High performance computations with multithreading

Gene Soudlenkov
(g.soudlenkov@auckland.ac.nz)

Outline

Introduction

- Parallel computing
- Motivation
- Definitions

Parallel computing

Figure: Multitasking

- Concurrency and multitasking
- Multiprocessing and multithreading
- History of parallel computing

Did you know?

True parallelism goes back to Luigi and Menabrea, "Sketch of the Analytic Engine Invented by Charles Babbage", 1842

Motivation

- Leverage hardware and software advances
- Increase performance
- Improve response time
- Increase scalability

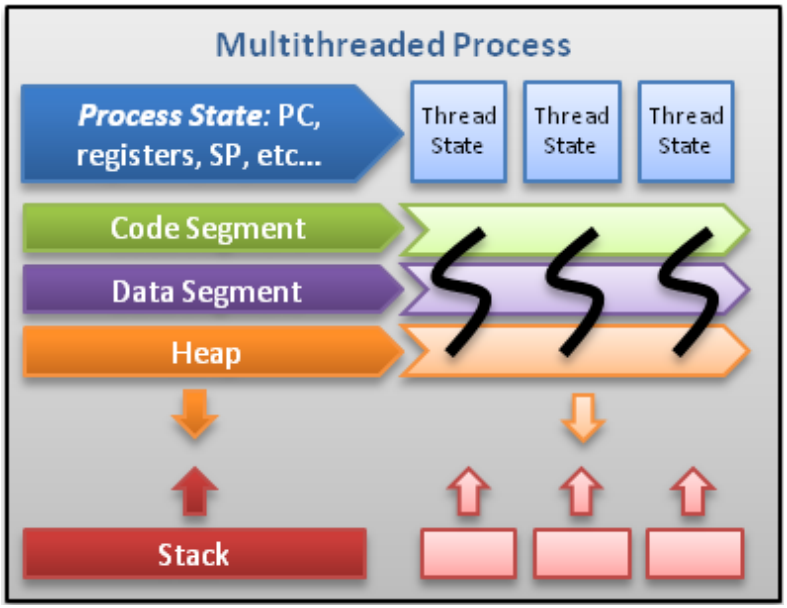
Did you know?

The ILLIAC IV was one of the first attempts to build a massively parallel computer. The ILLIAC IV design featured fairly high parallelism with up to 256 processors, used to allow the machine to work on large data sets in what would later be known as vector processing.

Threads and processes

- Multitasking/concurrency
- Multiprocessing and multithreading

Figure: Structure of a process



Process

A **PROCESS** is created by the operating system as a set of physical and logical resources to run a program. Each process has:

- heap, static, and code memory segments.
- environment information, including a working directory and file descriptors.
- process, group, and user IDs.
- interprocess communication tools and shared libraries.

Thread

A **THREAD** is the execution state of a program instance, sometimes called an independent flow of control. Each thread runs within the context of a parent process and is characterised by:

- registers to manage code execution.
- a stack.
- scheduling properties (such as priority).
- its own set of signals.
- some thread-specific data.

Multitasking

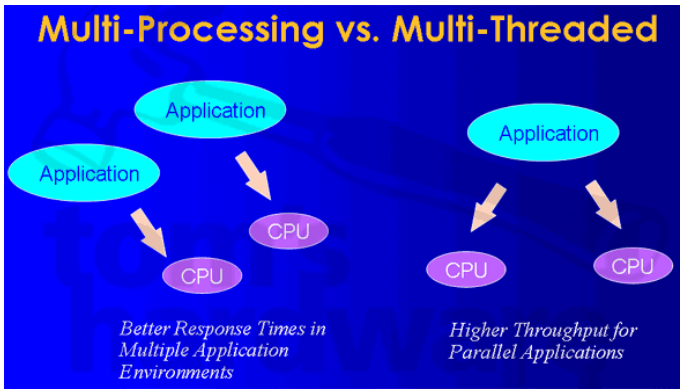


Figure: Stack

- Process
is heavy weight or resource intensive.
- Thread is light weight
taking lesser resources than a process.
- Process switching needs more
interaction with operating system.
- Thread switching has smaller
overhead when switching contexts.

