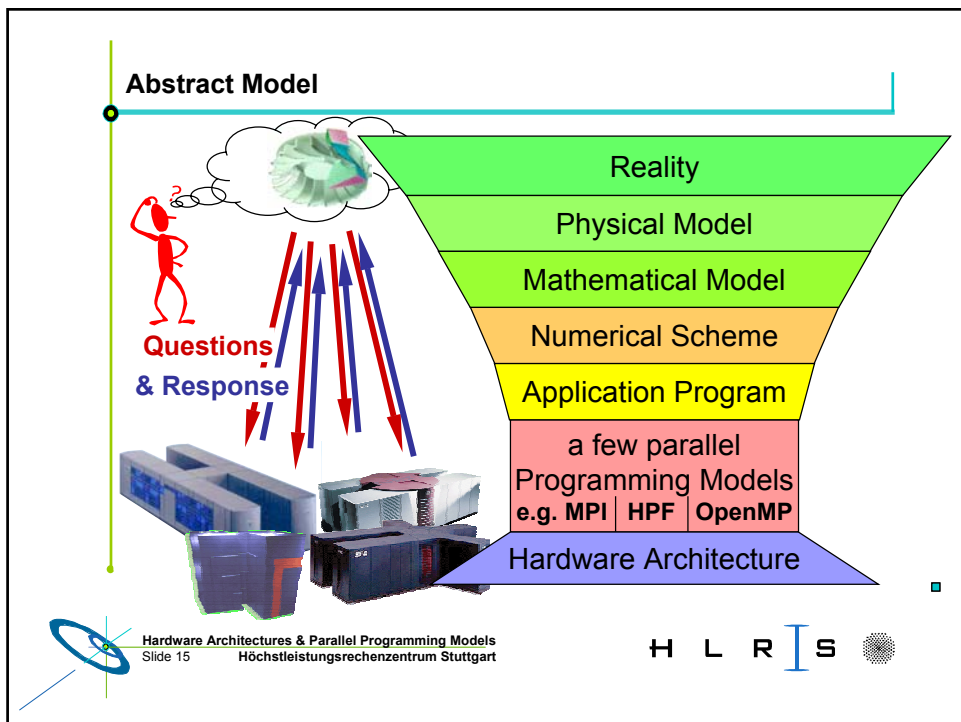
- Parallel hardware architectures
- **Parallel programming models**
- Which parallel programming model is the best for my application?



Why?

- Why should I use parallel hardware architectures?
- Possible answers:
 - The response of only one processor is **not** just in time
 - Moore's Law:
 - The number of transistors on a chip will double approximately every 18 month
 - ➔ in the future, the number of processors on a chip will grow
 - You own a
 - network of workstations (NOW)
 - Beowulf-class systems
= Clusters of Commercial Off-The-Shelf (COTS) PCs
 - a dual-board or quad-board PC
 - Huge application with huge memory needs





Parallelization strategies — hardware resources

- Two major resources of computation:
 - processor
 - memory
- Parallelization means
 - distributing work** to processors
 - distributing data** (if memory is distributed)
 and
 - synchronization** of the distributed work
 - communication** of *remote* data to *local* processor (if memory is distr.)
- Programming models offer a combined method for
 - distribution of work & data, synchronization and communication

Hardware Architectures & Parallel Programming Models
Slide 16 Höchstleistungsrechenzentrum Stuttgart

H L R I S

Distributing Work & Data

Work decomposition

- based on loop decomposition

do i=1,100

→ i=1,25

i=26,50

i=51,75

i=76,100

Data decomposition

- all work for a local portion of the data is done by the local processor

A(1:20, 1: 50)

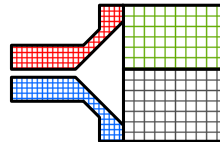
A(1:20, 51:100)

A(21:40, 1: 50)

A(21:40, 51:100)

Domain decomposition

- decomposition of work and data is done in a higher model, e.g. in the reality



Synchronization

Do i=1,100

a(i) = b(i)+c(i)

Enddo

Do i=1,100

d(i) = 2*a(101-i)

Enddo

i=1..25 | 26..50 | 51..75 | 76..100

execute on the 4 processors

BARRIER synchronization

i=1..25 | 26..50 | 51..75 | 76..100

execute on the 4 processors

Synchronization

- is necessary
- may cause
 - idle time on some processors
 - overhead to execute the synchronization primitive



Communication

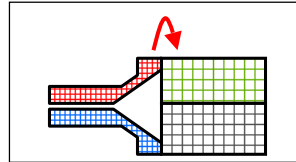
```
Do i=2,99
  b(i) = a(i) + f*(a(i-1)+a(i+1)-2*a(i))
Enddo
```

