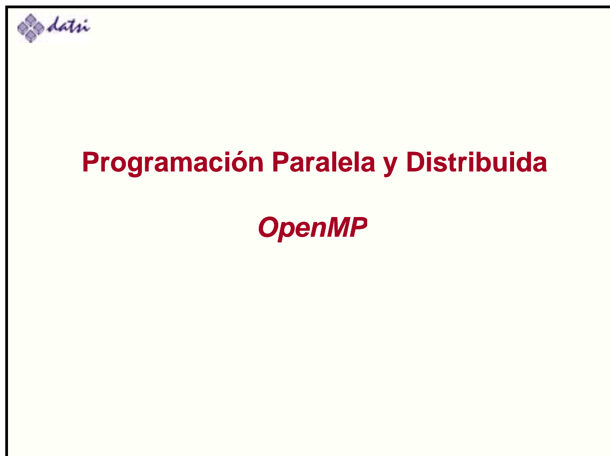
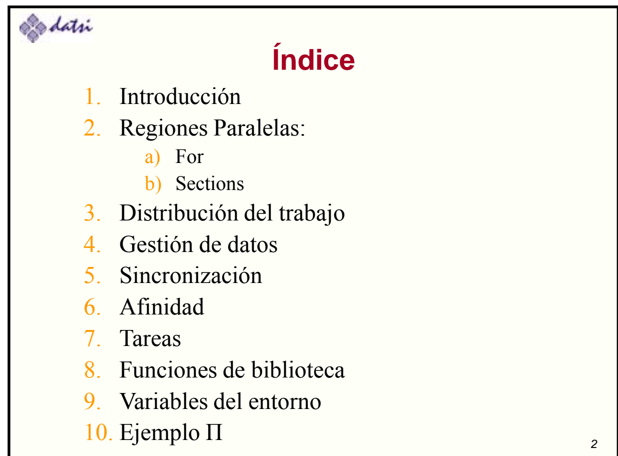
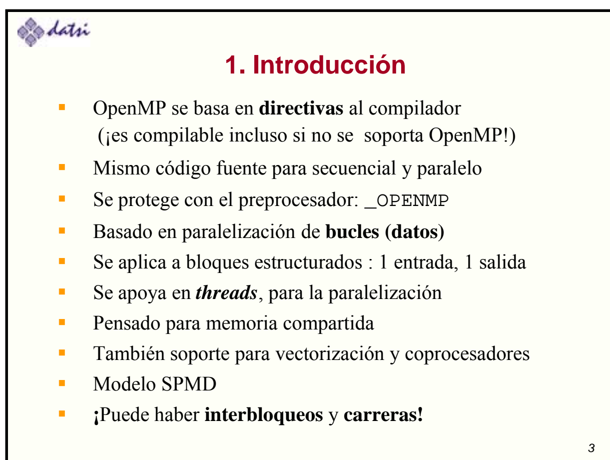
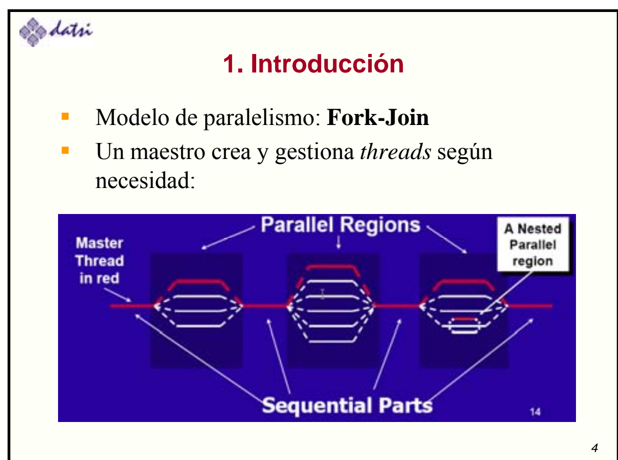
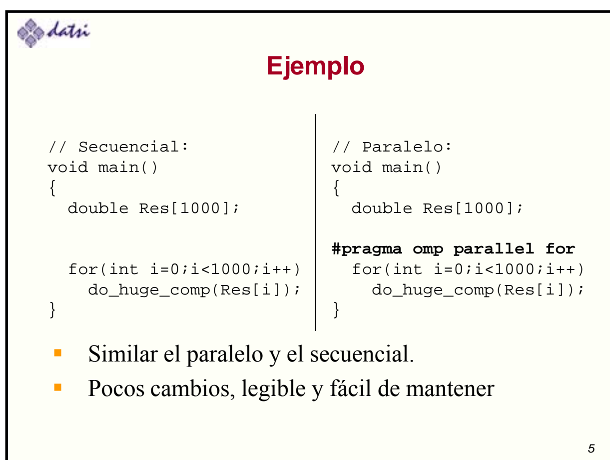
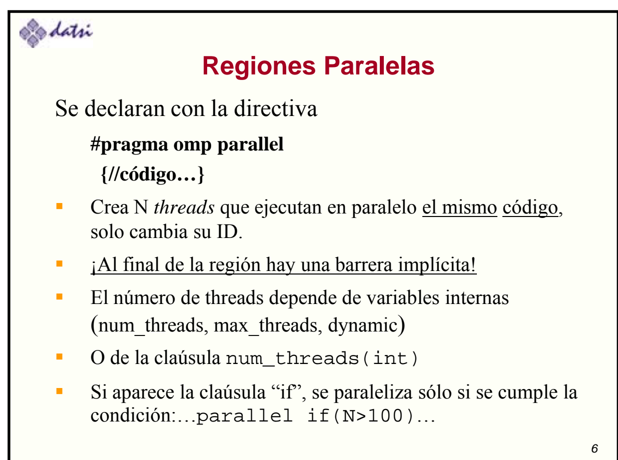

