Module 2
Shared-memory parallel programming with OpenMP

Slides are based on Chapters 4,5,6,7,8 of the following book:

Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Overview of Module 2

- An introduction to the fundamental variants of parallel computers
- Basic strategies for parallelization
- OpenMP programming for shared-memory systems

Introducing parallel computers

Shared-memory parallel computers

A *shared-memory parallel computer* has a number of CPUs (cores) that work on a shared physical address space.

Two varieties:

- *Uniform Memory Access* (UMA) systems have a "flat" memory model: latency and bandwidth are the same for all processors and all memory locations. (Typically, single multicore processor chips are "UMA machines".)
- *Cache-coherent Nonuniform Memory Access* (ccNUMA) systems have a physically distributed memory that is *logically shared*. The aggregated memory appears as one single address space. Memory access performance depends on the which CPU (core) accesses which parts of memory ("local" vs. "remote" memory access).

Note: Caches are not (completely) shared

A shared-memory system, no matter UMA or ccNUMA, has multiple CPU cores.

Although there is a single address space (shared memory), there are private caches, or partially shared caches, for the different CPU cores.

Therefore, copies of the same cache line **may** reside in several local caches.

Cache coherence

Problematic situations can arise when the same cache line resides in several caches:

- If the cache line in one of the caches is modified, the other caches' contents are *outdated* (thus invalid).
- If different parts of the same cache line are modified by different processors in their respective local caches $\rightarrow$ no one has the correct cache line anymore.

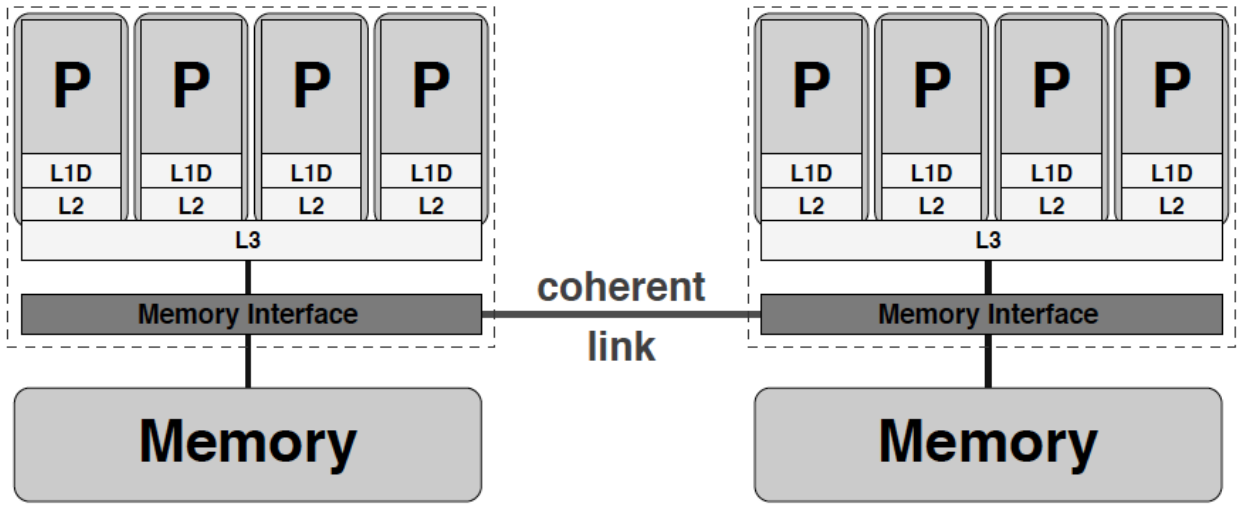
Cache coherence protocols (supported in hardware) guarantee *consistency* between cached data and data in the shared memory at all times.

ccNUMA for scalable memory bandwidth

- A *locality domain* (LD) is a set of processor cores together with locally connected memory. This "local" memory can be accessed by the set of processor cores in the most efficient way, without resorting to a network of any kind.
- Each LD is a UMA building block.
- Multiple LDs are linked via a coherent interconnect, which can mediate cache-coherent memory accesses. (This mechanism is transparent for the programmer.)
- The whole ccNUMA system has a shared address space (memory), runs a single OS instance.

Example of ccNUMA

Figure 4.5: A ccNUMA system with two locality domains (one per socket) and eight cores.



Penalty for non-local transfers

The *locality problem*: Non-local memory transfers (between LDs) are more costly than local transfers (within a LD).

The *contention problem*: If two processors from different LDs access memory in one LD, they will fight for the same memory bandwidth.

Both problems can be "solved" (alleviated) by carefully observing the data access patterns of an application and restricting data access of each processor (mostly) to its own LD, through proper programming.

Typical modern distributed-memory systems

A cluster of shared-memory "*compute nodes*", interconnected via a *communication network*.

Each node comprises at least one network interface (NI) that mediates the connection to the communication network.

A serial process runs on each CPU (core). Between the nodes, processes can communicate by means of the network.

The layout and speed of the network has a considerable impact on application performance.

Hierarchical hybrid systems

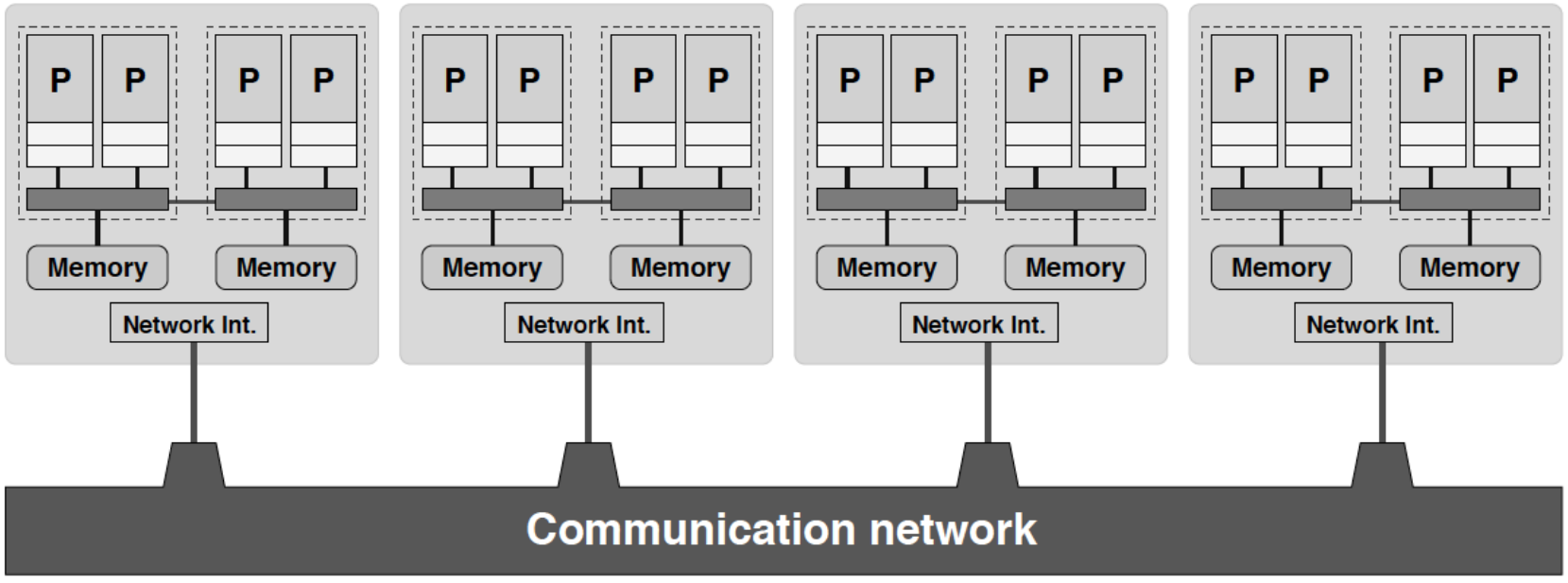


Figure 4.8: Typical hybrid system with shared-memory nodes (ccNUMA type). Two-socket building blocks represent the price vs. performance "sweet spot" and are thus found in many commodity clusters.

Parallelization

Parallelization, why?

- We want to solve the problems faster, but the speed of a single CPU core has "saturated".
- We want to solve larger problems, but the main memory available on a single system is not large enough.

So, we need to identify parallelism in a given computational problem, so that parallel programming can produce a parallel implementation that can efficiently use many processor cores, on a shared- or distributed-memory system.

Identifying parallelism

The first step is to identify the parallelism inherent in the algorithm at hand—how can *multiple* compute elements collaborate to solve the computational problem?

