High-Productivity Languages

for

Peta-Scale Computing

Hans P. Zima

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA
and
University of Vienna, Austria
zima@jpl.nasa.gov

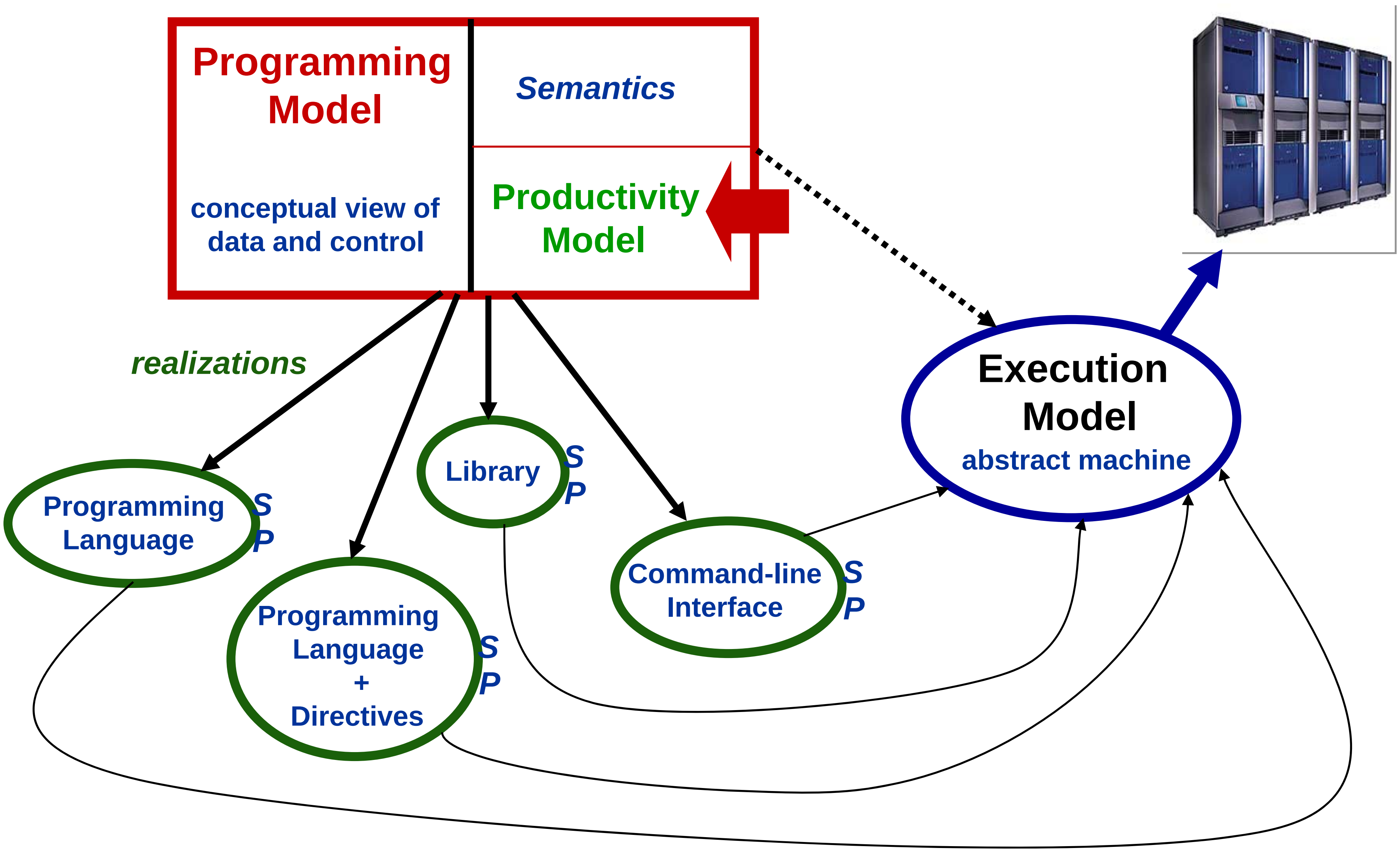
Fujitsu HPC Forum 2008
Tokyo, Japan, August 27th, 2008

Contents

The Meaning of “High-Productivity”

- **“High productivity” implies three properties:**
  1. *human-centric: programming at a high level of abstraction*
  2. *high-performance: providing “abstraction without guilt”*
  3. *reliability*
- **Raising the level of abstraction is acceptable only if target code performance is not significantly reduced**
- **This relates to a broad range of topics:**
  - *language design*
  - *architecture- and application-adaptive compiler technology*
  - *operating and runtime systems*
  - *library design and optimization*
  - *intelligent tool development*
  - *fault tolerance*

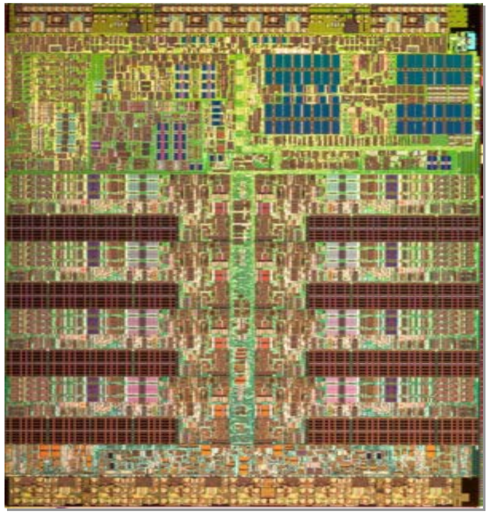
High-Productivity
Programming and Execution Models



Contents

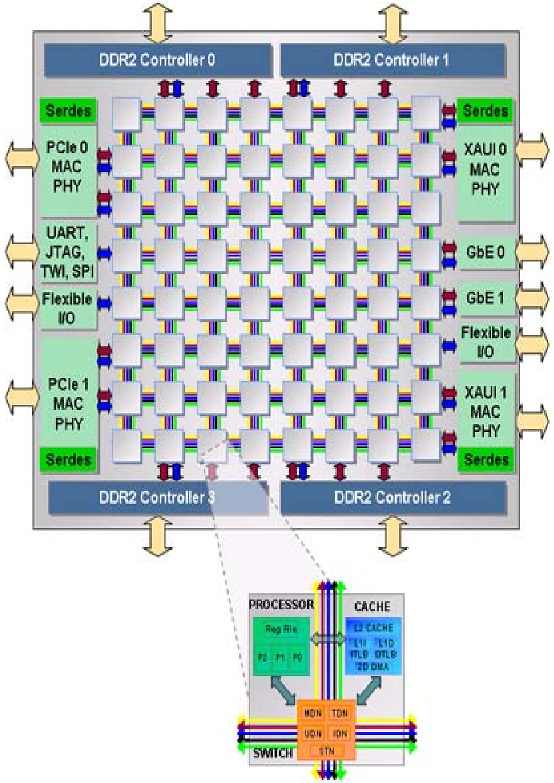
Multicore: An Emerging Technology

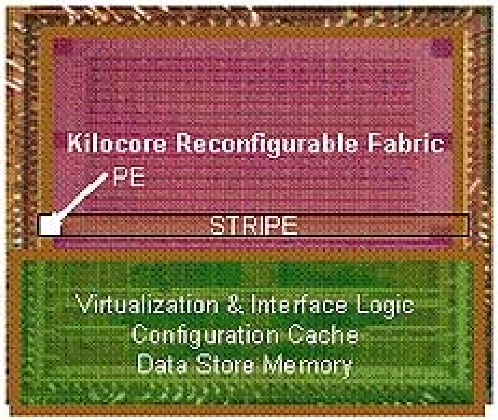
- **The era of faster sequential processors is over—exponential growth of frequency can no longer be maintained**
  - *CMOS manufacturing technology approaches physical limits*
  - *power wall, memory wall, instruction-level parallelism (ILP) wall*
  - *Moore's Law still in force: number of transistors on chip increasing*
- **Multicore technology provides continued performance growth**
  - *a multicore chip is a single chip with two or more independent processing units*
  - *improvements by multiple cores on a chip rather than higher frequency*
  - *on-chip resource sharing for cost and performance benefits*
- **Multicore systems have been produced since 2000**
  - *IBM Power 4; Sun Niagara; AMD Opteron; Intel Xeon;…*
  - *Quadcore systems by AMD, Intel*
  - *IBM/Sony/Toshiba: Cell Broadband Engine*
    - *Power Processor (PPE) and 8 Synergistic PEs (SPEs)*
    - *peak 100 GF double precision (IBM Power XCEll 8i)*
- **1000 cores on a chip possible with 30nm technology**
- **"Manycore" chips are already emerging …**



Future Multicore Architectures: From 10s to 100s of Processors on a Chip

- **Tile64 (Tilera Corporation, 2007)**
  - *64 identical cores, arranged in an 8X8 grid*
  - *iMesh on-chip network, 27 Tb/sec bandwidth*
  - *170-300mW per core; 600 MHz – 1 GHz*
  - *192 GOPS (32 bit)—about 10 GOPS/Watt*
- **Kilocore 1025 *(Rapport Inc. and IBM, 2008)***
  - *Power PC and 1024 8-bit processing elements*
  - *125 MHz per processing element*
  - *32X32 "stripes" dedicated to different tasks*
- **512-core SING chip (Alchip Technologies, 2008)**
  - *for GRAPE-DR, a Japanese supercomputer project*
- **80-core 2 TF research chip from Intel (2011)**
  - *2D on-chip mesh network for message passing*
  - *1.01 TF (3.16 GHz); 62W power—16 GOPS/Watt*
  - *Note: ASCI Red (1996): first machine to reach 1 TF*
    - *4,510 Intel Pentium Pro nodes (200 MHz)*
    - *500 KW for the machine + 500 KW for cooling of the room*

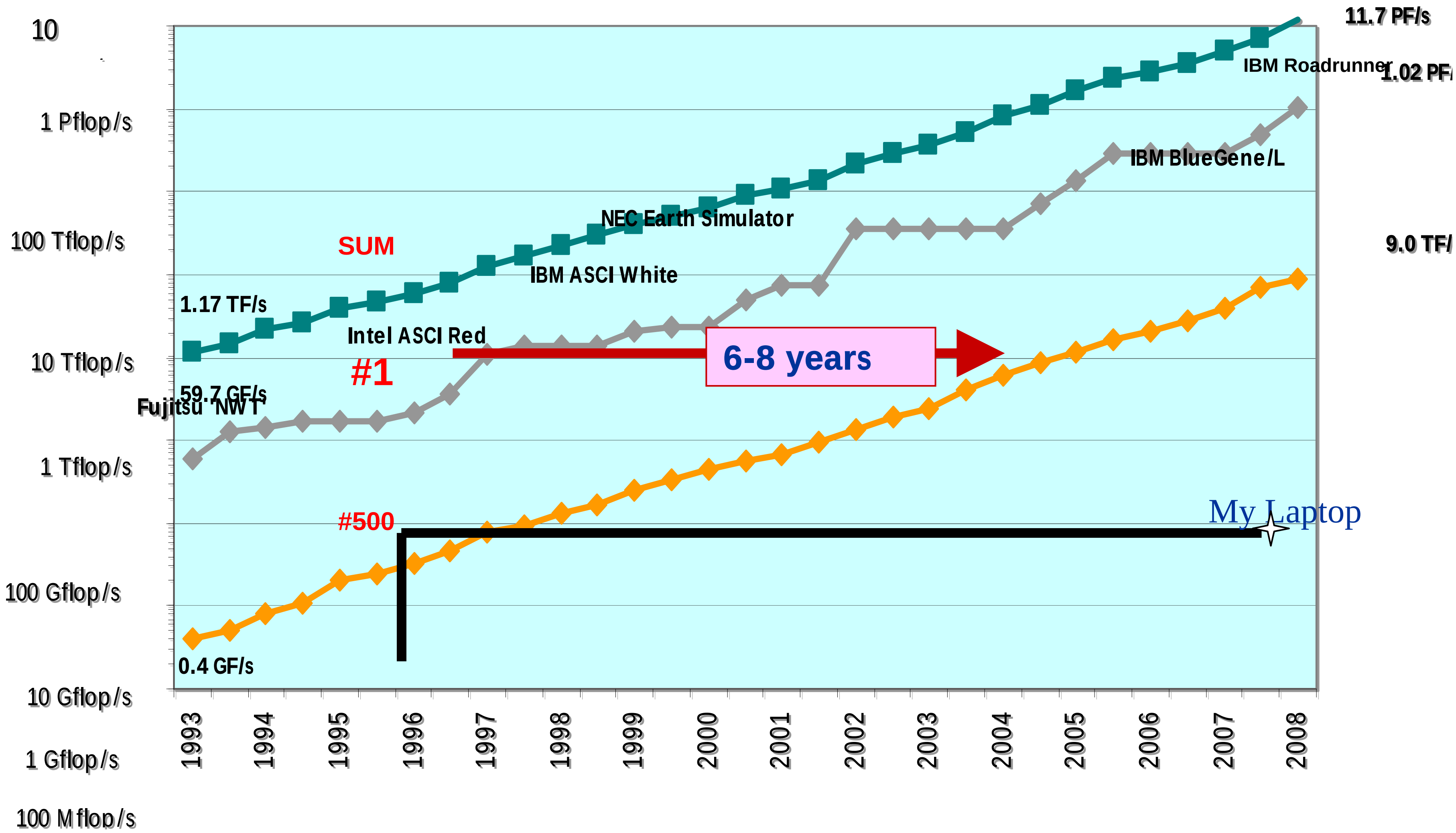




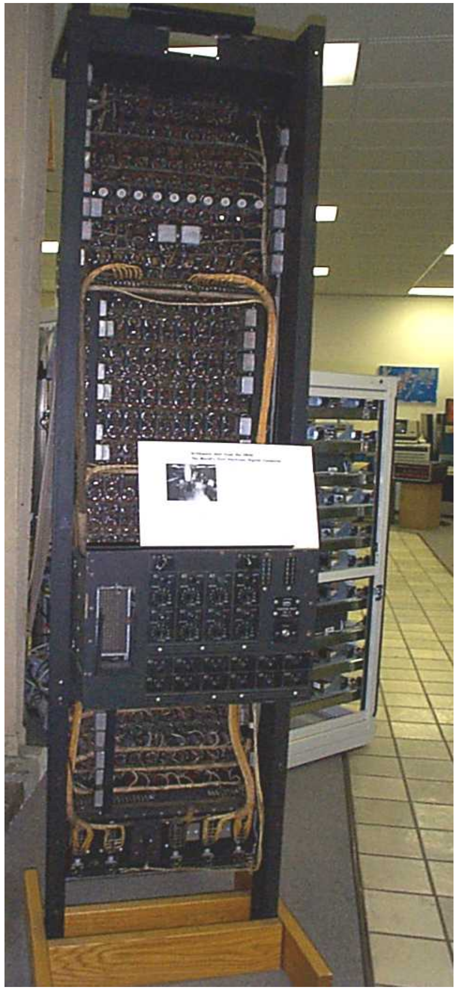
Multicore Systems are Not Just Small SMPs or MPPs

