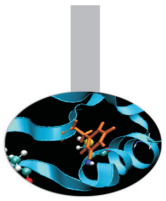
CINECA SCAI
SuperComputing Applications and Innovation

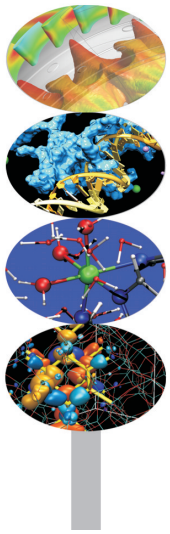


Introduzione alla parallelizzazione ibrida con MPI+OpenMP

Modulo 1

Claudia Truini Luca Ferraro Vittorio Ruggiero

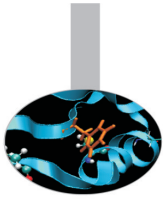
CINECA Roma - SCAI Department



CINECA

CINECA SCAI
SuperComputing Applications and Innovation

Outline

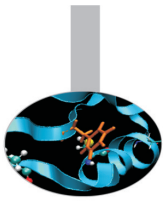


Analisi delle performance e parallelizzazione ibrida

- Analisi della scalabilità
- Parallelizzazione ibrida MPI-OpenMP
- Parallelizzazione e scaling per Laplace 2D

CINECA

Scalabilità (*scaling*)



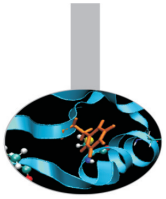
- Capacità di un sistema di elaborazione - hardware, software (OS, librerie, applicazione, ...) - di aumentare la quantità di lavoro eseguito per unità di tempo
- Ingredienti fondamentali
  - hardware
  - algoritmo
  - implementazione
- Misura dello scaling
  - *speed-up*: rapporta il tempo seriale $T(1)$ col tempo usando n processi $T(n)$

$$S(n) = T(1)/T(n)$$

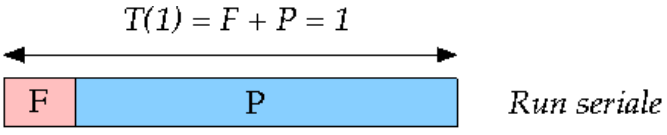
  - efficienza: rapporta lo speed-up col valore lineare (scaling ottimo atteso)

$$E(n) = S(n)/n$$

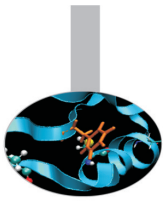
Fixed-size model e legge di Amdahl



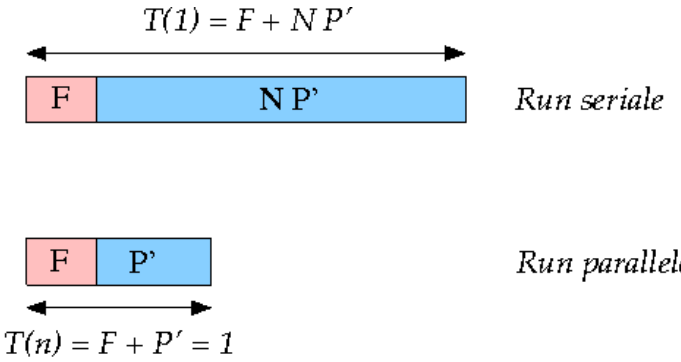
- Tempo di esecuzione del codice seriale: $T(1) = F + P$
  - F tempo della parte seriale
  - P tempo della parte parallela
- Tempo di esecuzione del codice con n processi: $T(n) = F + P/n$
- Speed-up
  - $S(n) = T(1)/T(n) = (F + P)/(F + P/n)$
  - fissando a 1 il tempo seriale: $S(n) = 1/(F + P/n)$
  - per n grandi esiste un limite asintotico $S = 1/F$



