Introduction to Parallel Algorithms

V. Balaji
SGI/GFDL

GFDL Princeton University
6 October 1998

Overview

- Review of parallel architectures and computing models
- Issues in message-passing algorithm performance
- Advection equation
- Elliptic equation
- Conclusions

Technology trends

A processor clock period is currently $\sim$ 2-4 ns, Moore's constant is $4\times/3$ years.

DRAM latency is $\sim$ 60 ns, Moore's constant is $1.3\times/3$ 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.

Concurrency

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.

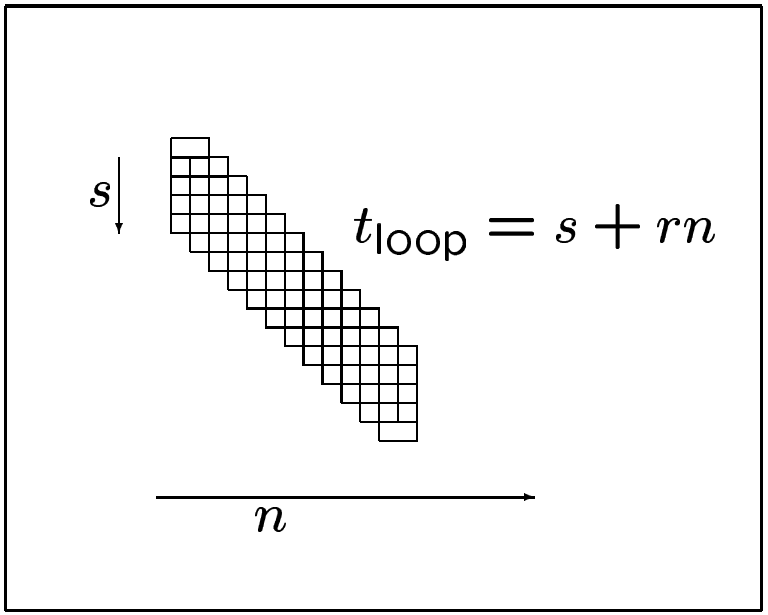
- Overlap separate computations and/or memory operations.
  - Pipelining.
  - Multiple functional units.
  - Overlap computation with memory operations.
  - Re-use already fetched information: **caching**.
  - Memory pipelining.
- Multiple computers sharing data.

The search for **concurrency** becomes a major element in the design of algorithms (and libraries, and compilers).

Vector computing

Seymour Cray: if the same operation is *independently* performed on many different operands, schedule the operands to stream through the processing unit at a rate r = 1 per CP. Thus was born **vector processing**.

```
do i = 1,n
   a(i) = b(i) + c(i)
enddo
```



Instruction-level parallelism

This is also based on the pipelining idea, but instead of performing the same operation on a vector of operands, we perform *different* operations simultaneously on different data streams.

```
a = b + c
d = e * f
```

The onus is on the compiler to detect ILP.

In the multiple-processor model, processors may have access to a single global memory pool, or independent memory units.

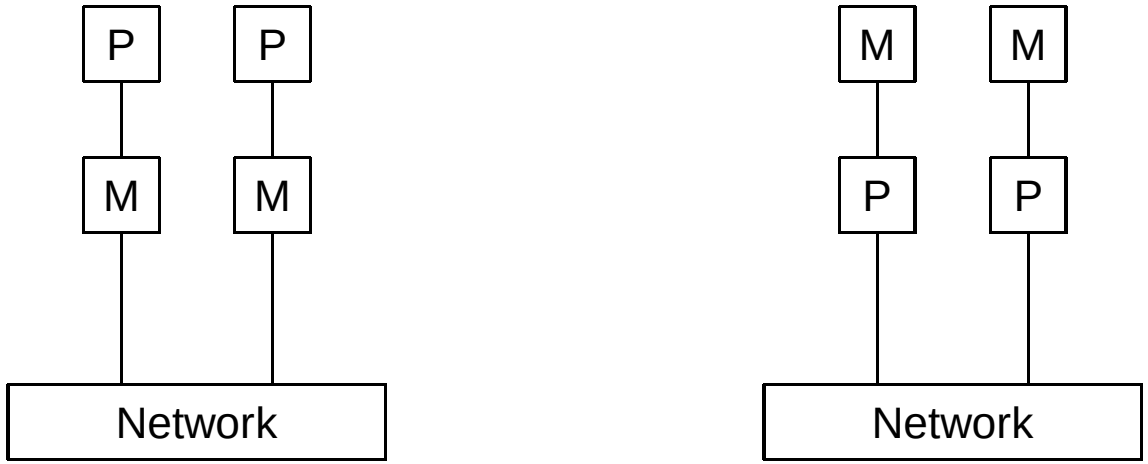
UMA architectures suffer from a crisis of aggregate memory bandwidth. The use of caches may alleviate the bandwidth problem, but requires some form of communication between the disjoint members of the system: processors or caches.