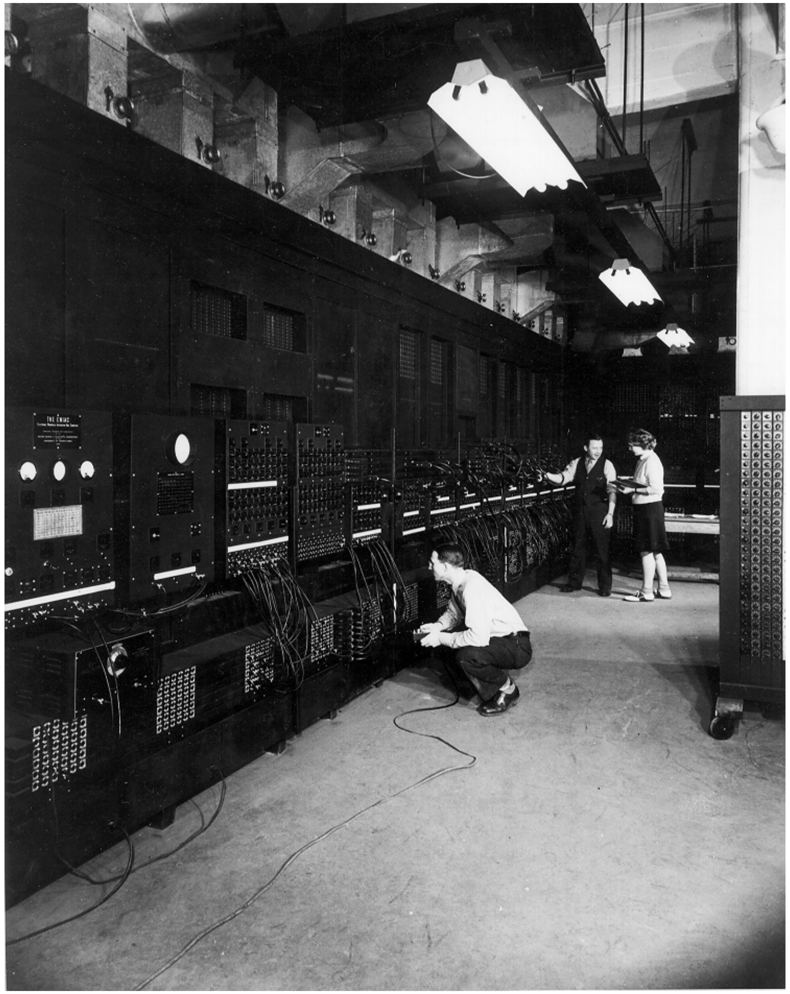
- Intra-chip inter-core bandwidth is much larger than for a typical parallel machine (SMP or MPP)
- Intra-chip inter-core latencies are much smaller
- Multicore systems can offer lightweight synchronization
- Lock-based synchronization is unacceptable: transactional memory and full/empty bits (Cray MTA) are alternatives
- Processing-In-Memory (PIM) technology offers additional methods for exploitation of locality

Top 500 Performance Development



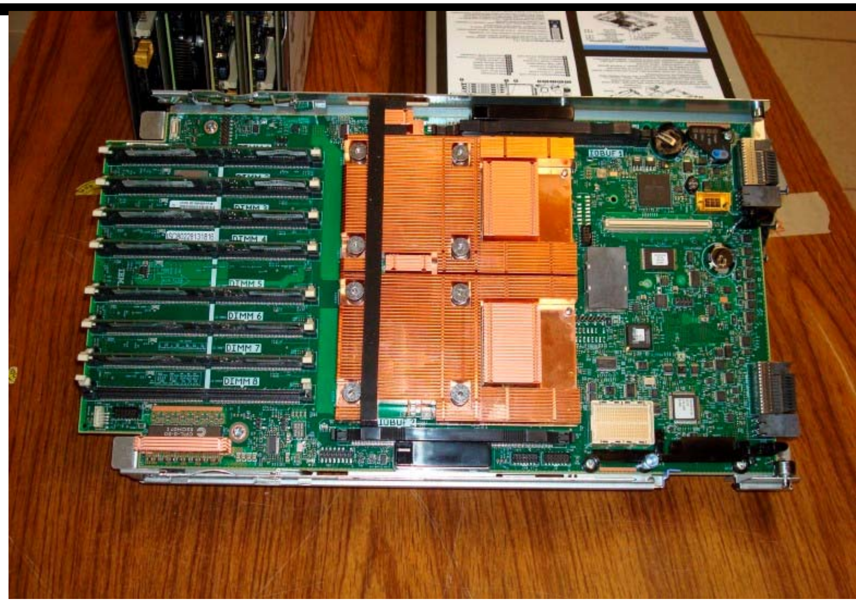
From Eniac (1946) …



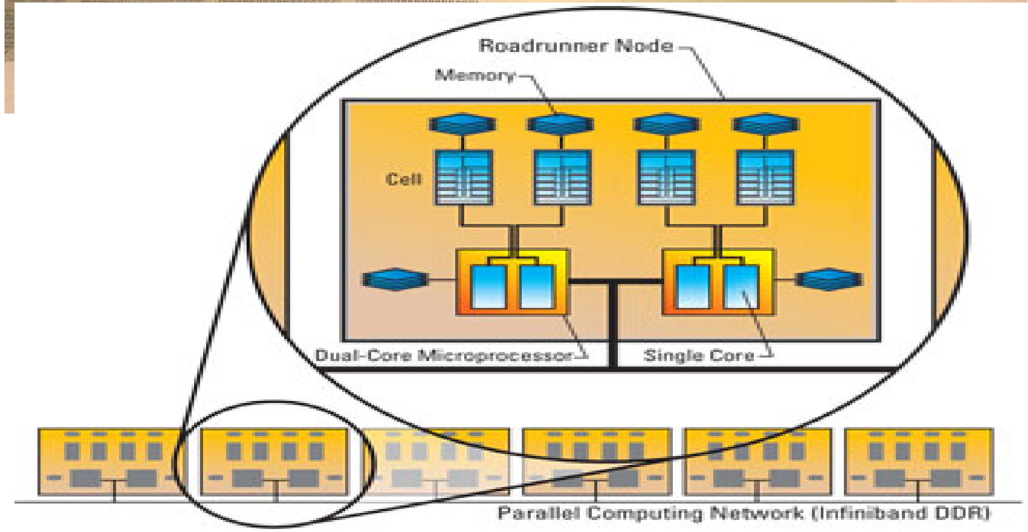


10^3 OPS

...to LANL Roadrunner: Top 500 #1



Cell Blade



1,026 TF=10^{15} OPS

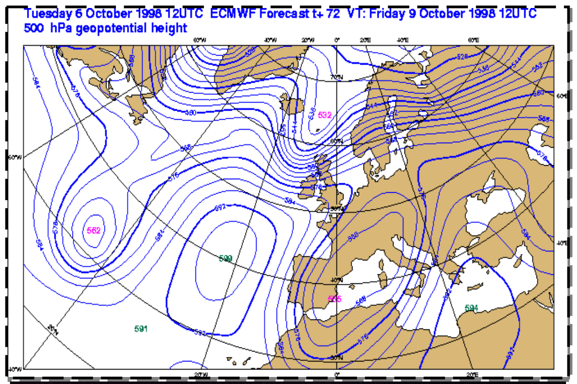
The first machine reaching Peta-scale performance

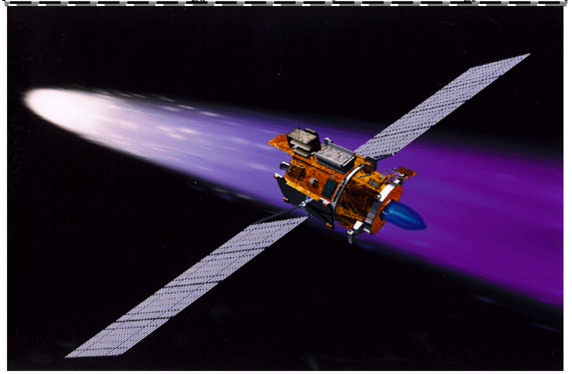
17 clusters, each with 192 nodes
Each node contains Opteron and 4 Cells
12,960 Cell chips (100 GF double precision)
Each Cell contains a PowerPC and 8 SPEs
6,948 dual-core Opterons
Total: 122,400 cores

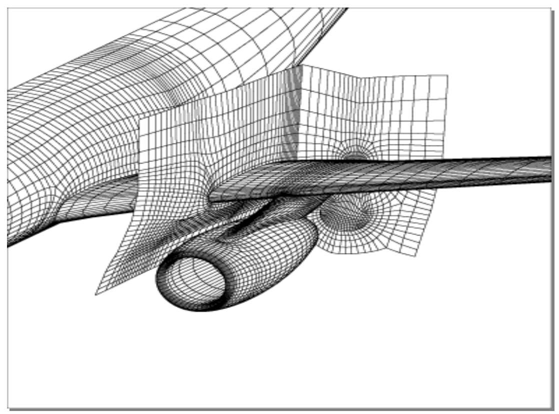
Applications

- HPC has become the third pillar of science and engineering, in addition to *theory* and *experiment*
- Traditional application areas include:
  - *DNA Analysis*
  - *Drug Design*
  - *Medicine*
  - *Aerospace*
  - *Manufacturing*
  - *Weather Forecasting and Climate Research*
- New architectures facilitate new applications:
  - *Graph Traversals*
  - *Dynamic Programming*
  - *…*
  - *Backtrack Branch & Bound*

UC Berkeley's "Dwarfs"







Contents

High-Level Sequential Languages

The designers of the very first high level programming language were aware that their success depended on the target code performance:

***John Backus (1957)*: "… It was our belief that if FORTRAN … were to translate any reasonable scientific source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger …"**

High-level algorithmic languages became generally accepted standards for sequential programming since their advantages outweighed any performance drawbacks

For programming of HPC systems no similar development took place

Programming Paradigm for MPPs and Clusters: MPI is State-of-the-Art

The MPI Message-Passing Model

- *widely adopted portable standard for full control of communication*
- *relatively simple execution model*
- *can achieve good performance on commodity clusters*

Drawbacks of the MPI Model

- *low-level paradigm: "the assembly language of parallel programming"*
- *lack of separation between algorithm and communication management*
- *complex, difficult-to-change communication structures*
- *scalability to peta-scale questionable*

Alternatives to MPI have been proposed

- *automatic vectorization and parallelization*
- *libraries for one-sided communication (SHMEM, ARMCI, GASNet)*
- *High Performance Fortran (HPF), PGAS languages, OpenMP, etc.*

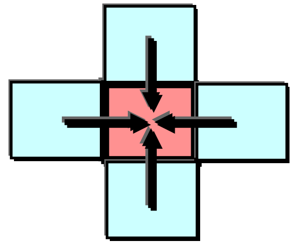
MPI vs HPF:
An Example for Locality Management (Jacobi Relaxation)

real, allocatable *A(:, :), B(:, :)*

…

Sequential Code

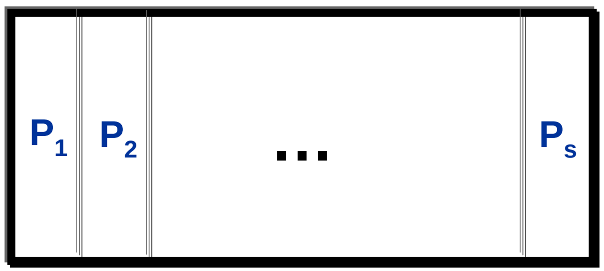
```
do while ( .not. converged )
   do J=1,N
        do I=1,N
             B(I,J)=0.25(A(I-1,J)+A(I+1,J)+A(I,J-1)+A(I,J+1))
         enddo
    enddo
A(1:N,1:N)=B
…
enddo
```



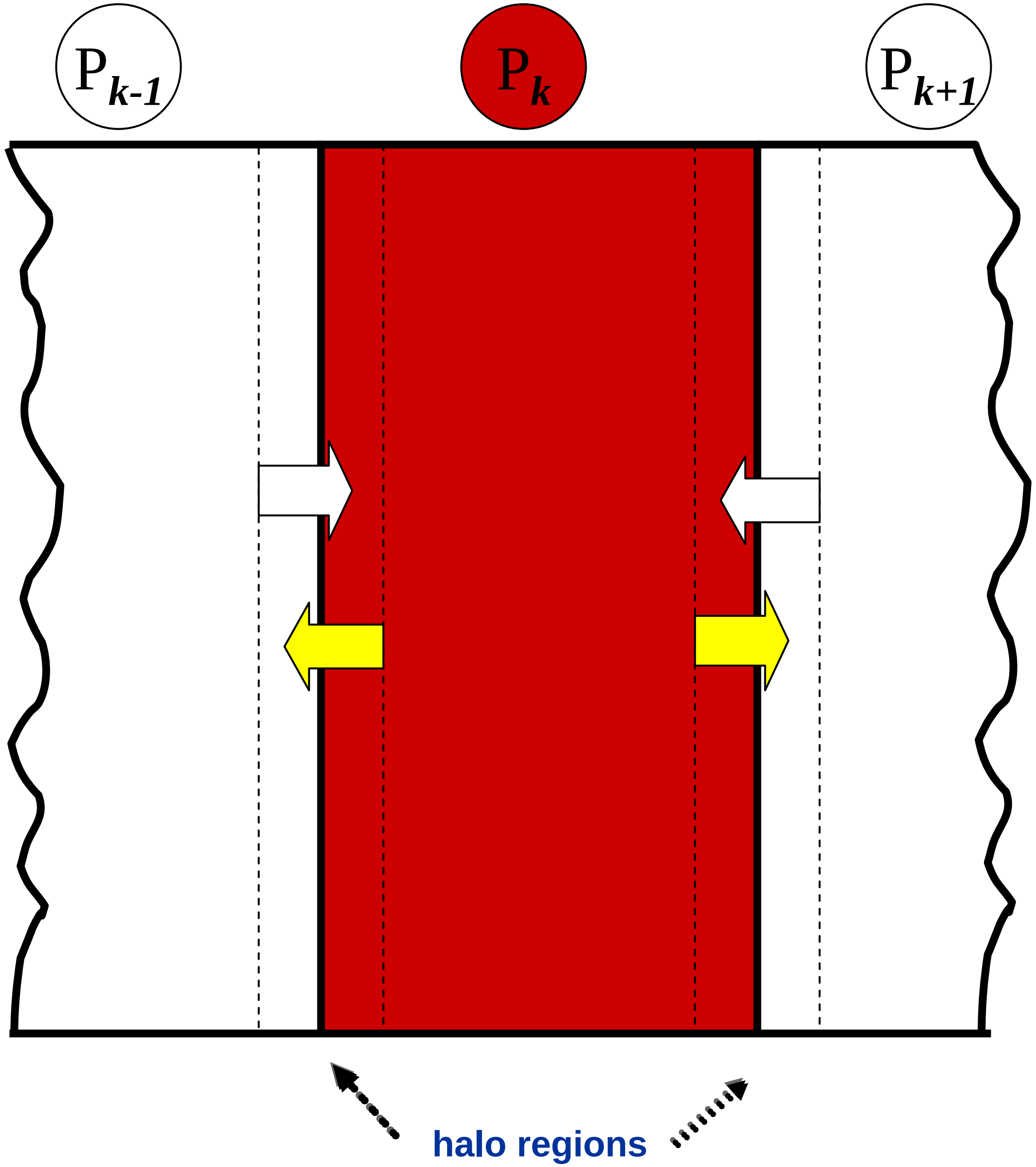
dependence pattern

Parallelization Based on Data Distribution

Let A and B be partitioned into blocks of columns mapped to different processors. All processors can work concurrently on their local data, but an exchange must take place at segment boundaries after each iteration…



