An Introduction to Multicore Programming and the NAG Library for SMP & multicore

Craig Lucas

November 2011
Agenda

- So what is multicore?
- The NAG Library for SMP and Multicore
- A brief introduction to OpenMP
MULTICORE
Terminology

- **Processor** – consists of cache memory, registers, functional units etc. Everything required for computation.
- **Socket** – is an alternative term to processor. Socket is used more in HPC, and avoids confusion between processors and cores.
- **Core** – is what we formally referred to as a processor when they were single core. The functional units, registers and lower levels of cache.
- **Node** – or shared memory node, is a collection of sockets and memory that is shared between them.
Terminology

- A node can be just one multicore processor, sharing memory. It is thus suitable for programming with OpenMP.

- Shared Memory – We use this term to describe any environment where a collection of cores can all access the same memory.

- SMP – This stands for *Symmetric Multi-Processing*. This could describe a multicore processor, as architecturally each core has equal access to the memory. (Some would argue that the latency is equal too. Or that that it is *Shared Memory Programming*)
Terminology

- NUMA – Non-Uniform Memory Access. In a “NUMA Node” the access time to memory is not equal. Perhaps:
  - a node consisting of two multicore processors each with their own memory
  - but the memory is shared, the cores on one processor can access the memory on the other
  - a core can access a data element in its own memory quicker than it can access one in the other's processors memory
- Due to cache effects, latencies can vary on a single multicore processor.
Terminology

- What is important when it comes to hardware in shared memory is the simple conceptual view:
Terminology

- We ignore that cache memory comes in a variety of “levels” (L1, L2, ...) and some of this may actually be shared.

- The overall principles are:
  - There are (in this case) 4 cores that can do some computation independently to and concurrently with the other cores
  - Data exists once in the shared memory and all cores can access this data
  - Data can be replicated in the separate caches. (Data is copied through cache, to the registers, where it is used by the functional units)
Terminology

- Data is moved in *cache lines*. The cache line containing the data element required is moved.

![Diagram showing cache lines and shared memory](image)
NUMA Architecture

- All cores can access all memory.
- But speed of data movement is not equal.
Current and Emerging Architectures

- **Intel Sandy Bridge (Core i5, Core i7, etc)**
  - Up to 8 physical cores running up to 16 *virtual* or *logical* cores through hyper-threading
  - Pairs of logical cores have separate registers but shared computational units

- **AMD Bulldozer (Opteron)**
  - Halfway between conventional processor and Sandy Bridge
  - Consists of *modules* of two of tightly coupled cores sharing floating point units and instruction handling
  - Each core has its own registers, hardware integer handling and own L1 cache
Current and Emerging Architectures

- **Intel MIC - Knights Corner**
  - Will provide over 50 cores, via PCIe, each with 4 hyper-threads (assuming it follows design of Knights Ferry)
  - Each core at around 1 GHz uses *vector* instructions. Each will have its own L1 cache and share an L2 cache
  - Importantly can run legacy code with little or no modification (data movement, Intel libraries)

- **AMD Fusion APU (Accelerated Processing Unit)**
  - Initial release this year, it joins AMD and ATI hardware to give 1-4 conventional cores and 10’s of GPU-like cores
  - All cores on the same *die*, so faster data movement than MIC (but more new coding required)
An Example – AMD Phenom

- 4 cores
- Private Level 1 and 2 cache (512KB L2 cache)
- Shared 2MB L3 cache
- Connections off to memory and for I/O
Shared Memory Parallelism: Strong Points

- **Dynamic Load balancing**
  - Amounts of work to be done on each processor can be adjusted during execution
  - Closely linked to a dynamic view of data

- **Dynamic Data view**
  - Data can be ‘redistributed’ on-the-fly
  - Redistribution through different patterns of data access

- **Portability**
- **Modularity**
- **Good programming model**
Shared Memory Dynamic View of Data

