The pC++ System Programming Environment

Part I:

The pC++ Language and Its Runtime System

Allen Malony Bernd Mohr

Dept. of Computer and Information Science
University of Oregon
Eugene, Oregon 97403

{malony,mohr}@cs.uoregon.edu

Platforms and Toolsets for C++ on Scalable Systems

Dennis Gannon, Indiana University
Allen Malony, University of Oregon
Mike Wolfe, Oregon Grad Institute

Project Goals:

To deliver a programming language and compiler toolset and program analysis infrastructure that supports

Fine Grained, Object Oriented Parallelism

that

1) will deliver multi-GFLOP to TFLOP performance across a *range* of scalable HPC platforms (*portability*);
2) is designed around software technology and libraries built for grand challenge problems in science and engineering (*extensibility*);
3) will provide complete HPF *interoperability*; and
4) will exploit performance analysis techniques based upon integration of detailed semantic analysis of the program with instrumentation and measurement (*integration*).

pC++ Basics

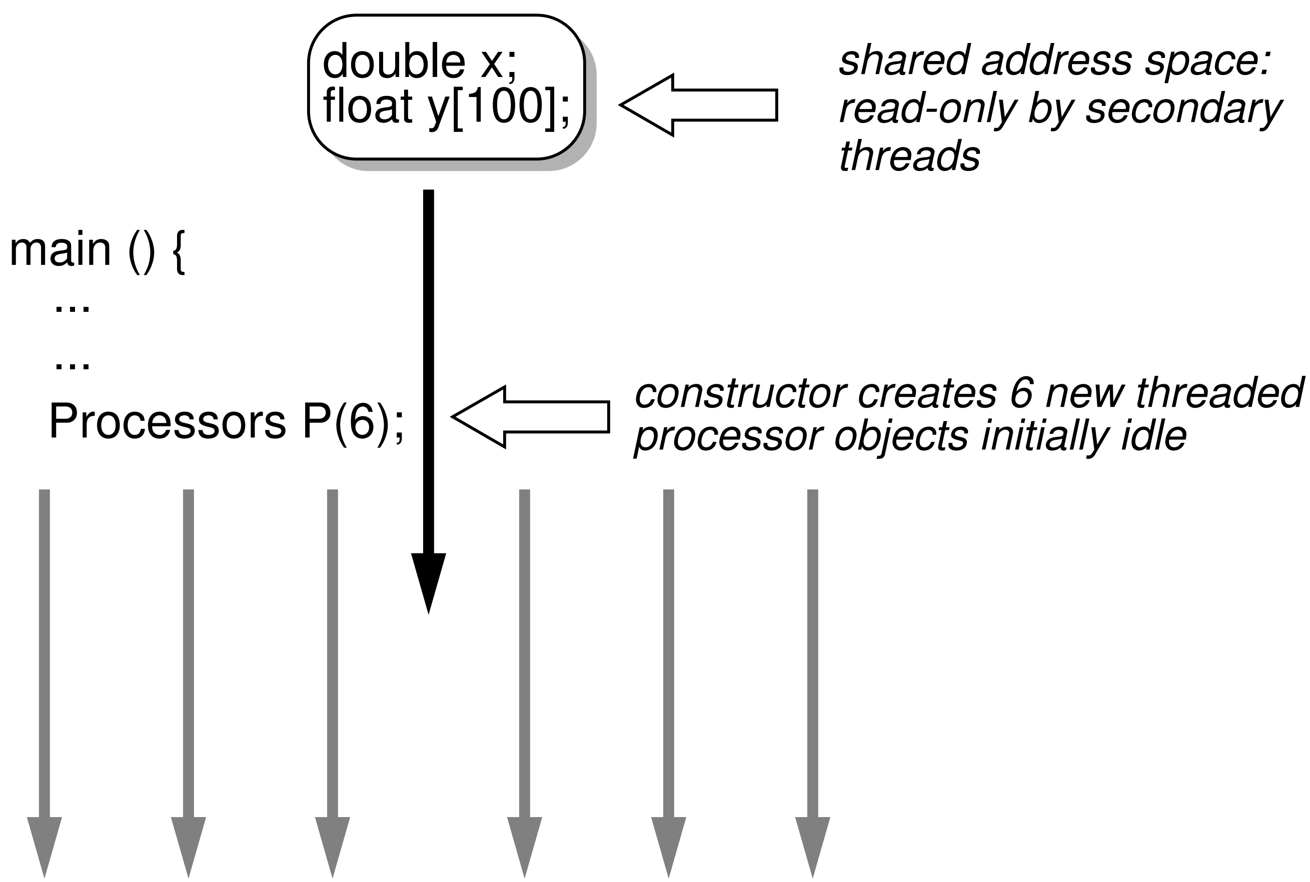
- pC++ is an extension to the C++ programming language. It provides a means to generalize data parallel execution in an MIMD environment.
- **Data Parallelism**: the parallel action of primitive operators on vectors and arrays of basic data values (Fortran 90)
- **Object Parallelism**: the parallel application of a class function to a set of object in a “concurrent aggregate”.
- pC++ follows HPF in the way it allows users to map data object to processors.
- It provides an interface to MIMD message passing libraries through **Thread Environment Objects**.

