OpenMP and Performance

Ruud van der Pas

Senior Staff Engineer
Technical Developer Tools
Sun Microsystems, Menlo Park, CA, USA

"Parallel Programming in Computational Engineering and Science"
RWTH Aachen University, Aachen, Germany
March 23-27, 2009

Outline

- *Case Study #1 - Neural Network*
- *Case Study #2 - Matrix Summation*
- *Case Study #3 - A 3D Matrix Update*
- *Case Study #4 - Matrix * Vector*

PPCES 2009
RWTH Aachen
March 23-27
2009

3

OpenMP and Performance

- *The transparency of OpenMP is a mixed blessing*
  - *Makes things pretty easy*
  - *May mask performance bottlenecks*
- *In the ideal world, an OpenMP application just performs well*
- *Unfortunately, this is not the case*
- *Two of the more obscure effects that can negatively impact performance are **cc-NUMA behavior** and **False Sharing***
- *Neither of these are restricted to OpenMP, but they are important enough to cover in some detail here*

PPCES 2009
RWTH Aachen
March 23-27
2009

4

Case Study #1
Neural Network

Neural Network application*

Performance Analyzer Output

Excl. User sec.	CPU %	Incl. User CPU sec.	Excl. Wall sec.	Name
120.710	100.0	120.710	128.310	<Total>
116.960	96.9	116.960	122.610	calc_r_loop_on_neighbours
0.900	0.7	118.630	0.920	calc_r
0.590	0.5	1.380	0.590	_doprnt
0.410	0.3	1.030	0.430	init_visual_input_on_V1
0.280	0.2	0.280	1.900	_write
0.200	0.2	0.200	0.200	round_coord_cyclic
0.130	0.1	0.130	0.140	__arint_set_n
0.130	0.1	0.550	0.140	__k_double_to_decimal
0.090	0.1	1.180	0.090	fprintf

Callers-callees fragment:

Attr. User CPU sec.	Excl. User CPU sec.	Incl. User CPU sec.	Name
116.960	0.900	118.630	calc_r
116.960	116.960	116.960	*calc_r_loop_on_neighbours

***) Program was said not to scale on a Sun SMP system....**

Source line information

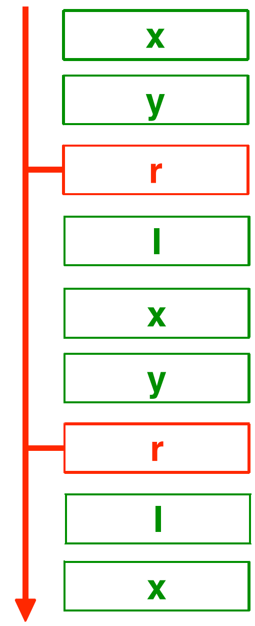
What is the problem ?

```
      struct cell{
        double x; double y; double r; double I;
       };
                           ......

       struct cell V1[NPOSITIONS_Y][NPOSITIONS_X];
       double       h[NPOSITIONS][NPOSITIONS];
                           ......
   Excl. User CPU  Excl. Wall
     sec.      %      sec.
                                 1040. void
                                 1041. calc_r_loop_on_neighbours
                                             (int y1, int x1)
     0.080    0.1      0.080     1042. {
                                 1043. struct interaction_structure *next_p;
                                 1044.
     0.130    0.1      0.130     1045. for (next_p = JJ[y1][x1].next;
     0.460    0.4      0.470     1046.      next_p != NULL;
                                 1047.      next_p = next_p->next) {
## 116.290   96.3    121.930     1048.     h[y1][x1] += next_p->strength *
                                              V1[next_p->y][next_p->x].r;
                                 1049.
                                 1052.    }
                                 1053. }
```

96% of the time spent in this single statement

Data structure problem

- ❑ *We only use 1/4 of a cache line !*
- ❑ *For sufficiently large problems this will:*
  - *Generate additional memory traffic*
    - ✔ *Higher interconnect pressure*
  - *Waste data cache capacity*
    - ✔ *Reduces temporal locality*
- ❑ *The above negatively affects both serial and parallel performance*
- ❑ *Fix: split the structure into two parts*
  - *One contains the "r" values only*
  - *The other one contains the {x,y,l} sets*



Fragment of modified code

```
double V1_R[NPOSITIONS_Y][NPOSITIONS_X];

void
calc_r_loop_on_neighbours(int y1, int x1)
{
  struct interaction_structure *next_p;

  double sum = h[y1][x1];

  for (next_p = JJ[y1][x1].next;
       next_p != NULL;
       next_p = next_p->next) {
       sum += next_p->strength * V1_R[next_p->y][next_p->x];
  }
  h[y1][x1] = sum;
}
```