Boundary Exchange in Overlap Regions




```
! purely local operation in each iteration:
 do while ( .not. converged )
    do J=1,M   ! Number of local columns
       do I=1,N
          B(I,J)=0.25(A(I-1,J)+A(I+1,J)+
                      A(I,J-1)+A(I,J+1))
       enddo
    enddo
    …
```

After iteration:
<u>Data Exchange</u>

Processor P_k **reads:**

- *rightmost column* of P_{k-1}
- *leftmost column* of $P_{k+1.}$

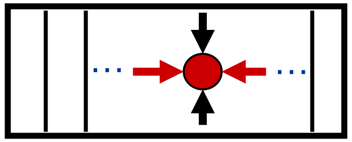
Processor P_k **copies:**

- *its leftmost column* to P_{k-1}
- *its rightmost column* to $P_{k+1.}$

The Key Idea of High Performance Fortran (HPF)

Message Passing Approach

local view of data, local control,
explicit two-sided communication



HPF Approach

global view of data, global control,
compiler-generated communication

initialize MPI

local computation

```
do while (.not. converged)
   do  J=1,M
      do  I=1,N
         B(I,J) = 0.25 * (A(I-1,J)+A(I+1,J)+
                          A(I,J-1)+A(I,J+1))
      end do
   end do
   A(1:N,1:N) = B(1:N,1:N)
```

communication

```
if (MOD(myrank,2) .eq. 1) then
   call MPI_SEND(B(1,1),N,…,myrank-1,..)
   call MPI_RCV(A(1,0),N,…,myrank-1,..)
   if (myrank .lt. s-1) then
      call MPI_SEND(B(1,M),N,…,myrank+1,..)
      call MPI_RCV(A(1,M+1),N,…,myrank+1,..)
   endif
                     else  …
          …
```

global computation

```
do while (.not. converged)
   do  J=1,N
      do  I=1,N
         B(I,J) = 0.25 * (A(I-1,J)+A(I+1,J)+
                          A(I,J-1)+A(I,J+1))
      end do
   end do
   A(1:N,1:N) = B(1:N,1:N)
```

data distribution

```
processors  P(NUMBER_OF_PROCESSORS)
distribute(*,BLOCK) onto P :: A, B
```

communication
compiler-generated

K. Kennedy, C. Koelbel, and H. Zima: The Rise and Fall of High Performance Fortran: An Historical Object Lesson

Proc. History of Programming Languages III (HOPL III), San Diego, June 2007

Example: Sweep Over Unstructured Mesh in HPF

```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
      TYPE NODE       ! type of a node in the unstructured grid
      …
      REAL::V1, V2    ! flow variables
      END TYPE NODE

      TYPE(NODE), ALLOCATABLE::GRID(:)
      REAL, ALLOCATABLE::EDGE(:,2)
      INTEGER, ALLOCATABLE::MAP(:)     ! mapping array
!HPF$ DYNAMIC, DISTRIBUTE(BLOCK)::GRID
!HPF$ DYNAMIC, DISTRIBUTE(BLOCK,*)::EDGE
!HPF$ DISTRIBUTE(BLOCK)::MAP
      …
! Read parameters; allocate GRID, MAP; initialize GRID, M
      CALL GRID_PARTITIONER(GRID,MAP)
!HPF$ REDISTRIBUTE GRID(INDIRECT(MAP))
      ALLOCATE(EDGE(M,2))
! Initialize and realign EDGE with GRID
! Sweep over edges of the grid:
!HPF$ INDEPENDENT,ON HOME(EDGE(J,1)),NEW(N1,N2,DELTAV),REDUCTION(V2)
     DO J=1,M
      N1=EDGE(J,1), N2=EDGE(J,2)
      …
      DELTAV=F(V1(N1),V1(N2))
      V2(N1)=V2(N1)-DELTAV
      V2(N2)=V2(N2)+DELTAV
     ENDDO
```

Fortran+MPI Communication for 3D 27-point Stencil (NAS MG rprj3)

```fortran
subroutine comm3(u,n1,n2,n3,kk)
use caf_intrinsics

implicit none

include 'cafnpb.h'
include 'globals.h'

integer n1, n2, n3, kk
double precision u(n1,n2,n3)
integer axis

if( .not. dead(kk) )then
   do  axis = 1, 3
      if( nprocs .ne. 1) then
         call sync_all()
         call give3( axis, +1, u, n1, n2, n3, kk )
         call give3( axis, -1, u, n1, n2, n3, kk )
         call sync_all()
         call take3( axis, -1, u, n1, n2, n3 )
         call take3( axis, +1, u, n1, n2, n3 )
      else
         call comm1p( axis, u, n1, n2, n3, kk )
      endif
   enddo
else
   do  axis = 1, 3
      call sync_all()
      call sync_all()
   enddo
   call zero3(u,n1,n2,n3)
endif
return
end


subroutine give3( axis, dir, u, n1, n2, n3, k )
use caf_intrinsics


implicit none

include 'cafnpb.h'
include 'globals.h'

integer axis, dir, n1, n2, n3, k, ierr
double precision u( n1, n2, n3 )

integer i3, i2, i1, buff_len,buff_id

buff_id = 2 + dir
buff_len = 0

if( axis .eq.  1 )then
   if( dir .eq. -1 )then

      do  i3=2,n3-1
         do  i2=2,n2-1
            buff_len = buff_len + 1
            buff(buff_len,buff_id ) = u( 2,  i2,i3)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i2=2,n2-1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( n1-1, i2,i3)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   endif
endif

if( axis .eq.  2 )then
   if( dir .eq. -1 )then

      do  i3=2,n3-1
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( i1,  2,i3)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len,  buff_id )= u( i1,n2-1,i3)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   endif
endif

if( axis .eq.  3 )then
   if( dir .eq. -1 )then

      do  i2=1,n2
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( i1,i2,2)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   else if( dir .eq. +1 ) then

      do  i2=1,n2
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( i1,i2,n3-1)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   endif
endif

return
end


subroutine take3( axis, dir, u, n1, n2, n3 )
use caf_intrinsics


implicit none

include 'cafnpb.h'
include 'globals.h'

integer axis, dir, n1, n2, n3
double precision u( n1, n2, n3 )

integer buff_id, indx

integer i3, i2, i1

buff_id = 3 + dir
indx = 0

if( axis .eq.  1 )then
   if( dir .eq. -1 )then

      do  i3=2,n3-1
         do  i2=2,n2-1
            indx = indx + 1
            u(n1,i2,i3) = buff(indx, buff_id )
         enddo
      enddo

   else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i2=2,n2-1
            indx = indx + 1
            u(1,i2,i3) = buff(indx, buff_id )
         enddo
      enddo

   endif
endif

if( axis .eq.  2 )then
   if( dir .eq. -1 )then

      do  i3=2,n3-1
         do  i1=1,n1
            indx = indx + 1
            u(i1,n2,i3) = buff(indx, buff_id )
         enddo
      enddo

   else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i1=1,n1
            indx = indx + 1
            u(i1,1,i3) = buff(indx, buff_id )
         enddo
      enddo

   endif
endif

if( axis .eq.  3 )then
   if( dir .eq. -1 )then

      do  i2=1,n2
         do  i1=1,n1
            indx = indx + 1
            u(i1,i2,n3) = buff(indx, buff_id )
         enddo
      enddo

   else if( dir .eq. +1 ) then

      do  i2=1,n2
         do  i1=1,n1
            indx = indx + 1
            u(i1,i2,1) = buff(indx, buff_id )
         enddo
      enddo

   endif
endif

return
end


subroutine comm1p( axis, u, n1, n2, n3, kk )
use caf_intrinsics


implicit none

include 'cafnpb.h'
include 'globals.h'

integer axis, dir, n1, n2, n3
double precision u( n1, n2, n3 )

integer i3, i2, i1, buff_len,buff_id
integer i, kk, indx

dir = -1

buff_id = 3 + dir
buff_len = nm2

do  i=1,nm2
   buff(i,buff_id) = 0.0D0
enddo


dir = +1

buff_id = 3 + dir
buff_len = nm2

do  i=1,nm2
   buff(i,buff_id) = 0.0D0
enddo

dir = +1

buff_id = 2 + dir
buff_len = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( n1-1, i2,i3)
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len,  buff_id )= u( i1,n2-1,i3)
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( i1,i2,n3-1)
      enddo
   enddo
endif

dir = -1

buff_id = 2 + dir
buff_len = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         buff_len = buff_len + 1
         buff(buff_len,buff_id ) = u( 2,  i2,i3)
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( i1,  2,i3)
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( i1,i2,2)
      enddo
   enddo
endif

do  i=1,nm2
   buff(i,4) = buff(i,3)
   buff(i,2) = buff(i,1)
enddo

dir = -1

buff_id = 3 + dir
indx = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         indx = indx + 1
         u(n1,i2,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         indx = indx + 1
         u(i1,n2,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         indx = indx + 1
         u(i1,i2,n3) = buff(indx, buff_id )
      enddo
   enddo
endif


dir = +1

buff_id = 3 + dir
indx = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         indx = indx + 1
         u(1,i2,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         indx = indx + 1
         u(i1,1,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         indx = indx + 1
         u(i1,i2,1) = buff(indx, buff_id )
      enddo
   enddo
endif

return
end
```

Chapel 3D NAS MG Stencil rprj3

```
function rprj3(S,R) {
  const Stencil: domain(3) = [-1..1, -1..1, -1..1],         // 27-points
        w: [0..3]real = (/0.5, 0.25, 0.125, 0.0625/),       // weights
        w3d: [(i,j,k) in Stencil] = w((i!=0) + (j!=0) + (k!=0));

forall ijk in S.domain do
    S(ijk) = sum reduce [off in Stencil] (w3d(off) * R(ijk + R.stride*off));
}
```

Productivity Challenges for Peta-Scale Systems

- **Large-scale hierarchical architectural parallelism**
  - *tens of thousands to hundreds of thousands of processors*
  - *component failures may occur frequently*
- **Extreme non-uniformity in data access**
- **Applications: large, complex, and long-lived**
  - *multi-disciplinary, multi-language, multi-paradigm*
  - *dynamic, irregular, and adaptive*
  - *survive many hardware generations ➔ portability is important*
- **How to exploit the parallelism and locality provided by the architecture?**
  - *automatic parallelization and locality management are not powerful enough to provide a general efficient solution*
  - *explicit support for control of parallelism and locality must be provided by the programming model and the language*

Parallel Programming Models

- **Fragmented Models**
  - *processor-centric view: code written from the viewpoint of single threads*
  - *local view of data segments*
- **Single Program Multiple Data (SPMD) Model**
  - *special class of fragmented model*
  - *single program executed in multiple instances*
- **Global-view Models**
  - *global view of data and computation*
    - *burden of partitioning shifts to compiler/runtime*
    - *user may guide this process via language constructs*
- **Locality-aware Models**
  - *features for mapping data and/or control to the architecture*

Languages for High Performance Computing

- **HPF Language Family**
  - *predecessors: CM-Fortran, Fortran D, Vienna Fortran*
  - *High Performance Fortran (HPF): HPF-1 (1993); HPF-2(1997)*
  - *successors: HPF+, HPF/JA*
- **OpenMP**
- **Partitioned Global Address Space (PGAS) Languages**
  - *Co-Array Fortran*
  - *UPC*
  - *Titanium*
- **High-Productivity Languages developed in the HPCS Program**
  - *Chapel*
  - *X10*
  - *Fortress*
- **Domain-Specific Languages and Abstractions**

Contents

HPCS Languages

<u>global</u> view of data, <u>global</u> control

- ***High-Productivity Computing Systems (HPCS)*** **is a DARPA-sponsored program for the development of peta-scale architectures (2002-2010)**
- **HPCS Languages**
  - ***Chapel (Cascade Project, led by Cray Inc.)***
  - ***X10 (PERCS Project, led by IBM)***
  - ***Fortress (HERO Project [until 2006], led by Sun Microsystems)***
- **These are new, memory-managed, object-oriented languages**
  - ***global view of data and computation → generally no distinction between local and remote data access in the source code***
  - ***support for explicit data and task parallelism***
  - ***explicit locality management***
  - **Chapel** ***is unique in that it provides user-defined data distributions***

Chapel Language Concepts

http://chapel.cs.washington.edu

- **Explicit high-level control of parallelism**
  - *data parallelism*
    - domains, arrays, indices: *support distributed data aggregates*
    - forall loops and iterators: *express data parallel computations*
  - *task parallelism*
    - cobegin statements: *specify task parallel computations*
    - synchronization variables, atomic sections
- **Explicit high-level control of locality**
  - *"locales": abstract units of locality*
  - *data distributions: map data domains to sets of locales*
  - *on clauses: map execution components to sets of locales*
- **Close relationship to mainstream languages**
  - *object-oriented*
  - *type inference and generic programming*
  - *modules for Programming-in-the-Large*

Note: Some of the features discussed in the following have the status of research proposals and are currently not part of the official Chapel language specification

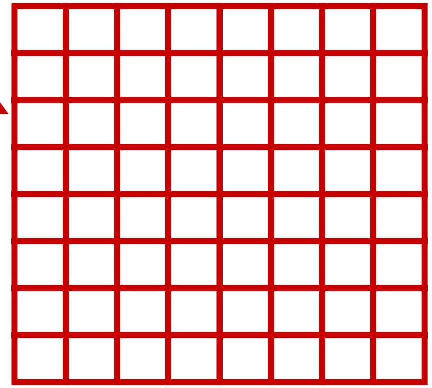
Example: Jacobi Relaxation in Chapel

```
const L:[1..p,1..q] locale = reshape(Locales);

const n= …, epsilon= …;
const DD:domain(2)=[0..n+1,0..n+1] distributed(block,block)on L;
      D: subdomain(DD) = [1..n, 1..n];
var delta: real;
var A, Temp: [DD] real; /*array declarations over domain DD */

A(0,1..n) = 1.0;

do {
    forall (i,j) in D {  /* parallel iteration over domain D */
       Temp(i,j) = (A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))/4.0;
       delta = max reduce abs(A(D) – Temp(D));
       A(D) = Temp(D);
    } while (delta > epsilon);

writeln(A);
```

