Integrating Task and Data Parallelism

Ian Foster
Argonne National Laboratory

Carl Kesselman
California Institute of Technology

Overview

- Task and Data Parallel Programming Paradigms
- Why Paradigm Integration
- Approaches to Paradigm Integration
- Compositional Programming
- Examples

Parallel Programming Paradigms

Parallel Programming Paradigms

- A parallel programming paradigm is a mechanism for structuring a parallel program to limit the complexity of parallel programming.
- Many different types of paradigms:
  - data parallelism
  - task parallelism
  - object parallelism
  - functional parallelism
- Support for a programming paradigm can be:
  - language constructs
  - programming libraries
  - architectural features

Data Parallelism

- Types of data parallelism
  - Single Instruction Multiple Data (SIMD)
  - Single Program Multiple Data (SPMD)
- Structured synchronization
  - communication and synchronization occur at predictable points in time
- A single logical thread of control
  - data parallel programs often can be given a sequential reading
- Deterministic
- Parallelism "follows" the data
  - owner compute rule

SIMD Programming

- Same exact instructions execute over every data element
- Masking out of data elements allowed
- Both sides of every conditional statement must be taken on every processor
- Typified by vector operations
- Examples of SIMD systems
  - C*
  - CM Fortran
  - Fortran 90

SPMD Programming

- Same program runs in every processor
- Operations performed on a given processor can depend on data
- Typical of most current applications
- Examples of SPMD systems include:
  - HPF (forall statement)
  - pC++
  - Message passing systems: NX, MPI, PVM

Observations on Data Parallelism

- Concurrency can be over general objects
  - data parallelism is not limited to arrays
- Elements in data parallel structure do not have to be limited to numeric types
  - we want to consider collections of arbitrary objects
- Operations being applied do not have to be limited to simple arithmetic operations
  - i.e., apply a function to every element of a collection producing a new collection

Task Parallelism (MIMD)

- Unstructured synchronization and communication
- Nondeterministic
- Multiple threads of control
- Generally cannot be reasoned about sequentially

Task Parallel Systems

- Synchronous communication languages
  - Ada, occam
- Strand, PCN
  - dynamic process creation and synchronization
- Object parallel systems
  - Actors, Chare Kernel, Concurrent Smalltalk
- General message passing libraries with selective receive and message tags
  - P4, PVM, MPI, NX
- Linda
  - generalized task pool

Task Parallel Applications

- Multiple programs
  - Multidisciplinary applications
  - Discrete event simulation
- Nondeterministic execution
  - Command and Control
  - Interactive systems
  - Load balancing algorithms
- Irregular data structures
  - N-Body Problems
  - Nested grids

Limitations of MP Systems for Task Parallelism

- Message passing libraries often provide no assistance in writing dynamic, nondeterminisic task parallel code
  - dynamic communication structures
  - dynamic process structures
- Messages are sent to a node with an absolute name
  - What task gets the message, where did the message come from?
  - Can be worked around by with polling and message tags
- Unshareable resources
  - E.g. Send buffers in PVM 2,0
- MPI attempts to address these problems

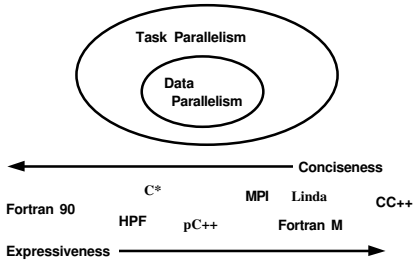
The Data Parallel Dilemma

``Give up generality in return for ease of expression and compiler assistance''

The Troubles with Task Parallelism

"Its often too hard do do simple things"

The Paradigm Continuum

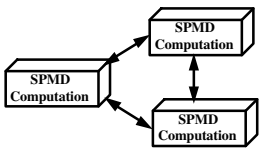


Why Paradigm Integration

- We need not be restricted to one point on the continuum!
- Paradigm integration: allow programmers to use both data parallelism and task parallelism, as appropriate
- Why? Data parallel view:
  - Some things just can't be written DP
  - Some things are unnecessarily hard to write DP
- Why? Task parallel view:
  - Conciseness of DP operations
  - Assistance of DP compilers

The Future: Integrated Systems

- Many future applications will consist of collections of parallel computations connected together to form parallel programs



Goals of Paradigm Integration

- Combine SPMD and MIMD Programming model within a single program
- Must address libraries and code reuse
  - will want to embed existing SPMD programs within new MIMD program.
  - calling an SPMD library from an SPMD code
- Other programming models might be accommodated as well:
  - functional programming, object parallel programming, database

Pros and Cons of Paradigm Integration

- Pro: Generality and ease of expression
- Con: More complex programming systems
- Challenge: Achieve integration without compromising advantages of either approach

Integration Issues

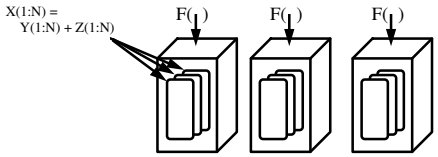
- languages vs. libraries
- heterogeneous environments
- sharing distributed data structures
- complex data structures
- local/global mappings
- retaining performance advantages of each approach
- code reuse
- libraries

