

How to Get Good Performance by Using OpenMP



Agenda

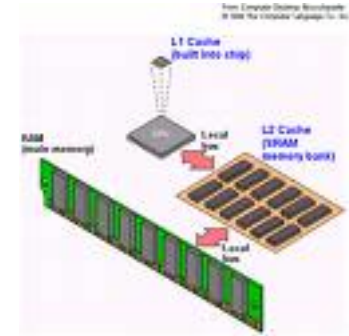
- Loop optimizations
- Measuring OpenMP performance
- Best practices
- Task parallelism



Correctness versus performance

It may be easy to write a correctly functioning OpenMP program, but not so easy to create a program that provides the desired level of performance.

Memory access patterns



A major goal is to organize data accesses so that values are used as often as possible while they are still in the cache.

Two-dimensional array access

In C, a two-dimensional array is stored in rows.

```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        sum += a[i][j];
```

Figure 5.1: Example of good memory access – Array **a** is accessed along the rows. This approach ensures good performance from the memory system.

```
for (int j=0; j<n; j++)
    for (int i=0; i<n; i++)
        sum += a[i][j];
```

Figure 5.2: Example of bad memory access – Array **a** is accessed columnwise. This approach results in poor utilization of the memory system. The larger the array, the worse its performance will be.

Two-dimensional array access

Empirical test on alvin:

$n = 50000$

row-wise access: 34.8 seconds

column-wise access: 213.3 seconds

Loop unrolling

```
for (int i=1; i<n; i++) {  
    a[i] = b[i] + 1;  
    c[i] = a[i] + a[i-1] + b[i-1];  
}
```

Figure 5.3: **A short loop nest** – Loop overheads are relatively high when each iteration has a small number of operations.

```
for (int i=1; i<n; i+=2) {  
    a[i] = b[i] + 1;  
    c[i] = a[i] + a[i-1] + b[i-1];  
    a[i+1] = b[i+1] + 1;  
    c[i+1] = a[i+1] + a[i] + b[i];  
}
```

Figure 5.4: **An unrolled loop** – The loop of Figure 5.3 has been unrolled by a factor of 2 to reduce the loop overheads. We assume the number of iterations is divisible by 2.

Loop fusion

```
for (int i=0; i<n; i++)
    a[i] = b[i] * 2;
for (int i=0; i<n; i++)
{
    x[i] = 2 * x[i];
    c[i] = a[i] + 2;
}
```

Figure 5.10: A pair of loops that both access array **a** – The second loop reuses element **a[i]**, but by the time it is executed, the cache line this element is part of may no longer be in the cache.

```
for (int i=0; i<n; i++)
{
    a[i] = b[i] * 2;
    c[i] = a[i] + 2;
    x[i] = 2 * x[i];
}
```

Figure 5.11: An example of loop fusion – The pair of loops of Figure 5.10 have been combined and the statements reordered. This permits the values of array **a** to be immediately reused.

Loop fission

```
for (int i=0; i<n; i++)
{
    c[i] = exp(i/n) ;
    for (int j=0; j<m; j++)
        a[j][i] = b[j][i] + d[j] * e[i];
}
```

Figure 5.12: A loop with poor cache utilization and bad memory access – If we can split off the updates to array *c* from the rest of the work, loop interchange can be applied to fix this problem.

```
for (int i=0; i<n; i++)
    c[i] = exp(i/n) ;

for (int j=0; j<m; j++)
    for (int i=0; i<n; i++)
        a[j][i] = b[j][i] + d[j] * e[i];
```

Figure 5.13: **Loop fission** – The loop nest of Figure 5.12 has been split into a pair of loops, followed by loop interchange applied to the second loop to improve cache usage.

Loop tiling

```
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
    b[i][j] = a[j][i];
```

Figure 5.14: A nested loop implementing an array transpose operation – Loop interchange does not improve its use of cache or TLB. A fresh approach is needed.

```
for (int j1=0; j1<n; j1+=nbj)
  for (int i=0; i<n; i++)
    for (int j2=0; j2 < MIN(n-j1,nbj); j2++)
      b[i][j1+j2] = a[j1+j2][i];
```

Figure 5.15: Loop tiling applied to matrix transpose – Here we have used loop tiling to split the inner loop into a pair of loops. This reduces TLB and cache misses.

cont'd on next page

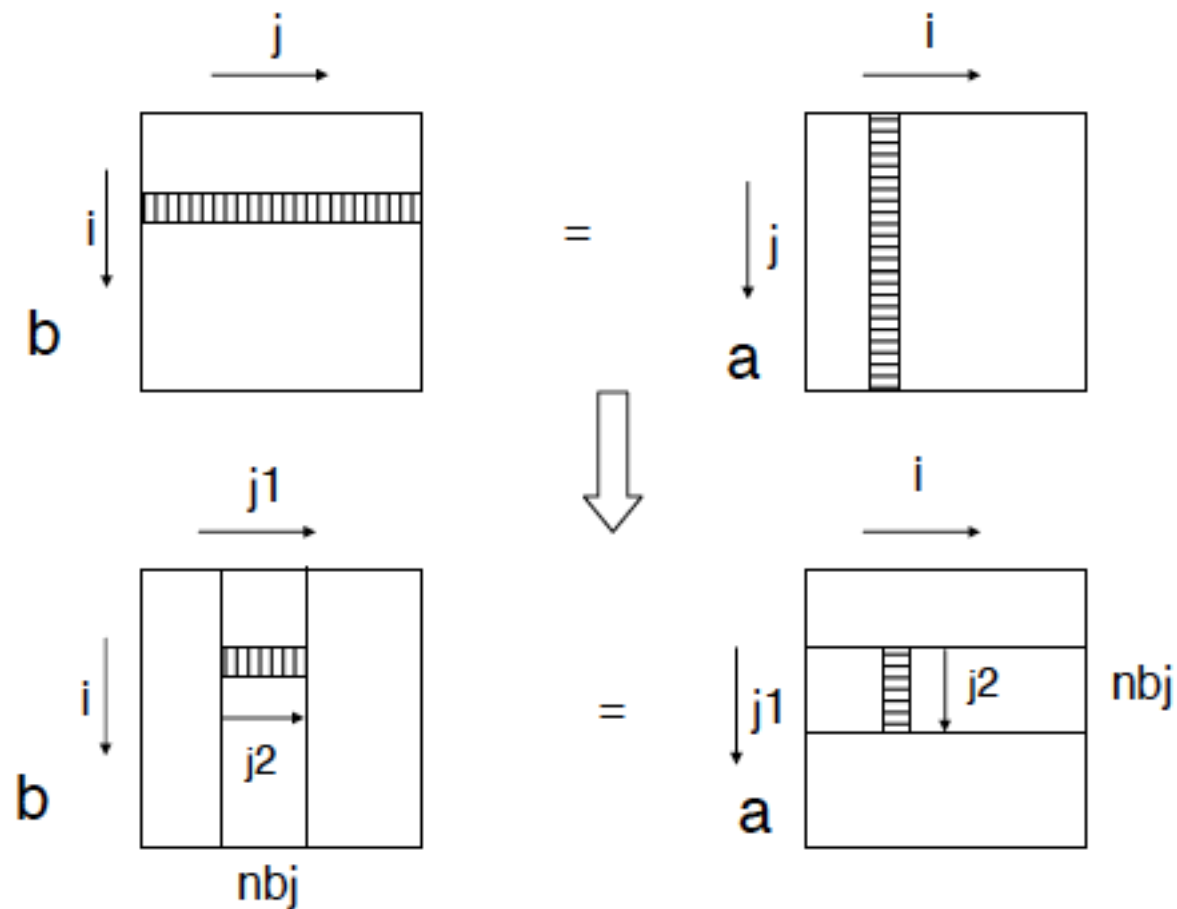


Figure 5.16: **Array access pattern** – Here we see how arrays **a** and **b** are accessed before and after *loop tiling*. The revised version accesses fewer pages per outer loop iteration.