Multitasking cont'd

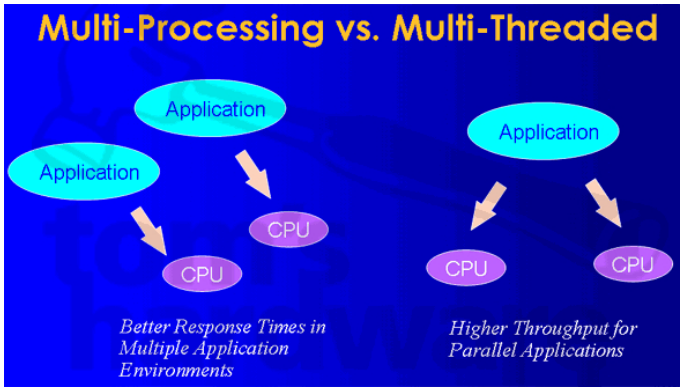


Figure: Stack

- In multiple processing environments each process executes the same code but has its own memory and file resources.
- All threads can share same set of open files, child processes.
- If one process is blocked then no other process can execute until the first process is unblocked.
- While one thread is blocked and waiting, second thread in the same task can run.
- Multiple processes without using threads use more resources.
- Multiple threaded processes use fewer resources.
- In multiple processes each process operates independently of the others.

Local variables

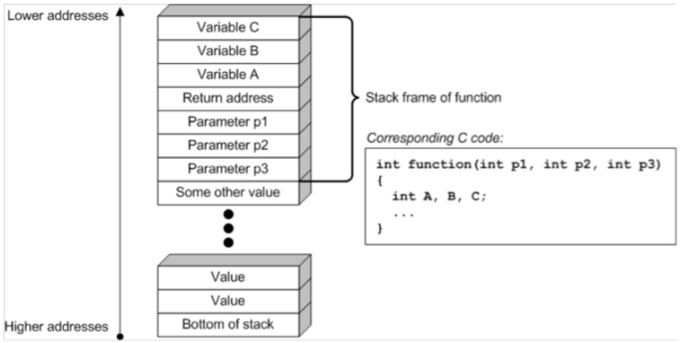


Figure: Stack

Whenever
a program calls a subroutine
by a CPU instruction CALL, it saves the current instruction pointer (IP) onto the stack. It marks its position before it branches, so it knows where to return after termination of the subroutine. This saved address on the stack is called a return address.

A high-level programming language like C or C++ also puts local variables of the subroutine on top of the stack. Thus, the subroutine gets its own memory area on the stack where it can store its own data. This principle is also the key of recursive routine calls because every new call to the subroutine gets its own return address and its own local variables on the stack.

Local variables

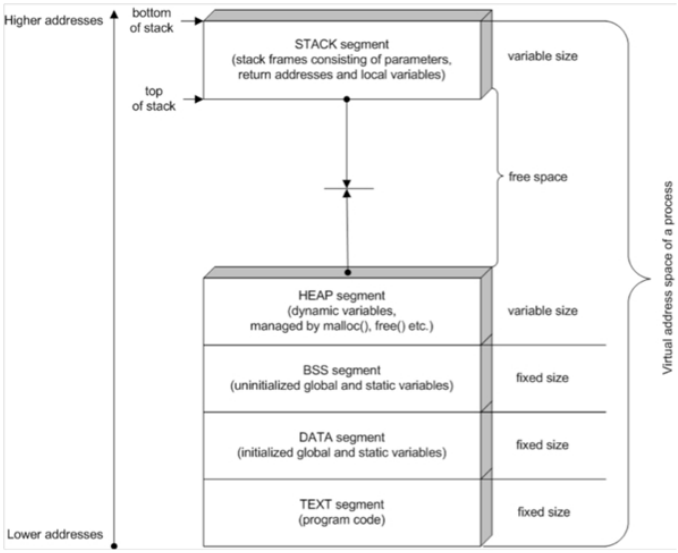


Figure: Memory layout of a process

Upon termination of the subroutine, high-level languages first clean up the stack by removing all local variables. After that, the stack pointer (SP) again points to the saved return address. At a RET or RETURN instruction, the processor reads the return address from stack, jumps back to the former IP position, and continues the original program flow.

