

# Introduction to High Performance Computing

**Johan Alwall**, together with Jan Conrad

*High Energy Physics Division*

*Uppsala University.*



# CONTENTS

- Single Processor
  - RISC Architecture
- Multi Processor
  - Shared Memory Architecture
  - Distributed Memory Architecture
- Parallel Computing Resources in Sweden
- Where is more material found

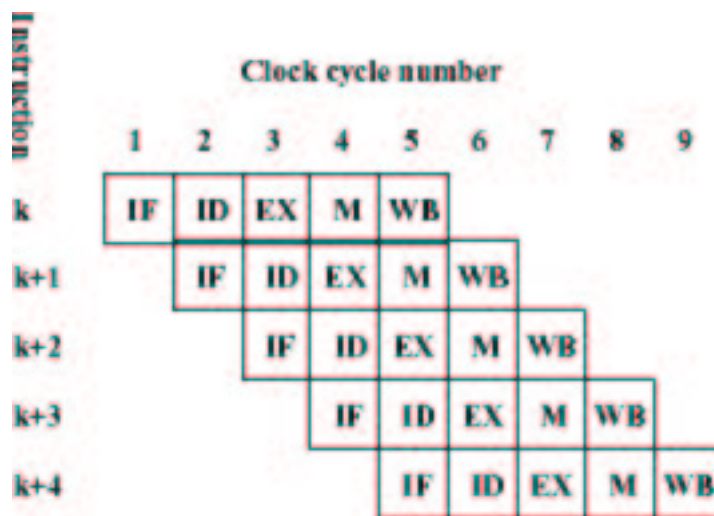
# Single Processor

- RISC Architecture
  - Pipelining
  - Floating point unit
- Memory structure
- Conclusion - Optimization

# RISC architecture

- Reduced instruction set computer
- Short instructions easy to pipeline
- As opposed to CISC:
  - Complex instruction set computer
  - Complicated, specialized instructions
  - Difficult to pipeline - Reduces clock frequency

# Pipelining



Typically but simplified:

**IF:** Instruction fetch cycle.

→ Fetch next instruction  
from memory to IR and increase PC

**ID:** Instruction decode

**EX:** Execution/effective address cycle, e.g.

→ ALU performs an operation

→ Compute branch address

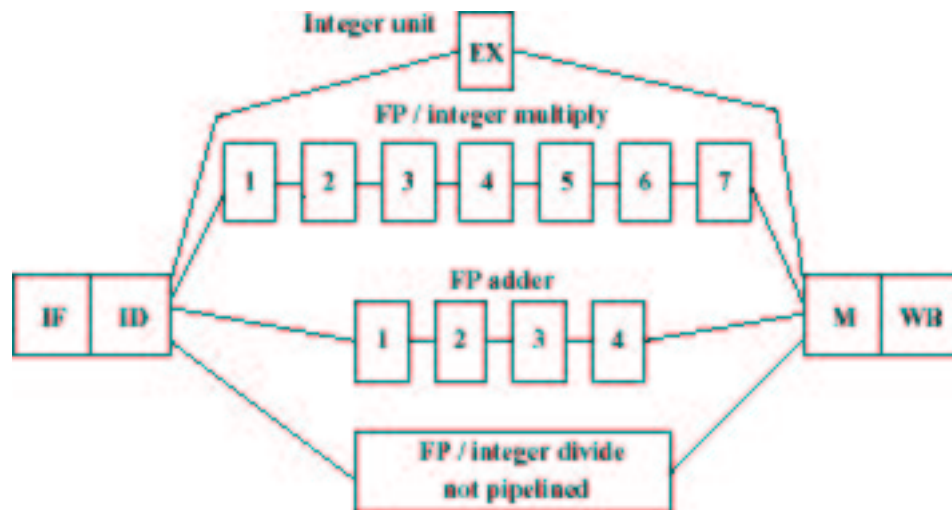
**M:** Memory access/branch completion cycle.