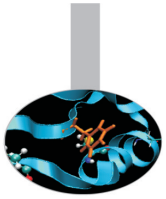
Scaled-size model e legge di Gustafson



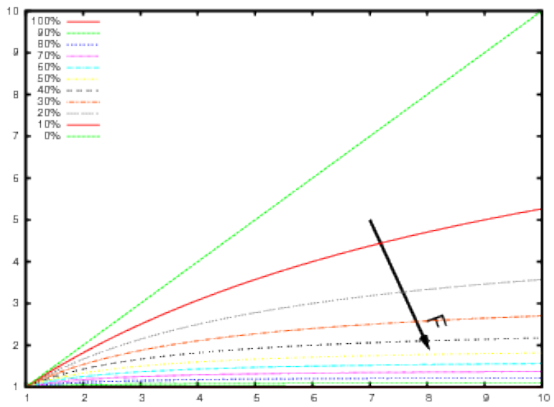
- Tempo di esecuzione del codice con n processi: $T(n) = F + P'$
  - F tempo della parte seriale
  - P' tempo della parte parallela
- Tempo di esecuzione del codice seriale: $T(1) = F + nP'$
- Speed-up
  - $S(n) = T(1)/T(n) = (F + nP')/(F + P')$
  - fissando il tempo parallelo pari a 1 $S(n) = F + nP' = F + n(1 - F)$
  - per n grandi non esiste limite asintotico

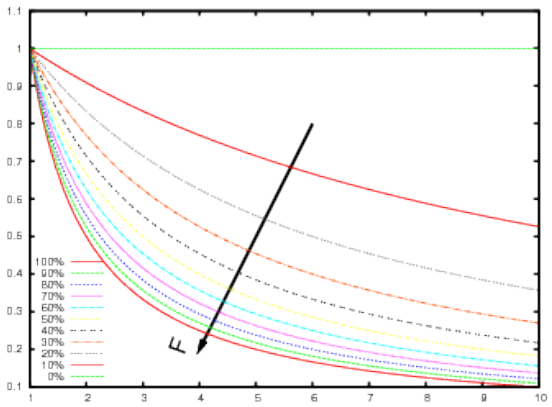


Fixed-size model: speed-up ed efficienza

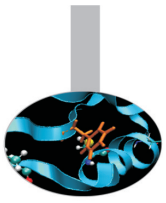


- F $\rightarrow$ percentuale tempo parte seriale su totale tempo seriale
- $S(n) = 1/(F + (1 - F)/n)$
- $E(n) = 1/(nF + 1 - F)$

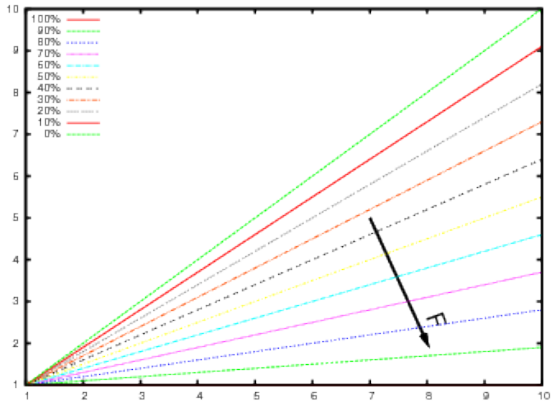


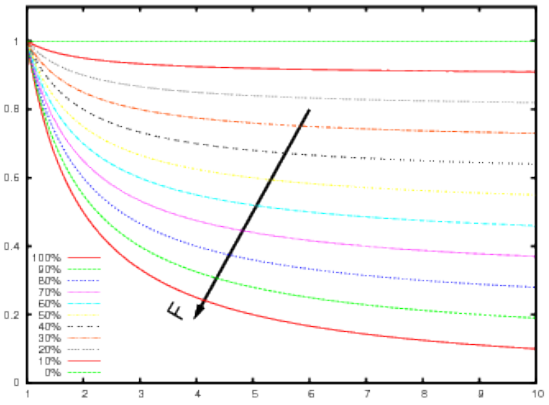


Scaled-size model: speed-up ed efficienza



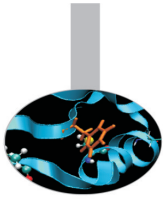
- F $\rightarrow$ percentuale tempo parte seriale su tempo a n processi
- $S(n) = F + n(1 - F)$
- $E(n) = F/n + (1 - F)$





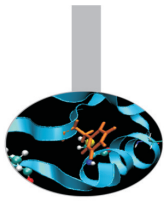
CINECA SCAI SuperComputing Applications and Innovation

Modelli di *scaling* avanzati



- Limitazioni modelli di *scaling* semplificati
  - il tempo della parte parallela si assume scali linearmente col numero di processori
  - si trascurano tempi di comunicazione, gestione, sincronizzazione e coordinamento
  - algoritmo fissato
- La realtà è più complessa:
  - i migliori algoritmi seriali sono di solito poco parallelizzabili
  - algoritmi che scalano bene sono di solito inefficienti in seriale
  - al variare del numero di processori possono convenire parallelizzazioni diverse
- Per una valutazione ‘onesta’:
  - miglior algoritmo seriale *vs* miglior algoritmo parallelo
- Per il proprio codice è bene predisporre un modello *ad hoc* di performance attese