Local variables

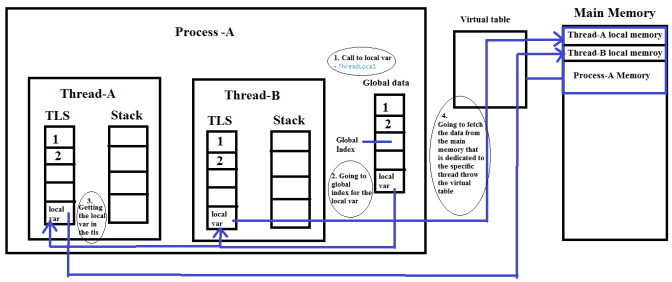


Figure: Memory layout of a process

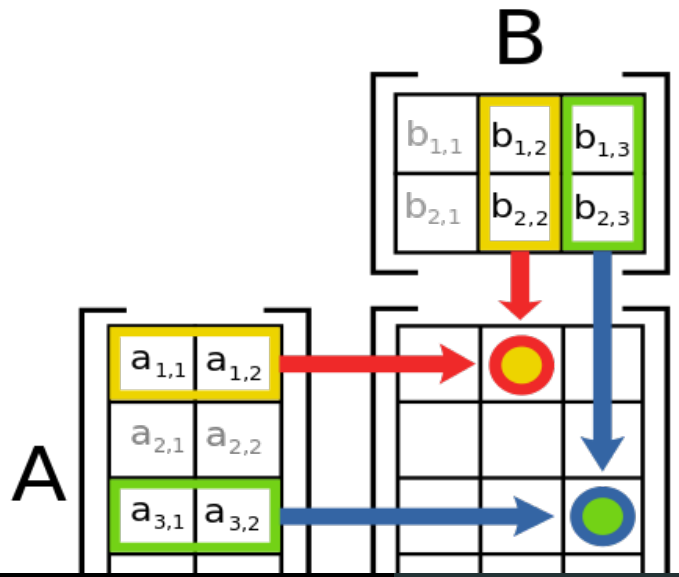
When creating a member variable (A field) or a static member (static field), the address of memory of the variable in the virtual address space (which is translated to a global memory address space) is shared between all threads that are created within the scope or the class.

TLS (Thread Local Storage)is a region in the heap that can only be accessed by a specific thread.

Sample problem - matrix multiplication

Multiply two matrices $matA$ and $matB$, storing the result in $matC$.
Dimensions of the sources matrices are passed in $dimxA$,$dimyA$,$dimxB$ and $dimyB$ variables.
Dimensions of the destination matrix is assumed to be $dimyA * dimxB$

Figure: Matrix multiplication algorithm



Matrix multiplication

Matrix multiplication algorithm

1. Check the sizes of two matrices $\mathbf{A}$ $(m \times n)$ and $\mathbf{B}$ $(t \times u)$: if $n = t$ then we can multiply them otherwise no (in that order $\mathbf{AB}$)
2. If they can be multiplied, then create a new matrix of size m by u
3. For each row in $\mathbf{A}$ and each column in $\mathbf{B}$ multiply and sum the elements and the place the results in the rows and columns of the result matrix $\mathbf{AB}$

Matrix multiplication has complexity of $O(n^3)$. It means that every time the dimension increases 10-fold, the time required will increase 1000-fold.

Reference serial code

```
int multiply(int dimxA,int dimyA,int *matA,int dimxB,
             int dimyB,int *matB,int *matC)
{
    int i,j,k,val;

    //run matrix multiplication loop
    for(i=0;i<dimyA;i++)
    {
        //clean destination line
        for(j=0;j<dimxB;j++)
        {
            val=0;
            for(k=0;k<dimxA;k++)
                val+=matA[i*dimxA+k]*matB[k*dimxB+j];
            matC[i*dimxB+j]=val;
        }
    }
    return 1;
}
```

Note

The sample code is straigtforward and simple. It does not in any way pretend to be

Reference serial code performance

Benchmarking is based on multiplying 100,300,500,1000,3000 and 5000-large square matrices.
Serial code execution gives the following results