→ Load or store from/to memory or branch  
(update PC)

**WB:** Write-back cycle.

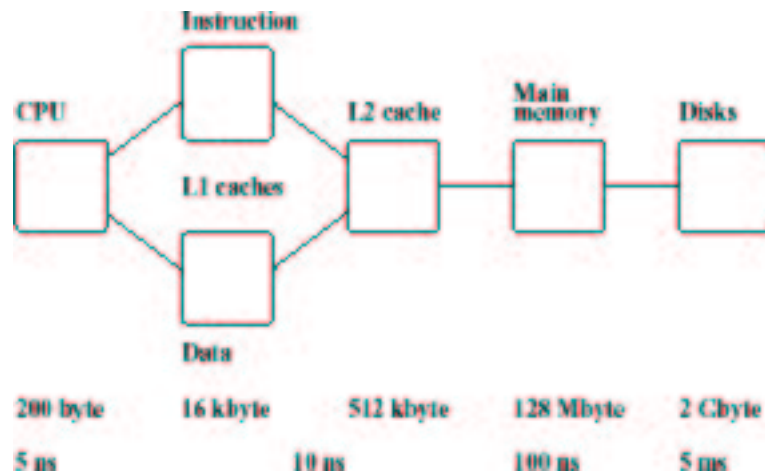
→ Write results from ALU or  
memory fetch into the register

## Floating point unit - the EX phase

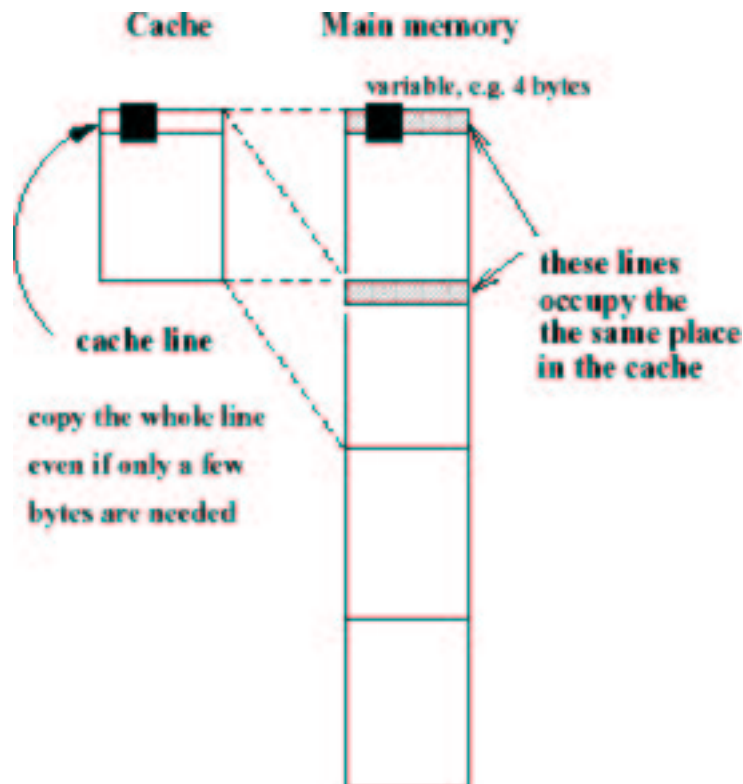


- All operations except division are pipelined
- All operations can be done in parallel
- Division is very time-consuming