pC++ Implementation

- currently supported systems
  - CM-5
  - KSR-1 / KSR-2
  - Sequent Symmetry
  - IBM SP-1 / SP-2
  - Intel Paragon
  - BBN TC2000
  - UNIX Workstations (SUN, HP, DEC, IBM, SGI, ...)
  - Workstation Clusters with PVM (homogeneous)
  - Awesime
  - Proteus
- planned
  - C90, Cray T3D
  - Convex SPP
  - Meiko-CS2
  - Intel IPSC/860

Language Concepts:
1. Processors Objects

- Processors P;
  - P is the name of a set of threads that run on a set of processors.
  - current default: one thread per processor for each processors in the user's current "partition"
- The main program runs in a special thread and global variables are visible (read only) in the set of Processor Object threads.

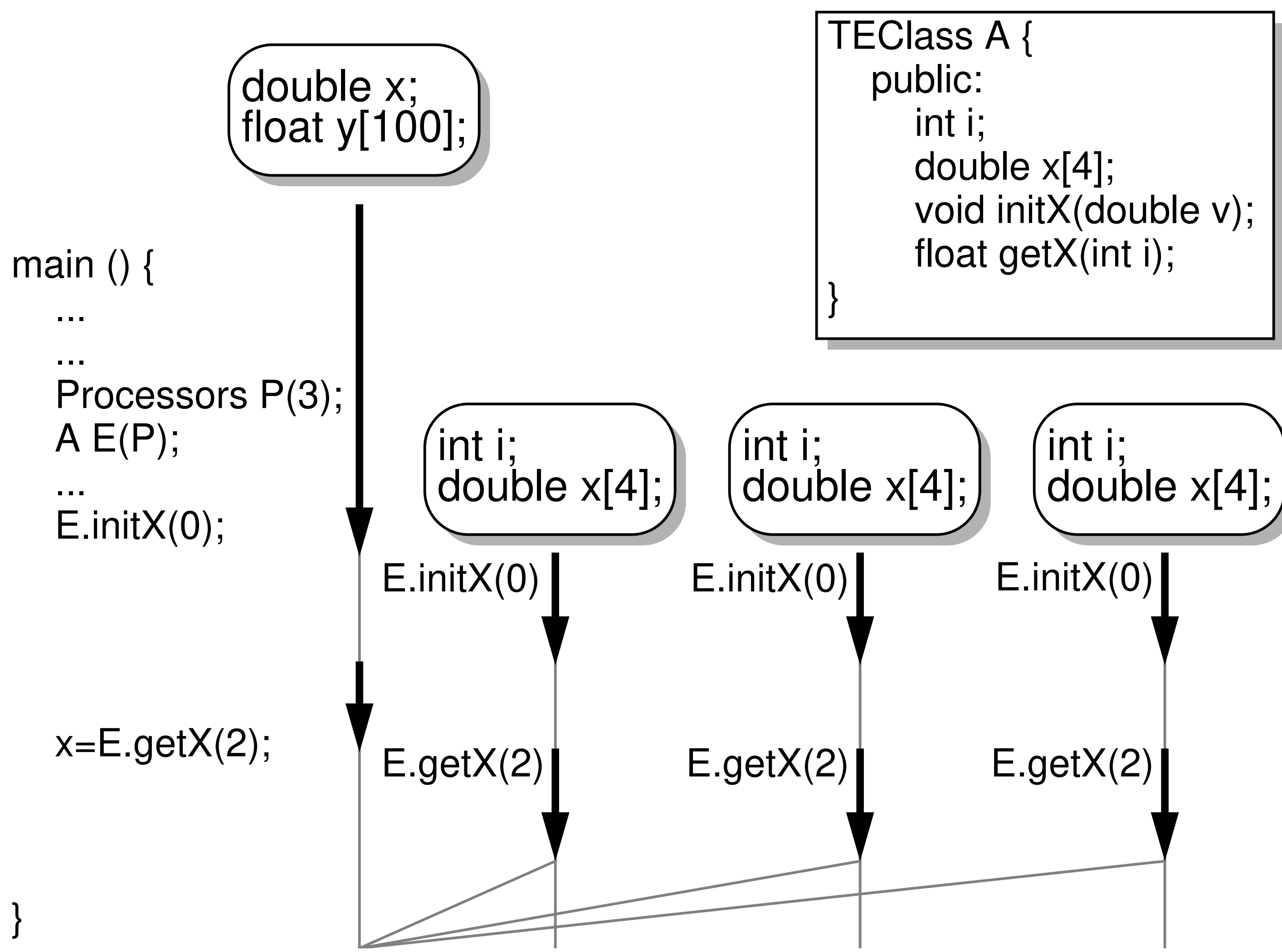


Language Concepts:
2. Thread Environment Classes

- **Thread Environment Classes** are bound to threads in a processors object. They are like ordinary C++ classes but allocating an instance of a TEClass creates one instance, called a **representative**, for each thread.

```
TEClass C {
   //private data for each thread
}
```

- Invoking a member function of the TEClass object causes the member function to be executed by each thread.



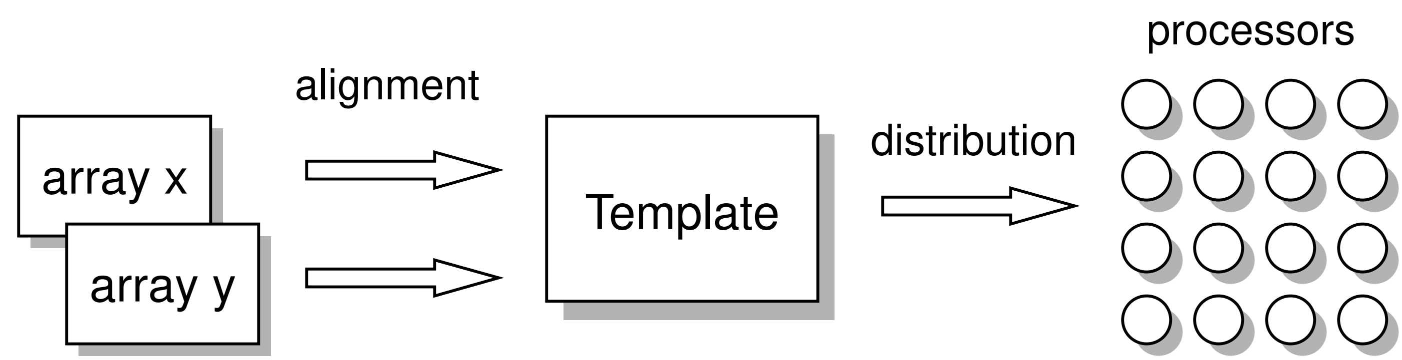
Language Concepts:
3. Collections

- **Collections** are structured sets of data objects from some "Element Class". Like

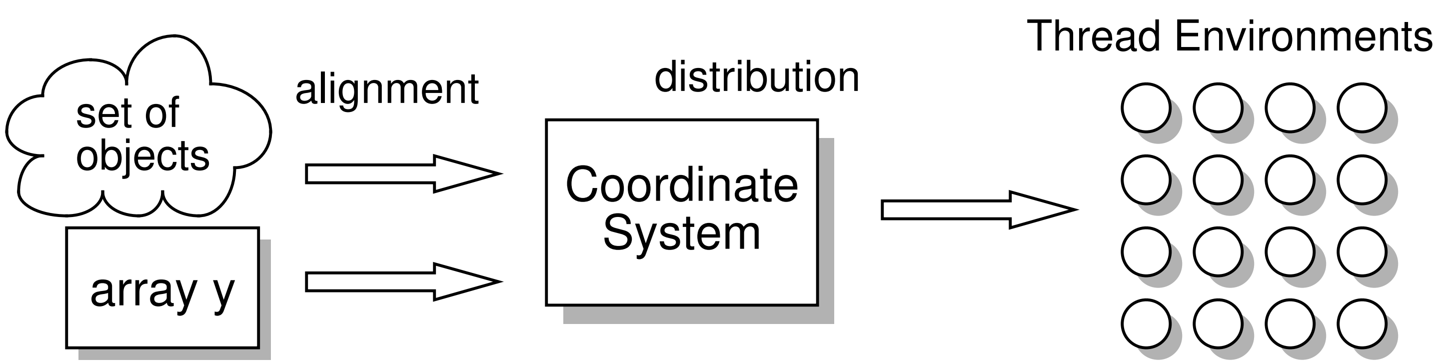
```
DistributedArray< Float >
   //read as distributed array of
   //floating point numbers
DistributedGrid< FiniteElement >
DistributedTree< particles >
```