Measuring OpenMP performance

(1) Using the `time` command available on Unix systems:

```
$ time program
```

```
real    5.4
```

```
user    3.2
```

```
sys     2.0
```

(2) Using the `omp_get_wtime()` function.

Returns the wall clock time (in seconds) relative to an arbitrary reference time.

Parallel overhead



The amount of time required to coordinate parallel threads, as opposed to doing useful work.

Parallel overhead can include factors such as:

- Thread start-up time
- Synchronization
- Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
- Thread termination time

A simple performance model

$$T_{CPU}(P) = (1 + O_p \cdot P) \cdot T_{serial}$$

$$T_{Elapsed}(P) = \left(\frac{f}{P} - f + 1 + O_p \cdot P\right) \cdot T_{serial}$$

In this model, T_{serial} is the CPU time of the original serial version of the application. The number of processors is given by P . The parallel overhead is denoted by $O_p \cdot P$, with O_p assumed to be a constant percentage (this is a simplification, as the overheads may well increase as the number of processors grows).

The fraction of execution time that has been parallelized is specified by $f \in [0, 1]$. Both $f = 0$ and $f = 1$ are extreme cases. A value of zero for f implies that application is serial. A perfectly parallel application corresponds to $f = 1$.



Table 5.1: Parallel performance and speedup for $f=0.95$ and 2% overhead
– The elapsed time goes down, whereas the total CPU time goes up. Parallel speedup is calculated from the elapsed time, using the serial version as the reference.

Version	Number of Processors	CPU time (seconds)	Elapsed time (seconds)	Speedup	Efficiency (%)
Serial	1	10.20	10.20	1.00	100
Parallel	1	10.40	10.40	0.98	98
	2	10.61	5.76	1.77	88
	4	11.02	3.75	2.72	68
	8	11.83	3.35	3.04	38

$$Speedup(P) = \frac{T_{Serial}(P)}{T_{Elapsed}(P)} = \frac{1}{\frac{f}{P} - f + 1 + O_p \cdot P} = \frac{1}{\frac{0.95}{P} + 0.05 + 0.02 \cdot P}$$
$$Efficiency(P) = \frac{Speedup(P)}{P}$$

Performance factors



- Manner in which memory is accessed by the individual threads.
- *Sequential overheads*: Sequential work that is replicated.
- *(OpenMP) Parallelization overheads*: The amount of time spent handling OpenMP constructs.
- *Load imbalance overheads*: The load imbalance between synchronization points.
- *Synchronization overheads*: Time wasted for waiting to enter critical regions.

Overheads of OpenMP directives

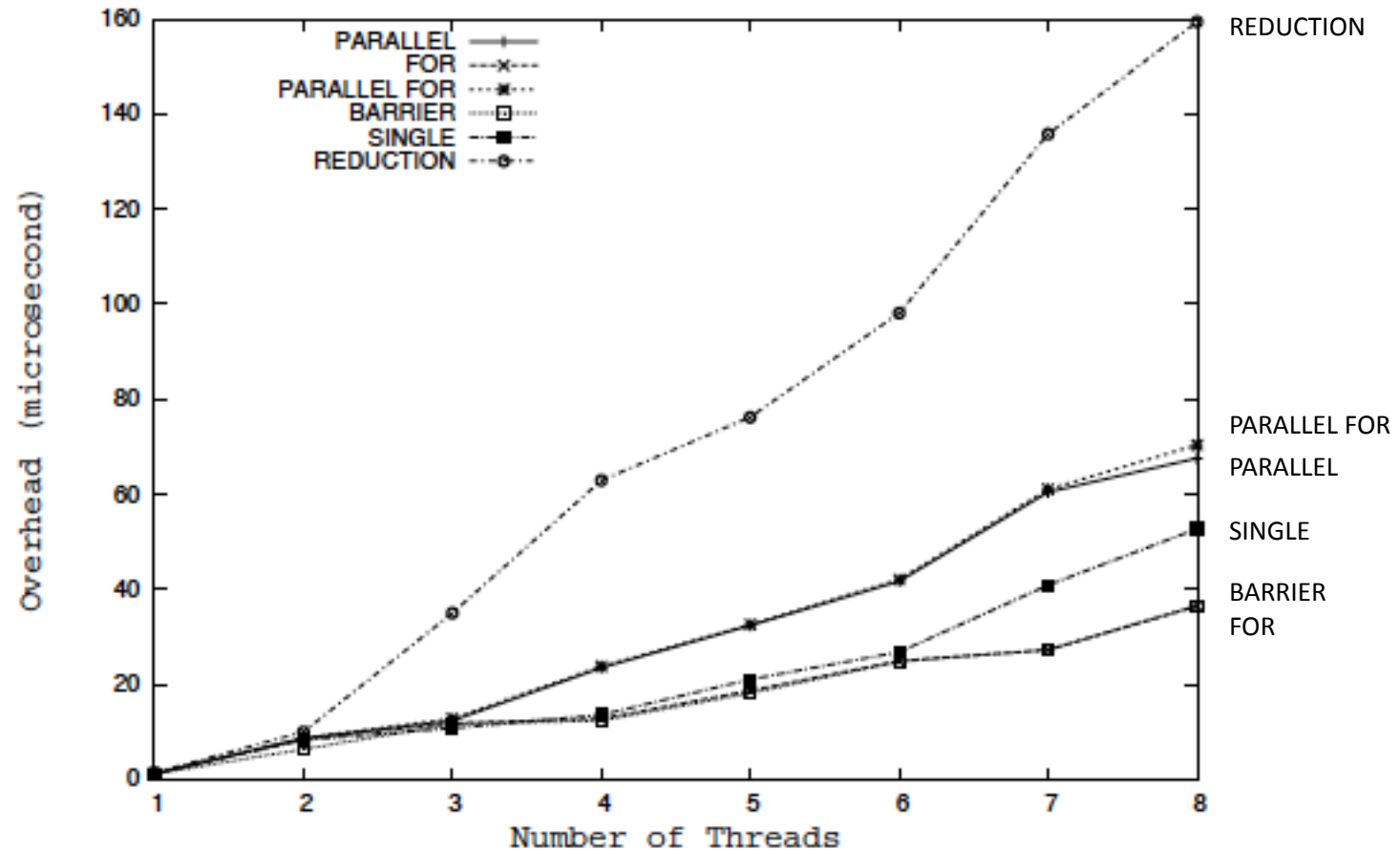
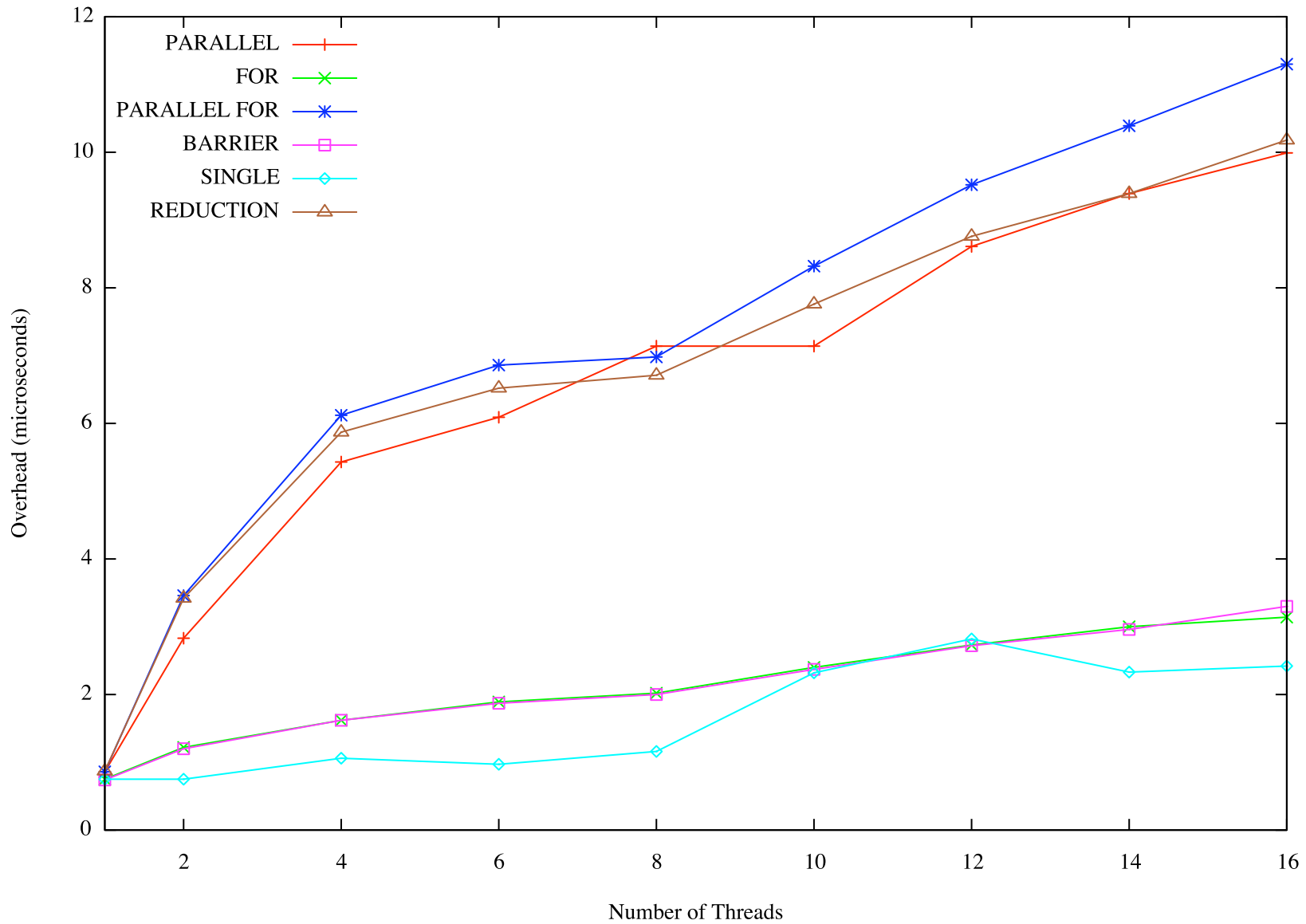


