Parallel Programming with OpenMP

Hinnerk Stüben

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Universität Bremen
13–18 September 2025

Contents

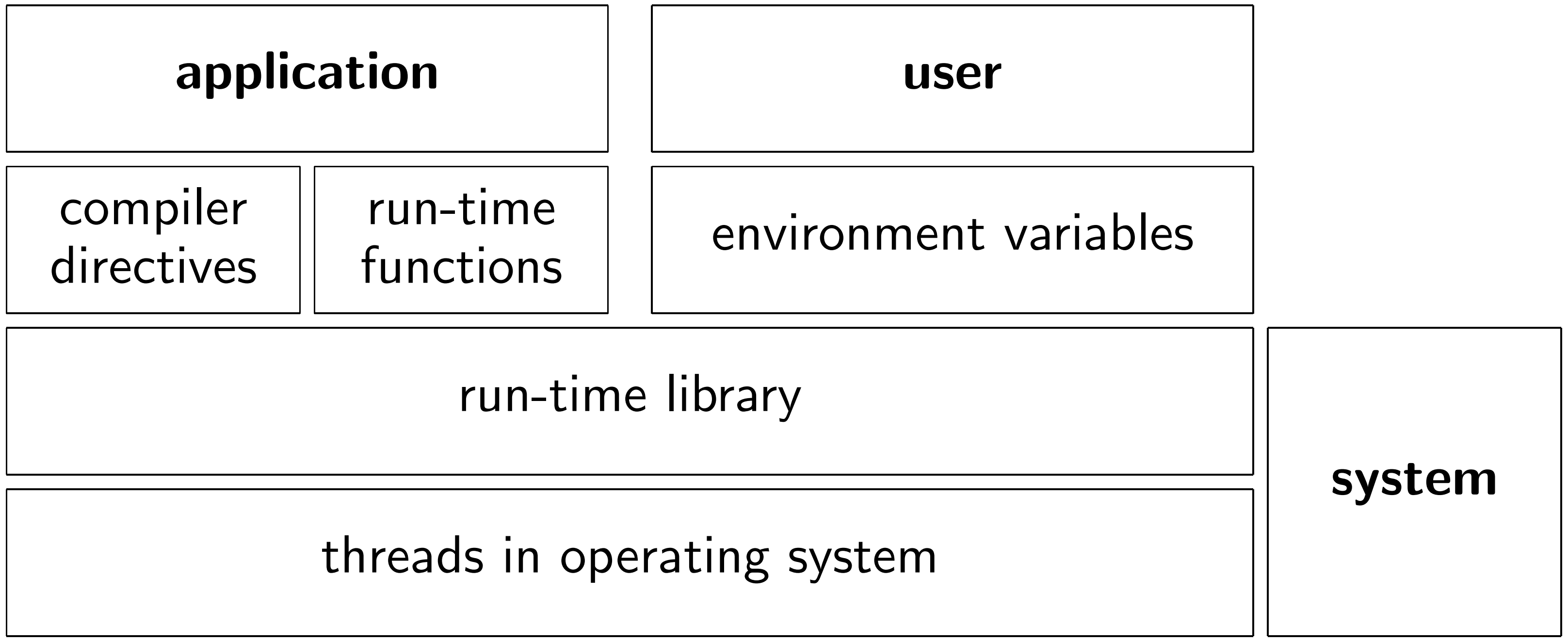
Introduction

- programming model for shared memory computers
- interfaces for Fortran and C/C++
- industry standard
- OpenMP programs
  - are portable
  - can be parallelised step by step (e.g. loop by loop)
- `http://www.openmp.org`

  → Specifications
  → Presentations

Concepts (I)
– software architecture, execution model –

Software architecture (I)



Software architecture (II)

- compiler directives (or pragmas)
  - are a kind of extension to existing programming languages
  - serve to express parallel execution
  - (can have *clauses*)

- environment variables
  - are used to set execution parameters

- run-time functions
  - allow to enquire and modify execution parameters
  - (provide additional functionality like time measurement or locks)

Elements of a first OpenMP program

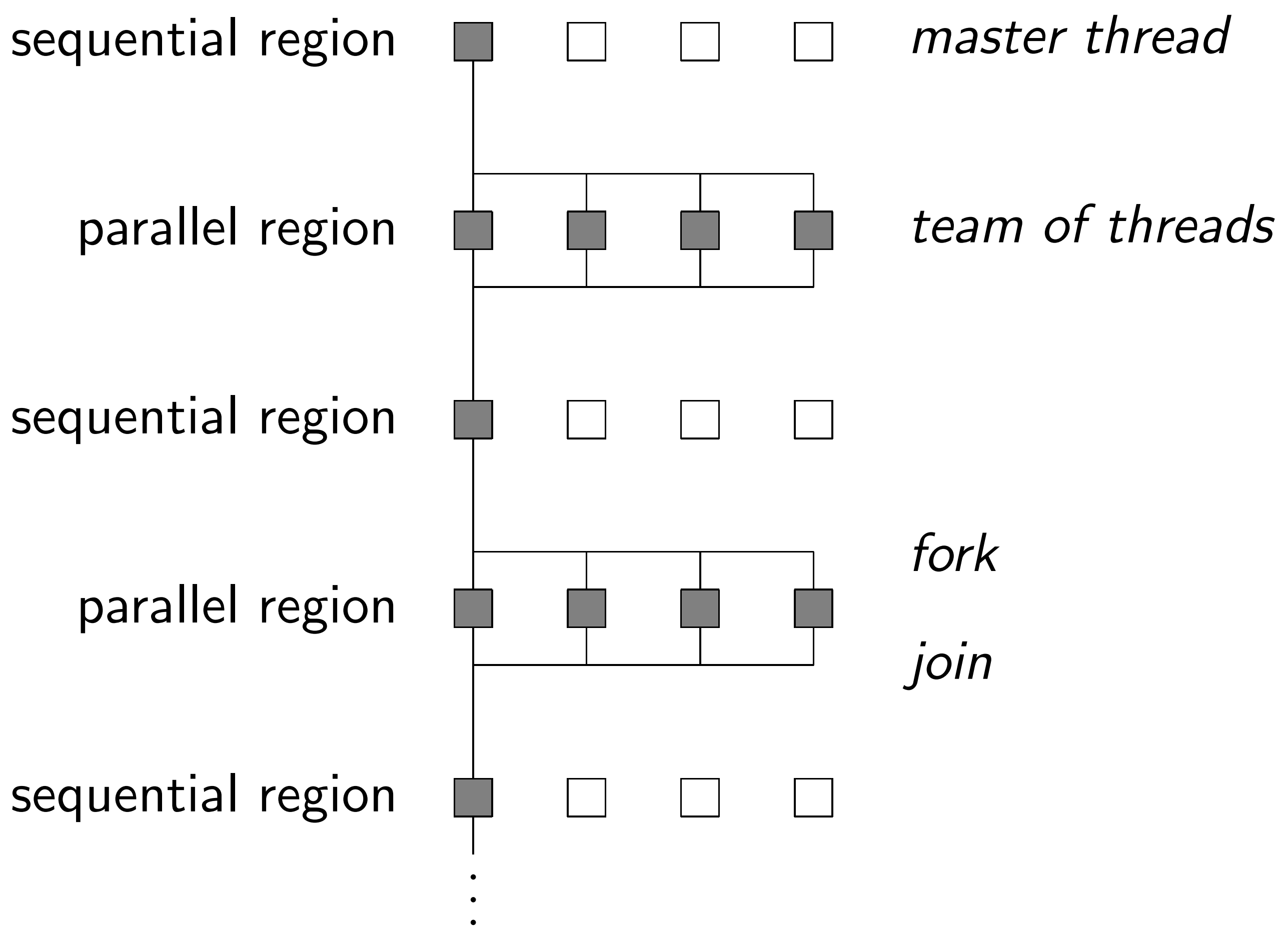
- compiler directive `parallel`
- run-time function `omp_get_thread_num()`
- environment variable `OMP_NUM_THREADS`

