

COMP4300: Other Parallel Programming Models

Alistair Rendell

17.1 Programming Shared and Distributed Memory

Shared Memory

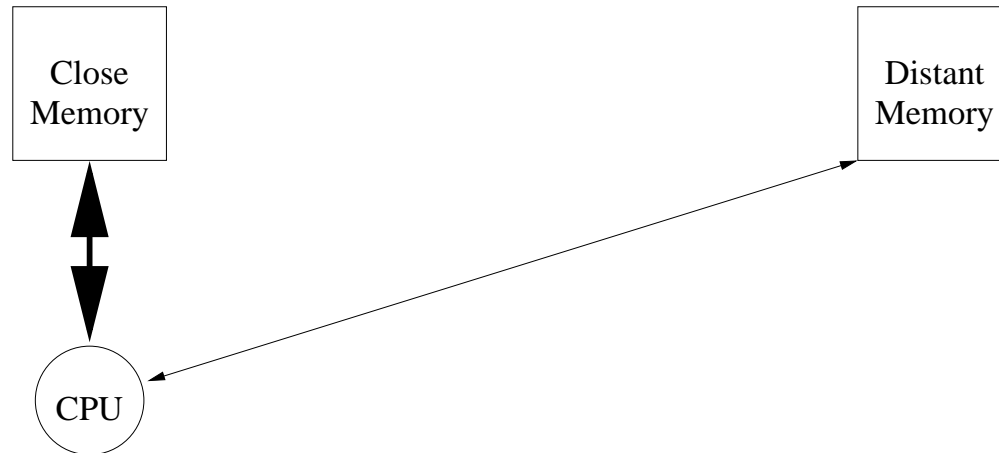
- Unix fork/shmget
- Pthreads
- OpenMP

Distributed Memory

- MPI-1 and MPI-2

Which would you rather use?

17.2 Scalable Shared Memory Architectures



Scalable parallel systems invariably include some level of NUMA

- Accept that:
 - These machines will work best with applications that exploit memory locality
 - Some extra programming effort is required to develop applications that recognise this hierarchy

17.3 Programming Trade-offs

Portability v efficiency v ease of coding

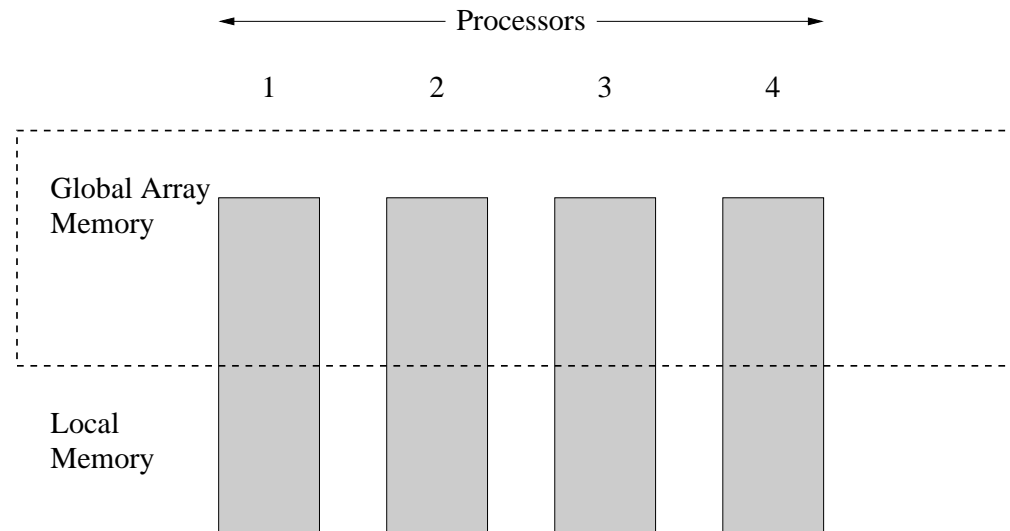
- MPI
 - Portable
 - Some applications are too complex to code while maintaining computational load balance and avoiding redundant computation
- Shared Memory
 - Simplifies coding, but until arrival of OpenMP it was not portable
 - Little control over interprocessor data transfer costs
- Global Arrays (GA), Distributed Shared Memory (DSM), and High Performance Fortran (HPF) are some of the attempts that have been made to address these issues

