海外招待講演

High-Productivity Languages for Peta-Scale Computing

Hans P. Zima
Principal Scientist, Jet Propulsion Laboratory, California Institute of Technology, Professor Emeritus, University of Vienna, Austria

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

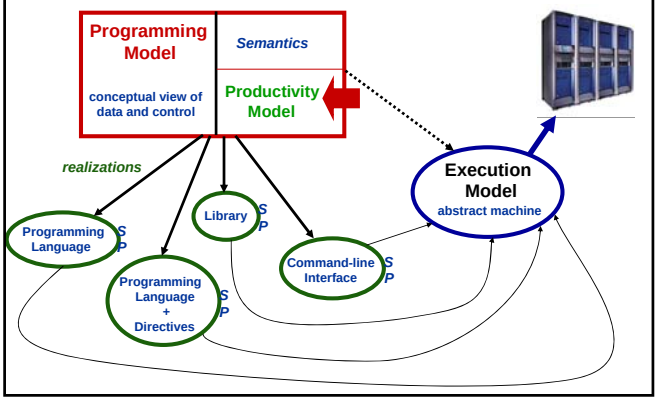
Fujitsu HPC Forum 2008
Tokyo, Japan, August 27th, 2008

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*

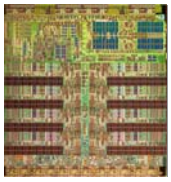
High-Productivity
Programming and Execution Models



Contents

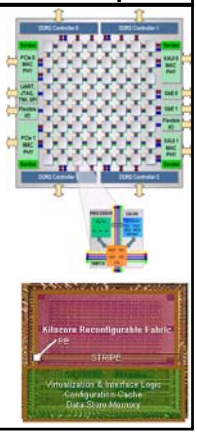
Multicore: An Emerging Technology

- 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 XCEII 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

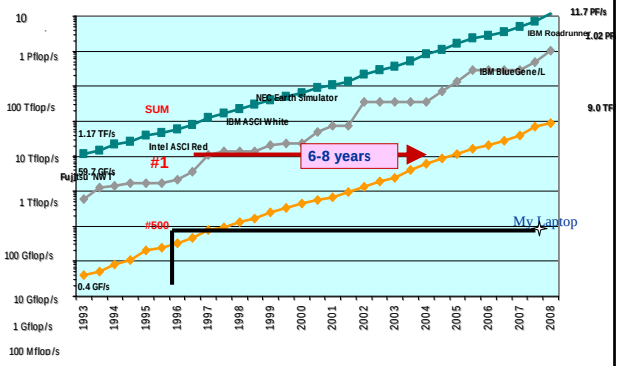
- **Tile64 (Tilera Corporation, 2007)**
  - *64 identical cores, arranged in an 8X8 grid*
  - *iMesh on-chip network, 27 Tb/sec bandwidth*
  - *170-300mW per core; 600 MHz – 1 GHz*
  - *192 GOPS (32 bit)—about 10 GOPS/Watt*
- **Kilocore 1025 *(Rapport Inc. and IBM, 2008)***
  - *Power PC and 1024 8-bit 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

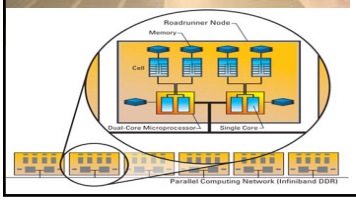


From Eniac (1946) …

10^3 OPS

…to LANL Roadrunner: Top 500 #1

Cell Blade



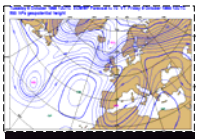
1,026 TF=10^{15} OPS

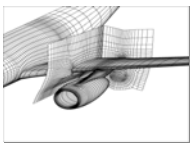
The first machine reaching Peta-scale performance

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*

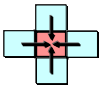
Alternatives to MPI have been proposed

- *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)

Sequential Code

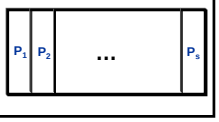
```
real, allocatable A(:, : ), B(:, : )
  …
do while ( .not. converged )
  do J=1,N
     do I=1,N
        B(I,J)=0.25(A(I-1,J)+A(I+1,J)+A(I,J-1)+A(I,J+1))
     enddo
  enddo
A(1:N,1:N)=B
…
enddo
```



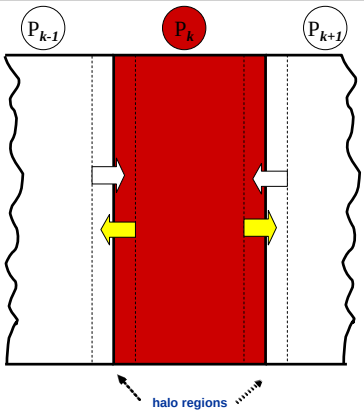
dependence pattern

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: Data Exchange

Processor P_k **reads:**

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

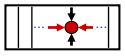
Processor P_k **copies:**

- *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)
```

***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)
```

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 …
                 …
```

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)

※最終ページに拡大印刷あり

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

global view of data, global 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

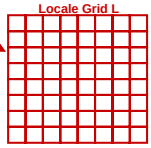
Example: Jacobi Relaxation in Chapel

```
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)=[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;

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**

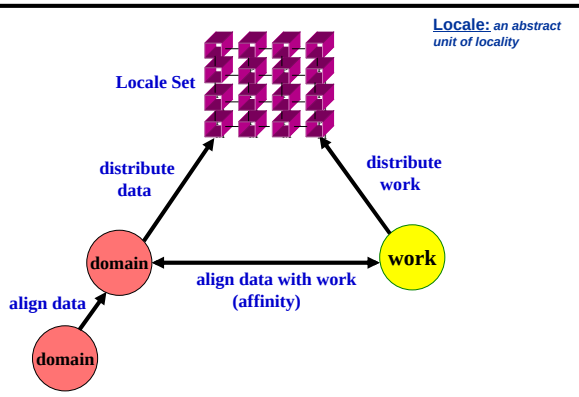
  cobegin *{ S_1,…S_n}* — *executes the S_i in parallel (i = 1,…n)*