Strong vs weak scaling



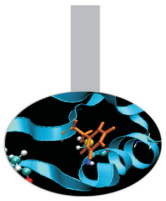
- Fixed-size e scaled-size sottolineano i due aspetti fondamentali della scalabilità in contesto HPC
  - aumento del numero di processori
  - aumento della taglia del problema
- *Strong scaling* (problem constraint): scalabilità tenendo fissa la taglia totale del problema
  - aumentando il numero di processori voglio risolvere più rapidamente il problema
- *Weak scaling* (time constraint): scalabilità tenendo fissa la taglia del problema per processore
  - aumentando il numero di processori voglio risolvere problemi più grandi

CINECA SCAI
SuperComputing Applications and Innovation

Motivi per combinare MPI e OpenMP

- Rispetta il trend delle architetture attuali formate da cluster di macchine multicore a memoria condivisa
  - usando OpenMP intra-nodo e MPI extra-nodo
- Alcune applicazioni mostrano due naturali livelli di parallelismo (e.g., multiscala, multiblocco, parametrici)
  - più semplice da gestire rispetto ad usare più comunicatori MPI
- In alcuni casi, il numero di processi MPI può essere limitato dalla struttura dell'applicazione
- È possibile compensare alcuni problemi di bilanciamento tra i processi MPI aggiungendo un livello OpenMP
- Un codice ibrido si presta più facilmente all'applicazione di moderni acceleratori (e.g. GPU), gestibili con direttive simili ad OpenMP (incluse in futuro in OpenMP stesso)
- Può permettere un miglioramento di scalabilità

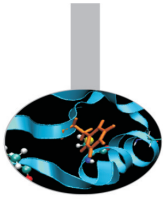
Implementazione ibrida MPI-OpenMP



- MPI-1 non definisce un modello di esecuzione per ogni processo che può quindi anche essere *multi-threaded*
- MPI-2 fornisce una funzione, al posto di **`MPI_Init`**, per inizializzare MPI in un ambiente *multi-threaded*:
  - **`MPI_Init_thread(int *argc, char *((*argv[]), int required, int provided)`**
  - **`MPI_INIT_THREAD(required, provided, ierr)`**
- **`provided`** e **`required`** specificano rispettivamente il livello di supporto *thread* fornito e richiesto i cui valori in ordine crescente sono:
  - **`MPI_THREAD_SINGLE`**: è equivalente a chiamare **`MPI_INIT`**
  - **`MPI_THREAD_FUNNELED`**: il processo può essere *multi-threaded* ma l'applicazione deve assicurare che solo il *main thread* (cioè quello che ha chiamato **`MPI_INIT_THREAD`**) effettui chiamate MPI
  - **`MPI_THREAD_SERIALIZED`**: il processo può essere *multi-threaded* ma l'applicazione deve assicurare che i diversi *thread* non effettuino chiamate MPI simultaneamente
  - **`MPI_THREAD_MULTIPLE`**: *thread* diversi possono eseguire chiamate MPI anche simultaneamente

CINECA SCAI SuperComputing Applications and Innovation

MPI_Init_thread



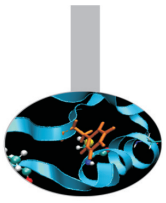
Fortran

```
required = MPI_THREAD_FUNNELED ;
call MPI_Init_thread(required, provided, ierr)
if(required /= MPI_THREAD_SINGLE) then
  if(provided == MPI_THREAD_SINGLE) then
     if(rank == 0) then
       print*,'Errore: Supporto thread NON superiore a MPI_THREAD_SINGLE'
       print*,'THREAD support required,provided:  ',required,provided
     endif  ;  call MPI_FINALIZE(ierr)  ;   STOP
  elseif(provided < required) then
     if(rank == 0) then
       print*,'Warning Supporto thread inferiore al richiesto'
       print*,'THREAD support required,provided:  ',required,provided
     endif  ;  endif  ;  endif
```

