DISTRIBUTED AND HIGH-PERFORMANCE COMPUTING

CASE STUDY - MONTE CARLO SIMULATION OF THE ISING MODEL

Paul Coddington

Department of Computer Science
University of Adelaide
Room 1052
http://www.dhpc.adelaide.edu.au
paulc@cs.adelaide.edu.au

July - October 2000

CASE STUDY: MONTE CARLO SIMULATION OF THE ISING MODEL

Case Study in Computational Physics

This case study will explore a typical example of using a high-performance parallel computer to study a scientific problem.

- Start with an interesting physical problem that a scientist wants to study numerically.
- Need a mathematical formulation of the problem, and an algorithm to solve it.
- Often find that compute requirements are infeasibly large, so need approximations such as use simplifying assumptions, smaller problem sizes, etc. Try to show that errors are small enough to give realistic results.
- Need a parallel version of the algorithm to run on high-performance computers - may be more than one possible parallel algorithm.
- Need to select a parallel programming language to implement the algorithms - must weigh tradeoffs in portability, ease of programming, performance, available (or future) computing facilities, etc.
- Write, debug and test the code.
- Performance analysis of speedup and efficiency, optimization of code.
- Production runs on HPCs to extract scientific results.

Physical Problem – Magnetism

Magnetism occurs when unpaired electron spins in different atoms or molecules couple and align.

The sum of the magnetic fields generated by each individual electron spin produces macroscopic magnetism.

Many magnetic crystals have a *phase transition* from a magnetized (ordered) state to a demagnetized (disordered) state.

This is analogous to the phase transition from ice to water, or water to steam.

low temperature ⇒ order
→ alignment of spins
→ large magnetization

high temperature ⇒ disorder (thermal fluctuations)
→ spins in random directions
→ spins and fields cancel
→ no magnetization

Turns out that interesting details of phase transitions depend only on very basic details of the model (e.g. dimension and symmetry).

So very simple mathematical models can be used to study magnetism, and phase transitions in general.

The Ising Model

The Ising Model is the simplest example of a *spin model* of magnetism, which use a crystalline lattice (regular grid) of spins.

Spins have only two states, $+1\,(\uparrow)$ or $-1\,(\downarrow)$.

Energy of a configuration C of spins is defined as

$$E(C) = -J \sum_{\langle i,j \rangle} S_i S_j$$

S_i = spin at site i of lattice
$\langle i, j \rangle$ = nearest neighbors in the lattice
J = interaction strength

$\uparrow\uparrow, \downarrow\downarrow$ spins aligned $\Rightarrow E = -J$
$\uparrow\downarrow, \downarrow\uparrow$ spins not aligned $\Rightarrow E = +J$

Object is to find average values of measurable quantities (e.g. energy, magnetization, etc), which are given by

$$\langle E \rangle = \sum_C p(C)\, E(C)$$

where $p(C)$ is the standard Boltzmann probability distribution from thermodynamics

$$p(C) = \frac{1}{Z} e^{-E(C)/kT}, \quad Z = \sum_C e^{-E(C)/kT}$$

`T` is temperature, `k` is Boltzmann's constant.

(Don't worry about the physics stuff, just look at it as a mathematical model).

Solving the Ising Model

Ising Model has been solved (for energy as a function of temperature) analytically in 2 dimensions, which gives important insights.

It has been the subject of many computer simulations for 3 (and more) dimensions.

Suppose we want to compute the exact result on a computer. Need to do this for all temperatures, but can choose a small number of values.

A real physical system has $O(10^{23})$ spins, but assume it can be approximated by a small system, e.g. a $32 \times 32 \times 32$ lattice.

Number of configurations in the sum is $2^{32,768} \sim 10^{10,000}$, so totally infeasible to compute results for all configurations.

Instead use **Monte Carlo simulation**, by summing (averaging) over sample configurations chosen with a Boltzmann probability distribution.

MC is used in many applications, since it is the most efficient way to solve large-dimensional integrals or sums.