- **Task Localization**

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

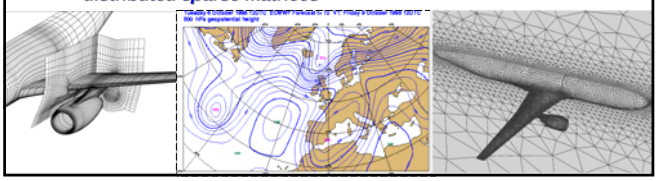
- **Task Synchronization**
  - *atomic sections*
  - *sync variables*
  - *single-assignment variables*

Aspects of Locality



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

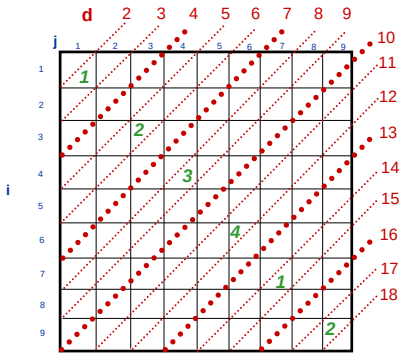
/* declaration of distribution classes MyC and MyB: */

```
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
19	0	0	0	0
0	0	0	0	0
0	0	0	17	0
0	0	0	0	93
0	0	0	0	0

0	0	0	
0	21	0	
0	0	16	
72	0	0	
0	0	0	
0	0	0	
0	13	0	

0	0	0	0
0	23	69	0
27	0	0	11

44	0	0	19
37	0	0	0
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

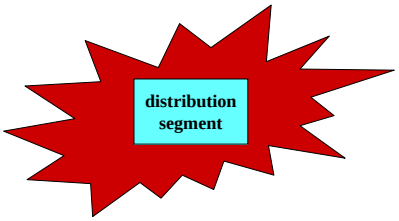
D^1	C^1	R^1
21	7	1
16	8	1
72	6	2
13	7	3
		4
		4
		4
		5

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 global view of data, but local 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

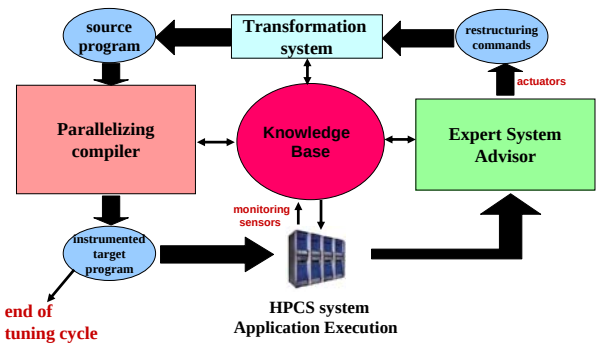
Issues in Programming Environments

- 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*

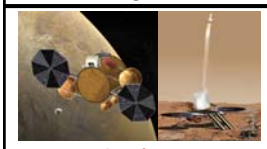
Legacy Code Migration

- **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

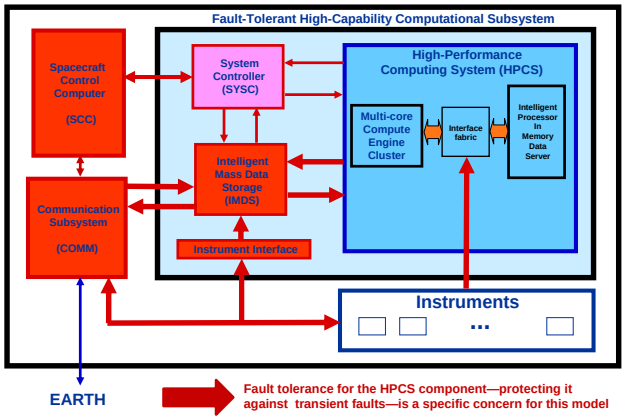

Titan Explorer


Neptune Triton Explorer


Europa Astrobiology Laboratory

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

High-Capability On-Board System: An Example

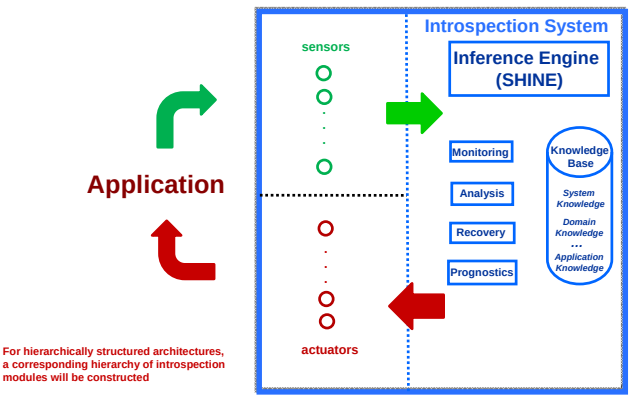


A Framework for Introspection

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

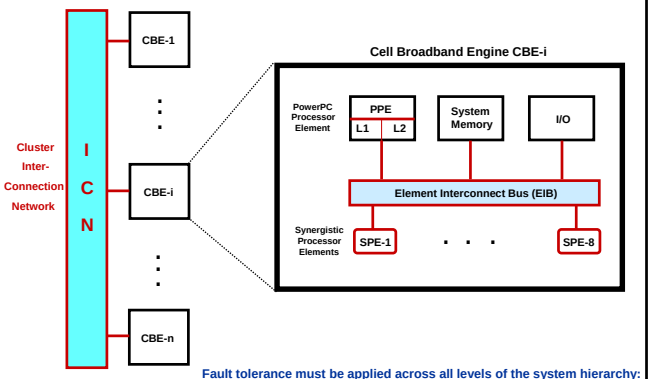
Conclusion

- 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

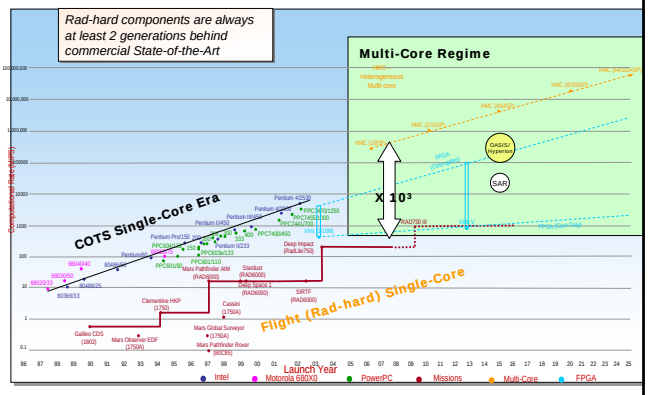
  Examples for sensor-provided inputs:
  - *state of a variable, data structure, synchronization object*
  - *value of an assertion*
  - *state of a temperature sensor or hardware counter*
- Actuators: provide *feedback* from the introspection system

  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

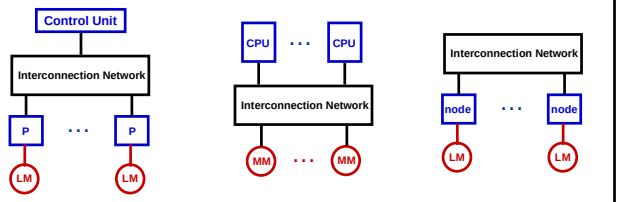


Source: Contributions from Dan Katz (LSU), Larry Bergman (JPL), and others

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**

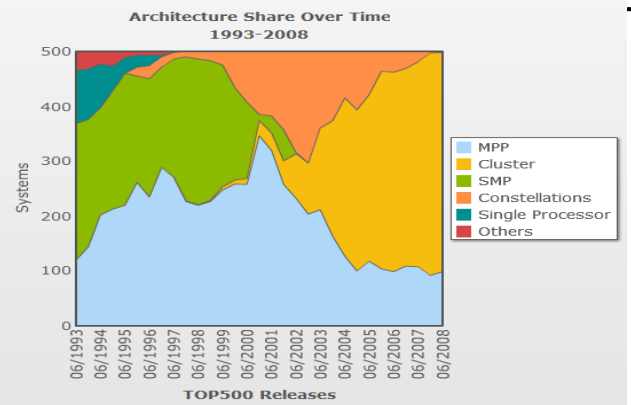
  $$\delta: I \rightarrow L$$

  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**

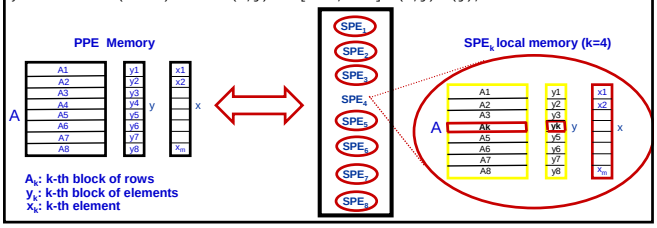
Example: Possible Extensions for the CELL Matrix-Vector Multiply