Types of Paradigm Integration

- SPMD(SPMD)
  - nested data parallelism
- MIMD(MIMD)
  - nested task parallelism
- SPMD(MIMD)
- MIMD(SPMD)
- SPMD(R)
- MIMD(R)

SPMD(SPMD)

- Data parallel operations called from data parallel routine

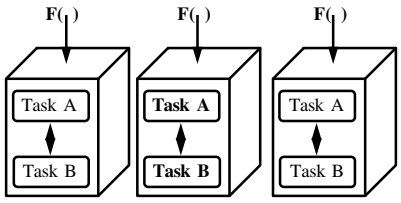


Uses of SPMD(SPMD)

- Data parallel libraries called from data parallel routines
  - forall X do something complicated and data parallel

SPMD(MIMD)

- Task parallelism called from within data parallelism



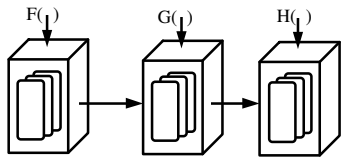
Uses of SPMD(MIMD)

- HPF extrinsic
- Heterogeneous collections
- Image processing
  - for each image, do . . .

SC93 25

MIMD(SPMD)

- SPMD operations within an MIMD framework
  - some instances can be thought as SPMD(SPMD)



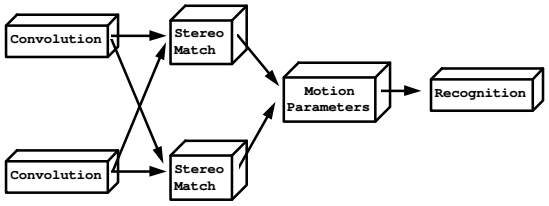
SC93 26

Uses of MIMD(SPMD)

- Data parallel pipelines
  - image processing
  - seismology, astrophysics
- Computer graphics and modeling
- Multi-disciplinary optimization
- Adaptive mesh refinement
- Image processing

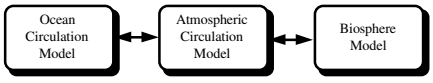
SC93 27

Example: Motion Estimation



SC93 28

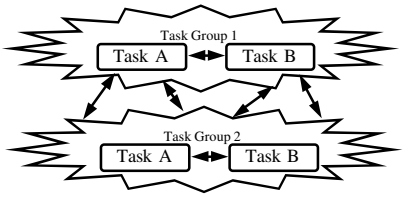
Example: Earth System Model



SC93 29

MIMD(MIMD)

- Nested MIMD computations, could use different MIMD paradigms.



SC93 30

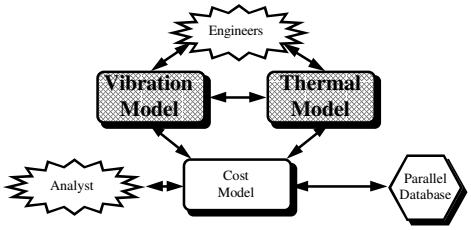
Uses of MIMD(MIMD)

- Shared memory, message passing programs
- Pipelines of message passing programs
- Load balanced MIMD tasks

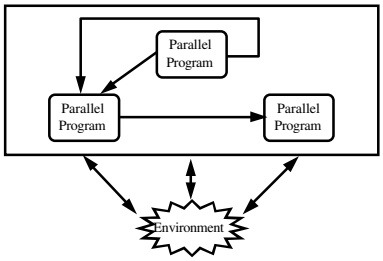
SPMD(R) and MIMD(R)

- SPMD(R): Recursive use of a any paradigm with a SPMD top level structure
- MIMD(R): Recursive use of a any paradigm with a task parallel top level structure
- Needed if any kind of code reuse in parallel programs is to be achieved

Example: Multi-Disciplinary Analysis



Parallel Reactive Systems



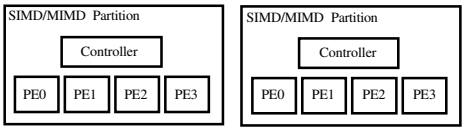
Approaches to Paradigm Integration

Approaches to Paradigm Integration

- Do it all yourself
  - If low level primitives are available, you can do anything in a task parallel framework.
- Data parallel libraries in task parallel languages
- Systems with primitives for both task and data parallelism
- MIMD coordination languages for data parallel modules
- Compiler directives

PASM: An Example of Paradigm Integration

- Partitionable SIMD/MIMD (Siegel)
- Computer architecture to support:
  - image processing
  - speech processing
- Dynamically reconfigurable partitions from SIMD to MIMD



SC93 37

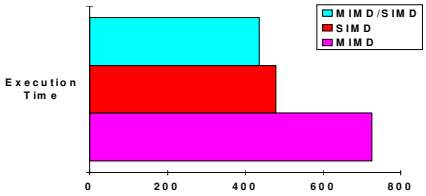
Paradigm Integration under PASM

- Multiple SIMD partitions
  - SIMD partition spans one or more controllers
- SIMD synchronization of MIMD computations
  - use SIMD barrier for synchronization of independent computation
  - Data parallel operations on non-array data structure

SC93 38

PASM Example: FFT

- Performance study (Jamieson89) of PASM modes
- FFT was example studied: integrated was best.