datsi
Programación Paralela y Distribuida
OpenMP


datsi
Índice
1. Introducción
2. Regiones Paralelas:
a) For
b) Sections
3. Distribución del trabajo
4. Gestión de datos
5. Sincronización
6. Afinidad
7. Tareas
8. Funciones de biblioteca
9. Variables del entorno
10. Ejemplo Π
2


datsi
1. Introducción
OpenMP se basa en directivas al compilador
(¡es compilable incluso si no se soporta OpenMP!)
Mismo código fuente para secuencial y paralelo
Se protege con el preprocesador: _OPENMP
Basado en paralelización de bucles (datos)
Se aplica a bloques estructurados : 1 entrada, 1 salida
Se apoya en threads, para la paralelización
Pensado para memoria compartida
También soporte para vectorización y coprocesadores
Modelo SPMD
¡Puede haber interbloqueos y carreras!
3


datsi
1. Introducción
Modelo de paralelismo: Fork-Join
Un maestro crea y gestiona threads según necesidad:
Parallel Regions
A Nested Parallel region
Master Thread in red
Sequential Parts
14
4


datsi
Ejemplo
// Secuencial:
void main()
{
double Res[1000];
for(int i=0;i<1000;i++)
do_huge_comp(Res[i]);
}
// Paralelo:
void main()
{
double Res[1000];
#pragma omp parallel for
for(int i=0;i<1000;i++)
do_huge_comp(Res[i]);
}
Similar el paralelo y el secuencial.
Pocos cambios, legible y fácil de mantener
5


datsi
Regiones Paralelas
Se declaran con la directiva
#pragma omp parallel
{//código…}
Crea N threads que ejecutan en paralelo el mismo código, solo cambia su ID.
¡Al final de la región hay una barrera implícita!
El número de threads depende de variables internas (num_threads, max_threads, dynamic)
O de la claúsula num_threads(int)
Si aparece la claúsula "if", se paraleliza sólo si se cumple la condición:…parallel if(N>100)…
6

Regiones Paralelas: Ejemplo

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  sumarElemA (ID,A);
}
```

7

Distribución del trabajo: *for*

Se declaran con la directiva

#pragma omp for
for (;;) {//código…}

- Divide las iteraciones de un bucle entre los *threads* disponibles.
- Al final hay una barrera implicita (evitable: *nowait*)
- Se puede especificar la planificación
- Se pueden "agrupar" varios bucles en uno, juntando sus iteraciones: `#pragma omp for collapse (3)`
- Resulta un bucle con TODAS las iteraciones

8

Distribución del trabajo: Ejemplo *for*

1. Código secuencial

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

2. Región Paralela OpenMP (id, i, Nthrds, istart, iend: son variables privadas, ¿por qué? ¿deben serlo?)