Example: Jacobi Relaxation in Chapel


```
const L:[1..p,1..q] locale = reshape(Locales);

const n= …, epsilon= …;
const DD:domain(2)…distributed(block,block on L);
      D: subdomain(DD) = [1..n, 1..n];
var delta: real;
var A, Temp: [DD] real;

A(0,1..n) = 1.0;

do {
    forall (i,j) in D {
       Temp(i,j) = (A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))/4.0;
       delta = max reduce abs(A(D) – Temp(D));
       A(D) = Temp(D);
    } while (delta > epsilon);

writeln(A);
```

Locale Grid L



Key Features

- global view of data/control
- explicit parallelism (forall)
- high-level locality control
- NO explicit communication
- NO local/remote distinction in source code

Task Parallelism in Chapel

- **Task Creation**

  cobegin *{ S_1,…S_n}* *executes the S_i in parallel (i = 1,…n)*

- **Task Localization**

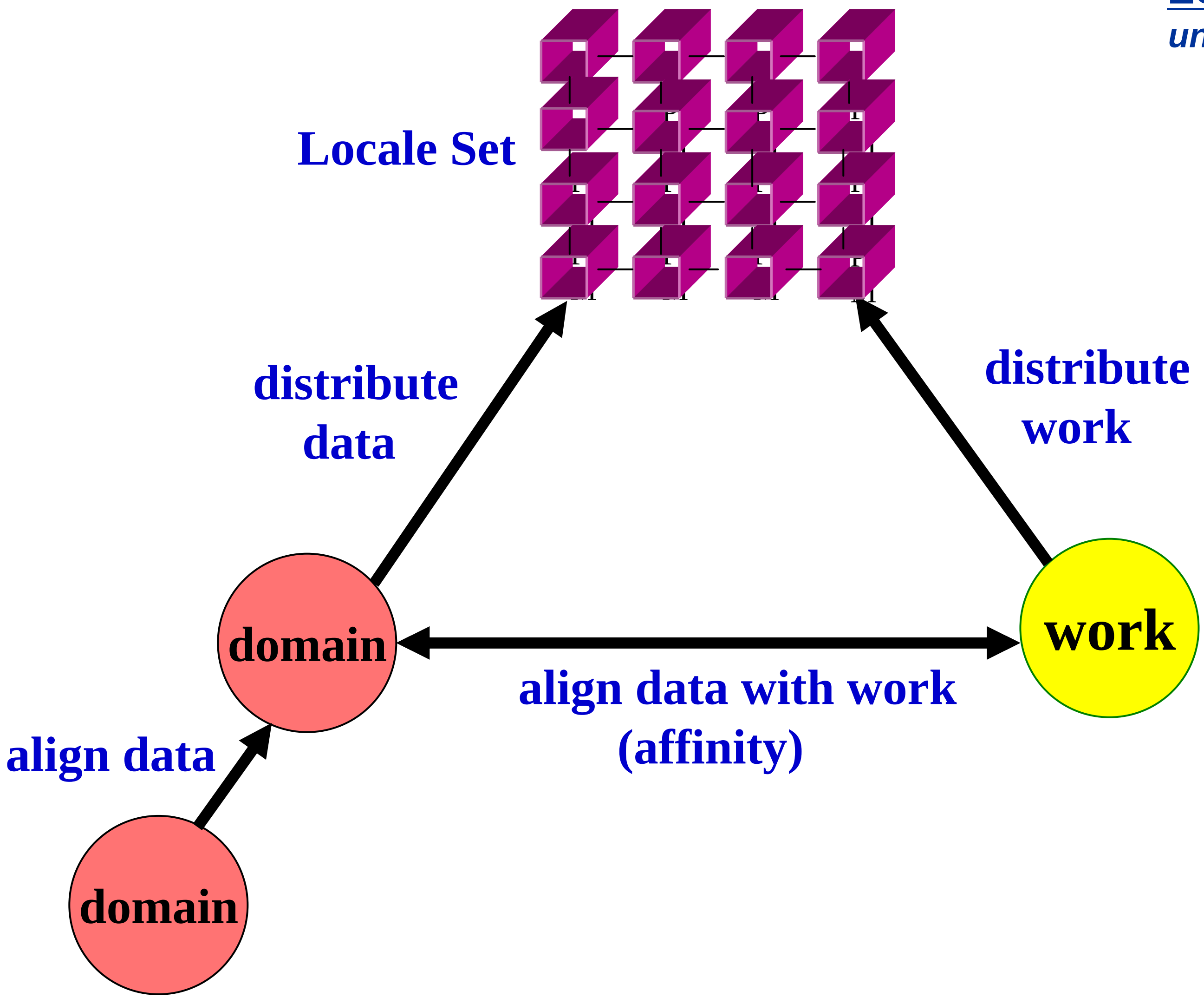
  on *L(i,j)* **do** *f(A(i,j))* *executes f(A(i,j) on locale L(i,j)*

- **Task Synchronization**

  - *atomic sections*
  - *sync variables*
  - *single-assignment variables*

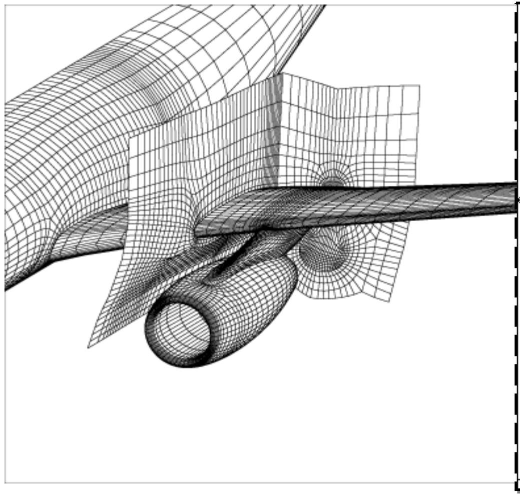
Aspects of Locality

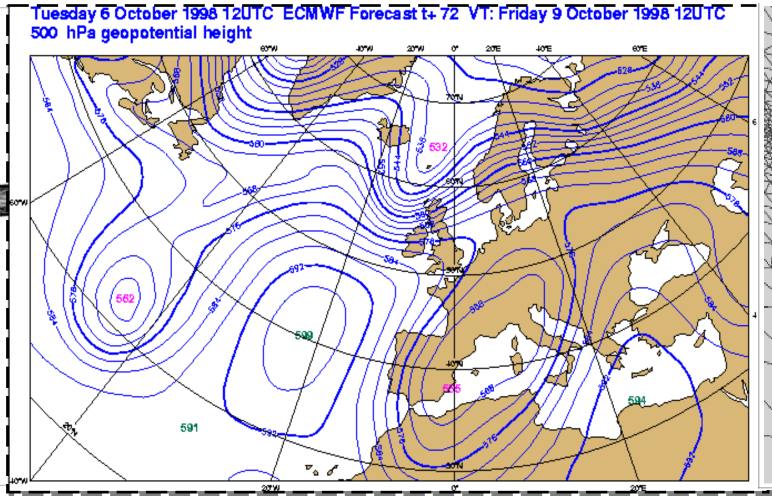
Locale: *an abstract unit of locality*

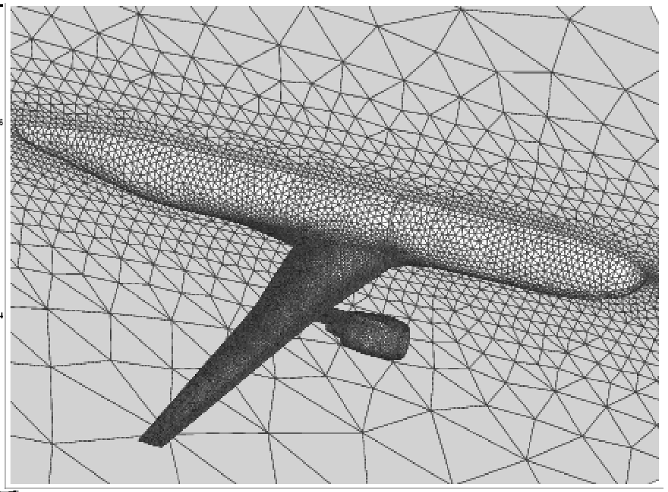


Chapel's Framework for User-Defined Distributions

- **Provides functionality for:**
  - ***distributing index sets across locales***
  - ***arranging data within a locale***
  - ***defining specialized distribution libraries***
- **This capability is in its effect similar to *function specification***
  - ***unstructured meshes***
  - ***multi-block problems***
  - ***multi-grid problems***
  - ***distributed sparse matrices***







Locality Control in Chapel: Basic Concepts

- **Domain: first class entity**
  - *components: index set, distribution, associated arrays, iterators*
- **Array—Mapping from a Domain to a Set of Variables**
- **Framework for User-Defined Distributions: three levels**
  1. *naïve use of a predefined library distribution (block, cyclic, indirect,…)*
  2. *specification of a distribution by*

     global mapping: index set → locales
     - *interface for the definition of mapping, distribution segments, iterators*
     - *system-provided default functionality can be overridden by user*
  3. *specification of a distribution by global mapping and*

     layout mapping: index set → locale data space
- **High-Level Control of *Communication***
  - *user-defined specification of halos; communication assertions*

User-Defined Distributions: Global Mapping

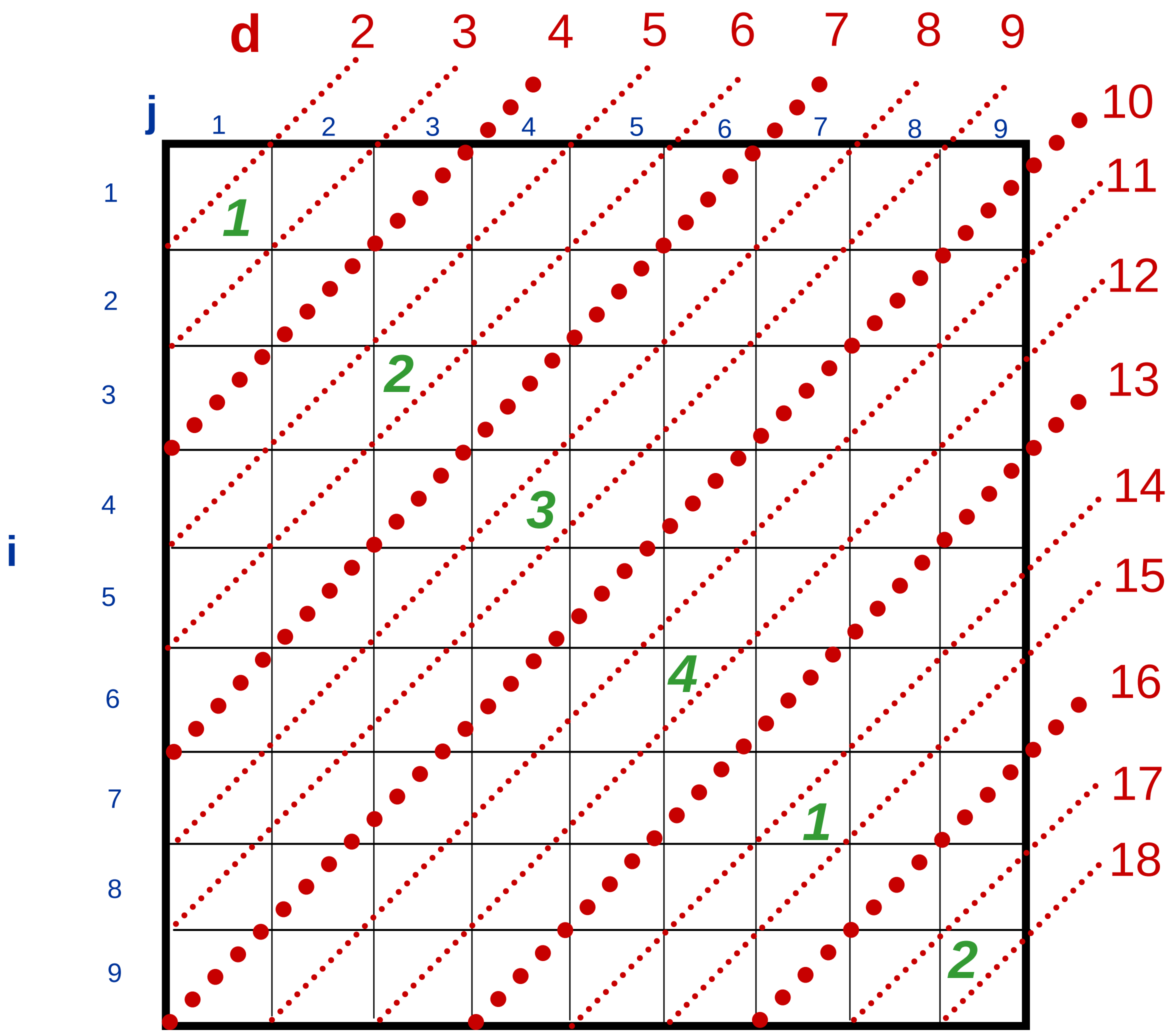
/* declaration of distribution classes MyC and MyB: */

```
class MyC: Distribution {
  const z:int;                                   /* block size */
  const ntl:int;                                 /* number of target locales*/

  function map(i:index(source)):locale {         /* global mapping for MyC */
    return Locales(mod(ceil(i/z-1)+1,ntl));
  }

class MyB: Distribution {
  var bl:int = ...;                              /* block length */

  function map(i: index(source)):locale {        /* global mapping for MyB */
    return Locales(ceil(i/bl));
  }
}
```

/* use of distribution classes MyC and MyB in declarations: */

```
const D1C: domain(1) distributed(MyC(z=100))=1..n1;
const D1B: domain(1) distributed(MyB) on Locales(1..num_locales/10)=1..n1;
var   A1: [D1C] real;
var   A2: [D1B] real;
```

Example: Banded Distribution



Diagonal A/d = { A(i,j) | d=i+j }

bw = 3 (bandwidth)

p=4 (number of locales)

Distribution—global map:

Blocks of bw diagonals are cyclically mapped to locales

Layout:

Each diagonal is represented as a one-dimensional dense array. Arrays in a locale are referenced by a pointer array

Example
Matrix-Vector Multiplication (sparse CRS)