First OpenMP program

```fortran
program main
  use omp_lib

  !$omp parallel

  write(6,*) omp_get_thread_num()

  !$omp end parallel

end
```

```c
# include <stdio.h>
# include <omp.h>

int main(int argc, char *argv[])
{

  #pragma omp parallel
  {
  printf("%d\n", omp_get_thread_num());

  }

}
```

Program execution model



Explanation of the first OpenMP program

- `omp parallel` embraces a parallel region
- `OMP_NUM_THREADS` sets the size of the team of threads
- `omp_get_thread_num()` returns the thread number

  range: `0 .. (OMP_NUM_THREADS - 1)`

Concepts (II)
– writing code –

Directive format

- C/C++:

```
#pragma omp directive-name [clause[[,] clause ...]] new-line
```

continuation lines:

```
#pragma omp ... \
            ... \
            ...
```

- Fortran90:

```
!$omp directive-name [clause[[,] clause ...]]
```

continuation lines:

```
!$omp ... &                   !$omp ... &
!$omp& ... &        or        !$omp ... &
!$omp& ...                    !$omp ...
```

Blocks

The scope of OpenMP directives is typically one block of statements.

- C:
  - scope is given by a C block: `{ ... }`
- Fortran:
  - scope is given by loop construct: `do ... enddo`
  - or end of block is marked by: `!$omp end ...`

Jumping out of an OpenMP block (`goto`, `setjmp`, `throw`, . . .) is *not* allowed.

Exceptions: `stop` in Fortran and `exit` in C.

Run-time library definitions

- C
  - `#include <omp.h>`
- C++
  - library routines are external functions with external “C” linkage
- Fortran
  - `include 'omp_lib.h'`
  - `use omp_lib`

Conditional compilation

- C preprocessor:

  predefined variable **`_OPENMP`**

- Fortran90:

```
!$ ...                  single line

!$ ... &                continuation lines
!$& ... &
!$& ...
```

have the same effect as the C preprocessor `#ifdef`:

```
#ifdef _OPENMP
   ...
   ...
#endif
```

Conditional compilation – examples

- C:

```
#ifdef _OPENMP
# include <omp.h>
#endif

...

#ifdef _OPENMP
thread_num = omp_get_thread_num();
#else
thread_num = 0;
#endif
```

- Fortran90:

```
!$ use omp_lib
```

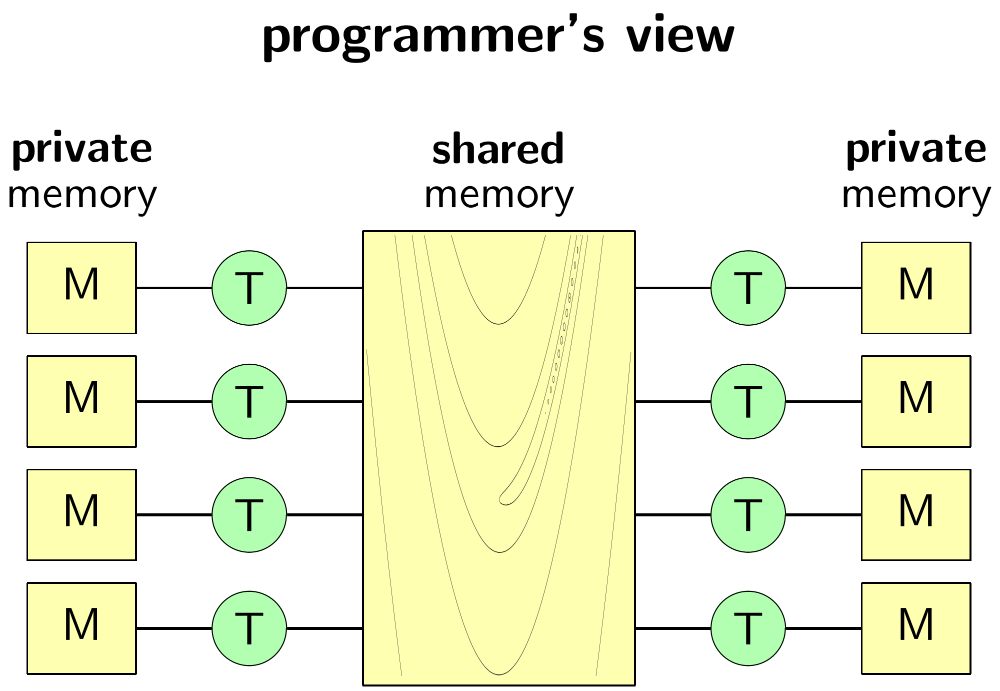
Concepts (III)
– memory model, synchronisation –

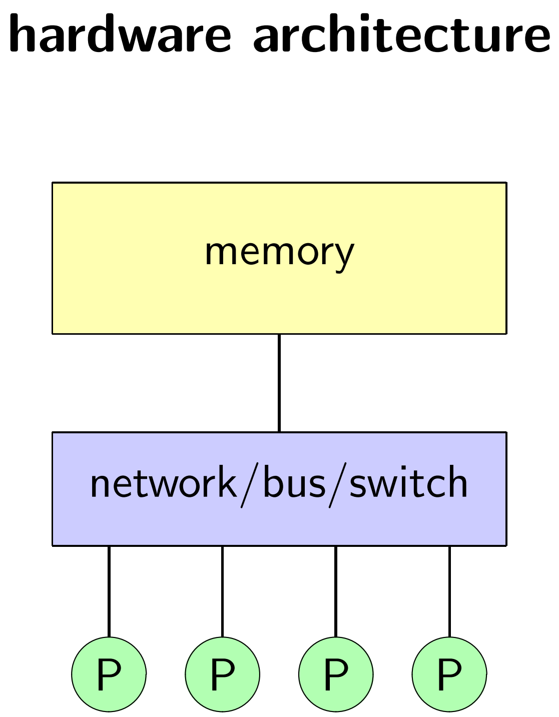
Second OpenMP program

```fortran
program main
  use omp_lib
  integer i

  !$omp parallel private(i)

  i = omp_get_thread_num()
  write(6,*) "thread_num = ", i

  !$omp end parallel

end
```

```c
# include <stdio.h>
# include <omp.h>

int main(int argc, char *argv[])
{
  int i;

  #pragma omp parallel private(i)
  {
  i = omp_get_thread_num();
  printf("thread_num = %d\n", i);

  }

}
```

