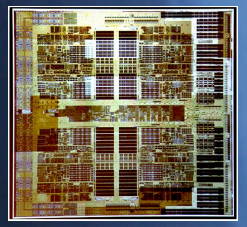
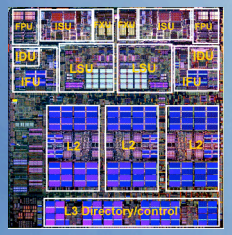
RICE COMPUTER SCIENCE
GEORGE R. BROWN SCHOOL OF ENGINEERING

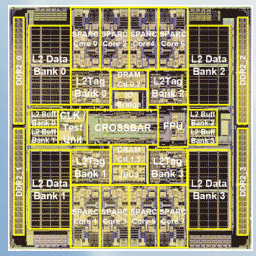
Portable Parallel Programming for Heterogeneous Multicore Computing

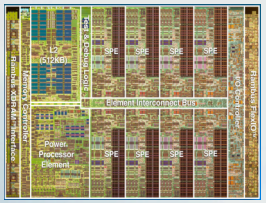
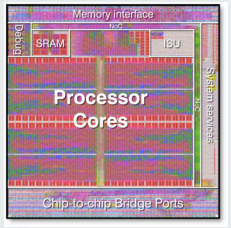
Vivek Sarkar

Parallel Computing Group, Rice University

vsarkar@rice.edu



Acknowledgments

- X10 project (x10.sf.net)
- Rice COMP 635 Seminar on Heterogeneous Processors (www.cs.rice.edu/~vsarkar/comp635)
- Rice Habanero Multicore Software project
  - Overview presentations on Habanero in Rice/GCAS booth #789
    - Tuesday, 12:00 - 12:30
    - Wednesday, 1:00 - 1:30
- Georgia Tech ECE 6100 course (Module 14), Multi-core Case Studies (Vince Mooney, Krishna Palem, Sudhakar Yalamanchili)
- UIUC ECE 498 AL1 course, Programming Massively Parallel Processor (David Kirk, Wen-mei Hwu)
- MIT 6.189 IAP 2007 course, Introduction to the Cell Processor (Michael Perrone)
- Overview slides on Clearspeed CSX600 (Simon McIntosh-Smith)

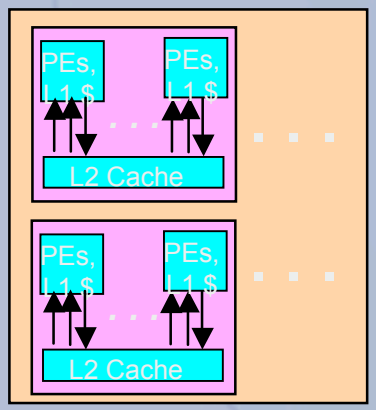
Outline

- Heterogeneous Processors
- X10 language
- Habanero Multicore Software project

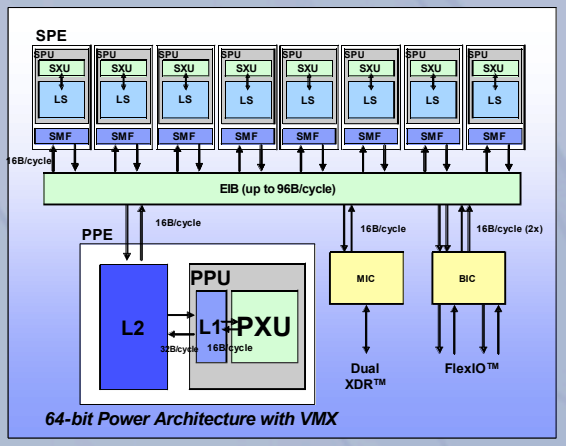
Future System Trends: a new Era of Mainstream & High End Parallel Processing

Hardware building blocks for mainstream *and* high-performance systems are varied and proliferating …

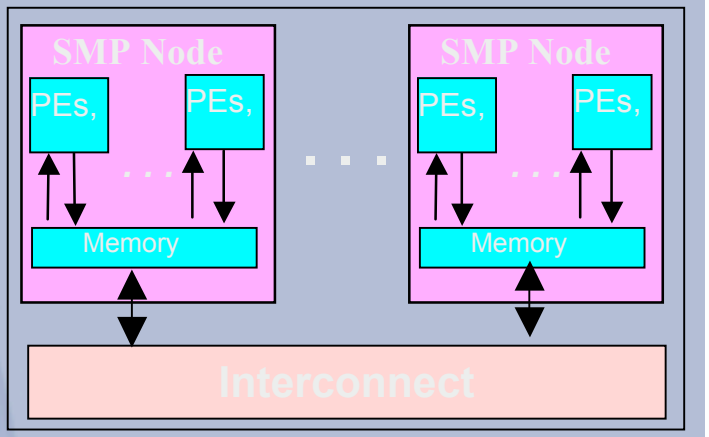
Homogeneous Multicore



Heterogeneous Multicore



High Performance Clusters

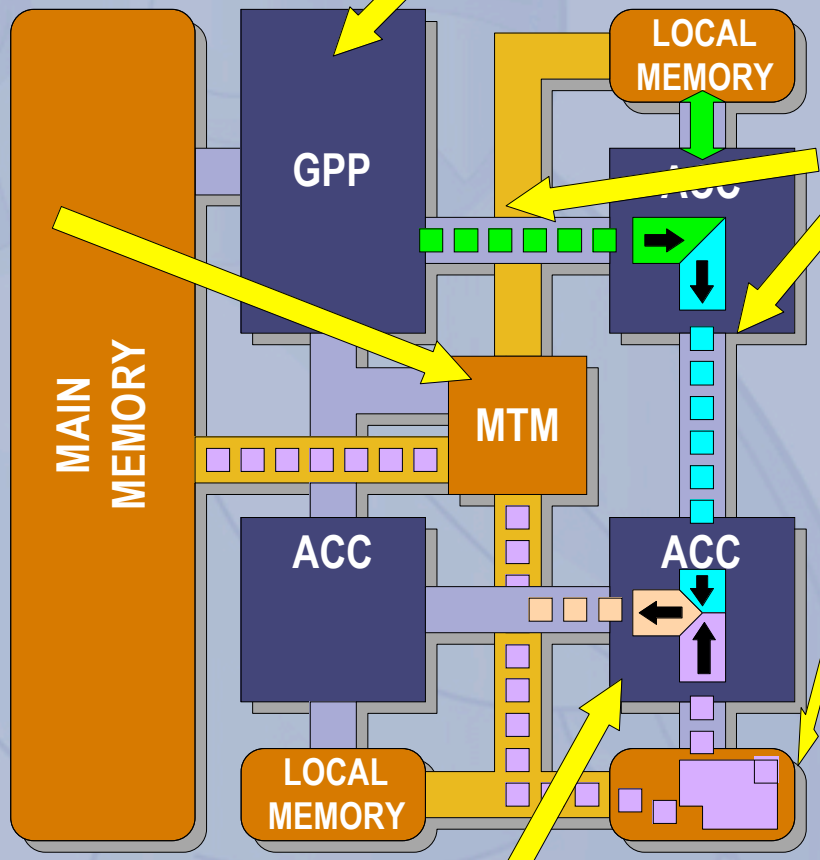


<u>Challenge:</u> Develop new programming technologies to support portable parallel abstractions for future hardware

Heterogeneous Processors

General-purpose processor orchestrates activity

Memory transfer module schedules system-wide bulk data movement



Accelerators can use scheduled, streaming communication…

or can operate on locally-buffered data pushed to them in advance

Accelerated activities and associated private data are localized for bandwidth, power, efficiency

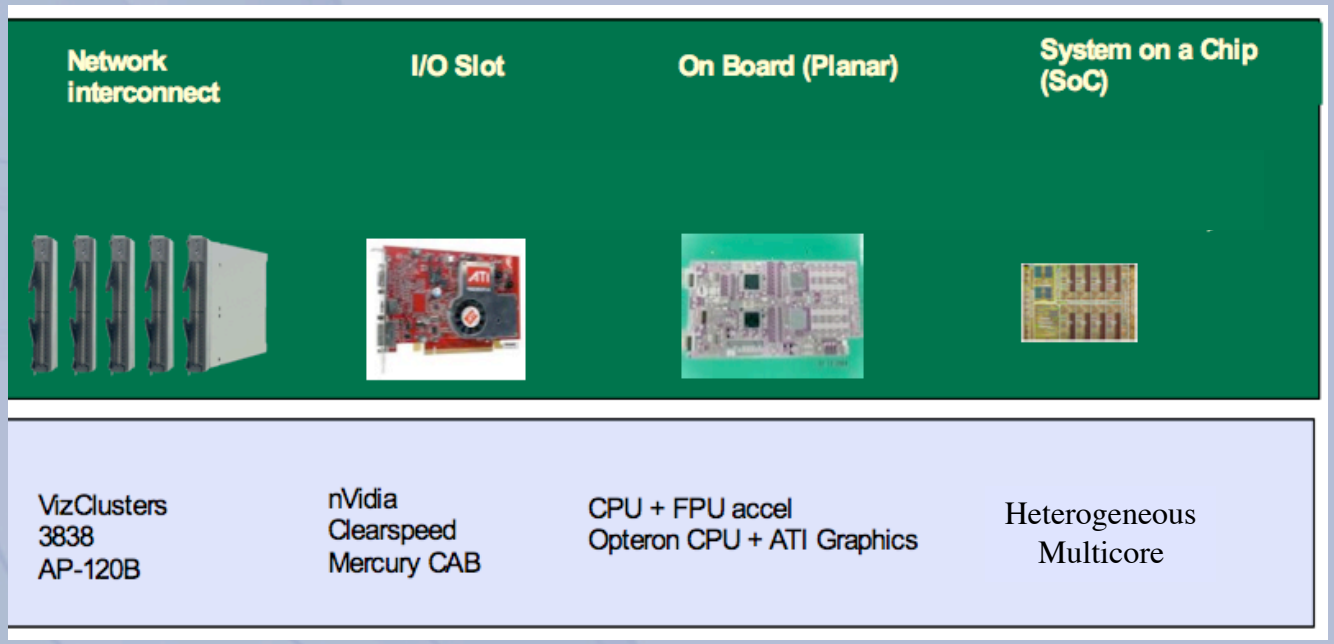
Motivation:

