Evolution of standards in modeling software

V. Balaji
SGI/GFDL Princeton University

4th International Workshop on High-End Climate Modeling
National Center for Atmospheric Research
Boulder, Colorado
12 March 2002

GFDL Computing

- Reliance on Cray vector architecture in previous decades.
- Transition to scalable computing begun in 1997 with the acquisition of Cray T3E.
- Current computing capability: $2 \times 256 + 6 \times 128 + 2 \times 64$p Origin 3000.

Technological trends

In climate research... increased emphasis on detailed representation of individual physical processes governing the climate; requires many teams of specialists to be able to contribute components to an overall coupled system;

In computing technology... increase in hardware and software complexity in high-performance computing, as we shift toward the use of scalable computing architectures.

The GFDL response: modernization of modeling software

- Abstraction of underlying hardware to provide *uniform programming model* across vector, uniprocessor and scalable architectures;

- Distributed development model: many contributing authors. Use high-level abstract language features to facilitate development process;

- Modular design for interchangeable dynamical cores and physical parameterizations, development of *community-wide standards* for components.

FMS: the GFDL Flexible Modeling System

Jeff Anderson, V. Balaji, Matt Harrison, Isaac Held, Paul Kushner, Ron Pacanowski, Pete Phillipps, Bruce Wyman, ...

- Develop high-performance kernels for the numerical algorithms underlying non-linear flow and physical processes in complex fluids;
- Maintain high-level code structure needed to harness component models and representations of climate subsystems developed by independent groups of researchers;
- Establish standards, and provide a shared software infrastructure implementing those standards, for the construction of climate models and model components portable across a variety of scalable architectures.
- Benchmarked on a wide variety of high-end computing systems;
- Run in production on very different architectures: parallel vector (PVP), distributed massively-parallel (MPP) and distributed shared-memory (NUMA).

FMS shared infrastructure: machine and grid layers

MPP modules communication kernels, domain decomposition and update, parallel I/O.

Time and calendar manager tracking of model time, scheduling of events based on model time.

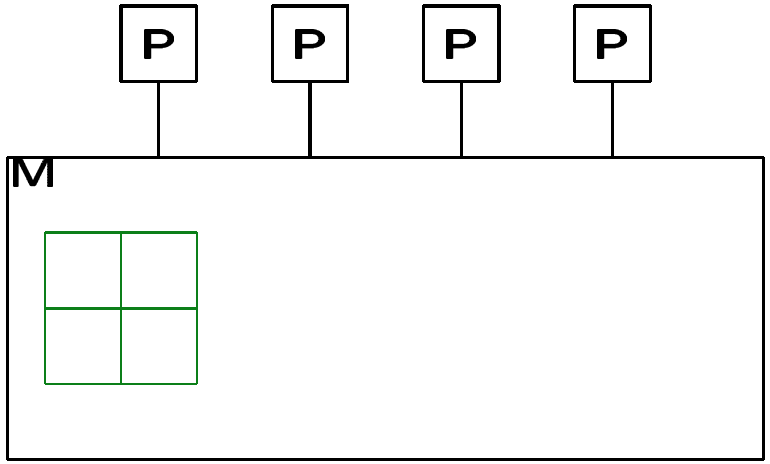
Diagnostics manager Runtime output of model fields.

Scientific libraries Uniform interface to proprietary and open scientific library routines.

Parallel programming models

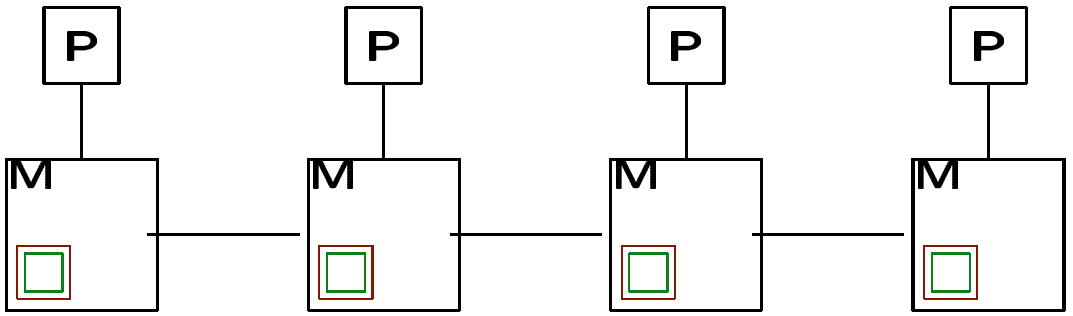
- Shared memory parallelism.
- Distributed memory parallelism.
- Hybrid parallelism.