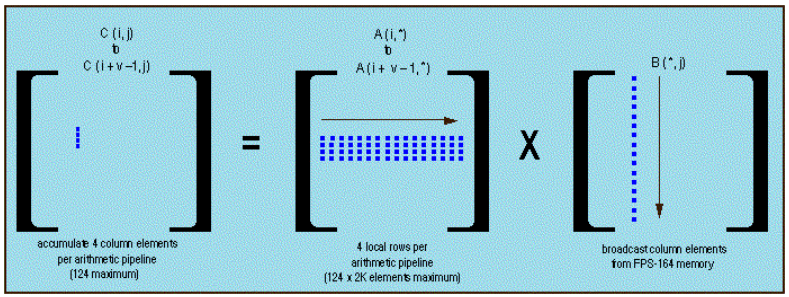
Dim	Time, ms
100	1
300	36
500	195
1000	8617
3000	267337
5000	1409010

Table: Serial code performance.

Native threads-based implementation

- Use native threads to improve performance
- Assume 12 cores machine
- Changes require in the code
- Keep the algorithm intact

Figure: Parallel matrix multiplication



Note

Higher-grade optimisation of matrix multiplication requires algorithm re-design. It is advisable to take into account the nature of the matrices.

Grid decomposition

(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)	(0, 6)	(0, 7)
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)	(1, 7)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6)	(2, 7)
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)	(3, 7)
(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	(4, 6)	(4, 7)
(5, 0)	(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)	(5, 6)	(5, 7)
(6, 0)	(6, 1)	(6, 2)	(6, 3)	(6, 4)	(6, 5)	(6, 6)	(6, 7)
(7, 0)	(7, 1)	(7, 2)	(7, 3)	(7, 4)	(7, 5)	(7, 6)	(7, 7)

(a) Original grid

(b) Grid decomposition

The example above demonstrates grid decomposition for 4 threads.

Native threads-based implementation

- Require additional routine
- Parallelise outermost loop only
- Assign a subset of lines per thread
- Need thread configuration structure
- Simple implementation - no messaging support required

Threads will receive configuration data upon creation - no need to deliver configuration data through external means (queues, messages, etc).

```
struct MatMult
{
    int dimxA,dimyA,dimxB,dimyB; //dimensions
    int *matA,*matB,*matC; //src and dst matrices
    int line_start,line_end; //start and end lines
                             //for the thread
};
```

Native threads-based implementation

```
#define THREADS_NUM 12
int multiply(int dimxA,int dimyA,int *matA,
             int dimxB,int dimyB,int *matB,
             int *matC)
{
    //store thread handles
    pthread_t threads[THREADS_NUM];
    //configuration structures per thread
    struct MatMult config[THREADS_NUM];
    int count_per_thread; //number of lines per thread
    int i;

    //calculate number of lines per thread
    count_per_thread=dimyA/THREADS_NUM;
```

Native threads-based implementation

```
    for(i=0;i<THREADS_NUM;i++)
    {
        config[i].dimxA=dimxA;
        config[i].dimxB=dimxB;
        config[i].dimyA=dimyA;
        config[i].dimyB=dimyB;
        config[i].matA=matA;
        config[i].matB=matB;
        config[i].matC=matC;
        config[i].line_start=i*count_per_thread;
        config[i].line_end=config[i].line_start+
                                count_per_thread;
    }
    config[THREADS_NUM-1].line_end+=
                            dimyA%THREADS_NUM;
    for(i=0;i<THREADS_NUM;i++)
        pthread_create(&threads[i],NULL,
                multiply_aux,&config[i]);

    for(i=0;i<THREADS_NUM;i++)
       pthread_join(threads[i],NULL);
    return 1;
}
```

Native threads-based implementation

The actual working routine should:
Note: threads are created asynchronously, which means that pthread_create call will not block and wait until the thread function is done.

- Accept thread configuration for matrices
- Work within the line limits

Native threads-based implementation

```
void *multiply_aux(void *ptr)
{
    struct MatMult *pv=(struct MatMult *)ptr;
    int i,j,k,val;

    //run matrix multiplication loop
    for(i=pv->line_start;i<pv->line_end;i++)
    {
        for(j=0;j<pv->dimxB;j++)
        {
            val=0;
            for(k=0;k<pv->dimxA;k++)
                val+=pv->matA[i*pv->dimxA+k]*
                        pv->matB[k*pv->dimxB+j];
            pv->matC[i*pv->dimxB+j]=val;
        }
    }
    return NULL;
}
```

Native threads-based implementation performance

Benchmarking is based on multiplying 100,300,500,1000,3000 and 5000-large square matrices.
Native threads-based code gives the following results

Dim	Time - serial	Time - native threads
100	1	1
300	36	8
500	195	45
1000	8617	696
3000	267337	23179
5000	1409010	126169

Table: Native threads-based performance.

