FMS communication and I/O kernels

V. Balaji
SGI/GFDL

FMS Workshop
22 August 2000

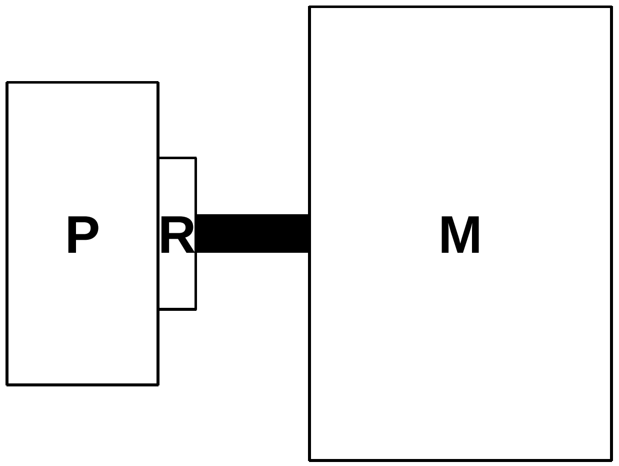
Overview

- Hardware models of parallelism
- Programming models for different architectures
- The MPP modules
- Abstract parallel numerical kernels
- Future directions

Sequential computing

The von Neumann model of computing conceptualizes the computer as consisting of a memory where instructions and data are stored, and a processing unit where the computation takes place. At each turn, we fetch an operator and its operands from memory, perform the computation, and write the results back to memory.

a = b + c



Computation speed

The speed of the computation is constrained by hardware limits: the rate at which instructions and operands can be loaded from memory, and results written back; and the speed of the processing units. The overall computation rate is limited by the slower of the two: memory.

Latency time to find a word.

Bandwidth number of words per unit time that can stream through the pipe.

Hardware trends

A processor clock period is currently $\sim$ 1 ns, growth rate is 4x/3 years.

DRAM latency is $\sim$ 50 ns, growth rate is 1.3x/3 years.

Maximum memory bandwidth is theoretically the same as the clock speed, but far less for commodity memory.

Furthermore, since memory and processors are built basically of the same “stuff”, there is no way to reverse this trend.

Within the raw physical limitations on processor and memory, there are algorithmic and architectural ways to speed up computation. Most involve doing more than one thing at once.

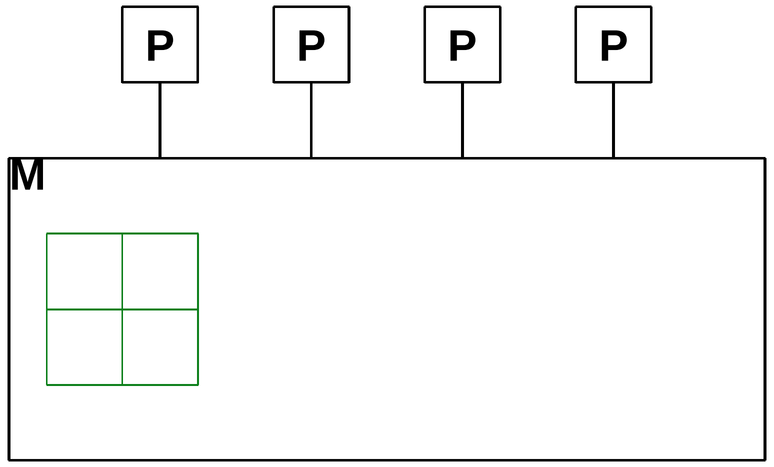
- Overlap separate computations and/or memory operations.
  - Pipelining.
  - Multiple functional units.
  - Overlap computation with memory operations.
  - Re-use already fetched information: **caching**.
- Multiple computers sharing data.

The search for **concurrency** becomes a major element in the design of algorithms.