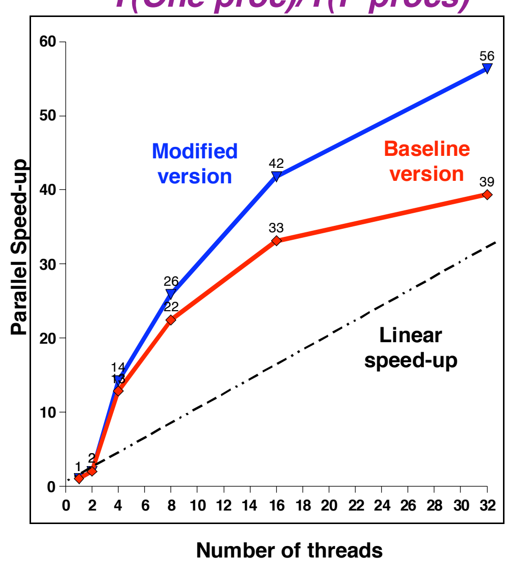
Parallelization with OpenMP

```
void calc_r(int t)
{

#include <omp.h>

#pragma omp parallel for default(none)  \
        private(y1,x1) shared(h,V1,g,T,beta_inv,beta)

  for (y1 = 0; y1 < NPOSITIONS_Y; y1++) {
    for (x1 = 0; x1 < NPOSITIONS_X; x1++) {

      calc_r_loop_on_neighbours(y1,x1);
      h[y1][x1] += V1[y1][x1].I;

      <statements deleted>

    }
  }

/*-- End of OpenMP parallel for --*/
```

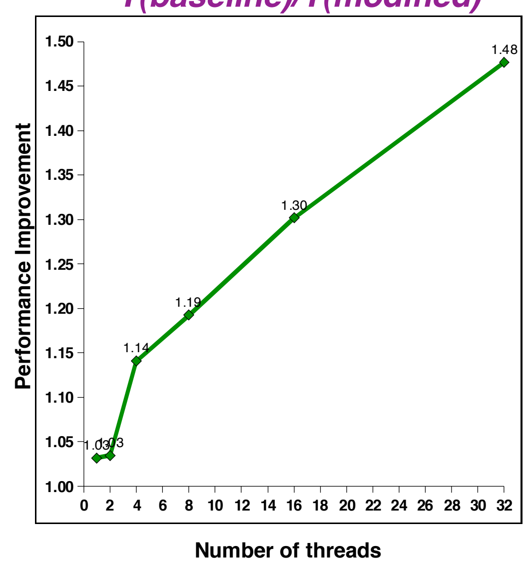
Can be executed in parallel

Scalability Results

T(One proc)/T(P procs)



T(baseline)/T(modified)



Note:
Single processor run time is 5001 seconds for the baseline version (4847 for the modified version)

Case Study #2
Matrix Summation

The Implementation

```
DO J = 1, P
   DO I = 1, N
      SUM = SUM + A(I,J)
   END DO
END DO
```

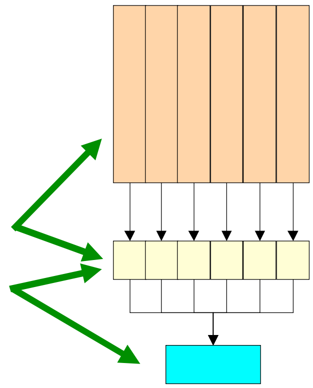
Proc	T=1	T=2
1	SUM = SUM+A(1,1)	
2	SUM = SUM+A(2,1)	... etc ...
3	SUM = SUM+A(3,1)	
4	SUM = SUM+A(4,1)	

Race Condition

Time

About this example

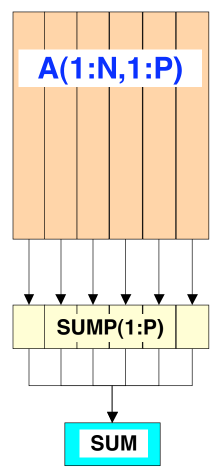
- *As written by the user, we can not execute this calculation in parallel*
- *The operation can easily be parallelized though:*
  1. *Sum the individual columns of the array*
  2. *Accumulate these individual values into the global sum*
- *We can now transform the code to implement this idea*



Parallelizing a summation

```
!$omp parallel do shared(N,P,sump,a) &
!$omp private(i,j)
  DO J = 1, P
     SUMP(J) = 0.0
     DO I = 1, N
        SUMP(J) = SUMP(J) + A(I,J)
     END DO
  END DO
!$omp end parallel do
```
parallel

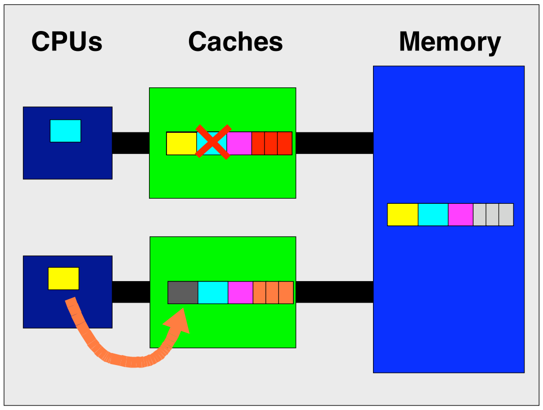
```
DO J = 1, P
   SUM = SUM + SUMP(J)
END DO
```
serial



☞ *Serial part can be parallelized with a binary tree algorithm*

False Sharing

A store into a shared cache line invalidates the other copies of that line:



The system is not able to distinguish between changes within one individual line

False Sharing Red Flags

- ***Be alert, when all of these three conditions are met:***
  - *Shared data is modified by multiple processors*
  - *Multiple threads operate on the same cache line(s)*
  - *Update occurs simultaneously and very frequently*
- ***Use local data where possible***
- ***Shared read-only data does not lead to false sharing***

Comments:

☞ *In a Distributed Memory programming model, data is local by default and explicitly shared by exchanging messages/buffers*

☞ *In a Shared Memory programming model, it is often the other way round: most data is shared by default and has to be made private explicitly*

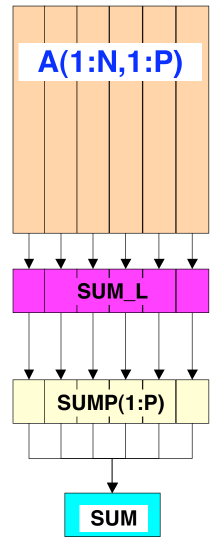
Reducing False Sharing

```
!$omp parallel do shared(N,P,sump,a) &
!$omp private(i,j,sum_L)
  DO J = 1, P
     SUM_L = 0.0
     DO I = 1, N
        SUM_L = SUM_L + A(I,J)
     END DO
     SUMP(J) = SUM_L
  END DO
!$omp end parallel
```

parallel

```
DO J = 1, P
   SUM = SUM + SUMP(J)
END DO
```

serial



Example - Code version 1

```
      subroutine sum_shared(m,n,a,sumP,sum)
             ....
!$omp parallel do default(none) private(j) &
!$omp shared(m,n,a,sumP)
      do j = 1, n
         call sum_vector(m,a(1,j),sumP(j))
      end do
!$omp end parallel do

      sum = 0.0
      do j = 1, n
         sum = sum + sumP(j)
      end do
        ....
      end
```

shared data

```
subroutine sum_vector(m,x,sumvec)
          ....
sumvec = 0.0
do i = 1, m
   sumvec = sumvec + x(i)
end do
     ....
end
```

Example - Code version 2

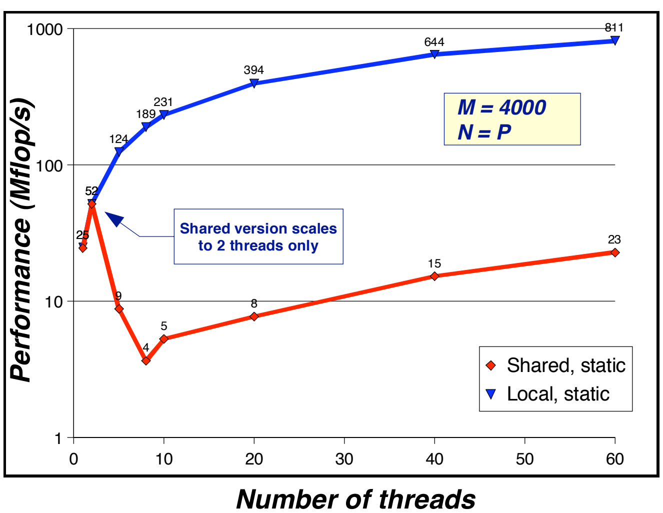
```
      subroutine sum_local(m,n,a,sumP,sum)
             ....
!$omp parallel do default(none) private(j,sum_L)&
!$omp shared(m,n,a,sumP)
      do j = 1, n
         call sum_vector(m,a(1,j),sum_L)
         sumP(j) = sum_L
      end do
!$omp end parallel do

      sum = 0.0
      do j = 1, n
         sum = sum + sumP(j)
      end do
        ....
```

local data

```
subroutine sum_vector(m,x,sumvec)
          ....
sumvec = 0.0
do i = 1, m
   sumvec = sumvec + x(i)
end do
     ....
end
```


PPCES 2009
RWTH Aachen
March 23-27
2009

Example - The performance




```
do j = 1, P
   do i = 1, M
      sum = sum + A(i,j)
   end do
end do
```

Each processor works on one column of the matrix only

Each column fits in the L2 cache

Routine sum_vector compiled with -xO2 -dalign

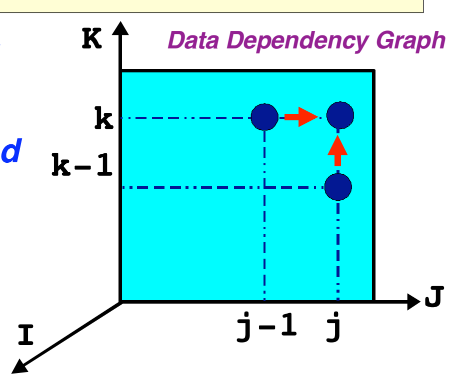
Shared version doesn't scale at all

Case Study #3
A 3D Matrix Update

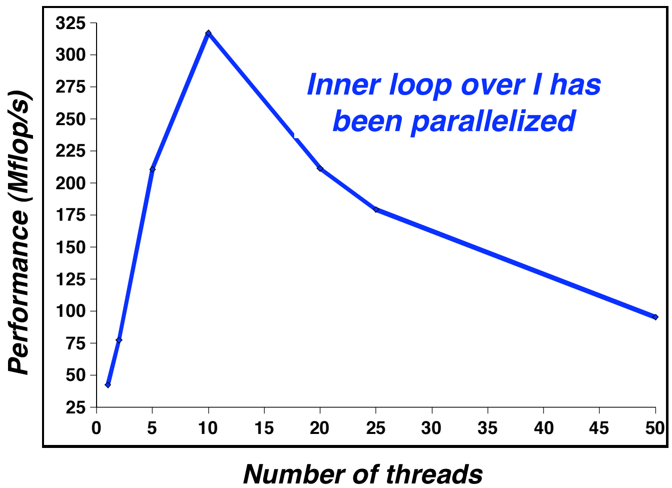
A 3D matrix update

```
      do k = 2, n
         do j = 2, n
!$omp parallel do default(shared) private(i) &
!$omp schedule(static)
            do i = 1, m
               x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
            end do
!$omp end parallel do
         end do
      end do
```