0	**53**	0	0	0	0	0	0
0	0	0	0	0	0	**21**	0
19	0	0	0	0	0	0	**16**
0	0	0	0	0	**72**	0	0
0	0	0	**17**	0	0	0	0
0	0	0	0	**93**	0	0	0
0	0	0	0	0	0	**13**	0

0	0	0	0	**44**	0	0	**19**
0	**23**	**69**	0	**37**	0	0	0
27	0	0	**11**	0	0	**64**	0

D^0	C^0	R^0
53	2	1
19	1	2
17	4	2
93	5	3
		3
		4
		5
		5

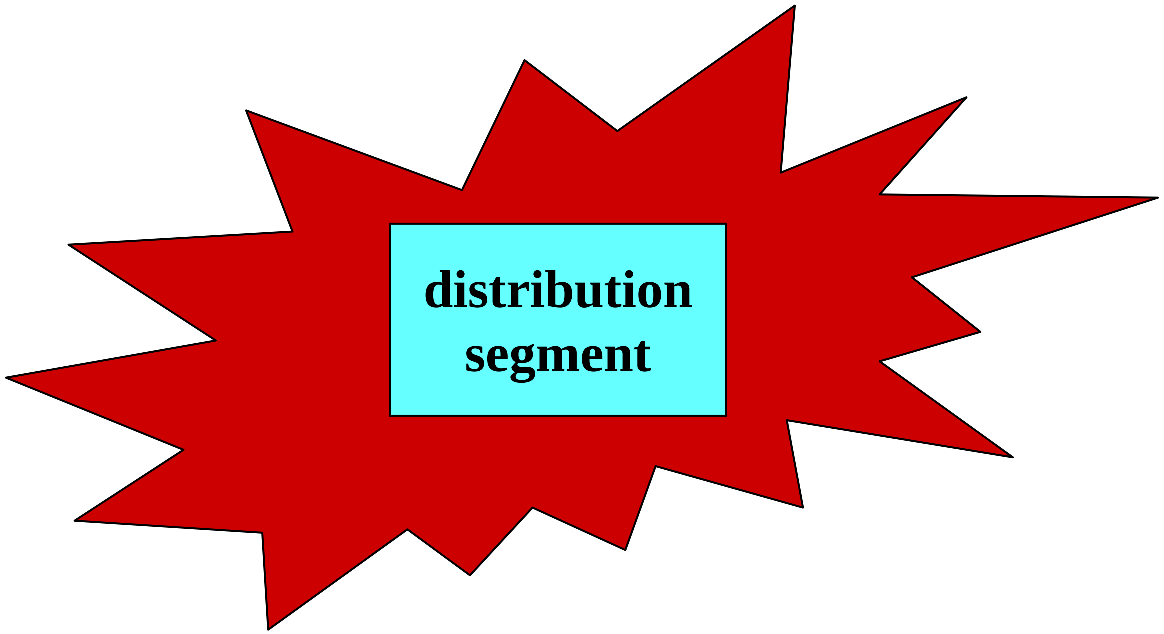
D^1	C^1	R^1
21	7	1
16	8	1
72	6	2
13	7	3
		4
		4
		4
		5

D^2	C^2	R^2
23	2	1
69	3	1
27	1	3
11	4	5

D^3	C^3	R^3
44	5	1
19	8	3
37	5	4
64	7	5

User-Defined Halos

- User-Defined Specification of halo (ghost cells)
- Compiler/Runtime System
  - allocates local images of remote data
  - defines mapping between remote objects and their images
- Halo Management
  - update
  - flush



Contents

PGAS Language Overview

Support for <u>global</u> view of data, but <u>local</u> control

- **Partitioned Global Address Space (PGAS) languages are based on the SPMD model**
- **Providing a shared-memory, *global view,* of data, combined with support for locality**
  - *global address space is logically partitioned, with each portion mapped to a processor*
  - *single-sided shared-memory communication (instead of MPI-style message passing)*
  - *in general, local and remote references distinguished in the source code*
  - *implemented via one-sided communication libraries (e.g., GASNet)*
- ***Local control* of execution via processor-centric view**
- **Main representatives: *Co-Array Fortran (CAF), Unified Parallel C (UPC), Titanium***

Domain-Specific Languages

- *General-purpose languages* are limited in their ability to accommodate the abstractions of a scientific domain
- *Domain-specific languages* provide abstractions tailored to a specific domain
  - *narrowing of the* semantic gap *between the programming language and the application domain*
  - separation of *domain expertise from parallelization and resource management*
- *Domain-specific knowledge* can be used to improve program analysis and support V&V and fault tolerance.
- *Telescoping* supports the automatic generation of domain-specific languages by generating specialized, optimized versions of libraries

Contents

Issues in Programming Environments

- **Legacy Code Migration**

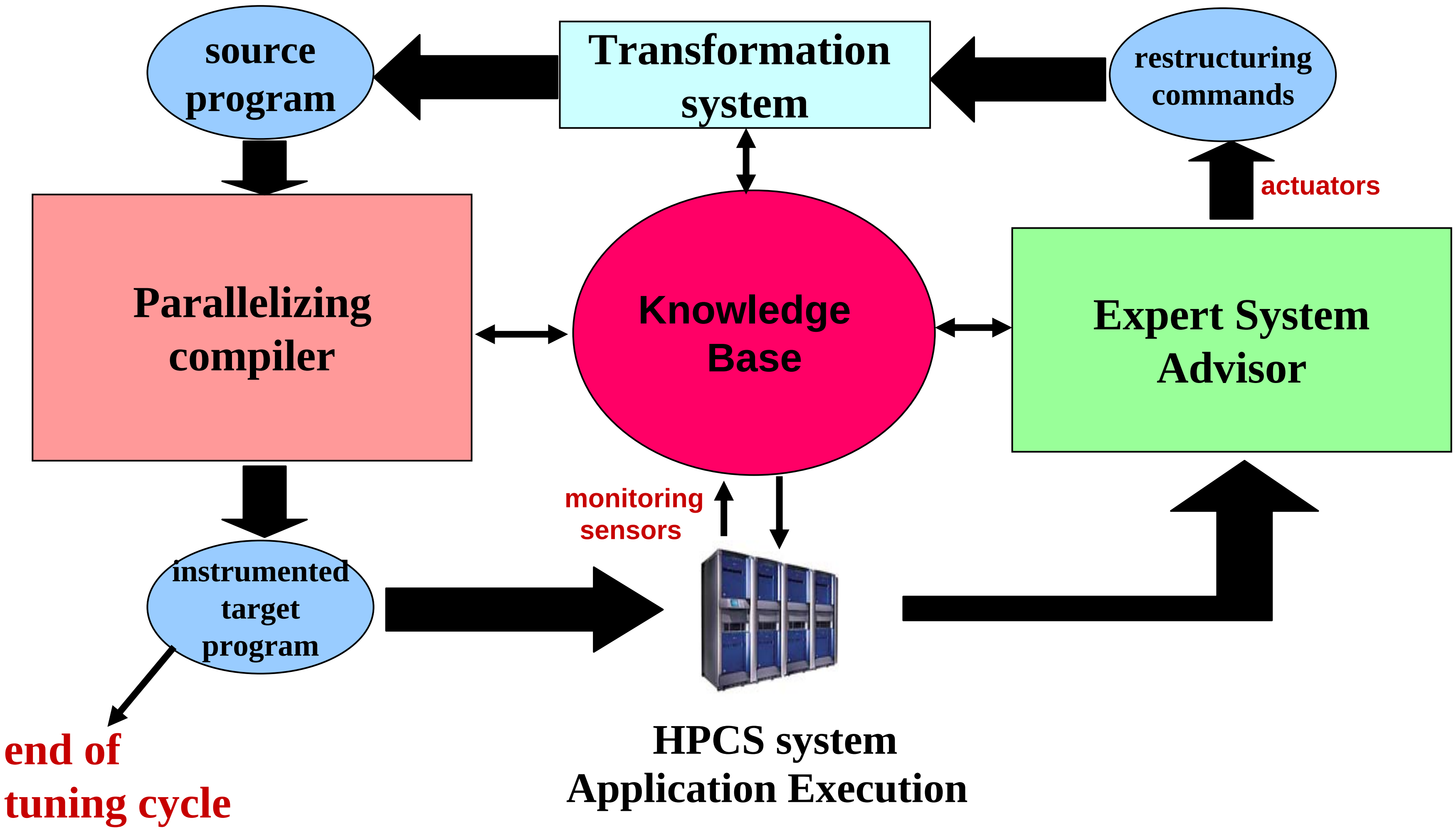
- **(Semi) Automatic Tuning**
  - *closed loop adaptive control: measurement, decision-making, actuation*
  - *information exposure: users, compilers, runtime systems*
  - *learning from experience: databases, data mining, reasoning systems*

- **Fault Tolerance**
  - *massive parallelism poses new reliability problems*
  - *fault anticipation, detection, localization, analysis, and recovery*

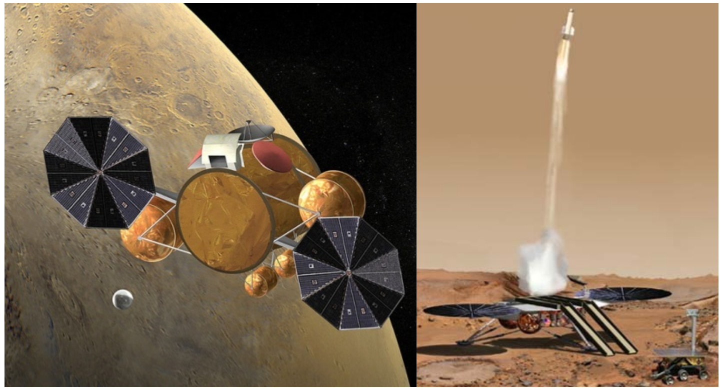
Legacy Code Migration

- **Rewriting Legacy Codes**
  - *preservation of intellectual content*
  - *performance portability: exploit new hardware and new algorithms*
  - *code size may preclude complete rewrite: incremental porting*
- **Language, compiler, tool, and runtime support**
  - *(semi) automatic tools for migrating code*
  - *translation of performance-critical sections requires highly-sophisticated software for automatic adaptation*
    - *reverse engineering of the original program*
    - *static analysis, using domain and/or architecture-specific knowledge*
    - *pattern matching and concept comprehension*
    - *optimizing code generation guided by the target architecture*

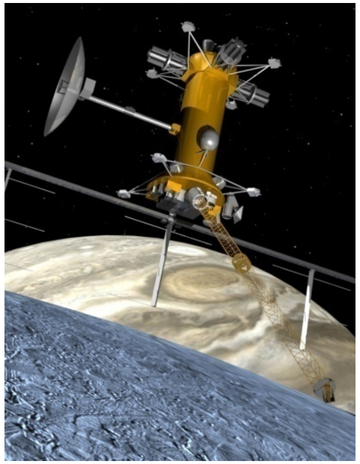
Offline Performance Tuning



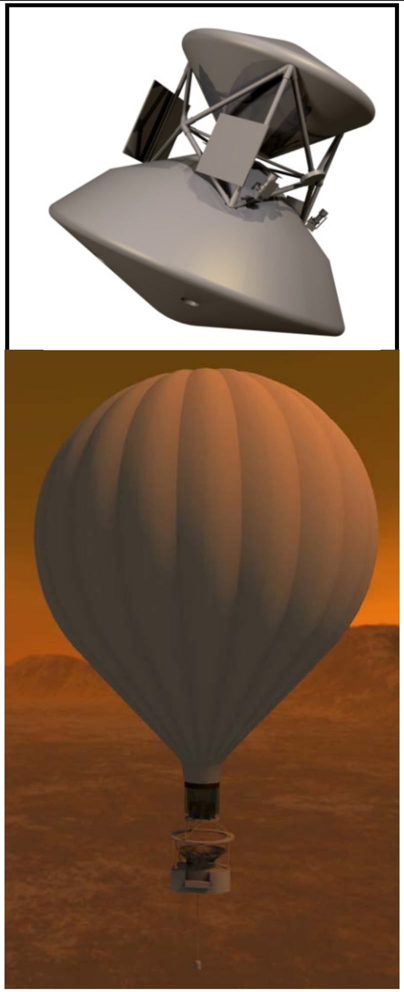
Autonomy and Fault Tolerance for High-Performance Space-Borne Computing



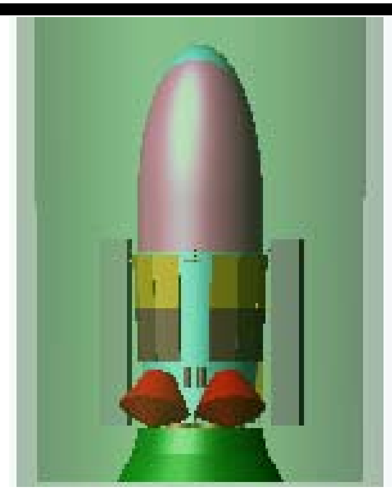
Mars Sample Return



Europa Explorer



Titan Explorer

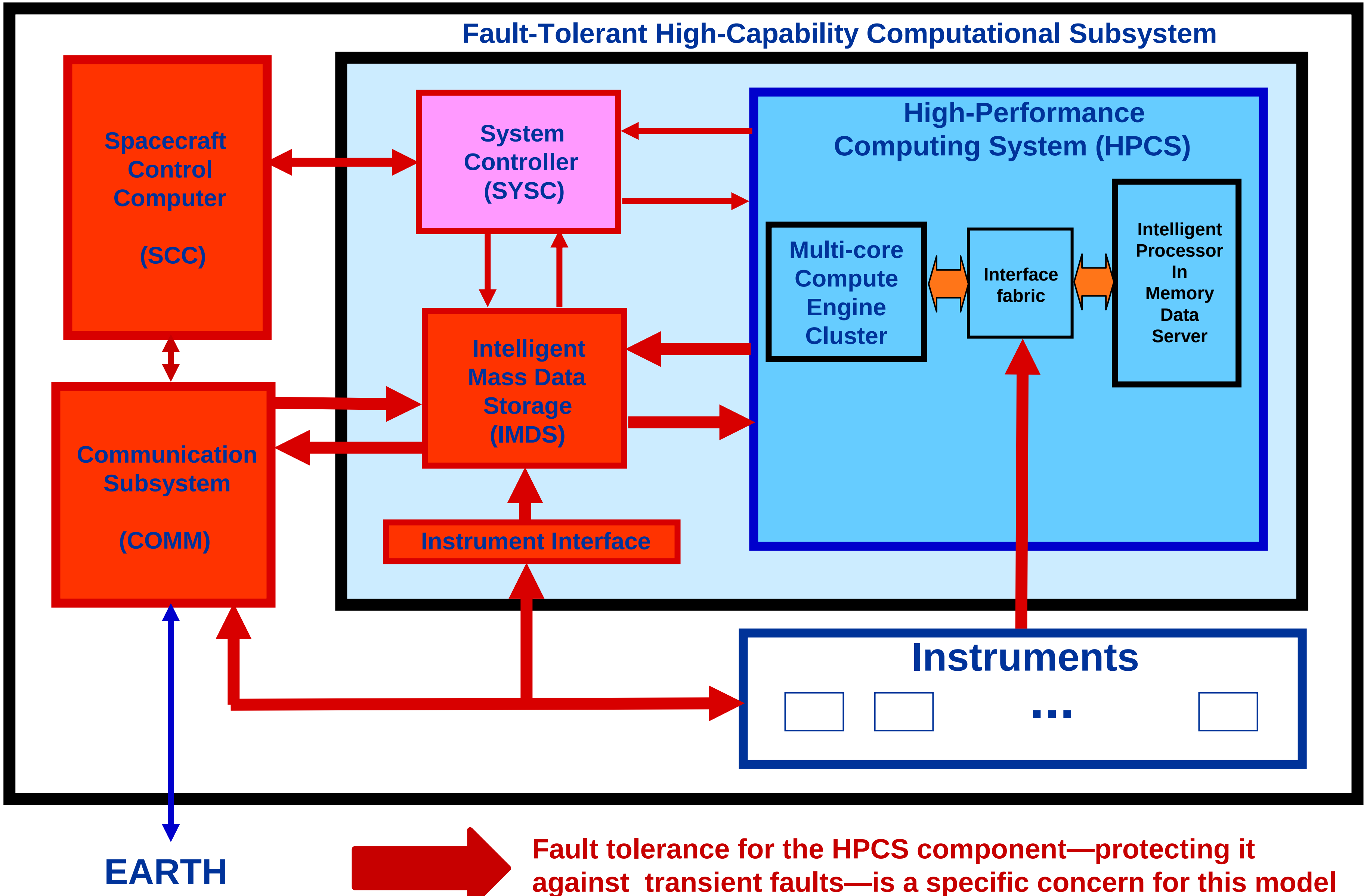


Neptune Triton Explorer

Europa Astrobiology Laboratory

Future missions need <u>Autonomy</u> *and* <u>High-Capability On-Board Computing</u>: this can be accomplished by extending traditional spacecraft architectures