# Memory structure

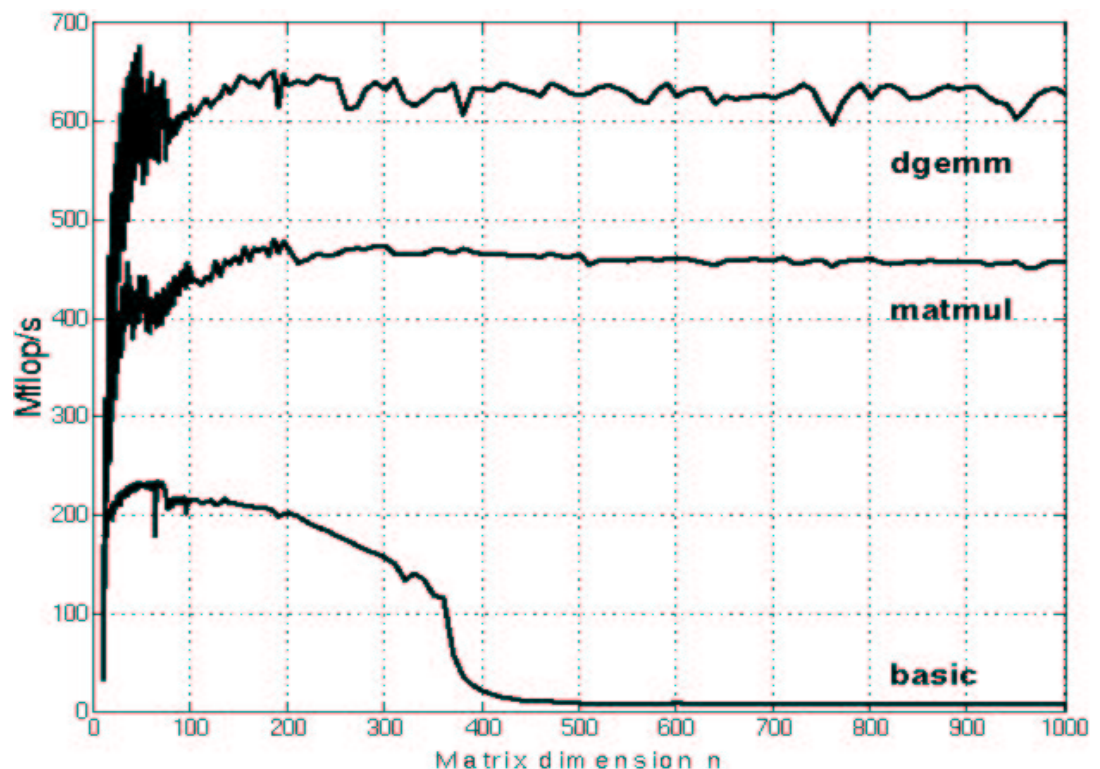


- Cache memory - fast but small
- Whole memory line always copied into cache memory
- Different algorithms for cache line replacement



## Conclusion - Optimization

- Make full use of pipeline - reduce unnecessary branching
- Don't divide - multiply!
- Never write your own matrix routines
- Use optimized routine libraries (e.g. BLAS 3 instead of BLAS 1 or 2)
- Program for locality - reuse data
  - Access matrix elements in the order they are stored
- Check out compiler optimization options - it does it better than you!



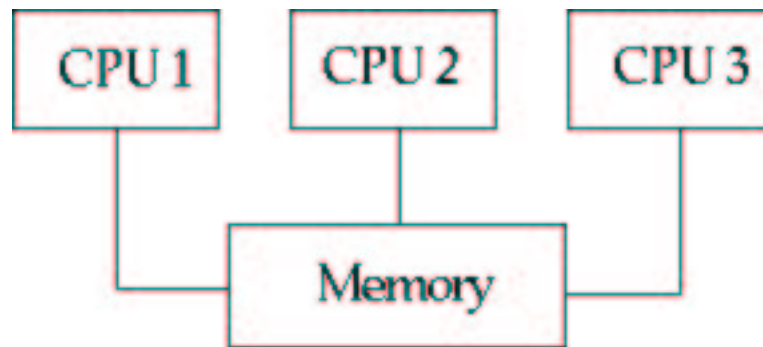
# Parallel Architectures

- Benefits and Costs of Parallel Programming
- Shared Memory Architecture
- Distributed Memory Architecture
- Applications and Examples

# Benefits and Costs of Parallel Programming

- Benefits of Parallel Processing:
  - Reduce wall-clock time
  - May be cheapest computing solution
  - Use non-local resources
  - Handle memory constraints
- Costs of Parallel Processing:
  - To analyze code for parallelism
  - To recode
  - Complicated debugging
  - Loss of portability of code
  - Total CPU time greater with parallel
  - Replication of code and data requires more memory