```
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);
```

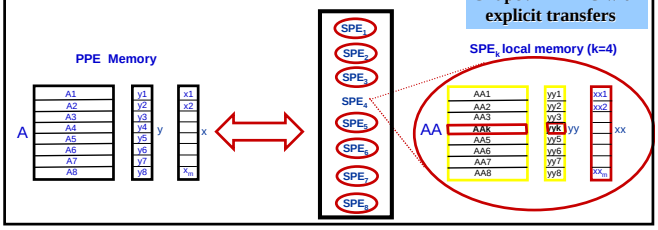
Chapel with (implicit) heterogeneous semantics



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 2 x + cos 2 x 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

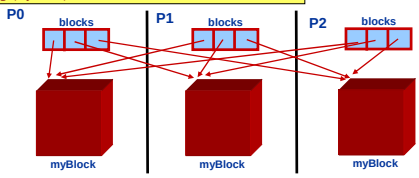
Titanium: *a dialect of Java that supports distributed multi-dimensional arrays, iterators, subarrays, and synchronization/communication primitives*

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

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,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,i1,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
```

科学技術計算分科会 選出
2008 年度 科学技術計算分科会 より

アプリケーションとアーキテクチャ

工学院大学

小柳 義夫

サイエンティフィック・システム研究会　2008年 8 月 27 日
SS研 HPCフォーラム2008「ペタスケール・アプリを支える基盤技術」

アプリケーションとアーキテクチャ

工学院大学
小柳義夫
oyanagi@cc.kogakuin.ac.jp
oyanagi@cc.kogakuin.ac.jp

[アブストラクト]

コンピュータのアーキテクチャとアプリケーションがどのような相互作用を行って来たかについて総括する。最初の電子式自動計算機 ENIAC は、弾頭計算という特定のアプリのために製作されたが、多くの目的のために利用された。1964 年の IBM 360 シリーズの発表以来「汎用コンピュータ」がコンピュータ設計の基本原理とされ、どのようなアプリケーションにも対応できるという幻想を振りまいた。1976 年の Cray-1 を嚆矢とするスーパーコンピュータの出現は、科学技術計算を目的としたコンピュータの意義を確認させた。その後、マイクロプロセッサの高速化と低価格化により「汎用チップを用いた専用計算機」が可能になったが、マルチコア化のためにデータ転送がボトルネックになりつつある。与えられたコンピュータシステムでいかに速く計算するかという努力は重要であるが、新しいコンピュータはアプリケーション（複数）を想定して設計するべきである。何にでも速いコンピュータを作るという目標では設計上の判断ができず中途半端なものになってしまうであろう。