Error is statistical, i.e. order $1/N^{1/2}$ for N configurations. whereas standard numerical integration techniques are order $1/N^{1/d}$ for d dimensions.

The Metropolis Algorithm

A method of generating variables with arbitrary probability distribution was invented by Metropolis, Teller^2, and Rosenbluth^2.

One of the first application programs run on early computers in the 50's.

The **Metropolis algorithm** is as follows:

1. Choose a new *trial configuration* (usually a small change to the existing configuration).
2. Compute the change in energy ΔE between the trial and current configurations.
3. If $\Delta E \leq 0$ make the proposed change (i.e. replace the current configuration with the trial configuration).
   If $\Delta E > 0$ make the change with probability $e^{-\Delta E/kT}$.

For the Ising model, change in the config is to try to change the sign of (flip) a spin.

If we try to change many spins at once, ΔE will be large, so the probability to accept the change $e^{-\Delta E/kT}$ will be small. So update a single spin at a time.

For a single spin flip, ΔE depends only on the spin values at the site and its nearest neighbors, i.e. the update is *local*.

Cluster Algorithms

Local MC algorithms (such as Metropolis) perform poorly for large lattices because they update only one spin at a time, so it takes many iterations to get a statistically independent configuration.

More recent *cluster algorithms* use clever ways of finding clusters of sites that can be updated at once.

Swendsen-Wang Algorithm

By placing bonds with probability p_b between neighboring sites with the same spin value, the Ising model can be written in terms of interacting *clusters* of connected spins.

For the particular choice $p_b = 1 - e^{-\beta}$ ($\beta = \frac{J}{kT}$) the interaction energy is zero, so the clusters are independent.

So can update all the clusters (by changing the current spin to a random new spin value) independently.

Wolff Algorithm

A site is chosen at random and a single cluster is grown around the site by adding bonds with the same probability as S-W. All spins in the cluster are then flipped.

This tends to favor larger clusters, and spins are always flipped, so the update is more effective.

Parallel Spin Model Algorithms

Metropolis algorithm for spin models is easy to parallelize efficiently, since it is regular and local.

- Regular data structures ⇒ simple domain decomposition
- Regular algorithm ⇒ lots of parallelism and perfect load balance
- Local interactions ⇒ local communications

Swendsen-Wang algorithm is difficult to parallelize efficiently, since it is irregular and non-local.

- Clusters are irregular (fractal!) in size and shape ⇒ hard to load balance, lots of non-local communication

Wolff algorithm is almost impossible to parallelize efficiently.

- Grow a single, irregular cluster starting at a random lattice site ⇒ little parallelism, very hard to load balance

Regular Local vs. Irregular Non-Local

In many computational science applications, there are standard algorithms that are regular, simple, often local, and thus parallelize very efficiently.

However, complex physical problems and physical systems are usually irregular, often non-local, and may change rapidly with time, so simple regular and local algorithms tend to have problems in simulating these complex systems, e.g. slow convergence.

More complex algorithms, e.g.

- non-local, irregular cluster algorithms
- multiscale and multigrid methods for PDEs and spin models
- adaptive, irregular grids for finite element calculations
- hierarchical, adaptive N-body solvers

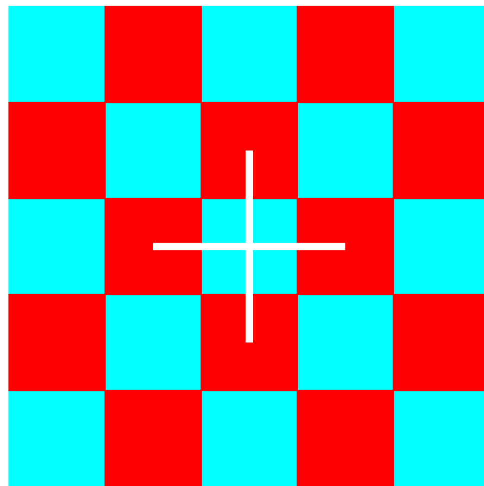
all work much better sequentially (more efficient simulations), but are much more difficult to parallelize efficiently!