Shared memory parallelism



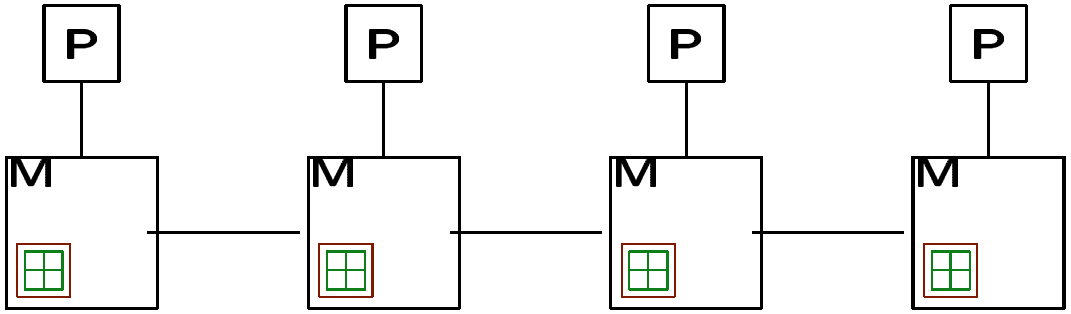
- Canonical architecture: shared memory (UMA), limited scalability.
- Private and shared variables.
- Critical regions.

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).

Hybrid parallelism



- 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 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

```
call mpp_transmit( send_buf, n, to_pe, recv_buf, m, from_pe )
```

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

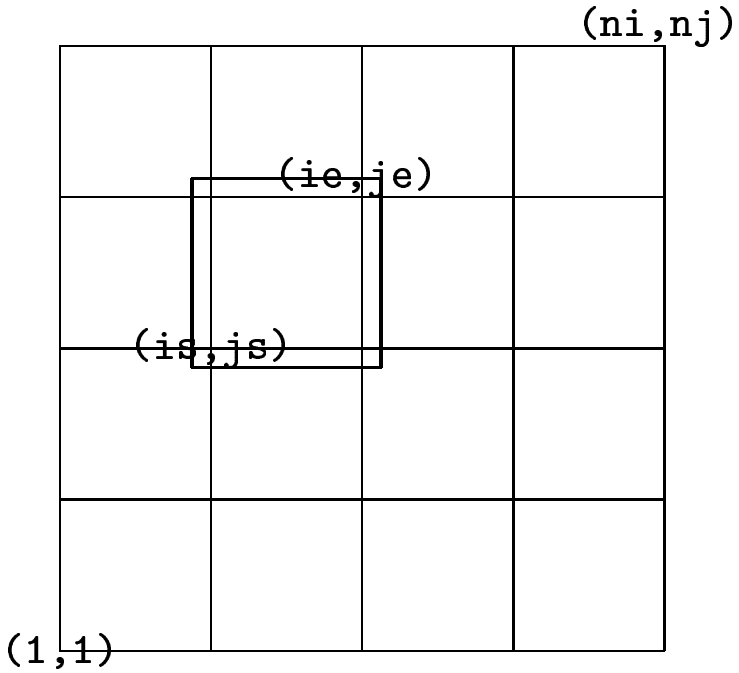
`mpp_domains_mod` : domain class library

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



mpp_domains_mod calls:

- mpp_define_domains()
- mpp_update_domains()

```
type(domain2D) :: domain
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 )
```

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 bears 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()`

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 )
```

Summary

- It is possible to write a data-sharing layer spanning flat shared memory, distributed memory, ccNUMA, cluster-of-SMPs. The API is not as extensive as, say, MPI, but has been designed to serve the climate/weather modeling community.

- It is possible to write another layer that expresses these operations in a manner natural to our algorithms ("halo update", "data transpose" instead of "buffered send", "thread nesting").

- The current standardization efforts (ESMF, PRISM) departs from BLAS, MPI, etc in that they are xplicitly formulated in high-level language constructs (classes, modules, types).

- The HPC industry must be actively involved if this is to be a success.