[キーワード]

アーキテクチャ、アルゴリズム、専用計算機、並列計算機、科学技術計算

[概要]

1．はじめに

この講演を依頼されて思い出したのは、現在神戸で建設が進んでいる次世代スーパーコンピュータの概念設計評価作業部会（2007 年）での議論である。文部科学省の提案によると、これまでの日本の主導的スーパーコンピュータであった、NWT，cp-pacs，地球シミュレータが専用計算機であったのに対し、これは汎用スーパーコンピュータであり、どんな問題にも対応できるということが強調された。私やI先生はこれに強く反対し、NWT も cp-pacs も地球シミュレータも、特定のアプリを目的として設計されたが、実は汎用的計算機であり、特定のアプリケーションを強く意識して設計したがゆえに、（逆説的に）多くのアプリケーションに役立つ汎用コンピュータになったことを指摘した。私が言うまでもなく、設計とは、有限の資源のもとにおける多くの技術要素のトレードオフであるから、最初から汎用を設計目標にすると、「あれもほしい、これもほしい」ということになって、中途半端なものになってしまう危険がある。

考えてみると、普通の計算機が汎用で特定目的の計算機が例外、という概念は 1960 年代以降のものであって、最初からそうだったわけではない。チューリングやフォン・ノイマンの計算機モデルは先駆的であったが、初期の計算機は特定目的の計算機であり、しばらくは事務用と科学技術計算用の計算機は別に作られていた。これを統一したIBMの System360 は画期的であり、その後の各社のコンピュータのモデルとなったが、これは直前の状況へのいわばアンチテーゼとして出されたものであった。1980 年代以降には、ベクトルスーパーコンピュータや並列計算機が出現し、状況は変わったにもかかわらず、「コンピュータは汎用であるべきである」という神話が生きているのは残念なことである。

以下では、「アプリを意識した計算機の方が役に立つ」という仮説を証明しようとしたのであるが、実際はなかなか難しい。

2. 機械式計算器におけるアーキとアプリ

現存する最古の歯車式計算器は、パスカルの作った「パスカリーヌ」(1642)と言われているが、そのアプリは父親の税金計算であった。ライプニッツは桁ずらしにより乗除算の出来る計算器を作ったが、そのアプリは天文計算であった。バベッジの階差機関(1822)の目的は、天文学や暦だと言われている。機械式計算器でも、アプリが想定されていることは(考えてみれば当たり前であるが)注目に値する。日本では、1902 年に矢頭良一が発明し製作した「自答算盤」は政府関係に多く売れたそうである。アプリは？

3. 計算機第1世代

真空管式計算機の ENIAC (1946)は配線によってプログラムするもので、その初期の目的は弾道計算であった。その後プログラム内蔵式に改良され、多くの目的に使われた。

4. 計算機第 2 世代(トランジスタ)＝HPC(-3)世代(STRETCH)

1955 年頃からトランジスタ式の計算機が登場したが、10 進演算が主の事務用計算機と、浮動小数演算が主の科学技術用計算機とに分かれていた。私は学生時代に OKITAC5090 を使ったことがあるが、10 進 12 桁の計算機であった。浮動小数もあったが、指数部 2 桁、仮数部 10 桁の 10 進表現であった。本来、事務用だったのでしょうか？

このころ IBM は STRETCH の開発をはじめ、1961 年にロスアラモス研究所に納入した。7 台しか製作されなかったが、既存の計算機より 100 倍高速で、スーパーコンピュータの走りと言えよう。アプリは 3 次元の流体力学計算などであったと思われる。

5. 計算機第 3 世代(メインフレーム)＝HPC(-2)世代(CDC6600, IBM2938)

1964 年IBMはICを使った 360 シリーズを発表し、科学技術用・事務用を問わず 360 度の全方位に向けたコンピュータを標榜した。汎用コンピュータの歴史がここに始まり、われわれもその呪縛のもとにある。

同じ年、アメリカ原子力委員会は各社に「革新的な計算機」を製作するよう促し、ここから後年の CDC Star-100、TI ASC、 ILLIAC-IV などが生まれたと言われている。やはりこの年、CDC 社は CDC6600 を発表し、科学技術用計算機として多数納入された。アメリカ空軍がイリノイ大学と ILLIAC-IV の契約を結んだのもこの年である。

IBM は最初のベクトル計算機 IBM 2938 Array Processor を 1968 年に出荷した。IBM360 の I/O チャンネルに接続するもので、アプリは石油探査であった。「汎用コンピュータ」を提唱した IBM 自身が、このようなアプリを明確に意識した計算機を出していることは注目に値する。

6. 計算機 3. 5 世代＝HPC(-1)世代(ASC, Star-100, ILLIAC IV, BSP)

IBM は 1970 年に LSI を使った 370 シリーズを発表し、各社もこれに追随したが、このころベクトル計算機が登場しつつあった。TI 社の ASC と、CDC 社の Star-100 である。また、並列計算機としては、ILLIAC IV (1976)や BSP(1980 年開発中止)などがある。

7. HPC 神代時代(Cray-1, 75APU, IAP, DAP, Cosmic Cube, MPP)

Cray-1 が登場したのは 1976 年であるが、そのアプリは科学技術全般であろう。同時にこのころ、様々な並列計算機も登場した。ICL DAP、CM*、LAU、Cosmic Cube、MPP などである。日本では FACOM 230-75 APU や、日立等の IAP が登場している。これらはまだ本格的なベクトル計算機とは言えないので、HPC 神代時代と名付けることにする。注目すべきことは、このころ日本でも多くの並列計算機の試作が行われていることで