OpenMP

- Industry standard
- Supported by major compiler developers
- Supproted for multiple languages
- Multiplatform
- Based on directives or pragmas
- Spawns team of threads to handle parallel requests
- Supports shared and thread-local variables
- Supports conditional parallelisation
- No need to change the code significantly

OpenMP directives

- Composed of a **sentinel** followed by the command and the clause:
  - C/C++ sentinel: **#pragma omp**
  - Fortran sentinel: **!$OMP**
- Commands include:
  - **parallel**: forms a team of threads and starts parallel execution
  - **for**: parallelise loop automatically
  - **sections**: Defines non-iterative worksharing construct
  - **single**: specifies a single-thread block in a team
  - **critical**: restrict execution of a block to a single thread
  - **barrier**: specifies an explicit barrier at the point
  - **atomic**: ensures atomicall access to a specific storage
  - **threadprivate**: specifies thread local storage for the specified variables

Example

```
#pragma omp parallel for private(i,j) schedule(dynamic)
```

PARALLEL directive

- Forms a team of threads
- Team size is controlled by either
  1. Environment variable OMP_NUM_THREADS
  2. Explicitly set by function call omp_set_num_threads()
  3. By default - using the number of CPUs in the system
- Can be conditioned with IF clause

Example

```
#pragma omp parallel
printf("Hello, world from thread %d\n",omp_get_thread_num());
```

Output

```
Hello, world from thread 0
Hello, world from thread 2
Hello, world from thread 1
Hello, world from thread 3
```

PARALLEL directive, continued

- Can be combined with **for** command to provide automatic loop parallelization
- The iterations of the loop will be distributed dynamically in evenly sized pieces
- Threads will not synchronize upon completing their individual pieces of work if NOWAIT is specified

Example

```
int N=10,result=0,i;
#pragma omp parallel for if (N>5) reduction(+:result)
for(i=0;i<N;i++) result+=i*5;
```

PARALLEL directive, **Data Scope Clauses**

- **IF** (scalar logical expression) - conditional parallel execution: check if there is enough work to do?
- **DEFAULT (PRIVATE—SHARED—NONE)** - establishes default value for sharing attribute
- **REDUCATION**(operator : variable) - variable is initialised with to relevant value and at the end of the loop the value of the variable processes through the reduction operator for each thread
- **SHARED**(list) - declares list to be shared by tasks
- **PRIVATE**(list) - declares list to be private to a task
- **FIRSTPRIVATE**(list) - same as private, initialises each member of the list with the value given
- **LASTPRIVATE**(list) - same as private, leaves the value of each member as it was before vacating the block
- **COPYIN**(list) - copies the value of the master thread variables to the thread variables of other threads

Data scoping - lets re-iterate

- Every variable has scope: **shared** or **private**
- Scoping can be controlled with scoping clauses:
  - shared
  - private
  - firstprivate
  - lastprivate
  - **reduction** clause explicitly identifies a reduction variable as private
- Scoping is one of the leading error sources in OpenMP
  - Unontended sharing of variables
  - Privatization of the variables that must be shared

PARALLEL directive, **Execution Control Clauses**

- **SCHEDULE** (type, chunk) - Describes how iterations of the loop are divided among the threads in the team. The following policies supported:
  1. STATIC
  2. DYNAMIC
  3. GUIDED
  4. RUNTIME
  5. AUTO
- **ORDERED** - ensures predictable order of threads scheduling
- **NOWAIT** - If specified, then threads do not synchronize at the end of the parallel loop.
- **COLLAPSE**(scalar) - Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause.

SCHEDULE STATIC

STATIC schedule means that iterations blocks are mapped statically to the execution threads in a round-robin fashion.