- *The loops are correctly nested for serial performance*
- *Due to a data dependency on J and K, only the inner loop can be parallelized*
- *This will cause the barrier to be executed (N-1) ^{2} times*



The performance



Scaling is very poor (as to be expected)

Dimensions : M=7,500 N=20
Footprint : ~24 MByte

Performance Analyzer data

Using 10 threads

Name	Excl. User CPU sec.	%	Incl. User CPU sec.	Excl. Wall sec.
<Total>	10.590	100.0	10.590	1.550
__mt_EndOfTask_Barrier_	5.740	54.2	5.740	0.240
__mt_WaitForWork_	3.860	36.4	3.860	0.
__mt_MasterFunction_	0.480	4.5	0.680	0.480
MAIN_	0.230	2.2	1.200	0.470
block_3d_ -- MP doall from line 14 [_$d1A14 ... d_]	0.170	1.6	5.910	0.170
block_3d_	0.040	0.4	6.460	0.040
memset	0.030	0.3	0.030	0.080

Using 20 threads

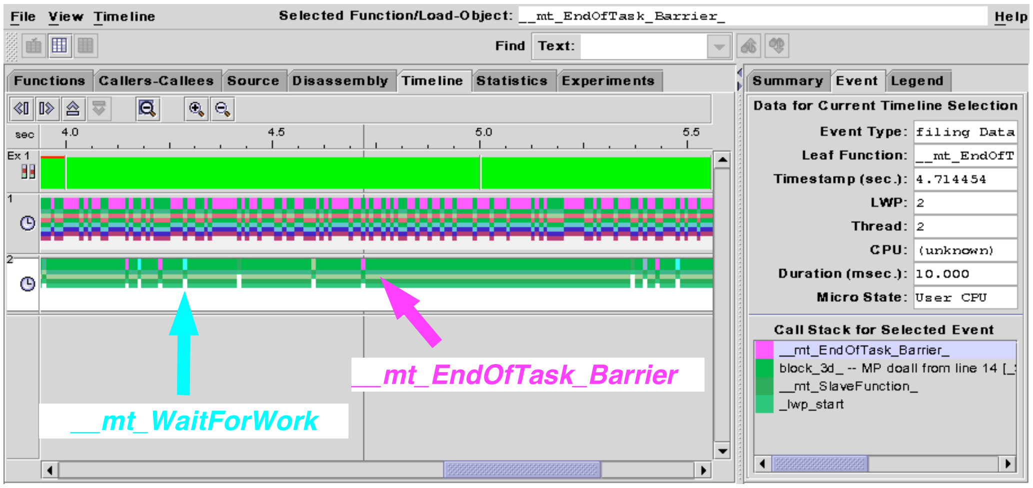
Name	Excl. User CPU sec.	%	Incl. User CPU sec.	Excl. Wall sec.
<Total>	47.120	100.0	47.120	2.900
__mt_EndOfTask_Barrier_	25.700	54.5	25.700	0.980
__mt_WaitForWork_	19.880	42.2	19.880	0.
__mt_MasterFunction_	1.100	2.3	1.320	1.100
MAIN_	0.190	0.4	2.520	0.470
block_3d_ -- MP doall from line 14 [_$d1A14.block_3d_]	0.100	0.2	25.800	0.100
__mt_setup_doJob_int_	0.080	0.2	0.080	0.080
__mt_setup_job_	0.020	0.0	0.020	0.020
block_3d_	0.010	0.0	27.020	0.010

do not scale at all

scales somewhat

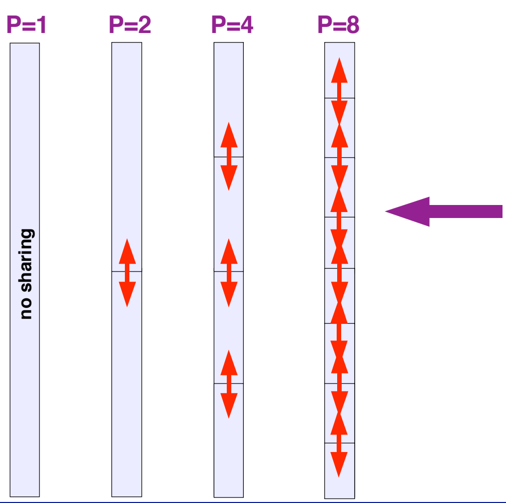
Question: Why is __mt_WaitForWork so high in the profile ?

The Analyzer Timeline overview



This is False Sharing at work !