Computation Stage 3

Data

Core 1
Core 2
Core 3
Core 4
Shared Memory Parallelism: Weaker Points

- Not very suitable for heterogeneous parallelism
- Not very suitable for complex applications
- Not easy to generate efficient code
- Non-deterministic results
  - Applies to least significant parts of solution
  - May be disconcerting to some users
  - Effects on ill-conditioned problems may be dramatic
THE NAG LIBRARY FOR SMP & MULTICORE
NAG Library for SMP & Multicore

- Based on standard NAG Fortran Library
  - Designed to better exploit SMP architecture
  - Mark 23 available soon

- Identical interfaces to standard Fortran Library
  - just re-link the application
  - easy access to parallelism for non-specialists
    - user is shielded from details of parallelism
    - assists rapid migration from serial code
  - can be used along with user’s own parallelism
    - for “expert” users
Target systems

- Multi-socket and/or multi-core SMP systems:
  - AMD, Intel, IBM, SPARC processors
  - Linux, Unix, Windows operating systems
  - Standalone systems or within nodes of larger clusters or MPPs

- Other possibilities:
  - Traditional vector (Cray X2, NEC SX8, etc)
  - Virtual Shared Memory over clusters in theory, but efficiency may be poor on many algorithms due to extreme NUMA nature of such configurations

- Notable exceptions:
  - IBM Cell?
  - Sun Niagara I
  - GPUs, FPGAs, etc
Interoperability

- We want to bring benefits of SMP to multiple environments

- Currently:
  - Fortran 77 and 90+ (with interface blocks available)
  - From C programs via NAG C Header files
  - Toolbox for Matlab (Windows now, Linux soon)

- Investigating technical issues for other possibilities:
  - Better C support via NAG C Library interfaces (“C SMP Library”)
  - Excel
Compatibility issues

- NAG Library for SMP & Multicore uses OpenMP to implement parallelism
  - More on OpenMP later!
- Can work in co-operation with user’s own OpenMP parallelism, or with MPI
- Generally not compatible with other threading models, e.g. POSIX or Windows threads
Parallelism in user’s own code

- What is most important: Throughput on multiple tasks or turnaround on a single task?
- If you require many runs of a program over different parameters, often more efficient to run individual jobs concurrently, rather than attempting to parallelize the individual task.
- If the number of tasks >> number of cores, load balancing should be automatic.
Reasons to parallelise individual tasks

- Reduce runtime for a single task
  - e.g. for Iterative processes

- Limited memory per core may restrict the number of independent tasks that can run side by side
  - Then look to make each task run faster, and to avoid wasted (idle) cores
  - Hardware trends may mean less RAM/core on average
    - e.g. SMP compute nodes on HECToR: 2-core, 3GB/core -> 4-core, 2GB/core -> 24-core, ~1.3 GB/core -> 32-core, 1 GB/core

- Greater number of slower cores -> need higher levels of parallelism to get same performance!?
Using the Library

- Distribute static (.a) & shareable (.so) libraries
- If the libraries are in the linker search path
  
gfortran -O3 -m64 -fPIC -fopenmp driver.f -lnagsmp
  -lacml_mp -lacml_mv

  gfortran -O3 -m64 -fPIC -fopenmp driver.f -Wl,-Bstatic -lnagsmp
  -lacml_mp -lacml_mv -Wl,-Bdynamic

  cf. serial library linkage (taking .so as an example):
  gfortran -O3 -m64 -fPIC driver.f -lnag_acml -lacml

- Set OpenMP environment variable
  
  setenv OMP_NUM_THREADS 16

  and run application
Using the Library

- The library is installed at /opt/nag/fsxxx22xxl (or /usr/local/fsxxx22xxl)

- If the libraries aren’t in the linker search path

```bash
gfortran -O3 -m64 -fPIC -fopenmp driver.f
/opt/nag/fs16a22dal/lib/libnagsmp.so -lacml_mp -lacml_mv
```

```bash
gfortran -O3 -m64 -fPIC -fopenmp driver.f
/opt/nag/fs16a22dal/lib/libnagsmp.a -lacml_mp -lacml_mv
```