- **Communication** is necessary on the boundaries

- e.g. $b(26) = a(26) + f*(a(25)+a(27)-2*a(26))$

| | |
|------------|-----------|
| a(1:25), | b(1:25) |
| a(26,50), | b(51,50) |
| a(51,75), | b(51,75) |
| a(76,100), | b(76,100) |

- e.g. at domain boundaries



Major Programming Models

1 OpenMP

- Shared Memory **Directives**
- to define the work decomposition
- no data decomposition
- synchronization is implicit (can be also user-defined)

- **HPF (High Performance Fortran)**

- Data Parallelism
- User specifies data decomposition with **directives**
- Communication (and synchronization) is implicit

- **MPI (Message Passing Interface)**

- User specifies how work & data is distributed
- User specifies how and when communication has to be done
- by calling MPI communication **library-routines**



Shared Memory Directives – OpenMP, I.

Real :: A(n,m), B(n,m)

➔ Data definition

!\$OMP PARALLEL DO

do j = 2, m-1

➔ Loop over y-dimension

do i = 2, n-1

➔ Vectorizable loop over x-dimension

B(i,j) = ... A(i,j)

➔ Calculate B,
using upper and lower,
left and right value of A

... A(i-1,j) ... A(i+1,j)

... A(i,j-1) ... A(i,j+1)

end do

end do

!\$OMP END PARALLEL DO



Shared Memory Directives – OpenMP, II.

Single Thread



Master Thread

Parallel Region



Team of Threads

!\$OMP PARALLEL

!\$OMP END PARALLEL

Single Thread



Master Thread

Parallel Region



Team of Threads

!\$OMP PARALLEL

!\$OMP END PARALLEL

Single Thread

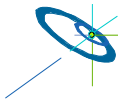


Master Thread



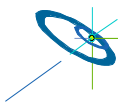
Shared Memory Directives – OpenMP, III.

- OpenMP
 - standardized shared memory parallelism
 - thread-based
 - the user has to specify the work distribution explicitly with directives
 - no data distribution, no communication
 - mainly loops can be parallelized
 - compiler translates OpenMP directives into thread-handling
 - standardized since 1997
- Automatic SMP-Parallelization
 - e.g., Compas (Hitachi), Autotasking (NEC)
 - thread based shared memory parallelism
 - with directives (similar programming model as with OpenMP)
 - supports automatic parallelization of loops
 - similar to automatic vectorization ■



Major Programming Models – HPF

- ① OpenMP
 - Shared Memory **Directives**
 - to define the work decomposition
 - no data decomposition
 - synchronization is implicit (can be also user-defined)
- ② HPF (High Performance Fortran)
 - Data Parallelism
 - User specifies data decomposition with **directives**
 - Communication (and synchronization) is implicit
- MPI (Message Passing Interface)
 - User specifies how work & data is distributed
 - User specifies how and when communication has to be done
 - by calling MPI communication **library-routines**



Data Parallelism – HPF, I.

Real :: A(n,m), B(n,m)

➡ Data definition

!HPF\$ DISTRIBUTE A(block,block), B(...)

do j = 2, m-1

➡ Loop over y-dimension

do i = 2, n-1

➡ Vectorizable loop over x-dimension

B(i,j) = ... A(i,j)

➡ Calculate B,

... A(i-1,j) ... A(i+1,j)

➡ using upper and lower,
left and right value of A

... A(i,j-1) ... A(i,j+1)

end do

end do



Data Parallelism – HPF, II.

- HPF (High Performance Fortran)
 - standardized data distribution model
 - the user has to specify the data distribution explicitly
 - Fortran with language extensions and directives
 - compiler generates message passing or shared memory parallel code
 - work distribution & communication is implicit
 - set-compute-rule:
the owner of the left-hand-side object computes the right-hand-side
 - typically arrays and vectors are distributed
 - draft HPF-1 in 1993, standardized since 1996 (HPF-2)
 - JaHPF since 1999



Major Programming Models – MPI

① OpenMP

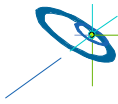
- Shared Memory **Directives**
- to define the work decomposition
- no data decomposition
- synchronization is implicit (can be also user-defined)

② HPF (High Performance Fortran)

- Data Parallelism
- User specifies data decomposition with **directives**
- Communication (and synchronization) is implicit

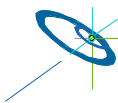
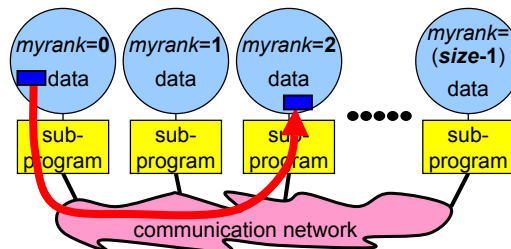
③ MPI (Message Passing Interface)