In the NUMA model memory segments are themselves distributed and communicate over a network. This involves a radical change to the programming model, since there is no longer a single **address space** in it. Instead communication between disjoint regions must be explicit: **message passing.**

More recently, with the advent of fast cache-coherency techniques, the single-address-space programming model has been revived within the **cc-NUMA** architectural model. Here memory is *physically* distributed, but *logically* shared. Since it still involves message-passing (though perhaps hidden from the user), message-passing is still the correct lens through which to view its performance.

Distributed Memory Systems: MPP

Processing elements (PEs), consisting of a processor *and* memory, are distributed across a network, and exchange data only as required, by explicit send and receive.



Tightly coupled systems: memory closer to network than processor.
Loosely coupled systems: processor closer to network than memory.

Loose coupling could include heterogeneous computing across a LAN/WAN/Internet.

A programming model for MPPs

The model we will be looking at here consists of:

- *Private*, as opposed to shared, memory organization.
- *Local*, as opposed to global, address space.
- *Non-uniform*, as opposed to uniform, memory access.
- *Domain decomposition*, as opposed to functional decomposition.

Parallel programming model

A *task* is a sequential (or vector) program running on one processor using local memory.

A parallel computation consists of two more tasks executing concurrently.

Execution does not depend on particular assignment of tasks to processors. (More than one task may belong to a processor.)

Tasks requiring data from each other need to synchronize and exchange data as required. (We do not consider *embarrassingly parallel* problems here, where there is no need for synchronization and data exchange.)

Partitioning

Issues to consider in partitioning the problem into tasks:

- Data layout in memory.
- Cost of communication.
- Synchronization overhead.
- Load balance.

Communication model

A *message* consists of a block of data *contiguously laid out in memory*.

Communication consists of an asynchronous `send()` and a synchronous `recv()` of a message. In loosely-coupled systems, PEs need to negotiate the communication, thus both a `send()` and a `recv()` are required. In tightly-coupled systems we can have a pure `send()` (`put()`) and a pure `recv()` (`get()`). The onus is on the user to ensure synchronization.

Communication costs: *latency* and *bandwidth*.

$$t_t = t_s + Lt_w \tag{1}$$

t_s can include *software latency* (cost of negotiating a two-sided transmission, gathering non-contiguous data from memory into a single message).

Note that we have considered t_t as being independent of inter-processor distance (generally well-verified).

On T3E:

- SMA latency is essentially 0, bandwidth 350 Mb/s.
- MPI latency 500 μsec, bandwidth 150 Mb/s.

Global reduction

Sum the value of a on all PEs, every PE to have a copy of the result.
Simplest algorithm: gather on PE 0, sum and broadcast.