Different variants of parallelism induce different methods of parallelization.

Data parallelism

Most problems in scientific computing involve processing of large quantities of data stored on a computer.

If this can be performed in parallel by multiple processors concurrently working on different parts of the data—**data parallelism**.

This is the dominant parallelization concept in scientific computing, also goes under the name *SPMD* (single program multiple data).

Example: medium-grained loop parallelism

Processing of array data by loops or loop nests is a central component in most scientific codes.

When computations performed on the individual array elements are independent of each other—typical candidates for parallel execution by multiple "workers" with help of shared memory. (Also possible on distributed-memory systems after appropriate data partitioning.)

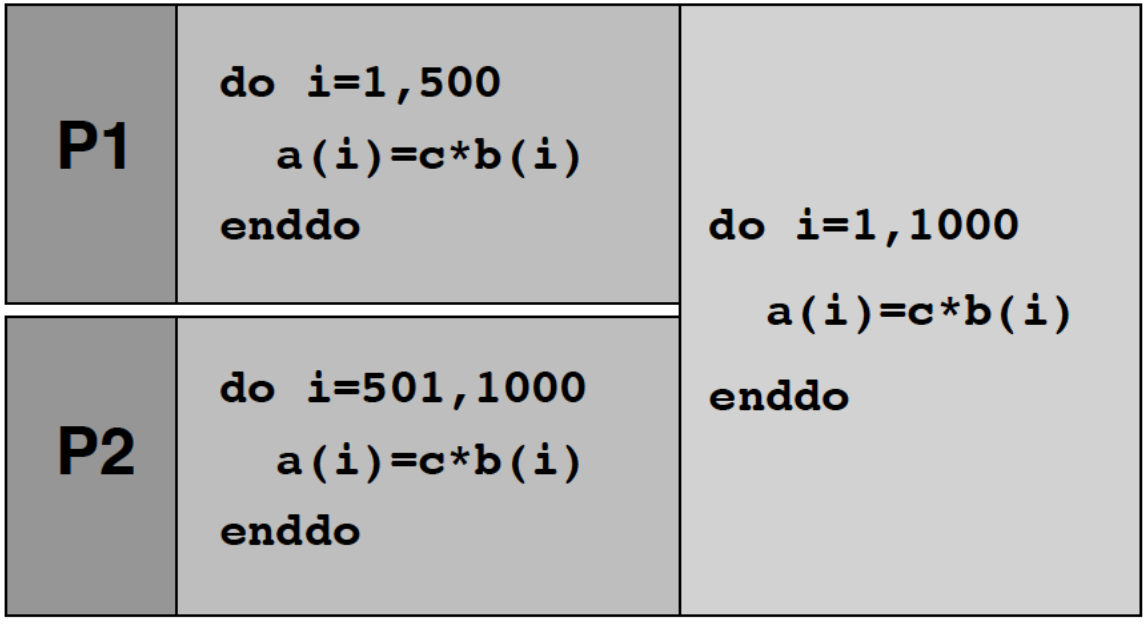


Figure 5.1: An example for medium-grained parallelism: The iterations of a loop are distributed to two processors P1 and P2 (in shared memory) for concurrent execution.

Example: parallelizing matrix-matrix multiplication

Multiplying two matrices A and B to yield matrix C. The output matrix C can for example be partitioned into four blocks (where each block is a sub-matrix):

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$\text{Process 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$\text{Process 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

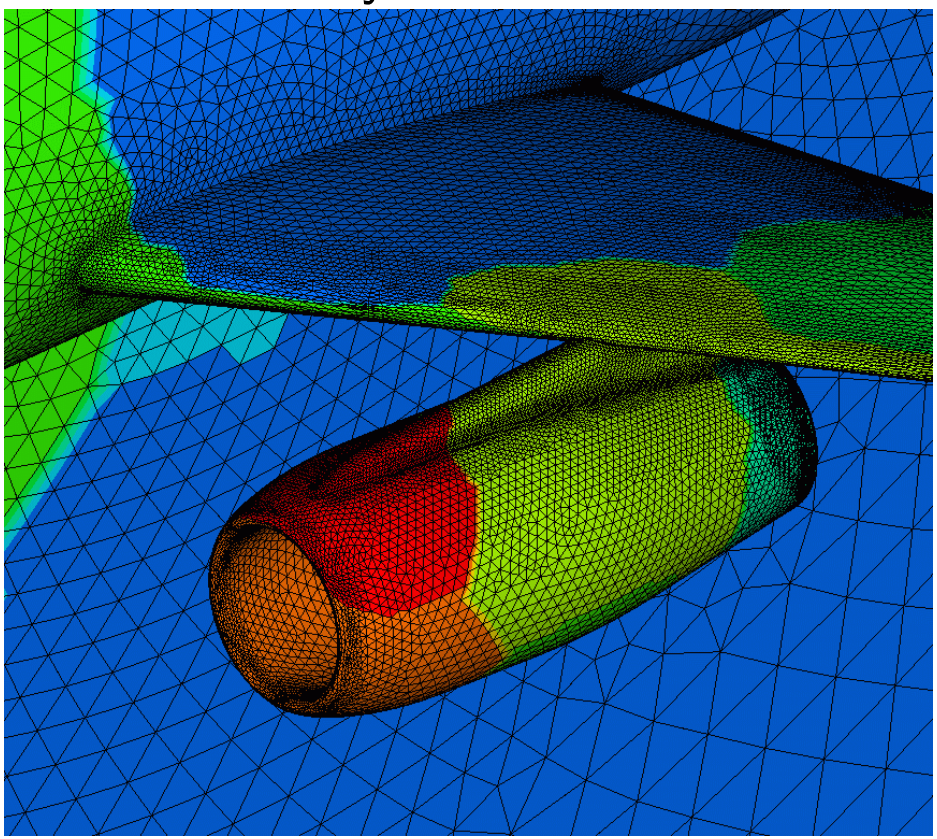
$$\text{Process 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$\text{Process 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Example: coarse-grained parallelism by domain decomposition

In case of a computational domain represented as a grid, a straightforward way to distribute the computation among "workers" is to assign a part of the grid to each worker—*domain distribution*.

Works on both shared-memory and distributed-memory computers.



https://fun3d.larc.nasa.gov/example-55.html

More about domain decomposition

Unless the computational grid is regular, domain decomposition can be a non-trivial task:

- Load balance—all "workers" get roughly the same amount of computation;
- Communication overhead—must be kept low;