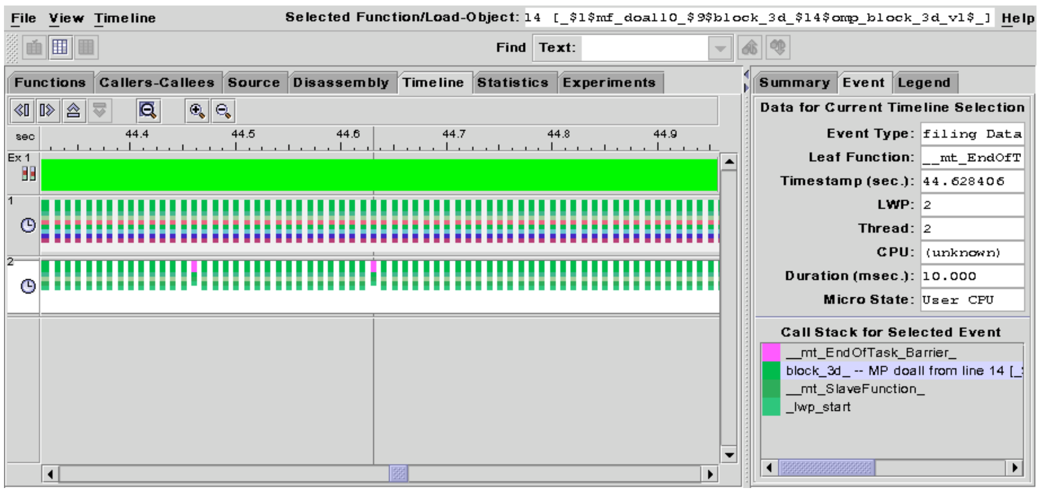
```
!$omp parallel do default(shared) private(i) &
!$omp schedule(static)
         do i = 1, m
            x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
         end do
!$omp end parallel do
```



False sharing increases as we increase the number of threads

PPCES 2009
RWTH Aachen
March 23-27
2009
27

Sanity Check: Setting M=75000*

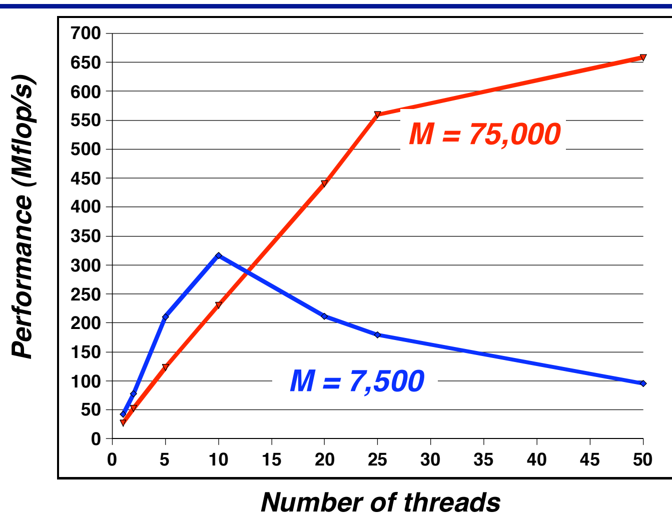


Only a very few barrier calls now

****) Increasing the length of the loop should decrease false sharing***

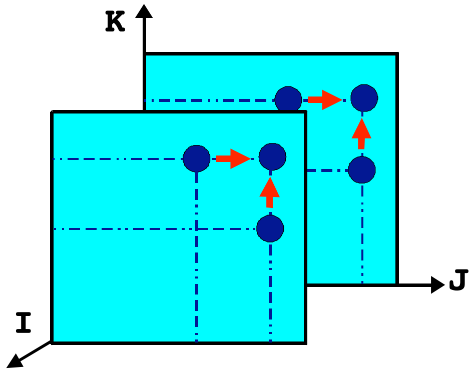
PPCES 2009
RWTH Aachen
March 23-27
2009
28

Performance comparison



For a higher value of M, the program scales better

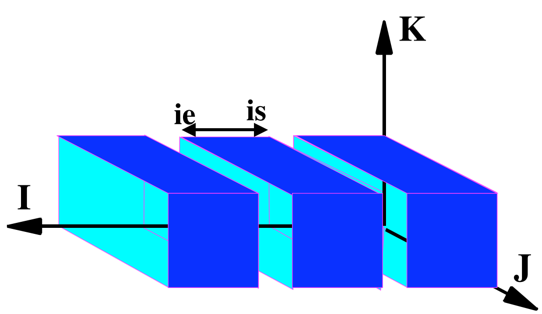
Observation



- No data dependency on 'I'
- Therefore we can split the 3D matrix in larger blocks and process these in parallel

```
do k = 2, n
   do j = 2, n
      do i = 1, m
         x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
      end do
   end do
end do
```


The Idea



- We need to distribute the M iterations over the number of processors
- We do this by controlling the start (IS) and end (IE) value of the inner loop
- Each thread will calculate these values for it's portion of the work

```
do k = 2, n
   do j = 2, n
      do i = is, ie
         x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
      end do
   end do
end do
```


The first implementation

```
      use omp_lib
         ........
      nrem   = mod(m,nthreads)
      nchunk = (m-nrem)/nthreads

!$omp parallel default (none)&
!$omp private (P,is,ie)      &
!$omp shared  (nrem,nchunk,m,n,x,scale)

      P = omp_get_thread_num()

      if ( P < nrem ) then
        is = 1 + P*(nchunk + 1)
        ie = is + nchunk
      else
        is = 1 + P*nchunk+ nrem
        ie = is + nchunk - 1
      end if

      call kernel(is,ie,m,n,x,scale)

!$omp end parallel
```

```
subroutine kernel(is,ie,m,n,x,scale)
         .......
do k = 2, n
 do j = 2, n
    do i = is, ie
       x(i,j,k)=x(i,j,k-1)+x(i,j-1,k)*scale
    end do
  end do
end do
```