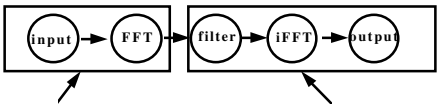
SC93 39

Coordinating Data Parallel C Modules

- Seevers, Quinn, Hatcher
- Coordination language for Data parallel C (C*)
- Data parallel C programs incorporate channel read and write statements
- Separate coordination language defines static process/channel networks
- Intermodule communication is sequentialized
- Implementation on iWARP
- Application to pipeline algorithms

SC93 40

Example: Signal Processing



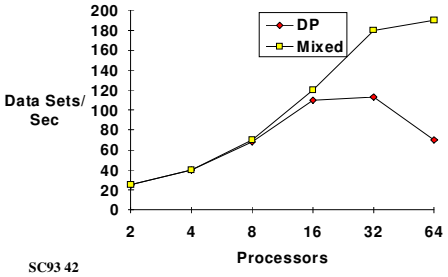

```
fp = fopen("outchan","wc");
...
fwrite(d, sizeof(d),1,fp);
...
fclose(fp);
```

```
fp = fopen("inchan","rc");
...
fread(d, sizeof(d),1,fp);
...
fclose(fp);
```

```
height 2
width 4
load module1 on [a0:b1]
load module2 on [c0:d1]
bind module1.outchan[b0] to module2.inchan[d1]
```

SC93 41

Signal Processing Example: Performance



SC93 42

AVS as a Coordination Language

- Chang, Fox, Mills
- Use AVS to define static task/channel structure, with tasks written in CM Fortran, CMMD, Fortran 77
- Convenient graphical programming environment
- Runs in heterogeneous environments
- Limited to static, acyclic graphs
- Performance issues

SC93 43

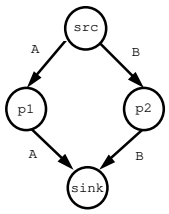
A Compiler-based Approach

- Subhlok, Stichnoth, O'Hallaron, Gross: CMU
- HPF for data parallelism; program directives to specify data flow
- Compiler generates task or data parallel code, depending on problem size, machine size, etc.
- Retains sequential semantics
- Compiler target is the iWARP multicomputer

SC93 44

iWARP: Example

```
C$    begin parallel
      do  i = 1, 10
          call src(A,B)
C$        output: A,B
          call p1(A)
C$        input: A
C$        output: A
          call p2(B)
C$        input: B
C$        output: B
          call sink(A,B)
C$        input: A,B
      enddo
C$    end parallel
```

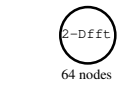


SC93 45

Two-Dimensional FFT

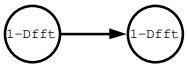
```
C$    begin parallel
      do  i = 1, m
          call rffts(A)
C$        output: A
          call cffts(A)
C$        input: A
      enddo
C$    end parallel
```

1. Purely data parallel
   -- one thread of control



64 nodes

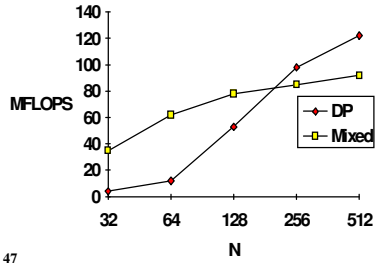
2. Mixed task/data parallel
   -- pipelined tasks



32 nodes 32 nodes

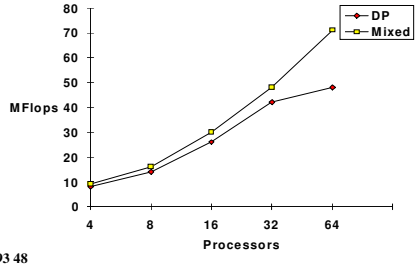
SC93 46

2-D FFT: Performance on 64 iWARP Processors



SC93 47

2dFFT 128x128 Performance on iWARP



SC93 48

HPF Extrinsic Procedures

- Data parallel top level
- Procedures can be declared extrinsic
- One copy of extrinsic procedure is called on each processor with access to local data and local names
- Extrinsic procedure can execute arbitrary MIMD code
- Barrier on termination of extrinsic procedure

Summary of Current Work

- Mostly targeted at static MIMD(SPMD) applications
- Does not address SIMD(R) and MIMD(R)
- Does not provide a general framework for paradigm integration.
- We propose a general framework through compositional programming

Compositional Programming

Our Approach

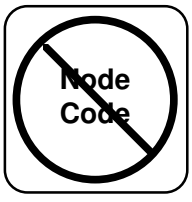
- Language extensions, not new languages
  - Augment existing programming languages with *small* set of extensions to support MIMD constructs in conjunction with existing SPMD models
- Initial focus C, C++ and Fortran 77
- Concerns include:
  - ease of learning
  - ease of use
  - determinacy
  - ease of debugging
  - code reuse
  - efficiency

The Goal

- Provide a framework to simplify the task of writing multi-paradigm programs
- Multi-paradigm programs inherently more complex then single paradigm programs.
- Central issues:
  - modularity
  - compositionality
- Focus on:
  - ease of use and expression
  - flexibility
  - correctness
  - modularity

Define Program Behavior in the Program

- The behavior of a program is defined by its text, ***not*** by the number of processors used.



