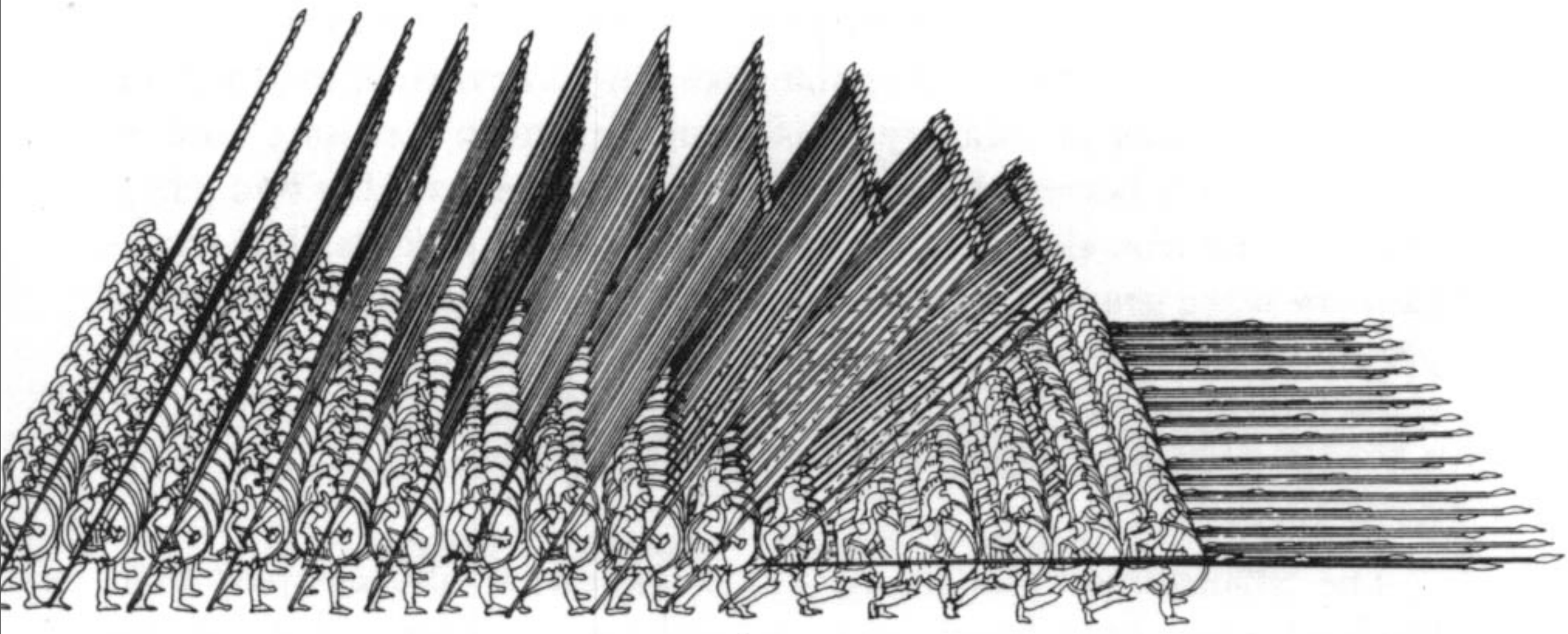
OpenMP:
Open Multiprocessing

Erik Schnetter

May 20-22, 2013, IHPC 2013, Iowa City

2,500 BC: Military Invents Parallelism



The Macedonian phalanx, here shown in its fighting formation of 256 men, the syntagma.

Outline

1. Basic concepts, hardware architectures
2. OpenMP Programming
3. How to parallelise an existing code
4. Advanced OpenMP constructs