Parallel programming models

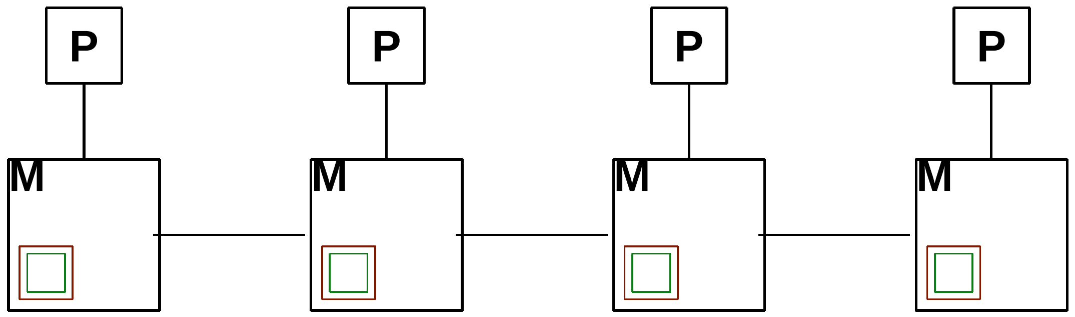
- Shared memory parallelism.
- Distributed memory parallelism.
- Multi-threading.

Shared memory parallelism



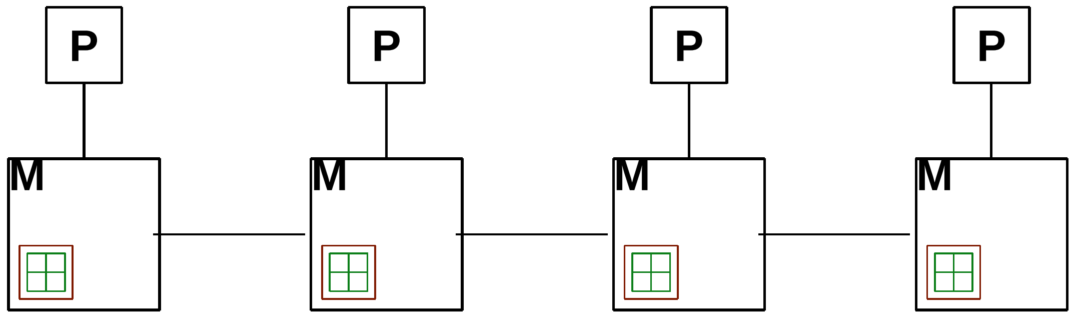
- Canonical architecture: shared memory (UMA), limited scalability.
- Private and shared variables.
- Critical regions.

Distributed memory parallelism



- Canonical architecture: distributed memory (NUMA).
- Decompose global domain `(1:I,1:J)` into `npes` subdomains. `(is:ie,js:je)` defines subdomain start and end.
- Copy data between PEs (message-passing or remote memory access).

Multi-threading



- Canonical architecture: cluster of SMPs.
- Divide global domain `(1:I,1:J)` into `nthreads*npes` threads on `npes` processors. Each processor receives `nthreads` threads.
- Each processor could also be a node on an SMP.

Computer architecture and programming models

- Memory speed will always lag processor speed.
- Shared memory will scale only so far.

Exotic new architectures (HTMT, MTA, etc) attempt various means of latency hiding. PIM attempts to reduce physical distance to memory. But physically distributed memory is a fact of life for the foreseeable future.

To deal with physically distributed memory, one must either have explicit communication (message-passing or remote memory access) or rely on compilers to do the dirty work (ccNUMA).

The MPP modules define a clean interface to various hardware models of physically distributed memory.

The MPP modules

GFDL has a homegrown parallelism API written as a set of 3 F90 modules:

- `mpp_mod` is a low-level interface to message-passing APIs (currently SHMEM and MPI; MPI-2 and Co-Array Fortran to come);
- `mpp_domains_mod` is a set of higher-level routines for domain decomposition and domain updates;
- `mpp_io_mod` is a set of routines for parallel I/O.

`http://www.gfdl.gov/˜vb`

`mpp_mod`

`mpp_mod` is a set of simple calls to provide a uniform interface to different message-passing libraries. It currently can be implemented either in the SGI/Cray native SHMEM library or in the MPI standard. Other libraries (e.g MPI-2, Co-Array Fortran) can be incorporated as the need arises.

`mpp_mod` is currently in use in all FMS models at GFDL.

mpp_mod design issues

- Simple, minimal API, with free access to underlying API for more complicated stuff.
- Design toward typical use in climate/weather CFD codes (rectilinear grid, halo update, data transpose).
- Performance to be not significantly lower than any native API.

mpp_mod API

- Basic calls:
  - `mpp_init()`
  - `mpp_exit()`
  - `mpp_transmit()`: basic message passing call. Typical use assumes *two* transmissions per domain, e.g halo update.
  - `mpp_sync()`
  - `mpp_error()`
- Reduction operators:
  - `mpp_max()`
  - `mpp_sum()`
  - etc.