Red/Black or Checkerboard Update

Since the update of a site depends on values of its nearest neighbors in the Metropolis algorithm, cannot update all sites at the same time – this would violate "detailed balance" (a kind of reversibility law for Monte Carlo updates).

Instead, the lattice is partitioned into a checkerboard of alternating red and black sites (this can also be done for dimensions $> 2d$).

Can then update all the black sites in parallel (since they are independent), followed by all the red sites in parallel.



This is known as a *red/black* or *checkerboard* update.

Note that unless there is more than one site per processor, the efficiency is at most 50% since half the processors will be idle.

Parallel Metropolis – Data Parallel

Parallel Metropolis using a checkerboard update is a data parallel SIMD algorithm. It is very simple to code in Fortran 90 and High Performance Fortran (HPF).

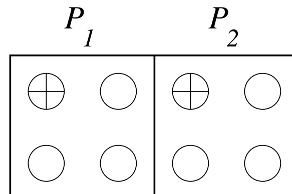
- Parallelism is encoded in Fortran 90 array syntax (no sequential loops over lattice sites, `WHERE` instead of `IF` within `DO` loop, etc).
- Use standard domain decomposition and distribute the 2-d rectangular lattice among processors using `(BLOCK,BLOCK)` form.
  `(BLOCK,*)` or `(*,BLOCK)` may also be suitable, depending on the problem.
- Use a *logical mask* (`black`) which only activates the appropriate (abstract) processors which deal with black sites. Then change the mask `(black = .NOT.black)` and update the other sites.
- To get data from neighboring sites, use periodic shift operations `(CSHIFT)` for periodic boundary conditions (edges wrap around). Grids with PBCs are commonly used for most scientific problems since they give a better approximation to an infinite system.

Ising Model in HPF

```
      PARAMETER (length = 128)
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,BLOCK) :: LATTICE(length,length)
      REAL, DIMENSION(1:length,1:length) :: rand
      INTEGER, DIMENSION(1:length,1:length) :: spin, new_spin
      LOGICAL, DIMENSION(1:length,1:length) :: black
!HPF$ ALIGN WITH LATTICE :: spin, new_spin, rand, black

C
C  Generate random numbers
C
      CALL RANDOM_NUMBER(rand)
C
C  New trial spin (= - old spin)
C
      new_spin = -spin

C
C  Red/black checkerboard update
C
      DO checkerboard = 1,2
C
C  Calculate old and new energy (new energy = - old energy)
C  by summing over neighboring spins.
C
        WHERE (black)
          oldE = - spin * ( CSHIFT(spin,1,1) + CSHIFT(spin,1,-1) +
     &                      CSHIFT(spin,2,1) + CSHIFT(spin,2,-1) )
          newE = -oldE
          deltaE = newE - oldE
      END WHERE
C
C  Metropolis accept/reject
C
        WHERE (black.AND.((deltaE.LE.0).OR.(EXP(-beta*deltaE).GT.rand)))
          spin = new_spin
        END WHERE
C
C  Change from black to red sites in checkerboard update
C
        black = .NOT.black

      END DO
```

Parallel Metropolis – Message Passing

If the number of lattice sites per processor is $> 2^d$, need not use the checkerboard scheme, since cannot update neighboring sites at once.



This simple algorithm can be written in a message passing language such as MPI, so that it looks **exactly** the same as the sequential algorithm!

The only difference is how the `shift` subroutine, which finds neighboring spin values, is implemented:

- sequential algorithm — `shift` handles the boundary conditions.
- parallel algorithm — `shift` handles boundary conditions *and* does message passing (`MPI_SEND and MPI_RECEIVE`) if the neighboring spin is on a different processor.

Also, for the parallel algorithm, the `size` variable refers to the size of the sub-lattice on each processor, not the total lattice size.

Message passing programming is of course much more complicated for more complicated algorithms.