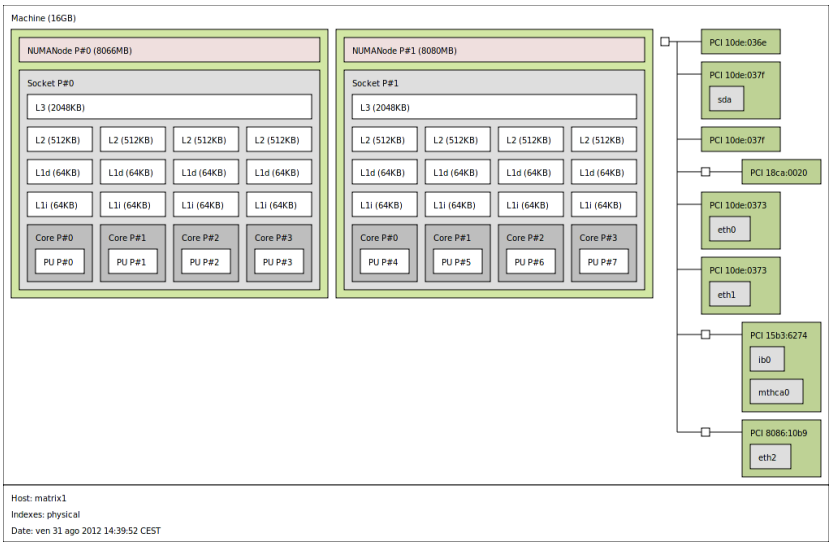
C

```
required = MPI_THREAD_FUNNELED;
ierr = MPI_Init_thread(&argc, &argv, required, provided);
if(required != MPI_THREADS_SINGLE) {
  if(provided == MPI_THREAD_SINGLE) {
    if(rank == 0) {
      fprintf(stderr,"Errore! Supporto thread NON superiore a MPI_THREAD_SINGLE'");
      fprintf(stderr,"required,provided: %d,%d ",required,provided); }
    ierr = MPI_FINALIZE()  ;   exit(-1); }
  else if(provided < required) {
    if(rank == 0) {
      fprintf(stderr,"Warning! Supporto thread inferiore al richiesto") ;
      fprintf(stderr,"required,provided: %d,%d ",required,provided") ; } ; }
```

Mappa e Binding di Processi con MPI

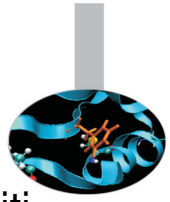


- Architetture Non-Uniform Memory Access (NUMA): architettura di memoria sviluppata per sistemi multiprocessore dove i tempi di accesso dipendono dalla posizione della memoria rispetto al processore



- Per sfruttare meglio la gerarchia di memoria, molte implementazioni di MPI consentono di controllare la distribuzione dei processi

Mappa e Binding di Processi con MPI - OpenMPI



- Assegnamento per nodi: ad esempio, processi MPI distribuiti uno per nodo

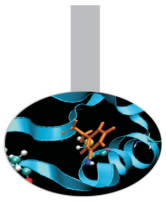
```
mpirun -npernode 1 <exe>
```

- Assegnamento per CPU: ad esempio, processi MPI distribuiti uno per socket

```
mpirun -report-bindings -num-sockets 2 -npersocket 1 <exe>
rank 0 bound to socket 0[core 0-3]:   [B B B B][....]
rank 1 bound to socket 1[core 0-3]:   [....][B B B B]
```

- ...la mappatura dei processi limita la loro migrazione tra i core del nodo
- È anche possibile evitare del tutto la migrazione tra i core con l'opzione **`-bind-to-core`**
  - questa operazione si chiama binding tra processi e processori/core

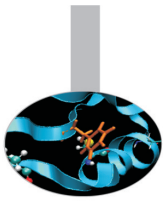
Mappa e Binding di thread con OpenMP



- OpenMP non specifica (ancora) come vengono distribuiti i threads sulla macchina né fornisce funzionalità per il loro controllo
- Alcuni vendor di OS e compilatori forniscono strumenti per controllare l'affinity dei threads tramite variabili di ambiente
- **`GOMP_CPU_AFFINITY`** compilatore GNU (e INTEL v12)
  - GOMP_CPU_AFFINITY="0 3 1-2 4-15:2" mappa il primo thread alla CPU 0, il secondo alla CPU 3, il terzo e quarto alle CPU 1 e 2, i thread dal quinto al decimo alle CPU 4, 6, 8, 10, 12, 14
- **`KMP_AFFINITY`** compilatore Intel (processore Intel)
  - **`compact`**: thread contigui a core/processori contigui
  - **`scatter`**: thread contigui a core/processori distanziati
- Il controllo e il binding dei threads consente di limitare gli effetti della gerarchia di memoria e di sfruttare al meglio le risorse e le caratteristiche della macchina