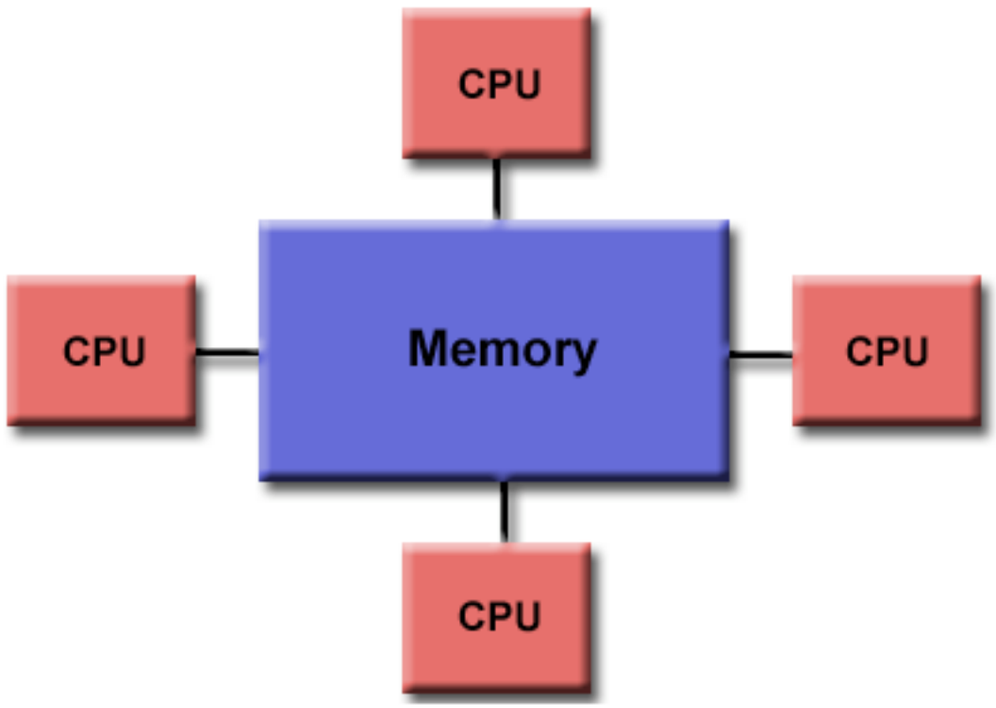
OpenMP: Basic Concepts, Hardware Architecture

- Parallel programming is much harder than serial programming; we use it (only) because it improves performance, enabling certain projects that are otherwise impossible
- Possible performance of a code is ultimately defined by the computing architecture on which it runs
- Need to have at least passing knowledge of hardware architectures