ある。しかしいずれも実験機であって商品にはなったものはない。アプリは、シミュレーション、画像処理、信号処理などであった。

このうち、寿命の長かったのは、Cosmic Cube→ipsc→Paragon→ASCI Red、pacs→pax→cp-pacs→pacs-cs や Grape シリーズなどで、いずれもアプリが明確なものである、と私には見える。

8. HPC 第 1 世代(FPS, Cyber205, XMP, S810, V200, SX-2, PAX, iPSC/1, NCUBE/1, CM-1, T)

その後、1980 年代になると、ベクトルコンピュータの全盛期を迎えるが、注目すべきことは、アメリカなどでは、同時に並列コンピュータの商品が山ほど登場していることである。日本では試作機は多いが、あくまで特定目的に限定されていた。

9. 専用シミュレータ(Ising, QCD, MD)

このころから ASIC が利用可能になり、Ising 模型シミュレータ、格子ゲージ模型シミュレータ、古典粒子系シミュレータ、モンテカルロ専用機などいろいろ登場している。

10. 1990 年代の HPC

1993 年に Cray が T3D を、IBM が SP-1 を発表して、超並列分野に参入した。その後、1980 年代に登場していた多くのベンチャー超並列企業の倒産・吸収が続いた。その原因は様々で、この事情の裏にアプリがどう関係しているかは不明である。

11. ベンチマークと性能評価

コンピュータの性能を評価するためにベンチマーク・プログラムが用いられる。だから、ベンチマーク性能の高いコンピュータがよいコンピュータだと思いがちである。ベンチマークとしては、Mix、MIPS、LFK、Linpack、SPEC などいろいろあるが、いずれも万能ではない。

そもそもベンチマークはアプリケーションとアーキテクチャとの界面である。アプリケーションから見ればそのアプリケーションの実行速度を推定するためのデータである。従って、ユーザ側としては、対象となるアプリケーションの実行速度をよく推定できるようなベンチマークが望ましい。逆にアーキテクチャから見れば、アーキテクチャを設計するときの目標である。すべてのアプリケーションに対して最適なシステムを設計することは困難であるので、代表的な負荷を取って設計を考える。この両面(とくに後者)を忘れてはならない。

結局、ベンチマークもアプリケーションと独立なものでなく、性能の絶対評価ではない。

12. おわりに

歴史的にどこまで証明できたかは問題であるが、コンピュータの歴史においてアプリケーションが重要な論点であったことはご理解いただけたと思う。今後、アーキテクチャが複雑化し、マルチコア・メニーコアが普及すると、かなりクセのあるコンピュータが利用されるようになることが予想される。どんな計算も高速に実行できることは望み薄で、アプリケーションとの相性が問題になる。さらには、モデル化、アルゴリズムまでも再考する必要が生じるであろう。

アプリケーションとアーキテクチャ

小柳義夫
工学院大学情報学部
oyanagi@cc.kogakuin.ac.jp

—企画委員会からの要望—

アーキテクチャと応用アプリの歴史総括

- 何を狙ってそのアーキができて、アプリはどの様に対応してきたか。
- 実装技術、その時の課題、ブレークスルー、応用アプリの特性、高速化技術など。
- 大変難しい要望である。どこまで答えられたかは自信がないが。

何を言いたかったのか

- 日本の次世代スーパーコンピュータ計画「汎用」スーパーである、と強調された。曰く「NWTやcp-pacsや地球シミュレータは専用スーパーであった」
- 「『汎用』では設計が出来ない。」 I先生や私が強調。
- 「普通の計算機が『汎用』で、特定目的の計算機は例外」という考え方は時代的な概念。
- 「汎用」コンピュータはアンチテーゼとして登場
- 仮説「アプリを意識した計算機の方が役立つ。」を歴史的に証明したかった。
- それは成功したか？？？？

概要

- 歯車式計算器におけるアーキとアプリ
- 計算機第1世代（真空管）
- 計算機第2世代（トランジスタ）＝HPC第(-3)世代(STRETCH)
- 計算機第3世代（メインフレーム）＝HPC第(-2)世代(CDC6600, IBM2938)
- 計算機3. 5世代＝HPC第(-1)世代(ASC, Star-100, ILLIAC IV, BSP)
- HPC神代時代(Cray-1, 75APU, IAP, DAP, Cosmic Cube, MPP)
- HPC第1世代(FPS, Cyber205, XMP, S810, V200, SX-2, PAX, iPSC/1, NCUBE/1, CM-1, T-series)
- 専用シミュレータ（Ising, QCD, MD）
- 性能とベンチマーク
- コンピュータの設計戦略

「汎用」コンピュータの神話

- 歴史的に、コンピュータは目的（アプリ）を想定して設計され、製造されてきた。
- アンチテーゼとしての「汎用」コンピュータの概念
  - チューリング・マシン（計算可能性）
  - フォン・ノイマン・アーキテクチャ（プログラム内蔵）
  - IBM System 360 --- 「メインフレーム」の登場
  - Attack of killer micros
- まず、歴史的な「計算器」から

歯車式計算器

- 最初の計算機械：ウィルヘルム・シッカート(1592－1635)チュービンゲン大学教授（現存せず）
- 1642年、パスカル「パスカリーヌ」（加減算）
  - アプリ：税金計算
- 1674年、ライプニッツ（桁ずらしにより乗除算も）
  - アプリ：天文計算
- 1902年、矢頭良一が発明し製作した「自答算盤」
  - 販売先：陸軍省、内務省、農事試験場
- 1923年、大本寅治郎「虎印計算器」、後のタイガー計算器

ライプニッツ(1646年-1716年)