# Shared Memory Architecture



- Can utilize parallelism on loop level
- Minor changes to code necessary
- No partitioning of data
- Fork-join idea; master thread creates a team of threads when reaching a parallel part of the code
- Can parallelize the code automatically, by compiler directives, or by directives in the code
- Requires special shared-memory computers
- Program using e.g. OpenMP

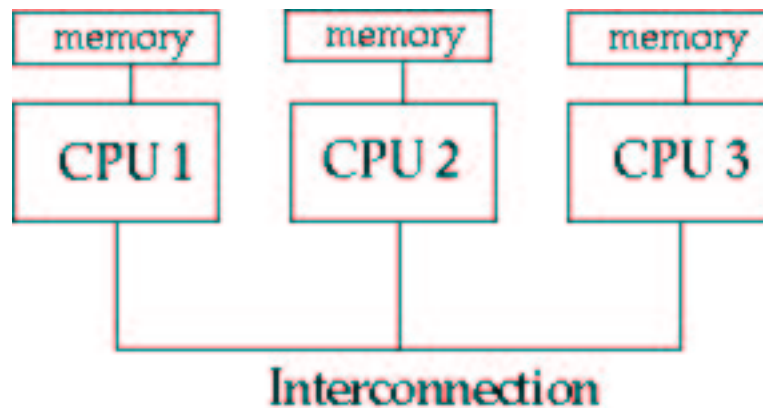
# OpenMP Code Example

```
main(){
    int i, A[100];

    #pragma omp parallel for
    #pragma omp private(i)
    #pragma omp shared(A)
    for (i=0; i<100; i++){
        A[i] = ...
        ...
    }
}
```

Variable *i* is private to each thread  
Array *A* is shared among threads

# Distributed Memory Architecture



- Requires large grain parallelism to be efficient
- Large rewrites of the code often necessary
- Domain decomposition; indexing relative to blocks
- Requires global understanding of the code
- Hard to debug
- Can be done on network of PCs
- Program using e.g. MPI

# MPI

- Message Passing Interface
- The first standard and portable message passing interface with good performance
- Program by calling special MPI methods
- Compile using special compiler