```
#pragma omp parallel
{
  int id, i,Nthrds,istart,iend;
  id = omp_get_thread_num();
  Nthrds =omp_get_num_threads();
  istart = id * N / Nthrds;
  iend = (id+1) * N / Nthrds;
  for(i=istart;i<iend;i++)
    { a[i] = a[i] + b[i];}
}
```

3. Región Paralela OpenMP y distribución de trabajo *for*

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

9

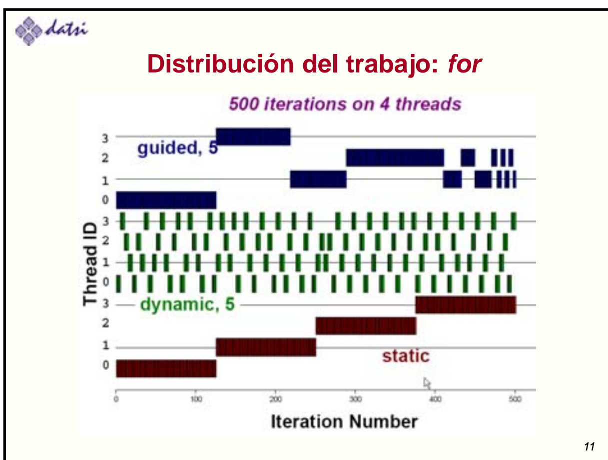
Distribución del trabajo: *for*

Planificación con la claúsula *schedule*:
`#pragma omp for schedule(..)`

- schedule (**static** [,chunk])
  - Asigna bloques fijos de iteraciones de tamaño "chunk" a cada *thread*. Es determinista, muy útil para depuración.
- schedule (**dynamic** [,chunk])
  - Cada *thread* coge "chunk" iteraciones hasta terminar con todas
- schedule (**guided** [,chunk])
  - Cada *thread* coge dinámicamente bloques de iteraciones
  - El bloque inicialmente es grande (iteraciones sin asignar dividido el número de threads) y va bajando hasta tamaño"chunk"
- schedule (**runtime**)
  - Planificación y tamaño de bloque determinado por la variable de entorno OMP_SCHEDULE

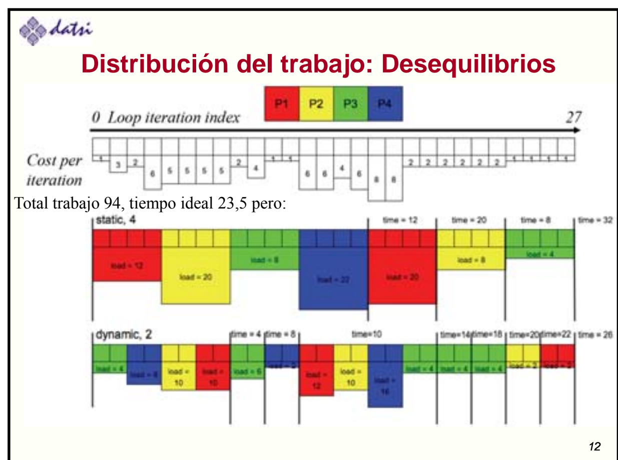
10

Distribución del trabajo: *for*



11

Distribución del trabajo: Desequilibrios



12

Distribución del trabajo: *sections*

- Asigna "secciones" a *threads*. Ejemplo