Another Idea: Use OpenMP !

```
      use omp_lib

      implicit none
      integer     :: is, ie, m, n
      real(kind=8):: x(m,n,n), scale
      integer     :: i, j, k

!$omp parallel default(none) &
!$omp private(i,j,k) shared(m,n,scale,x)
      do k = 2, n
         do j = 2, n
!$omp do schedule(static)
            do i = 1, m
               x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
            end do
!$omp end do nowait
         end do
      end do
!$omp end parallel
```

How this works on 2 threads

Thread 0 Executes:		Thread 1 Executes:
`k=2` `j=2`	parallel region	`k=2` `j=2`

`do i = 1,m/2`
`   x(i,2,2) = ...`
`end do`
	work sharing	`do i = m/2+1,m`
		`   x(i,2,2) = ...`
		`end do`

`k=2` `j=3`	parallel region	`k=2` `j=3`

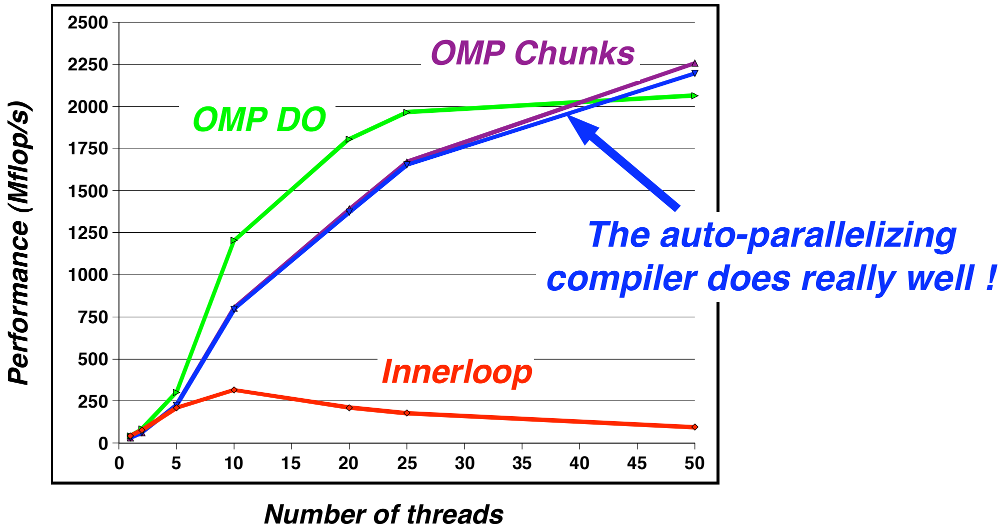
`do i = 1,m/2`
`   x(i,3,2) = ...`
`end do`
	work sharing	`do i = m/2+1,m`
		`   x(i,3,2) = ...`
		`end do`

This splits the operation in a way that is similar to our manual implementation

Performance

- ***We have set M=7500 N=20***
  - *This problem size does not scale at all when we explicitly parallelized the inner loop over 'I'*
- ***We have have tested 4 versions of this program***
  - *Inner Loop Over 'I' - Our first OpenMP version*
  - *AutoPar - The automatically parallelized version of 'kernel'*
  - *OMP_Chunks - The manually parallelized version with our explicit calculation of the chunks*
  - *OMP_DO - The version with the OpenMP parallel region and work-sharing DO*

The performance (M=7,500)



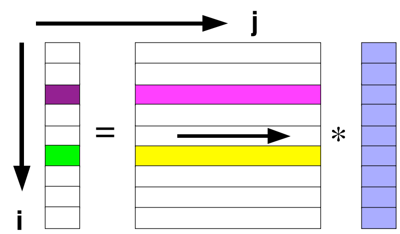
Dimensions : M=7,500 N=20
Footprint : ~24 MByte

Case Study #4
*Matrix * Vector*

PPCES 2009
RWTH Aachen
March 23-27
2009
37

The Sequential Source

```
for (i=0; i<m; i++)
{
  a[i] = 0.0;
  for (j=0; j<n; j++)
     a[i] += b[i*n+j]*c[j];
}
```