1) Different parts of programs have different requirements
   - Control-intensive portions need good branch predictors, speculation, big caches to achieve good performance
   - Data-processing portions need lots of ALUs, have simpler control flows

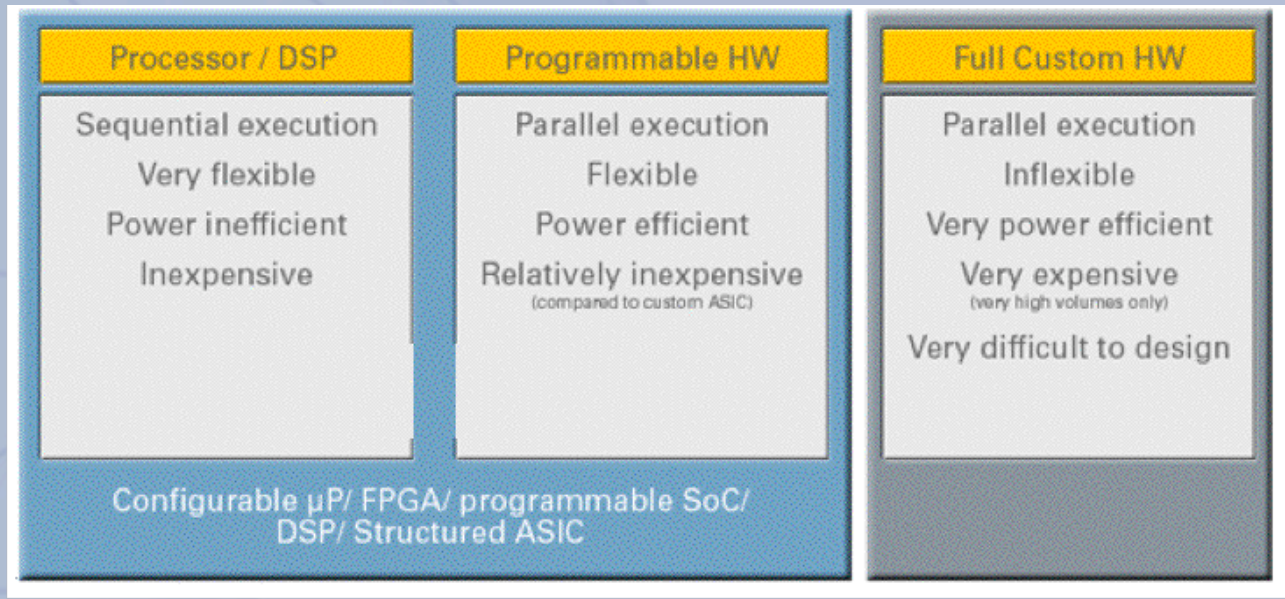
2) Power consumption
   - Features like branch prediction, out-of-order execution, tend to have very high power/performance ratios.
   - Applications often have time-varying performance requirements

Heterogeneous Processor Spectrum

<u>Dimension 1:</u>
Distance of accelerator from main processor (decreasing latency, increasing bandwidth)



<u>Dimension 2:</u>
Hardware customization in accelerator (decreasing energy per operation)

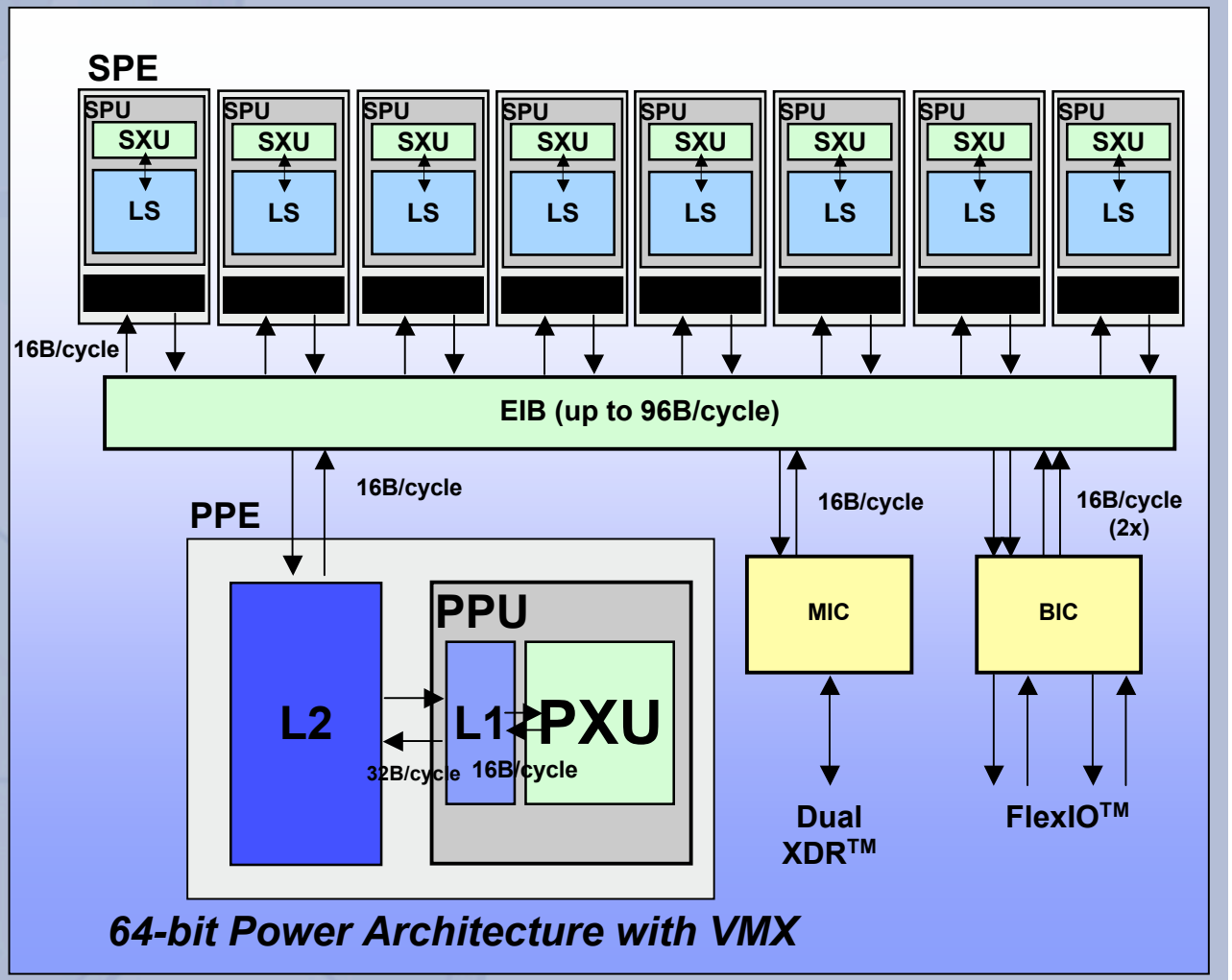


Sample Application Domains for Heterogeneous Processors

- Cell Processor
  - Medical imaging, Drug discovery, Reservoir modeling, Seismic analysis, …
- GPGPU (e.g., Nvidia G80)
  - Graphics, Computer-aided design (CAD), Digital content creation (DCC), emerging HPC applications, …
- FPGA (e.g., Xilinx DRC)
  - HPC, Petroleum, Financial, …
- HPC accelerators (e.g., Clearspeed)
  - HPC, Network processing, Graphics, …
- Stream Processors (e.g., Imagine)
  - Image processing, Signal processing, Video, Graphics, …
- Others
  - TCP/IP offload, Crypto, …

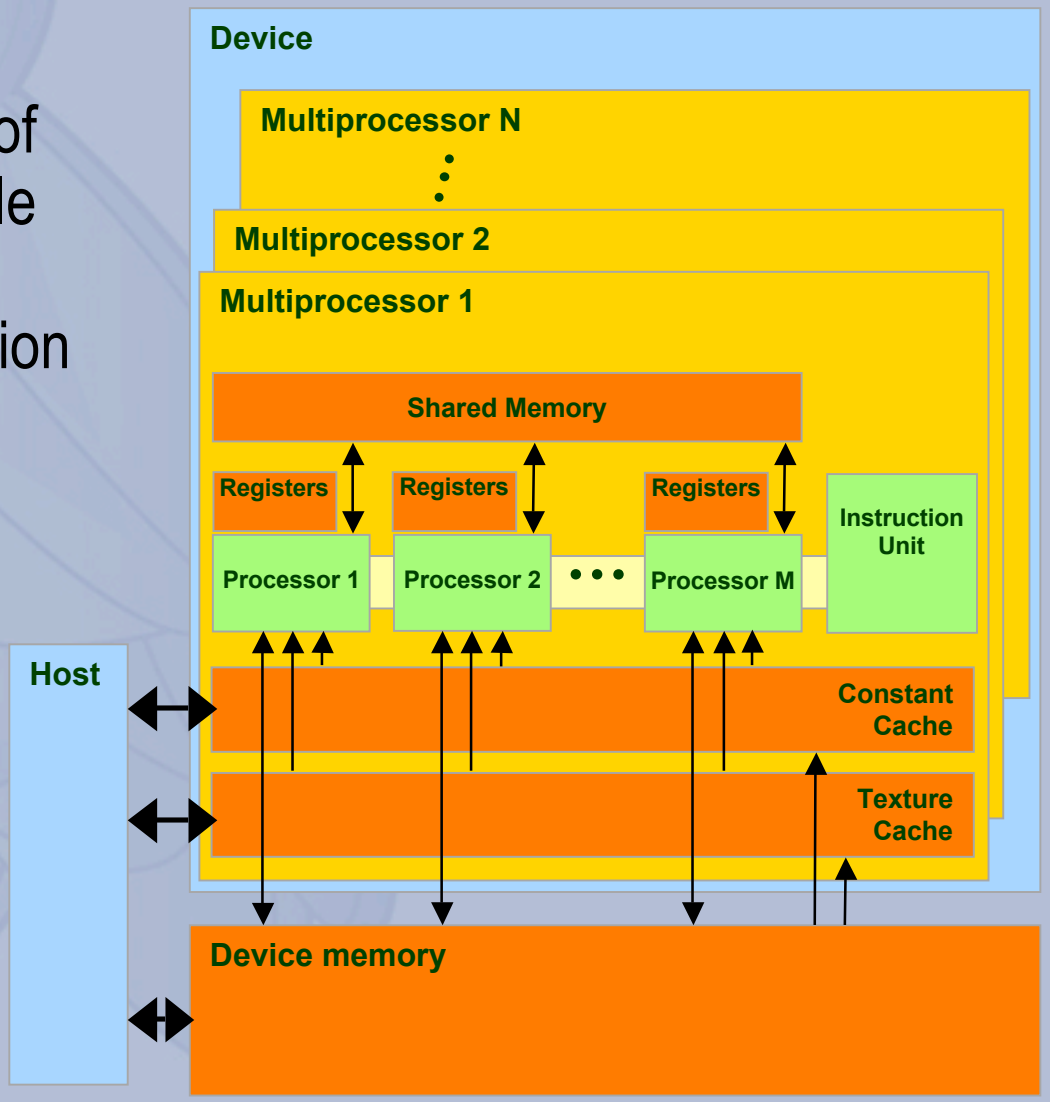
Cell Broadband Engine

- Heterogeneous multicore system architecture
  - Power Processor Element for control tasks
  - Synergistic Processor Elements for data-intensive processing
- Synergistic Processor Element (SPE) consists of
  - Synergistic Processor Unit (SPU)
  - Synergistic Memory Flow Control (MFC)
    - Data movement and synchronization
    - Interface to high-performance Element Interconnect Bus