Figure 5.18: Overheads of some OpenMP directives – The overhead of several common directives and constructs is given in microseconds.

Overheads of OpenMP directives on alvin (gcc)



Overhead of OpenMP scheduling

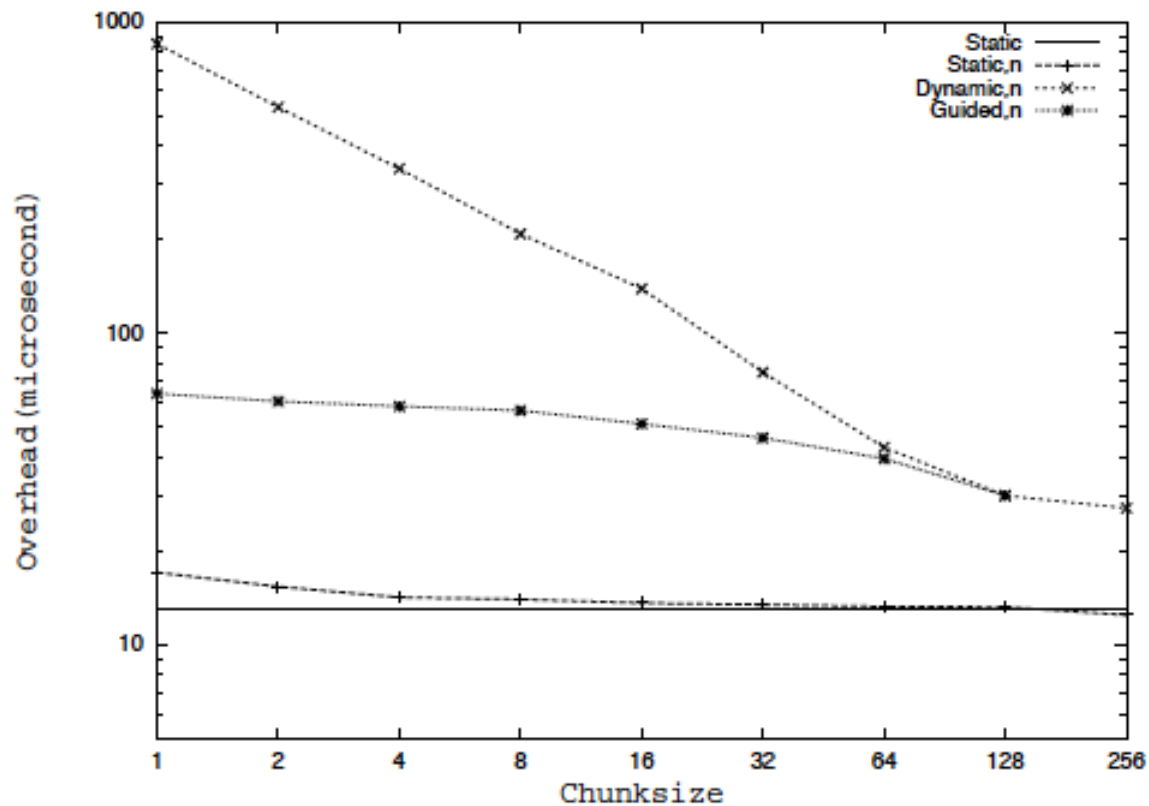
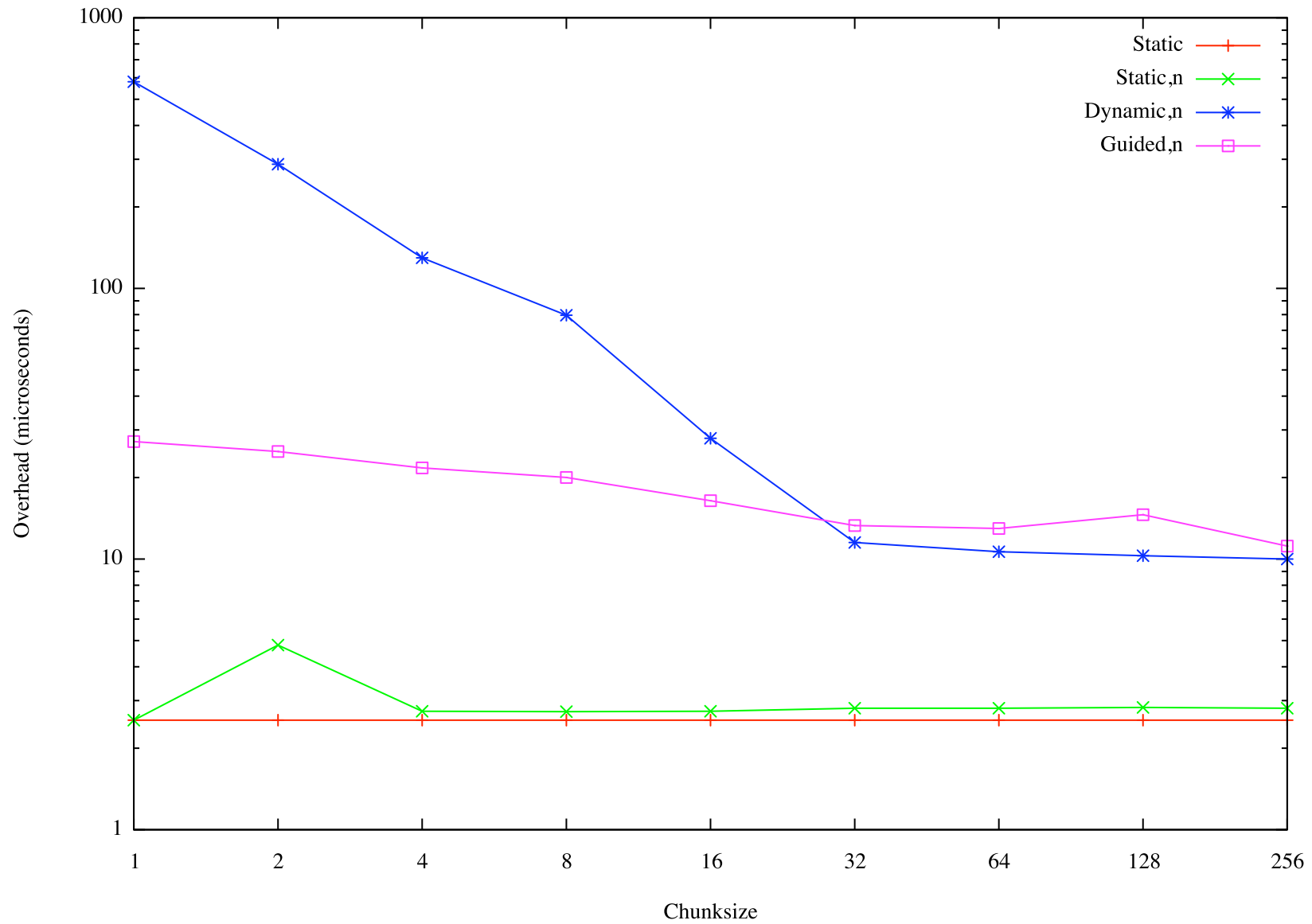


