DARK 2
Programming of Multiprocessors

Sverker Holmgren
IT/Division of Scientific Computing

Multiprocessors at IT/UPPMAX

Ra. 280 proc. Cluster with SM nodes

Ngorongoro. 64 proc. SM (cc-NUMA)

Debet&Kredit. 32+24 proc. SM (cc-UMA)

ISIS. Cluster with 800 cores. SM nodes

Multiprocessors at Earth

Cray XT Jaguar, 181504 Opteron cores, 362 TB main memory,
1,65 Pflop peak performance (Nov 10 2008). Located at ORNL, USA

This lecture:
How do we program such computers?

If you want to know more:

-High Performance Computing and Programming, VT08
    (Jan-March). Given together with Stanford University

-Programming of Parallel Computers. VT08 (March-May)

-Scientific Computing, project course. VT08 (March-May)
    Ask Erik or Sverker for a project proposal, or give
    one yourselves

-MSc thesis work?

Why?

Parallelization for representation of concurrent entities

- Examples: System programming, real-time systems, user interfaces…
- Concurrency is part of the problem
- Single or multiple processors

Parallelization for performance

- Concurrency is under the control of the programmer
- Possible to write a sequential program. May or may not exist
- Multiple processors or multiple cores within a processor
- ”Easy” if the right programming model is used. However, to get the expected performance we normally have to know about the architecture and the software tools.

Programming models

• Local namespace - ”Message passing”

• Shared namespace - ”Shared memory parallelization”, ”Multithreading”, ….

Warning! No consensus on terminology.

Local namespace models used both on local memory hardware (”Message passing architectures”, MPP, Beowulf,...) and on shared memory hardware.

Shared namespace model normally used on shared memory hardware (for now…)

Programming models

Note:

Research area today: Shared namespace on a local memory architecture, distributed shared memory, DSM. Example: ”OpenMP using DSM on a cluster of workstations”, ”Replace hardware coherency with software techniques”...

(Several IT dissertations lately: **Zoran Radovic**: Software Techniques for Distributed Shared Memory, **Håkan Zeffer**: Towards Low-Complexity Scalable Shared Memory Architectures)

A combination of both models might be used in the same program, two-level parallelization (Because you have to, or because it gives better performance).

Example: MPI+OpenMP on a cluster of shared memory MP (e.g. Isis or Ra)

Message passing modell

•Standard processes communicate via explicit calls to a message passing library. Standard: Message Passing Interface (MPI). MPI 1.0 1994, now MPI 2.0. Bindings for C, C++, Fortran, (Java).

```
mpi_send(buffer,count,datatype,destination,tag,communicator)
```

•Synchronization is maintained by the messages. All communcation and syncronization has to be introduced by the programmer (who also has complete control of this).

•Mature and standardized. Often scalable and efficient

•Local namespaces makes programming hard (= less like sequential programming). Normally static distribution of data, and a fixed number of processes. Rescheduling and redistribution has to be done ”by hand”, and is expensive.

•The programmer has to deside about and has control over replication vs. communication (C.f. S-COMA on WildFire).

•Interaction with operating system is important (scheduling etc)

Shared memory programming

Early SMP programming:

•Vendor-specific pragmas/libraries and specialized (Fortran) compilers.

•Specialized hardware

•Large-scale numerical computations, science and technology

```
CMIC$  DO PARALLEL VECTOR
       DO 10 J = 1,N
          FORCE(J) = FORCE(J) + DT * UPDATE(J)
 10    CONTINUE
```

Shared memory programming

Trend:

• Standardized pragmas/libraries (built from low-level operations) => Portability and ease of use.

• Commodity hardware, from multiprocessor PCs to multiprocessor supercomputers (clusters of SMP)

• The multicore revolution! All new microprocessor generations have several processor cores! Soon they will have a large number (100) of cores.

• A large variety of applications. Databases, business, user interfaces, games….