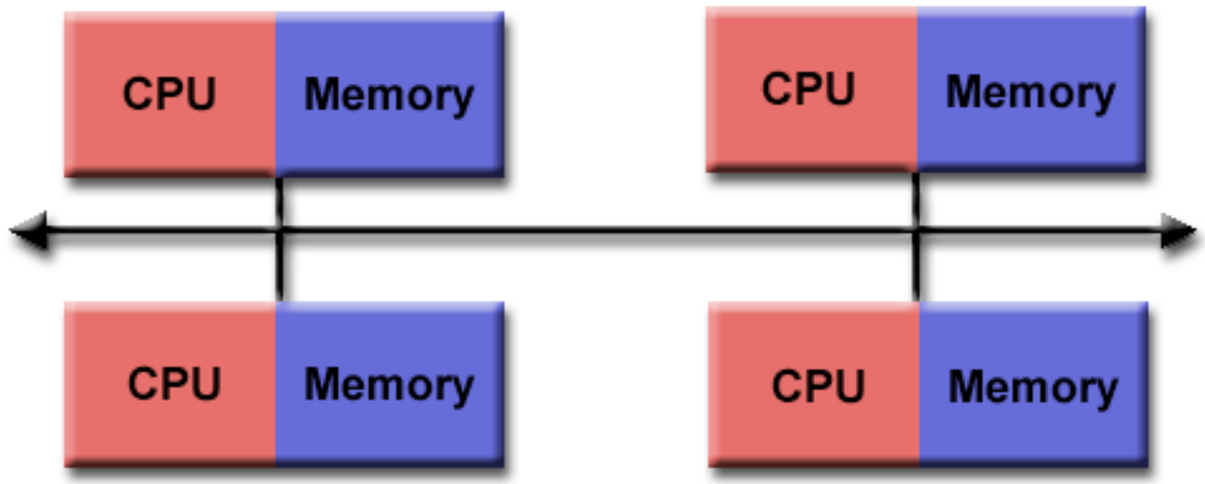
Parallel Computer Memory Architectures

THE UNIVERSITY OF IOWA

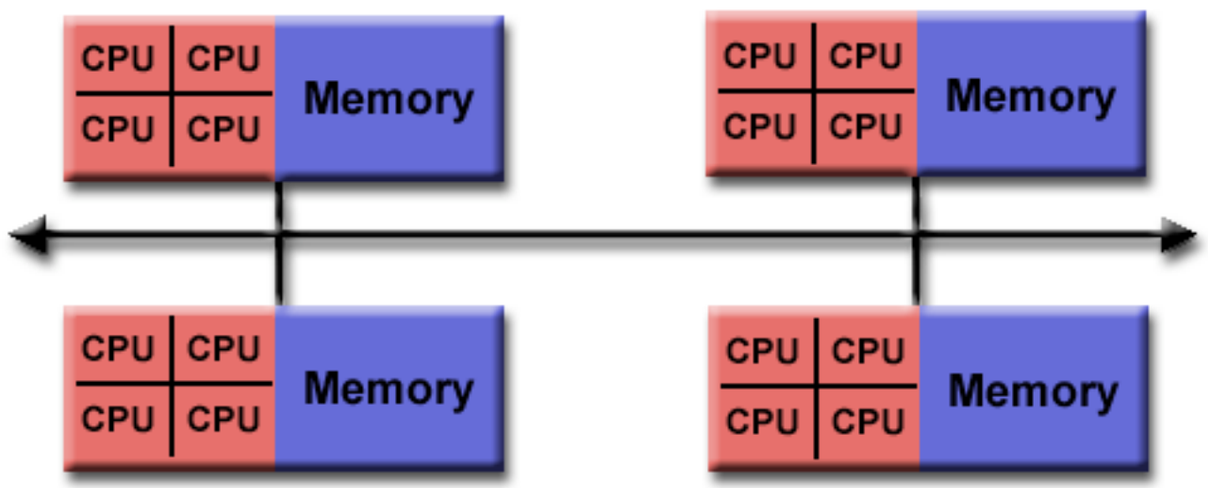
Shared Memory



Distributed Memory



Hybrid Distributed Shared Memory



Relation to Parallel Programming Models

THE UNIVERSITY OF IOWA

• OpenMP: Multi-threaded calculations occur within shared-memory components of systems, with different threads working on the same data.

• MPI: Based on a distributed-memory model, data associated with another processor must be communicated over the network connection.

• GPUs: Graphics Processing Units (GPUs) incorporate many (hundreds) of computing cores with single Control Unit, so this is a shared-memory model.

• Processors vs. Cores: Most common parallel computer, each processor can execute different instructions on different data streams

-Often constructed of many SIMD subcomponents

MPI vs. OpenMP

- MPI: Difficult to use, but makes it *possible* (not easy!) to write highly efficient code
  - like writing machine code
- OpenMP: Easy to use
- 90/10 rule: Compared to MPI, OpenMP gives 90% of the performance with 10% of the effort
  - OpenMP requires shared memory system

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?

(ACROSS FIVE YEARS)

HOW MUCH TIME YOU SHAVE OFF	HOW OFTEN YOU DO THE TASK: 50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS				2 MONTHS	2 WEEKS	1 DAY
1 DAY					8 WEEKS	5 DAYS

http://www.xkcd.com/1205/

Single Image View vs. Communicating Processes

system performance →

programming difficulty ↓

	Shared memory (small systems)	Distributed memory (large systems)
Single image (one program) (easy)	OpenMP	e.g. HPF, CAF
Communicating processes (difficult)	e.g. pthreads	MPI