Impact of decomposition on communication overhead

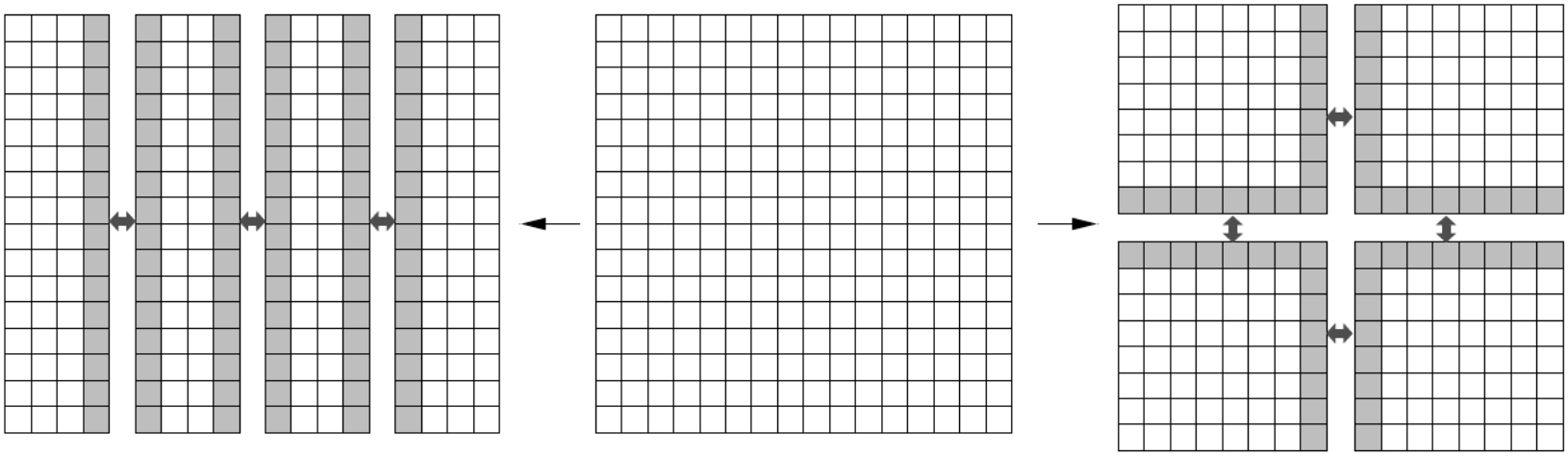
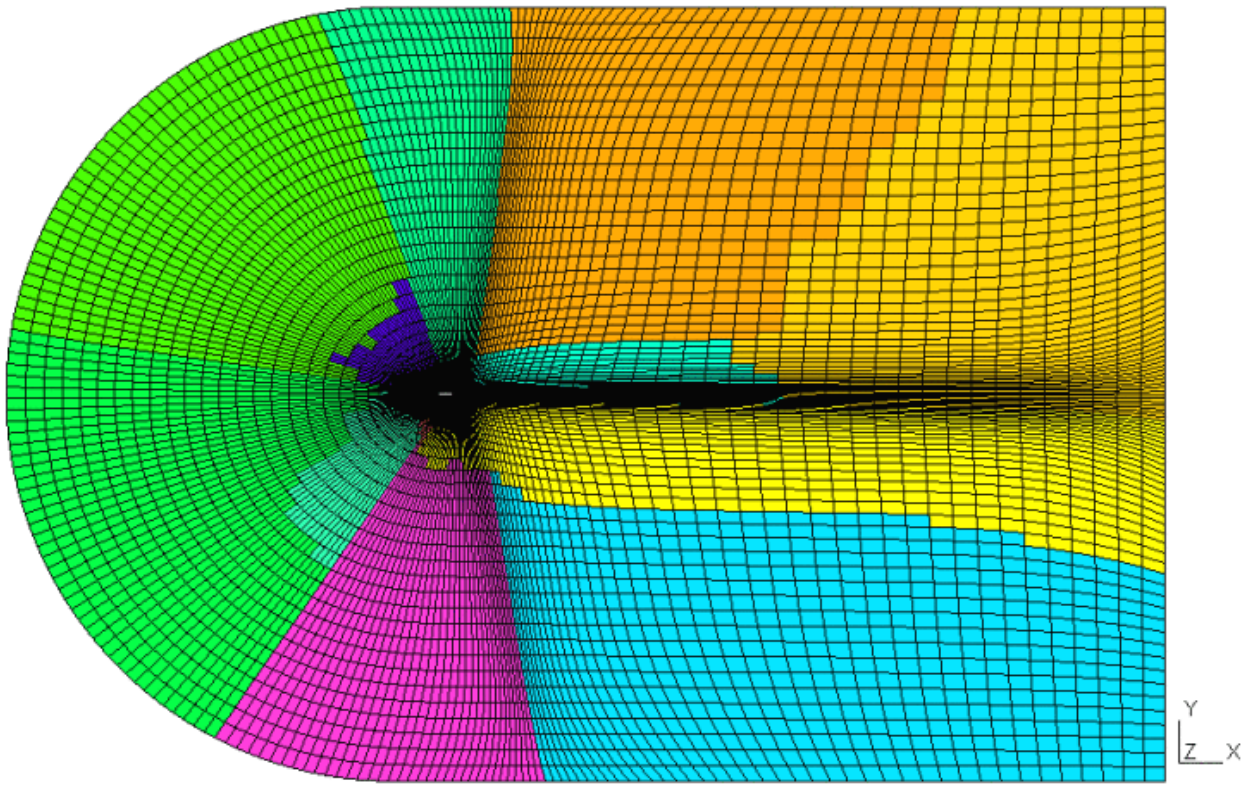


Figure 5.3: Domain decomposition of a two-dimensional Jacobi solver, which requires next-neighbor interactions. Cutting into stripes (left) is simple but incurs more communication than optimal decomposition (right). Shaded cells participate in network communication.

Example of general decomposition



https://nutscfd.wordpress.com/2017/03/06/mesh-partitioning-using-parmetis/

Parallel scalability

The *ideal* goal: If a problem takes time T to be solved by one worker, we expect the solution time by using N identical workers to be T/N—a perfect **speedup** of N.

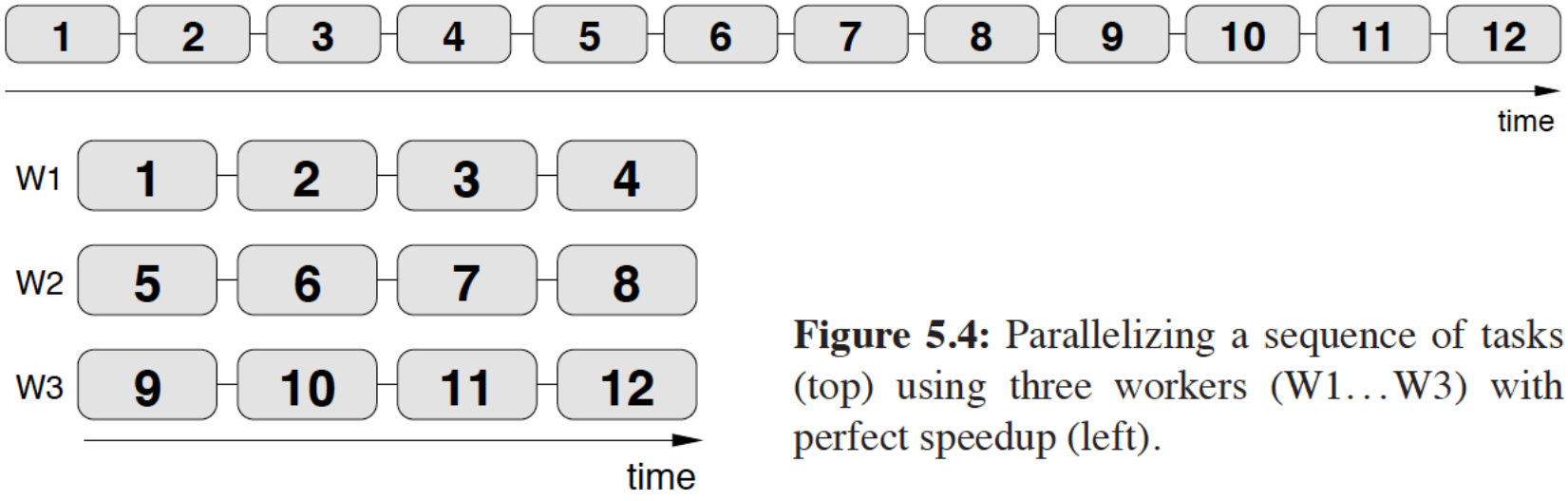


Figure 5.4: Parallelizing a sequence of tasks (top) using three workers (W1...W3) with perfect speedup (left).

However, perfect speedup is often not achievable in reality, why?

Factors that limit parallel execution

Reasons for non-perfect speedup:

- Not all workers might execute their tasks equally fast, because the problem was not (or could not be) partitioned into equal pieces—load imbalance;
- There might be shared resources which can only used by one worker at a time—serialization;
- New tasks may arise due to parallelization, such as communication between workers—overhead.

Speedup

How to calculate speedup?

$$\text{application speedup} = \frac{\text{serial runtime}}{\text{parallel runtime}}$$

or equivalently

$$\text{application speedup} = \frac{\text{parallel performance}}{\text{serial performance}}$$

where “performance” is defined as “work over time”.

Amdahl's law

For a fixed problem size $s + p = 1$, the application speedup ("scalability") is

$$S_{\mathrm{f}} = \frac{T_{\mathrm{f}}^{\mathrm{s}}}{T_{\mathrm{f}}^{\mathrm{p}}} = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{1-s}{N}}$$

This is "Amdahl's law"—maximum speedup is $1/s$ when $N \to \infty$.

Gustafson's law

The problem size is scaled with the number of workers N.

Recall that for $\alpha = 1$ we have $T_{\mathrm{v}}^{\mathrm{s}} = s + pN$ and $T_{\mathrm{v}}^{\mathrm{p}} = s + p$. Therefore the application speedup is

$$S_{\mathrm{v}} = \frac{T_{\mathrm{v}}^{\mathrm{s}}}{T_{\mathrm{v}}^{\mathrm{p}}} = \frac{s + pN}{s + p} = \frac{s + (1 - s)N}{1} = s + (1 - s)N$$

This is "Gustafson's law"—speedup can be arbitrarily large when $N \to \infty$.

Parallel efficiency

How effectively is the resource used by parallel program?

Parallel efficiency is defined as

$$\varepsilon = \frac{\text{speedup}}{N}$$

This will be a value between 0 and 100%.