Figure 5.19: **Overhead of OpenMP scheduling** – The overheads for the different kinds of loop schedules are shown. Note that the scale to the left is logarithmic.

Overhead of OpenMP scheduling on alvin (gcc)



Best practices



Optimize barrier use



```
#pragma omp parallel default(none) \  
    shared(n,a,b,c,d,sum) private(i)  
{  
    #pragma omp for nowait  
    for (i=0; i<n; i++)  
        a[i] += b[i];  
  
    #pragma omp for nowait  
    for (i=0; i<n; i++)  
        c[i] += d[i];  
  
    #pragma omp barrier  
  
    #pragma omp for nowait reduction(+:sum)  
    for (i=0; i<n; i++)  
        sum += a[i] + c[i];  
} /*-- End of parallel region --*/
```

Figure 5.21: A reduced number of barriers – Before reading the values of vectors a and b all updates on these vectors have to be completed. The barrier ensures this.

Avoid the ordered construct



The ordered construct is expensive.

The construct can often be avoided. It might be better to perform I/O outside the parallel loop.

Avoid the critical region construct

```
#pragma omp parallel shared(a,b) private(c,d)
{
    .....
    #pragma omp critical
    {
        a += 2 * c;
        c = d * d;
    }
} /*-- End of parallel region --*/
```

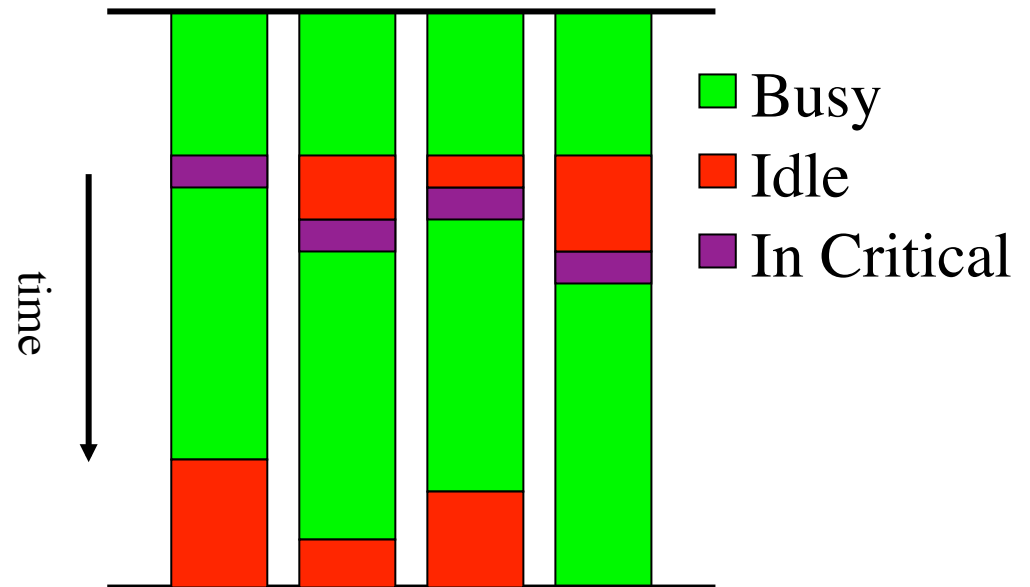
Figure 5.22: **A critical region** – Without the critical region, the first statement here leads to a data race. The second statement however involves private data only and unnecessarily increases the time taken to execute this construct. To improve performance it should be removed from the critical region.

If at all possible, an atomic update is to be preferred.

Avoid large critical regions

Lost time waiting for locks

```
#pragma omp parallel  
{  
    #pragma omp critical  
    {  
        ...  
    }  
    ...  
}
```



Maximize parallel regions

```
#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop 1 --*/
}

#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop 2 --*/
}
.....

#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop N --*/
}
```

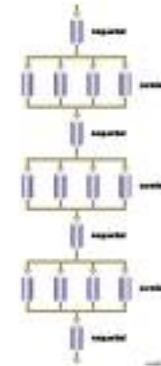


Figure 5.23: Multiple combined parallel work-sharing loops – Each parallelized loop adds to the parallel overhead and has an implied barrier that cannot be omitted.

Maximize parallel regions

```
#pragma omp parallel
{
    #pragma omp for /*-- Work-sharing loop 1 --*/
    { ..... }

    #pragma omp for /*-- Work-sharing loop 2 --*/
    { ..... }

    .....

    #pragma omp for /*-- Work-sharing loop N --*/
    { ..... }
}
```

Figure 5.24: **Single parallel region enclosing all work-sharing for loops** – The cost of the parallel region is amortized over the various work-sharing loops.

Large parallel regions offer more opportunities for using data in cache and provide a bigger context for compiler optimizations.