- 微分や積分の記号を発明
- 形式言語に当たるものを初めて考案
- 2進法を研究
- どんな推論も代数計算のように単純で機械的な作業に置き換えることができる(人工知能？)
- チューリングの先駆？

チャールズ・バベッジ(Charles Babbage)

- 1792～1871
- 1822年に階差機関(difference engine)構想
  - 完成せず。1991年、バベッジの本来の設計に基づいて階差機関が組み立てられ、完全に機能。
  - アプリ:天文学、暦
- 1830年代、さらに汎用的な解析機関を構想。いかなる数学の関数も計算できる機能を持つ。
  - パンチカード(**Jacquard Loomで使用**)で制御
  - 世界で初めての「プログラム可能」な計算機
  - これも実現せず。

パンチカードシステム(Punch Card System)

- ハーマン・ホレリスが発明。1890年の米国国勢調査のデータ処理で初めて使用された。
- 余談:Fortranで3HABCのHは彼に由来。
- 80桁のパンチカード(私も使った)
- 1896年、ホレリスはTabulating Machine社を設立(IBM社の源流の一つ)

コンピュータの前夜

- 1936年、アラン・チューリングが万能計算機械(チューリングマシン)の論文を発表。計算可能性、unsolvable problem
- 1938年、ドイツのコンラート・ツーゼが、機械式の計算機V1(後にZ1と改名)を作成。アプリは暗号か？
- 1939年、ツーゼが演算部がリレー、記憶部が機械式の計算機Z2を作成。
- 1940年、ツーゼが全リレー式のZ3を作成。Z3はプログラム可能な最初の計算機である。その後Z4も作成。
- 1942年、ジョン・アタナソフとクリフォード・ベリーが電子素子を使って演算処理をする世界初の機械 ABCを作成。アプリは連立1次方程式。実際には稼働せず。
- 1944年、Harvard Mark I出荷(電気機械式計算器)アプリは？

計算機第1世代(真空管)

- 1946年:最初の真空管式コンピュータENIACが完成
  - プログラムは配線による。後にプログラム内蔵へ。
  - 10進法の回路。
  - アプリ:弾道計算が初期の目的。その後いろいろ。
- まもなく商用化:UNIVAC-1(1951年)、IBM701(1953年)

計算機第2世代(トランジスタ)

- 1955年頃からトランジスタが利用される
- NEAC2201(1958)、IBM7090(1958)、CDC1604 (1958)、UNIVAC1100(1960)、TOSBAC2100(1959)、HITAC301(1959)、OKITAC5090(1960)、FACOM222(1961)など
- 証券会社、商事会社、保険会社などに納入
- 科学技術計算用:浮動小数演算が主。
  事務計算用:十進演算が主。
  両者が区別されていた。

HPC(-3)世代

- 1956:IBM starts 7030 project (known as STRETCH) 100 times faster than IBM 704 using transistors (2進法)
  - Atomic Energy Commission at Los Alamos.
  - 1961: first STRETCH computer to LANL (only 7 built)
  - アプリ:三次元の流体力学計算など
  - much of the technology re-surfaces in the later IBM 7090 and 7094.

IBM STRETCH (1960)
http://www03.ibm.com/ibm/history/history/images/ibm_stretch.jpg

第3世代(メインフレーム)

- 1964年:IBMがICを使った360シリーズを発表
- 科学技術用・事務用を問わず360度の全方位に向けたコンピュータを標榜
- 汎用コンピュータの歴史が始まる。
- ファミリー概念
  - 同じアーキテクチャをもった小型機から大型機まで用意
  - 新機種開発にあたっても過去の機種との互換性を重視

第3世代

- 第3世代メインフレームの代表例
- UNIVAC1108(1965) 、GE635/645(1965)、FACOM230(1965)、HITAC 8400(1966)、NEAC2200(1964)、CDC6400(1966)、B8500(1967)、NCRのCentury100(1968)、TOSBAC5600(1971)
- 汎用大型計算機の時代

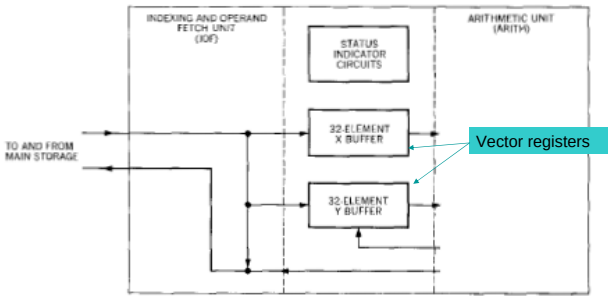
HPC (-2)世代

- 1964:Atomic Energy Commission urges manufacturers to look at "radical" machine structures. This leads to CDC Star-100, TI ASC, and ILLIAC-IV.
- 1964: Control Data Corporation produces CDC 6600(科学技術用計算機)
- 1964: Air Force signs ILLIAC-IV contract with University of Illinois. (Burroughs and TI)

HPC (-2)世代

- ベクトル計算機の登場
  - 演算パイプラインによってベクトル演算を高速に実行
  - 1965年にIBM社のSenzigらによって最初に提案。
  - パイプライン方式と並列方式を比較
- 石油探査のためIBM 2938 Array Processor (1968)
  - IBM360のI/Oチャンネルに接続する付加プロセッサとして開発

Figure 2 The IBM 2938 Array Processor



出典：IBM Systems Journal 8, #2, pp. 118-135 (1969)