- Pick up shareable library at runtime using

  `% export LD_LIBRARY_PATH=/opt/nag/fsxxx22xxl/lib`

- Pick up license using

  `% export NAG_KUSARI_FILE=/opt/nag/license/kusari.dat`
Using the Library

- Read “Users’ Note” for details of your installation. In /opt/nag/fsxxx22xx1/doc or on NAG websites
  - http://www.nag.co.uk/numeric/FL/FSinuns.asp

- Documentation installed or easy to navigate version on NAG websites. Read:
  - “Essential Introduction”
  - Chapter Introductions
  - Routines Documents

- Interface blocks included for Fortran 90+ for compile checks of arguments and their type.
Running the Examples

- Examples are included for all the routines.
- Scripts are available in `/opt/nag/fsxxx22xxl/doc` to run the examples: `nagsmp_example` (static) and `nagsmp_example_shar`
- Data copied to current directory and compilation echo’d to screen.
- Easy way to see how the library should be linked to.
- Output can be checked by comparing with results given in `/opt/nag/fsxxx22xxl/examples/results/yyyyyyye.r` where `yyyyyy` is routine name.
- For example:
F07ABF Example Program

- Running the example from home on 2 threads:

```
home> nagsmp_example f07abf 2

Copying f07abfe.f to current directory
cp /opt/nag/fs16a22dal/examples/source/f07abfe.f .

Compiling and linking f07abfe.f to produce executable f07abfe.exe
gfortran -O3 -m64 -fPIC -fopenmp f07abfe.f -o f07abfe.exe -lnagsmp
   -lacml_mp -lacml_mv

OMP_NUM_THREADS=2 ; export OMP_NUM_THREADS

Copying f07abfe.d to current directory
cp /opt/nag/fs16a22dal/examples/data/f07abfe.d .

Running f07abfe.exe with data from f07abfe.d
./f07abfe.exe < f07abfe.d > f07abfe.r
```
F07ABF Example Program

F07ABF Example Program Results

Solution(s)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0000</td>
<td>3.0000</td>
</tr>
<tr>
<td>2</td>
<td>-1.0000</td>
<td>2.0000</td>
</tr>
<tr>
<td>3</td>
<td>3.0000</td>
<td>4.0000</td>
</tr>
<tr>
<td>4</td>
<td>-5.0000</td>
<td>1.0000</td>
</tr>
</tbody>
</table>

Backward errors (machine-dependent)

8.6E-17   9.7E-17

Estimated forward error bounds (machine-dependent)

2.5E-14   3.7E-14

A has been row scaled as diag(R)*A

Reciprocal condition number estimate of scaled matrix

1.8E-02

Estimate of reciprocal pivot growth factor

7.4E-01

Compare with

/opt/nag/fs16a22dal/examples/results/f07abfe.r
NAG Library Parallel Functionality

- Root Finding
- Summation of Series (e.g. FFT)
- Quadrature
- Ordinary Differential Equations
- Partial Differential Equations
- Numerical Differentiation
- Integral Equations
- Mesh Generation
- Interpolation
- Curve and Surface Fitting
- Optimisation
- Approximations of Special Functions
- Wavelet Transforms

- Dense Linear Algebra
- Sparse Linear Algebra
- Correlation and Regression Analysis
- Multivariate Analysis of Variance
- Random Number Generators
- Univariate Estimation
- Nonparametric Statistics
- Smoothing in Statistics
- Contingency Table Analysis
- Survival Analysis
- Time Series Analysis
- Operations Research
Coming next year in Mark 23

- New parallel routines in the areas of:
  - Particle Swarm Optimization
  - Pseudo Random Number Generator
  - 2D wavelet routines
  - 4D and 5D interpolation routines
Exploiting SMP parallelism (1)