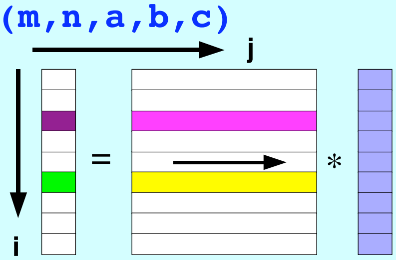


RvdP/V1 OpenMP and Performance

PPCES 2009
RWTH Aachen
March 23-27
2009
38

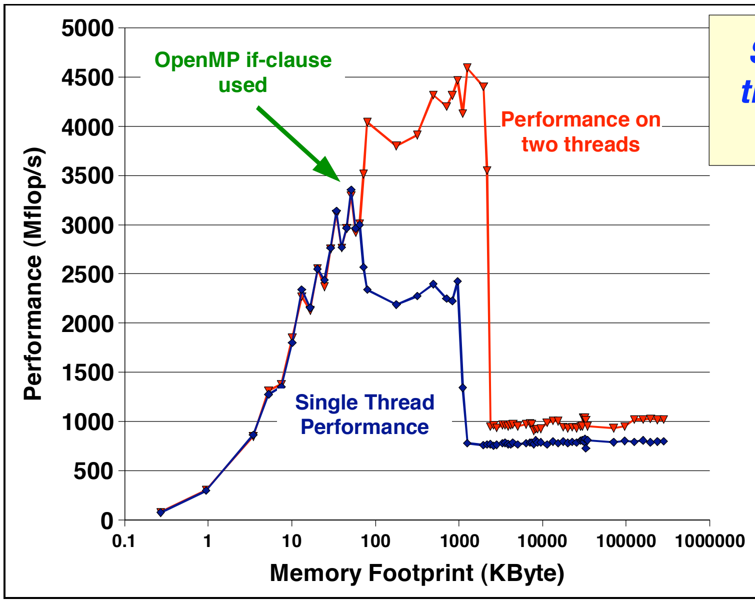
The OpenMP Source

```
#pragma omp parallel for default(none) \
        private(i,j) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
  a[i] = 0.0;
  for (j=0; j<n; j++)
     a[i] += b[i*n+j]*c[j];
 }
```



RvdP/V1 OpenMP and Performance

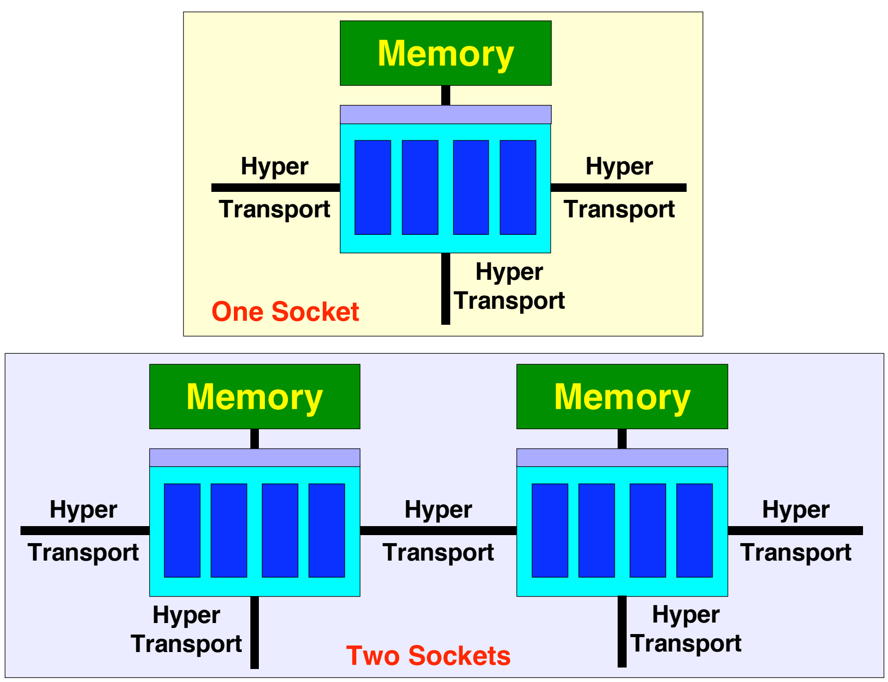
Example - Matrix times vector



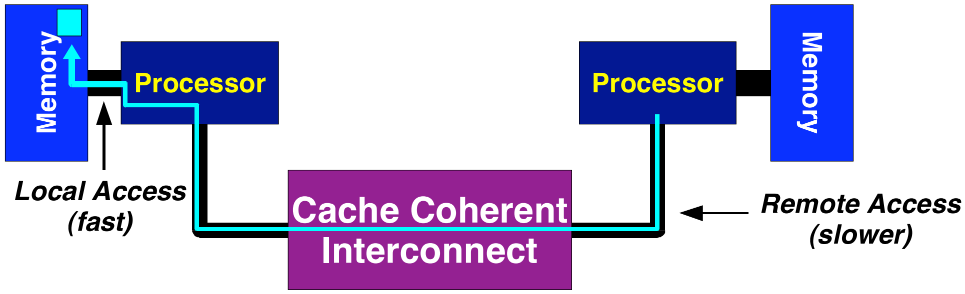
Scalability on 2 threads is rather disappointing

AMD Opteron Rev F @2600MHz
L1 cache : 64 KByte
Peak speed : 5200 Mflop/s

The Opteron Processor



A generic cc-NUMA architecture



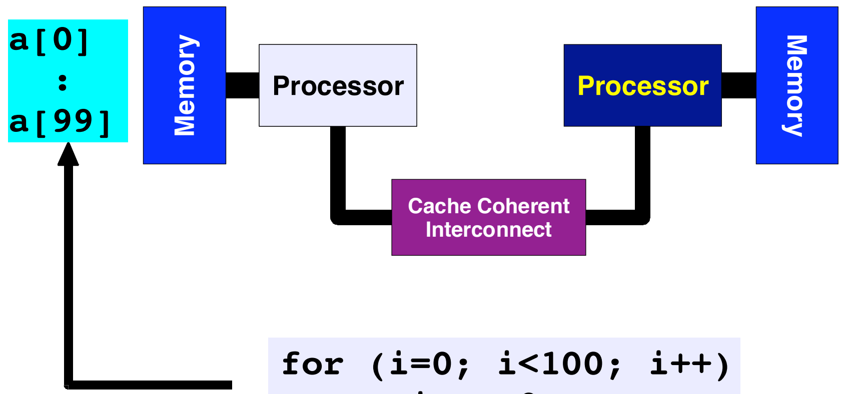
Main Issue: How To Distribute The Data ?