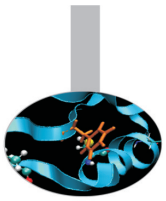
CINECA SCAI SuperComputing Applications and Innovation

Laplace 2D: parallelizzazione



- Algoritmo semplice per mostrare le strategie di parallelizzazione OpenMP, MPI e ibrida MPI+OpenMP
  - OpenMP: sicuramente conveniente almeno per la semplicità di implementazione
  - MPI: necessaria per sfruttare architetture multinodo
  - MPI+OpenMP: per l'algoritmo testato non mostra vantaggi particolari $\rightarrow$ esempio solo a scopo didattico!
- Decomposizione MPI a blocchi 2D: conveniente soprattutto per problemi grandi perché limita l'impatto delle comunicazioni
  - decomponendo a blocchi 1D ogni processo invia e riceve $2N$ dati
  - decomponendo a blocchi 2D ogni processo invia e riceve $4N/\sqrt{N_{PROC}}$ dati, se il numero di blocchi nelle due direzioni è uguale

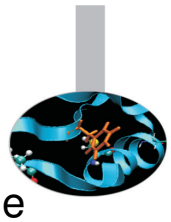
Laplace 2D: parallelizzazione MPI



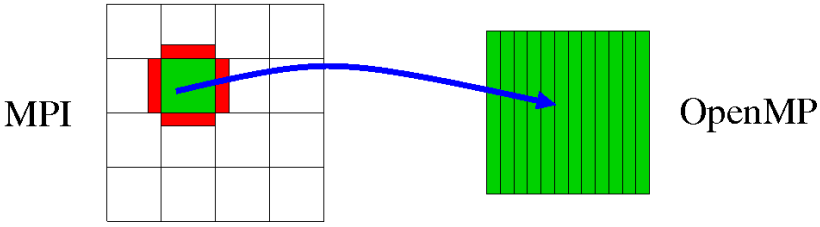
Fortran

```
program laplace
   ...DICHIARAZIONE DI VARIABILI...
   call MPI_Init(ierr)
   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
   call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
   ...INPUT, ALLOCAZIONE, GESTIONE GRIGLIA CARTESIANA E CONDIZIONE INIZIALE...
   do while (var > tol .and. iter <= maxIter)
      iter = iter + 1 ; var = 0.d0 ; myvar = 0.d0
      buffer_s_rl(1:mymsize_y) = T(1,1:mymsize_y)
      !--- exchange boundary data with neighbours (right->left)
      call MPI_Sendrecv(buffer_s_rl,mymsize_y,MPI_DOUBLE_PRECISION,dest_rl, tag, &
                        buffer_r_rl,mymsize_y,MPI_DOUBLE_PRECISION,source_rl,tag, &
                        cartesianComm, status, ierr)
      if(source_rl >= 0) T(mymsize_x+1,1:mymsize_y) = buffer_r_rl(1:mymsize_y)
      ...SCAMBIO ALTRE HALO TRA PROCESSI MPI...
      do j = 1, mymsize_y
         do i = 1, mymsize_x
            Tnew(i,j) = 0.25d0 * ( T(i-1,j) + T(i+1,j) + T(i,j-1) + T(i,j+1) )
            myvar = max(myvar, abs( Tnew(i,j) - T(i,j) ))
         enddo
      enddo
      Tmp =>T; T =>Tnew; Tnew => Tmp;
      call MPI_Allreduce(myvar,var,1,MPI_DOUBLE_PRECISION,MPI_MAX,MPI_COMM_WORLD,ierr)
   end do
   ...DEALLOCAZIONE...
   call MPI_Finalize(ierr)
end program laplace
```

Laplace 2D: parallelizzazione ibrida I

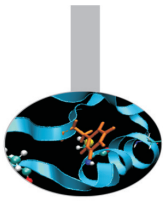


- Strategia di parallelizzazione ibrida: ogni blocco MPI esegue operazioni parallelizzate con OpenMP



- La strategia più semplice è parallelizzare con OpenMP aprendo e richiudendo la regione parallela direttamente in corrispondenza del loop di aggiornamento di T
- Le chiamate MPI funzionano normalmente, perché avvengono al di fuori delle regioni *multi-threaded*
  - si dovrebbe usare almeno **`MPI_THREAD_FUNNELED`** ...
  - ma funzionano sia **`MPI_THREAD_SINGLE`** che **`MPI_INIT`**