## MPI Code example (F90)

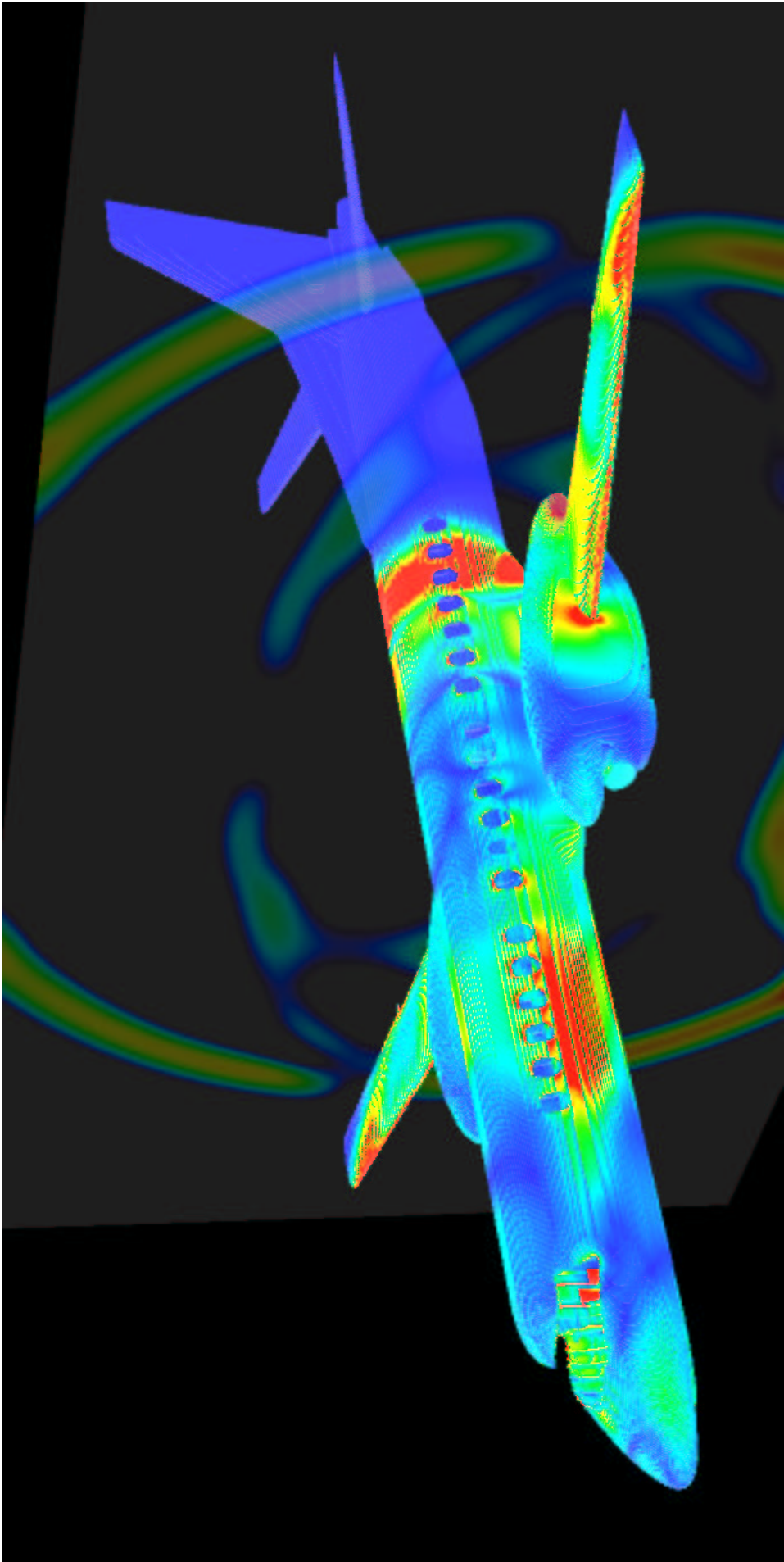
```
program hello
include 'mpif.h'
integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)
character(12) message
call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
tag = 100
if (rank .eq. 0) then
    message = 'Hello, world'
    do i=1, size-1
        call MPI_SEND(message, 12, MPI_CHARACTER, i,
            & tag, MPI_COMM_WORLD, ierror)
    enddo
else
    call MPI_RECV(message, 12, MPI_CHARACTER, 0,
        & tag, MPI_COMM_WORLD, status, ierror)
endif
print*, 'node', rank, ':', message
call MPI_FINALIZE(ierror)
end
```