- Collections are mapped to Thread Environments in the same way as arrays are mapped to processors in HPF.

in HPF:

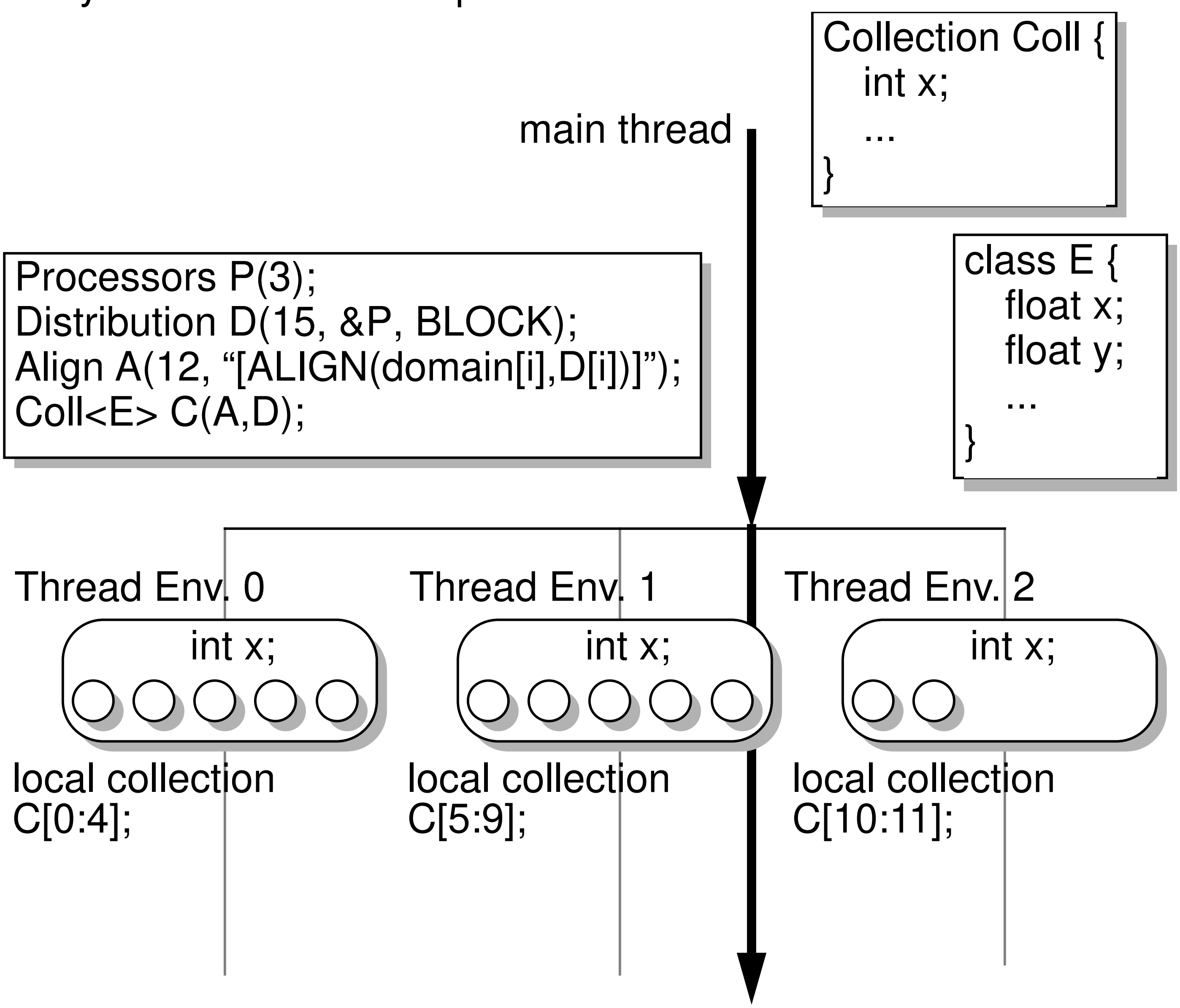


in pC++:



Collections (cont.)