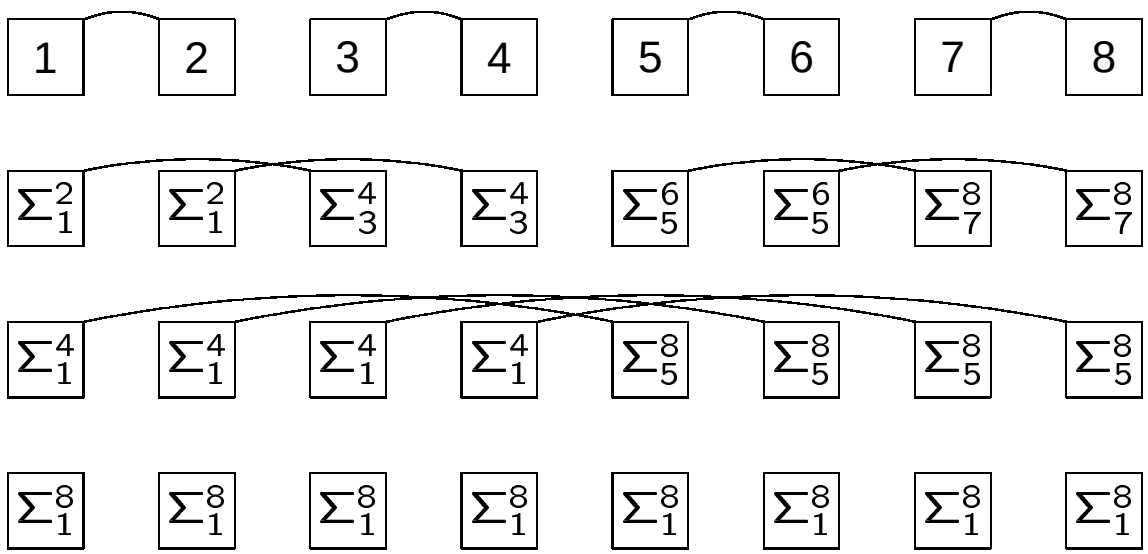
```
program test
real :: a, sum
a = my_pe()
call BARRIER()
if( my_pe().EQ.0 )then
    sum = a
    do n = 1,num_pes()-1
       call SHMEM_GET( a, a, 1, n )
       sum = sum + a
    enddo
    do n = 1,num_pes()-1
       call SHMEM_PUT( sum, sum, 1, n )
    enddo
endif
call BARRIER()
print *, 'sum=', sum, ' on PE', my_pe()
end
```

Same code in MPI:

```
program test
real :: a, sum
a = my_pe()
if( pe.NE.0 )call MPI_ISEND( a, 1, ..., 0, ... )
if( pe.EQ.0 )then
    sum = a
    do n = 1,num_pes()-1
       call MPI_RECV( a, 1, ..., n, ... )
       sum = sum + a
    enddo
    do n = 1,num_pes()-1
       call MPI_ISEND( a, 1, ..., 0, ... )
    enddo
endif
if( pe.NE.0 )call MPI_RECV( a, 1, ..., 0, ... )
print *, 'sum=', sum, ' on PE', my_pe()
end
```

This algorithm on p processors involves $2(p-1)$ communications and p summations, all sequential.

Here's another algorithm for doing the same thing: a binary tree. It executes in $\log_2 p$ steps, each step consisting of one communication and one summation.



There are two ways to perform each step:

```
if( mod(pe,2).EQ.0 )then !execute on even-numbered PEs
    call SHMEM_GET( a, sum, 1, pe+1 )
    sum = sum + a
    call SHMEM_PUT( sum, sum, 1, pe+1 )
endif

if( mod(pe,2).EQ.0 )then !execute on even-numbered PEs
    call SHMEM_GET( a, sum, 1, pe+1 )
    sum = sum + a
else                      !execute on odd-numbered PEs
    call SHMEM_GET( a, sum, 1, pe-1 )
    sum = sum + a
endif
```

The second is faster, even though a redundant computation is performed.

```
do level = 0,lognpes-1    !level on tree
   pedist = 2**level      !distance to sum over
   b(:) = a(:)            !initialize b for each level of the tree
   call mpp_sync()
   if( mod(pe,pedist*2).GE.pedist )then
       call mpp_transmit( b, c, size(b), pe-pedist, pe-pedist )
       a(:) = c(:) + b(:) !if c came from the left,  sum on the left
   else
       call mpp_transmit( b, c, size(b), pe+pedist, pe+pedist )
       a(:) = a(:) + c(:) !if c came from the right, sum on the right
   endif
enddo
call mpp_sync()
```

This algorithm performs the summation and distribution in $\log_2 p$ steps.

In general it is better to avoid designating certain PEs for certain tasks. Not only is a better work distribution likely to be available, it can be dangerous code:

```
if( pe.EQ.0 )call mpp_sync()
```

While this is not necessarily incorrect (you could have

```
if( pe.NE.0 )call mpp_sync()
```

further down in the code), it is easy to go wrong.