Avoid parallel regions in inner loops

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    #pragma omp parallel for
    for (k=0; k<n; k++)
      { ..... }
```

Figure 5.25: **Parallel region embedded in a loop nest** – The overheads of the parallel region are incurred n^2 times.

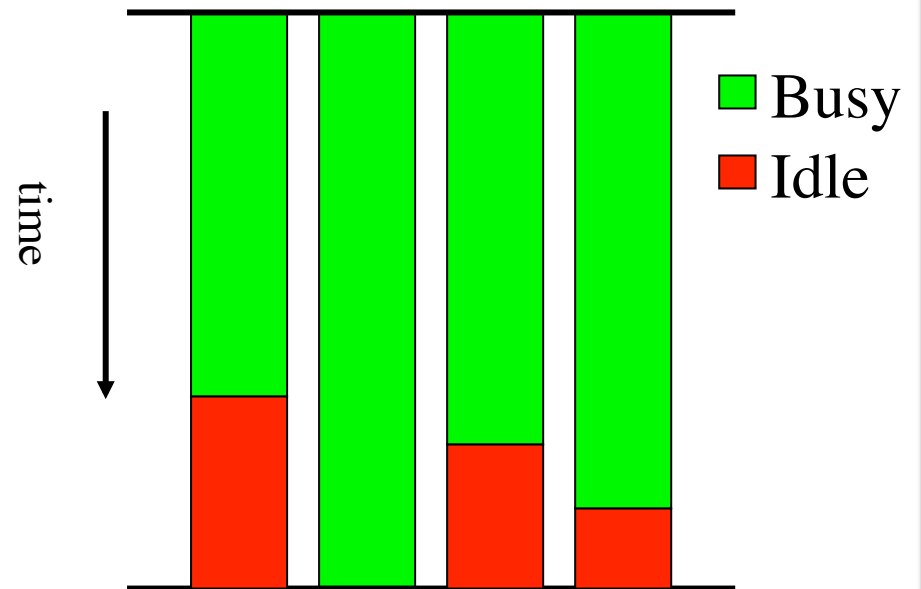
```
#pragma omp parallel
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      #pragma omp for
      for (k=0; k<n; k++)
        { ..... }
```

Figure 5.26: **Parallel region moved outside of the loop nest** – The parallel construct overheads are minimized.

Load imbalance

Unequal work loads lead to idle threads and wasted time.

```
#pragma omp parallel  
{  
  #pragma omp for  
  for ( ; ; ) {  
  }  
}
```



Load balancing



- Load balancing is an important aspect of performance
- For regular expressions (e.g. vector addition), load balancing is not an issue
- For less regular workloads, care needs to be taken in distributing the work over the threads
- Examples of irregular workloads:
 - multiplication of triangular matrices
 - parallel searches in a linked list
- For these irregular situations, the schedule clause supports various iteration scheduling algorithms

Address poor load balancing

```
for (i=0; i<N; i++) {  
    ReadFromFile(i,...);  
    for (j=0; j<ProcessingNum; j++ )  
        ProcessData(); /* lots of work here */  
    WriteResultsToFile(i);  
}
```

Figure 5.27: **Pipelined processing** – This code reads data in chunks, processes each chunk and writes the results to disk before dealing with the next chunk.

cont'd on next page

```

#pragma omp parallel
{
    /* preload data to be used in first iteration of the i-loop */
    #pragma omp single
        {ReadFromFile(0,...);}

    for (i=0; i<N; i++) {

        /* preload data for next iteration of the i-loop */
        #pragma omp single nowait
            {ReadFromFile(i+1...);}

        #pragma omp for schedule(dynamic)
            for (j=0; j<ProcessingNum; j++)
                ProcessChunkOfData(); /* here is the work */
        /* there is a barrier at the end of this loop */

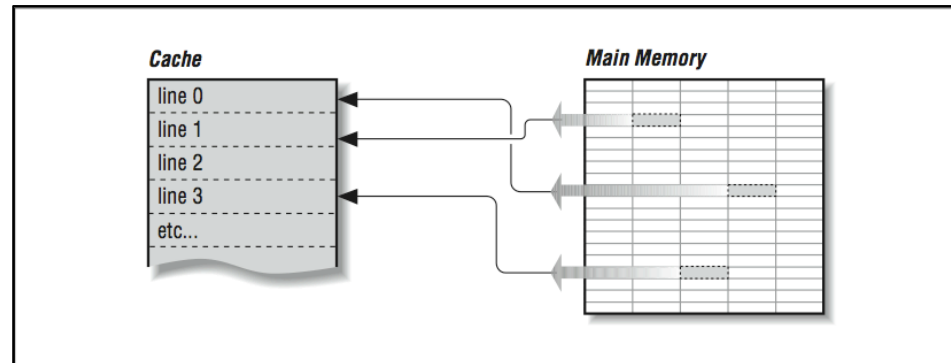
        #pragma omp single nowait
            {WriteResultsToFile(i);}

    } /* threads immediately move on to next iteration of i-loop */
} /* one parallel region encloses all the work */

```

Figure 5.28: **Parallelized pipelined processing** – This code uses a dynamic work-sharing schedule to overlap I/O and computation.

False sharing



The state bit of a cache line does not keep track of the cache line state on a byte basis, but at the line level instead.

Thus, if **independent** data items happen to reside on the same cache line (cache block), each update will cause the cache line to “ping-pong” between the threads.

This is called *false sharing*.

False sharing



False sharing is likely to significantly impact performance under the following conditions:

1. Shared data are modified by multiple threads.
2. The access pattern is such that multiple threads modify the same cache line(s).
3. These modification occur in rapid succession.

False sharing example

```
#pragma omp parallel for shared(Nthreads,a) schedule(static,1)
  for (int i=0; i<Nthreads; i++)
    a[i] += i;
```

Figure 5.29: **Example of false sharing** – `Nthreads` equals the number of threads executing the `for`-loop. The chunk size of 1 causes each thread to update one element of `a`, resulting in false sharing.

False sharing

Array elements are contiguous in memory and hence share cache lines.

Result: False sharing may lead to poor scaling

Solutions:

- When updates to an array are frequent, work with local copies of the array instead of an array indexed by the thread ID.
- Pad arrays so elements you use are on distinct cache lines.

Array padding

```
int a[Nthreads];  
#pragma omp parallel for shared(Nthreads,a) schedule(static,1)  
    for (int i=0; i<Nthreads; i++)  
        a[i] += i;
```

```
int a[Nthreads][cache_line_size];  
#pragma omp parallel for shared(Nthreads,a) schedule(static,1)  
    for (int i=0; i<Nthreads; i++)  
        a[i][0] += i;
```

Case study: Matrix times vector product