- User specifies how work & data is distributed
- User specifies how and when communication has to be done
- by calling MPI communication **library-routines**

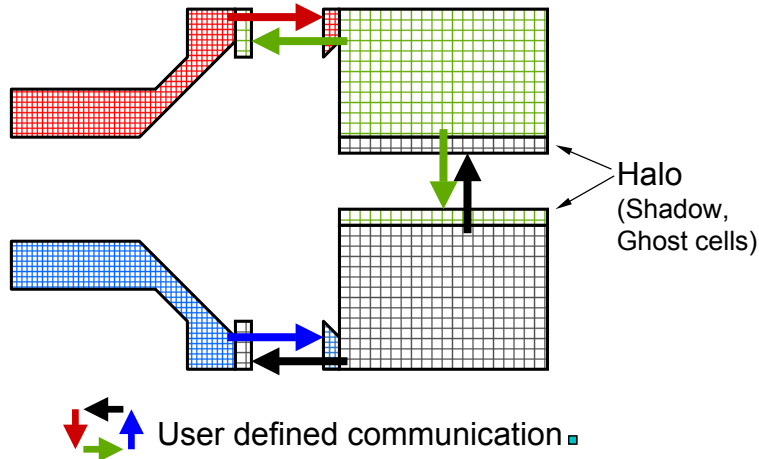


Message Passing Program Paradigm – MPI, I.

- Each processor in a message passing program runs a **sub-program**
 - written in a conventional sequential language, e.g., C or Fortran,
 - typically the same on each processor (SPMD)
- All work and data distribution is based on value of **myrank**
 - returned by special library routine
- Communication via special send & receive routines (**message passing**)



Additional Halo Cells – MPI, II.



Message Passing – MPI, III.

```
Call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
Call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierror)
m1 = (m+size-1)/size; ja=1+m1*myrank; je=max(m1*(myrank+1), m)
jax=ja-1; jex=je+1 // extended boundary with halo
```

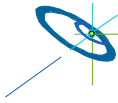
| | |
|--------------------------------------|--------------------------------------|
| Real :: A(n, jax:jex), B(n, jax:jex) | ➡ Data definition |
| do j = max(2,ja), min(m-1,je) | ➡ Loop over y-dimension |
| do i = 2, n-1 | ➡ Vectorizable loop over x-dimension |
| B(i,j) = ... A(i,j) | ➡ Calculate B, |
| ... A(i-1,j) ... A(i+1,j) | using upper and lower, |
| ... A(i,j-1) ... A(i,j+1) | left and right value of A |
| end do | |
| end do | |

```
Call MPI_Send(.....) ! - sending the boundary data to the neighbors
Call MPI_Recv(.....) ! - receiving from the neighbors,
                      ! storing into the halo cells
```



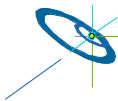
Summary — MPI, IV.

- MPI (Message Passing Interface)
 - standardized distributed memory parallelism with message passing
 - process-based
 - the user has to specify the work distribution & data distribution & all communication
 - synchronization implicit by completion of communication
 - the application processes are calling MPI library-routines
 - compiler generates normal sequential code
 - typically domain decomposition is used
 - communication across domain boundaries
 - standardized
 - MPI-1: Version 1.0 (1994), 1.1 (1995), 1.2 (1997)
 - MPI-2: since 1997



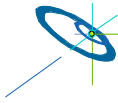
Distribution methods

| Decomposition | easiest Model | Memory distribution & communication | Work distribution |
|---------------|---------------|--|---|
| • Work | OpenMP | – none – | → explicit by program (with directives) |
| • Data | HPF | → explicit by program (with directives) & implicit comm. | → implicit by set-compute-rule or explicit with ON directive |
| • Domain | MPI | → explicit by program (via process' ranks) & explicit communication (with MPI library routines) | → explicit by program (via process' ranks) |



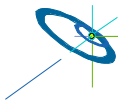
Limitations, I.

- Automatic Parallelization
 - the compiler
 - has no global view
 - cannot detect independencies, e.g., of loop iterations
 - parallelizes only parts of the code
 - only for shared memory and ccNUMA systems, see OpenMP
- OpenMP
 - only for shared memory and ccNUMA systems
 - mainly for loop parallelization with directives
 - only for medium number of processors
 - explicit domain decomposition also via rank of the threads



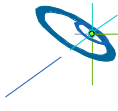
Limitations, II.