Multi-Threading

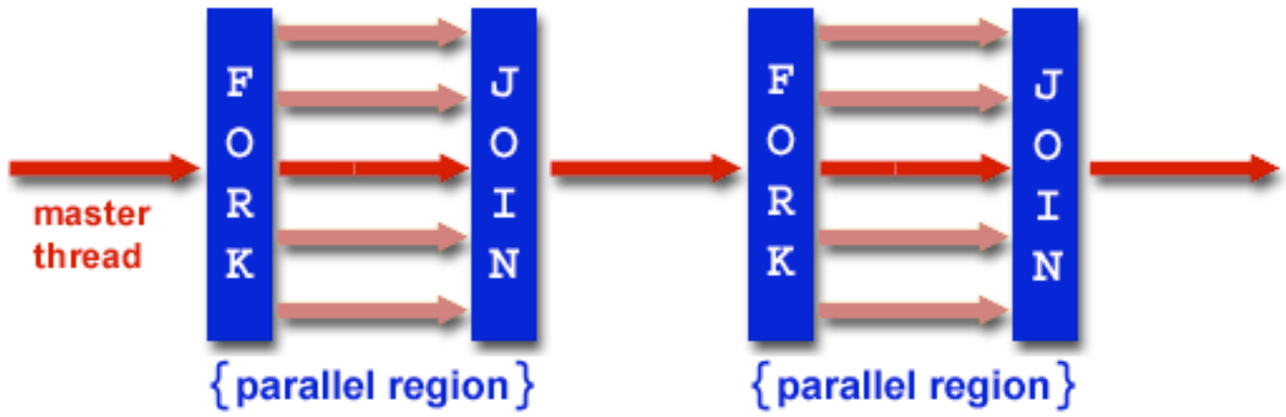
THE UNIVERSITY OF IOWA

- Threading involves a single process that can have multiple, concurrent execution paths

- Works in a shared memory architecture

- Most common implementation is OpenMP (Open Multi-Processing)

```
serial code
        .
        .
        .
!$OMP PARALLEL DO
do i = 1,N
  A(i)=B(i)+C(i)
enddo
!$OMP END PARALLEL DO
        .
        .
        .
serial code
```



- Relatively easy to make inner loops of a serial code parallel and achieve substantial speedups with modern multi-core processors

OpenMP Design Principle

- Parallel code has same semantics as serial code (and looks very similar)
- Parallelisation via *directives*, which are comments inserted into the code
  - parallel code remains also a serial code
- Main advantage: Can parallelise a serial code incrementally, starting with most expensive parts

More Information:

- http://www.openmp.org/
- Many tutorials available on the web, standard definition freely available
- Built into nearly every C/C++/Fortran compiler, including GNU
  - available everywhere, easy to use, there is no excuse for not using it (except if your algorithm is not parallel)

Current CPU/Memory Hardware Architecture

- Today's CPU/memory hardware architecture is surprisingly complex
  - nearly impossible to precisely predict performance, even for experts
- Most systems have several processors, multiple cores, and several memory elements (!) on each node
- Relevant for performance:
  Flop/s (computations) and GByte/s (memory accesses)

Helium, Compute node

Indexes: physical

Date: Wed 06 Jun 2012 11:12:29 AM CDT

First Steps in OpenMP

- Fortran:

```
program hello
   implicit none
   integer :: i
   print ‘(“Hello, World!”)‘
   !$omp parallel do
   do i=1,10
      print ‘(“iteration: ”,i0)‘, i
   end do
end program hello
```

First Steps in OpenMP

- C/C++:

```
#include <stdio.h>
int main()
{
    printf("Hello, world!\n");
#pragma omp parallel for
    for (int i=0; i<10; ++i) {
        printf("iteration %d\n", i);
    }
    return 0;
}
```

No Plumbing Necessary!

- Different from MPI code, it is generally not necessary to look at the thread number ("rank"), or at the total number of threads
- Easy to combine serial and parallel parts of an algorithm
  - if you need to execute certain operations in-order, just don't parallelise the loop

Fortran vs. C/C++

- In Fortran, OpenMP directives begin with !$omp, and are usually paired with a corresponding end directive
- In C or C++, OpenMP directives begin with #pragma omp, and apply to the next statement or { } block

Important OpenMP Directives

- parallel/end parallel: define a parallel region
- do/end do: parallelise a do loop
- critical/end critical: serialise a region within a parallel region
- Clauses for parallel regions:
  - private: list variables that should not be shared between threads
  - reduction: list variables that should be reduced (their values "combined")

omp do
(omp for in C/C++)

- To parallelise a loop, the number of iterations must be known before the loop begins
- The loop iterations must also be *independent*
- OpenMP will split iterations automatically over all available threads
- The parallelised loop may be executed in an arbitrary "order" (actually, it will execute in parallel)

Example: Fibonacci Series

The Fibonacci series is defined by:

$$f(k+2) = f(k+1) + f(k) \quad \text{with } f(1) = f(2) = 1$$

The Fibonacci series is therefore $(1, 1, 2, 3, 5, 8, 13, 21, \ldots)$

The Fibonacci series can be calculated using the loop

```
f(1)=1
f(2)=1
do i=3, N
      f(i)=f(i-1)+f(i-2)
enddo
```

How do we do this computation in parallel?