What Are the Options

- Introduce completely new programming languages
  - PCN
  - SISAL
  - Occam
- Introduce parallelism through libraries
  - PVM
  - Express
  - NX
  - MPI
- Extend existing languages
  - Fortran M
  - Compositional C++

Why Not New Languages

- New syntax/semantics is often a "barrier to entry"
- Often hard to do simple things
- Limited reuse of existing code
- Our experience has been that it is hard to people to try something new.
  - Scientists want fast programs not parallel programs

Libraries for Parallel Programming

- Perceived as minimal cost solution
  - not necessarily true in practice
- Program text does not reflect program behavior
- No compiler support for error checking
- Limits on optimizations
- Limited types of constructs available
  - procedure interface
- Requirement for multiple language bindings can make interface awkward

Language Extensions for Parallel Programming

- New semantics are represented directly in the language
- Extensions close to the base language minimize the amount of new things that must be learned
- Extensions can provide a general framework
- Compile time type-checking of shared data or messages
- Seamless integration of data and task parallelism

Designing Language Extensions

- Base MIMD programs on existing programming languages.
  - Fortran 77/Fortran 90/HPF
  - C/C++
- Make extensions consistent with base language
  - avoid temptation to *fix* things
  - existing programs in base language should be legal programs in extended language
- Build extensions on existing language concepts
- Keep extensions consistent with the "philosophy" of the base language

Language Design for Paradigm Integration

- Modularity/compositionality is essential to multi-paradigm programs.
  - a general framework requires that program components must be usable in *any* environment
- Capabilities that aid in paradigm integration:
  - Explicit parallel constructs
  - Typed communications
  - Logical communication
  - Virtual processors

Explicit Parallel Constructs

- Language should have explicit constructs for parallelism
  - not strictly necessary for data parallel execution
  - task parallelism is harder to identify than data parallelism
- Explicit parallelism required to plug together modules
- some applications are naturally expressed in parallel
  - command and control systems
  - realtime systems
  - embedded systems

Explicit Parallel Constructs

- Parallelism should be explicitly represented in a program with some form or parallel block
  - Basic model is parbegin/parend
- Semantics of the program are independent of the execution environment
  - Execution of statements in parallel block is *fair* and *interleaved*
  - Parallel blocks used for concurrent execution and to hide latency
- Easier to reason about program behavior
  - explicit start and end of parallel execution

Typed Communications

- Communication is a mechanism for sharing and should interact with the type system as other mechanisms for sharing.
  - e.g. variables, procedure calls
- Interactions between modules in multi-paradigm programs can be complex.
  - Need to ensure that components are plugged together correctly

Logical Communication

- Most current communication mechanisms process based
  - send msg to node number
- Multi-paradigm programs invalidate the assumptions associated with process based communications
  - Communication is between logical entities in program, not nodes.
  - need to ensure that communication is routed to appropriate destinations.

Virtual Processors

- SPMD computation often makes assumptions about the environment
  - One thread per processor
- Modularity requires independence from environment
- Task parallel component will need to manipulation data parallel environment
- Code needs to function properly regardless of environment

Language Implementations

- Modular Fortran (Fortran M)
  - Based on Fortran 77 (Fortran M)
- Compositional C++ (CC++)
  - Based on C++ 3.0

Fortran M Design Goals

- Scientific users
- Combine existing program modules into parallel programs
- Flexible communication structures
- Extension philosophy:
  - Extensions based on arrays, subroutines and I/O statements
  - Communication by extension of Fortran I/O
    - » One type of communication structure (virtual channel)
  - Deterministic programs by default

CC++ Design Goals

- Preserve C++ efficiency
- Provide infrastructure for building parallel paradigm class libraries
- Extension philosophy:
  - Extensions based on type modifiers, objects, blocks and pointers
  - Communications by extension of extension of C type system
  - Full use of C pointers
    - » Deterministic programs by design
    - » Flexibility to construct many different programming and communication paradigms

Parallel Blocks

CC++

```
par {
 STATEMENT;
 ...
 STATEMENT;
}

parfor(int i; i n;i++)
  STATEMENT;
```

Fortran M

```
   processes
      PROCESSCALL
      ...
      PROCESSCALL
   endprocesses

   processdo 10 i = 0, N
   PROCESSCALL
10 continue
```


Parallel Execution Semantics

- Parallel blocks and loop statements terminate when composed statements terminate.
- The execution of statements in a parallel composition is:
  - interleaved
  - fair
- Interleaving is nondeterministic, but other language constructs ensures that the computation is deterministic.

Concurrency in Fortran M

- Processes analogous to program
  - a program can have many processes
  - each process has its own data set, including common blocks
  - declared with process keyword
- Processes are dynamically created
  - processes block
  - processdo loop
- Implicit barrier at end of process block and processdo.