```
1 void mxv(int m, int n, double * restrict a,
2         double * restrict b, double * restrict c)
3 {
4     int i, j;
5
6     #pragma omp parallel for default(none) \
7         shared(m,n,a,b,c) private(i,j)
8     for (i=0; i<m; i++)
9     {
10        a[i] = b[i*n]*c[0];
11        for (j=1; j<n; j++)
12            a[i] += b[i*n+j]*c[j];
13    } /*-- End of parallel for --*/
14 }
```

Figure 5.31: OpenMP version of the matrix times vector product in C – The result vector is initialized to the first computed result here.

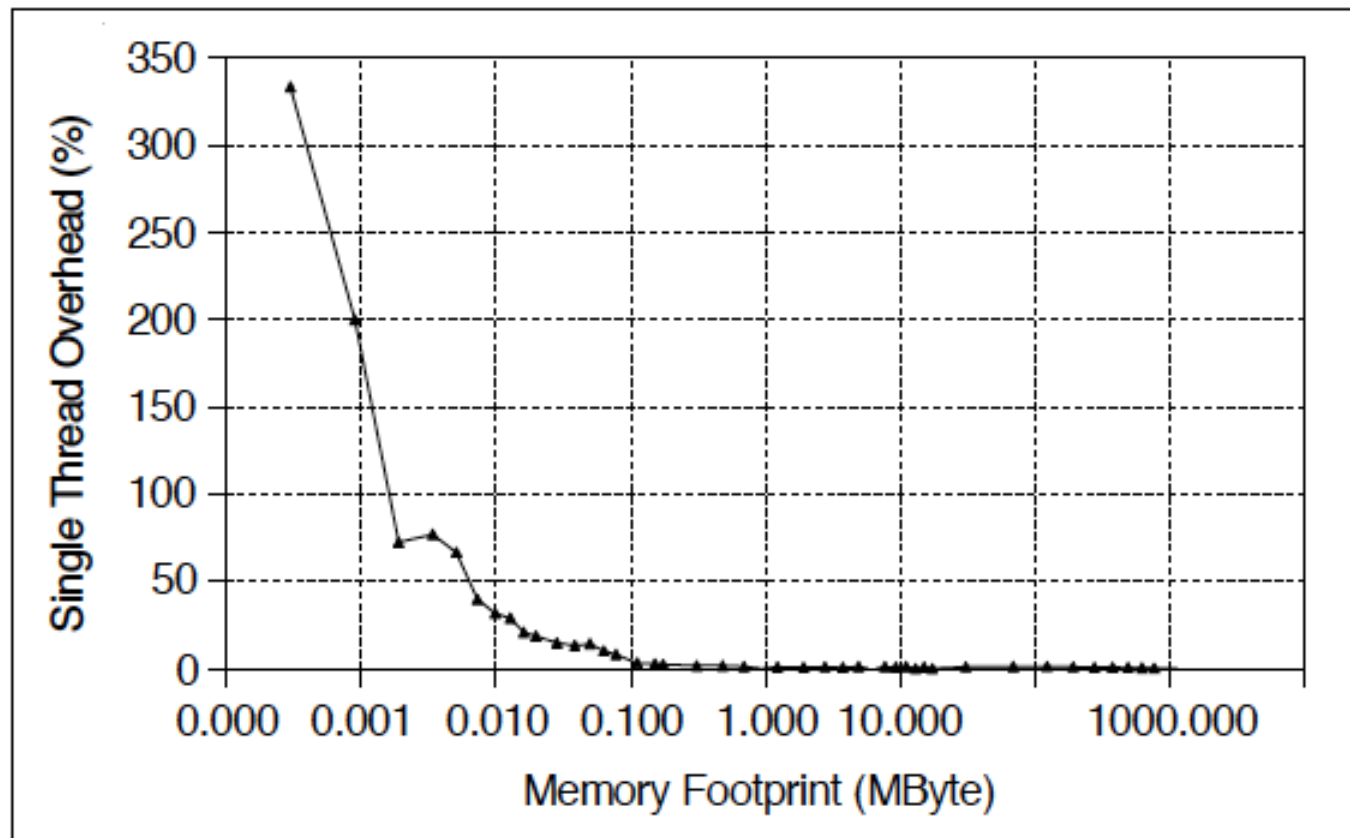


Figure 5.33: **Single-thread overheads for the matrix times vector product in C** – Formula (5.3) has been used to compute the overhead for a wide range of matrix sizes. For a matrix of size 200 x 200 the overhead is 2%. It is less for larger matrices.

$$Overhead_{single\ thread} = 100 * \left(\frac{ElapsedTime(OpenMP_{single\ thread})}{ElapsedTime(Sequential)} - 1 \right) \% \quad (5.3)$$