Ising Model in MPI

```
DO n=1,iterations

  DO i=1,size
  DO j=1,size

    old_spin = spin(i,j)
    new_spin = -old_spin
C
C -------- Get neighboring spins.
C          shift is a function defined to handle the
C          periodic boundary conditions and passing of
C          data between processors.
C
            spin1 = shift(i-1,j)
            spin2 = shift(i+1,j)
            spin3 = shift(i,j-1)
            spin4 = shift(i,j+1)
C
C -------- Sum neighboring spins to get energy.
C
    spin_sum = spin1 + spin2 + spin3 + spin4
    old_energy = old_spin * spin_sum
    new_energy = - old_energy
    energy_diff = new_energy - old_energy
C
C -------- Metropolis accept/reject step.
C
    IF ( ( energy_diff.LE.0 ) .OR.
     &           ( EXP(-beta*energy_diff).GT.random() ) ) THEN
      spin(i,j) = new_spin
    ENDIF

  ENDDO
  ENDDO

ENDDO
```

Ising Model in MPI – shift routine

```
FUNCTION shift(i,j)
```

Sequential

```
IF (i.GT.length) THEN
  shift = spin(1,j)
ELSE IF (i.LT.1) THEN
  shift = spin(length,j)
ELSE IF (j.GT.length) THEN
  shift = spin(i,1)
ELSE IF (j.LT.1) THEN
  shift = spin(i,length)
ELSE
  shift = spin(i,j)
```

Parallel

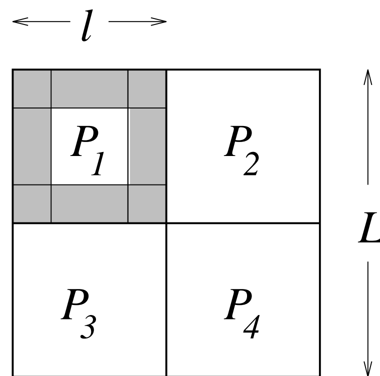
```
   IF (i.GT.length) THEN
     CALL MPI_SENDRECV(spin(1,j),len,MPI_INT,left,tag,
&                      shift,len,MPI_INT,right,tag,
&                      comm2d, status, ierr)
   ELSE IF (i.LT.1) THEN
     CALL MPI_SENDRECV(spin(length,j),len,MPI_INT,right,tag,
&                      shift,len,MPI_INT,left,tag,
&                      comm2d, status, ierr)
   ELSE IF (j.GT.length) THEN
     CALL MPI_SENDRECV(spin(i,1),len,MPI_INT,bottom,tag,
&                      shift,len,MPI_INT,top,tag,
&                      comm2d, status, ierr)
   ELSE IF (j.LT.1) THEN
     CALL MPI_SENDRECV(spin(i,length),len,MPI_INT,top,tag,
&                      shift,len,MPI_INT,bottom,tag,
&                      comm2d, status, ierr)
   ELSE
     shift = spin(i,j)
```

Efficiency of the Parallel Algorithm

Since the Metropolis algorithm for spin models is local and regular, it should parallelize very efficiently, even for the Ising model, which has very little computation.

For a 2-d grid of $N = P \times P$ processors, use a `(BLOCK,BLOCK)` distribution of the $V = L \times L$ sites of a 2-d lattice over the processor grid so that every processor has an $l \times l$ sub-lattice ($l = L/P$).

- Communication time $\propto l$
  (# edge sites of sub-lattice, i.e. perimeter).
- Calculation time $\propto l^2$
  (# sites of sub-lattice, i.e. volume).



Thus, as long as l is large enough, the communication/calculation ratio will be small, and the efficiency will be near 1.