Concurrency in CC++

- Parallel blocks
  - Can be nested like any C block
  - Implicit barrier at end
- parfor statements
  - Index evaluation unfolds sequentially
  - Loop body for each iteration executed in parallel
  - Implicit barrier at end
- spawn statement
  - Unstructured parallelism, no implicit barrier.

Communication

- Communication is between logical entities, not physical
  - interaction is between structures in the programming language, not components in the environment
- Requirement for reusable code
- Communications should be typed
  - typing improves the reliability of a parallel program
- C already has a way of sharing data, the pointer
- Fortran lacks such a construct, so we must introduce a way of sharing

SC93 73

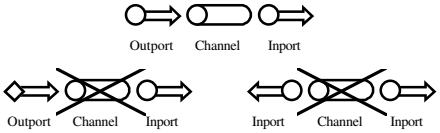
Communication in Fortran M

- Design considerations
  - need a mechanism for sharing
  - Fortran 77 does not have arbitrary data structures
- File I/O provides a good model
  - A file is a non-array shared structure
  - Sharing provided in that once written data can be read at some later time
  - File I/O is "logical" in nature
- Fortran M introduces a *virtual channel*
  - Readers and writers are represented by *ports*
  - Ports can be manipulated like any other data structure
  - Ports are dynamically connected together to form a *channel*

SC93 74

Fortran M Ports

- Ports are unidirectional and typed
  - `outport (integer, real) source`
  - `inport (integer, real) sink`
- Ports are connected by channels
  - `channel(out=source, in=sink)`



SC93 75

Operations on Channels

- Send and receive statements put data and take data out of a channel

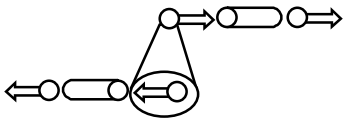
```
send(source) n, xin
receive(sink) n,xout
```

- Attempting to receive from an empty channel will cause the statement to block

SC93 76

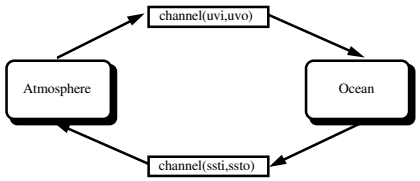
Dynamic Channels

- Ports can be placed onto a channel
- Port connections are statements not declarations
- Dynamic configuration of channels is possible
- Example: enable back communication from unknown source



SC93 77

Using Channels: Coupled Models



SC93 78

Code for Coupled Models

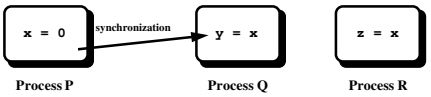
```
process coupled_model
parameter(NLAT=128, NLON=256)
inport (real x(NLAT,NLOG)) ssti
outport (real x(NLAT,NLOG)) ssto
inport (real x(NLAT,NLON), real y(NLAT,NLON)) uvi
outport (real x(NLAT,NLON), real y(NLAT,NLON)) uvo
channel (out=ssto, in =ssti)
channel(out=uvo, in=uvo)
processes
  processcall atmosphere(ssti,uvo)
  processcall ocean(uvi,ssto)
endprocesses
end
```


Communication in CC++

- Since C has pointers and structures, virtual channels can be constructed given the correct primitives for synchronization
- Base synchronization in CC++ on a generalization of the ANSI C `const` keyword
  - Variables declared `const` cannot be modified
  - CC++ adds type modifier `sync`
  - A `sync` is like a `const` with delayed initialization
- Sync objects can express
  - Synchronization over all C data structures
  - More complex modes of communication then producer/consumer relationship of virtual channel

Sync is Constant With Delayed Initialization

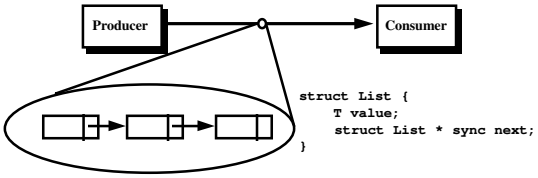
```
sync int x; // Create data object
...
x = 0; // delayed initialization
```



No problem with multiple-copy consistency, both copies are 0

Channel Communications in CC++

- A producer/consumer is a virtual channel.
- Use sync list cells to communication data between producer and consumer.



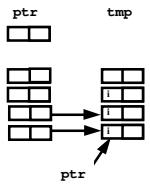
Producer/Consumer

```
Sequential Code

{
    producer(ListHead);
    consumer(ListHead);
}
```

```
Parallel Code

par {
    producer(ListHead);
    consumer(ListHead);
}
```


Producer

```
producer(List * ListHead) {
   List  *tmp, *ptr;
   int i;
   ptr = ListHead;
   for(i = 0 ; i < n; i ++) {
      tmp = new List;
      tmp->value = i;
      ptr->next = tmp;
      ptr = tmp;
   }
   ptr->next = NULL;
}
```



Consumer

```
consumer(List * ListHead) {
   List *ptr;
   ptr = ListHead->next;
   while (ptr != NULL) {
      consume(ptr->value);
      ptr = ptr->next;
   }
}
```


Determinism

- Scientific users need repeatability
  - chaotic systems
  - debugging
- Determinism in computation, compute the same answer every time
  - interleaving of parallel components for efficiency and portability
- We need determinism in results
  - Actual order of statement execution can vary