High-Capability On-Board System: An Example

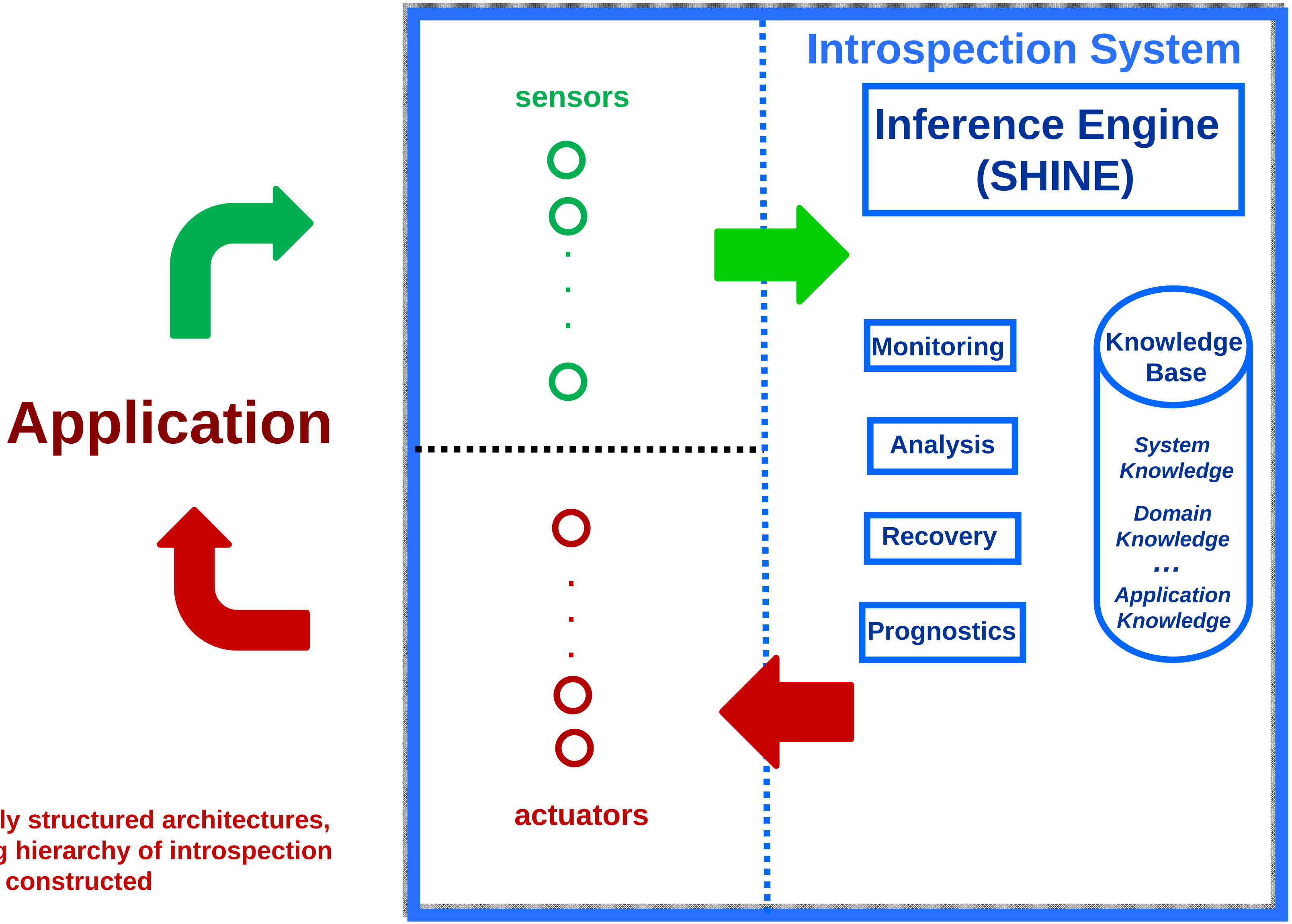


Fault tolerance for the HPCS component—protecting it against transient faults—is a specific concern for this model

Introspection…

- provides *dynamic* monitoring, analysis, and feedback, enabling system to become self-aware and context-aware:
  - *monitoring execution behavior*
  - *reasoning about its internal state*
  - *changing the system or system state when necessary*
- exploits adaptively the available threads
- can be applied to different scenarios, including:
  - *fault tolerance*
  - *performance tuning*
  - *power management*
  - *behavior analysis*
  - *intrusion detection*

An Introspection Module (IM)



For hierarchically structured architectures, a corresponding hierarchy of introspection modules will be constructed

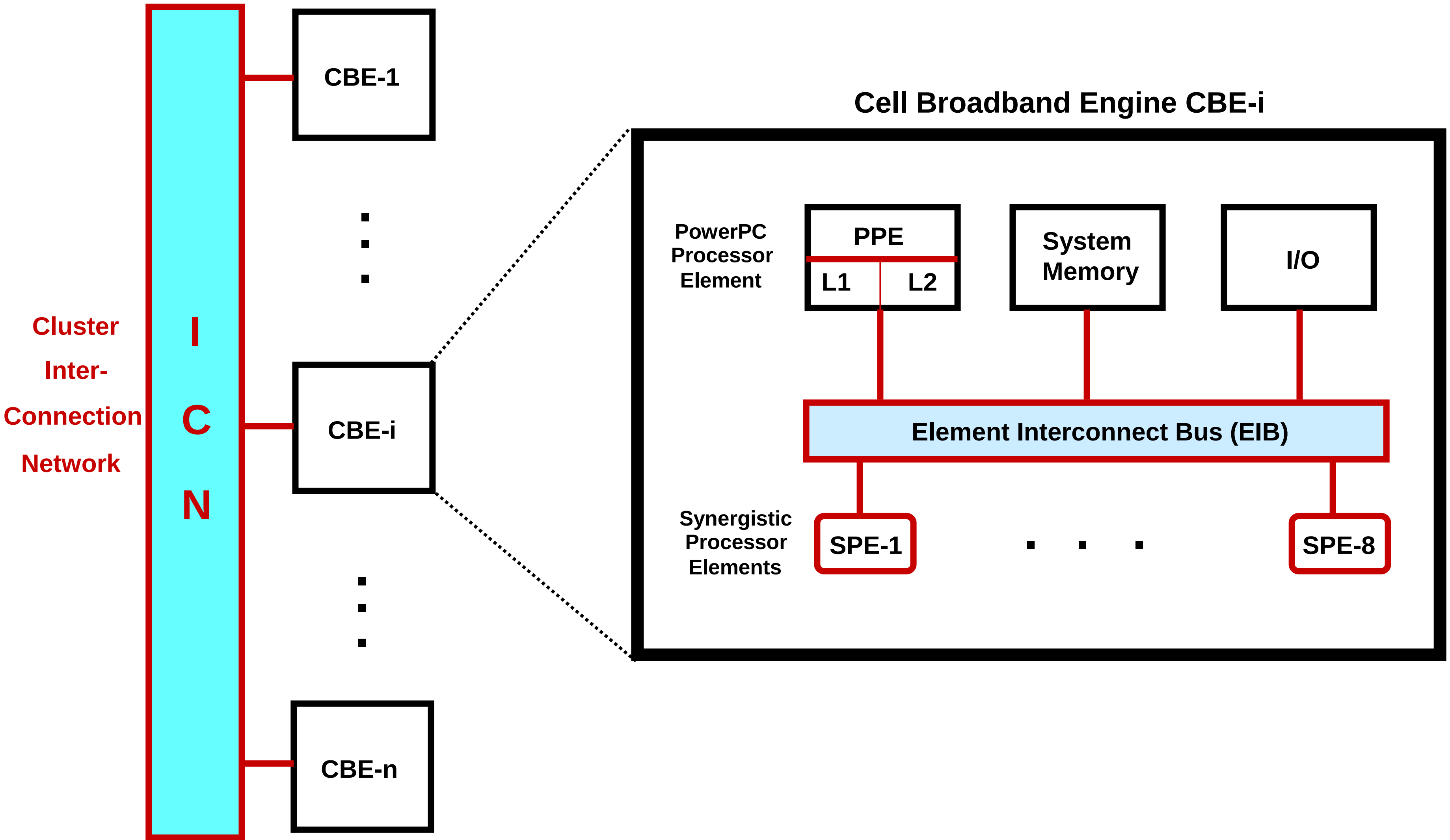
Conclusion

- **HPCS languages constitute an important step towards high-productivity programming for massively parallel peta-scale architectures**
- **Acceptance of a new language depends on many criteria, including:**
  - ***functionality and target code performance***
  - ***mature, industrial-strength compiler and runtime system technology***
  - ***easy integration/migration of legacy codes***
  - ***familiarity of users with conventional features***
  - ***flexibility to deal with new hardware developments***
- **Many research challenges remain**
  - ***high-level language features for multi-threading***
  - ***architecture- and application-adaptive compilation and runtime systems that employ intelligent search strategies (ATLAS-like)***
  - ***intelligent tools and middleware that provide efficient support for program development, performance tuning, fault tolerance, and power management***
  - ***performance-porting of legacy applications***

Example
BRD Distribution with CRS Layout

```
class BRD: Distribution {
  …
 function map(i:index(source)):locale{…}; /* global mapping for dense domain */
 function GetDistributionSegment(loc:locale):domain(1){…};  /* “box” for loc */
  …
}

class CRS: LocalSegment {
 const loc: locale = this.getLocale();
  /* declaration of dense and sparse distribution segment for locale loc: */
 const locD: domain(2);
 const locDD: sparse domain(locD) = GetDistributionSegment(loc);
  …
 const LocalDomain:  domain(1)=1..nnz;     /* local data domain */
  /* persistent data structures in the local segment: */
 var cx: [LocalDomain] index(locD(2));     /* column index vector */
 var ro: [l1..u1+1] index(xLocalDomain);  /* row vector */
  …
 function define_column_vector(): {[z in LocalDomain] cx(z)=nz2x(z)(2)}
 function define_row_vector(): {…}
  …
/* mapping global index to index in local data domain: */
 function    layout(i: index(D)): index(LocalDomain) return(x2nz(i))
 constructor LocalSegment(){define_column_vector(); define_row_vector(); }
}
```

Implementation Target Architecture: Cluster of Cell Broadband Engines



Fault tolerance must be applied across all levels of the system hierarchy:

SPE → PPE → CBE → Cluster

Case Study: Introspection Sensors for Performance Tuning

Introspection *sensors* yield information about the execution of the application:

- **Hardware Monitors**
  - *accumulators: counting standard events (cache misses, loads, FP ops,..)*
  - *timers: analysis of latencies and stalls*
  - *programmable watch events for special conditions*
- **Low-level Software Monitoring *(at message-passing level)***
  - *waiting times for blocking send and receive*
  - *communication transfer times*
  - *barrier synchronization times*
  - *...*
- **High-Level Software Monitoring (*at the level of a high-level language)***
  - *timing for redistribution of a globally distributed collection*
  - *timing for function invocation, loop, or program region*
  - *timing for computing a communication schedule ("inspector")*
  - *evaluation of assertions and invariants*
  - *...*

Case Study: Introspection Actuators for Performance Tuning

Introspection actuators provide mechanisms, data, and control paths for implementing feedback to the application, depending on results of analysis and prediction:

- **Instrumentation and Measurement Retargeting**
- **Resource Reallocation**
- **Computational Steering**
  - *changing the implementation of an application section*
    - *changing a function implementation by choosing a more efficient algorithm*
    - *changing the implementation of a loop*
    - *changing the distribution of key data structures, with the goal of load balancing*
- **Program Restructuring and Recompilation (offline)**

Berkeley's 7 +6 Dwarfs

1. Dense Linear Algebra (BLAS, ScaLAPACK,MATLAB)
2. Sparse Linear Algebra (SpMV, SuperLU)
3. Spectral Methods (FFT)
4. N-Body Methods (Barnes-Hut, Fast Multipole)
5. Structured Grids (Cactus, Magneto-Hydrodynamics)
6. Unstructured Grids (ABAQUS,FIDAP)
7. Monte Carlo
8. Combination Logic (Encryption; Cyclic Redundancy Codes—CRC)
9. Graph Traversal (Quicksort)
10. Dynamic Programming
11. Backtrack and Branch and Bound
12. Construction of graphical models (Bayesian networks, Hidden Markov Models)
13. Finite State Machines