The nice thing with static scheduling is that OpenMP run-time guarantees that if you have two separate loops with the same number of iterations and execute them with the same number of threads using static scheduling, then each thread will receive exactly the same iteration range(s) in both parallel regions.

This is quite important on NUMA systems: if you touch some memory in the first loop, it will reside on the NUMA node where the executing thread was. Then in the second loop the same thread could access the same memory location faster since it will reside on the same NUMA node.

SCHEDULE DYNAMIC

DYNAMIC scheduling works on a "first come, first served" basis. Two runs with the same number of threads might (and most likely would) produce completely different "iteration space" - "threads" mappings as one can easily verify

Since precompiled code could be run on various platforms it would be nice if the end user can control the scheduling. That's why OpenMP provides the special **RUNTIME** scheduler clause. With runtime scheduling the type is taken from the content of the environment variable OMP_SCHEDULE. This allows to test different scheduling types without recompiling the application and also allows the end user to fine-tune for his or her platform.

SCHEDULE GUIDED

There is another reason to choose between static and dynamic scheduling - workload balancing. If each iteration takes vastly different from the mean time to be completed then high work imbalance might occur in the static case.

Take as an example the case where time to complete an iteration grows linearly with the iteration number. If iteration space is divided statically between two threads the second one will have three times more work than the first one and hence for 2/3 of the compute time the first thread will be idle. Dynamic schedule introduces some additional overhead but in that particular case will lead to much better workload distribution.

A special kind of dynamic scheduling is the **GUIDED** where smaller and smaller iteration blocks are given to each task as the work progresses.

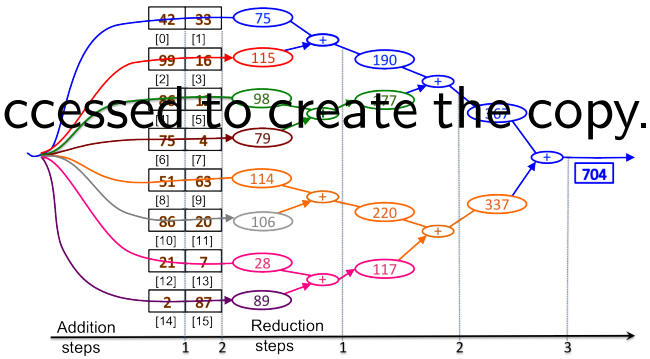
REDUCTION clause

- **REDUCTION** clause applies operation to the variable stated after all threads are done
- Variables listed are automatically declared private
- Reduces synchronisation overhead

```
int factorial(int number)
{
    int factor=1, i;
#pragma omp parallel for reduction(*:factor)
    for(i=2;i<number;i++)
        factor*=i;
    return factor;
}
```

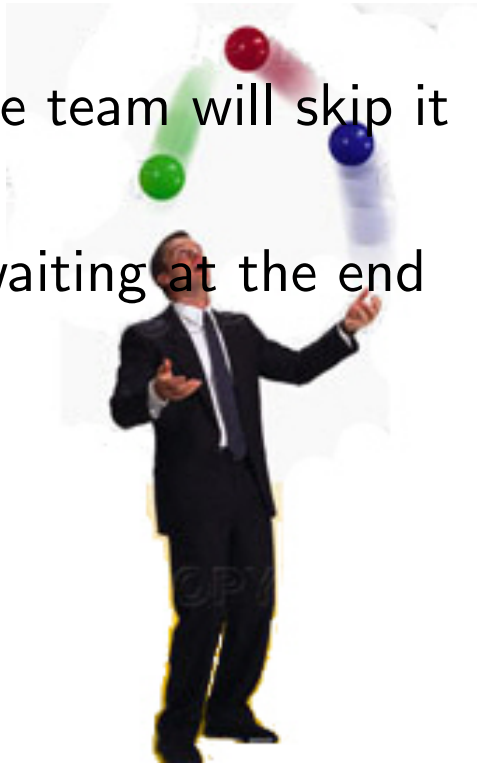
REDUCTION clause

- At the beginning of the paralle block a private copy is made of the variable and pre-initilized to a certain value
- At the end of the parallel block the private copy is atomically merged into the shared variable using the defined operator
- The private copy is actually just a new local variable by the same name and type, the original variable is not accessed to create the copy.