This calculation cannot be made parallel.

- We cannot calculate $f(k+2)$ until we have $f(k+1)$ and $f(k)$

- This is an example of data dependence that results in a non-parallelizable problem

Example: omp do

- alpha = 0.24

```
!$omp parallel do
do i=2,N-1
   anew(i) = alpha * (aold(i-1) + aold(i+1))
end do
```

Example: omp do with nested loops

- alpha = 0.24

```
!$omp parallel do collapse(2)
do i=2,N-1
   do i=2,N-1
      anew(i,j) = alpha * &
         (aold(i-1,j) + aold(i+1,j) + aold(i,j-1) + aold(i,j+1))
   end do
end do
```

omp critical

- A *critical region* is a section of code (within a parallel region) that must not be executed simultaneously by multiple threads
  - example: modifying a global variable, writing something to the screen
- Critical regions are slow; use them only if necessary, e.g. to handle exceptional cases

Example: omp critical

```
errcount = 0
!$omp parallel do
do i=2,N-1
   if (anew(i) < 0) then
      !$omp critical
      print '("error: anew<0 at ",i0), i
      errcount = errcount + 1
      !$omp end critical
   end if
end do
```

private

- By default, all variables are *shared* between all threads, i.e. there is a single instance of the variable
- Variables can be declared *private*, which means that each thread has its own, independent instance of the variable
- Rule of thumb:
  - read-only variables can be shared
  - temporary variables must be private
  - variables that are written can only be accessed in critical sections

Example: private

- alpha = 0.24

```
!$omp parallel do collapse(2) private(tmp)
do j=2,N-1
   do i=2,N-1
      tmp = aold(i-1,j) + aold(i+1,j) + aold(i,j-1) + aold(i,j+1)
      anew(i,j) = alpha * tmp
   end do
end do
```

reduction

- Reduction clauses allow reducing values (i.e. combining values) from multiple threads
  - for example: sum, min, max, ...
- Much more efficient than critical regions – try to rewrite critical regions as reductions, if possible

Example: reduction

- poscount = 0
  !$omp parallel do reduction(sum: poscount)
  do i=2,N-1
    if (anew(i) > 0) then
      poscount = poscount + 1
    end if
  end do
  print '("error count:",i4)', errcount

Applying OpenMP to an Existing Program

- Adding MPI parallelism to a serial program typically requires much surgery, and needs to be done all at once
  - however, MPI can speed up a program by a factor of 100,000 or more
- Adding OpenMP parallelism is much easier, and can be done incrementally
  - OpenMP can speed up a program probably by a factor of 10

How to Parallelise a Code (How to Modify a Code)

1. Understand the structure of the program
2. Define a simple test case, record its output
3. Find out which parts take a long time (this requires timing measurements)
4. Look for loops, examine data dependencies, add OpenMP directives
5. Check correctness (see 2.)
6. Compare performance

Loops

- General observation: the code inside a loop is executed (many time) more often than the code outside of a loop
- Therefore, optimising and parallelising the loops (aka *loop kernels)* is likely to lead to the largest performance improvements
- Parallelising via OpenMP usually means adding omp parallel do statements around do loops

Compiler Optimisations

- When measuring performance, it is necessary to use good compiler options to optimise the executable
  - typical flags: -O2, -O3, Intel: -fast, GNU: -Ofast, etc.
- It pays off to optimise for the particular hardware architecture (Intel: -xHOST, GNU: -march=native)
- Do not measure performance for a non-optimised executable; performance can differ significantly (by a factor of several)

Profiling

- *Profiling* means recording for each function how often it is called and how much time it takes during execution
- All compilers support adding profiling code to executables ("instrumenting")
  - note: instrumented code may run slower
- After running the instrumented executable, the profiling results can be analysed, e.g. with gprof (see Cheat Sheet)