The Traditional Approach will not Scale

- **Traditional approach based on rad-hard processors and fixed redundancy (e.g.,Triple Modular Redundancy—TMR)**
  - *Current Generation (Phoenix and Mars Science Lab –'09 Launch)*
    - *Single BAE Rad 750 Processor*
    - *256 MB of DRAM and 2 GB Flash Memory (MSL)*
    - *200 MIPS peak, 14 Watts available power (14 MIPS/W)*
  - *ST8 Honeywell Dependable Multiprocessor*
    - *COTS system with Rad 750 controller (100 MIPS) and IBM PowerPC 750FX (1300 MIPS)*
    - *120 MIPS/Watt Performance*
    - *Fault tolerant architecture*
- **Rad-hard processors today lag commercial architectures by a factor of about 100 (and growing)**
- **By 2015: a single rad-hard processor may deliver about 1 GF—orders of magnitude below requirements**
- **COTS-based multicore systems will be able to provide the required capability, but there are serious issues to be addressed…**

Introspection versus Traditional V&V

- **Introspection**
  - *focuses on execution time monitoring, analysis, recovery*
  - *actual work considers transient and hard faults, not design errors*
- **Verification & Validation:**
  - *focuses on design errors*
  - *is applied before actual program execution*
- **Verification has the goal to prove that a program conforms to its specification for *all* legal inputs**
- **Test proves or disproves correctness of the program for *specific* (range of) inputs**
- **Both verification and test are *not complete*:**
  - *problems may be undecidable or intractable*
  - *tests can prove existence of faults, not their total absence*

Key Applications for Future Architectures: Berkeley's "Dwarfs"

1. Dense Linear Algebra (BLAS, ScaLAPACK,MATLAB)
2. Sparse Linear Algebra (SpMV, SuperLU)
3. Spectral Methods (FFT)
4. N-Body Methods (Barnes-Hut, Fast Multipole)
5. Structured Grids (Cactus, Magneto-Hydrodynamics)
6. Unstructured Grids (ABAQUS,FIDAP)
7. Monte Carlo
8. Combination Logic (Encryption; Cyclic Redundancy Codes—CRC)
9. Graph Traversal (Quicksort)
10. Dynamic Programming
11. Backtrack and Branch and Bound
12. Construction of graphical models (Bayesian networks, Hidden Markov Models)
13. Finite State Machines

Early Alternatives to MPI

- **Automatic Vectorization and Parallelization**
  - *automatic vectorization (for inner loops) and parallelization (for SMPs) were successful in limited contexts*
  - *in general, automatic parallelization is essentially intractable*
- **Data parallel languages for MPPs and clusters**
  - *pioneered by compiler projects at Caltech (Cosmic Cube) and U of Bonn (SUPERB Fortran parallelizer)*
  - *key features of data parallel languages*
    - *global name space*
    - *single thread of control*
    - *loosely synchronous parallel computation*
    - *automatic generation of communication*
  - *key language developments*
    - *IVTRAN (1973) – for the SIMD ILLIAV IV – first language to allow control of data layout*
    - *MPP languages: Kali, Fortran D, Vienna Fortran, Connection Machine Fortran*
    - *High Performance Fortran (HPF) result of a standardization effort*

Sensors and Actuators

- ***Sensors*** **and *actuators* link the introspection framework to the application and the environment**
- **Sensors: provide *input* to the introspection system**

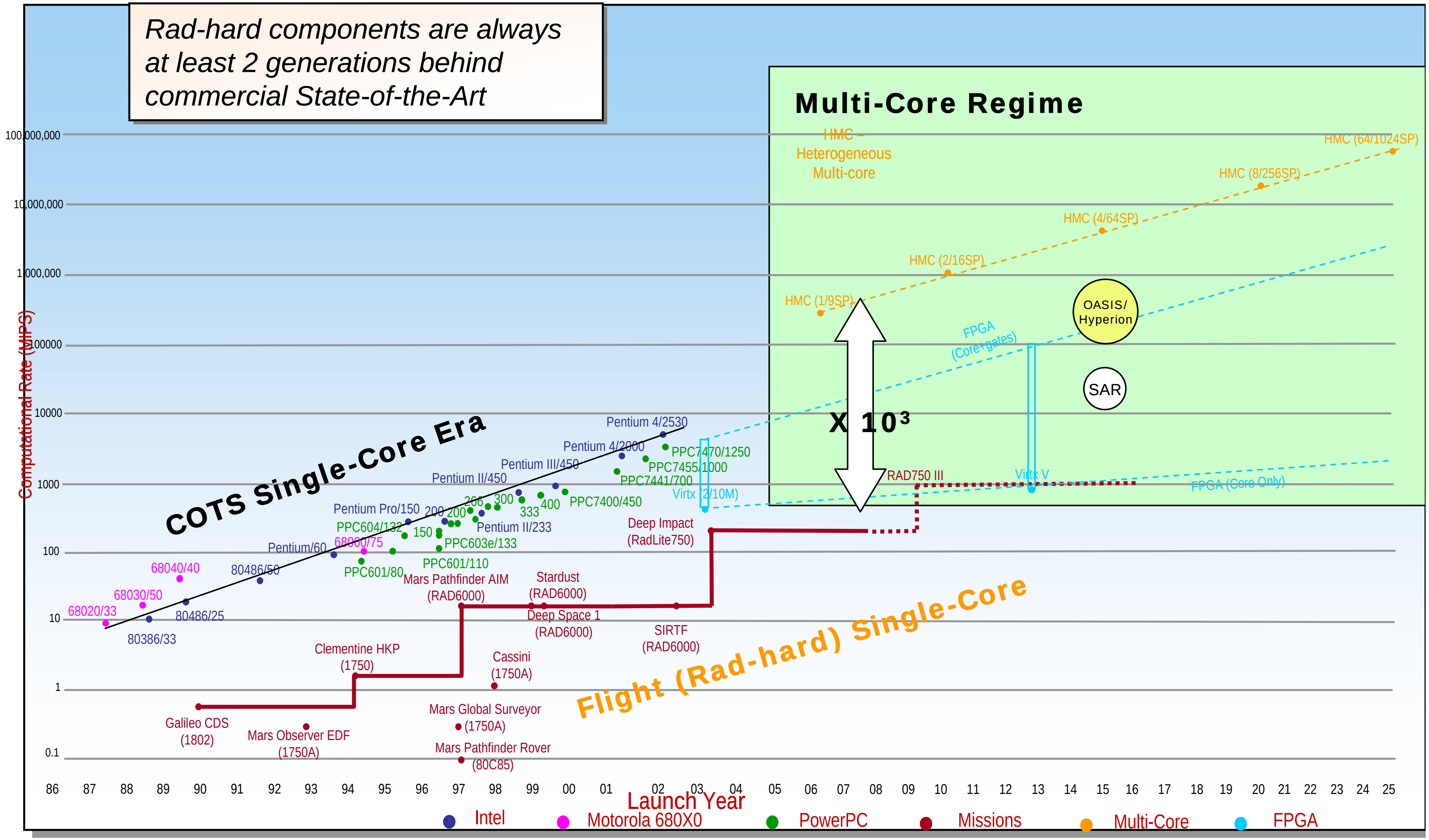
  Examples for sensor-provided inputs:
  - *state of a variable, data structure, synchronization object*
  - *value of an assertion*
  - *state of a temperature sensor or hardware counter*
- **Actuators: provide *feedback* from the introspection system**

  Examples for actuator-triggered actions:
  - *modification of program components (methods and data)*
  - *modification of sensor/actuator sets (including activation and deactivation)*
  - *local recovery*
  - *signaling fault to next higher level in a hierarchical system*
  - *requesting actions from lower levels in a hierarchical system*

Application-Aware Fault Detection Assertions: Examples

- **Assertions based on general program structures**
  - *values and value ranges for variables, subscript expressions, pointers*
  - *sequential and parallel control flow patterns*
  - *locality and communication assertions*
  - *independence assertions for data-parallel loops*
  - *real-time constraints*
  - *safety and liveness properties*
- **Domain-specific assertions: exploiting knowledge about:**
  - *target system: hardware and software*
  - *application domain*
    - *libraries: pre- and post conditions, argument constraints*
    - *data structure invariants*
    - *control constraints*
    - *data representation and distribution knowledge (e.g., CRS for distributed sparse matrices)*
    - *communication patterns and schedules for parallel constructs*

Space Flight Avionics and Microprocessors History and Outlook

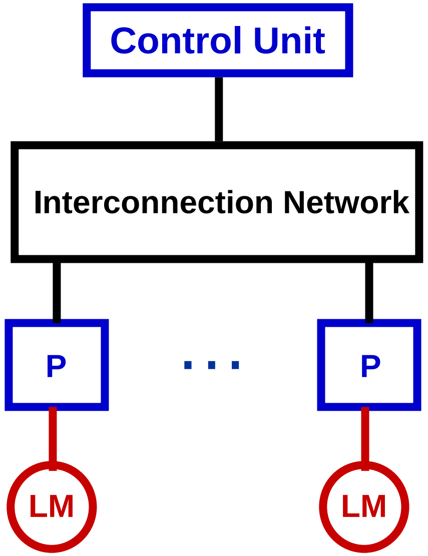


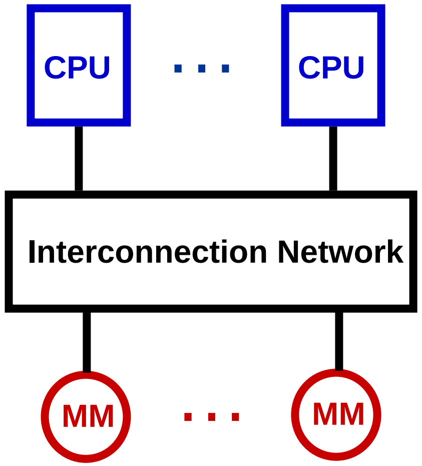
Source: Contributions from Dan Katz (LSU), Larry Bergman (JPL), and others

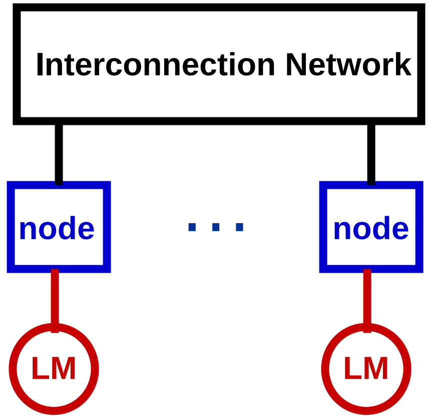
Transient Faults

- **SEUs and MBUs are radiation-induced transient hardware errors, which may corrupt software in multiple ways:**
  - *instruction codes and addresses*
  - *user data structures*
  - *synchronization objects*
  - *protected OS data structures*
  - *synchronization and communication*
- **Potential effects include:**
  - *wrong or illegal instruction codes and addresses*
  - *wrong user data in registers, cache, or DRAM*
  - *buffer overflows*
  - *control flow errors*
  - *unwarranted exceptions*
  - *hangs and crashes*
  - *synchronization and communication faults*

Basic Parallel Architecture Paradigms







Single-Instruction-Multiple-Data (SIMD)
Hardware implementation of a data parallel model of computation

ILLIAC IV, DAP, BSP, CM-1, CM-2, MasPar MP-1…

Vector Computers
Cray 1, Cyber 205, …

Symmetric Multiprocessors (SMP)
Uniform Memory Access (UMA)

Alliant FX, Digital VAX 8800, Sequent Balance,…

Distributed-Memory Multiprocessors (MPPs)
Non-Uniform Memory Access (NUMA)

Cosmic Cube, Suprenum, Transputers,…

Clusters

In modern multicore-based architectures, such building blocks may be hierarchically combined in many different configurations

Distributions: Outline of a Formal Framework

- Let I denote the index set of a domain, and L the index domain for a set locales. A **data distribution**

  $$\delta: I \rightarrow L$$

  is a total function that specifies for each element in I an associated locale

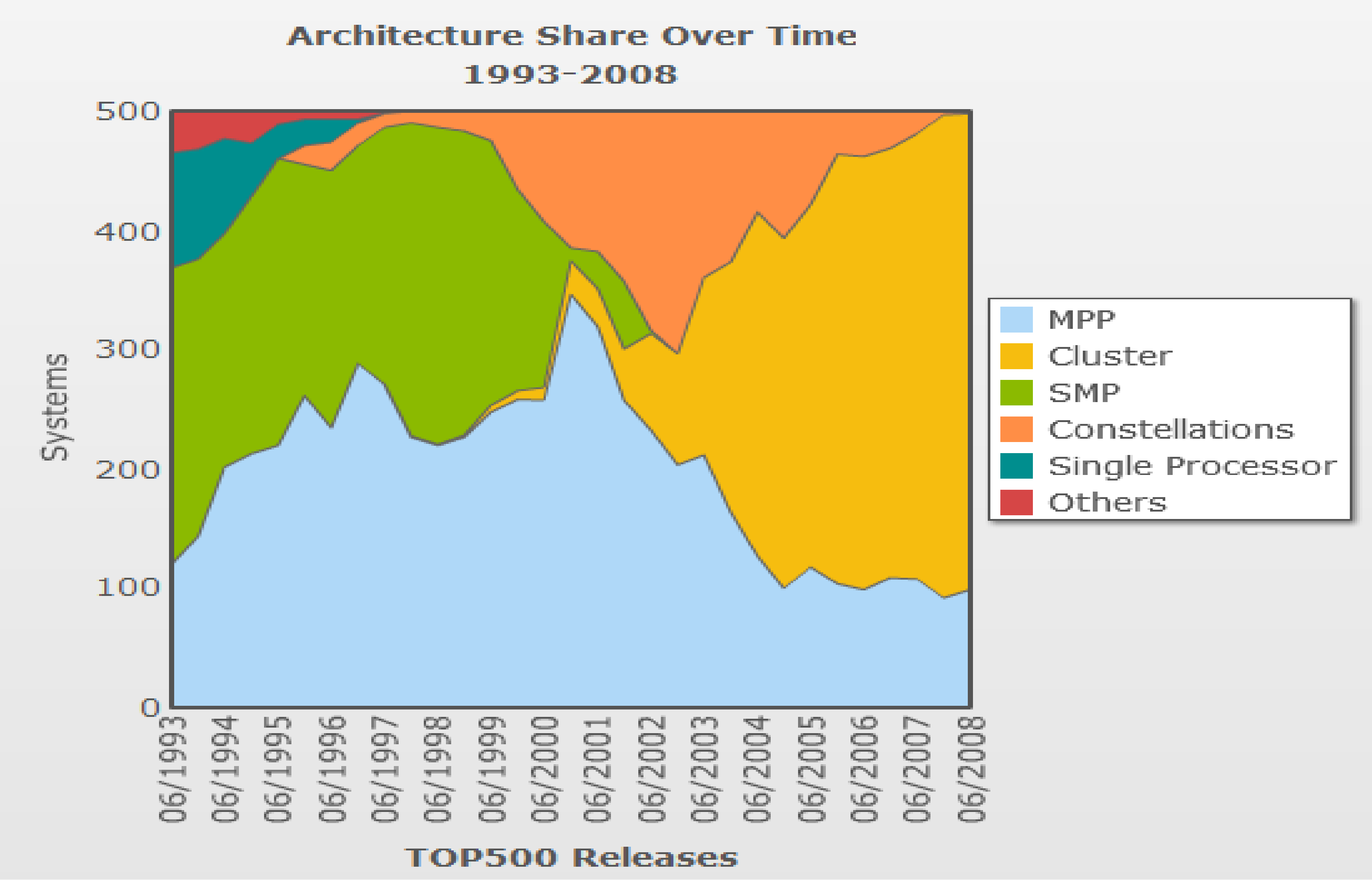
- Let I_1, I_2, denote index domains. An **alignment** from I_1 to I_2, is a total function

  $$\alpha: I_1 \rightarrow I_2$$

  that associates an index in I_2, with every index of I_1. If I_2 has a distribution, δ_2, then a distribution, δ_1, for I_1, is obtained as $\delta_1 = \delta_2 \circ \alpha$