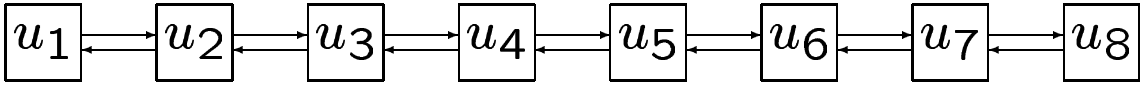
Advection equation

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{2}$$

In discrete form:

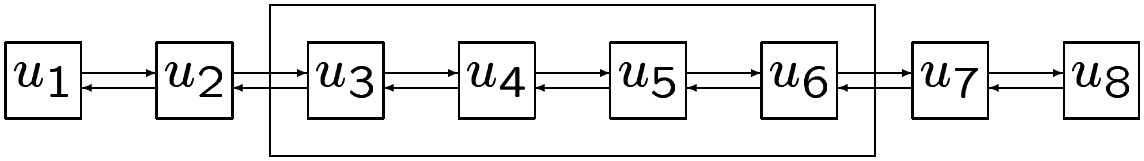
$$u_i^{n+1} = u_i^n + c\frac{\Delta t}{2\Delta x}\left(u_{i+1}^n - u_{i-1}^n\right) \tag{3}$$

Assume $P < N$, and that P is an exact divisor of N.



Assign each u_i to a task. Assign each task by rotation to a different processor (*round-robin* allocation).

```
real :: u(1:N)
do i = 1,N
   if( my_pe().NE.pe(i) )cycle
!pass left,  send u(i) to task i-1, receive u(i+1) from task i+1
   call mpp_transmit( u(i), u(i+1), 1, pe(i-1), pe(i+1) )
!pass right, send u(i) to task i+1, receive u(i-1) from task i-1
   call mpp_transmit( u(i), u(i-1), 1, pe(i+1), pe(i-1) )
   u(i) = u(i) + a*( u(i+1)-u(i-1) )
enddo
```



We could also choose to assign N/P adjacent tasks to the same PE (*block* allocation).

```
real :: u(l-1:r+1)
!pass left,  send u(l) to task l-1, receive u(r+1) from task r+1
   call mpp_transmit( u(l), u(r+1), 1, pe(l-1), pe(r+1) )
!pass right, send u(r) to task r+1, receive u(l-1) from task l-1
   call mpp_transmit( u(r), u(l-1), 1, pe(r+1), pe(l-1) )
   do i = l,r
      u(i) = u(i) + a*( u(i+1)-u(i-1) )
   enddo
```

Communication is vastly reduced.

```
!pass left,  send u(l) to task l-1
   call mpp_transmit( u(l), u(r+1), 1, pe(l-1), NULL_PE )
!pass right, send u(i) to task i+1
   call mpp_transmit( u(r), u(l-1), 1, pe(r+1), NULL_PE )
   do i = l+1,r-1
      u(i) = u(i) + a*( u(i+1)-u(i-1) )
   enddo
!pass left,  receive u(r+1) from task r+1
   call mpp_transmit( u(l), u(r+1), 1, NULL_PE, pe(r+1) )
!pass right, receive u(l-1) from task l-1
   call mpp_transmit( u(r), u(l-1), 1, NULL_PE, pe(l-1) )
   u(l) = u(l) + a*( u(l+1)-u(l-1) )
   u(r) = u(r) + a*( u(r+1)-u(r-1) )
```

The effective communication cost must be measured from the end of the do loop.

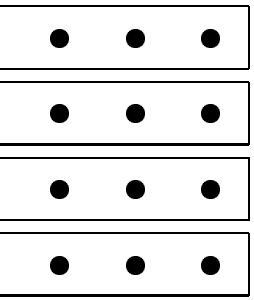
The **computational domain** is the set of gridpoints that are computed on a domain. The **data domain** is the set of gridpoints that needs to be available on-processor to carry out the computation.

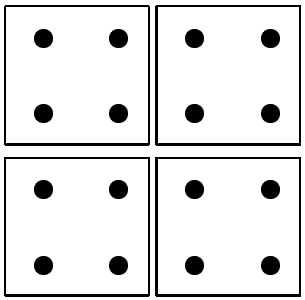
The data domain may consist of a **halo** of a certain width, or it might be global along an axis (e.g polar filter).

Computational domain: `u(l:r)`. Data domain: `u(l-1:r+1)`.

The halo region could be 2 wide, e.g. 4th-order advection.