```
#pragma omp parallel
#pragma omp sections
{
 x_calculation();
#pragma omp section
 y_calculation();
#pragma omp section
z_calculation();
}
```

- Al final hay una barrera (opción *nowait*)

13

Gestión de los datos

- Todos las variables globales son **compartidas** (memoria compartida)
- Pero las variables ubicadas en cada pila propia (parámetros y variables locales) son **privadas** (cada *thread* tiene su pila)
- Se puede cambiar los 'atributos' de las variables heredadas del maestro mediante cláusulas
- OpenMP usa un modelo relajado de la consistencia (las actualizaciones no siempre son visibles a otras CPUs). Se puede forzar con ***flush***
  - Automático en `parallel, critical, for, sections, barriers`

14

Gestión de los datos

- **shared**: Variable común a todos los *threads*
  - Cuidado: puede necesitar regiones críticas
- **private**: Cada *thread* tiene "su" copia local.
  - Sin inicializar e indefinido tras la región paralela
- **firstprivate**: Es *private*, pero se inicializa con el valor de la 'original' (variable del maestro)
- **lastprivate**: Es *private*, pero al final se le asigna el valor que toma en la última iteración o sección
- **default**: `shared, private` (FORTRAN) o **`none`**

15

Gestión de los datos

- **threadprivate**: hace una copia "privada" de un dato asociada a cada *thread*. Existe a lo largo de la ejecución, en secuencial se ve la copia del maestro
- **copyin**: inicializa los *threadprivate* con el maestro
- **copyprivate**: pasa el valor dado en un hilo al resto (se usa con *single*)
- **reduction** (op:list): deben ser variables *shared*
  - Hará una copia local y la inicializará según "op"
  - Al final las copias locales se reducen a una global
  - Operadores: + * - & | ^ && || min max
  - Definido por el usuario (*declare reduction*)

16

Gestión de los datos

1.
```
#pragma omp parallel private(i) firstprivate(j)
{   i = 3 + func(id);
    j = j + 2; }
```
2.
```
#pragma omp parallel for lastprivate(i)
     for (i=0; i<n-1; i++)
       a[i] = b[i] + b[i+1];
     a[i]=b[i]; // caso n-1 fuera bucle
```
3.
```
#pragma omp parallel for shared(x, y, n) reduction(+:a)
        reduction(^:b) reduction(min:c) reduction(max:d)
     for (i=0; i<n; i++) {
       a += x[i];
       b ^= y[i];
       if (c > y[i]) c = y[i];
       if (d < x[i]) d = x[i];
     }
```
4. Tambien para controlar convergencia: **reduce(||:notdone) o reduce(&&:done)**

17

Gestión de los datos

1.
```
int counter = 0; // Nº tareas ejecutadas por cada hilo…
#pragma omp threadprivate(counter)
```
2.
```
#pragma omp threadprivate(work,size)
#pragma omp parallel copyin(size) shared(N)
{
   work = build(tol,size+N);
}
```
3.
```
#pragma omp parallel for reduction(+:res) lastprivate(Z,i)
 for (i=0; i< 1000; i++){
   Z = func(I);
   res = res + Z;
 }
```

18

Sincronización

- **critical [name]**: Define una región crítica. Los *threads* esperan al comienzo de la región crítica hasta que no haya ningún hilo ejecutando una región crítica con el **mismo nombre.** Todas las que no tienen nombre, se consideran que tienen un mismo nombre no especificado
- **atomic [read|write|update|capture]**: Asegura una actualización atómica. Por defecto **update**
  - Lectura atómica: v = x;
  - Escritura atómica: x = expr;   //expr no atómico
  - Actualización atómica: x = x binop expr;
  - “Captura” atómica: v = x binop= expr;   // x atómico

19

Sincronización

- **barrier**: Implementa una barrera
- **ordered**: Asegura que las iteraciones se ejecutan en el mismo orden que en secuencial: ¡¡evítese en lo posible!!
- **master**: Sólo el maestro ejecuta dicho código, los demás se lo saltan. Sin barrera.
- **single**: Un solo *thread* ejecuta dicho código, tiene una barrera implícita. Con **copyprivate** puede hacer visible a los demás *threads* los cálculos hechos.

20

Sincronización

```
1.  #pragma omp parallel for schedule(dynamic) private(a)
      for (i=0; i<N; i++){
        a = work(i); // en paralelo
    #pragma omp ordered      // espera que le toque
         printf("%d\n", a); // impresión resultados
                            // ordenada
    }
2.  #pragma omp threadprivate(x, y)
    void init(float a, float b ) { // a y b privados
    #pragma omp single copyprivate(a,b,x,y)
            scanf("%f %f %f %f", &a, &b, &x, &y);
    }
3.  #pragma omp critical qlock
       enqueue(job);
    …
    #pragma omp critical qlock
       dequeue(job);
```

21

Sincronización