SINGLE execution

- Allows code to be executed serially in parallel region
- Any thread will run the code, the rest of the team will skip it and wait until SINGLE block is done
- **NOWAIT** attribute can be used to avoid waiting at the end of the block

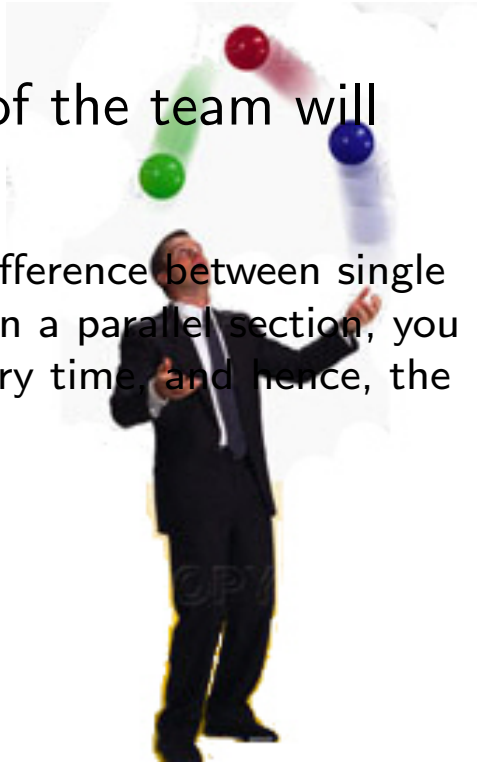


Example

```
#pragma omp parallel
{
    printf("In␣parallel,␣all␣threads
#pragma␣omp␣single
␣␣␣␣printf("Printed from only one thread\n");
␣␣␣␣printf("In parallel again, all threads\n");
}
```

MASTER execution

- Allows code to be executed serially in parallel region by the master thread only
- Only master thread run the code, the rest of the team will skip it without waiting



Unless you use the threadprivate clause, the only important difference between single nowait and master is that if you have multiple master blocks in a parallel section, you are guaranteed that they are executed by the same thread every time, and hence, the values of private (thread-local) variables are the same.

Example

```
#pragma omp parallel
{
    printf("In␣parallel,␣all␣threads
#pragma␣omp␣master
␣␣␣␣printf("Printed from only one thread\n");
␣␣␣␣printf("In parallel again, all threads\n");
}
```

FLUSH directive

Even when variables used by threads are supposed to be shared, the compiler may take liberties and optimize them as register variables. This can skew concurrent observations of the variable.
The flush directive can be used to ensure that the value observed in one thread is also the value observed by other threads.
In the example, it is enforced that at the time either of a or b is accessed, the other is also up-to-date.
You need the flush directive when you have writes to and reads from the same data in different threads.

Example from the OpenMP specification

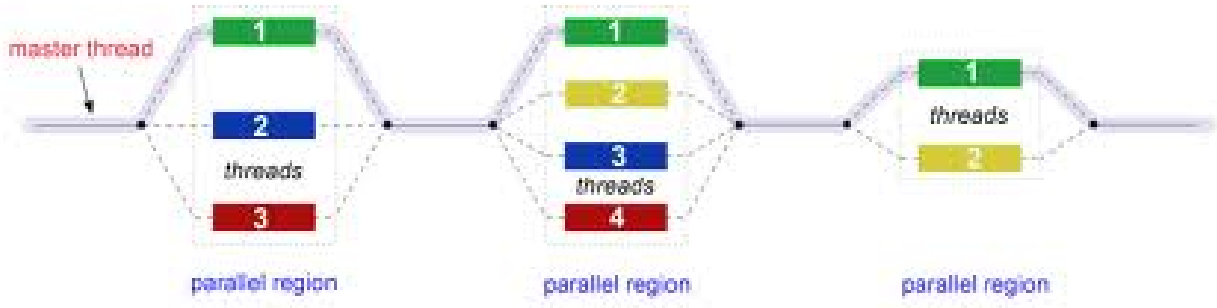
```
/* First thread */
b = 1;
#pragma omp flush(a,b)
if(a == 0)
{
  /* Critical section */
}
```

```
/* Second thread */
a = 1;
#pragma omp flush(a,b)
if(b == 0)
{
  /* Critical section */
```