Figure 3 An application program using array processor facilities

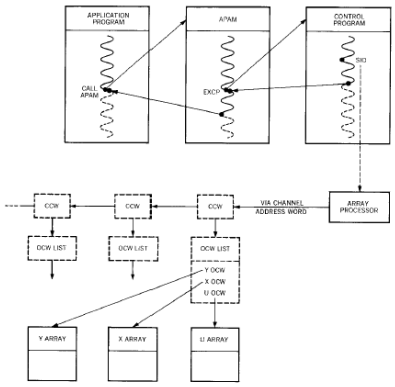


Table 22 FORTRAN IV matrix-multiply program

```
        Call APTIM
        Do 100 I = 1, N
        Do 200 J = 1, N
        A(I, J) = 0
        Do 300 K = 1, N
        A(I, J) = B(I, K)*C(K, J) + A(I, J)
300     Continue
200     Continue
100     Continue
        Call APTIM
```


Table 21 SYSTEM/360 model 65 performance comparison for matrix multiplication

Matrix size	FORTRAN IV *processing (seconds)*	*Array processor (seconds)*	*Performance improvement ratio*
100	60.85	1.26	48
92	47.56	0.94	50
84	36.22	0.74	48
76	26.84	0.57	47
68	19.26	0.41	46
60	13.23	0.28	47
52	8.62	0.19	45
44	5.24	0.12	43
36	2.86	0.07	40

1.58 MFlops

メインフレーム第3.5世代

- IBMは1970年にLSIを使ったIBM370シリーズを発表し、各社もこれに追随
- 日本では、通産省がIBM対抗機の開発を促進するため行政指導を行って企業連合
  - 富士通・日立はMシリーズ（IBM互換）
  - 三菱・沖はCOSMOシリーズ(IBM非互換）
  - 日本電気・東芝はACOSシリーズ(IBM非互換）
- いずれも1974年に発表した。

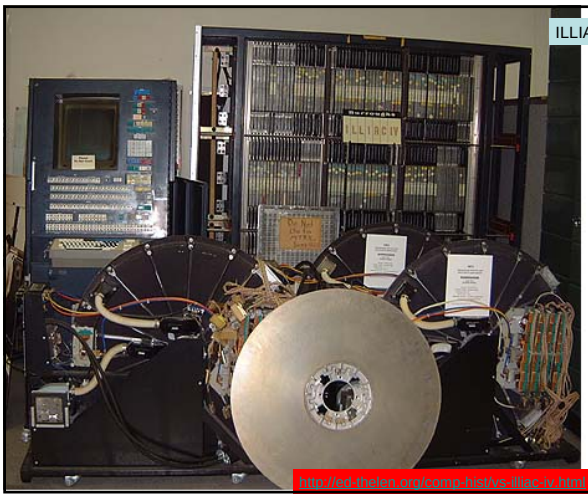
HPC（-1）世代

- ベクトル（パイプライン）計算機
  - ASC (1972, Texas Instruments社）
    - 30 MFlops、pipeline 1本、7機製作
    - 高速性。拡張性のある設計。アプリは？
    - Cray-1の登場により一夜のうちに販売停止
  - Star-100(1973, CDC社）STring ARray Computer
    - CDC7600の流れ。100 Mflopsを目標。
    - Memory-to-memory architecture。立ち上がり遅い。
    - スカラー速度を犠牲にした。結局性能は出ず。
    - 50 MFlops、4機製作。2台はLLNLへ。アプリは？

HPC(-1)世代

並列計算機

- ILLIAC IV (1973, Burroughs社、1976完全稼働)
  - 並列度64、50 MFlops、1機製作
  - Illinois大学からNASAへ。アプリ:流体力学など
  - 並列処理の人材を世界中に(日本にも)送り出した。
- BSP (1974設計開始、1980開発中止、Burroughs社)
  - 並列とパイプライン技術を併用
  - 並列度16、理論性能50 MFLops
  - 17個のmemory unit+ crossbar switch
  - アプリは?



HPC神代時代

- Seymour Cray
  - CDC (Control Data Corporation) 社において、CDC6600 (1964, 1 MFlops) およびCDC7600 (1969、5 MFLops) を設計
  - 1972年、CDC8600 計画が社内で拒否される
  - 同社を退社し、CRI社 (Cray Research Inc.) を設立
- CRI社は1976年160 MFlopsの性能をもつCray-1を出荷し、ロスアラモス研究所に納入

HPC神代時代(Cray-1)

- 実装
  - 4ゲートのICを高密度に実装するという画期的技術でこのような性能を実現したことは驚異
  - メモリを高速化。クロック12.5ns。スカラーも速かった。
- 自動ベクトル化コンパイラ
  - 画期的であった。でも、
  - 性能は十分でなく、使いこなすには技能が必要であった
- アプリは? 科学技術計算全般

HPC神代時代(富士通)

- FACOM 230-75 APU (1977) これまで誤解
  - 22 MFlops、2機製作、航空技術研究所に納入
  - ベクトルレジスタを持つベクトル計算機
  - AP-FORTRANという拡張言語方式
  - 間接参照や条件文のベクトル演算をサポートしていた点は注目される(CrayではXMP後半から)
  - アプリ:流体計算など?
- 商業的には成功とは言えないが、日本の最初のベクトル計算機であった。

HPC神代時代（日立）

- IAP (Integrated Array Processor)というメインフレームに対する付加プロセッサ
  - アプリ：汎用
- HITAC M-180 IAP (1978),
- M-200H IAP (1979, 48MFlops)（筆者が利用）
- M-280H IAP (1982, 67 MFlops)
  - 仮想空間上のデータに対してベクトル演算
  - 性能向上はほどほど（数倍程度）
  - 高度な自動ベクトル化コンパイラを装備
  - TSSでも使える

HPC神代時代（日立）

- 間接参照、総和、内積、1次漸化式の自動並列化
  - Cray-1がまだ完全にはサポートしていなかった機能を有していたことが特徴である。
- M-280H IAP は、世界で初めて条件付きdo loopを自動ベクトル化できた。
- キャッシュに頼ったベクトル演算には限界がありメインフレーム自体の高速化とともに姿を消した。