Coarse Grained Algorithm

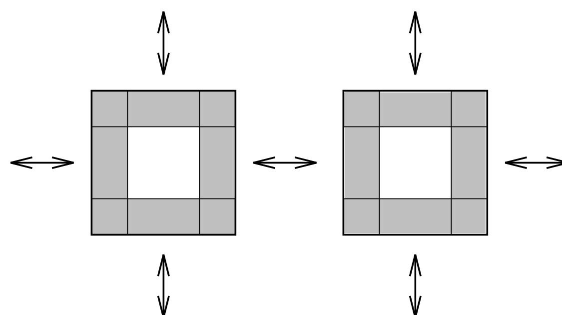
On a coarse-grained MIMD machine, there is a more efficient parallel Metropolis algorithm.

Each processor stores a copy of all the edge rows and columns from all the neighbouring processors, so it can update using local values, and then pass updated edge values in large blocks rather than having to access single values from neignbouring processors when they are required.

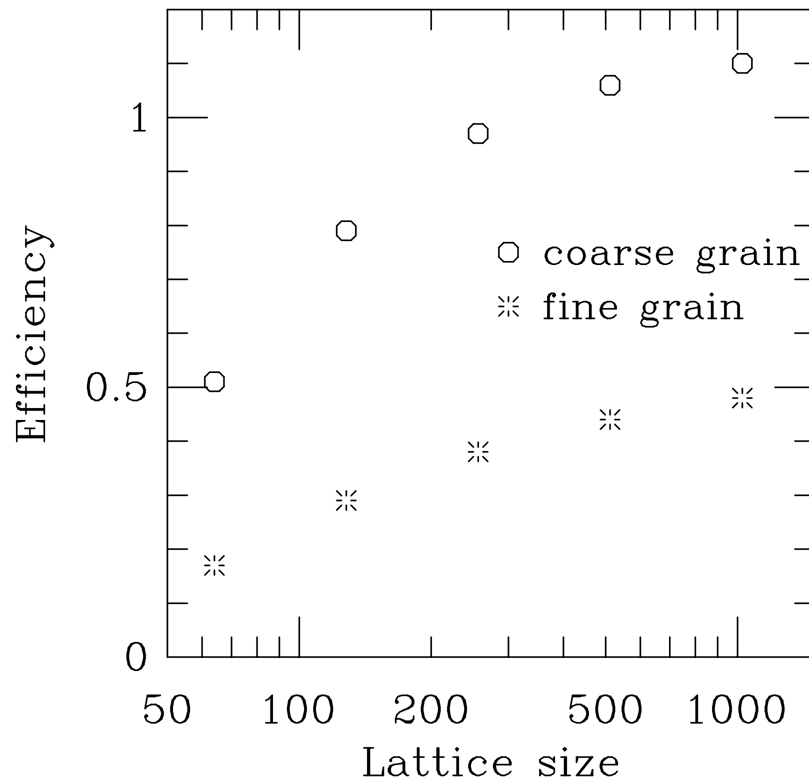
This requires a checkerboard partitioning of the lattice. A block of edge data is passed to neighboring processors after each red or black update.

This coarse-grain data transfer, or *block communication*, greatly reduces the latency time for communication, thereby improving efficiency.

It is also possible to overlap communications with computation, by computing edge values first, and sending them while the interior points are being computed.