Example: summing an array of values

```
sum=0.;
for (i=0; i<N; i++)
  sum += y[i];
```

Basic strategy of parallelization (each processor core with index `k`):

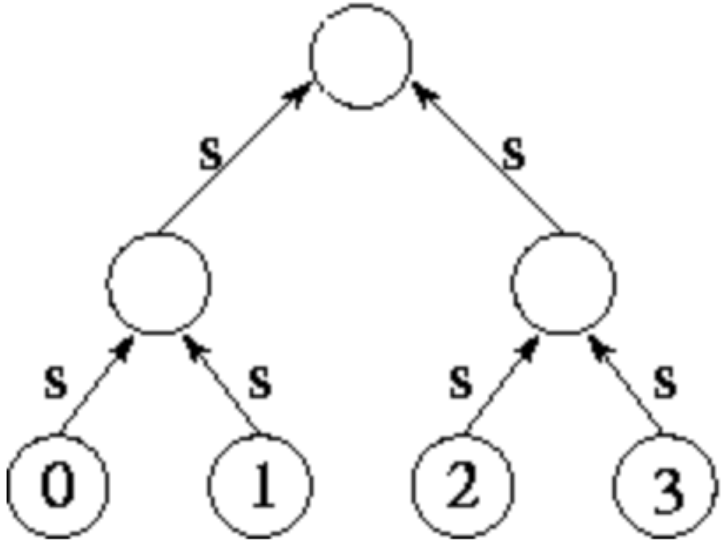
- Divide the entries of array `y` into as equal-sized chunks as possible (when `N` is not divisible by `P`). For each `k` compute
  `start_k = k*N/P, stop_k = (k+1)*N/P`
- Each processor core *independently* computes a partial sum as
  `y[start_k]+ y[start_k+1]+...+y[stop_k-1]`
- When all the P partial sums are computed, they are added up to produce the correct value of `sum`

How to sum up P values from P processor cores?

Approach 1: Pick a "master" processor core, and let the master add the P values together.

Downside of this approach: The master core can become a bottleneck if P is large.

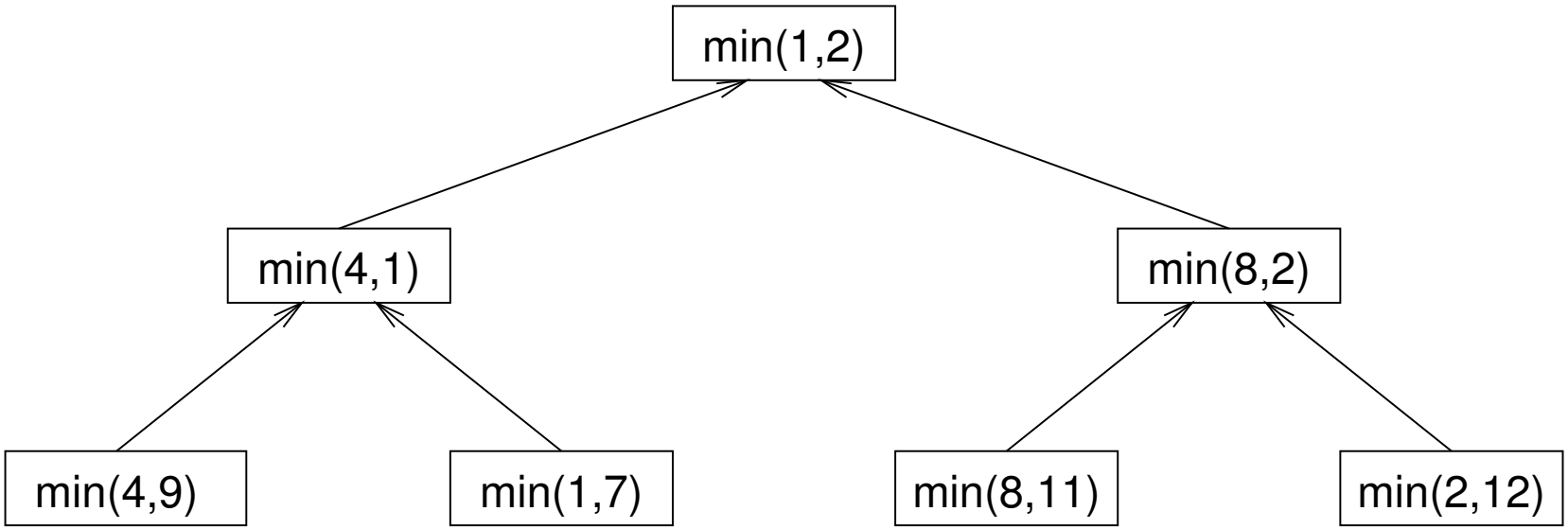
Approach 2: *reverse recursive decomposition*



The "bottom" tasks represent individual partial sums on the processor cores, the other tasks are pair-wise additions until `sum` is computed at the "top".

Another example of reverse recursive decomposition

Suppose we want to find the minimum value in the set $\{4, 9, 1, 7, 8, 11, 2, 12\}$.



OpenMP programming

Programming shared-memory systems, a quick overview

The applicable hardware context: **shared memory**

- All processors can directly access all data in a shared memory, no need for explicit communication between the processors
- OpenMP: A parallel programming standard for shared-memory parallel computers
  - A set of compiler directives (with additional clauses)
  - A small number of library functions
  - A few environment variables

Advantages of compiler directives

- An OpenMP compiler directive is a "parallelization hint" intended for a compatible compiler to automatically create parallel code, will be ignored by non-OpenMP compilers otherwise.

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

- It is **possible** to maintain a same code for both the serial implementation and the parallel OpenMP implementation.

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

- Allows incremental parallelization—simplifies coding effort—easier to debug

Threads in OpenMP

- The central execution entities in an OpenMP program are **threads**—lightweight processes.
- The OpenMP threads share a common address space and mutually access data.
- Spawning a thread is much less costly than forking a new process, because threads share everything except the instruction pointer, stack pointer and register state.
  - If wanted, each thread can have a few "private variables" (by means of the local stack pointer).

"Fork-join" model

- In any OpenMP program, a single thread, called **master thread**, runs immediately after startup.
- The master thread can spawn (also called **fork**) a number of additional threads when entering a so-called **parallel region**.
- Inside a parallel region, the master thread and the spawned threads execute instruction streams **concurrently**.
  - Each thread has a unique ID.
  - Different threads work on different parts of the shared data or carry out different tasks.
  - OpenMP has compiler directives for dividing the work among the threads.
- At the end of a parallel region, the threads are **joined**: all the threads are terminated but the master thread.
- There can be multiple parallel regions in an OpenMP program.

OpenMP's parallel execution model

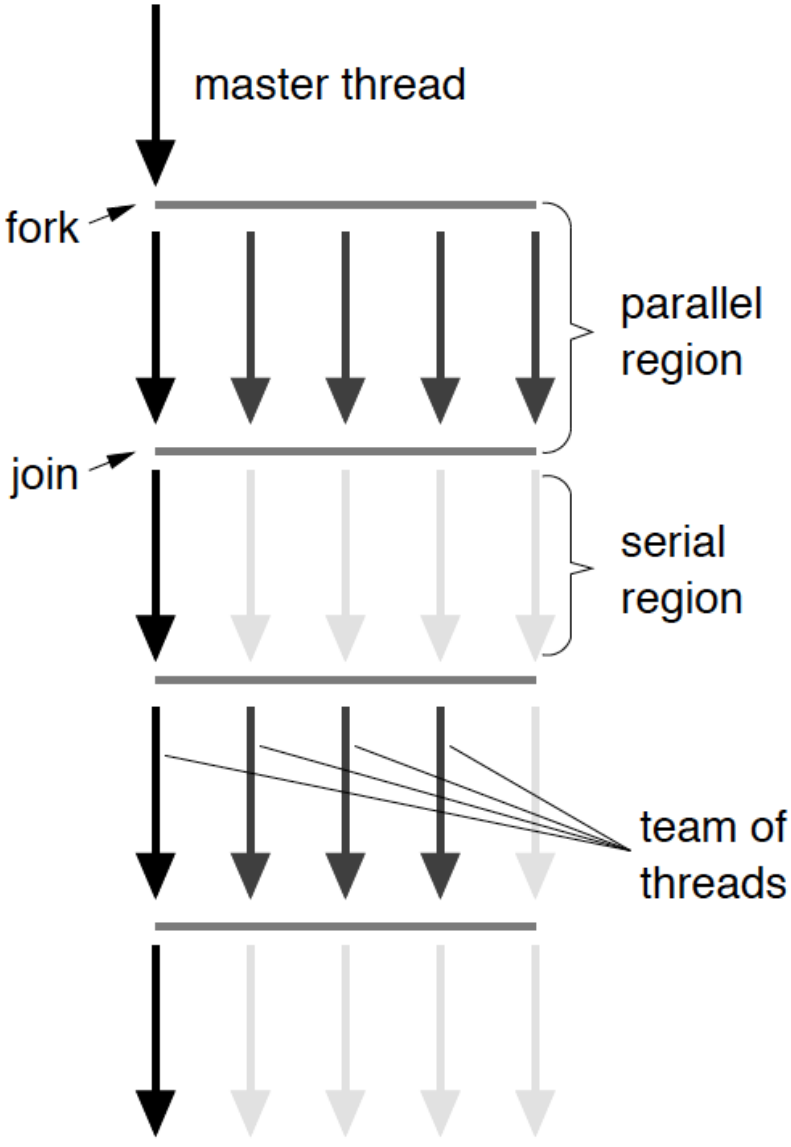


Figure 6.1: Model for OpenMP thread operations: The master thread "forks" team of threads, which work on shared memory in a parallel region. After the parallel region, the threads are "joined," i.e., terminated or put to sleep, until the next parallel region starts. The number of running threads may vary among parallel regions.

