Why Parallel programming?

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
- "Easy" if the right programming model is used. However, to get the expected performance we normally have to be well informed about the architecture and 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, DSM. Example: "OpenMP using DSM on a cluster of workstations"

DSZOOM presented at later lecture.

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 SMP".

Message passing model


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

- Synchronization is maintained by the messages. All communication and synchronization has to be introduced by the programmer (who also has complete control of this).
- Rather 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 decide about and has control over replication vs. communication (c.f. CMR on WildFire).
- Interaction with operating system might be 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

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

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)
- 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 often used to get the parallelism

- A process has its own virtual address 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., address 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

Shared data

<table>
<thead>
<tr>
<th>SP1</th>
<th>Stack 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP2</td>
<td>Stack 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC1</th>
<th>Routine 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC2</td>
<td>Routine 2</td>
</tr>
</tbody>
</table>
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:

```c
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.

• 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.
• The whole matrix B is used by all threads. On a NUMA architecture, should we replicate B in all local memories?
• On WildFire, B might be automatically replicated by the CMR mechanism.
A Second Example

- Simplified version of iterative solver used in (old and inefficient!) applications
- 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

\[
\]

Expression for updating each interior point:

```
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);
8. Solve (A); /*call the routine to solve equation*/
9. end main
10. procedure Solve (A)
11. /*solve the equation system*/
12. begin
13. float **A;
14. /*A is an (n + 2)-by-(n + 2) array*/
15. int i, j, done = 0;
16. float diff = 0, temp;
17. while (!done) do /*outermost loop over sweeps*/
18. diff = 0;
19. for i ← 1 to n do /*sweep over nonborder points of grid*/
20. for j ← 1 to n do
21. temp = A[i,j]; /*save old value of element*/
23. diff += abs(A[i,j] - temp);
24. end for
25. end for
26. if (diff/(n*n) < TOL) then done = 1;
27. end while
28. 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

```
15. for s ← 2 to 2*n -1 do /*sweep over anti-diagonals*/
16. for (j,k) such that j+k=s do
17. temp = A[i,j]; /*save old value of element*/
19. diff += abs(A[i,j] - temp);
20. end for
21. end for
22. if (diff/(n*n) < TOL) then done = 1;
23. end while
```

- 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, \(2 \Rightarrow n \Rightarrow 1\)
- Many start-ups of threads with small amount of work in each one, alternatively load-imbalance.
V2: 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: Normally slower convergence!
- Red sweep and black sweep are each fully parallel

V3: Change the Algorithm

Use asynchronous algorithm, simply ignore dependences within sweep.
- The parallel program (and the convergence) will be nondeterministic.

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: Beneficial 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

Global Event Synchronization

BARRIER(nprocs): wait here till nprocs processes get here
• Built using lower level primitives
• Global sum example: wait for all to accumulate before using sum
• Often used to separate phases of computation

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!

Determining Task Granularity

Task granularity: amount of work associated with a task

General rule:
• Coarse-grained => often less load balance
• Fine-grained => more overhead; often more comm., contention

Reducing Serialization

Event synchronization
• Reduce use of conservative synchronization
  – e.g. point-to-point instead of barriers, or increase granularity of pt-to-pt
• But fine-grained synch more difficult to program, more synch ops.

Mutual exclusion
• Separate locks for separate data
  – e.g. locking records in a database: lock per process, record, or field
• Smaller, less frequent critical sections
• Stagger critical sections in time

Important: How are the synchronization primitives implemented? Algorithm? Busy-wait or blocking?

Exploiting Spatial Locality

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

Often most important in practice: False sharing could reduce performance significantly (leading to that the programmer thinks that shared memory programming is inefficient…)

All depend on how spatial access patterns interact with data structures
• Fix problems by modifying data structures, or layout/alignment
Spatial Locality Example

- Repeated sweeps over 2-d grid, each time adding 1 to elements
- Natural 2-d versus higher-dimensional array representation

Tradeoffs with Inherent Communication

Partitioning grid solver: blocks versus rows
- Blocks still have a spatial locality problem on remote data
- Rowwise can perform better despite worse inherent c-to-c ratio

Result depends on $n$ and $p$