Implementation of mpp_transmit

- MPI: `MPI_Isend()` and `MPI_Recv()`.
- SHMEM: `shmem_get`.
- on shared memory: direct copy.
- on ccNUMA: send address, then direct copy.

mpp_transmit performance

SHMEM implementation of mpp_transmit on T3E:

- Latency: 11 μs (3 μs for bare `shmem_get`).
- Peak bandwidth: 300 Mb/s.
- For messages longer than 1000 words, the two are not distinguishable.

Latency increase is due to code to handle dynamic arrays.

MPI bandwidth is 150 Mb/s. T90 bandwidth is 5 Gb/s.

`mpp_domains_mod` : domain class library

Definition of *domain*:

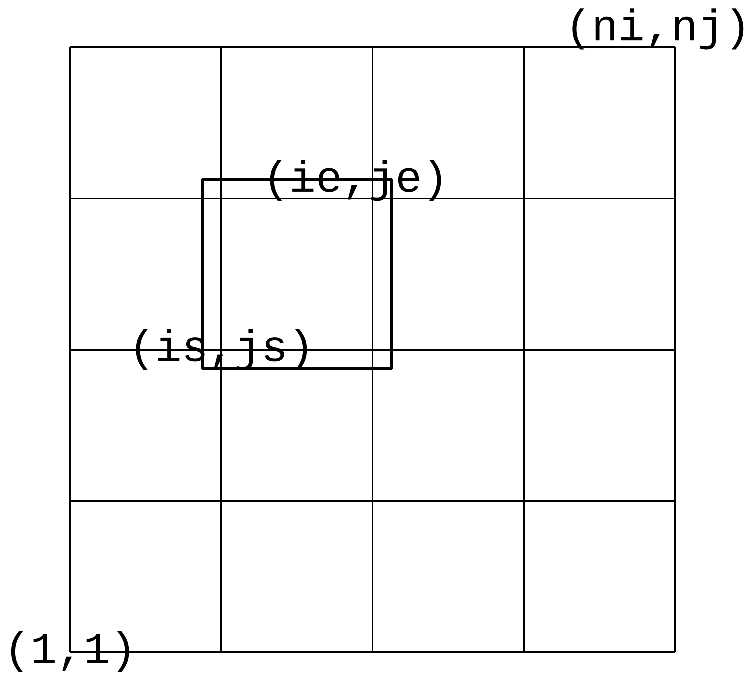
- *Global domain*: the entire model grid.
- *Compute domain*: set of points calculated by a PE.
- *Data domain*: set of points required by the computation (i.e including halo).

All the information required for domain-related operations are maintained in compact form in the *domain* types supplied by `mpp_domains_mod` . Complicated grids, such as the bipolar grid and the cubed sphere can be represented in this class, so long as they are logically rectilinear.

The *domain* type

```
type, public :: domain_axis_spec
   integer :: start_index, end_index, size, max_size
   logical :: is_global
end type domain_axis_spec
type, public :: domain1D
   type(domain_axis_spec) :: compute, data, global
   integer :: ndomains
   integer :: pe, node
   integer, dimension(:), pointer :: pelist
   type(domain1D), pointer :: prev, next
end type domain1D
```

```
!domaintypes of higher rank can be constructed from type domain1D
  type, public :: domain2D
     sequence
     type(domain1D) :: x
     type(domain1D) :: y
     integer :: pe, node
     type(domain2D), pointer :: west, east, south, north
  end type domain2D
```



The `domain2D` type contains all the necessary information to define the global, compute and data domains of each task, as well as the PE associated with the task. The PEs from which remote data may be acquired to update the data domain are also contained in a linked list of neighbours.

mpp_domains_mod calls:

- mpp_define_domains()
- mpp_update_domains()

```
type(domain2D) :: domain(0:npes-1)
call mpp_define_domains( (/1,ni,1,nj/), domain, xhalo=2, yhalo=2 )
...
!allocate f(i,j) on data domain
!compute f(i,j) on compute domain
...
call mpp_update_domains( f, domain(pe) )
```

mpp_io_mod: a parallel I/O interface

`mpp_io_mod` is a set of simple calls to simplify I/O from a parallel processing environment. It uses the domain decomposition and communication interfaces of `mpp_mod` and `mpp_domains_mod` . It is designed to deliver high-performance I/O from distributed data, in the form of self-describing files (verbose metadata).