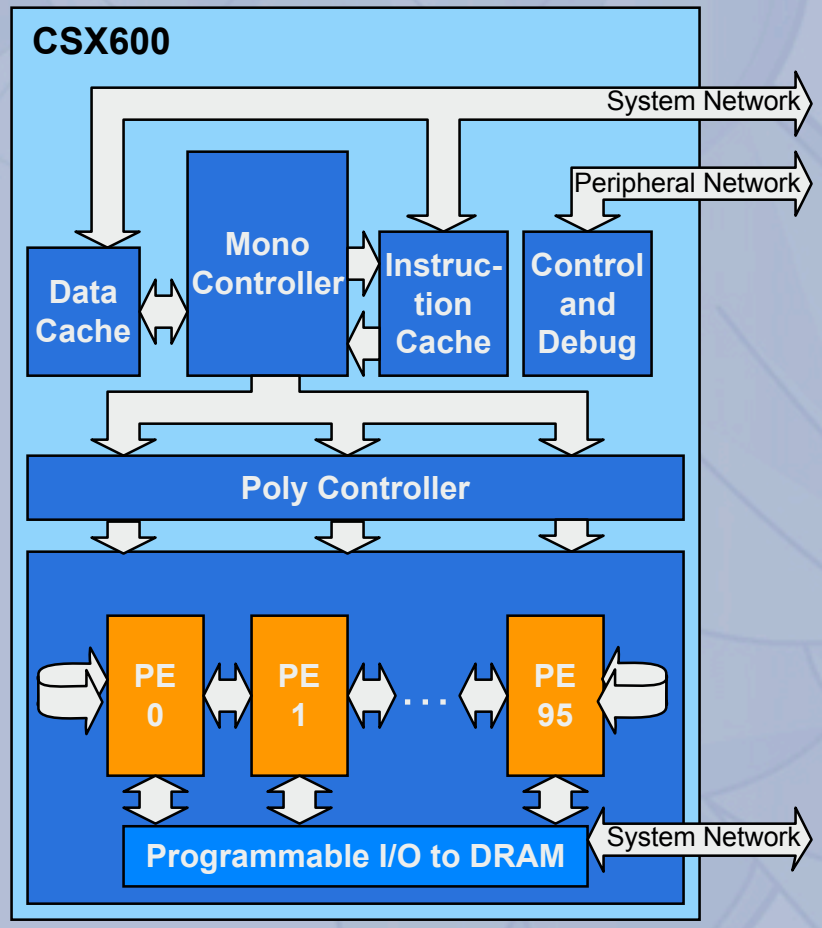


Nvidia GeForce 8800 GTX

- The device is a set of 16 multiprocessors
- Each multiprocessor is a set of 32-bit processors with a Single Instruction Multiple Data architecture – shared instruction unit
- Each multiprocessor has:
  - 32 32-bit registers per processor
  - 16KB on-chip shared memory per multiprocessor
  - A read-only constant cache
  - A read-only texture cache



Clearspeed MTAP processor core



- **Multi-Threaded Array Processing**
  - Hardware multi-threading
  - Asynchronous, overlapped I/O
  - Run-time extensible instruction set
- **Array of 96 Processor Elements (PEs)**
  - Each is a Very Long Instruction Word (VLIW) core, not just an ALU
  - Coarse-grained data parallel processing
- **Cn is the natural language**
  - Single “poly” data type modifier
  - Rich expressive semantics

Comparison of Parallelism Levels in Example Accelerators

	Cell BE	**Nvidia G80**	**ClearSpeed CSX600**
Board-level	2 Cell BE chips/board	1 G80 chip + 1 NVIO chip	2 CSX600 chips/board
Chip-level MIMD	1 PPE + 8 SPE's	16 multiprocessors (MPs)	1 mono controller + 1 poly controller
Chip-level SIMD	128-bit native SIMD on SPE & 128-bit VMX on PPE	8 SIMD "stream processors" per MP = 128 procs/chip	96 SIMD PE's per poly controller
Thread-level	2 PPE threads + 8 SPE threads	32 threads per MP = 512 threads/chip	8 mono threads
Instruction-level	Dual-issue SPE pipelines		4-stage floating point add & multiply units + ILP among mono, poly, I/O units

Other Hardware Characteristics

	Cell BE	**Nvidia G80**	**ClearSpeed CSX600**
32-bit FP	200+ GFLOPS	360+ GFLOPS	25+ GFLOPS
64-bit FP	20+ GFLOPS		25+ GFLOPS
Clock frequency	3.2 GHz	575 MHz	210 MHz
Transistors/ chip	~ 241M	~ 681M	~ 128M
Power	~ 110 Watts	~ 145 W (for GeForce 8800 GTX board)	~ 10W

Sample Cell PPE code

(from http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/dateien/matmul.tar.gz)

```
/* Create an SPE group */
 gid = spe_create_group (SCHED_OTHER, 0, 1);
 if ((gid == NULL) || (spe_group_max (gid) < 1)) exit(1);

 /* allocate the SPE tasks */
 for (i = 0; i < num_spes; i++) {
   speid[i] = spe_create_thread (gid, &matmul_spu, (unsigned long
  long *) &cb[i], NULL, -1, 0);
   if (speid[i]== NULL) print_error("FAILED: spe_create_thread");
 }

 /* wait for a synchronisation signal of each SPE */
 for (i = 0; i < num_spes; i++) {
   while (!spe_stat_out_mbox(speid[i]));
   spe_read_out_mbox(speid[i]);
 }

 printf("Starting SPE calculations... ");
 fflush(stdout);
```

Sample Cell PPE code (contd.)

(from http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/dateien/matmul.tar.gz)

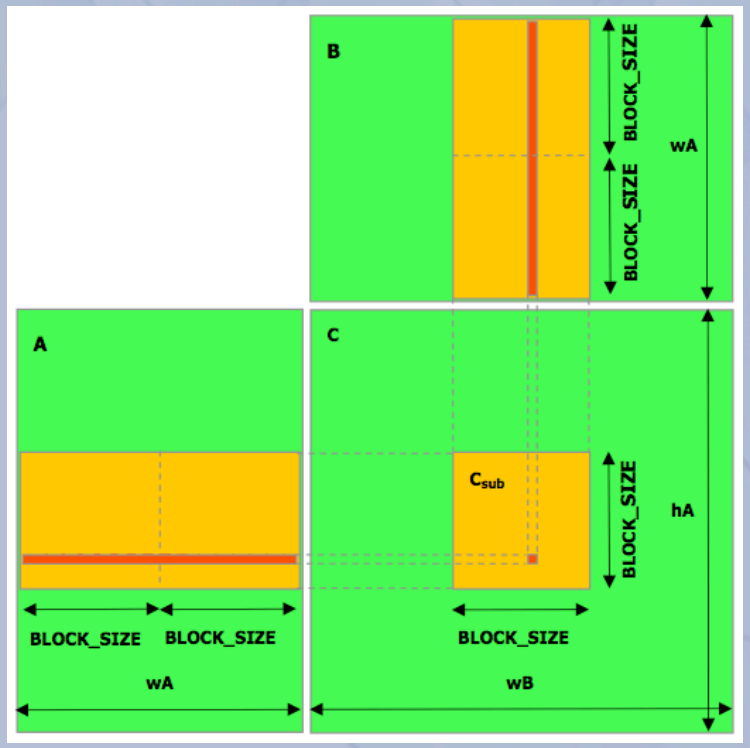
```
/* start PPE-side measurement of the execution time */
t_start = my_gettimeofday();

/* send a start signal to each SPE */
for (i = 0; i < num_spes; i++) spe_write_in_mbox(speid[i], 0);

/* get the performance data of each SPE */
for (i = 0; i < num_spes; i++) {
  while (!spe_stat_out_mbox(speid[i]));
  spe_time[i] = spe_read_out_mbox(speid[i]);
  while (!spe_stat_out_mbox(speid[i]));
  spe_count[i] = spe_read_out_mbox(speid[i]);
}

/* stop PPE-side measurement of execution time */
t_all = my_gettimeofday() - t_start;

/* wait until all SPE threads have finished */
for (i = 0; i < num_spes; i++) spe_wait(speid[i], &status, 0);
```

Sample Cell SPE code

(from http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/dateien/matmul.tar.gz)

```
. . .
    wait_for_DMA(1);
    matmul_SIMD64(matrix_A_a, matrix_B_a, matrix_C_a);          /*
  mul a a a */
    load_data(PPE_matrix_A_Ptr, matrix_A_b, dma_list1b,
  offset_1_b_m, offset_1_b_n + cb.p_blocks - 1, 3);
    store_data(matrix_C_a, dma_list4a, offset_3_a_m, offset_3_a_n,
  4);
    wait_for_DMA(2);
    matmul_SIMD64(matrix_A_a, matrix_B_b, matrix_C_b);          /*
  mul a b b */
    store_data(matrix_C_b, dma_list4b, offset_3_b_m, offset_3_b_n,
  4);
    wait_for_DMA(3);
    matmul_SIMD64(matrix_A_b, matrix_B_a, matrix_C_c);          /*
  mul b a c */
    store_data(matrix_C_c, dma_list4c, offset_3_c_m, offset_3_c_n,
  4);
    matmul_SIMD64(matrix_A_b, matrix_B_b, matrix_C_d);          /*
  mul b b d */
    store_data(matrix_C_d, dma_list4d, offset_3_d_m, offset_3_d_n,
  4);
    wait_for_DMA(4);
```

Matrix Multiply Example in CUDA (Host Code)



Source: NVIDIA Compute Unified Device Architecture Programming Guide Version 1.0

```
void Mul(const float* A, const float* B, int hA, int wA, int wB,
         float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
```

Matrix Multiply Example in CUDA (Device Code)

```
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;
```

```
    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

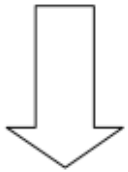
ClearSpeed: Host Program

Here is a simple example of an inner loop of a host program which passes data to the co-processor. For simplicity, status checking has been omitted from this example.