- HPF
 - set-compute-rule may cause a lot of communication
 - HPF-1 (and 2) not suitable for irregular and dynamic data
 - JaHPF may solve these problems, but with additional programming costs
 - can be used on any platform
- MPI
 - the amount of your hours available for MPI programming
 - can be used on any platform, but communication overhead on shared memory systems



Other Concepts

- shmem and MPI-2 one-sided communication
- Distributed memory programming (DMP) language extensions
 - Co-array Fortran
 - UPC (Unified Parallel C)
- Multi level parallelism (MLP)
- Threads: A single process having multiple execution paths
- Remote Memory Operation: A set of processes in which one process can access the memory of another process without its participation
- Shared Virtual Memory (SVM)
Software based Distributed Shared Memory (SoftDSM)
Distributed Virtual Shared Memory (DVSM)



SHMEM - Shared Memory Interface

- SHMEM allows a user to access remote memory locations with `shmem_..._put()` and `shmem_..._get()` routines.
- For parallel machines with global address space, this means no OS intervention => high bandwidth and low latency.
- Targeted for SPMD programs.
- No forced syncs: User has control of (and responsibility for) integrity of data from remote transfers.
- High BW, low latency and minimal syncs make SHMEM very fast, but dangerous if not carefully used.
- Cache coherency must be programmed explicitly.
- Example Cray T3E:
 - MPI and SHMEM bandwidth ~ the same
 - MPI latency about 10x longer than SHMEM latency



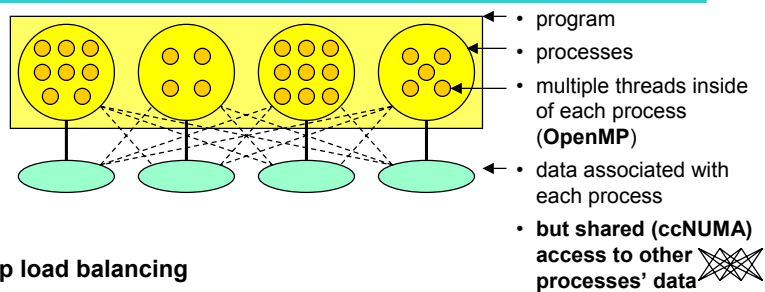
DMP Language Extensions, I.

- Programmable access to the memory of the other processes
- Language bindings:
 - Co-array Fortran
 - UPC (Unified Parallel C)
- Special additional array index to explicitly address the process
- Examples (Co-array Fortran):

| | |
|---------------------------------|--|
| integer a[*], b[*] | ! Replicate a and b on all processes |
| a[1] = b[6] | ! a on process 1 := b on process 6 |
| <hr/> | |
| dimension (n,n) :: u[3,*] | ! Allocates the nxn array u |
| | ! on each of the 3x* processes |
| p = THIS_IMAGE(u,1) | ! first co-subscript of local process |
| q = THIS_IMAGE(u,1) | ! second co-subscript of local process |
| u(1:n,1)[p+1,q] = u(1:n,n)[p,q] | ! Copy right boundary u(1,.) on process [p,] |
| | ! to right neighbor [p+1,] into left boundary u(n,.) |

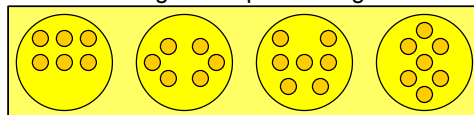


Multi Level Parallelism (MLP)



Cheap load balancing

- by changing the number of threads per process
- before starting a new parallel region



Programming Models on Hardware Platforms

Hardware allows:

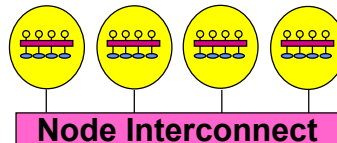
→ Usable programming model:

- only reliable message transfer → MPI, HPF
- remote DMA (direct memory access) → "", "" + SVM, shmem, UPC, Co-array Fortran
- SMP and PVP, MTA, ccNUMA → "", "" + "", "", "", "" + OpenMP



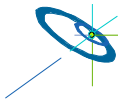
Programming Models on Hybrid Systems

- MPI based:
 - the MPP model
 - massively parallel processing
 - each CPU = one MPI process
 - MPI + OpenMP
 - each SMP node = one MPI process
 - MPI communication on the node interconnect
 - OpenMP inside of each SMP node
 - DMP with MPI & SMP with OpenMP
 - MPI + automatic parallelization
 - Compas on Hitachi, Autotasking on NEC, ...
 - same model as MPI+OpenMP
- Other models:
 - HPF, MLP, ... ■



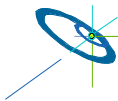
Outline

- Parallel hardware architectures
- Parallel programming models
- **Which parallel programming model is the best for my application?**



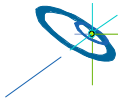
Which parallel programming model is the best for my application?

- no absolute answer
- only hints on the next slides

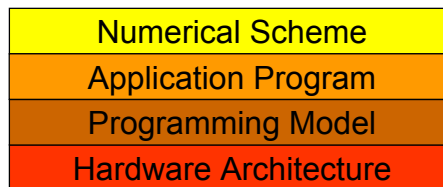


Why do I want to parallelize my code?