Hello world in OpenMP

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

int main (int nargs, char **args) {

#pragma omp parallel
{
  printf("Hello world!\n");
}

  return 0;
}
```

- Example of compilation: `gcc -fopenmp hello_world.c`
- Parallel execution: `./a.out`
- What do you get?

Control the number of threads at runtime

Example on a Linux system:

```
gcc -fopenmp hello_world.c
export OMP_NUM_THREADS=6
./a.out
export OMP_NUM_THREADS=8
./a.out
```

`OMP_NUM_THREADS` is an environment variable (understood by OpenMP)

It is also possible to hard-code the number of threads inside an OpenMP program:

```
#pragma omp parallel num_threads(6)
{
  // ......
}
```

But this approach is normally **not** recommended!

Hello world in OpenMP (a bit more interesting example)

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

int main (int nargs, char **args) {

  printf("I'm the master thread, I'm alone.\n");

#pragma omp parallel
{
  int num_threads, thread_id;
  num_threads = omp_get_num_threads();
  thread_id = omp_get_thread_num();
  printf("Hello world! I'm thread No.%d out of %d threads.\n",
         thread_id,num_threads);
}

  return 0;
}
```

“Manual” loop parallelization

Now, let’s try to “manually” parallelize a for-loop, that is, divide the iterations evenly among the threads.

Example:

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

Important observation: Assuming that the arrays `a`, `b` and `c` do not overlap in memory (that is, no aliasing), then the iterations of this for-loop are independent of each other, thus safe to be executed by multiple threads concurrently.

"Manual" loop parallelization (2)

How to divide the iterations `evenly` among the threads?

Given `num_threads` as the total number of threads, **one** way to divide N iterations for thread with ID `thread_id` is as follows:

```
int blen, bstart;

blen = N/num_threads;

if (thread_id < (N%num_threads)) {
  blen = blen + 1;
  bstart = blen * thread_id;
}
else {
  bstart = blen * thread_id + (N%num_threads);
}
```

Why is this a fair division?

“Manual” loop parallelization (3)

OpenMP coding

```
#pragma omp parallel
{
  int num_threads, thread_id;
  int blen, bstart, bend, i;

  num_threads = omp_get_num_threads();
  thread_id = omp_get_thread_num();

  blen = N/num_threads;
  if (thread_id < (N%num_threads)) {
    blen = blen + 1;
    bstart = blen * thread_id;
  }
  else {
    bstart = blen * thread_id + (N%num_threads);
  }
  bend = bstart + blen;

  for (i=bstart; i<bend; i++)
    a[i] = b[i] + c[i];

} // end of parallel region
```

Data scoping

Any variables that existed before a parallel region still exist inside the parallel region, and are by default **shared** between all threads.

Often it will be necessary for the threads to have some *private* variables.

- Each thread can either declare new local variables inside the parallel region, these variables are private "by birth";
- Or, each thread can "privatize" some of the shared variables that already existed before a parallel region (using the `private` clause):

```
int blen, bstart, bend;
#pragma omp parallel private(blen, bstart, bend)
{
  // ...
}
```

- Each "privatized" variable has one (*uninitialized*) instance per thread;
- The private variables' scope is until the end of the parallel region.

Actually, parallelizing a for-loop is easy in OpenMP

- Parallelizing for-loops is OpenMP's main work-sharing mechanism.
- OpenMP has several built-in strategies for dividing the iterations among the threads.
  - No need to manually calculate each thread's loop bounds

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

or simply

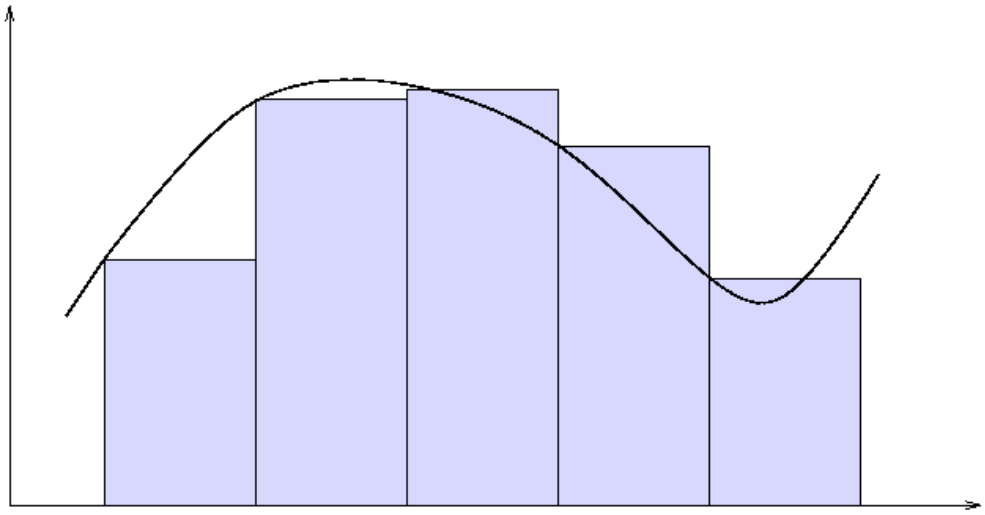
```
#pragma omp parallel for
  for (i=0; i<N; i++)
    a[i] = b[i] + c[i];
```

More remarks

- The loop can not contain `break`, `return`, `exit` statements.
- The `continue` statement is allowed.
- The index update has to be an increment (or decrement) by a fixed amount.
- The loop index variable is automatically private, and changes to it inside the loop are not allowed.

Numerical integration

How to numerically calculate $\int_{x_0}^{x_1} f(x)dx$?



$$w = \frac{x_1 - x_0}{N}$$

$$\int_{x_0}^{x_1} f(x)dx \approx w\left(f(x_0 + \frac{w}{2}) + f(x_0 + w + \frac{w}{2}) + \ldots \right.$$
$$\left. \ldots + f(x_0 + (N-1)w + \frac{w}{2})\right)$$

Numerical integration for calculating π

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

Serial implementation:

```
int N, i;
double w = 1.0/N, x, approx_pi;
double sum = 0.;

for (i=1; i<=N; i++) {
  x = w*(i-0.5);
  sum = sum + 4.0/(1.0+x*x);
}

approx_pi = w*sum;
```

A naive OpenMP implementation

```
int N, i;
double w = 1.0/N, x, approx_pi = 0.;
double sum = 0.;

#pragma omp parallel private(x) firstprivate(sum)
{

#pragma omp for
for (i=1; i<=N; i++) {
  x = w*(i-0.5);
  sum = sum + 4.0/(1.0+x*x);
}

#pragma omp critial
{
approx_pi = approx_pi + w*sum;
}

} // end of the parallel region
```

OpenMP critical regions

- Concurrent write accesses to a shared variable must be avoided by all means to circumvent **race conditions**.
- An OpenMP `critical` code block is executed by **one thread at a time**. This is one way to avoid race conditions. (There are other ways.)
- The variable `approx_pi` in the above example is a shared variable, to which all the threads will write. Thus, "protection" is provided by a `critical` code block.
- Use of the `critical` directive will incur overhead.
- Improper use of the `critical` directive may lead to **deadlock**.