Memory model





- logically there is *private* and *shared* memory
- at the hardware level all memory is shared

Synchronisation

- threads communicate via shared variables
- (read/write) access to shared variables by more than one thread must be coordinated (by the programmer)
- this kind of coordination is called *synchronisation*
- missing synchronisation results in *race conditions*:
  - if two threads write to the same memory location the sequence of writes is accidental *(output dependence)*
  - if one thread reads and another thread writes to the same memory location the sequence is accidental *(flow- or anti-dependence)*
- the main mechanisms for synchronisation are provided by the `barrier` and `critical` directives

Third OpenMP program

```fortran
program main
  use omp_lib
  integer sum

  sum = 0
  !$omp parallel shared(sum)

  !$omp critical

  sum = sum + omp_get_thread_num()

  !$omp end critical
  !$omp end parallel

  write(6,*) "sum = ", sum
end
```

```c
# include <stdio.h>
# include <omp.h>

int main(int argc, char *argv[])
{
  int sum;

  sum = 0;
  #pragma omp parallel shared(sum)
  {
  #pragma omp critical
  {
  sum += omp_get_thread_num();

  }
  }

  printf("sum = %d\n", sum);
}
```

Parallelising loops (I)
– data dependence analysis –

The `parallel do/for` directive

- One strategy to use OpenMP is to parallelise at loop level.
- This can be easily accomplished with the `parallel do/for` construct. It starts and ends a parallel region for the execution of the loop directly following the directive, and distributes the work.
- We will apply this construct to the loops we have studied in the *Thinking Parallel* section.
- The focus of this section is the question whether loops can be parallelised (have no data dependences). This is the first question, the **programmer** has to answer!
- At the same time one sees that using OpenMP is easy in many cases.

Work-sharing with `parallel do/for`

- pseudo code:

```
omp parallel for
for i := 1 to 300 do
   a[i] := b[j[i]]
```

→ work is **decomposed** by compiler and run-time system (here for 3 threads):

```
for i:= 1 to 100 do    for i:=101 to 200 do    for i:=201 to 300 do
   a[i] := b[j[i]]        a[i] := b[j[i]]         a[i] := b[j[i]]
```

Effect of `parallel` alone

- pseudo code:

```
omp parallel
for i := 1 to 300 do
   a[i] := b[j[i]]
```

→ work is **replicated**:

```
for i:= 1 to 300 do       for i:= 1 to 300 do       for i:= 1 to 300 do
   a[i] := b[j[i]]           a[i] := b[j[i]]           a[i] := b[j[i]]
```

Remark on `parallel do/for`

- The construct

```
omp parallel do/for
...
omp end parallel do/for
```

is equivalent to

```
omp parallel
omp do/for
...
omp end do/for
omp end parallel
```

Loop 1

Simple loop:

```
!$omp parallel do
do i = 1, N
   a(i) = b(i)
enddo
```

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    a[i] = b[i];
}
```

Loop 2

Gather loop:

```
!$omp parallel do
do i = 1, N
   a(i) = b(j(i))
enddo
```

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    a[i] = b[j[i]];
}
```

Loop 3 (I)

Scatter loop – if there are no dependences (e.g. if `j` is a permutation of the indices `i`):

```fortran
!$omp parallel do
do i = 1, N
   a(j(i)) = b(i)
enddo
```

```c
#pragma omp parallel for
for (i = 0; i < N; i++) {
    a[j[i]] = b[i];
}
```

Loop 3 (II)

Scatter loop – ordered execution:

```
!$omp parallel do ordered
do i = 1, N
   !$omp ordered
   a(j(i)) = b(i)
   !$omp end ordered
enddo
```

```
#pragma omp parallel for ordered
for (i = 0; i < N; i++) {
    #pragma omp ordered
    a[j[i]] = b[i];

}
```

- The loop is executed *sequentially!*
- `ordered` is an optimisation for NUMA architectures (improvement of data locality).

Loop 3 (III)

Scatter loop – sequential execution for unkown `j`:

```
!$omp parallel
...
...
!$omp barrier
!$omp master
do i = 1, N
   a(j(i)) = b(i)
enddo
!$omp end master
!$omp barrier
...
...
!$omp end parallel
```

```
#pragma omp parallel
{
...
#pragma omp barrier
#pragma omp master
{
for (i = 0; i < N; i++)
    a[j[i]] = b[i];
}  // end master
#pragma omp barrier
...
...
}  // end parallel
```

- First barrier: wait for completion of shared work on `b`.
- Second barrier: wait for completion of `master` work on `a`.

Loop 4

Atomic update loop:

```
!$omp parallel do
do i = 1, N
   !$omp atomic
   a(j(i)) = a(j(i)) + b(i)
enddo
```

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    #pragma omp atomic
    a[j[i]] += b[i];
}
```

Loop 5

Global sum:

```fortran
s = 0.0
!$omp parallel do reduction(+:s)
do i = 1, N
   s = s + a(i)
enddo
```

```c
s = 0.0;
#pragma omp parallel for reduction(+:s)
for (i = 0; i < N; i++) {
    s += a[i];
}
```

Loop 6 (nearest neighbours I)

Jacobi iteration:

```
!$omp parallel do
do i = 1, N
   a(i) = (b(i-1) + b(i+1)) * 0.5
enddo
```

```
#pragma omp parallel for
for (i = 1; i <= N; i++) {
    a[i] = (b[i-1] + b[i+1]) * 0.5;
}
```

Loop 7 (nearest neighbours II)

Gauß-Seidel iteration:

```
for i := 1 to N do
   a[i] := (a[i - 1] + a[i + 1]) / 2
```

- This loop is not parallelisable in this form.

Loop 8 (nearest neighbours III)

Gauß-Seidel iteration (fine grained chessboard decomposition):

```fortran
do j = 1, 2
  !$omp parallel do
  do i = j, N, 2
    a(i) = (a(i-1) + a(i+1)) * 0.5
  enddo
enddo
```

```c
for (j = 1; j <= 2; j++) {
  #pragma omp parallel for
  for (i = j; i <= N; i += 2) {
    a[i] = (a[i-1] + a[i+1]) * 0.5;
  }
}
```

- **`parallel do/for`** includes a **barrier** at the end of the loop.

Conditions for loops in C

`for (`*`init-expr`*`;` *`var relational-op b`*`;` *`incr-expr`*`)` *`statement`*

`init-expr`	allowed forms:
	`var = lb`
	`integer-type var = lb`
`incr-expr`	allowed forms:
	`++var var++ --var var--`
	`var += incr`
	`var -= incr`
	`var = var + incr`
	`var = incr + var`
	`var = var - incr`
`var`	a signed integer variable according to the definition of the base language, becomes automatically *private*, *`var`* must not be manipulated in the loop body
`relational-op`	allowed forms:
	`< <= > >=`

The `restrict` keyword of C (I)

- consider the following example:

```
subroutine sub(a, b, c, n)

   integer n, i
   real(8) a(n), b(n), c(n)

   do i = 1, n
      a(i) = b(i) + c(i)
   enddo
end

program main

   real(8) x(200)

   ...
   call sub(x(2), x(1), x(3), 100)
   ...
end
```

```
void sub(double *a, double *b,
         double *c, int n)
{
   int i;

   for (i = 0; i < n; i++) {
      a[i] = b[i] + c[i];
   }
}

int main(int argc, char *argv[])
{
   double x[200];

   ...
   sub(&x[1], &x[0], &x[2], 100);
   ...
}
```

The `restrict` keyword of C (II)

- At the syntactic level there are no data dependences in `sub()`.
  At the semantic level there are data dependences. Effectively, the loop reads:

```
do i = 2, n + 1
   x(i) = x(i - 1) + x(i + 1)
enddo
```

```
for (i = 1; i < n + 1; i++) {
   x[i] = x[i - 1] + x[i + 1];
}
```

- In Fortran the statement `call sub(x(2), x(1), x(3), 100)` is illegal!
  The Fortran standard demands that data from parameter lists do not overlap.

- In C calling `sub(&x[1], &x[0], &x[2], 100)` is allowed!
  One can enforce Fortran behaviour by using the `restrict` keyword:

```
void sub(double * restrict a, double * restrict b, double * restrict c,
                                                                 int n)
```

  The `restrict` keyword is standard C.
  → Still one has to check (also in Fortran) that the now illegal call is not made.

Parallelising loops (II)
– `private` variables –

The `private` clause

Example:

```
!$omp parallel do private(tmp)
do i = 1, N
   tmp = a(i) + b(i)
   c(i) = tmp**2
enddo
```

```
#pragma omp parallel for private(tmp)
for (i = 0; i < N; i++) {
    tmp = a[i] + b[i];
    c[i] = tmp * tmp;
}
```

- Typically, all variables are `shared` by default.
- The loop variable `i` is `private` by default.
- `private` variables are uninitialised by default.
- At the end of the region `private` variables are undefined.

The `firstprivate` clause

Example:

```fortran
x(0) = f()
!$omp parallel do private(j) &
!$omp firstprivate(x)
do i = 1, N
   do j = 1, 3
      x(j) = g(i, x(j - 1))
   enddo
   y(i) = x(1) - x(3)
enddo
```

```c
x[0] = f();
#pragma omp parallel for private(j) \
                  firstprivate(x)
for (i = 0; i < N; i++) {
    for (j = 1; j <= 3; j++) {
        x[j] = g(i, x[j - 1]);
    }
    y[i] = x[1] - x[3];
}
```

- Assume that `f()` and `g()` have no side effects.
- `x` is initialised for all threads (once).
- Part of the array `x` is *read-only*.

The **lastprivate** clause

Example:

```
!$omp parallel do lastprivate(i)
do i = 1, N
   ...
enddo

write(6,*) i
```

```
#pragma omp parallel for lastprivate(i)
for (i = 0; i < N; i++) {
    ...
}

printf("%d\n", i);
```

- Updates `i` at the end of the region.
- Can appear together with `firstprivate`.

Overview of `private` clauses

- `private`
  - variables are not initialisied
  - variables are undefined at the end of the region
- `firstprivate`
  - variables are initialisied
  - variables are undefined at the end of the region
- `lastprivate`
  - variables are not initialised
  - variables have correct values at the end of the region
    (i.e. the values they would have in sequential execution)

private declarations in C

Variables with *block scope* are automatically `private`:

```
sum = 0;
#pragma omp parallel for \
    reduction(+:sum) \
    private(x, d)
for (y = 1; y <= Ny; y++) {
    for (x = 1; x <= Nx; x++) {
        d = v1[y][x] - v2[y][x];
        sum += d * d;
    }
}
```

```
sum = 0;
#pragma omp parallel \
    reduction(+:sum)

for (int y = 1; y <= Ny; y++) {
    for (int x = 1; x <= Nx; x++) {
        double d = v1[y][x] - v2[y][x];
        sum += d * d;
    }
}
```

Parallelising loops (III)
– the `reduction` clause –

The `reduction` clause

- Format: `reduction(`*`operator`*`:`*`list`*`)`

- Operators available in C:

```
+  *  -  &  |  ^  &&  ||  max  min
```

- Operators and intrinsic functions available in Fortran:

```
+  *  -
.and.  .or.  .eqv.  .neqv.
max  min  iand  ior  ieor
```

- OpenMP initialises reduction variables appropriately.

- Outside the loop variables listed in *`list`* must be shared.

- If a reduction operation is not available one can use the `critical` directive (see slide 49).

The reduction clause – Fortran example

```
xsum = 0
xsum2 = 0
xmin = x(1)
xmax = x(1)

!$omp parallel do reduction(+:xsum, xsum2) &
!$omp             reduction(min:xmin) &
!$omp             reduction(max:xmax)
do i = 1, N
   xsum = xsum + x(i)
   xsum2 = xsum2 + x(i)**2
   xmin = min(xmin, x(i))
   xmax = max(xmax, x(i))
enddo
```

The reduction clause – C example

```
xsum = 0;
xsum2 = 0;
xmin = x[0];
xmax = x[0];

#pragma omp for reduction(+:xsum, xsum2) \
                reduction(min:xmin) \
                reduction(max:xmax)
for (i = 0; i < N; i++) {
    xsum += x[i];
    xsum2 += x[i] * x[i];
    xmin = x[i] < xmin ? x[i] : xmin;
    xmax = x[i] > xmax ? x[i] : xmax;
}
```

The **reduction** clause – C example (before OpenMP 3.1)

```
xsum = 0; xsum2 = 0; xmin = x[0]; xmax = x[0];

#pragma omp parallel private(pmin, pmax)
{
    pmin = x[0]; pmax = x[0];
    #pragma omp for reduction(+:xsum, xsum2) nowait
    for (i = 0; i < N; i++) {
        xsum += x[i];
        xsum2 += x[i] * x[i];
        pmin = x[i] < pmin ? x[i] : pmin;
        pmax = x[i] > pmax ? x[i] : pmax;
    }
    #pragma omp critical
    {
       xmin = pmin < xmin ? pmin : xmin;
       xmax = pmax > xmax ? pmax : xmax;
    }
}
```

Parallelising loops (IV)
– the `default` clause –

The `default` clause

- One can set a default data-sharing attribute with the `default` clause.
  - C/C++: `default(shared|none)`
  - Fortran: `default(shared|private|none)`
  - `default(none)` forces the programmer to declare the attributes of all variables in the parallel region explicitly.
  - The attributes to be assigned are `shared`, `private`, `firstprivate`, `lastprivate`, or `reduction` respectively.

Parallelising loops (V)

– default data-sharing attributes –

– loops with function/subroutine calls –

Storage classes and data attributes

- Storage classes:
  - *static:* storage exists during the whole run-time of the program
  - *automatic:* storage exists during the execution of a routine or block
- Data scopes:
  - *local:* data is visible only inside a routine or block
  - *global:* data is visible in the whole program
  - *file:* data is visible everywhere in a source file (C only)
- OpenMP data attributes:
  - *shared:* data is visible to all threads
  - *private:* data is visible to the current thread only

Storage classes and data scopes in Fortran