- Core-math routines (LAPACK, FFTs)
  - We aim to give best combination of vendor library and NAG routines
  - Choice varies from platform to platform
    - NAG Library version may be faster on some platforms
    - If not, we recommend you just use the relevant vendor library
    - In particular, NAG works with AMD on ACML, hence all NAG SMP LAPACK routines are available in ACML
  - NAG FFT routines provide a portable interface to different underlying vendor FFT routines
    - No BLAS-equivalent standard for FFT interfaces
NAG & vendor libraries (e.g. ACML, MKL)
LU factorisation (DGETRF)

Performance (Mflops)

Problem size (N)

netlib DGETRF + ACML BLAS
LU factorisation (DGETRF)
S.V.D. (DBDSQR)

Improved serial performance too

![Graph showing performance (Mflops) vs problem size (N) for different NCPUs and libraries.]

- netlib DBDSQR + ACML BLAS
- SMP DBDSQR + ACML BLAS
Exploiting SMP parallelism (2)

- NAG routines which use core-math routines
  - Exploit parallelism in underlying BLAS, LAPACK and FFT routines where possible
  - Development programme includes renovation of existing routines as well as adding new functionality
- Following on from (1), best choice of NAG Fortran Library vs NAG Library for SMP & Multicore varies from platform to platform
G03AAF: Principal Component Analysis

Calls parallel SVD

AMD Barcelona 2.0 GHz, N=10000, M=2000
C05NCF: Non-linear equation solver

Intel Xeon E5310 1.6 GHz, N=4000

Number of cores

Time (secs)

Fortran Library Mark 21 + MKL
Fortran Library Mark 22 + MKL
QR Factorisation in MKL

MKL parallelized
Exploiting SMP parallelism (3)

- NAG-specific routines parallelised with OpenMP
  - Focus of future NAG SMP library development work
  - Continuing to broaden scope of parallelism to different parts of the library
E01TGF/E01THF: Interpolation

Intel Xeon E5310 1.6 GHz, N=100,000

E01THF: evaluate interpolant at given points
E01TGF: generate 3D interpolant

Time (secs)

Number of cores

1 2 4 8
C09EAF: 2-D discrete wavelet transform

AMD Magny-Cours, dual socket, 2.2 GHz
M=N=15000, DB8, Half, Forward
Future algorithms

- Not all algorithms can be parallelised
  - Thus it may be better to replace existing routines rather than try to parallelise them
  - NAG Global Optimization: introduced a new routine that was less efficient in serial but was parallelizable

- Potential for parallelism is now a key criteria for selecting future algorithms
  - Accuracy and stability still most important
  - Note: Not all algorithms are computationally demanding enough to need to be parallelised
Performance considerations

- In ideal world, performance would scale linearly with the number of cores used
  - Sadly this is rarely achieved

- Performance and scalability depends upon
  - Nature of algorithm
  - Problem size(s) and other parameters
  - Hardware design
  - OS, compiler and load on system

- These factors are interconnected!
Performance considerations

- Choice of algorithm, i.e global optimization
- Nature of algorithm
  - Some algorithms *embarrassingly parallel*, others require threads to synchronise frequently
- Not all parts of an algorithm may have been or can be parallelised, *Amdahl’s Law.*
- Problem size(s)
  - Spawning threads incurs an overhead that may be significant for small problem sizes
  - In some cases we try to determine an appropriate threshold and only parallelise problems bigger than this
Performance advice

- In general, don’t oversubscribe the cores
  - Sharing is bad!
- Invest time to benchmark commonly used problem sizes/parameters
  - e.g. calculation performed during every timestep of long simulation
- Do this on the system you plan to use for real calculations if possible
Performance advice

- For NAG library routines
  - Consult documentation to see which routines have been parallelised
  - Also which routines may get some benefit because they internally call one or more of the parallelised routines
  - Library introduction document gives some extra advice one using some of the parallelised routines
  - Consult NAG for advice if required
  - Feedback on which routine(s) you use and typical problem parameters etc is very useful for us for planning
NAG Library for SMP & Multicore: Summary