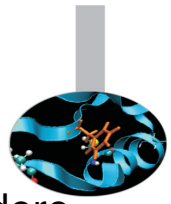
Laplace 2D: *mpi_thread_single*



Fortran

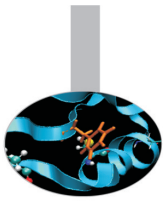
```
program laplace
   ...DICHIARAZIONE DI VARIABILI...
   call MPI_Init(ierr)
   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
   call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
   ...INPUT, ALLOCAZIONE, GESTIONE GRIGLIA CARTESIANA E CONDIZIONE INIZIALE...
   do while (var > tol .and. iter <= maxIter)
      iter = iter + 1 ; var = 0.d0 ; myvar = 0.d0
      buffer_s_rl(1:mymsize_y) = T(1,1:mymsize_y)
      !--- exchange boundary data with neighbours (right->left)
      call MPI_Sendrecv(buffer_s_rl,mymsize_y,MPI_DOUBLE_PRECISION,dest_rl, tag, &
                        buffer_r_rl,mymsize_y,MPI_DOUBLE_PRECISION,source_rl,tag, &
                        cartesianComm, status, ierr)
      if(source_rl >= 0) T(mymsize_x+1,1:mymsize_y) = buffer_r_rl(1:mymsize_y)
      ...SCAMBIO ALTRE HALO TRA PROCESSI MPI...
      !$omp parallel do reduction(max:myvar)
      do j = 1, mymsize_y
         do i = 1, mymsize_x
            Tnew(i,j) = 0.25d0 * ( T(i-1,j) + T(i+1,j) + T(i,j-1) + T(i,j+1) )
            myvar = max(myvar, abs( Tnew(i,j) - T(i,j) ))
         enddo
      enddo
      !$omp end parallel do
      Tmp =>T; T =>Tnew; Tnew => Tmp;
      call MPI_Allreduce(myvar,var,1,MPI_DOUBLE_PRECISION,MPI_MAX,MPI_COMM_WORLD,ierr)
   end do
   ...DEALLOCAZIONE...
   call MPI_Finalize(ierr)
end program laplace
```

Laplace 2D: parallelizzazione ibrida II



- Allarghiamo la regione parallela OpenMP in modo da includere le chiamate MPI di scambio halo
- Il vantaggio è la possibilità di sovrapporre calcolo e comunicazioni MPI
  - i *master thread* eseguono le comunicazioni delle halo e poi aggiornano le variabili che sono vicino al bordo
  - gli altri *thread* con il *master* quando ha terminato il punto precedente eseguono i calcoli che non richiedono le halo
  - è opportuno indicare uno *scheduling* per dare al *master* meno carico sui nodi interni da aggiornare
- Con la direttiva **`master`** il blocco di codice associato è eseguito solo dal *Master Thread*, gli altri thread lo saltano senza fermarsi
- Si dovrebbe usare **`MPI_THREAD_FUNNELED`** ...

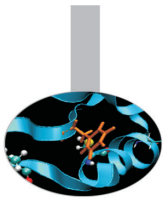
Laplace 2D: *mpi_thread_funneled*



Fortran

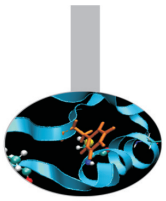
```
do while (var > tol .and. iter <= maxIter)
   iter = iter + 1 ; var = 0.d0 ; myvar = 0.d0 ; mastervar = 0.d0
   !$omp parallel
   !$omp master
   ...SCAMBIO HALO TRA PROCESSI MPI...
   do j = 1, mymsize_y, mymsize_y-1  ;  do i = 1, mymsize_x
       Tnew(i,j) = 0.25 * ( T(i-1,j) + T(i+1,j) + T(i,j-1) + T(i,j+1) )
       mastervar = max(mastervar, abs( Tnew(i,j) - T(i,j) ))
   enddo  ; enddo
   do j = 1, mymsize_y  ; do i = 1, mymsize_x, mymsize_x-1
       Tnew(i,j) = 0.25 * ( T(i-1,j) + T(i+1,j) + T(i,j-1) + T(i,j+1) )
       mastervar = max(mastervar, abs( Tnew(i,j) - T(i,j) ))
   enddo  ; enddo
   !$omp end master
   !$omp do reduction(max:myvar) schedule(dynamic,125)
   do j = 2, mymsize_y-1  ; do i = 2, mymsize_x-1
         Tnew(i,j) = 0.25d0 * ( T(i-1,j) + T(i+1,j) + T(i,j-1) + T(i,j+1) )
         myvar = max(myvar, abs( Tnew(i,j) - T(i,j) ))
   enddo  ; enddo
   !$omp end do
   !$omp master
   myvar = max(myvar,mastervar)
   !$omp end master
   !$omp end parallel
   Tmp =>T; T =>Tnew; Tnew => Tmp;
   call MPI_Allreduce(myvar,var,1,MPI_DOUBLE_PRECISION,MPI_MAX,MPI_COMM_WORLD,ierr)
end do
```