```
real(8) function f(x)

  real(8) :: x
  real(8) :: t1, t2                  ! t1, t2: local, automatic
  integer, save :: count = 0         ! count: local, static

  real(8) :: scratch(10000)
  common /scratch/ scratch           ! scratch: global, static

  count = count + 1
  ...
end
```

Storage classes and data scopes in C

```
extern double scratch[10000];    // scratch: global, static

static scratch2[100];            // scratch2: file scope, static

double f(double x)
{
  double t1, t2;                 // t1, t2: local, automatic
  static int count = 0;          // count: local, static

  count++;
  ...
}
```

Static/lexical extent, dynamic extent, and orphaned directives

- static/lexical extent and dynamic extent (the work of the loop is shared)
- here the orphaned directive has no effect

```
program main

   !$omp parallel
   call sub()
   !$omp end parallel
   call sub()
end

subroutine sub()

   !$omp do            orphaned directive
   do i = 1, N
      ...
   enddo
end
```

```
int main(int argc, char *argv[])
{
   #pragma omp parallel
   sub();

   sub();
}

void sub()
{
   #pragma omp for
   for (i = 0; i < N; i++)
       ...;

}
```

Data-sharing attributes

- By default
  - all data is shared (in the *static extent*),
  - in parallel loops the iteration variable is private,
  - automatic variables in the *dynamic extent* are private,
  - static variables in the *dynamic extent* are shared.
  - formal/dummy arguments of functions/subroutines inherit data-sharing attributes of the actual arguments.

Data-sharing attributes example

Compare this situation . . .

```fortran
program main

  call main2()

end

subroutine main2()

  integer        :: i ! shared
  real(8)        :: x ! shared
  integer, save  :: j ! shared
  real(8), save  :: y ! shared

  !$omp parallel
  ...
  !$omp end parallel
end
```

```c
int main(int argc, char *argv[])
{
  main2();
}


void main2(void)
{
  int i;             // shared
  double x;          // shared
  static int j;      // shared
  static double y;   // shared

  #pragma omp parallel
  ...

}
```

Data-sharing attributes example

. . . with this one

```fortran
program main

  !$omp parallel
  call main2()
  !$omp end parallel
end

subroutine main2()

  integer          :: i ! private
  real(8)          :: x ! private
  integer, save :: j ! shared
  real(8), save :: y ! shared

  ...
end
```

```c
int main(int argc, char *argv[])
{
  #pragma omp parallel
  main2();
}

void main2(void)
{
  int i;              // private
  double x;           // private
  static int j;       // shared
  static double y; // shared

  ...
}
```

Data-sharing attributes example

- The second case `main2` resembles an MPI main program.
  (In MPI all variables are 'private'.)

- OpenMP needs shared variables to enable communication between threads.

- In MPI processes communicate by passing messages.

`threadprivate` directive and `copyin` clauses

- Sometimes each thread needs a *private copy* of *static* data:
  - Fortran example: common blocks
  - C example: static variables with file scope

- This can be achieved by employing the `threadprivate` directive.

- `threadprivate` variables can be initialised by using the `copyin` clause.

Data dependence analysis

- If a function/subroutine is called in the loop body, the data dependence analysis **must** include that function/subroutine and possibly the whole calling tree below that function/subroutine! → This can be very demanding.

- Such functions/subroutines are in the dynamic extent.

- According to slide 57 a data dependence can only occur if *static* variables are modified (because static variables are `shared`).

- Static data can be viewed as some internal state of a program. A function/subroutine that does not modify such data is said to have *no side effects*. Sometimes they are called *pure*.

→ It is a good idea to clearly separate functions with side effects from those without and to keep side effects to a minimum.

→ When using libraries one has to make sure that they are *thread-safe*.

Example – Random numbers

- Consider a simple random number generator

```
real(8) function random()
 implicit none
 integer, parameter :: mr = 714025
 integer, parameter :: ia = 1366
 integer, parameter :: ic = 150889
 real(8), parameter :: qdnorm=1.0_8/mr
 integer, save      :: irandom = 0

 irandom = mod(ia*irandom + ic, mr)
 random  = irandom * qdnorm
end
```

```
double random(void)
{
  const int mr = 714025;
  const int ia = 1366;
  const int ic = 150889;
  const double qdnorm = 1.0 / mr;
  static int irandom = 0;

  irandom = (ia*irandom + ic) % mr;
  return(irandom * qdnorm);
}
```

- Typically, random number generators have an internal state like `irandom`. This leads to a data dependence and prevents concurrent calls.

→ `./demos/lib/random_number.c`
  `./demos/lib/random_number.f90`

Parallelising loops (VI)
– removing dependences –

The `atomic` directive

- The `atomic` directive ensures that a specific storage location is updated atomically, i.e. by one thread at a time preventing multiple simultaneous writes (which would be a race condition).
- The `atomic` directive applies to the statement following it (one statement only).
- There can be several `atomic` directives in a loop.
- The statement must be of the form `x = x` *`op expr`* or equivalent (e.g. in C: `x++`, `x -= ...`).
- In Fortran intrinsic procedures are allowed too (e.g. `min` or `max`).
- The order of `atomic` operations is undefined/random.

Anti-dependences (I)

Anti-dependences can be removed by introducing auxiliary fields. This loop

```
do i = 1, N
   a(i) = a(i) + a(i + 1)
enddo
```

```
for (i = 1; i <= N; i++)
    a[i] += a[i + 1];
```

becomes

```
!$omp parallel do
do i = 1, N
   b(i) = a(i + 1)
enddo

!$omp parallel do
do i = 1, N
   a(i) = a(i) + b(i)
enddo
```

```
#pragma omp parallel for
for (i = 1; i <= N; i++)
    b[i] = a[i + 1];

#pragma omp parallel for
for (i = 1; i <= N; i++)
    a[i] += b[i];
```

Anti-dependences (II)

- By introducing `b` memory traffic is roughly doubled.

  $\rightsquigarrow$ No speed-up can be expected for 2 threads.

- This trick does not work for random numbers. Without a parallel random number generator random numbers must be generated sequentially.

Flow dependences

Flow dependences like recurrences are harder to parallelise

```
do i = 1, N
   a(i) = a(i) + a(i - 1)
enddo
```

```
for (i = 1; i <= N; i++)
    a[i] += a[i - 1];
```

Random numbers

This loop

```
do i = 1, N
   r = random()
   phi = random()
   x(i) = r * cos(phi)
   y(i) = r * sin(phi)
enddo
```

```
for (i = 0; i < N; i++) {
    r = random();
    phi = random();
    x[i] = r * cos(phi);
    y[i] = r * sin(phi);
}
```

becomes

```
do i = 1, N
   r(i) = random()
   phi(i) = random()
enddo

!$omp parallel do
do i = 1, N
   x(i) = r(i) * cos(phi(i))
   y(i) = r(i) * sin(phi(i))
enddo
```