- Shared Memory systems now the norm
  - in large part due to multi-core chips
- NAG Library for SMP & Multicore provides an easy-to-use option for exploiting SMP hardware
  - Identical interfaces to standard NAG Fortran Library
  - Interoperable with other languages
  - Works with vendor core-math library to get best performance on dense linear algebra and FFT routines
  - Increasing number of NAG-specific routines parallelised
  - Potential for parallelism key criteria for future routines
- Mark 22 available, Mark 23 next year
A BRIEF INTRO TO OPENMP
OpenMP

- We will give just a brief overview of OpenMP.
- Our aim is to give you enough information to decide if you need to learn more.
- We will look at two code segments:
  - Hello world
  - Parallel data generation (for a NAG Multicore routine, perhaps)
What is OpenMP?

- OpenMP (Open Multi-Processing) is a specification for shared memory parallelism.
- That is, it is a mechanism for writing multi-process/or multi-thread code for SMP machines.

OpenMP uses:

- Compiler Directives
- Environment Variables, and
- Run-time Library Routines
OpenMP on Linux

- OpenMP has been around since 1997, so the compilers are relatively advanced.

- Compilers available from:
  - Intel
  - gcc (from version 4.2, 4.4 for v3.0)
  - Portland Group compilers
  - Pathscale
  - Sun Studio Compilers
  - Fujitsu
Status

- OpenMP specification is an *agreement* between vendors and users and *not* a formal standard.
- Version 2.5 (May 2005, combined Fortran, C/C++ standards.)
- Version 3.0 (May 2008,) with major addition of tasks:
  - Many more opportunities for parallelism, like recursive algorithms or producer/consumer schemes
  - Defines work to be done and associated data
- Version 3.1 (July 2011)
  - Improvements to tasks
  - The start of affinity support, binding threads to cores etc
OpenMP Execution Model

- OpenMP uses the “fork-join” model:
  - start as a single (master) thread, on one core
  - continue as single thread until parallel construct
  - create required number of threads, on remaining cores
  - program statements within this parallel region are executed in parallel by each thread in this team of threads (includes master thread and slaves)
  - at end of the parallel construct, threads synchronize and only master thread continues execution, on the original single core

- Can have many parallel regions, although there is an overhead with thread creation/destruction.
Execution Model

Parallelism carried out in distinct parallel regions

Spawn threads

Multi-threading in parallel region, on all multicores

Destroy threads

Serial execution, on a single core

Serial execution
Three elements of OpenMP

- Recall we use three elements when programming OpenMP.
- In the most part we use (1) Compiler Directives to initialize our threads and distribute the work.
- We also need to set (2) Environment Variables, although this may only be just one variable.
- And (3) Run-time Library Routines, for getting the number of threads we have, or timing our parallel code. (Which also requires a header file.)
Three elements of OpenMP

- We add (1) and (3) to our existing serial code.
- This makes OpenMP parallelisation much easier than, say, MPI. But more importantly:
- We do not (necessarily) destroy our serial code so we can compile it for, and run it on, serial platforms.
- We can also incrementally add parallelism to a serial code. Starting with the most computationally expensive.

- Lets take a look in more detail...
Three elements of OpenMP

- So first up is **Compiler Directives**.
- The parallel directive defines the parallel region, the code that will be executed in parallel on different threads.
- Fortran example:
  
  ```fortran
  !$OMP PARALLEL [clause1, clause2, ...]  
  < code in here >  
  !$OMP END PARALLEL
  ```

- C/C++ example:
  
  ```c
  #pragma omp parallel [clause1, ...] new-line
  ```

- All directives have optional clauses
Three elements of OpenMP

- Secondly, we have Run Time library Routines.
- There are two very commonly ones here

  ```
  integer function omp_get_num_threads()
  int function omp_get_num_threads(void)
  ```