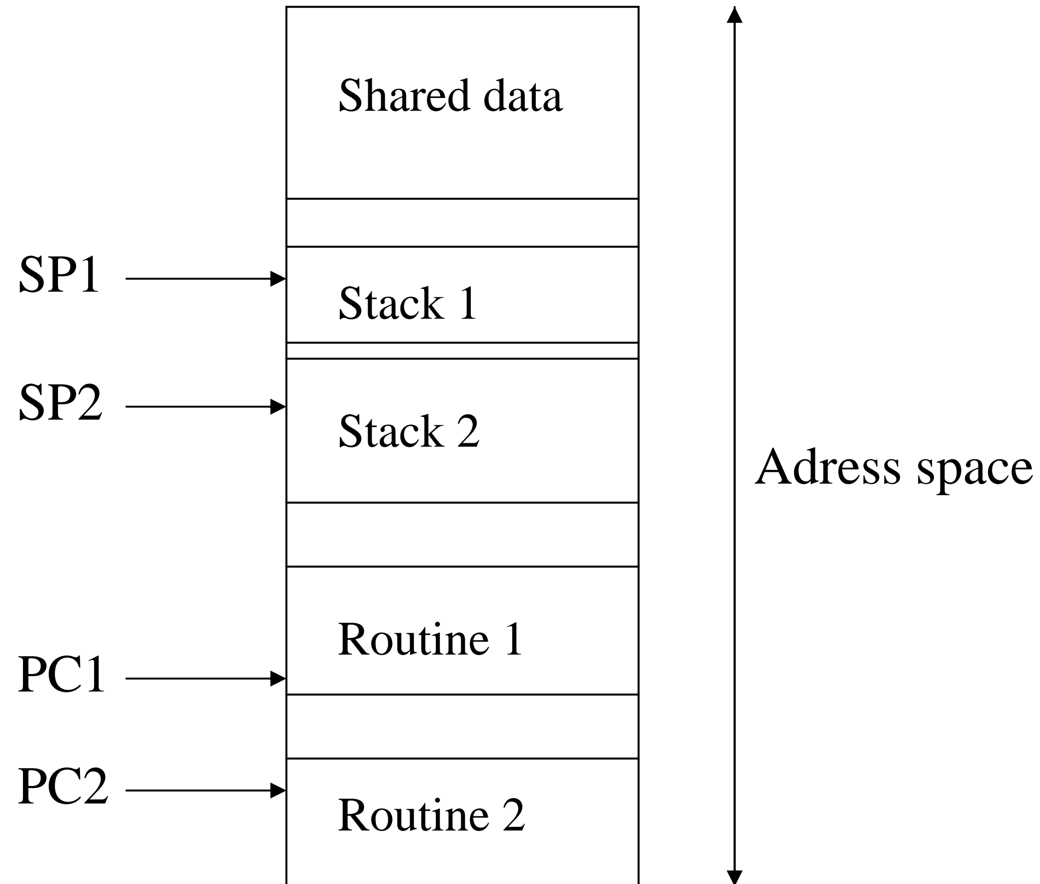
```
#pragma omp parallel for
    for ( year = 1; year < placement_horizon; year ++) {
      stock_value[year] = exp( coeff * year )  }
```

Threads/Processes

In practice, ”Multithreading” is normally used to get the parallelism

- A process has its own virtual adress space, open files etc.
- A process usually start of as/with one thread, but may start and stop more threads.
- A thread has a thread-private program counter and stack/stack pointer. All threads in a process share the other entities, e.g., adress space => synchronisation required!
- Neither processes nor threads need to map one-to-one onto the processors. Again, the scheduling policy used by the OS is often important for the performance.
- Thread creation and context switch is normally much cheaper than for processes.

Threads



Different levels of abstraction

• Atomic operations for synchronization in the instruction set, e.g., Test and Set.

• Basic multithreading libraries, e.g., Posix Thread Interface (Pthreads), Java threads, Win32 thread API.

• Medium level multithreading pragmas / preprocessors, e.g., OpenMP, Parmacs.

• Compiler with ”automatic parallelization” capability.

What do we need?