```
for (i = 0; i < N; i++) {
    r[i] = random();
    phi[i] = random();
}

#pragma omp parallel for
for (i = 0; i < N; i++) {
    x[i] = r[i] * cos(phi[i]);
    y[i] = r[i] * sin(phi[i]);
}
```

Parallelising loops (VII)
– synchronisation –

Synchronisation

Recall the synchronisation constructs we have already seen:

- `barrier`
  - synchronisation point of all threads in the team
- `critical`
  - restricts execution of a block to a single thread at a time
- `atomic`
  - ensures that a storage location is updated by one thread at a time

In addition there are run-time library *lock* routines.

Locks – C example

```
#include <stdio.h>
#include <omp.h>
void skip(int i) {}
void work(int i) {}
int main()
{
  omp_lock_t lck;
  int id;
  omp_init_lock(&lck);
  #pragma omp parallel shared(lck) private(id)
  {
    id = omp_get_thread_num();
    omp_set_lock(&lck); /* BLOCKING */
    /* only one thread at a time can execute this printf */
    printf("My thread id is %d.\n", id);
    omp_unset_lock(&lck);
    while (! omp_test_lock(&lck)) { /* NON-BLOCKING */
      skip(id);  /* we do not yet have the lock, so we must do something else */
    }
    work(id);    /* we now have the lock and can do the work */
    omp_unset_lock(&lck);
   }
   omp_destroy_lock(&lck);
   return 0;
}
```

Locks – Fortran example

```
SUBROUTINE SKIP(ID)
END SUBROUTINE SKIP

SUBROUTINE WORK(ID)
END SUBROUTINE WORK

PROGRAM A39
  INCLUDE "omp_lib.h"  ! or USE OMP_LIB
  INTEGER(OMP_LOCK_KIND) LCK
  INTEGER ID
  CALL OMP_INIT_LOCK(LCK)
  !$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
    ID = OMP_GET_THREAD_NUM()
    CALL OMP_SET_LOCK(LCK) ! blocking
    PRINT *, 'My thread id is ', ID
    CALL OMP_UNSET_LOCK(LCK)
    DO WHILE (.NOT. OMP_TEST_LOCK(LCK)) ! non-blocking
       CALL SKIP(ID)    ! We do not yet have the lock so we must do something else
    END DO
    CALL WORK(ID)       ! We now have the lock and can do the work
    CALL OMP_UNSET_LOCK(LCK)
  !$OMP END PARALLEL
  CALL OMP_DESTROY_LOCK(LCK)
END PROGRAM A39
```

Locks – reference

- The examples were taken from the OpenMP specification
  OpenMP Application Program Interface (Version 2.5, May 2005)

  → `http://www.openmp.org/mp-documents/spec25.pdf`

Parallelising loops (VIII)
– scheduling loops –

Loop schedules

- A *loop schedule* is an assignment of loop iterations to threads.
- The number of consecutive iterations assigned to a thread is called *chunk*.
- There are two basic characteristics of loop scheduling:
  - *static:* The assignment of iterations to threads is given by the number of iterations and the number of threads alone. It is determined at the beginning of the loop.
  - *dynamic:* The assignment of iterations to threads is determined at run-time. The assignment happens chunk by chunk. It can vary from run to run (at identical parameters).
- Another characteristic is the chunk size.
- Loop scheduling can be used to balance the load.
- Static scheduling without specifying chunks has the lowest overhead.

Loop schedules

- The type of scheduling can be specified by the `schedule` clause:

```
omp parallel do/for schedule(static[,chunk])
omp parallel do/for schedule(dynamic[,chunk])
omp parallel do/for schedule(guided[,chunk])
omp parallel do/for schedule(runtime)
```

- The default type of scheduling is implementation dependent.

- If `schedule` is `runtime` the type of scheduling is taken from the value of the environment variable `OMP_SCHEDULE`, for example (*bash* syntax):

```
export OMP_SCHEDULE=dynamic,5
```

Functional parallelism

Functional parallelism

- In the previous section we had data-parallel loops in mind. Although loops do not necessarily have to be data-parallel (e.g. if they contain *if*-statements).

- Functional parallelism can be expressed directly by `sections` . . .

```
omp parallel
omp section
    this code will be executed by thread 0
omp end section
omp section
    this code will be executed by thread 1
omp end section
...
omp end parallel
```

Functional parallelism

- . . . or by using `omp_get_thread_num()`, for example

```
if (omp_get_thread_num() == 0)
    execute this code
else
    do something else
```

Reduction operations

Reduction operations (I)

- Employing the `reduction` clause

```
s = 0
!$omp parallel do reduction(+:s)
do i = 1, N
   s = s + a(i)
enddo
```

```
s = 0;
#pragma omp parallel for reduction(+:s)
for (i = 0; i < N; i++)
    s += a[i];
```

Reduction operations (II)

- Introducing `private` reduction variables and employing `critical`, e.g. if a reduction operation is not available (here one could also use `atomic`)

```
s = 0
!$omp parallel private(sp, i)

sp = 0
!$omp do
do i = 1, N
   sp = sp + a(i)
enddo

!$omp critical
s = s + sp
!$omp end critical
!$omp end parallel
```

```
s = 0;
#pragma omp parallel private(sp, i)
{
sp = 0;
#pragma omp for
for (i = 0; i < N; i++)
    sp += a[i];


#pragma omp critical
s += sp;

} // end parallel
```

Reduction operations (III)

- Avoiding race-conditions (rounding differences) introduced by `critical` and possibly by `reduction` (→ performance issue: *false sharing*)

```fortran
!$omp parallel shared(s, psum) &
!$omp          private(me, i)

me = omp_get_thread_num()
psum(me) = 0
!$omp do
do i = 1, N
   psum(me) = psum(me) + a(i)
enddo

!$omp master
s = 0
do i = 0,omp_get_num_threads()-1
   s = s + psum(i)
enddo
!$omp end master
!$omp end parallel
```

```c
#pragma omp parallel shared(s, psum) \
                     private(me, i)
{
me = omp_get_thread_num();
psum[me] = 0;
#pragma omp for
for (i = 0; i < N; i++)
    psum[me] += a[i];