  Gets the current number of threads. (Will return 1 outside of a parallel region.)

  ```
  integer function omp_get_thread_num()
  int function omp_get_thread_num(void)
  ```

  Get the unique thread number.
Hello World in Fortran

```fortran
PROGRAM HELLO
  IMPLICIT NONE
  INCLUDE "omp_lib.h"
  INTEGER :: iam, nthreads

  !$OMP PARALLEL default(none), private(iam,nthreads)
  iam = omp_get_thread_num()
  nthreads = omp_get_num_threads()
  WRITE(*,*) 'Hello World from thread ', iam, 'of', &
               nthreads, 'threads'

  !$OMP END PARALLEL

END PROGRAM HELLO
```

I have added the red directive to my existing serial Hello World code. The directive spawns threads – we now have 4 (see later) threads working.
Hello World in C

#include <stdio.h>
#include <omp.h>
int main () {
int nthreads;
int thread_no;

#pragma omp parallel default(none) private(iam, nthreads)
  {
    nthreads = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello World from thread %d of %d threads \n", \n          iam, nthreads);
  }

return 0;
}"
Hello World in Fortran

PROGRAM HELLO
  IMPLICIT NONE
  INCLUDE "omp_lib.h"
  INTEGER :: iam, nthreads

  !$OMP PARALLEL default(none), private(iam,nthreads)
      iam = omp_get_thread_num()
      nthreads = omp_get_num_threads()
  WRITE(*,*) 'Hello World from thread ', iam, ' of ', nthreads, ' threads'
  !$OMP END PARALLEL

END PROGRAM HELLO
Hello World in C

#include <stdio.h>
#include <omp.h>
int main () {
    int nthreads;
    int thread_no;

    #pragma omp parallel default(none) private(iam, nthreads)
    {
        nthreads = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hello World from thread %d of %d threads \n", \n               iam, nthreads);
    }

    return 0;
}
Hello World

- Output:

  Hello World from thread 2 of 4 threads
  Hello World from thread 0 of 4 threads
  Hello World from thread 1 of 4 threads
  Hello World from thread 3 of 4 threads

- Note the order is “random”
- The master thread is number 0
- Before we ran the code we use the third element, Environment Variables, and set the number of threads with OMP_NUM_THREADS. In Linux:

  ```bash
echo OMP_NUM_THREADS=4
```
OMP_NUM_THREADS in Visual Studio

- While developing you set the value:

  - Set in a batch file with: `set OMP_NUM_THREADS=4`
  - Or via Control Panel -> System -> Environment Variables
Compiling OpenMP

- In Linux, set the appropriate flag, for example:

  gfortran -fopenmp my_code.f95
  gcc -fopenmp my_code.c
  pgf95 -mp=nonuma my_code.f95 ! options?
  pgcc -mp=bind my_code.c
  ifort -openmp my_code.f95
  icc -openmp my_code.c
Compiling OpenMP

- In Visual Studio use the project's Property Pages dialog box:

  ![Console 1.f90 Property Pages](image)

  - Configuration Properties
    - General
    - Fortran
      - General
      - Optimization
      - Debugging
      - Preprocessor
      - Code Generation
      - Language
      - Compatibility
      - Diagnostics
      - Data
      - Floating Point
      - External Procedures
      - Output Files
      - Run-time
      - Command Line
  - Process OpenMP Directives
    - Generate Parallel Code (/Qopenmp)

Enables the compiler to generate multi-threaded code based on the OpenMP directives. (/Qopenmp, /Qopenmp-stubs)
Worksharing

- So we have seen how to use the three elements to write a simple parallel hello world program.

- We now look at how to actually divide up work to be done to so it is done faster than on one core. Which is the whole point of course!
Worksharing

- For this we use a work sharing constructs. The most common one is for loops, “do” and “for.”