- **Affinity** between distributed data and threads can be formalized in a similar way

Top 500 Architectures



Domains

- **Concept influenced by HPF templates, ZPL regions**
- **Domains are first-class objects**
- **Domain components**
  - ***index set***
  - ***distribution***
  - ***set of arrays***
- **Index sets are general sets of "names"**
  - ***Cartesian products of integer intervals (as in Fortran95 etc.)***
  - ***sparse subsets of Cartesian products***
  - ***sets of object instances, e.g., for graph-based data structures***
- **Iterators based on domains**

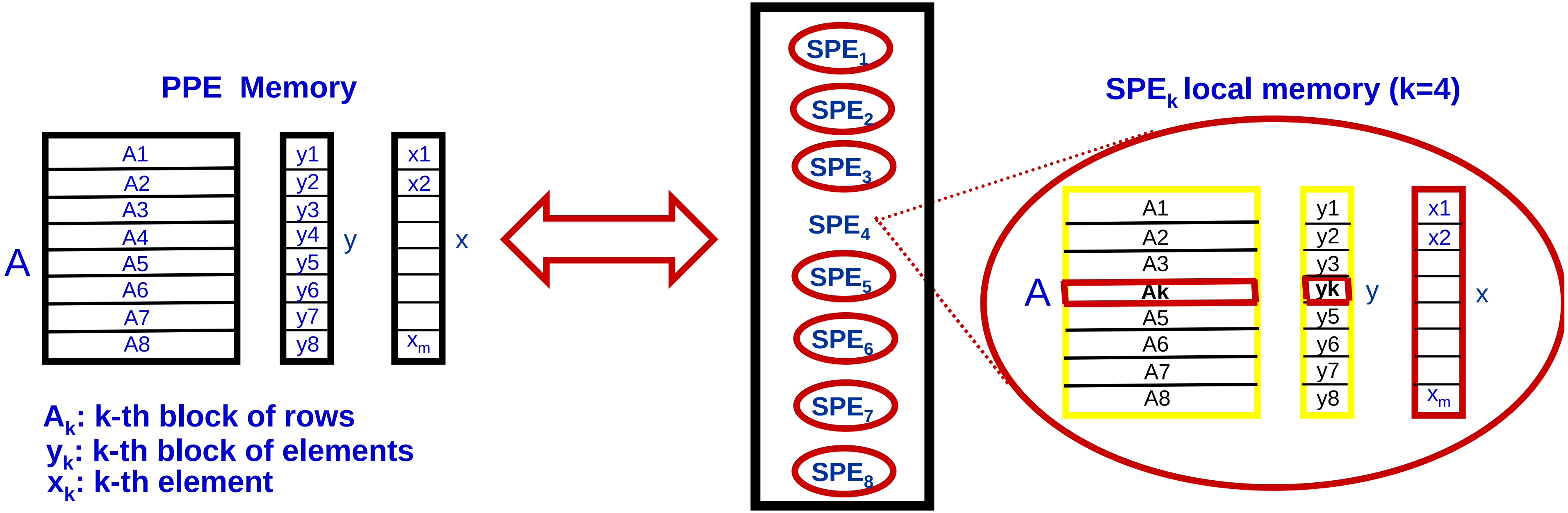
Example: Possible Extensions for the CELL Matrix-Vector Multiply

```
var A: [1..m,1..n] real;
var x: [1..n]      real;
var y: [1..m]      real;

y = sum reduce(dim=2) forall (i,j) in [1..m,1..n] A(i,j)*x(j);
```

(original) Chapel version

```
param n_spe = 8;                  /* number of synergistic processors (SPEs) */
const SPE:[1..n_spe] locale;      /* declaration of SPE array */

var A: [1..m,1..n] real distributed(block,*) on SPE;
var x: [1..n]      real replicated           on SPE;
var y: [1..m]      real distributed(block)   on SPE;

y = sum reduce(dim=2) forall (i,j) in [1..m,1..n] A(i,j)*x(j);
```

Chapel with (implicit) heterogeneous semantics

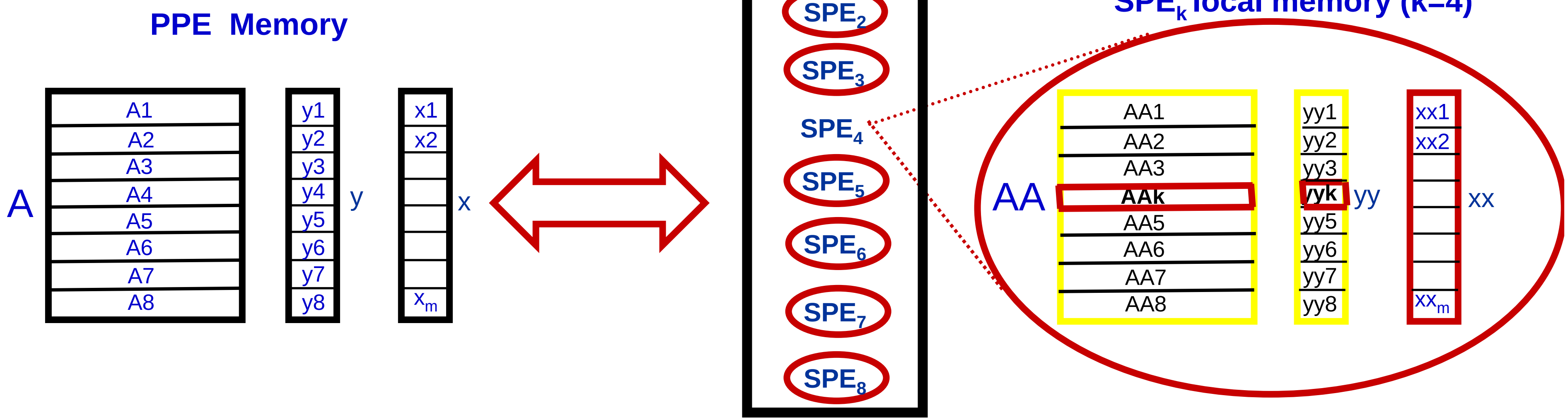


A_k: k-th block of rows
y_k: k-th block of elements
x_k: k-th element

Example
Matrix-Vector Multiply on the CELL: V2

```
param n_spe = 8;                    /* number of synergistic processors (SPEs) */
const SPE:[1..n_spe] locale;                   /* declaration of SPE locale array */
const PPE: locale                                    /* declaration of PPE locale */

var A: [1..m,1..n] real on PPE linked(AA) distributed(block,*) on SPE;
var x: [1..n]      real on PPE linked(xx) replicated           on SPE;
var y: [1..m]      real on PPE linked(yy) distributed(block)   on SPE;

AA=A; xx=x;                                    /* copy and distribute A, x to SPEs */
yy=sum reduce(dim=2) forall (i,j) in [1..m,1..n] on locale(xx(j)) AA(i,j)*xx(j);
y=yy;                                                       /* copy yy back to PPE */
```

Chapel/HETMC with explicit transfers



User-Defined Distributions: Global Mapping(2)

/* declaration of distribution class MyC1: */

```
class MyC1: Distribution {                         /* cyclic(1) */
  const ntl:int;                                   /* number of target locales */
  function map(i:index(source)):locale {           /* global mapping for MyC1 */
    return Locales(mod(i-1,ntl)+1);
  }

  /* set of local iterators : */
  iterator DistSegIterator(loc: index(target)): index(source) {
   const N: int = getSource().extent;
   const k: int = locale_index(loc);
   for i in k..N by ntl { yield(i); }
  }
  /* distribution segment : */
  function GetDistributionSegment(loc: index(target)): Domain {
   const N: int = getSource().extent;
   const k: int = locale_index(loc);
   return (k..N by ntl);
  }
}
```

/* use of distribution class MyC1 in declarations: */

```
const D1C1: domain(1) distributed(MyC1()) on Locales(1..4)=1..16;
var   A1: [D1C1] real;
...
```

An Approach to Application-Oriented Introspection-Based Fault Tolerance in the HPCS

- **Approach based on a (mission-dependent) fault model**
  - *classifies faults (fault types, severity)*
  - *specifies fault probabilities, depending on environment*
  - *prescribes recovery actions*
- **Addressing fault detection, analysis, isolation, recovery**
- **Exploiting knowledge from different sources**
  - *automatic generation of assertions based on:*
    - *static analysis and profiling*
    - *properties of target system hardware and software*
    - *application domain (libraries, data structures, data distributions)*
  - *user-provided assertions and invariants*
- **Leveraging existing technology**
  - *fixed-redundancy for small critical areas in a program*
  - *Algorithm-Based Fault Tolerance (ABFT): standard matrix methods*
  - *integration of high-level generator systems such as CMU's "SPIRAL"*

X10 and Fortress: Some Key Properties

- **X10 --- the IBM HPCS Language**
  - *object-oriented; serial sublanguage based on Java*
  - *an array sublanguage supports the distribution of multi-dimensional arrays via standard methods*
  - *sequential and parallel iterators, either local or global*
  - *asynchronous activities*
- **Fortress --- the SUN HPCS Language**
  - *object-oriented, with some relationship to Java*
  - *supports Unicode and conventional mathematical notation: e.g.,* $y = a \sin 2x + \cos 2x \log \log x$
  - *strong security model*
  - *support for language "growth" via inclusion of libraries*
  - *by default, arrays are distributed and loops are parallel*

Co-Array Fortran

- **Extension of Fortran to allow SPMD-style programming**
- **Introduces a new type of array dimension (*co-array*) to refer to the cooperating instances ("*images*") of an SPMD program, making processor boundaries explicit:**

```
integer :: a(n.m) [*]
```

this introduces a shared co-array *a* with *n*m* integers local to each processor image

- **Non-local variables can be directly referenced based on a corresponding syntax extension:**

```
a(1,:) [p]
```

references the first row of co-array *a* in processor *p*

- **a *barrier* provides synchronization between images**

UPC

- Support for a global address space model for SPMD parallel programs, in which threads share part of their address space
- The shared space is logically partitioned into fragments, each of which is associated with a thread
- *Shared* arrays are distributed in block-cyclic fashion among threads
- The *upc_forall* construct supports work sharing for a parallel loop
- Additional features include special constructs for pointers (private/shared), non-blocking barriers, and collective operations

An Approach to Application-Oriented Introspection-Based Fault Tolerance in the HPCS

- **Approach based on a (mission-dependent) fault model**
  - *classifies faults (fault types, severity)*
  - *specifies fault probabilities, depending on environment*
  - *prescribes recovery actions*
- **Addressing fault detection, analysis, isolation, recovery**
- **Exploiting knowledge from different sources**
  - *automatic generation of assertions based on:*
    - *static analysis and profiling*
    - *properties of target system hardware and software*
    - *application domain (libraries, data structures, data distributions)*
  - *user-provided assertions and invariants*
- **Leveraging existing technology**
  - *fixed-redundancy for small critical areas in a program*
  - *Algorithm-Based Fault Tolerance (ABFT): standard matrix methods*
  - *integration of high-level generator systems such as CMU's "SPIRAL"*

Example: PGAS vs. HPCS
Setting up a block-distributed array in Titanium vs. Chapel

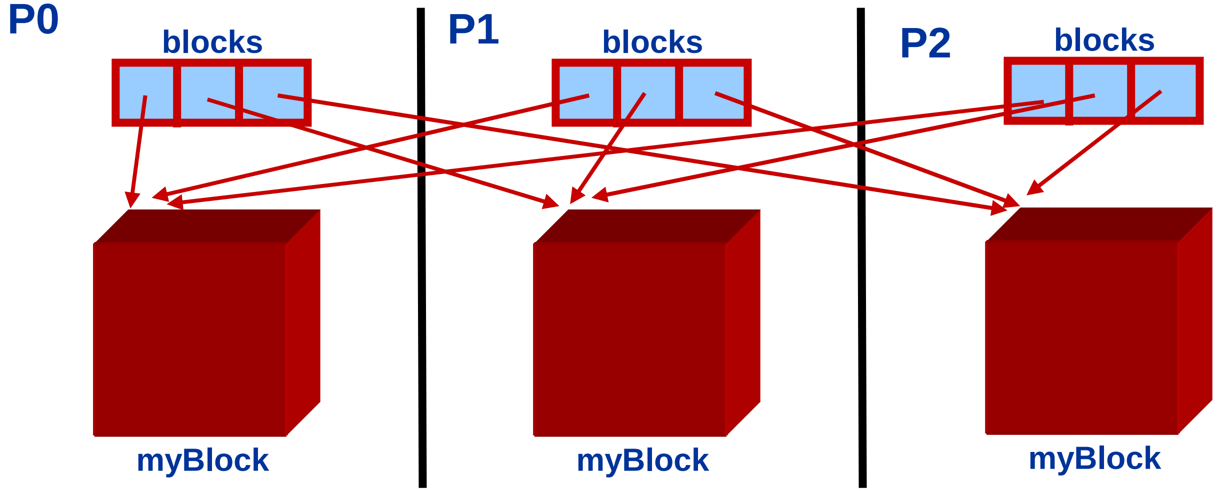
Titanium: *a dialect of Java that supports distributed multi-dimensional arrays, iterators, subarrays, and synchronization/communication primitives*

Titanium Code Fragment

```
// determine parameters of local block:
Point<3> startCell = myBlockPos * numCellsPerBlockSide;
Point<3> endCell   = startCell + (numCellsPerBlockSide-[1,1,1]);

//create local myBlock array:
double [3d] myBlock = new double[startCell:endCell];

//build the distributed structure:
//declare blocks as 1D-array of references (one element per processor)
blocks.exchange(myBlock);
```

Chapel Code Fragment

```
const D: domain(3) distributed (block)
                              = [l1..u1,l2..u2,l3..u3];
…
var A: [D] real;
…
```



Source: K.Yelick et al.: Parallel Languages and Compilers: Perspective from the Titanium Experience