```
// obtain the address of the shared memory represented by the symbol
_SHARED_MEMORY_
CSAPI_get_symbol_value("/home/mh/simple.csx", "_SHARED_MEMORY_",
&shared_memory );

while ( ... there is data to process ... ) {
   // prepare the data
   pl.a = 1; pl.b = 2; ...

   // copy the data onto the card side
   CSAPI_write_mono_memory(state, shared_memory, sizeof(MyPayload),
(void*)(&pl));

   // signal the start semaphore
   CSAPI_signal(state, _SEM_DATA_IN_READY_);

   // wait for results
   // in case of a multi-threaded applications
   // something useful could be done during the wait
   // for single-threaded applications, polling could be used
   CSAPI_wait(state, _SEM_PROCESSING_COMPLETE_);

   // consume the data after processing, assuming here that the
   // shared memory is used both for receiving and sending data
   CSAPI_read_mono_memory(state, shared_memory, sizeof(MyPayload),
(void*)(&pl));
}
```

ClearSpeed: DAXPY example in Cn

```
void daxpy(double *c, double *a, double alpha, uint N) {
  uint i;
  for (i=0; i<N; i++)
    c[i] = c[i] + a[i]*alpha;
}
```



```
void daxpy(double *c, double *a, double alpha, uint N) {
  uint i;
  poly double cp, ap;
  poly int pe_num=get_penum();
  for (i=0; i<N; i+= get_numpes()) {
    memcpym2p(&cp, &c[i+pe_num], sizeof(double));
    memcpym2p(&ap, &a[i+pe_num], sizeof(double));
    cp = cp + ap*alpha;
    memcpyp2m(&c[i+pe_num], &cp, sizeof(double))
}
```

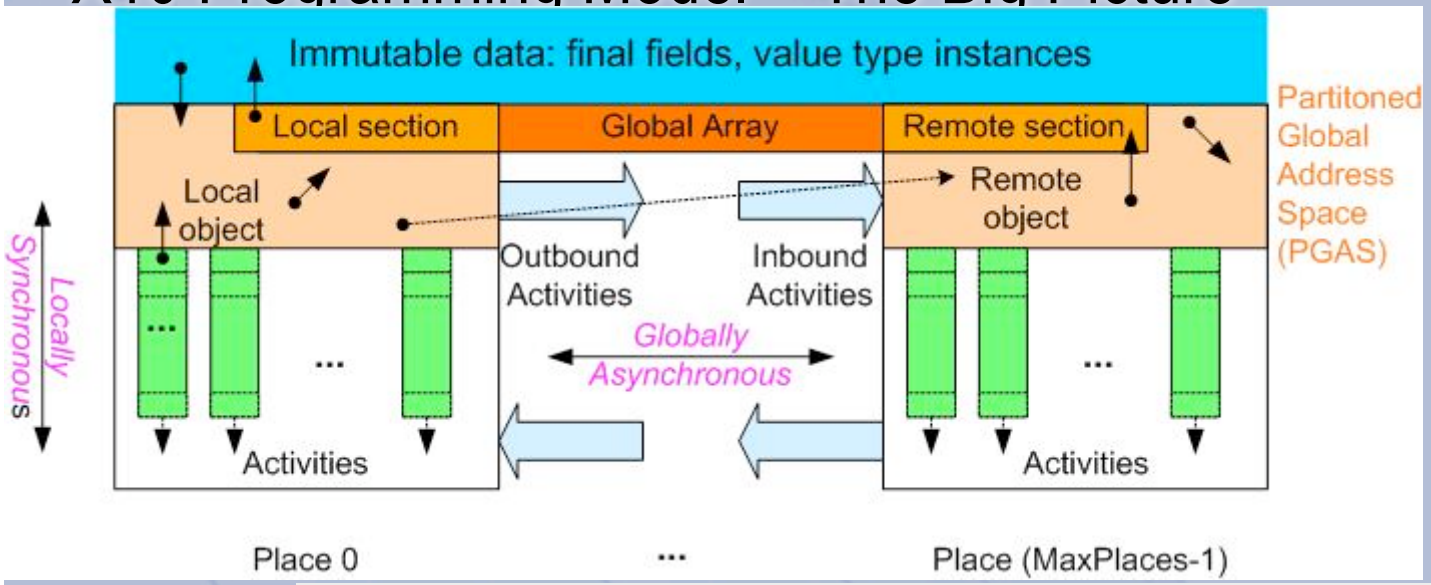
Outline

- Heterogeneous Processors
- X10 language
- Habanero Multicore Software project

X10 Approach

- Unified abstractions of asynchrony and concurrency for use in
  - Multi-core SMP Parallelism
  - Messaging and Cluster Parallelism
- Productivity
  - High Level Language designed for portability and safety
  - X10 Development Toolkit for Eclipse
- Performance
  - Extend VM+JIT model for high performance
    - 80% of new code is written for execution on VMs and managed runtimes
  - Performance transparency – don't lock out the performance expert!
    - expert programmer should have controls to tune optimizations and tailor distributions & communications to actual deployment
- Build on sequential subset of Java language
  - Retain core values of Java --- productivity, ubiquity, maturity, security
  - Target adoption by mainstream developers with Java/C/C++ skills
  - Efficient foreign function interfaces for libraries written in Fortran and C/C++
- Reference: "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing", P.Charles et al, OOPSLA 2005 Onward! track.

X10 Programming Model – The Big Picture



Storage classes:

- Activity-local
- Place-local
- Partitioned global
- Immutable

- Dynamic parallelism with a *Partitioned Global Address Space*
- *Places* encapsulate binding of activities and globally addressable data
  - Number of places currently fixed at launch time
- All concurrency is expressed as *asynchronous activities* – subsumes threads, structured parallelism, messaging, DMA transfers, etc.
- *Atomic sections* enforce mutual exclusion of co-located data
  - No place-remote accesses permitted in atomic section
- *Immutable* data offers opportunity for single-assignment parallelism

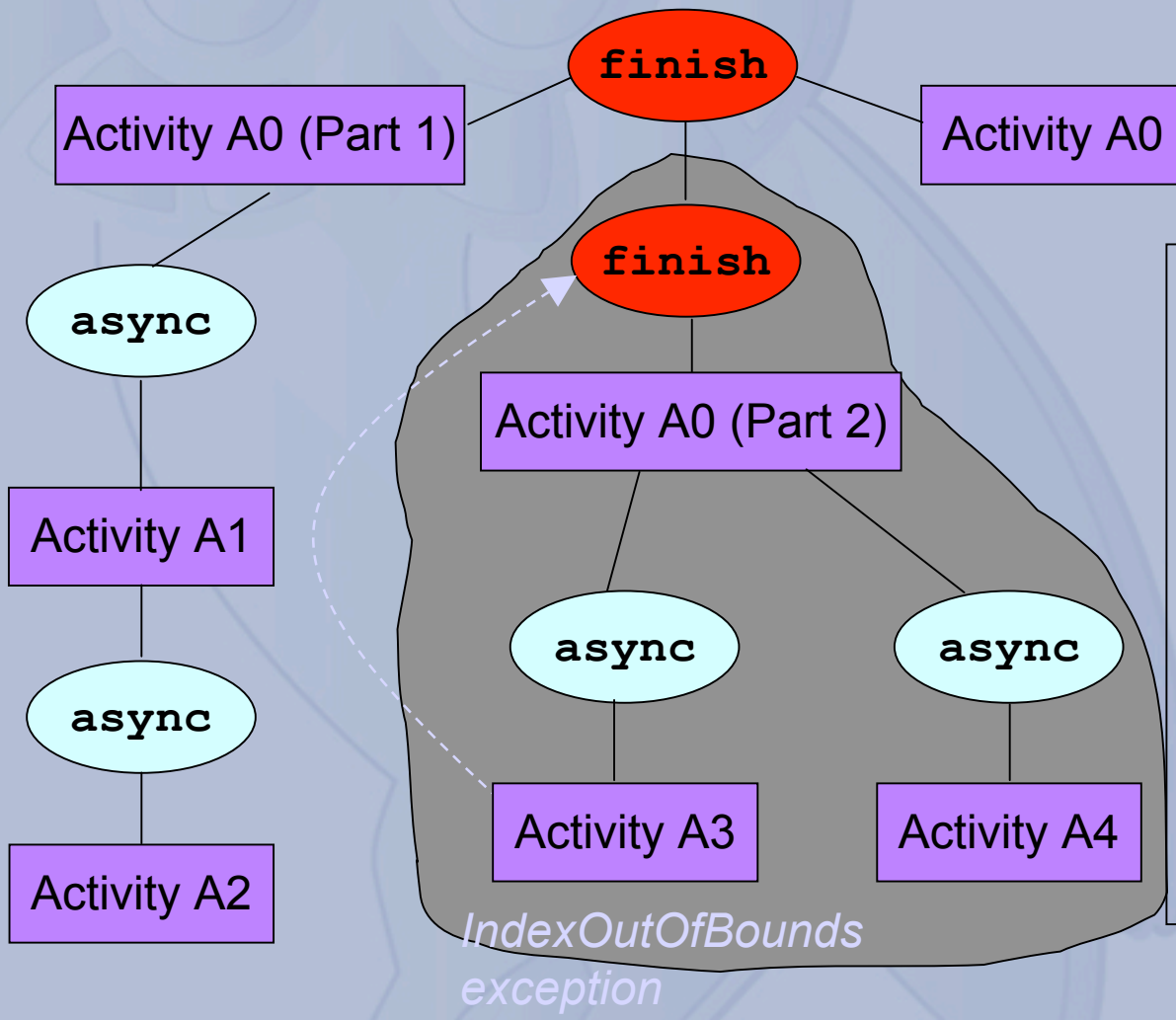
X10 Language Summary

- async [(*Place*)] [clocked(c…)] *Stm*
  - Run Stm asynchronously at Place
- finish *Stm*
  - Execute s, wait for all asyncs to terminate (generalizes join)
- foreach (point *P* : *Reg*) *Stm*
  - Run Stm asynchronously for each point in region
- ateach (point *P* : *Dist*) *Stm*
  - Run Stm asynchronously for each point in dist, in its place.
- atomic *Stm*
  - Execute Stm atomically
- new T
  - Allocate object at this place (here)
- new T[d] / new T value [d]
  - Array of base type T and distribution d
- Region
  - Collection of index points, e.g.

    region r = [1:N,1:M];
- Distribution
  - Mapping from region to places, e.g.
    - dist d = block(r);
- next
  - suspend till all clocks that the current activity is registered with can advance
  - Clocks are a generalization of barriers and MPI communicators
- future [(*Place*)] [clocked(c…)] *Expr*
  - Compute Expr asynchronously at Place
- F. force()
  - Block until future F has been computed
- extern
  - Lightweight interface to native code

Deadlock safety: any X10 program written with async, atomic, finish, foreach, ateach, and clocks can never deadlock

Asynchronous Activities -- the first step in Parallel Programming