- In Fortran the syntax is:

```fortran
!$OMP DO [clause1, clause2, ...]
DO ...
  ! Existing serial for loop
  body_of_do_loop
END DO
!$OMP END DO
```

- And in C/C++ it is:

```c
#pragma omp for [clauses] new-line
for ...
  // Existing serial for loop
  body_of_do_loop
```
Worksharing

- What these work sharing constructs do is simply divide out the work in the loop that immediately follows the directive.
- This is between the available threads running on the cores of our multicore socket or node.
- Like other directive there are options that we will not look at here.
- Here is an example...
A Parallel Loop in Fortran

!$OMP PARALLEL default(none), shared(a,b), private(i,j)
!$OMP DO
do j = 1, n
  do i = 1, n
    a(i,j) = a(i,j) + b(i,j)
  end do
end do
!$OMP END DO
!$OMP END PARALLEL

CALL F07ADF(M, N, A, LDA, IPIV, INFO)
A Parallel Loop in C

```c
#include <nagmk22.h>

#pragma omp parallel default(none) shared(a,b,n) private(i,j)
{
    #pragma omp for
    for (i=1; i<=n; i++)
    {
        for (j=1; j<=n; j++)
        {
            a[i][j] += b[i][j]
        }
    }
}

f07adf_(&m, &n, (double *) a, &lda, ipiv, &info)
```

Spawn threads - we now have 4 threads working

The outer i loop is executed in parallel, with each thread taking a subset of the iterations
A Parallel Loop

- So what is happening?
- Well something like this, for n=200

Thread 0

\[
\text{do } j = 1, 50 \\
\quad \text{do } i = 1, n \\
\quad \quad a(i, j) = a(i, j) \\
\quad \text{end do} \\
\text{end do}
\]

Thread 1

\[
\text{do } j = 51, 100 \\
\quad \text{do } i = 1, n \\
\quad \quad a(i, j) = a(i, j) + b(i, j) \\
\quad \text{end do} \\
\text{end do}
\]

Thread 2

\[
\text{do } j = 101, 150 \\
\quad \text{do } i = 1, n \\
\quad \quad a(i, j) = a(i, j) \\
\quad \text{end do} \\
\text{end do}
\]

Thread 3

\[
\text{do } j = 151, 200 \\
\quad \text{do } i = 1, n \\
\quad \quad a(i, j) = a(i, j) + b(i, j) \\
\quad \text{end do} \\
\text{end do}
\]
A Parallel Loop

- The four threads have each taken a share of the work to do.
- Recall that all threads can access all the shared data, like the arrays in this example.
- We need to consider something called “scoping” of the variables too.
- This is what we have done with the “shared” and “private” clauses in the previous example which actually shows the usual scoping of arrays and loop counters.
Variable Scoping

- All threads need to access the arrays, these are in a **shared** memory. Every thread accesses the data, but note that they don’t write to the same array elements. This is very important.

- The threads will need **private** copies of the loop variables as they independently execute the loops, and in particular set the value of these variables. Consider if they shared the variables, the threads would get “confused”, each trying to update them.

- We use **default**(none) to tell the compiler we are going to explicitly scope all variables.
That was easy!

- Yes, OpenMP parallelism can be that simple.
- But we have left out a lot of information.
- In particular race conditions, cache coherency, data dependency etc etc
- We would need much more time to talk in detail about the memory subsystem to get into these now.
Lots more OpenMP too

- There is a lot of OpenMP we have not covered, like:
  - Sections, of parallelisation independent code blocks
  - Tasks
  - Loop scheduling, if work is uneven in loop iterations
  - Thread synchronisation, barriers, ensuring shared data is updated consistently
  - Reductions, like a sum within a loop
  - And all the variable scoping options and rules

- But we hope we have whetted your appetite!
Learn More

- Google (naturally)
- Books:
  "Parallel Programming in OpenMP" by Rohit Chandra et al.
  "Using OpenMP" by Barbara Chapman et al.
- Learn more on a 2 day HECToR OpenMP course, see www.hector.ac.uk/cse/training (Imperial Jan, UCL May 2012)