Coarse-grain vs Fine-grain

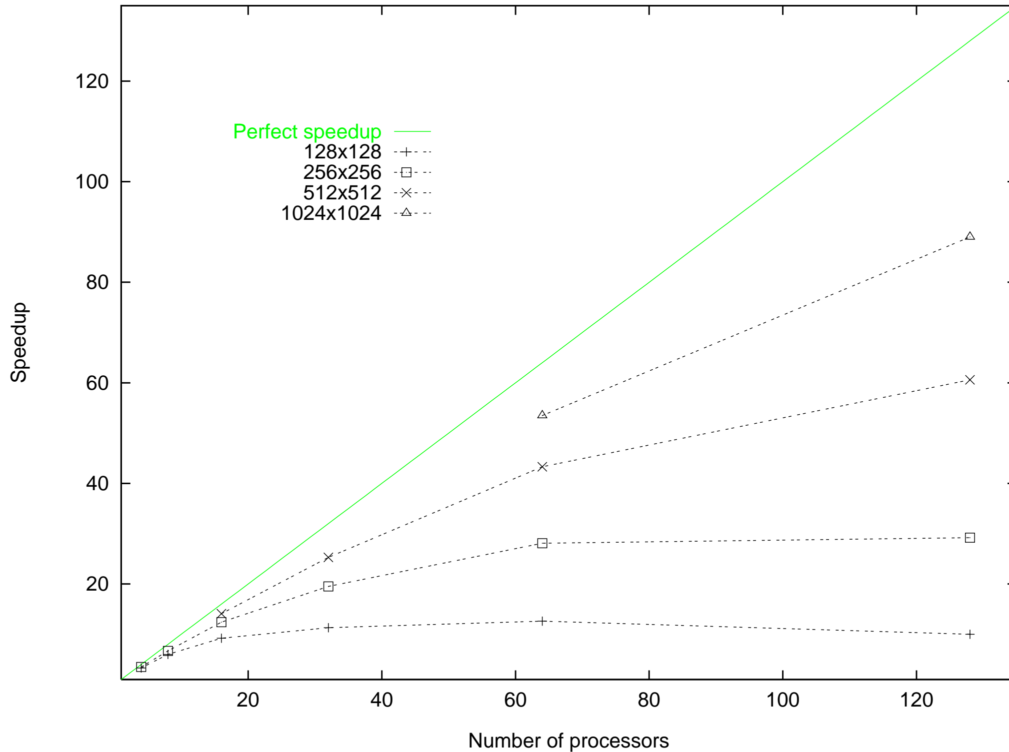


Efficiency of coarse and fine grain Metropolis algorithms on 16 nodes of a distributed memory machine.

Difference is quite pronounced for Ising model, for which ratio of computation to communication is low.

Note that efficiency can be greater than 1! This is due to effects like greater use of cache memory when data is distributed over many processors.

Speedups for Parallel Algorithms



Speedup for parallel Swendsen-Wang algorithm. Note that speedup improves as lattice size is increased.

It is important to check speedup and efficiency on different numbers of nodes and for different problem sizes, in order to figure out what size of problem will run well on what size processor grid.

For example, in this case probably wouldn't want to use more than about 64 procs for a 1024^2 lattice, 32 procs for a 512^2 lattice, 16 procs for a 256^2 lattice, etc. Plot of efficiency is perhaps a better indicator of this.

Parallel Metropolis – Threads

Can use a checkerboard update to avoid data dependencies in the loop.

Easy to do this using standard sequential loops, e.g.

```
for (start=0; start<=1; start++) {
  istart = start;
  jstart = (1 + start) % 2;
  for (i=istart; i<isize; i+=2) {
    for (j=jstart; j<jsize; j+=2) {
      /* update spin at (i,j)   */
      /* newspins = red sites   */
      /* oldspins = black sites */
      /* newspin <- oldspins    */
}}}
```

In OpenMP, can update in parallel by just using OMP PARALLEL directives around the loop.

Compiler will allocate different sites to different threads.

In Java, could allocate different threads to work on different domains, e.g. blocks or rows or columns of the array.

No data dependencies using checkerboard, so no synchronization problems.

Measurements

Measurements of standard physical quantities such as the total energy are global sums of local quantities (energy per site involves only values of nearest neighbors), and are easily done in parallel.

1. Calculate the quantities locally (e.g., energy per site), which can be done completely in parallel (no checkerboarding required)
2. Sum the results over all processors to get the global sum.

Summing the results over all processors is done by a global reduction operation.

For data parallel languages like HPF, use the `SUM` intrinsic to sum over all abstract processors. For example,
`energy = SUM (energy-per-site) / volume`

For message passing languages like MPI:

1. First, do a partial sum over sites in each physical processor
2. Then, combine the partial sums on each processor, using e.g. `MPI_REDUCE` and `MPI_SUM`.

OpenMP uses similar approach with the `reduce` construct.