```
// X10 pseudo code
main(){ // implicit finish
  Activity A0 (Part 1);
  async {A1; async A2;}
  try {
    finish {
      Activity A0 (Part 2);
      async A3;
      async A4;
    }
  catch (…) { … }
  Activity A0 (Part 3);
}
```

Atomic blocks --- the Second Step in Parallel Programming

An atomic block is *conceptually* executed in a single step

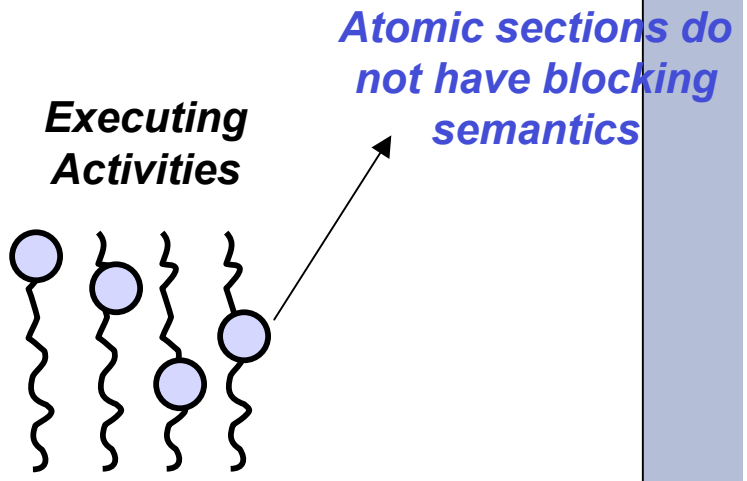
- **Result is equivalent to that of suspending all other activities during execution of atomic block**
- **Atomic block may not include blocking operations (force, next)**
- **Programmer does not manage any locks explicitly**

Example 1: **Read-modify-write operation on an array element**

```
atomic Table[j] ^= k ;
```

Example 2: **Insertion in a linked list**

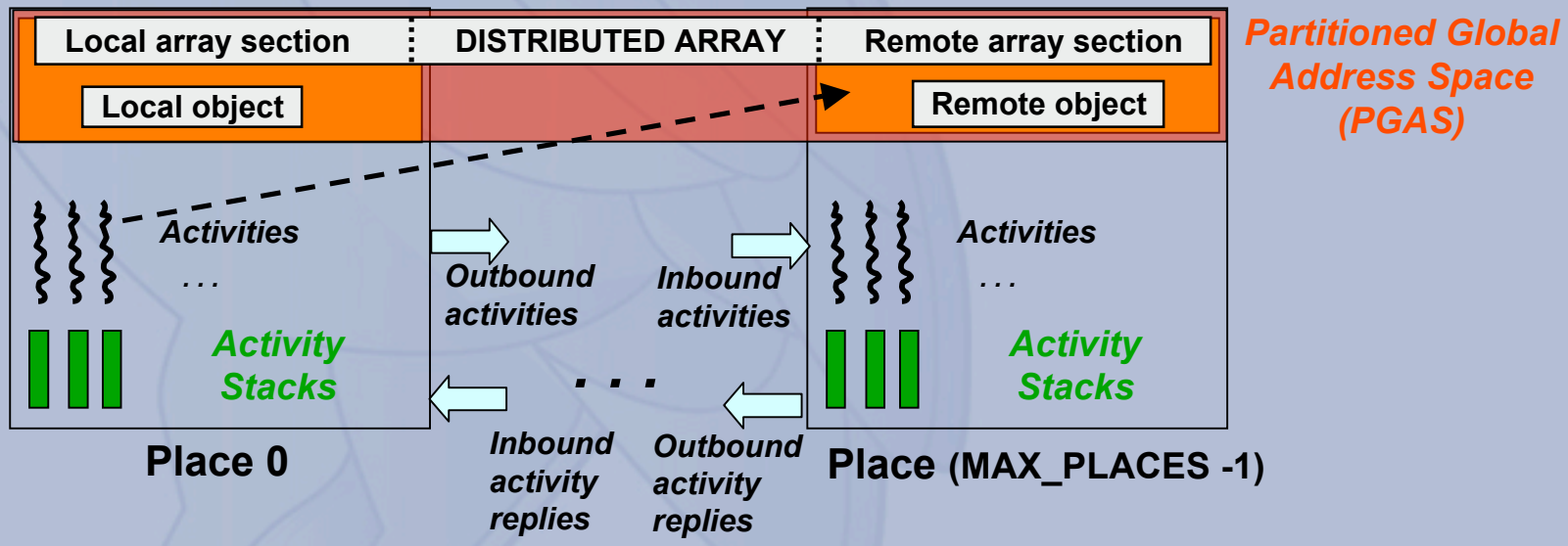
```
node = new Node(data);
atomic {
  node.next = head;
  head = node;
}
```



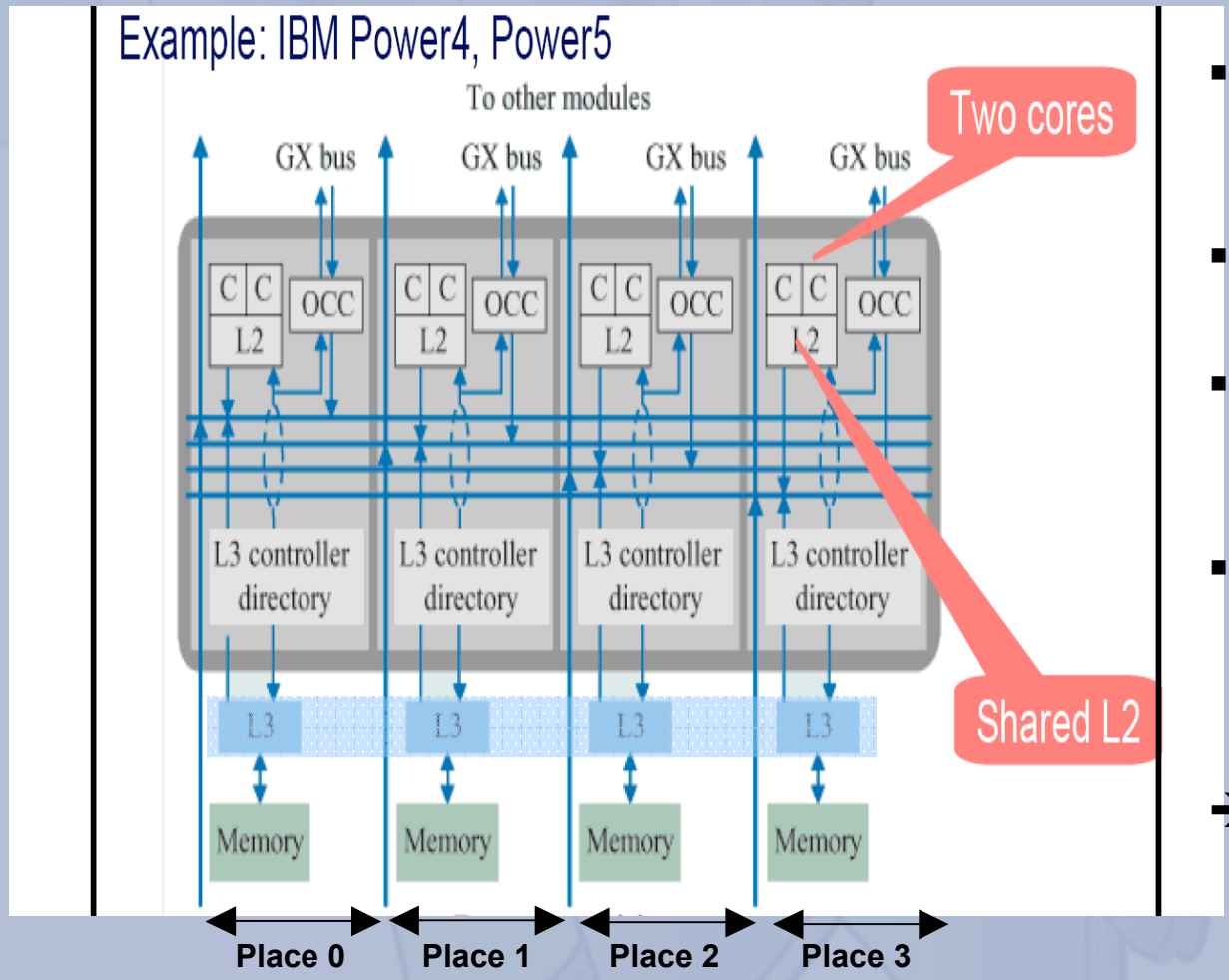
Places --- the Third Step in Parallel Programming

Question: how should an activity access remote data?

Answer: by implicitly or explicitly creating an activity at the remote *place*



X10 Deployment on a Multicore SMP



- Basic Approach -- partition X10 heap into multiple place-local heaps
- Each X10 object is allocated in a designated place
- Each X10 activity is created (and pinned) at a designated place
- Allow an X10 activity to synchronously access data at remote places outside of atomic sections

➔ Thus, places serve as affinity hints for intra-SMP locality

X10 Places (contd.)

Examples