- A Collection is a **Thread Environment Object** together with a set of elements that are mapped to the thread environment object **representative**.
- The set of elements mapped to each representative is called the **local collection**.
- Below, a 12 element collection is mapped to a set of three processor threads. The distribution is of size 15 and is mapped by block to the thread/processors.



Collections (cont.)

- Individual elements of a collection may be referenced with the selection function ()

```
class E {
   float x;
   float y;
   void g();
   float f();
}
```

```
Processors P;
Distribution D(1000, &P, BLOCK);
Align A(1000, “[ALIGN(domain[i],D[i])]”);
DistributedArray< E > C(A,D);
```

C(i) ⇐ returns a ‘global pointer” to collection element i

C(i).x ⇐ the value of x in the i-th element

C(i).f() ⇐ go to the thread containing the i-th element and make that thread execute f() and return the value (remote service request)

Collections (cont.)

- Parallel actions are generated by applying a member function from the element to the collection.
- There are several forms of this operation.

```
class E {
   float x;
   float y;
   void g();
   float f();
}
```

```
Processors P;
Distribution D(1000, &P, BLOCK);
Align A(1000, “[ALIGN(domain[i],D[i])]”);
DistributedArray< E > C(A,D);
```

C.g();	⟸	apply g() to each member of collection C
C.x = C.y+C.x;	⟸	apply addition operator in parallel
C.x = C.f();	⟸	C.f() returns a collection of the same alignment and distribution and of type DistributedArray< float>
C[0:30:4].g();	⟸	Application of g() to a subset F90 style

Runtime System Requirements

- ❑ Communication between Processor Objects
  - ❍ Distributed Memory architectures (DM):
    native message passing layer
  - ❍ Shared memory machines (SM):
    shared global variables
- ❑ Memory Allocation for Collection Elements
  - ❍ DM: malloc
  - ❍ SM: often require a OS-specific allocator
- ❑ Thread Generation and Termination
  - ❍ DM: handled by operating system
  - ❍ SM: Machine dependant code to “fork_and_bind” threads
- ❑ Barrier Synchronization
- ❑ User-code Runtime Primitives
  - ❍ `pcxx_NodeNumber()`
  - ❍ `pcxx_TotalNodes()`
  - ❍ `pcxx_WallTimerStart()`
  - ❍ `pcxx_UserTimerStop()`
  - ❍ . . .

Runtime Communication Primitives

- User Interface: Global Namespace

  Collection elements are enumerated by a one or two dimensional index. Any element can request a local copy of any other element by using its index

- `C` is a collection of type `Coll<E>`

  i-th element of `C` is obtained through Kernel functions

  - `C->Get_Element(i)`

    Returns a pointer of type `(ElementType *)` to a locally allocated copy of the i-th Collection element.

  - `C->Get_ElementPart(i, offset, size)`

    Returns a pointer to a copy of a chunk of data within the Collection element.

  - `x = C(i).g;`

    A simple user interface is provided by overloading “`()`”

- “owner computes” semantics

Current Implementation

COMMON

pcxx_stdio	877 / 50
parameter handling	117 / 1
timers	257 / 11
profiling	121 / 6
tracing	445 / 9
total	1817 / 77

DMARTS 71 / 8			SMARTS 168 / 16		
Active 32 / 0	Passive 818 / 28				
CM - 5 626 / 41	Paragon 276 / 16	SP -1 249 / 6	KSR -1 227 / 10	TC2000 2000 / 37	Symmetry 204 / 10

- numbers show `lines-of-code / number-of-functions`
- DMARTS implements already RSR, Fortran interface, reduction functions
- TC2000 includes gang scheduler / cluster allocation handling

pC++ Compiler/Runtime System Limitations

- currently, SPMD code is generated
  - processor objects are the same as processors
  - one thread per processor
  - one thread for all local collection objects
  - “owner-computes” rule applies
- only one parallel collection operation at a time
- no nested concurrency

➠ Model of execution is consistent with all current HPF implementations.