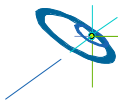
- time
 - parallelization of the work on many CPUs
 - to speedup the execution
- memory space
 - the application does not fit to the available memory of one CPU
 - parallelization on distributed memory



Additional questions



- Available parallelization strategies in my numerical scheme?
 - loop parallelism
 - domain decomposition
- Which hardware architecture?
 - today
 - in the future
- How many working hours do I want to spend for parallelizing the code?



Execution time

- My application runs too slow
 - (Floating point) operations / second on each CPU?
 - **vectorization** (memory→CPU→memory)
 - expensive hardware / cheap programming effort
 - **cache oriented optimization**
 - cheap hardware / expensive programming effort
 - **such optimization is impossible for me**
 - Parallelization
 - **shared memory** → [OpenMP](#), HPF, MPI
 - expensive & limited hardware / cheap programming effort
 - **distributed memory** → [MPI](#), [HPF](#), [shmem](#), [MLP](#), [CoArrayFort](#)., ...
 - cheap hardware / expensive programming effort

→ **Today / in the future** ■



Speedup, Efficiency, and Scaleup

- Definition:
 - $T(p, N)$ = **time** to solve **problem of size N** on **p processors**
- Speedup:
 - $S(p, N) = T(1, N) / T(p, N)$
 - compute **same problem** with more processors in **shorter time**
- Efficiency:
 - $E(p, N) = S(p, N) / p$
- Scaleup:
 - $Sc(p, N) = N / n$ with $T(1, n) = T(p, N)$
 - compute **larger problem** with more processors in **same time**
- Problems:
 - Absolute MFLOPS rate / hardware peak performance?
 - $S(p, N)$ close to **p** or far less? → see Amdahls Law on next slide
 - Or super-scalar speedup: $S(p, N) > p$, e.g., due to cache usage



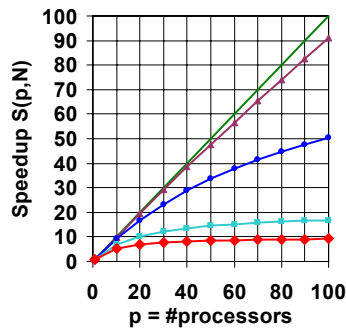
Amdahls Law

$$T(p,N) = f \cdot T(1,N) + (1-f) \cdot T(1,N) / p$$

f ... sequential part of code that can not be done in parallel

$$S(p,N) = T(1,N) / T(p,N) = 1 / (f + (1-f) / p)$$

For $p \rightarrow \infty$, speedup is limited by $S(p,N) < 1 / f$



— $S(p,N) = p$ (ideal speedup)

— $f=0.1\% \Rightarrow S(p,N) < 1000$

— $f= 1\% \Rightarrow S(p,N) < 100$

— $f= 5\% \Rightarrow S(p,N) < 20$

— $f= 10\% \Rightarrow S(p,N) < 10$



Hardware Architectures & Parallel Programming Models
Slide 47
Höchstleistungsrechenzentrum Stuttgart

H L R I S

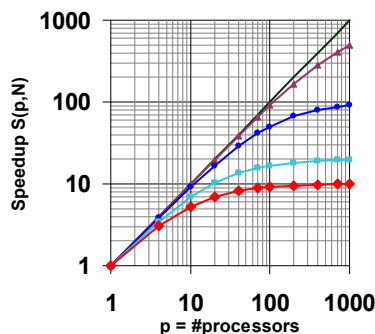
Amdahls Law (double-logarithmic)

$$T(p,N) = f \cdot T(1,N) + (1-f) \cdot T(1,N) / p$$

f ... sequential part of code that can not be done in parallel

$$S(p,N) = T(1,N) / T(p,N) = 1 / (f + (1-f) / p)$$

For $p \rightarrow \infty$, speedup is limited by $S(p,N) < 1 / f$



— $S(p,N) = p$ (ideal speedup)

— $f=0.1\% \Rightarrow S(p,N) < 1000$

— $f= 1\% \Rightarrow S(p,N) < 100$

— $f= 5\% \Rightarrow S(p,N) < 20$

— $f= 10\% \Rightarrow S(p,N) < 10$

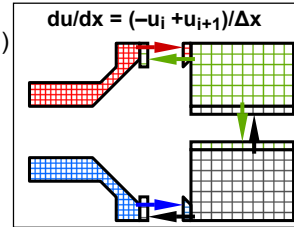


Hardware Architectures & Parallel Programming Models
Slide 48
Höchstleistungsrechenzentrum Stuttgart