A few basic primitives are required to write a multithreaded program using a medium level approach. Examples from PARMACS:

Start and stop threads (”A parallel region”)

- CREATE(p,proc,args) - Create *p* threads that starts to execute the procedure *proc* with arguments *args*
- WAIT_FOR_END(p) - Wait for *p* spun-off threads to terminate

Syncronization

- LOCK(name) and UNLOCK(name) - Acquire and release mutually exclusive access to a critical region of code
- BARRIER(name,p) - Global synchronization of *p* threads
- WAIT(flag) and SET(flag) - Thread-to-thread event synchronization

A first example

Matrix multiplication using OpenMP:

```
void M_mult(float A[n][n], float B[n][n], float C[n][n]) {
   int i,j,k;
   float sum;
#pragma omp parallel for, local(i,j,k,sum)
   for (i=0; i<n; i++)
      for (j=0; j<n; j++)
         for (k=0; k<n; k++)
            sum=sum+A[i][k]*B[k][j];
         C[i][j]=sum;
        }
     }
}
```

Questions:

• How many threads are started, and how are they scheduled?

• How are the n independent tasks (computations of rows in C) partitioned over the threads?

• How is the (implicit) barrier at the end of the parallelized loop implemented?

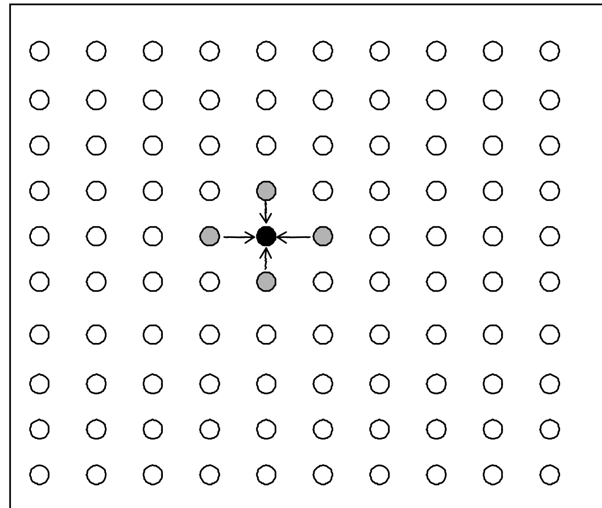
OpenMP (and most other tools) implicitly assumes an UMA architecture. There is no control of where the data is (geographical locality).

• On a NUMA architecture: Where are the matrices stored? For each thread, we would like the parts of C and A used to be stored in local memory (cache).

• The whole matrix B is used by all threads. On a NUMA architecture, should we replicate B in all local memories (caches)?

• On a system with automatic NUMA optimization features, B might be automatically replicated.

A Second Example



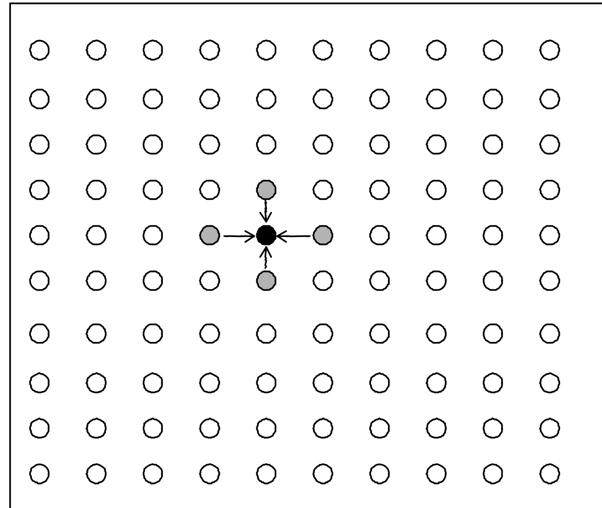
Expression for updating each interior point:

for i=1:n, for j=1:n

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$

- Gauss-Siedel: Iterative solver for systems of equations/PDE used in (old and inefficient!) applications
- BUT: GS sweeps are also often used as a basic building block in multigrid algorithms. Such schemes are highly efficient and used in many modern codes.
- Ex: GS -> 1600 iterations, MG -> 5 iterations. Work(MG iteration) = 2*Work(GS iteration)

Gauss-Siedel iteration



Expression for updating each interior point:

for i=1:n, for j=1:n

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$

- Gauss-Seidel (nearest-neighbor) sweeps to convergence
  - interior n-by-n points of (n+2)-by-(n+2) updated in each sweep
  - updates done in-place in grid, and diff. from prev. value computed
  - accumulate partial diffs into global diff at end of every sweep
  - check if error has converged (to within a tolerance parameter)
  - if so, exit solver; if not, do another sweep