Features of `mpp_io_mod`

- Simple, minimal API, with freedom of access to native APIs.
- Strong focus on performance of parallel write.
- Accepts netCDF format, widely used in the climate/weather community. Extensible to other formats.
- May require post-processing, generic tool for this to be provided by GFDL.
- Compact dataset (comprehensively self-describing).
- Final dataset may bear no trace of parallelism.

mpp_io_mod output modes

`mpp_io_mod` supports three types of parallel I/O:

- Single-threaded I/O: a single PE acquires all the data and writes it out.
- Multi-threaded, single-fileset I/O: many PEs write to a single file.
- Multi-threaded, multi-fileset I/O: many PEs write to independent files (requires post-processing).

mpp_io_mod API

- `mpp_io_init()`
- `mpp_open()`
- `mpp_close()`
- `mpp_read()`
- `mpp_write()`
- `mpp_write_meta()`

Metadata

Since the datasets are required to be compact (comprehensively self-describing) we associate metadata in the file header associated with each axis and field in the file. Metadata contains names and units for each variable, as well as associating each field with a number of axes. Optional attributes can be specified to describe data masks, missing data, scaling, packing, etc. These use the derived types `axistype` and `fieldtype` use associ-ated from `mpp_io_mod` .

mpp_open

The key call is `mpp_open()`. Most information about type of I/O to be performed is contained here:

```
call mpp_open( unit, file, action=MPP_WRONLY, format=MPP_IEEE32, &
      access=MPP_SEQUENTIAL, threading=MPP_SINGLE, iospec='-F cachea' )
```

Format can be one of `MPP_ASCII, MPP_IEEE32, MPP_NATIVE`, or `MPP_NETCDF`.

Single-threaded I/O from multiple PEs means PE0 will acquire all the data and do the actual write.

Multi-threaded I/O

```
call mpp_open( unit, file, action=MPP_WRONLY, format=MPP_IEEE32,
      access=MPP_SEQUENTIAL, threading=MPP_MULTI, fileset=MPP_MUL
```

Multi-threaded I/O can have all PEs write to a single file or each to an independent file, which must later be assembled (a generic tool for this is available). It offers the possibility of high-performance I/O when parallel filesystems are buggy or slow.

mpp_io_mod calling sequence

```
type(domain2D) :: domain(0:npes-1)
type(axistype) :: x, y, z, t
type(fieldtype) :: field
integer :: unit
character*(*) :: file
real, allocatable :: f(:,:,:)
call mpp_define_domains( (/1,ni,1,nj/), domain )
call mpp_open( unit, file, action=MPP_WRONLY, format=MPP_IEEE32, &
  access=MPP_SEQUENTIAL, threading=MPP_SINGLE )
call mpp_write_meta( unit, x, 'X', 'km', ... )
...
call mpp_write_meta( unit, field, (/x,y,z,t/), 'Temperature', 'kelvin', ... )
...
call mpp_write( unit, field, domain(pe), f, tstamp )
```

Parallel numerical kernels

$$\frac{\eta^{n+1} - \eta^n}{\Delta t} = -H(\nabla \cdot \mathbf{u})^n \quad (1)$$

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = -g(\nabla\eta)^n + f\mathbf{k} \times \left(\frac{\mathbf{u}^{n+1} + \mathbf{u}^n}{2}\right) + \mathbf{F} \quad (2)$$

```
program shallow_water
  type(scalar2D) :: eta(0:1)
  type(hvector2D) :: utmp, u, forcing
  integer tau=0, taup1=1
...
  f2 = 1./(1.+dt*dt*f*f)
  do l = 1,nt
     eta(taup1) = eta(tau) - (dt*h)*div(u)
     utmp = u - (dt*g)*grad(eta(tau)) + (dt*f)*kcross(u) + dt*forcing
     u = f2*( utmp + (dt*f)*kcross(utmp) )
     tau   = 1 - tau
     taup1 = 1 - taup1
  end do
end program shallow_water
```

- Runs and reproduces answers on t90, t3e, SGI, Beowulf.
- No parallel calls.
- Memory scaling (except for halo region overhead).
- 400 Mflops, 800 Mmops, on t90 125×125.
- 80% scaling on 5×5 PEs on t3e.
- Abstraction penalty about 20% on MOM 2p.
- Standard f90 (Cray, SGI, PGF90...)