Use of OpenMP's `reduction` clause

Actually, using `critical` to prevent concurrent writes to the variable `approx_pi` is an "over-kill". The `reduction` clause of OpenMP is designed for this particular purpose:

```
int N, i;
double sum = 0.;
double w = 1.0/N, x, approx_pi;

#pragma omp parallel for private(x) reduction(+:sum)
for (i=1; i<=N; i++) {
  x = w*(i-0.5);
  sum = sum + 4.0/(1.0+x*x);
}

approx_pi = w*sum;
```

Another example of using the `reduction` clause

$$s = a_0^2 + a_1^2 + \ldots + a_{N-1}^2$$

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

Loop scheduling

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

How are the loop iterations exactly divided among the threads?

- Mapping of loop iterations to threads is configurable in OpenMP.
- The "secret" is the `schedule` clause:

  ```
  #pragma omp parallel for schedule(static|dynamic|guided [,chunk])
  ```

- Default scheduling is `static` (no need to specify), which divides the iterations into contiguous chunks of (roughly) equal size.
- Other alternatives of scheduling: `dynamic` and `guided`
- It is possible to prescribe a chunksize in the `schedule` clause.

  ```
  #pragma omp parallel for schedule(dynamic,3)
  ```

Examples of different schedulers

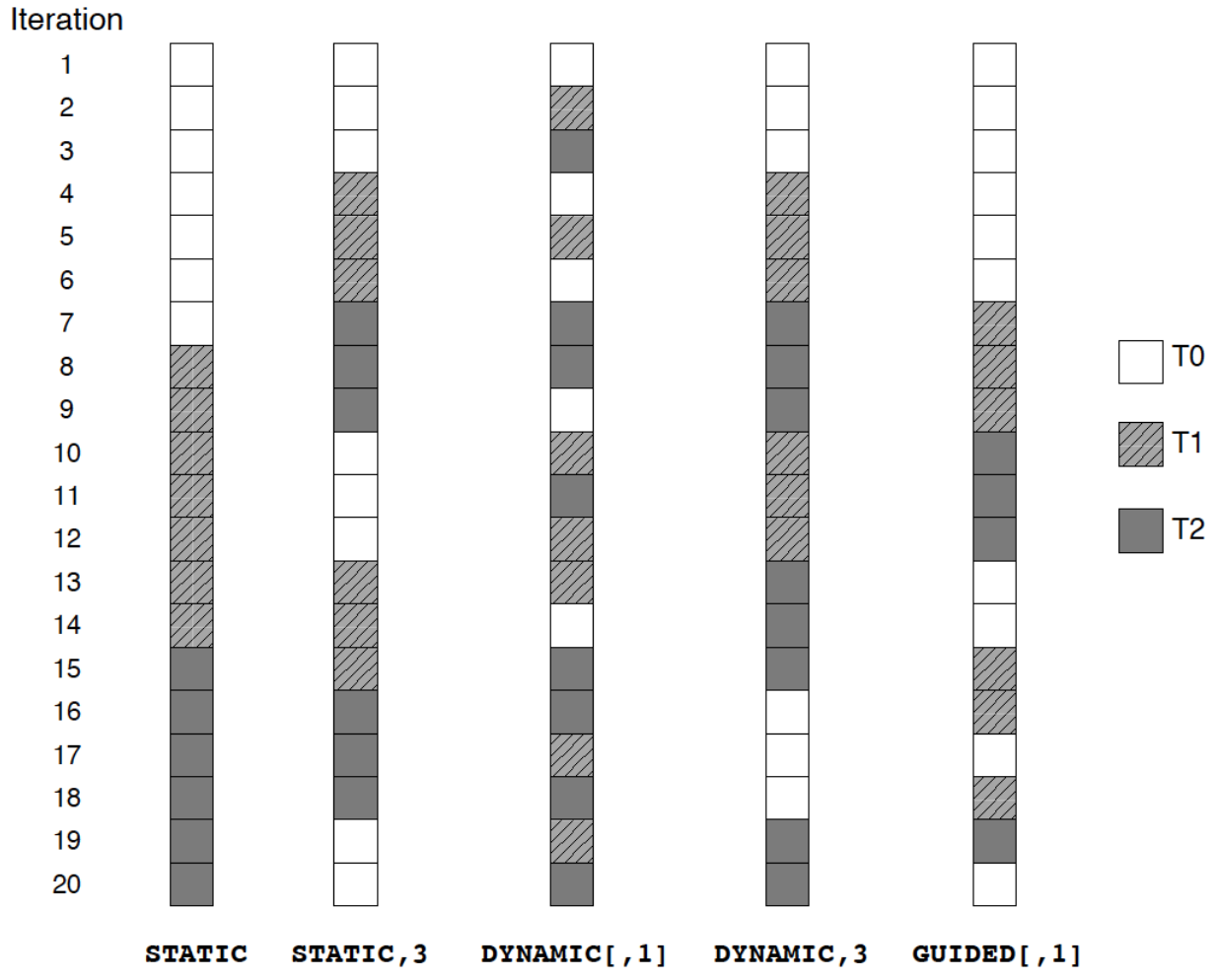


Figure 6.2: Loop schedules in OpenMP. The example loop has 20 iterations and is executed by three threads (T0, T1, T2). The default chunksize for `DYNAMIC` and `GUIDED` is one. If a chunksize is specified, the last chunk may be shorter. Note that only the `STATIC` schedules guarantee that the distribution of chunks among threads stays the same from run to run.

OpenMP compiler directives

An OpenMP compiler directive is a "parallelization hint" intended for a compatible compiler to automatically create parallel code.

To write an OpenMP program "means" to insert OpenMP compiler directives (plus additional code) at suitable locations of an existing serial program.

Some of the most important OpenMP directives:

- `#pragma omp parallel`
- `#pragma omp for`
- `#pragma omp single`
- `#pragma omp master`
- `#pragma omp critical`
- `#pragma omp barrier`

Performance pitfalls

- OpenMP is prone to the "standard problems" of parallel programming: **serial fraction** (Amdahl's law) and **load imbalance**.
- Communication (in terms of data transfer) on shared memory is usually much less costly than on distributed memory, but ccNUMA can potentially cause performance problem (will be discussed in Chapter 8).
- There are specific performance problems inherent with shared-memory programming.

Non-negligible overhead

- Whenever a parallel region is started or stopped, or a parallel loop is initiated or ended, there is some overhead involved, because
  - threads must be spawned (or least woken up from an idle state)
  - the "loop work" for each thread must be determined
  - for dynamic and guided scheduling, each thread that becomes available must be supplied with a new task
  - the default **barrier** synchronizes the threads
- These costs can be counted as "communication overhead" in refined scalability models:

$$S_{\mathrm{omp}}(N) = \frac{1}{s + (1-s)/N + \kappa N + \lambda}$$

where N denotes the number of threads, λ denotes overhead that is independent of N

Run serial code if parallelism does not pay off

If the worksharing construct does not contain enough "work" per thread because, e.g., each iteration of a short loop executes in a short time, OpenMP overhead will lead to very bad performance.

Use the "`if`" clause:

```
#pragma omp parallel for if (N>some_threadshold)
for (i=0; i<N; i++)
  A[i] = B[i] + C[i]*D[i];
```