```
1.  omp_lock_t *new_lock()
    {
      omp_lock_t *lock_ptr;
      #pragma omp single copyprivate(lock_ptr)
      {
        lock_ptr = (omp_lock_t *)malloc(sizeof(omp_lock_t));
        omp_init_lock( lock_ptr );
      } //barrera, todos salen con el cerrojo!!!
      return lock_ptr;
    }
```

22

Afinidad de los *threads*

- La idea es evitar trasiego para reutilizar los datos de cache
- Si no se indica afinidad los hilos pueden moverse, con afinidad no pueden moverse
- OpenMP nos permite decir dónde van los hilos
- Con **OMP_PLACES** se indica las ubicaciones posibles
- Con **OMP_PROC_BIND** cómo queremos repartir los *threads* en esas ubicaciones

23

Afinidad de los *threads:* OMP_PLACES

- **OMP_PLACES** se puede indicar con una lista
- La lista indica las ubicaciones y para cada una comienzo, longitud e incremento (opcional)

```
OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"
OMP_PLACES="{0:4},{4:4},{8:4},{12:4}" //Equivalentes

OMP_PLACES="{0,2,4,8},{1,3,5,7}"
OMP_PLACES="{0:4:2},{1:4:2}" // Equivalentes
```

- También de modo abstracto con nombres y cuántos hay, organizando las ubicaciones por cores, hilos o *sockets*

```
OMP_PLACES="cores(8)"
OMP_PLACES="threads(4)"
OMP_PLACES="sockets(2)"
```

24

Afinidad de los *threads:* OMP_PROC_BIND

- **OMP_PROC_BIND** puede ser `false` o `true`. *`master`*, *`close`* o *`spread`* (*`true`* por defecto)
- **False** desactiva afinidad (hilos se mueven)y **true** la activa
- **Master** indica ubicar los hilos en la misma partición que el maestro
- **Close** indica ubicarlos cerca, en particiones contiguas
- **Spread** indica ubicarlos lejos, distribuidos por las particiones
- Si se indican varios valores, cada uno es para un nivel `OMP_PROC_BIND="spread, spread, close"`
- Hay una claúsula **proc_bind** en **parallel**, válida si no está la afinidad desactivada

25

OMP_PROC_BIND: master

- **OMP_PLACES**="cores(8)" Reutilizar datos cache

master 2 ●master ●worker ■partition

p0 p1 p2 p3 p4 p5 p6 p7

master 4

p0 p1 p2 p3 p4 p5 p6 p7

master 8

p0 p1 p2 p3 p4 p5 p6 p7

26

OMP_PROC_BIND: close

- **OMP_PLACES**="cores(8)" Buen equilibrio y proximidad

close 2 ●master ●worker ■partition

p0 p1 p2 p3 p4 p5 p6 p7

close 4

p0 p1 p2 p3 p4 p5 p6 p7

close 16

p0 p1 p2 p3 p4 p5 p6 p7

27

OMP_PROC_BIND: spread

- **OMP_PLACES**="cores(8)" No compartir cache, buen equilibrio. Crea subparticiones. Nuevos *Threads* anidados en su nueva partición

spread 2 ●master ●worker ■partition

p0 p1 p2 p3 p4 p5 p6 p7

spread 8

p0 p1 p2 p3 p4 p5 p6 p7

spread 16

p0 p1 p2 p3 p4 p5 p6 p7

28

Tareas

- A partir de OpenMP 3.0
- Permite definir una "task": conjunto de código y datos a ajecutar
- Si hay algún *thread* disponible se pone a ejecutarla, si no espera uno libre.
- Permite una flexibilidad imposible antes
- Se espera por ellas en las barreras implícitas y explícitas
- Se puede esperar por ellas con `taskwait`

29

Tareas: Ejemplo

- ¿Cómo se pueden paralelizar este código?

```
while(my_pointer) {
    do_independent_work (my_pointer);
    my_pointer = my_pointer->next ;
} // End of while loop
```

- En general cualquier código con un número indefinido de iteracciones. Solución bestia: contar los elementos de la lista y montar un bucle: feo
- Solución: Tareas. Puede ser condicional (clausula if)

30

Tareas: Ejemplo

```
my_pointer = listhead;
#pragma omp parallel
{
  #pragma omp single nowait
  {
    while(my_pointer) {
      #pragma omp task firstprivate(my_pointer)
      {
        (void) do_independent_work (my_pointer);
      }
      my_pointer = my_pointer->next ;
    }
  }// End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```

31

Tareas: Dependencias

- Se pueden indicar dependencias entre tareas con la cláusula `depend`.
- Se debe indicar el tipo de dependencia:`in, out, inout`
- Hasta que no se cumplen todas sus dependencias no puede ejecutar

```
depend(tipo:lista_tareas)
```