About Data Distribution

- ❑ *Important aspect on a cc-NUMA system*
  - *If not optimal - longer access times, memory hotspots*
- ❑ *OpenMP does not provide support for cc-NUMA*
- ❑ *Placement comes from the Operating System*
  - *This is therefore Operating System dependent*
- ❑ *Solaris: Memory Placement Optimization (MPO)*
  - *By default uses "First Touch" to place data*
  - *Use the "madvise" (3C) system call to control/change*
    - ✔ *Low level, but flexible, API to handle placement*
  - *Useful on some Sun Fire™ servers as well as AMD Opteron multi-processor systems*

PPCES 2009
RWTH Aachen
March 23-27
2009
43

About "First Touch" placement/1



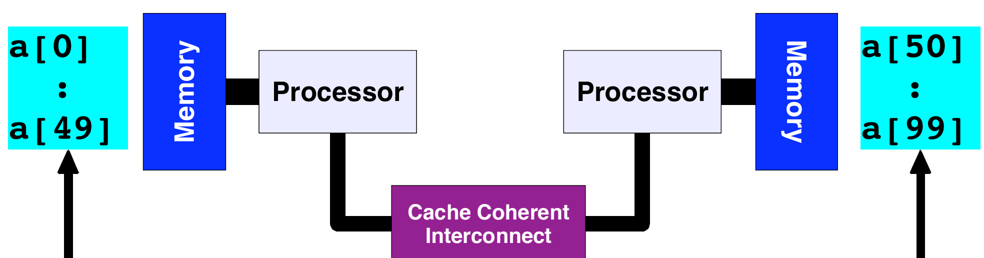

```
for (i=0; i<100; i++)
    a[i] = 0;
```

First Touch
All array elements are in the memory of the processor executing this thread

RvdP/V1 OpenMP and Performance

PPCES 2009
RWTH Aachen
March 23-27
2009
44

About "First Touch" placement/2



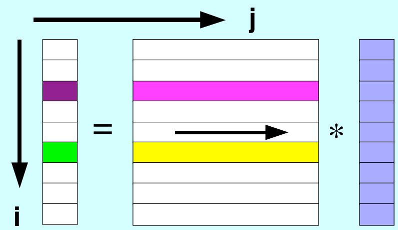

```
#pragma omp parallel for num_threads(2)
for (i=0; i<100; i++)
    a[i] = 0;
```

First Touch
Both memories each have "their half" of the array

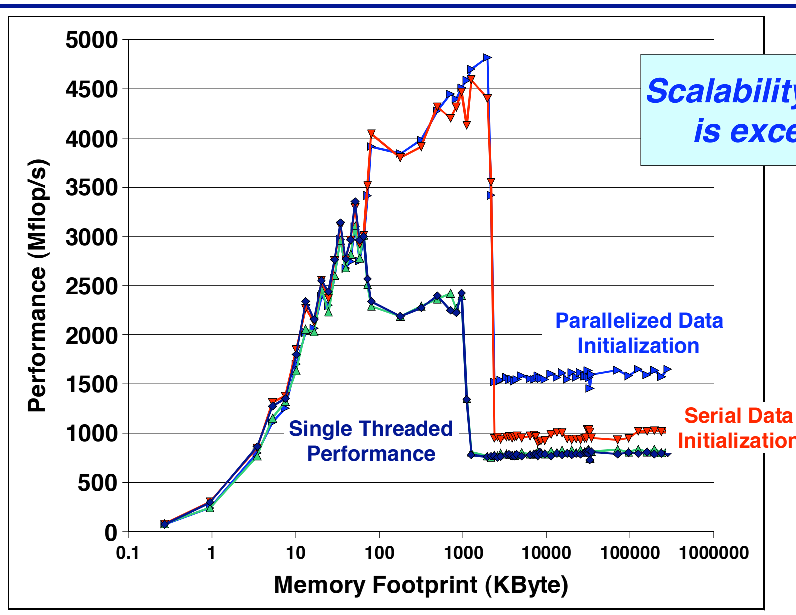
RvdP/V1 OpenMP and Performance

Data Initialization

```
#pragma omp parallel default(none) \
        shared(m,n,a,b,c) private(i,j)
{
#pragma omp for
   for (j=0; j<n; j++)
      c[j] = 1.0;

#pragma omp for
   for (i=0; i<m; i++)
   {
      a[i] = -1957.0;
      for (j=0; j<n; j++)
         b[i*n+j] = i;
   } /*-- End of omp for --*/

} /*-- End of parallel region --*/
```



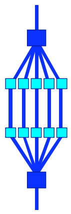
Example - Exploit First Touch



Scalability on 2 threads is excellent now !

The only change is the way the data is distributed over the system

AMD Opteron Rev F @2600MHz
L1 cache : 64 KByte
Peak speed : 5200 Mflop/s



That's It

Thank You and ..... Stay Tuned !