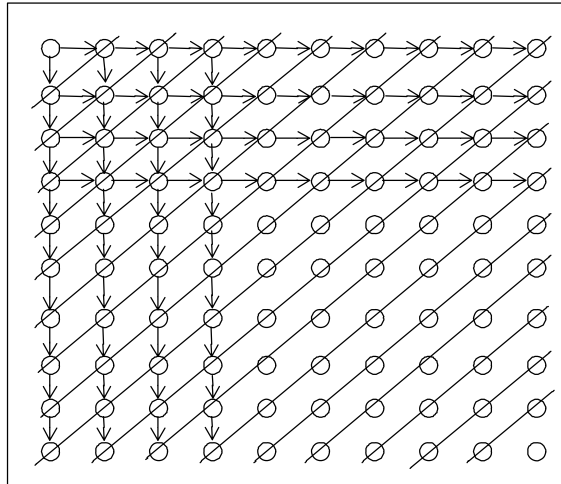
```
1.  int n;                                      /*size of matrix: (n + 2-by-n + 2) elements*/
2.  float **A, diff = 0;

3.  main()
4.  begin
5.     read(n) ;                                /*read input parameter: matrix size*/
6.     A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
7.     initialize(A);                           /*initialize the matrix A somehow*/
8.     Solve (A);                               /*call the routine to solve equation*/
9.  end main

10. procedure Solve (A)                         /*solve the equation system*/
11.    float **A;                               /*A is an (n + 2)-by-(n + 2) array*/
12. begin
13.    int i, j, done = 0;
14.    float diff = 0, temp;
15.    while (!done) do                         /*outermost loop over sweeps*/
16.       diff = 0;                             /*initialize maximum difference to 0*/
17.       for i ← 1 to n do                     /*sweep over nonborder points of grid*/
18.          for j ← 1 to n do
19.             temp = A[i,j];                  /*save old value of element*/
20.             A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                A[i,j+1] + A[i+1,j]); /*compute average*/
22.             diff += abs(A[i,j] - temp);
23.          end for
24.       end for
25.       if (diff/(n*n) < TOL) then done = 1;
26.    end while
27. end procedure
```

Where is the Parallelism?

- Simple way to identify concurrency is to look at loop iterations
  - *dependence analysis*; if not enough concurrency, then look further
- Not much concurrency here at this level (all loops *sequential*)
- Examine fundamental dependencies, ignoring loop structure



- Concurrency *O(n)* along anti-diagonals, serialization *O(n)* along diag.

V1: Reorder the Computations

```
17.   for s ← 2 to 2*n -1 do              /*sweep over antidiagonals*/
18.     for (j,k) such that j+k=s do
19.       temp = A[i,j];              /*save old value of element*/
20.       A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.         A[i,j+1] + A[i+1,j]); /*compute average*/
22.       diff += abs(A[i,j] - temp);
23.     end for
24.   end for
```

- Outer loop over anti-diagonals. Sequential.
- Inner loop over entries in anti-diagonal. Parallelizable!
- Ignoring round-off error, the answer will be the same as for the original code.

Problems:

- Variable parallelism, 2 => n => 1
- Many start-ups of threads with small amount of work in each one, alternatively load-imbalance.