Performance example

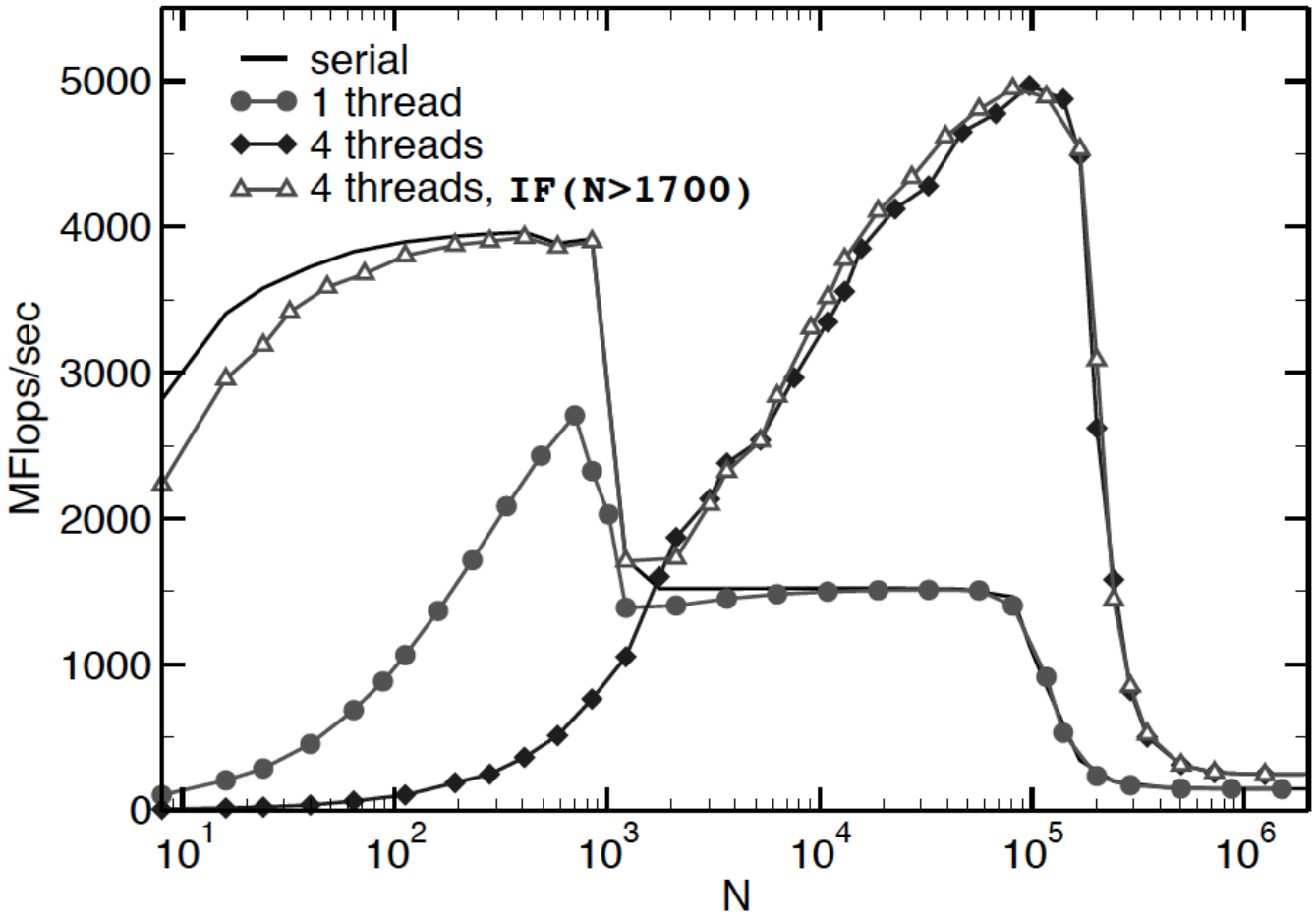


Figure 7.3: OpenMP overhead and the benefits of the `IF(N>1700)` clause for the vector triad benchmark. (Dual-socket dual-core Intel Xeon 5160 3.0 GHz system like in Figure 7.2, Intel compiler 10.1).

Another option

It is possible to reduce the number of threads for a particular part.

```
#pragma omp parallel for num_threads(2)
for (i=0; i<N; i++)
  A[i] = B[i] + C[i]*D[i];
```

"`num_threads`" is a clause.

Avoid implicit barriers

Be aware that most OpenMP worksharing constructs (including `#pragma omp for`) insert an automatic barrier at the end.

If you're **certain** that the default barrier (synchronization) is not necessary, then specify the "`nowait`" clause.

Parallelizing loop nests

Try to parallelize loop nests on a level as far out as possible.

Parallelizing inner loop levels will lead to increased OpenMP overhead because a team of threads is spawned or woken up repeatedly.

A bad example:

```
int i,j;
for (i=0; i<N; i++) {
  double R=0.;

#pragma omp parallel for reduction(+:R)
  for (j=0; j<M; j++)
    R += A[i][j]*B[j];

  C[i] = R;
}
```

How would you improve?

Merging parallel regions

Before:

```
int i;
doule S, R=0.;

#pragma omp parallel for reduction(+:R)
for (i=0; i<N; i++) {
  A[i] = do_work(B[i]);
  R += A[i];
}

S = sin(R);

#pragma omp parallel for
for (i=0; i<N; i++)
  A[i] += S;
```

There are two parallel regions.

Merging parallel regions

After:

```
int i;
doule S, R=0.;

#pragma omp parallel private(S)
{

#pragma omp for reduction(+:R)
for (i=0; i<N; i++) {
  A[i] = do_work(B[i]);
  R += A[i];
}

S = sin(R);

#pragma omp for nowait
for (i=0; i<N; i++)
  A[i] += S;

}  // end of the parallel region
```

Only one paralel region.

Beware of "trivial" load imbalance

The parallel loop trip count should be large compared with the number of threads, otherwise (serious) load imbalance can easily arise.

```
double res=0;
#pragma omp parallel for reduction(+:res) private(j,k)
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      res += A[i][j][k];
```

High possibility of load imbalance if the value of `N` is small.

“Collapsing” a loop nest

```
double res=0;
#pragma omp parallel for reduction(+:res) private(j,k) collapse(3)
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      res += A[i][j][k];
```

“collapse” is a clause, can help to improve the load balance.

Be careful about dynamic/guided scheduling or tasking

The dynamic/guided loop scheduling options and tasking constructs require some amount of nontrivial computation or bookkeeping in order to figure out which thread is to compute the next chunk or task. This overhead can be significant if each task contains only a small amount of work.

Be careful about “critical” regions

How to parallelize the following code?

```
int i,j, r_number;

for (i=0; i<N; i++) {
  r_number = row_calc(i);
  for (j=0; j<10; j++)
    M[r_number][j] = M[r_number][j] + some_function(i,j);
}
```

One “bad” solution of parallelization

```
int i,j, r_number;

#pragma omp parallel for privite(j, r_number)
for (i=0; i<N; i++) {
  r_number = row_calc(i); // different threads may get same 'r_number'!

  #pragma critical
  {
  for (j=0; j<10; j++)
    M[r_number][j] = M[r_number][j] + some_function(i,j);
  }
}
```

This is a bad solution, because the work is serialized.

A better solution

To improve the previous solution, we can use an array of "locks".

```
int i,j, r_number;
omp_lock_t locks[N];
for (i=0; i<N; i++)
  omp_init_lock(&locks[i]);

#pragma omp parallel for privite(j, r_number)
for (i=0; i<N; i++) {
  r_number = row_calc(i); // different threads may get same 'r_number'!

  omp_set_lock(&locks[r_number]);
  for (j=0; j<10; j++)
    M[r_number][j] = M[r_number][j] + some_function(i,j);
  omp_unset_lock(&locks[r_number]);
}

for (i=0; i<N; i++)
  omp_destroy_lock(&locks[i]);
```

This solution is better, because it allows concurrent execution among the threads (working on different `r_number`), while avoiding race conditions. (There is overhead with setting/unsetting the locks.)

Let's look at another example

How to parallelize the following code?

```
int i, ind;
int S[8];

for (ind=0; ind<8; ind++)
  S[ind] = 0;

for (i=0; i<N; i++) {
  ind = A[i]%8;
  S[ind] += 1;
}
```

One solution that will suffer from "false sharing"

```
int i, ind;
int S[max_num_threads+1][8];
for (ind=0; ind<8; ind++)
  S[0][ind] = 0;

#pragma omp parallel private(ind)
{
  int thread_id = omp_get_thread_num()+1;

  for (ind=0; ind<8; ind++)
    S[thread_id][ind] = 0;

  #pragma omp for nowait
  for (i=0; i<N; i++) {
    ind = A[i]%8;
    S[thread_id][ind] += 1;
  }

  #pragma critical
  {
    for (ind=0; ind<8; ind++)
      S[0][ind] += S[thread_id][ind];
  }
}
```

Avoiding false sharing

The problem with the previous solution is that each row of the 2D array "S" has only 8 integers (32 bytes). Therefore, multiple rows of "S" will occupy the same cache line $\rightarrow$ **false sharing**.

"**Padding**" the 2D array "S" will remove this potential bottleneck. For example:

```
int S[max_num_threads+1][64];
```

Locality and contention problems on ccNUMA systems

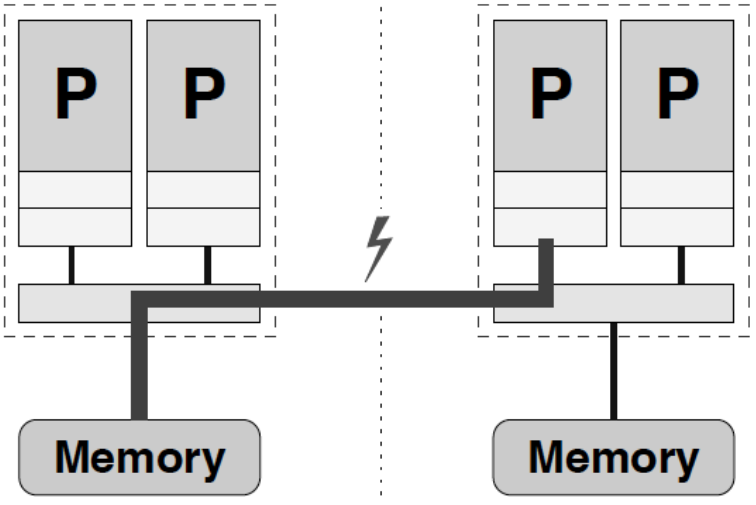


Figure 8.1: Locality problem on a cc-NUMA system. Memory pages got mapped into a locality domain that is not connected to the accessing processor, leading to NUMA traffic.