H L R I S

Parallelization problems

- Two major resources of computation:
 - processor
 - memory
- Parallelization means
 - distributing work to processors
 - load balancing necessary
 - synchronization overhead should be minimized
 - to achieve optimal speedup
 - distributing data (if memory is distributed)
 - implies communication to bring data to processor
 - communication is overhead
 - is reducing the speedup



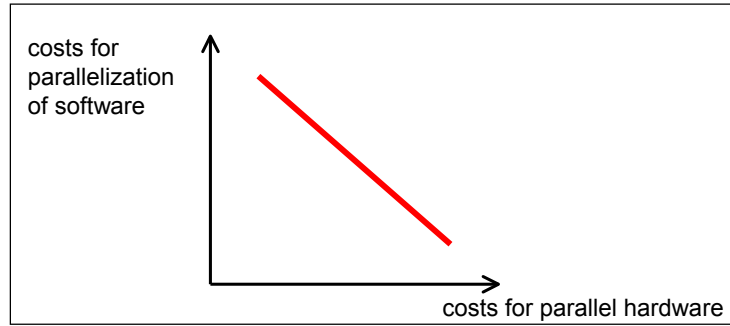
Which hardware should I use / buy?

- Network Of Workstations (NOW)
 - Beowulf class system /
Clusters of Commercial Off-The-Shelf PCs (COTS)
 - PVP (Parallel Vector Processor)
with shared memory
 - Hybrid systems (cluster of SMPs)
- many processors, distributed memory
- small number of CPUs
- medium number of CPUs
- for **same** hardware costs

Implications for software costs?



Parallelization costs



- low costs for parallel hardware → high parallelization costs
- high costs for parallel hardware → low parallelization costs



Advantages and Challenges

| | OpenMP | HPF | MPI |
|--|--------|-----|-----|
| Maturity of programming model | ++ | + | ++ |
| Maturity of standardization | + | + | ++ |
| Migration of serial programs | ++ | 0 | -- |
| Ease of programming (new progr.) | ++ | + | - |
| Correctness of parallelization | - | ++ | -- |
| Portability to any hardware architecture | - | ++ | ++ |
| Availability of implementations of the stand. | + | + | ++ |
| Availability of parallel libraries | 0 | 0 | 0 |
| Scalability to hundreds/thousands of processors | -- | 0 | ++ |
| Efficiency | - | 0 | ++ |
| Flexibility – dynamic program structures | - | - | ++ |
| – irregular grids, triangles, tetrahedrons, load balancing, redistribut. | - | - | ++ |



Implications on Hybrid Systems

- Hybrid system = cluster of SMPs, e.g., with vector CPUs
 - MPP (massively parallel processing) model (pure MPI):
 - one MPI process on each CPU
 - hybrid model: MPI+OpenMP or MPI+automatic parallelization
 - each MPI process is multi-threaded with OpenMP/...
 - lousy communication speed, if MPI is done only by master thread (all other threads are sleeping)
 - highest costs for parallelizing the software
 - Amdahl's law with reduced number of CPUs on several levels
 - HPF may also fit,
 - e.g., on the Earth Simulator in Japan ■



Why not hybrid models?

DMP (Distributed memory parallelism)

→ MPI

SMP (Symmetric multiprocessing)

→ OpenMP / automatic parallelization

cache / vectorization

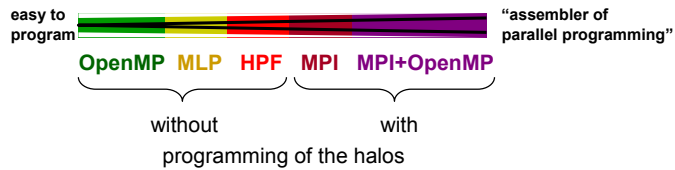
→ optimization by hand / by compiler

- Typically, hybrid models can achieve only 10 % more efficiency,
- but often: hybrid model less efficient than MPI-MPP model !!!
- Programming effort should be invested into
 - cache optimization
 - vectorization } you may win factors and not only percents !



Which Model is the Best for Me?

- Depends on
 - your application
 - your platform
 - which efficiency do you need on your platform
 - how much time do you want to spent on parallelization



Acknowledgements

- Thanks to Alfred Geiger and Michael Resch (HLRS)
 - pictures and slides from their *Parallel Programming* lectures
- Thanks to Uwe Küster (HLRS) and Tim Lanfear (Hitachi)
 - pictures about vectorization



Summary

- Hardware architectures
 - hybrid (hierarchical) systems are the future
 - cluster of dual-board PC
 - ...
 - clusters of PVP-SMP systems
- Parallel Programming models
 - MPI and OpenMP are dominating
 - HPF still alive (→ JaHPF on Earth Simulator)
- Which model is the best
 - depends on your needs & hardware, today and in future
 - OpenMP is limited to shared memory platforms, but may be extended with a data distribution model (like HPF)
 - MPI is the *assembler of parallel programming*
 - invest your working effort into single-CPU-optimization, rather than into hybrid programming