The distributed grid class

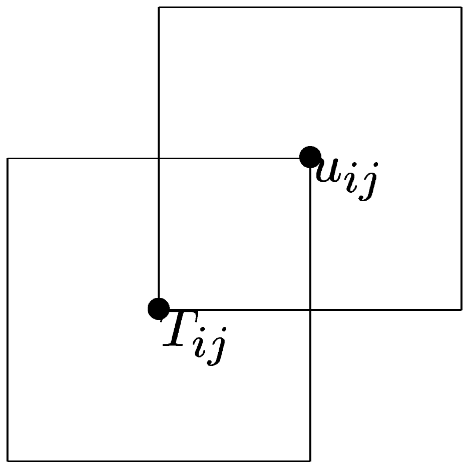
```
module distributed_grids
  use mpp_domains_mod
  implicit none
  private
  type, public :: scalar2D
     real, pointer :: data(:,:)
     integer :: is, ie, js, je
  end type scalar2D
  type, public :: hvector2D
     type(scalar2D) :: x, y
     integer :: is, ie, js, je
  end type hvector2D
```

Abstract difference operators

$$\nabla \cdot \mathbf{u} = \delta_x(\overline{u}^y) + \delta_y(\overline{v}^x) \tag{3}$$

$$(\nabla T)_x = \delta_x(\overline{T}^y) \tag{4}$$

$$(\nabla T)_y = \delta_y(\overline{T}^x) \tag{5}$$



```
    function grad_scalar2D(scalar)
      type(hvector2D) :: grad_scalar2D
      type(scalar2D), intent(inout) :: scalar
...
      if( scalar%ie.LE.ie .OR. scalar%je.LE.je )then
          call mpp_update_domains( scalar%data, domain, EUPDATE+NUPDATE )
          scalar%ie = ied
          scalar%je = jed
      end if
      grad_scalar2D%is = scalar%is; grad_scalar2D%ie = scalar%ie - 1
      grad_scalar2D%js = scalar%js; grad_scalar2D%je = scalar%je - 1

!dir$ IVDEP
      do j = grad_scalar2D%js,grad_scalar2D%je
         do i = grad_scalar2D%is,grad_scalar2D%ie
            tmp1 = scalar%data(i+1,j+1) - scalar%data(i,j)
            tmp2 = scalar%data(i+1,j) - scalar%data(i,j+1)
            work2D(i,j,nbuf2) = gradx(i,j)*( tmp1 + tmp2 )
            work2D(i,j,nbufy) = grady(i,j)*( tmp1 - tmp2 )
         end do
      end do
```

Features of differencing operators

- Details of numerics are hidden from high-level code.
- Highly optimized numerical kernels without sacrificing readability.
- Extensible: can overload different algorithms as required or desired.
- Grid metrics are set once, at initialization.
- Update domains only as required, with no user intervention, including one-sided updates.
- Builtin use of *wide halos* for balancing computation with communication.

Wide halos

On a machine with a slow interconnect, we can choose to replace communication by redundant computation:

- Points in the active domain may be computed on more than one PE.
- Active domain is reduced until there are not enough points left to update the computational domain.
- Then update halos. This may only occur once every several timesteps.

```
call mpp_update_domains( ..., xhalo=1, yhalo=1 )
call mpp_update_domains( ..., xhalo=6, yhalo=6 )
```

MPP modules: summary

- Minimalist approach: directly use underlying APIs for more complex requirements.
- Uniform interface to various software expressions and hardware models of parallelism through an abstract representation of a domain.
- Specialized to typical weather/climate codes (logically rectilinear grids, permuted communication, halo update, data transpose).
- Simple, modular, extensible and debuggable by “ordinary users”.
- Simple, extensible parallel I/O interface, with focus on parallel writes.

Future directions

- Implementation of abstract parallel numerical kernels in FMS.
- Multi-threading.
- Asynchronous model coupling.
- Dynamic load balancing.
- Development of adjoint operators.
- Possible future needs:
  - Code-controlled checkpointing.
  - Unstructured grids.
  - Distributed computing.