Non-local Measurements

Some measurements involve averaging over non-local quantities, e.g. the correlation function $\Gamma(n) = \Sigma_i S_i S_{i+n}$.

This can be easily calculated in parallel, by continually shifting a copy of the spin variables one site at a time.

1. After the first shift, every (abstract) processor calculates the local value of $\Gamma(1)$, i.e., $S_i S_{i+1}$, at each site. Then and the sum over all sites is taken (using `SUM` for HPF or `MPI_REDUCE` and `MPI_SUM` for MPI).
2. After the second shift every processor calculates the local value of $\Gamma(2)$, and the sum over sites is done.
3. This is continued until the spin has been shifted halfway around the lattice (only halfway due to periodic boundary conditions).

For message passing languages it is more efficient to only do the partial sums on each processor after every shift.

The global sums (`MPI_SUM`) are done at the end on the vector $\Gamma(n)$ rather than each of the individual values $\Gamma(1), \Gamma(2), \ldots$ in turn, so as to reduce latency by combining all values at once.

Relation to PDE Solvers

The Metropolis algorithm for a 2-d spin model is similar in many ways to numerical methods for solving differential equations, such as Laplace's equation $\nabla^2\phi = 0$.

This can be discretized onto a 2-d grid, with the update depending only on nearest neighbor points, e.g., an iterative scheme to solve this equation would replace ϕ_i by the average value of its four neighbors.

Very similar to Ising model Metropolis update, which also requires values of four nearest neighbours.

It is possible to iteratively update all sites at once (Jacobi algorithm), but this gives very poor convergence.

For sequential program, get much faster convergence by using Gauss-Seidel algorithm, which uses a combination of neighboring values from previous update step and current update step.

In parallel, can get similar improvement in convergence by using checkerboard update.

So this problem is parallelized just like coarse grain parallel Metropolis algorithm.

- standard block data decomposition
- red/black or checkerboard update
- local (block) communications

Independent Parallelism

Another approach is to use independent or job-level parallelism. This is used for many scientific simulations.

There are two main approaches:

- Run independent jobs with different parameters, to explore a parameter space.
- For probabilistic and Monte Carlo simulations, run independent jobs using different random numbers, to reduce statistical error.

Advantages:

- No need to do parallel programming.
- Gives a parallel efficiency of 100%!
- Can be used over a network of workstations.

Disadvantages:

- Problem size is restricted by memory on a single processor, rather than the whole parallel computer.

Independent parallelism is currently the only way to get good efficiency for some problems such as the Wolff algorithm.

Can also use a "hybrid" approach, e.g. on a 256-processor machine, can run 16 independent jobs of 16 processors each.

Message Passing vs Data Parallel

Coarse grain parallel algorithms using message passing on MIMD machines give excellent performance for regular algorithms such as Metropolis, and fairly good performance for irregular problems such as cluster algorithms.

This is because the efficient sequential algorithms can be used on the sub-domain on each processor, and communication can be amortized (or even overlapped) by the computation.

Data parallel languages such as High Performance Fortran handle regular algorithms very efficiently, and are much easier to program that explicit message passing languages. However they do not perform as well for irregular problems such as cluster algorithms.

Performance may be reduced a little on regular problems, depending on how well the compiler maps the problem onto the machine.

Poor performance for irregular problems is mainly because the (usually) slower SIMD algorithms must be used even for the domain within each processor.

There are also load balancing problems, which are harder to handle in data parallel programs.

Irregular Problems in HPF

Irregular, non-local problems such as cluster algorithms for spin models provide a real challenge (for algorithms and compilers) to efficient implementation in data parallel languages such as HPF.

It may be that good performance can only come by using library routines, e.g. for connected component labeling in this case, which would be implemented using an efficient message passing routine.

This can be done using the `EXTRINSIC` construct in HPF, which allows the program to call a subroutine that is written in a message passing language.

HPF can handle independent and hybrid parallelism using the `INDEPENDENT` construct.