Laplace 2D: parallelizzazione ibrida III



- Per limitare gli overhead OpenMP, è possibile far sì che la regione parallela OpenMP includa anche il ciclo **`while`**
- Analogamente a prima, è necessario utilizzare barriere OpenMP esplicite per gestire il ciclo iterativo
  - È necessaria inoltre una barriera (implicita) dopo **`MPI_Allreduce`**. Perchè?
- La chiamata **`MPI_Allreduce`** all'interno della direttiva **`single`** richiederebbe almeno **`MPI_THREAD_SERIALIZED`**
  - ma sia livelli di supporto *thread* inferiori che **`MPI_Init`** possono funzionare

Laplace 2D: *mpi_thread_serialized*

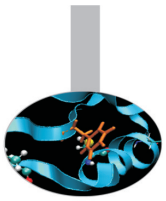


Fortran

```
!$omp parallel
do while (var > tol .and. iter <= maxIter)
   !$omp barrier
   !$omp single
   iter = iter + 1 ; var = 0.d0 ; myvar = 0.d0 ; mastervar = 0.d0
   !$omp end single
   !$omp master
   ...SCAMBIO HALO TRA PROCESSI MPI...
   do j = 1, mymsize_y, mymsize_y-1  ;  do i = 1, mymsize_x
       Tnew(i,j) = 0.25 * ( T(i-1,j) + T(i+1,j) + T(i,j-1) + T(i,j+1) )
       mastervar = max(mastervar, abs( Tnew(i,j) - T(i,j) ))
   enddo  ; enddo
   do j = 1, mymsize_y  ; do i = 1, mymsize_x, mymsize_x-1
       Tnew(i,j) = 0.25 * ( T(i-1,j) + T(i+1,j) + T(i,j-1) + T(i,j+1) )
       mastervar = max(mastervar, abs( Tnew(i,j) - T(i,j) ))
   enddo  ; enddo
   !$omp end master
   !$omp do reduction(max:myvar) schedule(dynamic,125)
   do j = 2, mymsize_y-1 ; do i = 2, mymsize_x-1
         Tnew(i,j) = 0.25d0 * ( T(i-1,j) + T(i+1,j) + T(i,j-1) + T(i,j+1) )
         myvar = max(myvar, abs( Tnew(i,j) - T(i,j) ))
   enddo  ; enddo
   !$omp end do
   !$omp single
   Tmp =>T; T =>Tnew; Tnew => Tmp;
   !$omp end single nowait
   !$omp single
   myvar = max(myvar,mastervar)
   call MPI_Allreduce(myvar,var,1,MPI_DOUBLE_PRECISION,MPI_MAX,MPI_COMM_WORLD,ierr)
   !$omp end single
enddo
!$omp end parallel
```

CINECA SCAI SuperComputing Applications and Innovation

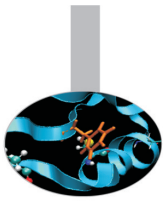
Laplace 2D: parallelizzazione ibrida IV



- Le 4 fasi di comunicazione sono effettuate ognuna da un thread usando **`single nowait`**
  - di fatto le chiamate MPI diventano non bloccanti
  - l'aggiornamento delle halo viene posticipato dopo quello dei nodi interni

- È necessario utilizzare **`MPI_THREAD_MULTIPLE`**
  - più *thread* che comunicano simultaneamente possono incrementare le performance di bandwidth
  - ma aumentando il supporto *thread* può aumentare anche l'*overhead* della libreria MPI