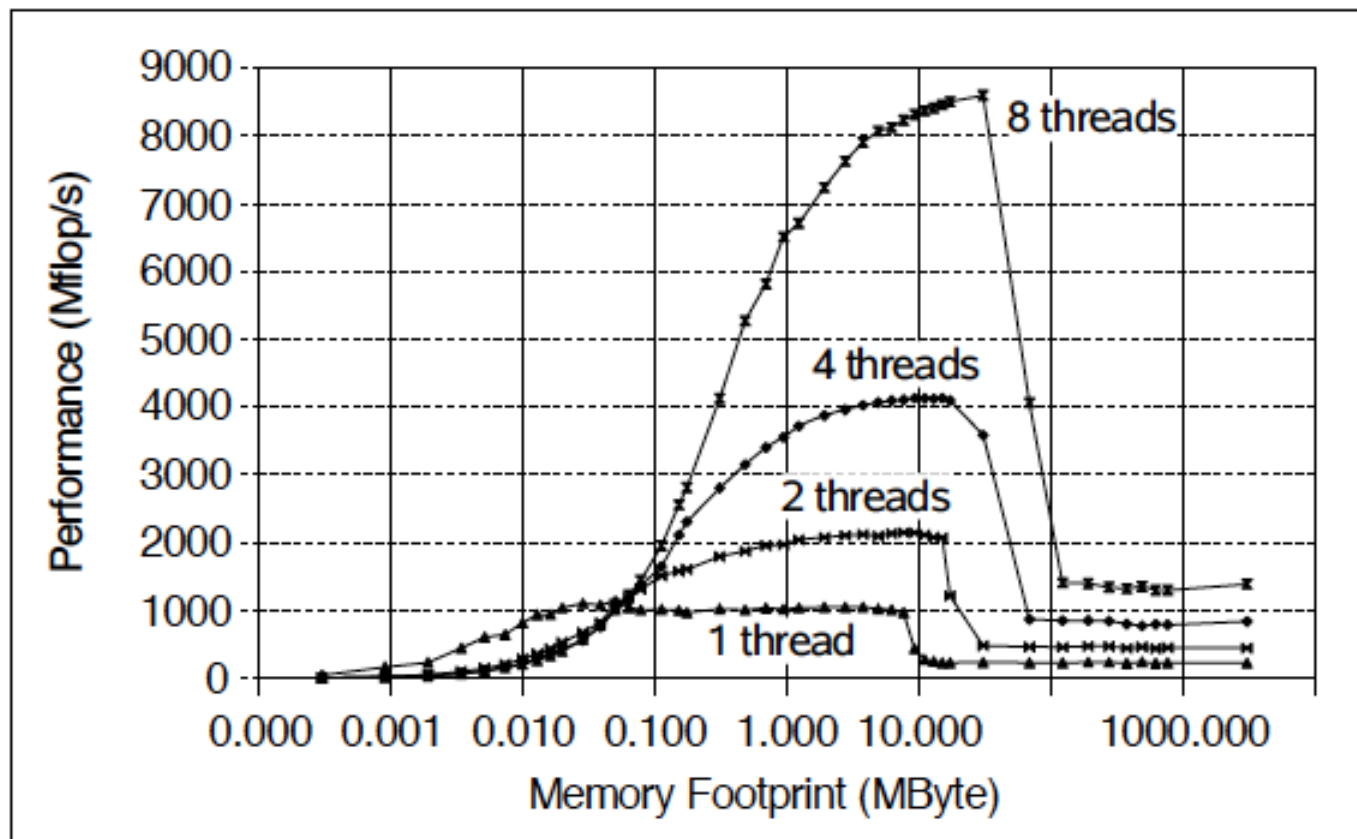


Figure 5.34: **OpenMP performance of the matrix times vector product in C** – If the memory footprint is less than 0.05 MByte, the single thread performance is higher than the performance for multiple threads. For a certain range of problem sizes, a superlinear speedup is realized. For problem sizes exceeding this range, the performance curves follow Amdahl's law.

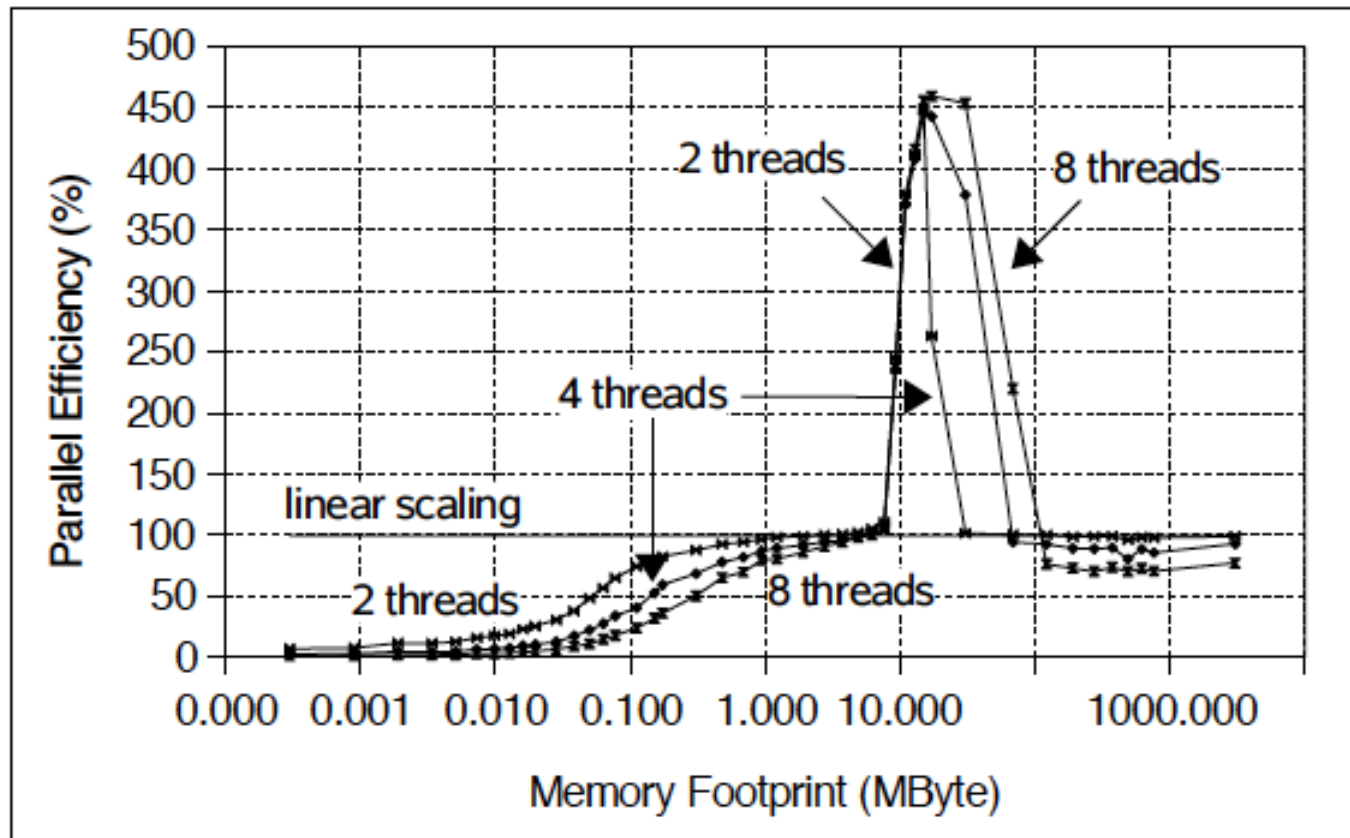


Figure 5.35: **Parallel efficiency of the matrix times vector product in C** – Several interesting effects are observed. Up to a specific memory footprint, the efficiency increases as the matrix gets larger. A superlinear speedup is even observed. The higher the number of threads, the longer this lasts. At a certain point, however, the efficiency drops, basically following Amdahl's law.