Appendix

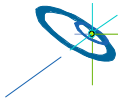
Additional Slides

- Abbreviations
- Classification of Flynn
- Co-Array Fortran examples
- MLP example and interface definition
- Pipelining and memory access
- Parallelization costs



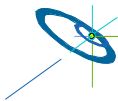
Abbreviations

- Network of workstations (NOW)? → Distributed memory
- Beowulf-class systems = Clusters of Commercial Off-The-Shelf (COTS) PCs → Distributed memory
- Multiboard workstations/PCs → Shared memory
- SMP → Symmetric multiprocessing → Shared memory
- PVP → Parallel vector processing
- MPP → Massively parallel processing
- PE → Processing Element, e.g., one node of an MPP system



The Classification of Flynn

- Classify architectures according to multiplicity of data and instructions
- SI: single instruction for all processors
- MI: multiple instructions for different processors
- SD: single data for all processors
- MD: multiple data for different processors
- SISD → classical processor
- SIMD → array processor
- MIMD → distributed or shared memory
- SPMD → single program & multiple data
- MPMD → multiple program & multiple data



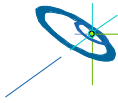
DMP Language Extensions – Co-Array Fortran Example

```
size = NUM_IMAGES()
p = THIS_IMAGE()      ! index of the invoking image (= MPI myrank+1)
m = size * m1

Real :: A(n, m1)[*], B(n, m1)[*]

do j = 2, m1-1
  do i = 2, n-1
    B(i,j) = ... A(i,j) ... A(i-1,j) ... A(i+1,j) ... A(i,j-1) ... A(i,j+1)
  end do
end do
if (p > 1) then          ! calculation on left boundary of each image
  do i = 1, n
    B(i,1) = ... A(i,1) ... A(i-1,1) ... A(i+1,1) ... A(i,m)[p-1] ... A(i,j+1)
  end do
endif
if (p < size) then       ! calculation on right boundary of each image
  do i = 1, n
    B(i,m) = ... A(i,m) ... A(i-1,m) ... A(i+1,m) ... A(i,m-1) ... A(i,1)[p+1]
  end do
endif

Call SYNC_IMAGES()
```

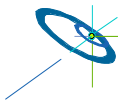


Hardware Architectures & Parallel Programming Models
Slide 61 Höchstleistungsrechenzentrum Stuttgart



Multi Level Parallelism (MLP)

- Two levels of parallelism (usually)
- Fine grained parallelism provided by the compiler (e.g., OpenMP) at loop level
- Coarse grained parallelism provided by forked processes
- communication by shared memory arenas, i.e. direct access to global arrays by compiler generated code
- Minimal latency (0.33–1.0 µsec on 512 processor Origin2000)
- Only four additional routines: **INITMEM**, **GETMEM**, **FORKIT**, **BARRIER**
- Targeted for large CPU count NUMA SMP systems
- Efficient and easy load balancing on ccNUMA, e.g., by adapting the number of threads on each process
- Method can also execute across clusters
- A Fortran interface for System V *shm*



Hardware Architectures & Parallel Programming Models
Slide 62 Höchstleistungsrechenzentrum Stuttgart

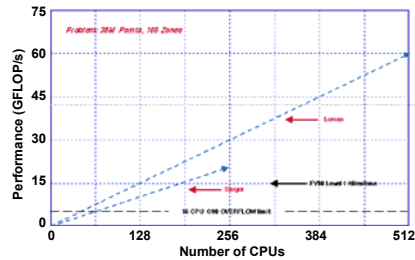
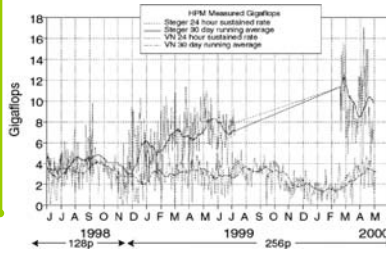


Example: Parallel Efficiency of OVERFLOW/MLP



- OVERFLOW CFD code at NASA/Ames
- high, sustained GFLOP/s rate
- with Multi Level Parallelism (MLP)
- scalable on large CPU counts
- on 512 processor ccNUMA Origin 2000

Ref.: Ciotti, Taft, Peterson: "Early Experiences with the 512p Origin2000" in proceedings of the Cray User Group conference SUMMIT 2000, www.cray.org



Hardware Architectures & Parallel Programming Models
Slide 63 Höchstleistungsrechenzentrum Stuttgart

H L R I S

MLPlib

The MLPlib routines for scalable parallel execution support are:

- Subroutine INITMEM(numbytes)
 - The INITMEM routine sets up a UNIX shared memory arena consisting of numbytes bytes to be used by all subsequently spawned processes
- Subroutine GETMEM(xarray,xpoint,numxbyt)
 - The GETMEM routine allocates numxbyt bytes to the xarray variable
 - xpoint is the Cray pointer to xarray
 - xarray is resident in the shared memory arena
 - The xarray data will be visible to all MLP processes using the shared memory arena.
- Subroutine FORKIT(numpro,myrank)
 - spawns a total of numpro additional processes
 - returns current process id myrank (0–numpro)
- Subroutine BARRIER(numpro)
 - The BARRIER routine waits until numpro processes have hit the barrier, then all drop through

Reference: Ciotti, Taft, Peterson: "Early Experiences with the 512p Origin2000" in proceedings of the Cray User Group conference SUMMIT 2000, www.cray.org

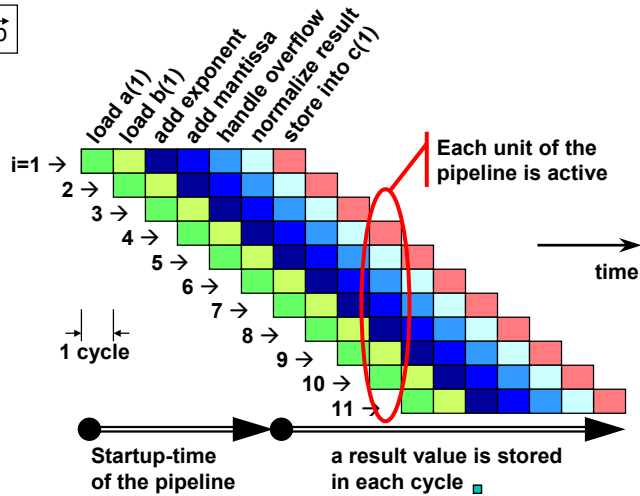


Hardware Architectures & Parallel Programming Models
Slide 64 Höchstleistungsrechenzentrum Stuttgart

H L R I S

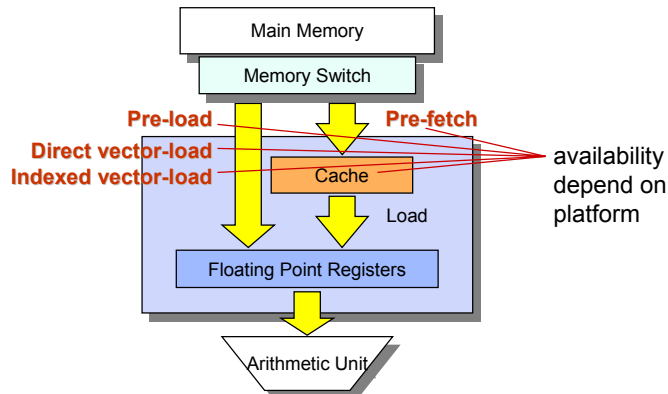
Pipelining

- $\vec{c} = \vec{a} + \vec{b}$



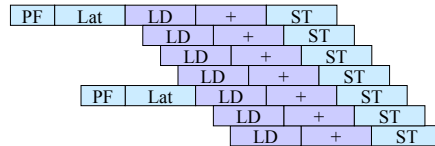
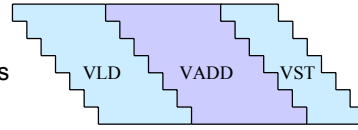
Parallelism & memory access, I.

- The memory access characterizes vector systems today!



Parallelism & memory access, II.

- Pipelined memory access
 - vector register & vector load/store operations
 - indexed vector load/store operations
 - pseudo-vectorization
 - e.g., with pre-fetch
- Parallel vector processing (PVP)
- **Without** vector memory access:
 - memory latency hiding with one to three levels of caches



Parallelization Costs on NOW (Network of Workstations) and COTS (Clusters of Commercial Off-The-Shelf PCs)

- Network of workstations (NOW) or Beowulf-class systems
 - = Clusters of Commercial Off-The-Shelf (COTS) PCs
 - probably a huge number of processors
 - distributed memory parallelization, e.g., with MPI or HPF
 - Amdahl's law may limit the speedup
 - communication overhead may reduce efficiency
 - high costs to correctly parallelize codes (e.g., multigrid codes)



Parallelization Costs on shared memory PVP (Parallel Vector Processor)

- Shared memory Parallel Vector Processing (PVP)
 - shared memory parallelization, e.g., with **OpenMP**
 - Amdahl's law with **reduced** number of CPUs, but on two levels:
 - vectorization
 - SMP parallelization
 - no communication overhead
 - probably easier to achieve good efficiency
 - but limited number of CPUs
 - cheap parallelization



but expensive hardware?