```
1) finish { // Inter-place parallelism
       final int x = … , y = … ;
       async (a) a.foo(x); // Execute at a's place
       async (b) b.bar(y); // Execute at b's place
   }

2) // Implicit and explicit versions of remote fetch-and-
   add
   a) a.x += b.y ;
   b) async (b) {
         final int v = b.y;
         async (a) atomic a.x += v;
      }
```

Examples of X10 Arrays, Points, Regions, Distributions

```
// A is a local 1-D array, B is a distributed 2-D array
int[.] A = new int[[0:N-1]];
int[.] B = new int[dist.blockRows([0:M-1,0:N-1])];
. . .
// serial pointwise for loop
for (point[j] : [1:N-1]) A[j] = f(A[j-1]);
. . .
// intra-place pointwise parallel loop
foreach (point[j] : A.region) A[j] = g(A[j]);
. . .
// inter-place pointwise parallel loop
ateach (point[i,j] : B.distribution) B[i,j] = h(B[i,j]);
. . .
// Rank-independent version of previous loop
ateach (point p : B.distribution) B[p] = h(B[p]);
```

X10 Parallelism = Activities + Atomic + Places

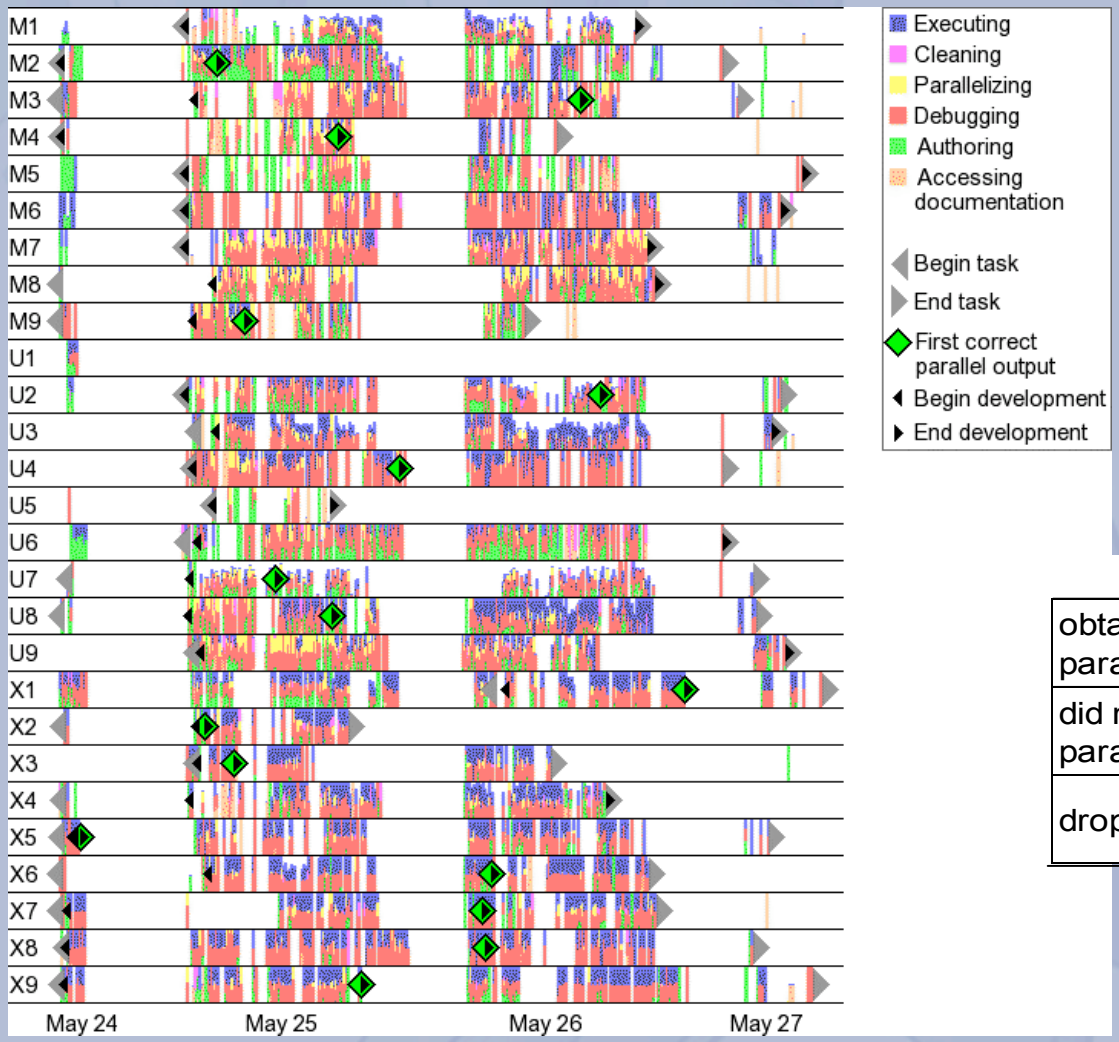
Advantages:

1. Any program written with atomic, async, finish, foreach, ateach, and clock parallel constructs *will never deadlock*
2. Inter-node and intra-node parallelism integrated in a single model
3. Remote activity invocation subsumes one-sided data transfer, remote atomic operations, active messages, . . .
4. Finish subsumes point-to-point and team synchronization
5. All remote data accesses are performed as activities --- rules for ordering of remote accesses simply follows rules for activity synchronization

Human Productivity Study (Comparison of MPI, UPC, X10)

- Goals
  - Contrast productivity of X10, UPC, and MPI for a statistically significant subject sample on a programming task relevant to HPCS Mission Partners
  - Validate the PERCS Productivity Methodology to obtain quantitative results that, given specific populations and computational domains, will be of immediate and direct relevance to HPCS.
- Overview
  - 4.5 days: May 23-27, 2005 at the Pittsburgh Supercomputing Center (PSC)
  - Pool of 27 comparable student subjects
  - Programming task: Parallelizing the alignment portion of Smith-Waterman algorithm (SSCA#1)
  - 3 language programming model combinations (X10, UPC, or C + MPI)
  - Equal environment as near as possible (e.g. pick of 3 editors, simple println stmts for debugging)
  - Provided expert training and support for each language
- References (3 papers in P-PHEC 2006, www.research.ibm.com/arl/pphec/PPHEC2006-Proceedings-FINAL.pdf)
  - "The Value Derived from the Observational Component in an Integrated Methodology for the Study of HPC Programmer Productivity", C.Danis, C.Halverson.
  - "An Experiment in Measuring the Productivity of Three Parallel Programming Languages", K.Ebcioglu, V.Sarkar, T.El-Ghazawi, J.Urbanic.