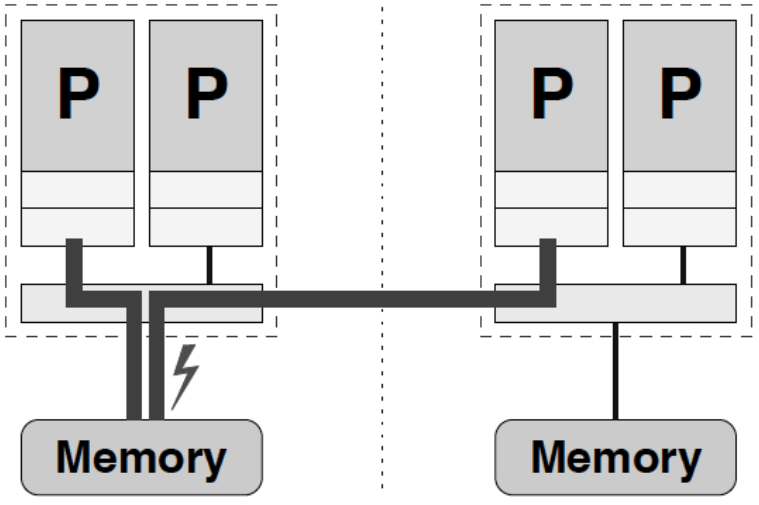


Figure 8.2: Contention problem on a cc-NUMA system. Even if the network is very fast, a single locality domain can usually not saturate the bandwidth demands from concurrent local and nonlocal accesses.

Both problems are due to "misplacement" of threads and data.

The “placement” problem

Proper page placement is essential.

- Make sure that memory gets mapped into the locality domains of processors that actually access them. This minimizes NUMA traffic across the network. (“Mapping” means that a page table entry is set up, which describes the association of a physical with a virtual memory page.)
- Threads or processes must be pinned to those CPUs which had originally mapped their memory regions in order not to lose locality of access. (Need to use appropriate thread affinity mechanisms.)

First touch

- Operating system and runtime libraries normally support a **first touch** policy for memory pages: A page gets mapped into the locality domain of the processor that first **writes** to it.
- **Note:** Merely *allocating* memory (such as `malloc`) is not sufficient.

Data initialization is key

- The data initialization part deserves close attention on ccNUMA.
  - Allocating arrays should use dynamic (heap) memory (`malloc`).
  - The `calloc` function will most probably be counterproductive.

A bad example

```
int *A, *B, *C, *D, i;

A = (int*)malloc(N*sizeof(int));
B = (int*)malloc(N*sizeof(int));
C = (int*)malloc(N*sizeof(int));
D = (int*)malloc(N*sizeof(int));

for (i=0; i<N; i++) {
  B[i] = i;
  C[i] = i%5;
  D[i] = i%10;
}

#pragma omp parallel for
for (i=0; i<N; i++)
  A[i] = B[i] + C[i]*D[i];
```

Why is the previous example bad?

- If the code is run across several locality domains, it will not scale beyond the maximum performance achievable on a single LD (if the working set does not fit into cache).
- This is because the initialization loop is executed by a single thread, writing to B, C, and D for the first time.
- Hence, all memory pages belonging to those arrays will be mapped into a single LD.
- Problem: many “remote” accesses and severe contention

Fixing the initialization problem

```
int *A, *B, *C, *D, i;

A = (int*)malloc(N*sizeof(int));
B = (int*)malloc(N*sizeof(int));
C = (int*)malloc(N*sizeof(int));
D = (int*)malloc(N*sizeof(int));

#pragma omp parallel for
for (i=0; i<N; i++) {
  B[i] = i;
  C[i] = i%5;
  D[i] = i%10;
}
```

First touch by “fake” initialization

- Sometimes initializing an array with useful values can only be done by one thread.
- Then, a parallelized “fake” (unnecessary) initiation beforehand can secure the desirable first touch.

Some requirements

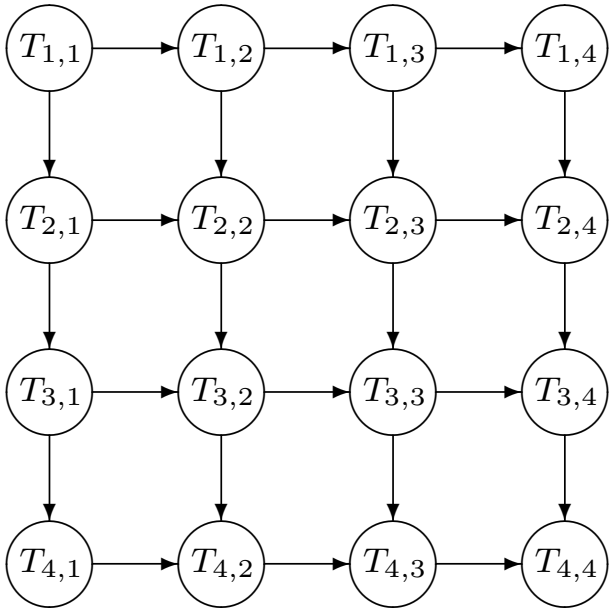
- The OpenMP loop schedules of initialization and work loops must be identical and reproducible.
  - Must use `static` scheduler and identical chunksize
- For successive parallel loops with the same number of iterations and the same number of parallel threads, each thread should get the same part of the iteration space in both loops.
  - Also must use `static` scheduler and identical chunksize
- Beware of cache-coherence traffic, which can destroy the achievable aggregate memory bandwidth.
- Beware of global data (which is initialized before the `main()` function). Properly mapped local copies of global data may be a possible solution.

NUMA-unfriendly OpenMP scheduling

- Dynamic/guided loop scheduling and OpenMP task constructs could be preferable over static work distribution in poorly load-balanced situations.
- On the other hand, any sort of dynamic scheduling (including tasking) will necessarily lead to scalability problems if the thread team is spread across several locality domains.
- This is because the assignment of tasks to threads is unpredictable and even changes from run to run, which rules out an "optimal" page placement strategy.
- In such cases, it may be best to distribute the working set's memory pages round-robin (choosing a suitable chunksize) across the locality domains.

Exercises

Exercise 1



The above figure shows the dependency relationship between 16 tasks: $T_{1,1}, T_{1,2}, \ldots, T_{4,4}$. Each directed edge in the graph connects a pair of "source" and "destination" tasks. A destination task cannot be started until all its source tasks are carried out. All the 16 tasks are equally time-consuming, requiring one hour of a worker.

How many hours minimum do 3 workers need to finish all the 16 tasks?

Exercise 2

What will be written to the screen by the following OpenMP program using 4 threads?

```
int main (int argc, char *argv[])
{
  int   i, n = 100;
  float a[100], sum = 0.0;

  for (i=0; i < n; i++)
    a[i] = i * 1.0;

#pragma omp parallel for default(shared) reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + a[i];

  printf("   Sum = %f\n",sum);
  return 0;
}
```

Exercise 3

```
void func (int N, int *arr)
{
  int pass, smallIndex, j, temp;
  for (pass = 0; pass < N-1; pass++)
  {
    smallIndex = pass;
    for (j = pass+1; j < N; j++) {
      if (arr[j] < arr[smallIndex])
        smallIndex = j;
    }
    temp = arr[pass];
    arr[pass] = arr[smallIndex];
    arr[smallIndex] = temp;
  }
}
```

Write an OpenMP parallelized version of the function "`func`".

Exercise 4

```
for (i=0; i<n-1; i++) {
    a[i] = b[i] + c[i];
    d[i+1] = d[i] + a[i]*c[i];
}
```

Explain why the above code segment cannot be directly parallelized by only inserting "`#pragma omp parallel for`" before the for-loop.

Note that `a`, `b`, `c`, `d` are 1D arrays of length n.

How to restructure the code so that OpenMP parallelization is achievable?