17.4 Global Arrays (GA)

A portable “shared-memory” programming model for distributed memory computers

- Aims to combine the better features of message passing and shared memory
- Provides a portable interface through which each process in an MIMD parallel program can independently, asynchronously and efficiently access logical blocks of physically distributed matrices, with no need for the explicit cooperation of other processors
- Similar to shared memory, but like message passing it acknowledges that remote data transfers take longer than local ones

17.5 GA Illustration



- Target applications
 - Task parallel (MIMD) maybe with additional data parallelism
 - Large distributed matrices
 - Wide variation in task execution time (load balancing)
 - Large ratio of compute to communication time

17.6 GA Programming Model

- MIMD parallelism using multiprocess approach
 - All non-GA data, file descriptors etc are replicated or unique to each process
- Communication achieved by creating and accessing GA matrices augmented (if desired) by conventional message passing
- Physically the GA matrices are blockwise distributed
- Each process can independently and asynchronously access any two-dimensional patch of a GA without cooperation between application processes
 - Access to GAs is via get, put, accumulate and get-and-increment functions
- Each process assumed to have fast access to local data and slower access to the remainder
 - Each process can determine which portion of a GA is stored locally

17.7 GA Calls

- Create (and destroy) an array controlling alignment and distribution
- Synchronize all processors
- Identify number of processors and my process ID
- Fetch, store, and atomic accumulate into a rectangular patch of an array
- Gather/scatter elements of a GA
- Atomic read and increment of an array element (shared counter)
- Inquire about location and distribution of an array
- Vector and matrix operations on entire GAs (SIMD like)

```
ga_create(mt_dbl, n, m, 'a', 10, 5, g_a)
ga_zero(g_a)
ga_put(g_a, ilo, ihi, jlo, jhi, local, ldim)
ga_acc(g_a, ilo, ihi, jlo, jhi, local, ldim, alpha)
ga_add(alpha, g_a, beta, g_b, g_c)
ga_distribution(g_a, iproc, ilo, ihi, jlo, jhi)
```

17.8 GA Implementation

- One sided communication is the key requirement
- Four strategies have been used on different machines
 - Interrupt messages/active messages (eg LAPI on the IBM SP2)
 - Shared memory primitives (eg SGI Origin)
 - Support of underlying hardware (eg Cray T3D/T3E)
 - Data server model (eg network of workstations)
- With some difficulty GA can be layer over the one-sided communication protocol of MPI-2

17.9 Data Parallel Languages (SIMD)

- Generate only a single instruction stream
- Synchronous execution of instructions (no race condition or deadlock)
- Code must explicitly specify parallelism
- Associates a virtual processor with each fundamental unit of parallelism
 - Programmer expresses computation in terms of operations performed by virtual processors
 - Programmer not concerned with number of physical processors available, they just specify how many processors they need
 - Each processor can access memory locations on any other processor creating the illusion of a shared address-space

17.10 Data Parallel Languages

- Writing data parallel programs easier than message passing
- Compilers for data parallel languages are much more complex
 - They must map virtual processors onto physical processors, generate communication, and enforce synchronous instruction execution
- Virtual processors must be mapped to real processors at some stage
 - If the number of virtual processors is greater than the number of physical processors then several processors are emulated by each physical processor
- Using virtual processor inappropriately may result in inefficient parallel programs, eg a mapping that has been implemented assuming a hypercube connectivity may not be suitable for a mesh