#pragma omp master
{
s = 0;
for (i=0; i < omp_get_num_threads();i++)
    s += psum[i];
} // end master
} // end parallel
```

Reduction operations (IV)

- The Intel compiler allows reduction operations to be deterministic if the environment variable **`KMP_DETERMINISTIC_REDUCTION`** is set to **`1`**.

Performance

False sharing

- *False sharing* is a situation were threads work simultaneously on the same cache line and data on that cache line is being updated (see next slide).

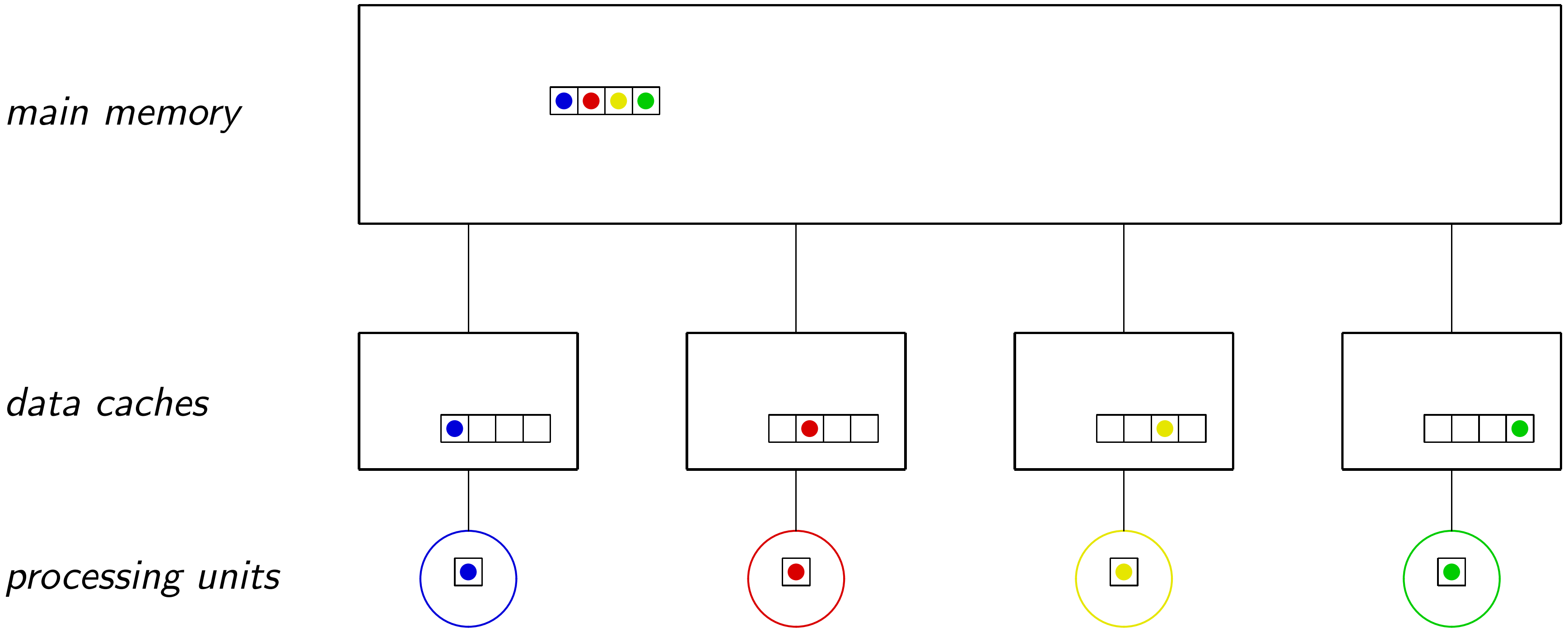
  Every time data from the cache line is being updated the cache line has to be copied to *all* other participating caches. This leads to poor performance.

- An example where false sharing is likely to occur is the implementation of the reduction operation on slide 84 (the elements of the shared array `psum` are used in a private context).

  The problem can be cured by introducing an explicit private reduction variable.

False sharing

- 4 processing units updating the same cache line



Avoiding false sharing

```fortran
!$omp parallel shared(lsum, gsum)
!$omp          private(s, i)

s = 0
!$omp do nowait
do i = 1, N
   s = s + a(i)
enddo
lsum(omp_get_thread_num()) = s
!$omp barrier

!$omp master
gsum = 0
do i = 0,omp_get_num_threads()-1
   gsum = gsum + lsum(i)
enddo
!$omp end master

!$omp end parallel
```

```c
#pragma omp parallel shared(lsum, gsum)\
                     private(s, i)
{
s = 0;
#pragma omp for nowait
for (i = 0; i < N; i++)
    s += a(i);

lsum[omp_get_thread_num()] = s;
#pragma omp barrier

#pragma omp master
{
gsum = 0;
for (i=0; i < omp_get_num_threads();i++)
    gsum += lsum[i];
} // end master

} // end parallel
```

General tips for achieving good performance

- Coverage: Amdahl's law holds

  $\rightarrow$ parallelise enough of the code

- Granularity: splitting up loops costs overhead

  $\rightarrow$ (very) small loops should not be parallelised

- Locality: consider data caches

  $\rightarrow$ chunks might be chosen to fit into data caches

Environment variables for thread placement and pinning

- generic

  `OMP_PROC_BIND=true` pins threads to CPUs
  `OMP_PLACES` placement and pinning (see OpenMP specs)

- GNU compiler

  `GOMP_CPU_AFFINITY` pins threads to specific CPUs
  example: `GOMP_CPU_AFFINITY="0 3 1-2 4-15:2"`

- Intel compiler

  understands `GOMP_CPU_AFFINITY`, too
  `KMP_AFFINITY` (more complicated to use)

- PGI compiler

  `MP_BIND=yes`
  plus `MP_BLIST=` comma separted lisf of CPU IDs

Low level OpenMP programming

Low level OpenMP programming

- Use less directives and clauses, e.g.
  - no `do`/`for` directives
  - no `shared`, `private`, or `reduction` clauses
- Motivation: more flexibility, e.g. explicit control over
  - work sharing
  - loop scheduling
  - reduction operations
- Explicit `barrier` synchronisation becomes important
- Low level replacements of other synchronisation mechanisms
  - `master` → `if (omp_get_thread_num() == 0)`
  - `critical` → locks
  - `atomic` → locks

Work sharing (I)

- High level

```
!$omp parallel do                    #pragma omp parallel for
do i = 1, N                          for (i = 0; i < N; i++)
   a(i) = b(i) + c(i)                    a[i] = b[i] + c[i];
enddo
```

- Low level

```
!$omp parallel private(i,i1,i2)      #pragma omp parallel private(i,i1,i2)
                                     {
call get_chunk(N, i1, i2)            get_chunk(N, &i1, &i2);

do i = i1, i2                        for (i = i1; i <= i2; i++)
   a(i) = b(i) + c(i)                    a[i] = b[i] + c[i];
enddo

!$omp end parallel                   } // end parallel
```

Work sharing (II)

- `get_chunk()` calculates `i1` and `i2` corresponding to `schedule(static)`

  → `./demos/openmp/get_chunk.c`
  `./demos/openmp/get_chunk.f90`

- In principle one can implement any kind of scheduling.

- Example: hybrid programming

  – overlapping communication and computation (functional parallelism)

  → compute threads have to manage work shares

Work sharing (III)

```fortran
subroutine get_chunk(n, i1, i2)
  use omp_lib
  integer n, i1, i2
  integer n_thread,me, chunk,rest

  n_thread = omp_get_num_threads()
  me = omp_get_thread_num()
  chunk = n / n_thread
  rest = mod(n, n_thread)

  if (me < rest) then
     chunk = chunk + 1
     i1 = chunk * me
  else
     i1 = chunk * me + rest
  endif
  i1 = i1 + 1
  i2 = i1 + chunk - 1
end
```

```c
# include <omp.h>
void get_chunk(int n, int *i1, int *i2)
{

  int n_thread = omp_get_num_threads();
  int me = omp_get_thread_num();
  int chunk = n / n_thread;
  int rest = n % n_thread;

  if (me < rest) {
    chunk++;
    *i1 = chunk * me;
  } else {
    *i1 = chunk * me + rest;
  }