```

1  #include "globals.h"
2
3  void mxv(int m, int n, double * restrict a,
4           double * restrict b, double * restrict c)
5  {
6      int i, j;
7
8      #pragma omp parallel for if (m > threshold_omp) \
9          default(none) \
10         shared(m,n,a,b,c) private(i,j)
11     for (i=0; i<m; i++)
12     {
13         a[i] = b[i*n]*c[0];
14         for (j=1; j<n; j++)
15             a[i] += b[i*n+j]*c[j];
16     } /*-- End of parallel for --*/
17 }

```

Figure 5.36: **Second OpenMP version of the matrix times vector product in C** – Compared to the source listed in Figure 5.31, the `if`-clause has been included. The `threshold_omp` variable can be used to avoid a performance degradation for small matrices. If the clause evaluates to false, only one thread executes the code.

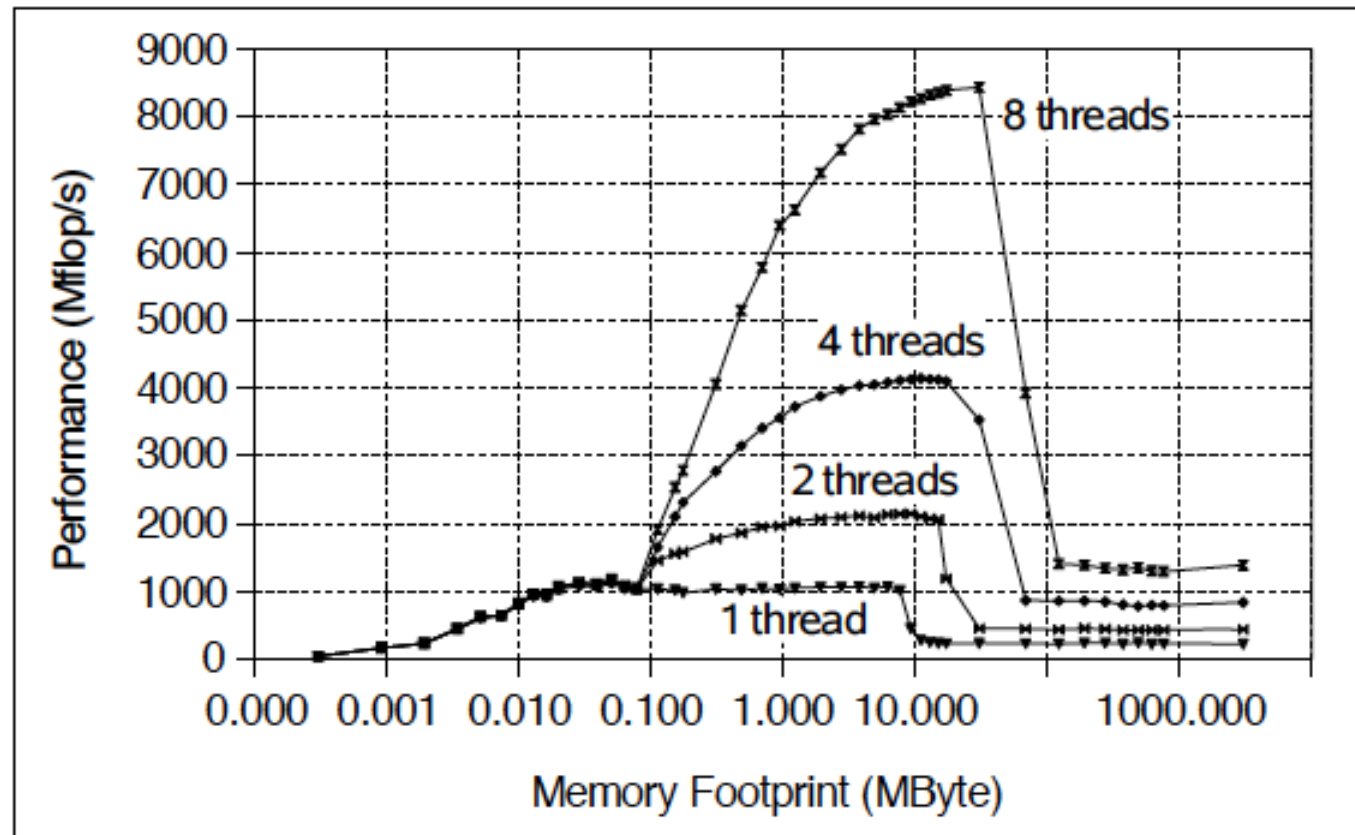


Figure 5.37: **OpenMP** performance of the matrix times vector product in **C** – The performance is now either equal to or higher than single-thread performance.

Task Parallelism in OpenMP 3.0



What are tasks?

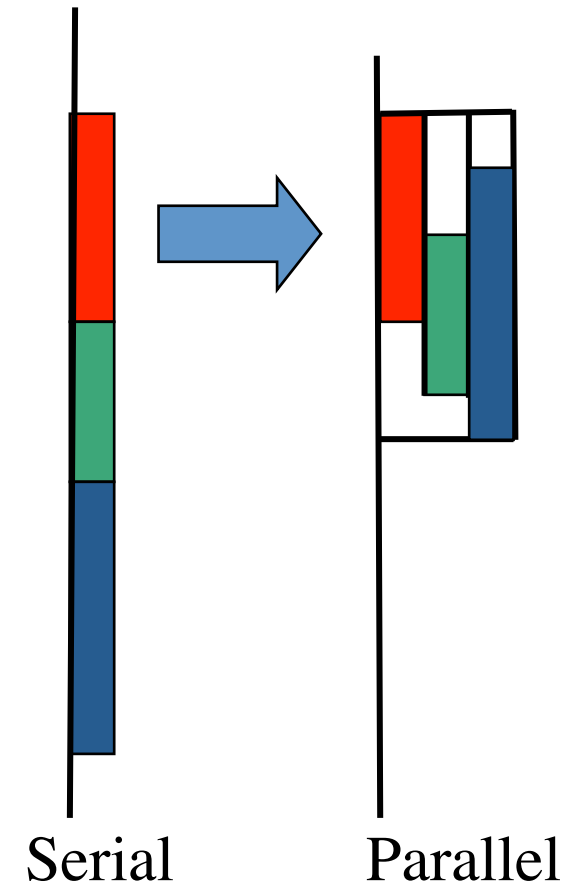
Tasks are independent units of work

Threads are assigned to perform the work of each task

- Tasks may be deferred
- Tasks may be executed immediately

The runtime system decides which of the above

- Tasks are composed of:
 - **code** to execute
 - **data** environment (it *own* its data)
 - **internal control variables**



Tasks in OpenMP

OpenMP has always had tasks, but they were not called that.

- A thread encountering a parallel construct packages up a set of *implicit* tasks, one per thread.
- A team of threads is created.
- Each thread is assigned to one of the tasks (and *tied* to it).
- Barrier holds master thread until all implicit tasks are finished.

OpenMP 3.0 adds a way to create a task explicitly for the team to execute.

The task construct

```
#pragma omp task [clause [[, clause] ... ]  
structured block
```

Each encountering thread creates a new task.

- Code and data are being packaged up
- Tasks can be nested

An OpenMP `barrier` (implicit or explicit):

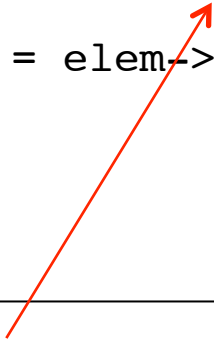
All tasks created by any thread of the current team are guaranteed to be completed at barrier exit.

Task barrier (`taskwait`):

Encountering thread suspends until all child tasks it has generated are complete.

Simple example of using tasks for pointer chasing

```
void process_list(elem_t *elem) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            while (elem != NULL) {
                #pragma omp task
                {
                    process(elem);
                }
                elem = elem->next;
            }
        }
    }
}
```



elem is `firstprivate` by default

Simple example of using tasks in a recursive algorithm

```
int fib(int n) {  
    int i, j;  
    if (n < 2)  
        return n;  
    #pragma omp task shared(i)  
    i = fib(n - 1);  
    #pragma omp task shared(j)  
    j = fib(n - 2);  
    #pragma omp taskwait  
    return i + j;  
}
```

```
int main() {  
    int n = 10;  
    #pragma omp parallel  
    #pragma omp single  
    printf("fib(%d) = %d\n",  
        n, fib(n));  
}
```

Computation of Fibonacci numbers
1,1,2,3,5,8,13,21,34,55,89,144,...

Using tasks for tree traversal

```
struct node {
    struct node *left, *right;
};

void traverse(struct node *p, int postorder) {
    if (p->left != NULL)
        #pragma omp task
        traverse(p->left, postorder);
    if (p->right != NULL)
        #pragma omp task
        traverse(p->right, postorder);
    if (postorder) {
        #pragma omp taskwait
    }
    process(p);
}
```

Task switching

Certain constructs have suspend/resume points at defined positions within them.

When a thread encounters a suspend point, it is allowed to suspend the current task and resume another. It can then return to the original task and resume it.

A *tied* task must be resumed by the same task that suspended it.

Tasks are tied by default. A task can be specified to be untied using

```
#pragma omp task untied
```

Collapsing of loops

The collapse clause (in OpenMP 3.0) handles perfectly nested multi-dimensional loops.

```
#pragma omp for collapse(2)
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    for (k = 0; k < K; k++)
      foo(i, j, k);
```

Iteration space from i-loop and j-loop is collapsed into a single one, if the two loops are perfectly nested and form a rectangular iteration space.

Removal of dependencies

Serial version containing anti dependency

```
for (i = 0; i < n; i++) {  
    x = (b[i] + c[i]) / 2;  
    a[i] = a[i + 1] + x;  
}
```

Parallel version with dependencies removed

```
#pragma omp parallel for shared(a,a_copy)  
for (i = 0; i < n; i++)  
    a_copy[i] = a[i + 1];  
#pragma omp parallel for shared(a,a_copy) private(x)  
for (i = 0; i < n; i++) {  
    x = (b[i] + c[i]) / 2;  
    a[i] = a_copy[i] + x;  
}
```

Removal of dependencies

Serial version containing flow dependency

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

Parallel version with dependencies removed by **loop skewing**

```
b[1] = b[1] - a[0];  
#pragma omp parallel for shared(a,b,c)  
for (i = 1; i < n; i++) {  
    a[i] = a[i] + c[i];  
    b[i + 1] = b[i + 1] + a[i];  
}  
a[n - 1] = a[n - 1] + c[n - 1];
```

Automatic parallelization

Some compilers can insert OpenMP can optimize a program automatically. However, they must be conservative, and programs spread over several files create difficulties.

The Intel compilers support automatic parallelization. Example,

```
icc -o matmult -O3 -parallel -par-report3 matmult.c
```