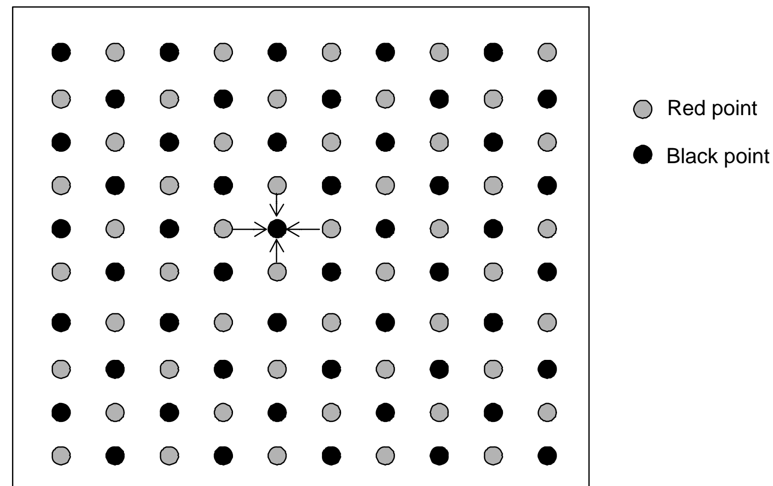
V2: Reorder the Computations

Use the wavefront algorithm on the last slide, but divide the matrix up into blocks and use a work pipeline + synchronization

Topic of next part of this lecture (results from recent research)

V3: Change the algorithm

- Reorder grid traversal: red-black ordering



- Different ordering of updates. The algorithm is changed, and the convergence properties are affected. Bad news: Convergence could be slower!
- Red sweep and black sweep are each fully parallel

V4: Change the Algorithm

Use asynchronous algorithm, simply ignore dependences within sweep.

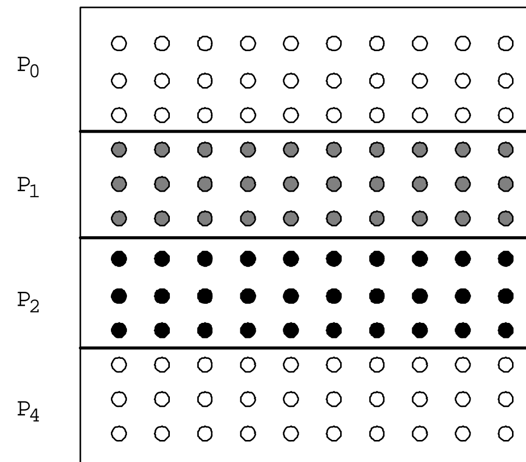
• The parallel program (and the convergence) will be *nondeterministic*.

```
15. while (!done) do                      /*a sequential loop*/
16.   diff = 0;
17.   for_all i ← 1 to n do               /*Parallelize over rows */
18.     for j ← 1 to n do
19.       temp = A[i,j];
20.       A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.         A[i,j+1] + A[i+1,j]);
22.       diff += abs(A[i,j] - temp);
23.     end for
24.   end for_all
25.   if (diff/(n*n) < TOL) then done = 1;
26. end while
```

Mapping of rows to threads

Static mapping

- block assignment of rows: Row i is assigned to process $\left\lfloor \frac{i}{p} \right\rfloor$
- cyclic assignment of rows: process i is assigned rows i, $i+p$, and so on



Dynamic mapping

– get a row index, work on the row, get a new row, and so on

Deciding How to Map to Threads

Static versus *Dynamic* techniques

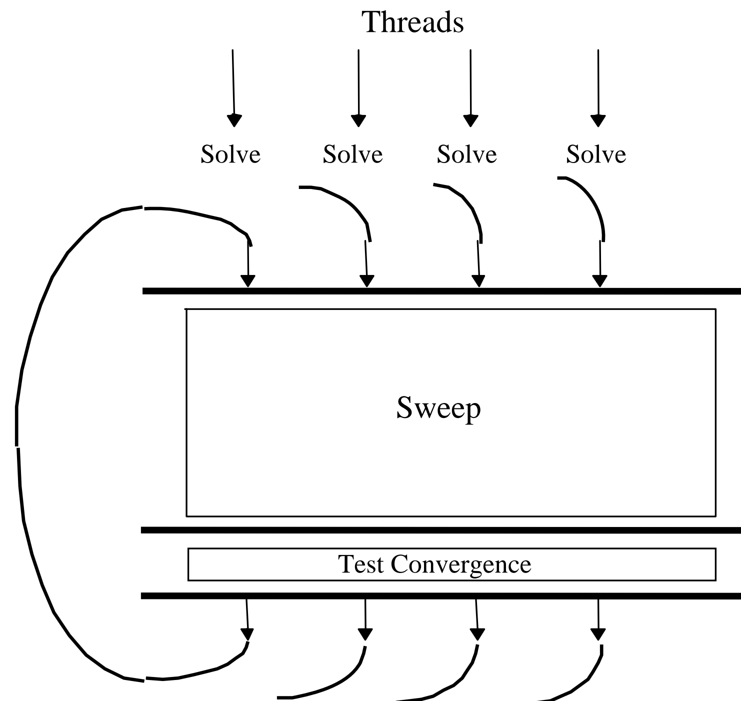
Static:

- Can decrease communication an increase locality. In our example: Benificial to keep adjacent rows together!
- Algorithmic mapping based on input; won't change
- Low runtime overhead
- Computation must be predictable
- Preferable when applicable (except in multiprogrammed/heterogeneous environment)

Dynamic:

- Adapt at runtime to balance load
- Can increase communication and reduce locality
- Can increase task management overheads

Code parallelized using Parmacs macros



```
1.        int n, nprocs;            /*matrix dimension and number of processors to be used*/
2a.       float **A, diff;          /*A is global (shared) array representing the grid*/
                                    /*diff is global (shared) maximum difference in current
                                    sweep*/
2b.       LOCKDEC(diff_lock);       /*declaration of lock to enforce mutual exclusion*/
2c.       BARDEC (bar1);            /*barrier declaration for global synchronization between
                                    sweeps*/

3.    main()
4.    begin
5.          read(n); read(nprocs);      /*read input matrix size and number of processes*/
6.          A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
7.          initialize(A);              /*initialize A in an unspecified way*/
8a.         CREATE (nprocs-1, Solve, A);
8.        Solve(A);                     /*main process becomes a worker too*/
8b.         WAIT_FOR_END (nprocs-1);    /*wait for all child processes created to terminate*/
9.    end main

10.   procedure Solve(A)
11.       float **A;                                  /*A is entire n+2-by-n+2 shared array,
                                                      as in the sequential program*/
12.   begin
13.       int i,j, pid, done = 0;
14.       float temp, mydiff = 0;                     /*private variables*/
14a.      int mymin = 1 + (pid * n/nprocs);           /*assume that n is exactly divisible by*/
14b.      int mymax = mymin + n/nprocs - 1            /*nprocs for simplicity here*/

15.       while (!done) do                      /*outer loop over all diagonal elements*/
16.         mydiff = diff = 0;                  /*set global diff to 0 (okay for all to do it)*/
16a.      BARRIER(bar1, nprocs);                /*ensure all reach here before anyone modifies diff*/
17.         for i ← mymin to mymax do           /*for each of my rows*/
18.               for j ← 1 to n do             /*for all nonborder elements in that row*/
19.               temp = A[i,j];
20.               A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                  A[i,j+1] + A[i+1,j]);
22.               mydiff += abs(A[i,j] - temp);
23.            endfor
24.         endfor
25a.        LOCK(diff_lock);                    /*update global diff if necessary*/
25b.        diff += mydiff;
25c.        UNLOCK(diff_lock);
25d.        BARRIER(bar1, nprocs);              /*ensure all reach here before checking if done*/
25e.        if (diff/(n*n) < TOL) then done = 1;        /*check convergence; all get
                                                        same answer*/
25f.        BARRIER(bar1, nprocs);
26.       endwhile
27.   end procedure
```

Notes

- Scheduling of rows to the threads is controlled by the loop bounds. (This is often called *domain decomposition*). The mapping is static and blockwise.
  - unique id per thread, used to control scheduling
- Done condition evaluated redundantly by all
- Code that does the update identical to sequential program
  - each process has private mydiff variable
- Most interesting special operations are for synchronization
  - accumulations into shared diff have to be mutually exclusive
  - why all the barriers?

Programming for performance

Balancing the workload and reducing wait time at synch points

Reducing inherent communication

Reducing extra work

Trade off between replication of data and communication

Even these algorithmic issues trade off:

- Minimize comm. => run on 1 processor => extreme load imbalance
- Maximize load balance => random assignment of tiny tasks => no control over communication
- Good partition may imply extra work to compute or manage it

The goal is to find an acceptable compromise!

Exploiting Spatial Locality

Besides capacity, granularity of data transfer and coherence is important. Reduce artifactual communication.