  *i2 = *i1 + chunk - 1;
}
```

More on work sharing constructs

The `workshare` construct (Fortran)

- with the `workshare` construct one can parallelise
  - array expressions
  - `forall` and `where` loops
  - intrinsic functions like `matmul` and `dot_product`
- example

```
!$omp parallel workshare
a(:) = b(:) + c(:)
!$omp end parallel workshare

!$omp parallel
...
!$omp workshare
a(:) = b(:) + c(:)
!$omp end workshare
...
!$omp end parallel
```

Random access iterators (OpenMP 3.0 for C++)

- loops with random access iterators can be parallelised with the known loop construct
- example using the `vector` class from the C++ Standard Library

```
#include <vector>

void iterator_example()
{
    std::vector<int> vec(23);
    std::vector<int>::iterator it;

    #pragma omp parallel for default(none) shared(vec)
    for (it = vec.begin(); it < vec.end(); it++)
    {
        // do work with *it //
    }
}
```

source: *OpenMP Application Program Interface* (Version 3.1, July 2011)
→ `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`

The `collapse` clause

- the `collapse(`*n*`)` clause can be used to parallelise *n* perfectly nested loops
- *n* must be know at compile time

```
!$omp parallel do collapse(2) &
!$omp             private(i,j,k)
do i = i1, i2, istep
do j = j1, j2, jstep
do k = k1, k2, kstep
   x(k, j, i) = ...
enddo
enddo
enddo
```

```
#pragma omp parallel for collapse(2) \
                         private(i,j,k)
for (i = i1; i <= i2; i += istep)
for (j = j1; j <= j2; j += jstep)
for (k = k1; k <= k2; k += kstep)
      x[i][j][k] = ... ;
```

- note that `k` must be declared `private`

New concepts in OpenMP 3 and 4

New concepts in OpenMP 3 and 4

- OpenMP 3
  - tasking
- OpenMP 4
  - SIMD
  - devices / accelerators

Tasking

- *tasking* was introduced in OpenMP 3.0
- can handle task parallelism efficiently
- examples: traversing lists or search trees

→ slides and specifications at `www.openmp.org`

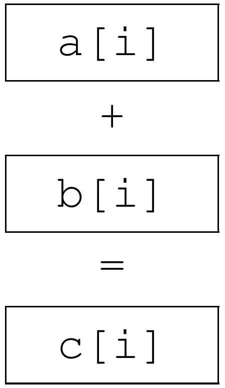
SIMD units

- processing of a loop

```
for i := 1 to 100 do
     c[i] := a[i] + b[i]
```

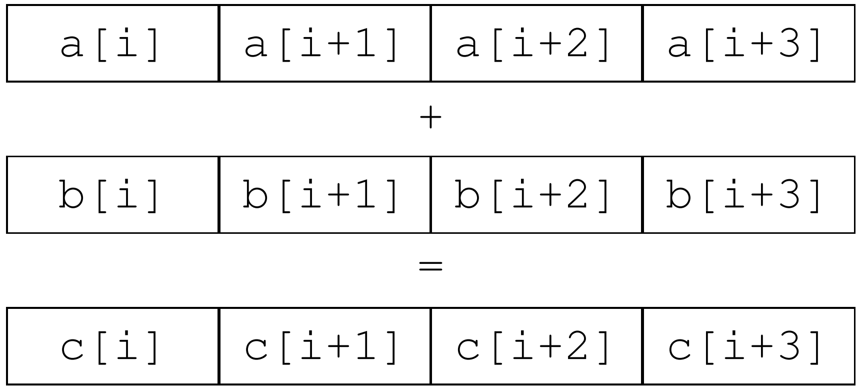
sequential processing

```
for i := 1 to 100 do
```



SIMD processing

```
for i := 1 to 100 step 4 do
```



SIMD constructs

- The **`simd`** directive indicates that a loop can be transformed into a SIMD loop (affects a single thread).

- The combined **`do simd`** / **`for simd`** directive splits a SIMD loop across the team of threads in a way that is consistent with the SIMD width.

  Example: for 2 threads (and SIMD width 4) this loop

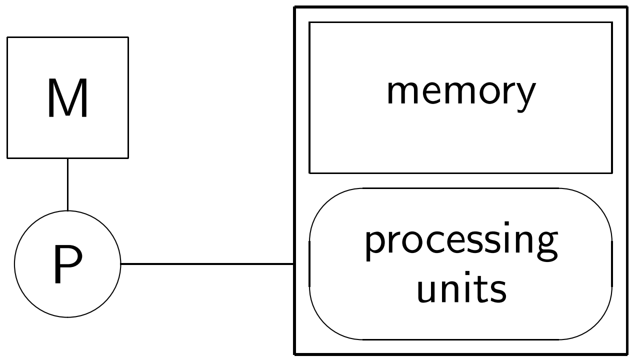
  ```
  for i := 1 to 20 do
      ...
  ```

  would be split into

  ```
  for i := 1 to 12 step 4 do            for i := 13 to 20 step 4 do
      ...                                   ...
  ```

Devices / accelerators

conventional computer with accelerator



OpenMP terminology:

host device *target device*

Device constructs

OpenMP 4 specifies *device constructs*.
These are similar to *OpenACC* directives. → `www.openacc.org`

There are constructs for:

- data management
  - allocating memory on the target device
  - copying data to and from the target device
- offloading code to the target device
- defining functions for the target device
- work-sharing on the target device

Device constructs – Fortran example

```
subroutine vec_mult(p, v1, v2, N)
  integer :: N, i;  real :: p(N), v1(N), v2(N)
  call init(v1, v2, N)
  !$omp target data map(to: v1, v2) map(from: p)
     !$omp target
     !$omp parallel do
     do i = 1, N
        p(i) = v1(i) * v2(i)
     end do
     !$omp end target
     call init_again(v1, v2, N)
     !$omp target update to(v1, v2)
     !$omp target
     !$omp parallel do
     do i = 1, N
        p(i) = p(i) + v1(i) * v2(i)
     end do
     !$omp end target
  !$omp end target data
  call output(p, N)
end
```

Device constructs – C example

```
void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);
    #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
    {
        #pragma omp target
        #pragma omp parallel for
        for (i = 0; i < N; i++)
            p[i] = v1[i] * v2[i];
        init_again(v1, v2, N);
        #pragma omp target update to(v1[:N], v2[:N])
        #pragma omp target
        #pragma omp parallel for
        for (i = 0; i < N; i++)
            p[i] = p[i] + (v1[i] * v2[i]);
    }
    output(p, N);
}
```

Device constructs – example reference

The example was taken from from *OpenMP Application Program Interface Examples*, Version 4.0.1, February 2014.

$\rightarrow$ `http://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf`

References

References

- R. Chandra, L. Dagum, D. Maydan, D. Kohr, J. McDonald, R. Menon, *Parallel Programming in OpenMP*
- OpenMP standards documents
  `http://openmp.org/wp/openmp-specifications`