17.11 Data Parallel Language: High Performance Fortran(HPF)

- HPF-1 appeared in '93 and HPF-2 in '97 (very slow uptake)
 - HPC++ is a similar (maybe less advanced) effort in C++
- Primary goals
 - Support for data parallel programming
 - Top performance on MIMD and SIMD computers with NUMA
 - Code tuning for various architectures
- Secondary goals
 - Portability(existing): allow relatively easy conversion from existing code
 - Portability(new): allow efficient code on one parallel machine to be reasonably efficient on other machines
 - Compatibility: minimal change from f77 and f90
 - Simplicity
 - Interoperability: work with other languages

17.12 HPF: What's Provided

- New directives (comment lines)

```
HPF$    CHPF$    *HPF$
```

- New language syntax

```
forall (i=1:4,j=1:4) : now in f95  
pure/extrinsic      : for calls to non-HPF routines)
```

- Library routines

```
iall_scatter  : scatter an array with bitwise AND  
HPF_alignment : align an array with a target  
number_of_processors: number of processors available
```

- Language restrictions

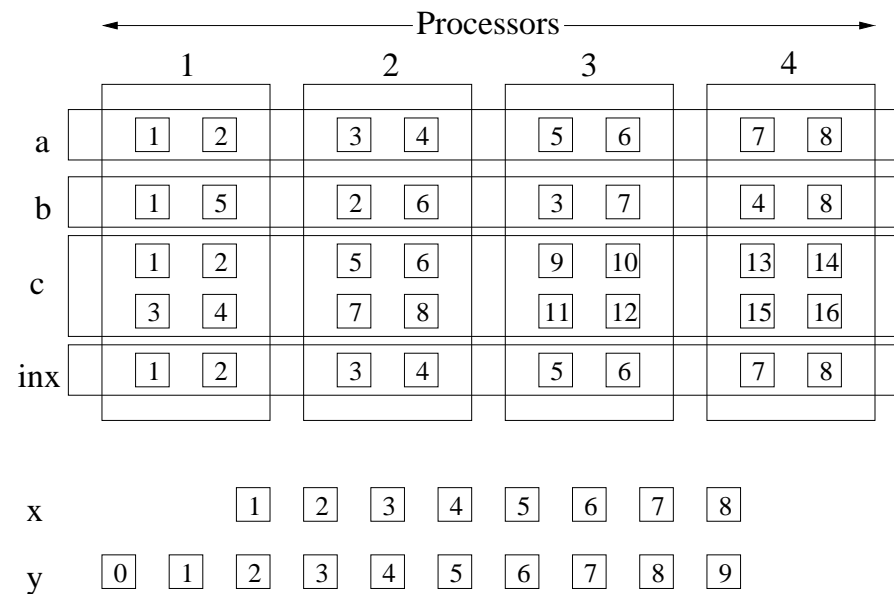
- sequence associate: mapping multidimensional arrays to 1-d
- storage association: real/integer variable to same address

17.13 HPF Example: Data Distribution

```

real, dimension(8)  ::a,b
real, dimension(16) ::c
real, dimension(8)  ::d
real, dimension(0:9) ::y
real, dimension(8) ::a,b
integer, dimension(8) :: inx
!hpf$ processors, dimension(4):: procs
!hpf$ distribute (block) onto procs :: a,c,inx
!hpf$ distribute (cyclic) onto procs :: b
!hpf$ align (i) with y(i+1) :: x

```



17.14 HPF Example: Data Alignment

- Relative alignments

```
!hpf$ template, distribute(block,block)::&
!hpf$           earth(n+1,n+1)
           real, dimension(n,n):: nw, ne, sw, se
!hpc$ align sw(i,j) with earth(i,j)
!hpc$ align nw(i,j) with earth(i,j+1)
!hpc$ align se(i,j) with earth(i+1,j)
!hpc$ align ne(i,j) with earth(i+1,j+1)
```

- Redistribution

```
           real, dimension(8) :: a
!hpf$ processors, dimension(4) :: procs
!hpf$ dynamic, distribute (block) onto procs :: a
.
.
!hpf$ redistribute (cyclic) onto procs :: a
```

- Potential alignment mismatch at subroutine interface

- As a consequence incoming data may be redistributed