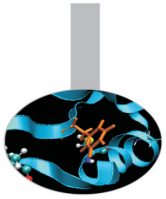
Laplace 2D: *mpi_thread_multiple*



Fortran

```
!$omp parallel
do while (var > tol .and. iter <= maxIter)
   !$omp barrier
   !$omp single
   iter = iter + 1 ; var = 0.d0 ; myvar = 0.d0
   !$omp end single
   !$omp single !--- exchange boundary data with neighbours (right->left)
   buffer_s_rl(1:mymsize_y) = T(1,1:mymsize_y) ; call MPI_Sendrecv(buffer_s_rl,...)
   if(source_rl >= 0) T(mymsize_x+1,1:mymsize_y) = buffer_r_rl(1:mymsize_y)
   !$omp end single nowait
   ...SCAMBIO ALTRE HALO TRA PROCESSI MPI TRA SINGLE NOWAIT...
   !$omp do reduction(max:myvar) schedule(dynamic,125)
   do j = 2, mymsize_y-1 ; do i = 2, mymsize_x-1
      Tnew(i,j) = 0.25d0 * ( T(i-1,j) + T(i+1,j) + T(i,j-1) + T(i,j+1) )
      myvar = max(myvar, abs( Tnew(i,j) - T(i,j) ))
   enddo  ; enddo
   !$omp end do
   !$omp do reduction(max:myvar)
   do j = 1, mymsize_y, mymsize_y-1  ;  do i = 1, mymsize_x
      Tnew(i,j) = 0.25 * ( T(i-1,j) + T(i+1,j) + T(i,j-1) + T(i,j+1) )
      myvar = max(myvar, abs( Tnew(i,j) - T(i,j) ))
   enddo  ; enddo
   !$omp end do
   ...AGGIORNAMENTO ALTRE HALO...
   !$omp single
   Tmp =>T; T =>Tnew; Tnew => Tmp;
   !$omp end single nowait
   !$omp single
   call MPI_Allreduce(myvar,var,1,MPI_DOUBLE_PRECISION,MPI_MAX,MPI_COMM_WORLD,ierr)
   !$omp end single
enddo
!$omp end parallel
```

Scaling Laplace 2D: esercitazione



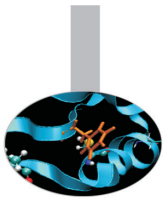
- Configuriamo l'ambiente e completiamo le tabelle
  - **module load env/experimental**
  - **module load compilers/gnu/4.7**
  - **module load libs/openmpi-mt/1.6.1**
- Strong scaling: griglia 5000 $\times$ 5000 (100 iterate)

MPI/OMP	1	2	4	8
1	18.4	10.7		7.9
2	9.5		4.2	4.2
4		4.0	2.2	2.2
8	4.0	2.1	1.1	-
16	3.1	1.1	-	-
32	1.3	-	-	-

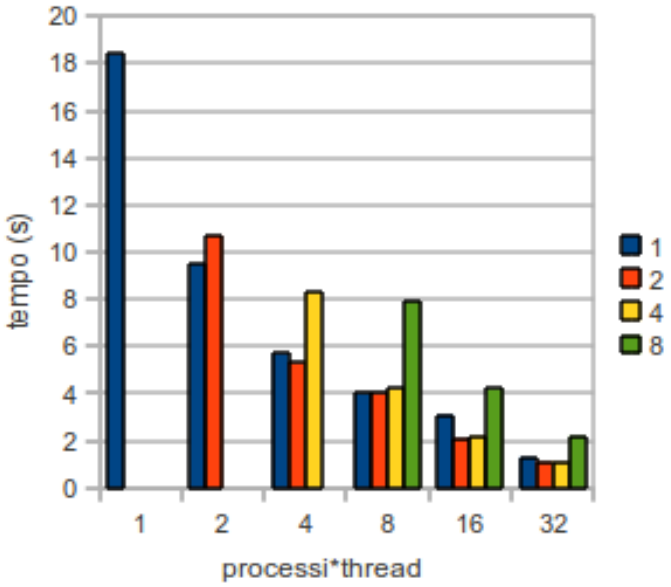
- Weak scaling: griglia 25000000 punti per processo/thread (100 iterate)

MPI/OMP	1	2	4	8
1	18.4	21.3		62
2	18.8		35	62
4		31.5	33	62.5
8	31.1	31.8	35.5	-
16	31.5	31.8	-	-
32	33	-	-	-

Strong scaling: conviene l'ibrido?



- Analizzando le performance per numero di processi*thread fissato al variare del numero di thread usati si può capire se e quale livello di parallelizzazione ibrida sia ottimale



CINECA SCAI
SuperComputing Applications and Innovation