32

Tareas: Tipos

- Tareas no diferidas: Con clausula `if(false)`.
  - La tarea generadora es suspendida hasta que la generada es ejecutada, que no tiene porque ejecutar inmediatamente
- Tareas incluidas: Se ejecutan inmediatamente por la tarea generadora (descendientes de una tarea *final*)
- Tarea final: Con clausula `final(true)`.
  - Todos sus descendientes son tareas finales e incluidas
- Tareas fusionadas: Clausula `mergeable`
  - Mismo entorno de datos (en los casos anteriores operan las clausulas de datos!!). Es como si no hubiese nueva tarea .
  - Sólo aplicable a incluidas o no diferidas

33

Funciones de biblioteca: Entorno de Ejecución

- **Gestión de threads**
  - `void omp_set_num_threads(int)`
  - `int omp_get_num_threads(void)`
  - `int omp_get_thread_num(void)`
  - `int omp_get_max_threads(void)`
  - `void omp_set_dynamic(bool)`
  - `bool omp_get_dynamic(void)`
- **Anidamiento del paralelismo**
  - `void omp_set_nested(bool)`
  - `bool omp_get_nested(void)`

34

Funciones de biblioteca: Entorno de Ejecución y Cerrojos

- Cerrojos
  - `void omp_init_lock(lock)`
  - `void omp_destroy_lock(lock)`
  - `void omp_set_lock(lock)`
  - `void omp_unset_lock(lock)`
  - `void omp_test_lock(lock)`
- ¿En una región paralela?
  `bool omp_in_parallel(void)`
- Número procesadores:
  `int omp_num_procs(void)`
- Tomar tiempos
  `omp_get_wtime(), omp_get_wtick()`

35

Variables del Entorno de Ejecución

- `OMP_SCHEDULE` *"schedule[, chunk_size]"*
- `OMP_NUM_THREADS` *int_literal //Máximo*
- `OMP_DYNAMIC` *bool // Ajusta el número de // hilos en cada región // paralela*
- `OMP_NESTED` *bool*
- `OMP_PLACES`
- `OMP_PROC_BIND`
- `OMP_STACKSIZE int [B|K|M|G}`
- `OMP_WAIT_POLICY` *passive || active*
- `OMP_THREAD_LIMIT` *int*
- `OMP_DISPLAY_ENV TRUE|FALSE|VERBOSE`

36

OpenMP 4.0

- Está disponible OpenMP 4.0 (http://www.openmp.org)
- Dos cambios importantes: **vectorización y coprocesadores**
- **Vectorización**: nuevo `#pragma omp simd` para vectorizar bucles. Toma la idea de Intel (icc). Con claúsulas
  - Busca aprovechar las unidades vectoriales: MMX, SSE, AVX, MIC). Se podrá combinar con regiones paralelas:

    `#pragma omp parallel for simd`
- **Coprocesadores**: Se busca aprovechar las GPUs y MICs, de gran potencia de cálculo.
  - Se introduce `#pragma omp target` para indicar el dispositivo sobre el que se quiere ejecutar
  - Toma la idea de PG, que ya dispone del OpenACC, un OpenMP de pago para GPUs

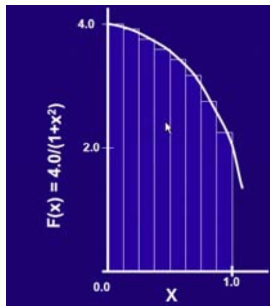
37

Compilación

- Incluir el fichero de cabecera: `#include <omp.h>`
- Se activa `_OPENMP`
- Compilador de GCC:
  ```
  gcc -fopenmp
  gfortran -fopenmp
  ```
- Compilador de Intel:
  ```
  icc -openmp
  ifort -openmp
  ```
- Compilador de Sun:
  ```
  suncc -xopenmp
  f95 -xopenmp
  ```

38

Ejemplo del cálculo de Pi



- Por cálculo númerico:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

- Se aproxima como:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

- con $F(x_i)$ el alto a mitad del intervalo y x el ancho del intervalo

39

Ejemplo del cálculo de Pi : Secuencial

```
static long num_steps = 100000;
double step;
void main (){
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

40