Domain decomposition in 2D





There are different ways to partition $N \times N$ points onto P processors.

```
call mpp_define_domains( (/1,N,1,N/), domains(1:P), XGLOBAL, margins=(/0,1/) )
call mpp_define_domains( (/1,N,1,N/), domains(1:P),          margins=(/1,1/) )
```

1D or 2D decomposition?

Assume a problem size $N \times N \times K$, with a halo width of 1.

Cost per timestep with no decomposition:

$$t_0 = N^2 K t_c \tag{4}$$

Cost per timestep with 1D decomposition ($N \times \frac{N}{P} \times K$):

$$t_{1D} = \frac{N^2 K}{P} t_c + 2t_s + 4NKt_w \tag{5}$$

Cost per timestep with 2D decomposition ($\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}} \times K$):

$$t_{2D} = \frac{N^2 K}{P} t_c + 4t_s + \frac{8NK}{\sqrt{P}} t_w \tag{6}$$

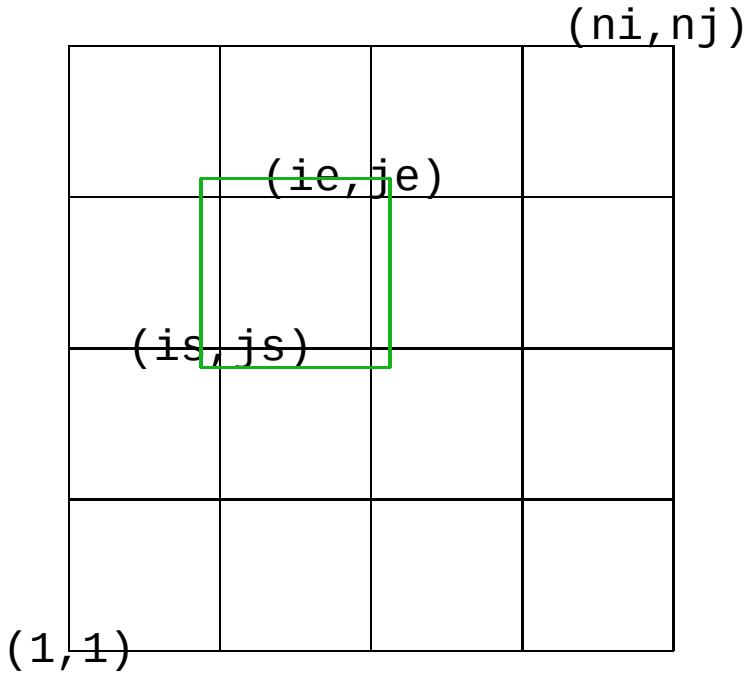
In the limit of asymptotic N, P (maintaining constant N^2/P), $t_{2D} \ll t_{1D}$.

The case for 2D decomposition is the argument that the communication to computation ratio is like a surface to volume ratio, which goes as $1/r$.

The flaw in this argument: outside the textbooks, only DoE is in the limit of asymptotic N and P!

For modest levels of parallelism, and realistic problem sizes, the additional cost incurred in software complexity is often hard to justify.

A second flaw, also serious, is that there is higher “software latency” associated with 2D halo exchanges, to gather non-contiguous data from memory into a single message.



Data layout in memory is a crucial issue (also for effective utilization of cache). The best method is to lay out data in memory to facilitate message-passing. In the limit, a $N \times N \times K$ problem could be partitioned into N^2 domains (tasks, columns), which are distributed over P processors:

```
call mpp_define_domains( (/1,N,1,N/), domains(1:N2), XGLOBAL, margins=(/0,1/)
```

The additional complexity is that rank of all arrays is increased by 1 when NDOMAINS exceeds NPES:

```
do n = 1, ndomains
   if( pe.NE.pe_of_domain(n) )cycle
   do k = 1,nk
      do j = js,je
         do i = is,ie
            u(i,j,k,n) = ...
         enddo
      enddo
   enddo
enddo
```

In this model, both cache and vector optimization are possible, by varying NDOMAINS.

Another feature of this style is that it permits the implementation of a global task queue:

```
do n = 1, ndomains
   if( .NOT.queued(n) )cycle
   unqueue(n)
   do k = 1,nk
      do j = js,je
         do i = is,ie
            u(i,j,k,n) = ...
         enddo
      enddo
   enddo
enddo
```

Global task queues (a shared-memory approach) is a potent solution to load balance problems when NDOMAINS $\gg$ NPES.