Controlling Determinism

- Determinism can be ensured by language constructs, compiler and runtime system.
- Nondeterminism is sometimes required for efficiency and expressiveness
- Introduce nondeterminism in a controlled manner

Determinism in Fortran M

- Focus on ensuring determinism
- Compiler and runtime ensure that each channel has a single producer and a single consumer
- Single producer/single consumer guarantees that a program is deterministic
- Nondeterminism can be explicitly introduced by using a merge statement

Determinism in CC++

- Determinism is provided by libraries and usage restrictions
  - programs that only share sync objects are deterministic
- Interleaving of program statements can be controlled via an atomic declaration

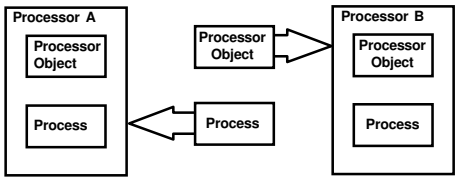
Mapping Computation to Processing Resources

- We cannot talk about absolute processor addresses.
- In a multiparadigm program, it is not wise to assume that the mapping between program components and processing elements is one to one.
- Portability concerns make separation of mapping from algorithm important
- Solution is to introduce notion of logical processor.

Virtualizing Processor Resources

- Fortran M processes
  - Process is a generalization of a subroutine
    - » Process declared with `process` keyword
  - Common blocks are local to a process
  - Instance of a process is created in a process block or process do
- CC++ Processor Objects
  - Processor object is like a C++ object
    - » Global and static variables are local to an instance of a processor object
  - C++ pointers require interprocessor object references.
    - » pointers can be declared to be `global`
  - Processor objects can be dynamically created like any other C++ object

Resource Management with Logical Processes



Fortran M Machines

- Virtual processors are grouped into a machine, the machine is organized as an array:

```
processors pr(NP,NP)
```

- Processes within a process block can be explicitly placed on a machine element

```
call  solve(x,y) location (i,j)
```


Subsetting Processing Resources

- Computations can be allocated to submachines

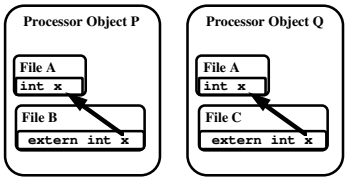
```
c Double the number of virtual processors
  processors pr(NP,2*NP)
  processes
c   map ocean to first NP processors in column
c   major order
     call ocean(ToAtm,FromAtm) submachine(1,1)
c   map atmosphere to second NP processors
c   in column major order
     processcall atm(FromAtm,ToAtm)submachine(1,NP+1)
  endprocesses
```


CC++ Resource Allocation: Processor Objects

- An object encapsulates data and code.
- An object provides an interface through which others agents can use the object.
- We apply the concept of objects to whole programs,
  - a C++ program is a CC++ *processor object.*
- A CC++ program consists of one or more processor objects.

Processor Objects are Self Contained

- There is no linkage between processor object instances
  - Every global and static variable is unique to an instance of a processor object.



Creating Processor Objects

- Members of a processor object must be declared:

```
global class ProcObj {
public:
   C global Var;
   f(int);
}
```

- Only global functions and global variables can be members of a processor object
- Binding of members to processor object occurs at link time.

CC++ Processor Objects

- Processor object pointers in CC++ can be manipulated like any other pointers
- Program can manage subsets of processor objects using C data structures
  - arrays, trees, etc
- Operator overloading can be used to simplify processor object management

Examples

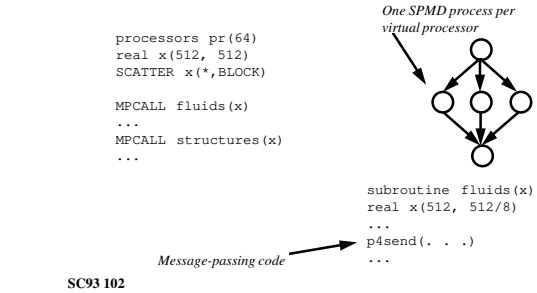
Fortran M for Paradigm Integration

- Provide an interface between tools
- We need mechanism for:
  - evoking SPMD computation
  - sharing data
  - resource management

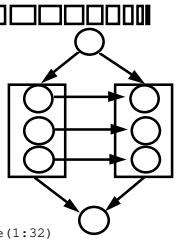
Example: Integrating Message-Passing and FM

- Goal: Support MIMD/MIMD and MIMD/SPMD integration
- Obtain modularity and portability properties of FM while reusing existing MP codes
- `MPCALL` statement initiates MP execution, with FM virtual computers controlling resource allocation
- MP computation can communicate:
  - internally, using MP statements
  - with other computations, using FM statements
  - with parent process, via arguments
- SCATTER statement allows data distribution

Example: Fork/Join Parallelism



Example: MIMD(SPMD)



```
processors pr(64)
real x(512, 512)
inport  (integer) pi(64)
outport (integer) po(64)
scatter x(:,BLOCK), pi(BLOCK), po(BLOCK)

channel(in=pi(:), out=po(:)

processes
   mpcall fluids(x, pi, po)      submachine(1:32)
   mpcall structures(y, pi, po) submachine(33:64)
endprocesses
```