  - "The SUMS Methodology for Understanding Productivity: Validation Through a Case Study Applying X10, UPC, and MPI to SSCA#1", N.Nystrom, D.Weisser, J.Urbanic.

Data Summary

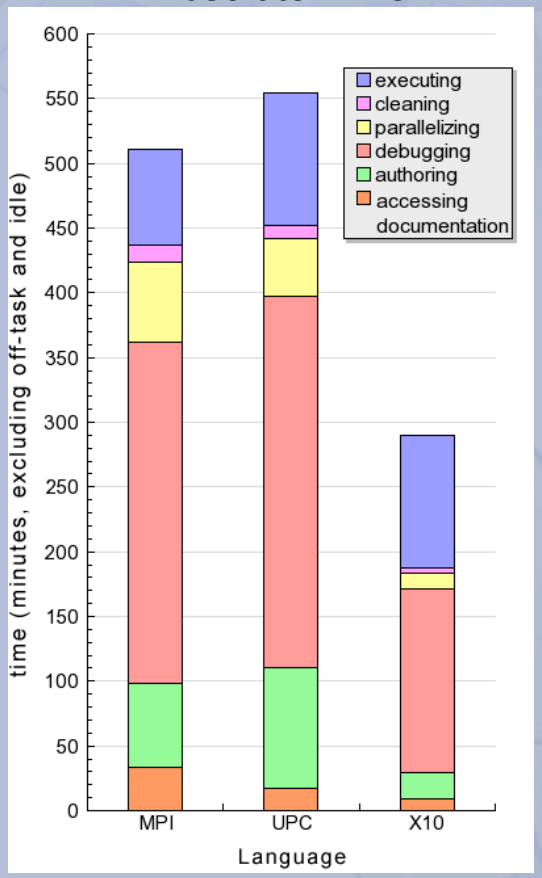


- Each thin vertical bar depicts 5 minutes of development time, colored by the distribution of activities within the interval.
- Development milestones bound intervals for statistical analysis:
  - begin/end task
  - begin/end development
  - first correct parallel output

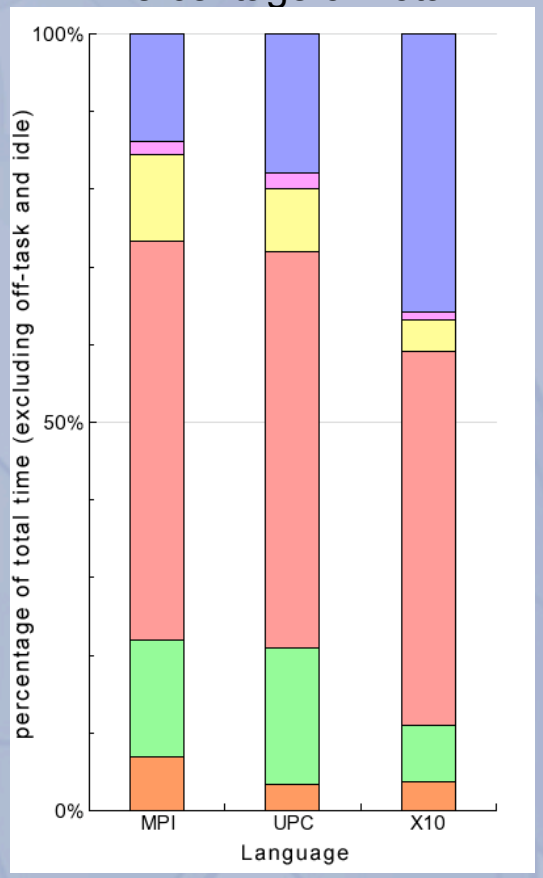
	MPI	UPC	X10
obtained correct parallel output	4	4	8
did not obtain correct parallel output	5	3	1
dropped out	0	2	0

Average Development Time by Language

Absolute Time



Percentage of Total



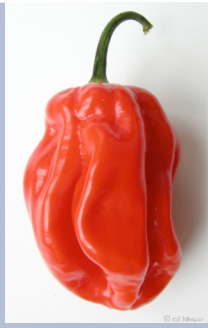
Comparing average development times between languages, several observations are clear:

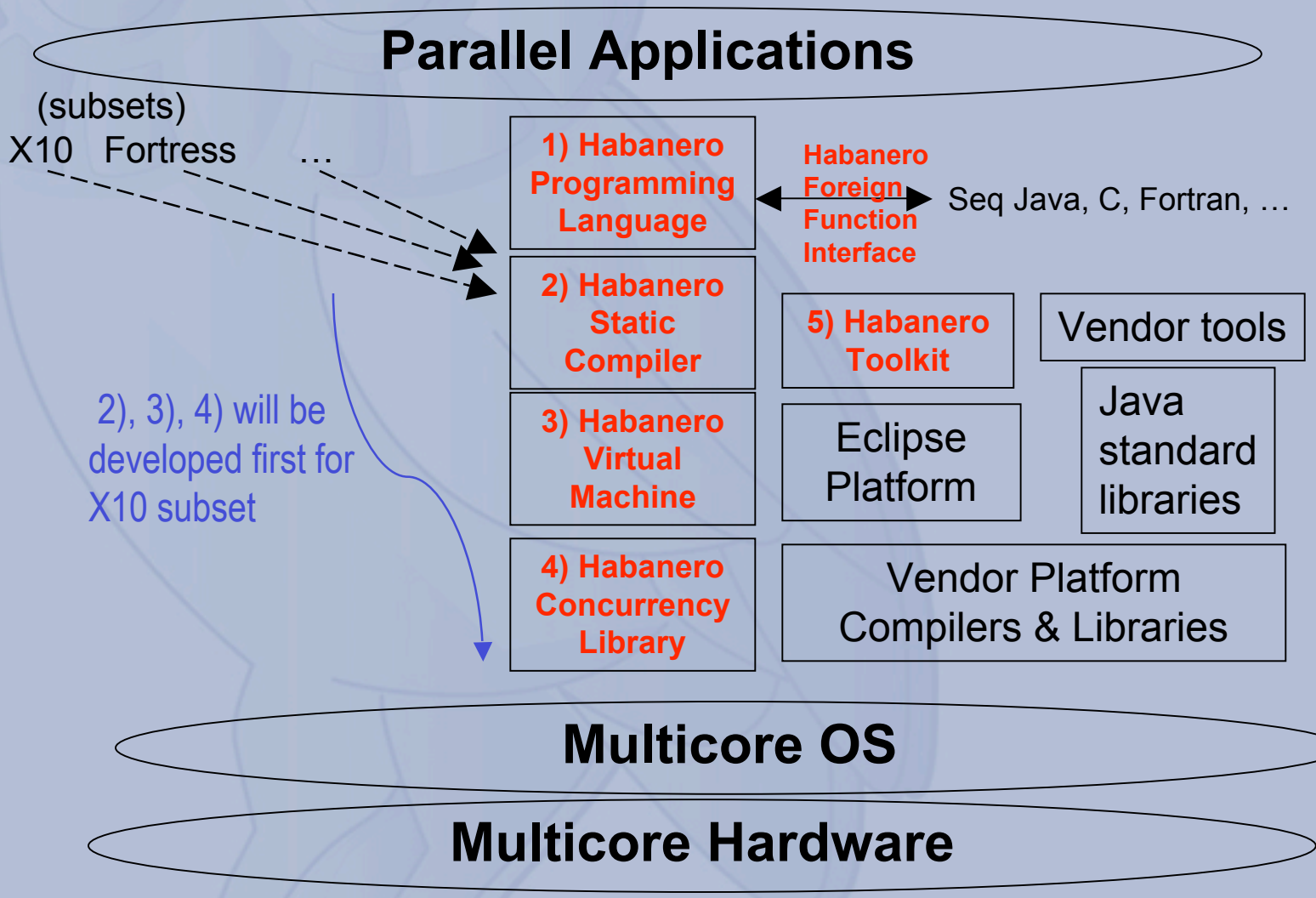
- Average development time for subjects using X10 was significantly lower than that for subjects using UPC and MPI.
- The relative time debugging was approximately the same for all languages.
- X10 programmers spent relatively more time executing code and relatively less time authoring and tidying code.
- Subjects using MPI spent more time accessing documentation (tutorials were online; more documentation is available).
- A batch environment was used in this study --- use of an interactive environment will probably have a significant.impact on development time results

Outline

- Heterogeneous Processors
- X10 language
- Habanero Multicore Software project

Habanero: the Big Picture





Habanero Research Topics

1) Language Research (builds on X10)

- Explicit parallelism: hierarchical places for multicore
- Implicit deterministic parallelism: array views, parameter intents, HPF-style forall, Sisal-style loops and arrays
- Implicit non-deterministic parallelism: unordered iterators, partially ordered statement blocks

2) Compiler research (new)

- New Parallel Intermediate Representation (PIR)
- Analysis and transformation of PIR
- Optimization of high-level arrays and iterators
- Strength reduction of synchronization and STM operations
- Code partitioning for accelerators

3) Virtual machine research (builds on Jikes RVM)

- VM support for work-stealing scheduling algorithms with extensions for places, transactions, task groups
- Integration and exploitation of lightweight profiling in VM scheduler and memory management system

4) Concurrency library (builds on JUC and DSTM2 libraries)

- Fine-grained signal/wait, efficient transactions, new nonblocking data structures

5) Toolkit research (builds on Rice HPCtoolkit & Eclipse PTP)

- Program analysis for common parallel software errors
- Performance attribution of loops and inlined code using static and dynamic calling context