Distributed Memory Runtime System

- Each collection element has a
  - **owner**:

    processor object that contains element in local collection (dynamic distribution)
  - **manager**:

    keeps track of which processor object owns the element

`Get_Element(i)` implementation

Manager		Requester		Owner
"interrupt"	←	get owner id		
return owner id	→	owner id		
		request i	→	"interrupt"
		copy of i	←	return copy of i

Distributed Runtime System Issues

- Processor interrupts.
  - no mechanism for interrupting a processor thread
  - `pcxx_Poll()` is called in element and collection methods
  - `pcxx_Poll()` is call in `pcxx_Barrier()`
- Table of owner identifiers storage.
  - table for local collection element pointers
  - table for element owners
- Barrier operation

Distributed Memory RTS: Implementation

- TMC CM-5 (256 nodes)
  - CMMD 3.0 beta (CMAM)
  - interrupt and polling
  - good short message performance
  - fast barrier synchronization (4 microseconds)

- Intel Paragon (56 nodes)
  - pre-release software
  - NX communication
  - polling only
  - poor short message performance
  - polling barrier (3 milliseconds)

- IBM SP-1 (128 nodes)
  - EUIH communication (currently)
  - MPI communication (soon)
  - polling only
  - polling barrier

Shared Memory Runtime System

- Differences from DM runtime system
  - shared addresses (obvious)
  - possibly shared collection tables
  - memory locality in collection allocation

- Strategy
  - replicate collection element tables
  - processor-local collection element allocation
  - barrier synchronization optimized for HW/SW
  - The function `Get_Element(i)` finds out where the data is stored, and creates a copy by simply reading shared memory.

Shared Runtime System: Implementation

- BBN TC2000 (96 nodes)
  - NX fork_and_bind
  - local node allocation of owner elements
  - write through caching
  - software log barrier (70 microseconds/32 nodes)

- Sequent Symmetry (24 nodes)
  - copy-back caching policy
  - interleaved memory
  - shared VM segment allocation
  - Sequent-supplied hardware-based barrier synchronization (70 microseconds/20 nodes)

- KSR KSR-1 (64 nodes)
  - POSIX (kernel-level) threads
  - KSR-supplied barrier synchronization
  - shared VM allocating
  - hierarchical memory architecture
  - different latencies between node caches (150 cycle on same ring, 500 cycles on different rings)

pC++ Version 2.0

- ❑ Language and runtime system support for dynamic creation of threads and “contexts”, address spaces in which program threads are scheduled and execute
  - ❍ nested parallelism (collections of collections)
  - ❍ asynchronous execution
  - ❍ intra/inter address space parallelism
- ❑ Global pointers for referencing data objects in other contexts
- ❑ Thread execution scheduling based on user-defined work sharing strategies
- ❑ Extended compiler technology for optimizations of object oriented languages and RTS
- ❑ Runtime libraries and language support for persistent objects and parallel I/O
- ❑ Operation on heterogeneous high-speed networks of MPP servers and clients

HPC++

- ❑ Merger of OO parallel language / RTS technologies
  - ❍ principally, CC++ and pC++
  - ❍ portable runtime system infrastructure (PORTS)
  - ❍ extended library interoperability

HLL programming environment

evaluation criteria

1) portability
2) extensibility
3) retargetability
4) user needs / requirements
5) integrability
6) scalability

technology

1) compiler based
2) program analysis based
3) language based
4) debugging / performance / visualization technology
5) Runtime System

Infrastructure

1) transformation / optimizations tracking technology
2) Sage++ / ELI
3) PORTS
4) tracing / profiling / checkpointing / ... support
5) debugging / performance / visualization tools

targets

1) pC++
2) C*
3) CC++
4) HPF
5) Fortran-M
6) ZPL

For more Information ...

τ

☆ Tools Are Us ☆

- **WWW/Mosaic**:
  - http://www.cica.indiana.edu/sage
- **Anonymous FTP**:
  - ftp.cs.indiana.edu:/pub/sage (129.79.254.191)
  - cica.cica.indiana.edu:/pub/sage (129.79.20.27)
- **finger**:
  - sage@cica.indiana.edu
- **mail**:
  - sage@cica.indaina.edu