Sample Profiling Output

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 72.48      0.79     0.79    10001     0.00     0.00  potential_mp_residual_
 26.61      1.08     0.29    10000     0.00     0.00  potential_mp_step_
  0.92      1.09     0.01        1     0.01     1.09  MAIN__
  0.00      1.09     0.00        1     0.00     0.00  potential_mp_initial_
```

- Here, most of the time is spent in "residual" and "step"
- Parallelising the main program or the initial data routine is pointless

Manual Timing Measurements

- The Unix *time* command can be used to measure execution time:
  - time ./calcpi
- Alternatively, you can time specific code section via omp_get_wtime():
  - use omp_lib
    double precision :: t0, t1
    t0 = omp_get_wtime()
    ... parallel section ...
    t1 = omp_get_wtime()
    print '("elapsed time:",f20.15," sec")', t1-t0

Compiling OpenMP Code

- By default, compilers will ignore all OpenMP directives, and will produce a serial executable
  - note: this serial executable will run correctly, it will only run more slowly
  - see the compiler documentation (or the Cheat Sheet) for enabling OpenMP

Running OpenMP Code

- You should explicitly choose the number of OpenMP threads when running a code
  - the default choice may be inefficient (it is unlikely to use a single thread)
- Unfortunately it's slightly complicated, see the Cheat Sheet for details
  - use qlogin to run on a compute node; timing measurements on the head node will be unpredictable
  - by default, the operating system likes to shift threads between cores, which is bad for performance

Advanced OpenMP Programming

- The current standard is OpenMP 3.1
  - However, some compilers only support version 3.0 or 2.x
  - Future versions will likely add support for defining memory locality for variables (for GPUs and other accelerators), and for SIMD vectorisation (see also to CUDA/ OpenCL)

Other OpenMP Directives

- OpenMP offers a range of other directives:
  - atomic: a fast version of critical
  - barrier: wait for other threads
  - master: execute only on the master thread
  - single: execute only once
  - workshare: parallelise array operations
  - sections: MPMD, functional decomposition
  - task: low-level task management

Other OpenMP Clauses

- OpenMP offers a range of other clauses:
  - schedule: choose strategy for splitting loops
  - nowait: disable some implicit barriers
  - copyin, copyprivate, firstprivate, lastprivate: manage private and shared variables
  - if: conditionally disable a parallel region
  - num_threads: choose number of threads

Other OpenMP Functions

- OpenMP also offers run-time functions that can be called:
  - Fortran: use omp_lib
  - C/C++: #include <omp.h>
  - omp_get_thread_num(): current thread id
  - omp_get_num_threads(): number of threads
  - omp_get_max_threads(): max number of threads
  - omp_set_num_threads(): set number of threads
  - omp_get_num_procs(): number of cores

Hybrid Parallelisation

- It makes sense to combine MPI and OpenMP:
  - MPI handles communication between nodes, OpenMP distributes the workload within a node
- This can help reduce overhead introduced by MPI:
  - MPI may require duplicating certain data structures for each process
  - there may be scaling problems for large numbers of MPI processes

Alternatives to OpenMP and MPI

- There is a large gap between OpenMP, which is rather easy to use, and MPI, which is quite difficult
- A range of other, much less widely used programming standards exist, targeting parallel programming, distributed programming, accelerators, etc.

HPF
(High Performance Fortran)

- HPF uses concepts similar to OpenMP, but for distributed memory systems, not just shared memory
- HPF adds directives that specify which variables (arrays) should be distributed over processes
- Example:

```
double precision a(N)
!$hpf distribute a(block)

!$hpf independent
do i=1,N
   a(i) = 0
end do
```

- Unfortunately, HPF is mostly dead

CAF
(Co-Array Fortran)

- CAF is ~~a proposed addition to~~ part of the Fortran standard
- CAF takes the same "communicating processes" approach as MPI
- CAF allows distributing arrays over multiple processes, and provides a simple way to access remote array elements (much simpler than MPI)
  - Example: a[myrank] = a[myrank+1]

UPC
(Unified Parallel C)

- Similar to CAF, but more flexible
- New constructs: shared arrays, pointers to shared objects
  - Example:

```
shared[nlocal] double a[N];
upc_forall (i=0; i<N; ++i; a[i]) {
   a[i] = 0.0;
}
```

- Available on most (modern) HPC systems