Target Applications

1) Parallel Benchmarks

- SSCA's #1, #2, #3 from DARPA HPCS program
- NAS Parallel Benchmarks
- Java Grande Forum benchmarks

2) Signal Processing

- Back-end processing for Compressive Sensing (www.dsp.ece.rice.edu/cs)
- Contact: Rich Baranuik (Rice)

3) Seismic Data Processing

- Rice Inversion project (www.trip.caam.rice.edu)
- Contact: Bill Symes (Rice)

4) Computer Graphics and Visualization

- Mathematical modeling and smoothing of meshes
- Contact: Joe Warren (Rice)

5) Fock Matrix Construction

- Contacts: David Bernholdt, Wael Elwasif, Robert Harrison, Annirudha Shet (ORNL)

6) Molecular Dynamics

- www.cs.sandia.gov/~sjplimp/download.html
- Contact: Steve Plimpton

Additional suggestions welcome

Target Hardware

- AMD Barcelona Quad-Core Opteron
- Clearspeed Advance X620
- DRC Coprocessor Module w/ Xilinx Virtex FPGA
- IBM Power6
- nVidia GeForce 8800GTX
- STI Cell
- Sun Niagara 2

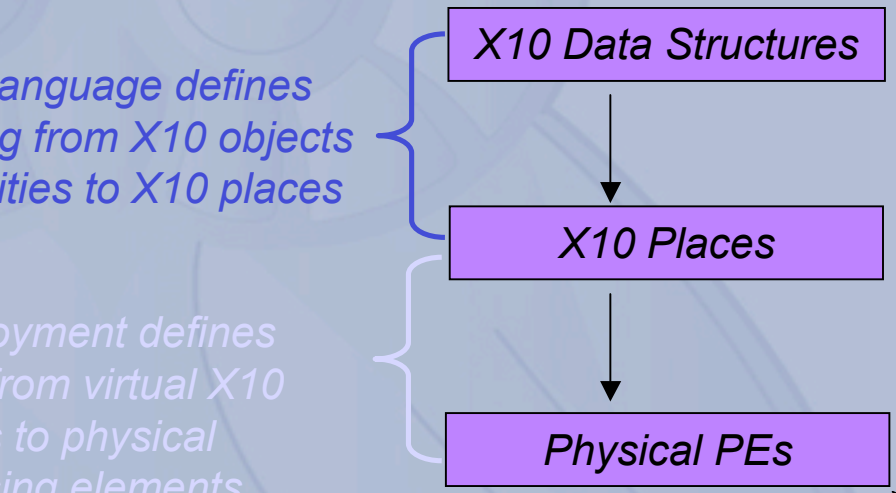
Additional suggestions welcome

Habanero Team

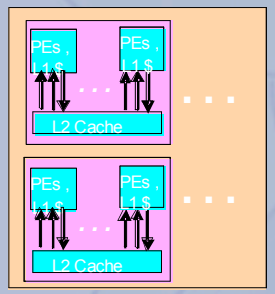
Habanero: Extending X10 Deployments for Heterogeneous Multicore

X10 language defines mapping from X10 objects & activities to X10 places

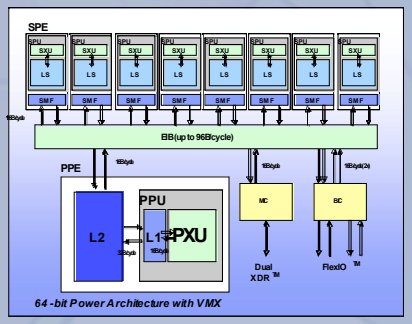
X10 deployment defines mapping from virtual X10 places to physical processing elements



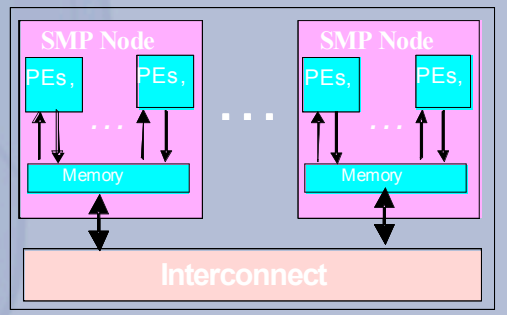
Homogeneous Multi-core



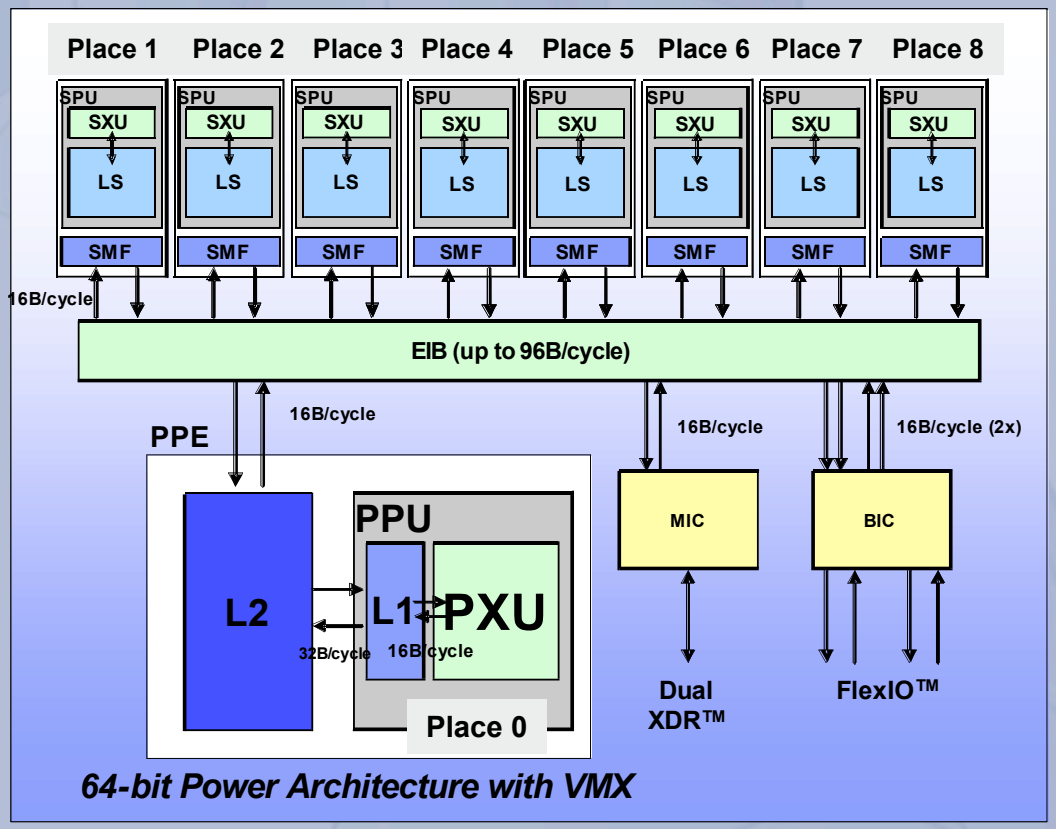
Heterogeneous Accelerators



Clusters

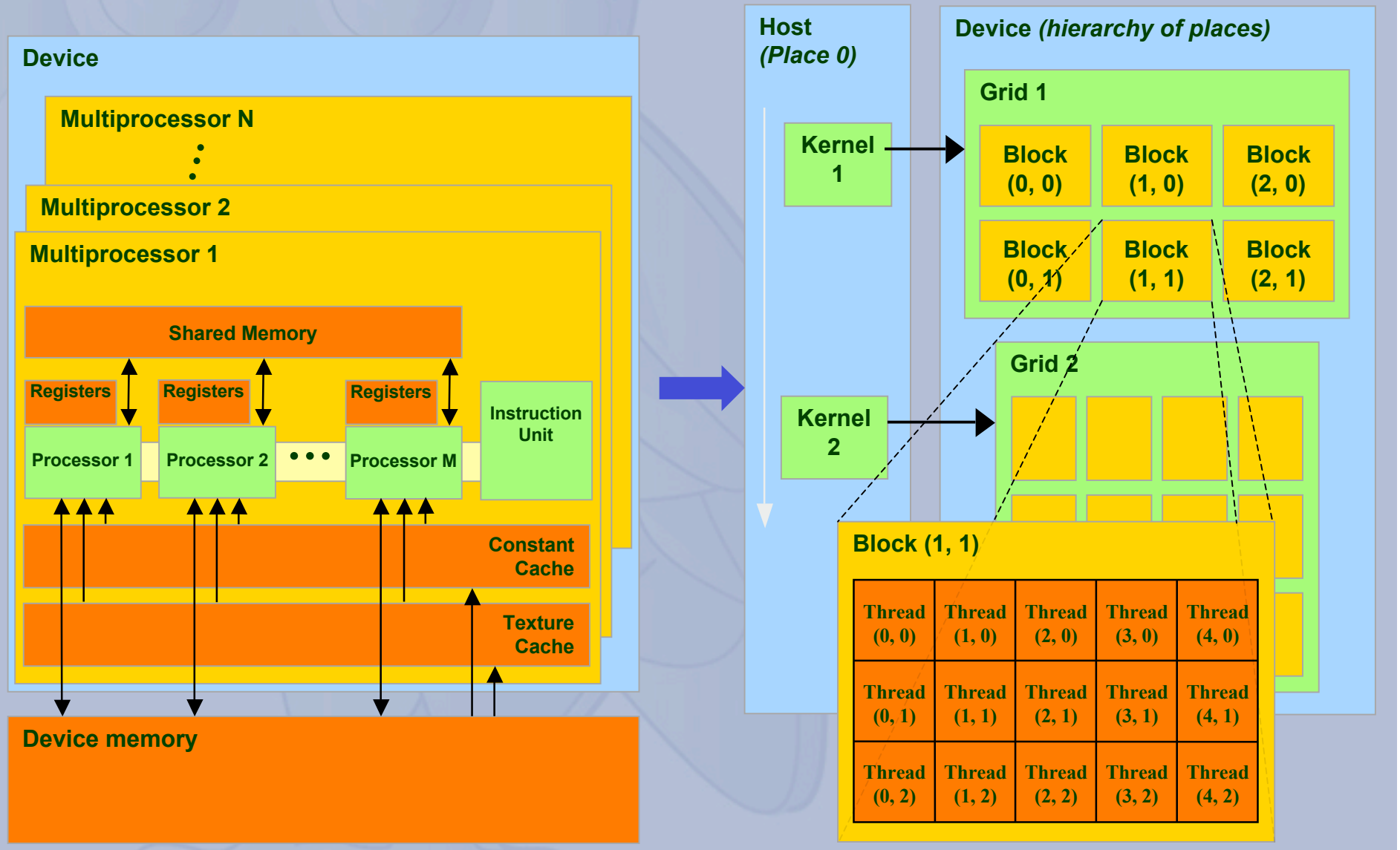


Possible X10 Deployment for Cell



- Basic Approach:
  - map 9 places on to PPE + eight SPEs
  - Use finish & async's as high-level representation of DMAs
- Challenges:
  - Weak PPE
  - SIMDization is critical
  - Lack of hardware support for coherence
  - Limited memory on SPE's
  - Limited performance of code with frequent conditional or indirect branches
  - Different ISA's for PPE and SPE.

Possible Deployment on Nvidia G80 (with extensions to support hierarchies of places)

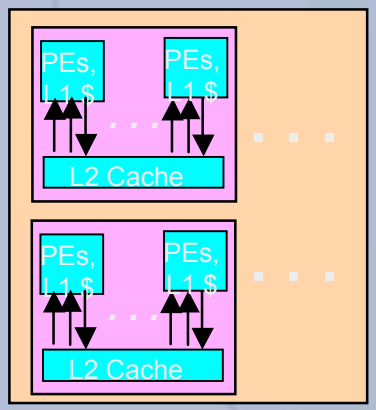


Opportunities for Broader Impact w/ Collaborators

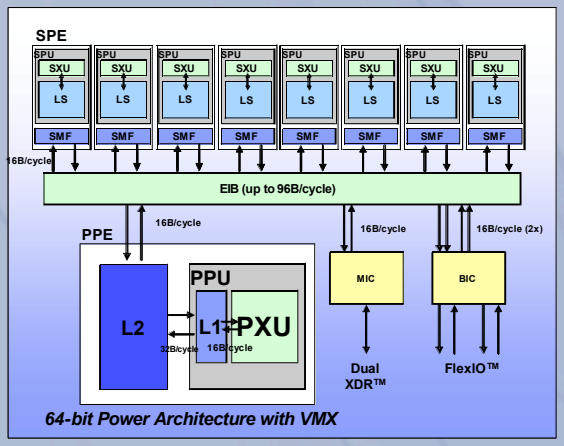
- Education
  - Influence how parallelism is taught in introductory Computer Science courses
- Open Source
  - Build an open source testbed to grow ecosystem for researchers in Parallel Software area
- Industry standards
  - Our research results can be used as proofs of concept for new features being standardized
  - Infrastructure can provide foundation for reference implementations

Conclusion

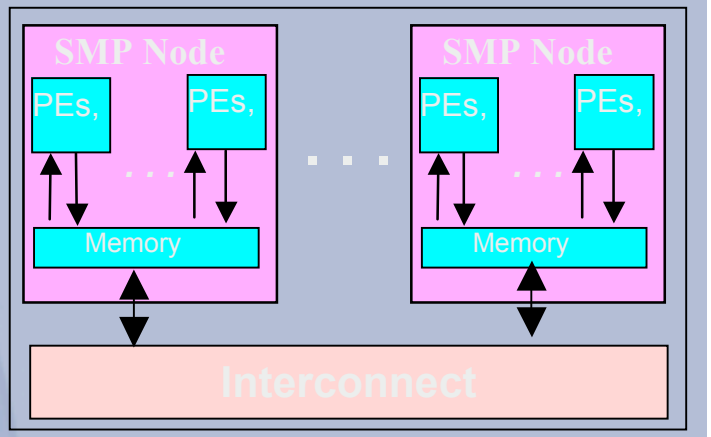
Homogeneous Multi-core



Heterogeneous Accelerators



High Performance Clusters



Portable Parallel Programming for Heterogeneous Multicore Computing is achievable with advances in languages, compilers, runtimes, and tools