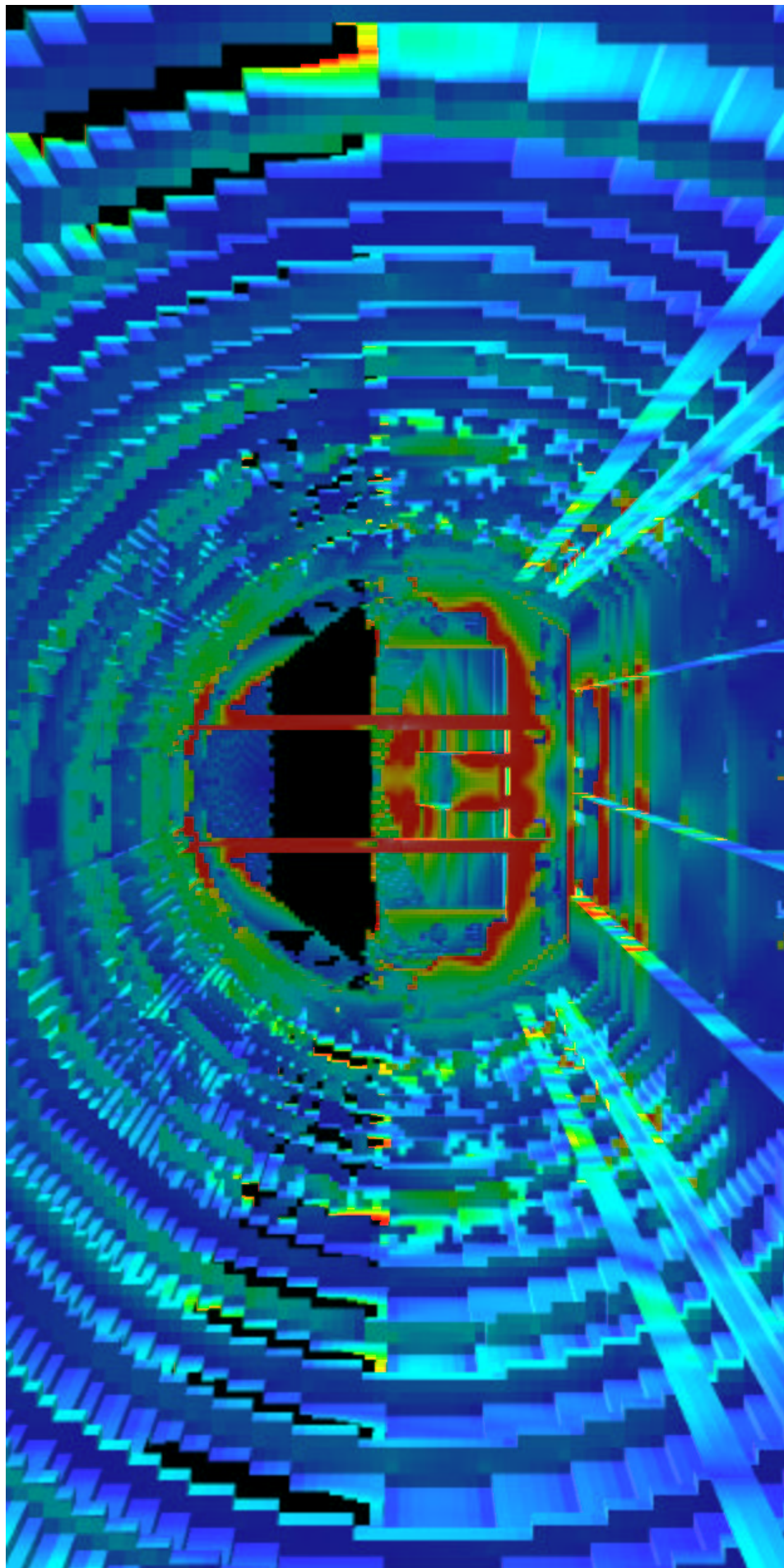
# Applications and Examples

- Partitioning Example, Finite Element Method
- Large-scale Electrodynamics Simulation
  - Lightning strikes SAAB 2000 aircraft
  - FD-TD-method
  - 1 billion cells used
  - 125 nodes used on an IBM SP
  - Figure 1: Surface currents after 1500 time steps and magnetic field on cutting plane across wings
  - Figure 2: Surface currents in interior, view towards cockpit. Large currents are shown in red.

# Parallel Computing Centra in Sweden

- PDC, <http://www.pdc.kth.se/>  
→ IBM SP, Fujitsu VX
- HPC2N, <http://www.hpc2n.umu.se/>  
→ IBM SP, SGI Onyx2
- NSC, <http://www.nsc.liu.se/>  
→ Cray T3E, SGI 3800, Beowolf, SGI Onyx2
- Here at the physics section in Uppsala  
→ 2 Sun 4800, 12 procs  
→ 8 2 processor-machines  
→ Each processor: 750 MHz, 1GB RAM (shared)





## Where is more material found

- This talk:
  - <http://www3.tsl.uu.se/theo/talks.html>
- Lecture notes at PDC:
  - <http://www.pdc.kth.se/training/2001/SummerSchool/CourseWork/ovning.html>
  - <http://www.pdc.kth.se/training/Talks/>