HPC神代時代（NEC他）

- 日本電気
  - ACOS-1000 IAP (1982, 28 Mflops)
- 三菱電機
  - MELCOM COSMO IAP
  - 詳細は不明

HPC神話時代の日本の並列計算機

- 1979：GMMCS(群馬大)、PACS-9(京大)
- 1980：PPA（北大）、KDSS-1(豊田工大）

  など1983年までに30近く（高橋義造氏による）。いずれも実験機。商品はなし。
- マイクロプロセッサの登場により可能
  - MC6800, Z80, LSI11, i8086など
  - 専用LSIを用いたものはHAL, QA2など少数
- アプリ：シミュレーション、画像、信号処理など

HPC神代時代の外国の並列計算機

- 1980：DAP（専用）, CM*(LSI-11), LAU(DF, i8085+Am2901)
- 1981：Manchester DFM（専用）
- 1983：Cosmic Cube（i8086+i8087), MPP（専用）

括弧内はCPUチップ。

ICL DAP

- 1972: paper by Dr Stewart F Reddaway
- 1974: prototype 製造開始
- 1979: Queen Mary Collegeに納入
  - 64x64 1 bit PE (SIMD)
  - アプリ：？？

Goodyear MPP

- 198?: Goodyear's array processor, a 4x256 1-bit processing element (PE) computer.
- 1983: Goodyear's MPP, a 128x128 2-D array of 1-bit PEs. 納入：NASA Goddard Space Flight Center
  - アプリ：初期は衛星画像解析、その後、開口合成レーダー、海流、宇宙線、NN,レイトレーシングなど。

神代のアプリとアーキ

- ベクトル計算機は汎用科学技術用、並列計算機は専用的計算機という位置づけ。
- マイクロプロセッサの登場により、アプリに特化したコンピュータが可能になった。
- 超並列機は、欧米では商品開発、日本では実験機のみ。
- アプリが明確なものは寿命が長い(?)
  - Cosmic Cube→ipsc→Paragon→ASCI Red
  - pacs, pax→cp-pacs, pacs-cs
  - Grape series

1st Generation (1H of 1980's)

1981 FPS-164 (64 bits)
1981 CDC Cyber 205 400MF
1982 Cray XMP-2 Steve Chen 630MF
1982 Cosmic Cube in Caltech, Alliant FX/8 delivered, HEP installed
1983 HITAC S-810/20 630MF
1983 FACOM VP-200 570MF
1983 Encore, Sequent and TMC founded, ETA span off from CDC

1st Generation (1H of 1980's)
(continued)

1984 Multiflow founded
1984 Cray XMP-4 1260MF
1984 PAX-64J completed (Tsukuba)
1985 NEC SX-2 1300MF
1985 FPS-264
1985 Convex C1
1985 Cray-2 1952MF
1985 Intel iPSC/1, T414, NCUBE/1, Stellar, Ardent…
1985 FACOM VP-400 1140MF
1986 CM-1 shipped, FPS T-series (max 1TF!!)

専用シミュレータ（この頃から）

- この頃からASICが利用可能になった。
- Ising模型シミュレータ：米UCSB(1982)、オランダDelft工科大学(1983)、東大(1988)など
- 格子ゲージ模型シミュレータ：イタリアのAPE(1986-)、筑波大学のqcdpax(1989)とcp-pacs(1996)、アメリカのGF11(1991), ACP-MAPS, QCDSP(1995)、QCDOC(2002)など
- 古典粒子系シミュレータ：Delft工科大学のDMDP(1988)、東大のGRAPE(1989～)、理研のMD-GRAPE (1999)、富士ゼロックスのMD Engine (1996)など
- モンテカルロ専用機：Monte-4(1993原研、vector型）

専用から汎用へ

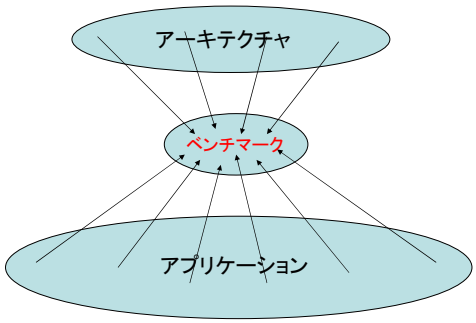
- 汎用指向の格子ゲージシミュレータ
  - cp-pacs→SR2201/SR8000
  - QCDSP, QCDOC→BG/L
  - APE, GF11（いずれもSIMD)は汎用化せず
- その他の例（果たして専用だったのか？）
  - NWT(数値風洞）→VPP500/300/700
  - 地球シミュレータ→SX-6/7/8

1990年代のHPC

- IBMとCrayの超並列参入
  - 1993　Cray T3D (Alpha chip)
  - 1993　IBM SP-1 (Power chip)
- ベンチャー超並列企業の倒産・吸収
  - 1995　Convex, KSR, nCUBE, Pyramid
  - 1996　MasPar, TMC
- アプリケーションとアーキテクチャの関係？？
  - 倒産の原因は？アプリが不明確だったから？？

性能とは

- コンピュータの性能評価：ベンチマーク



ベンチマーク

- Mix, MIPSからLFK, Linpack, Euroben, SPEC, HPCC, ….
- アプリケーションから見れば：自分のアプリケーションの計算速度を推定するためのデータ
- アーキテクチャから見れば：アーキテクチャをチューニングする時の目標
- ベンチマークの選択：コンピュータの利用目的の選択。

コンピュータの設計戦略

- 設計：有限な資源の中での選択
  「あれもほしい」「これもほしい」ではだめ
- 「汎用」の陥穽（おとしあな）：汎用といっても、暗黙のアプリのクラスを前提にしている。無前提の「汎用」はない。
- 多くの成功したHPCは、特定のアプリを前提に設計されている。結果的に多くのアプリでもよい性能を出すことが多い。