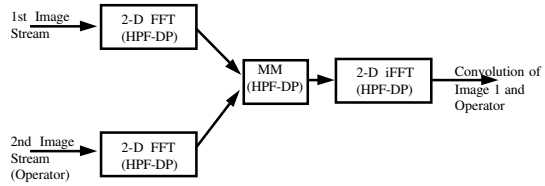
SC93 103

Example: Integrating HPF and FM

- Joint work with Bhaven Avalani and Alok Choudhary, Syracuse; and Ming Xu, Argonne
- Also involve Ken Kennedy, Chuck Koelbel, Rice
- Goal: Language constructs, compiler techniques, runtime mechanisms for paradigm integration
- Use FM as coordination language for HPF computations
  - use channels to connect HPF computations
  - declare distributed data structures in FM; pass to HPF
  - dynamic tasks, channels, data structures
- Initial focus: Integration of Syracuse HPF compiler and Argonne FM compiler

SC93 104

Convolution Algorithm



SC93 105

Data Parallel Algorithm: HPF Implementation

```
      program data_parallel
      complex A(512,512), B(512,512), C(51,512)
!HPF$ processors p(32)
!HPF$ align B, C with A
!HPF$ distribute B(*,BLOCK)
      do i = 1,nimages
         call read(A,B)
         call fft(512,512,A)
         call fft(512,512,B)
         C = A*B
         call ifft(512,512,C)
         call write(C)
       enddo
```

```
subroutine fft(m,n,A)
complex A(m,n)
...
call fft1d(m,n,A)
A = transpose(A)
call fft1d(m,n,A)
```

SC93 106

Mixed Task/Data Parallel Algorithm: FM Main Program

```
program mixed_parallel
processors p(24)
outport (complex x(512,512/8)) o1(8), o2(8)
inport  (complex x(512,512/8)) i1(8), i2(8)
channel(in=i1(:), out=o1(:))
channel(in=i2(:), out=o2(:))
processes
  hpfcall dofft(nimages, o1)      submachine(1:8)
  hpfcall dofft(nimages, o2)      submachine(9:16)
  hpfcall doifft(nimages, i1,i2) submachine(17:24)
endprocesses
```

SC93 107

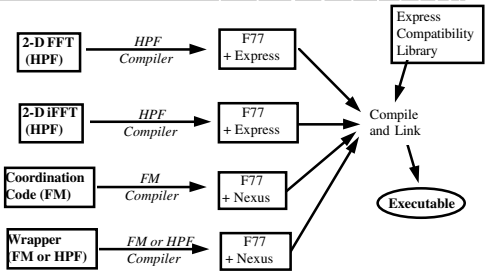
Mixed Algorithm: FFT Wrapper

```
subroutine dofft(nimages, outs)
outport (complex x(512,512)) outs
complex A(512,512)
...
do i = 1,nimages
  call initialize(A)
  call fft(512,512,A)    ! Call HPF 2-D FFT code
  send(outs) A
enddo
```

Parallel Communication Operation

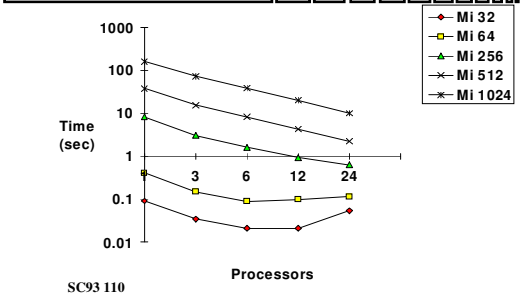
SC93 108

Mixed Algorithm: Implementation



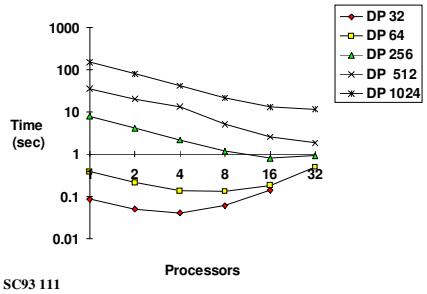
SC93 109

Convolution Performance: IBM SP1 with TCP/IP



SC93 110

Convolution Performance: IBM SP1 with TCP/IP



SC93 111

Paradigm Integration in CC++

- Paradigm libraries
  - current focus
- Language extensions
  - research direction

SC93 112

Paradigm Integration via Libraries

- CC++ provides the basic structures and glue
- Libraries provide actual functionality
- Integration through a common language framework
  - because of the types of constructs defined, libraries can interoperate with each other
- Current library efforts
  - MIMD programming libraries
  - Parallel array libraries

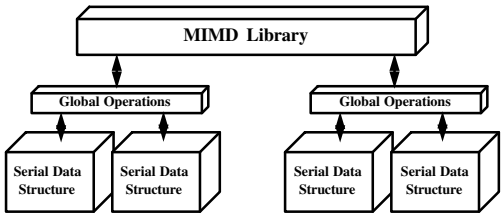
SC93 113

SPMD and MIMD Programming

- SPMD libraries built on top of CC++ constructs
  - Processor objects, not nodes.
  - Global pointers, not node numbers.
- MIMD libraries built on top of CC++ constructs
- SPMD library can be *composed* with MIMD codes
- Full generality of CC++ available to the composed program.