Elliptic equations

Consider a 2D Poisson equation:

$$\nabla^2 u(x, y) = f(x, y) \tag{7}$$

The solution at any point to a boundary value problem in general depends on all other points, therefore incurring a high communication cost on distributed systems.

Jacobi iteration

$$u_{ij}^{(n+1)} = \frac{1}{4}\left(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)} + u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)} - \Delta x^2 f_{ij}\right) \quad (8)$$

Iterate until $|u_{ij}^{(n+1)} - u_{ij}^{(n)}| < \epsilon$.

This method converges under known conditions, but convergence is slow.

Gauss-Seidel iteration

Update values on RHS as they become available:

$$u_{ij}^{(n+1)} = \frac{1}{4}\left(u_{i-1,j}^{(n+1)} + u_{i+1,j}^{(n+1)} + u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)} - \Delta x^2 f_{ij}\right) \quad (9)$$

Iterate until $|u_{ij}^{(n+1)} - u_{ij}^{(n)}| < \epsilon$.

This method converges faster, but contains data dependencies that inhibit parallelization.

```
!receive halo from south and west
   call recv(...)
   call recv(...)
!do computation
   do j = js,je
      do i = is,ie
         u(i,j) = u(i-1,j)+u(i,j-1)+u(i+1,j)+u(i,j+1)-a*f(i,j)
      enddo
   enddo
!pass halo to north and west
   call send(...)
   call send(...)
```

4	5	6	7
3	4	5	6
2	3	4	5
1	2	3	4

Red-Black Gauss-Seidel method

```
do parity = red,black
   if( parity.NE.my_parity(pe) )cycle
!receive halo from south and west
   call recv(...)
   call recv(...)
!do red domains on odd passes, black domains on even passes
   do j = js,je
      do i = is,ie
         u(i,j) = u(i-1,j)+u(i,j-1)+u(i+1,j)+u(i,j+1)-a*f(i,j)
      enddo
   enddo
!pass halo to north and west
   call send(...)
   call send(...)
enddo
```

2	1	2	1
1	2	1	2
2	1	2	1
1	2	1	2

More sophisticated methods of hastening the convergence are generally hard to parallelize. The conjugate gradient method accelerates this by computing at each step the optimal vector in phase space along which to converge. Unfortunately, computing the direction involves a global reduction.

In summary, if there are alternatives to solving an elliptic equation over distributed data, they should be given very serious consideration.

Conclusions

Considerations in implementing parallel algorithms:

- Uniformity of workload. Designated PEs for some operations can be useful in certain circumstances, but in general a better division of work can probably be found.

- Design data layout in memory to facilitate message passing and cache behaviour. Sometimes redundantly computing data on all PEs is preferable to communication.

- Be wary of asymptotic scalability theory. The cost of achieving maximal parallelism often includes a considerable complexity burden, with dubious returns at modest levels of parallelism.

An F90 module for domain decomposition and message passing is available for anyone who wishes to try it. It can accommodate most useful features of both SMA and MPI-style message-passing, as well as versatile decomposition, within a succinct set of calls:

```
mpp_init()
mpp_define_domains()
mpp_update_domains()
mpp_sum()
mpp_transmit(), mpp_transmit_dyn()
mpp_sync()
mpp_close()
```

A comment on elliptic equations

Low order models benefit from being formulated in terms of a series of balance assumptions that reduce the number of prognostic equations. In the limit, atmospheric and oceanic dynamics could in principle be formulated in terms of a single prognostic variable, the *potential vorticity*, and a balance model that allows us to recover the mass and momentum fields from it. This would lead to a single parabolic equation to step forward in time, and several elliptic equations to recover the other variables.

As we move to higher resolutions, it becomes less easy to justify balance models, and models tend to solve more independent prognostic equations.

Happily, these are also the algorithms that lend themselves best to parallel formulation.

More conclusions

- Message-passing is a good model to understand distributed shared memory as well. This is not to say that there are no algorithms that can be devised that will perform better on shared memory machines, but these involve taking advantage of complex protocols for memory management, and can be machine and implementation-specific.

- T3E is a good learning machine. Predictable performance, clean message-passing programming interface.