SC93 114

Structure of Programs Integrating with Libraries



Task Parallelism in CC++

- Message passing (channel) libraries
- Synchronization primitive libraries
  - semaphores, monitors
- Self scheduling task libraries

Message Passing in CC++

- Channel class can be defined in CC++:

```
class channel {
   public:
       send();
       receive();
   }
```

- Message passing becomes remote procedure call

```
spawn mPtr->send(X);
```

- Emulation of existing message passing routines can be layered on top of channel class:

```
class PVMchannel : channel { ...}
```

- Message libraries can be efficiently compiled.

SPMD Programming in CC++

- SPMD Programming can be achieved via a class library.
  - E.G. a distributed array class.

```
index I;
distribution d(BlockCyclic);
distributedDouble A(100,100,d);
A(I,I) = (A(I+1,I+1) + A(I+1,I-1) +
   A(I-1,I+1) + A(I-1,I-1) )/ 4;
```

  - Can be generalized to other structures such as sets.
- Since SPMD is a library, can be extended to accommodate application needs.

Distributed Arrays

- Serial component: A++
  - developed by Dan Quinlen at Los Alamos
- Global component: P++
  - developed by Dan Quinlen (University of Colorado, Denver) and Max Lempke (University of Bonn).

Index Objects

- Index objects allow specification of Fortran-90 type array operations.
- Index declaration:

```
Index i(base,length,stride);
```

- Arithmetic expressions over indexes allowed:
  - `I+1, I+J 2*I+J`
- Index is a regular C++ class so additional index types can be constructed:
  - Arbitrary collections of indexes
  - Operations over index sets

A++: A Serial Array Language

- A++ implements serial array operations via index objects
- A++ only manages operations, actual array manipulation take place in a machine dependent layer
  - machine dependent layer does not have to be in C++, e.g. CM-5 machine dependent layer is written in CMF

P++: Distributed Arrays in C++

- Based on virtual shared grids, programs have a shared name space
- Fortran-90 style array operations
- Can utilize HPF style distribution declarations
- Generates minimal amount of communication for one and two layer overlap regions

Coupling P++ and CC++

- P++ moves around array sections
  - can be directly implemented as remote procedure call
- Build P++ on top of a CC++ virtual channel library
  - Sequence of data can be placed on input of virtual channel and can be read in order from the output of the virtual channel.
  - Ends of the channel are tied to threads of control, not processors or nodes.
  - Channels are typed
- An SPMD computation consists of a collection of processor objects.
  - all coupling between computation and nodes is eliminated

Task Parallelism in A++

- A++ can *defer* evaluation of array operations

```
Expression::startDefered()
A(I) = B(I) + C(I);
D(I) = B(I) + C(I);
Espression::stopDefered()
```

- Runtime analysis of deferred expressions determines independent sets of array operations
- CC++ implementation of deferred evaluation
  - Use parfor statement
  - Arrays can be distributed with load balancing library

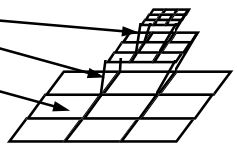
AMR++: An Example of a SPMD/MIMD Program

- AMR++ is a C++ class library written on top of P++.
- Basic mechanism is multigrid, written as an SPMD computation
- Mesh refinement adds a new (finer) mesh. Coupling is asynchronous.
- Creation, control of new meshes and interaction between meshes is ideally suited to CC++.

Example: Adaptive Mesh Refinement

Each level is an independent multigrid computation.
Each multigrid computation is written as a SPMD program.

```
par {
    sync projection x;
    sync projection y;
    solve(grid0,x);
    solve(grid1,y)
    solve(grid2,NULL);
}
```

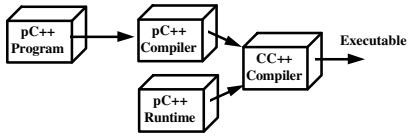


Adding SPMD Constructs to CC++

- pC++ from Indiana University
  - concurrent operations on user defined "container" classes, i.e. sets, lists, trees.
- Looking at using CC++ as pC++ compiler target
- Looking a unified language that combines CC++ and pC++
- Still at preliminary stages

SC93 127

SPMD Programming via Language Extension



SC93 128

Summary

- Mixed paradigm programming will become more important as more complex applications are developed for parallel and distributed computers.
- Modes of paradigm integration should not be restricted. This means flexible mechanisms for developing SIMD(R) and MIMD(R) programs are required.
- We have presented an approach that we feel is viable

SC93 129

Future Issues

- Can we achieve data parallel performance in integrated programs
- Do mechanisms like CC++ and Fortran M solve a reasonable spectrum of paradigm integration problems
- Development of reusable paradigm libraries
- Integration of CC++ and Fortran M
- Support tools for paradigm integration
  - debuggers, performance evaluation
- Validation of concepts through large scale application development

SC93 130