Sections

- **SECTIONS** directive is a non-iterative work-sharing construct
- Each **SECTION** is executed once by a thread in the team
- It is possible for a thread to execute more than one section if it is quick enough and the implementation permits this

Figure: Sections execution



Sections: Example

Example

```
#pragma omp parallel
 {
     printf("Parallel 1. Going from thread %d\n",
         omp_get_thread_num());
#pragma omp sections
     {
         printf("Entered into the sections , thread %d\n",
             omp_get_thread_num());
#pragma omp section
             printf("Section 1. Going from thread %d\n",
                 omp_get_thread_num());
#pragma omp section
             printf("Section 2. Going from thread %d\n",
                 omp_get_thread_num());
     }
     printf("Parallel 2. Going from thread %d\n",
         omp_get_thread_num());
 }
```

Sections: Example output

As the output suggests the only thread executing the sections was thread 3 - the rest of the threads skipped sections part.

Output

```
Parallel 1. Going from thread 3
Entered into the sections, thread 3
Section 1. Going from thread 3
Section 2. Going from thread 3
Parallel 1. Going from thread 2
Parallel 1. Going from thread 0
Parallel 1. Going from thread 1
Parallel 2. Going from thread 3
Parallel 2. Going from thread 0
Parallel 2. Going from thread 1
Parallel 2. Going from thread 2
```

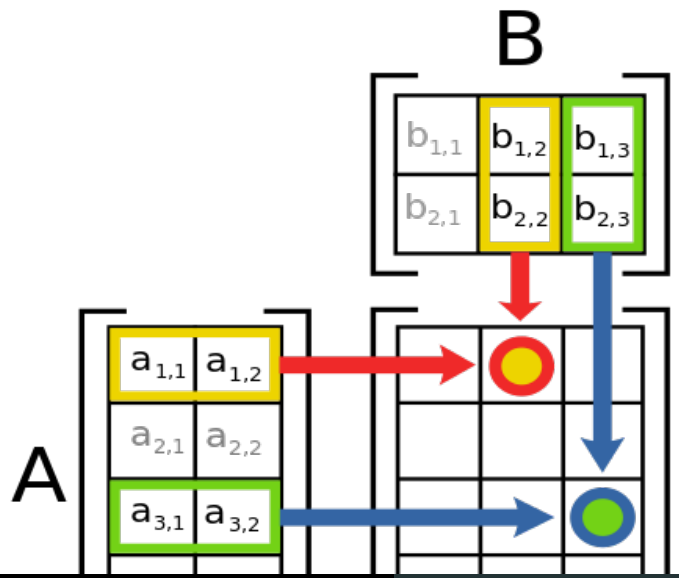
Synchronization in OpenMP

- **MASTER** forces the block to be executed only by the master thread
- **CRITICAL**(name) is used to serialize work in parallel block
- if **CRITICAL** is named, all critical block with the same name are serialized
- **ATOMIC** is used to ensure that only a single thread will execute the **statement** followed. The directive is **not** structured
- **BARRIER** is used to force all threads in the team wait upon reaching the barrier point. Barriers are costly. They should not be used inside other synchronization blocks, such as **CRITICAL**, **SINGLE**, **SECTIONS** or **MASTER**

Sample problem - matrix multiplication with OpenMP

Multiply two matrices $matA$ and $matB$, storing the result in $matC$.
Dimensions of the sources matrices are passed in $dimxA$,$dimyA$,$dimxB$ and $dimyB$ variables.
Dimensions of the destination matrix is assumed to be $dimyA * dimxB$

Figure: Matrix multiplication algorithm



Reference OpenMP code

```
int multiply(int dimxA,int dimyA,int *matA,int dimxB,
             int dimyB,int *matB,int *matC)
{
    int i,j,k,val;

    //run matrix multiplication loop
#pragma omp parallel for shared(matC) private(j,k,val)
    for(i=0;i<dimyA;i++)
    {
        //clean destination line
        for(j=0;j<dimxB;j++)
        {
            val=0;
            for(k=0;k<dimxA;k++)
                val+=matA[i*dimxA+k]*matB[k*dimxB+j];
            matC[i*dimxB+j]=val;
        }
    }
    return 1;
}
```

OpenMP-based implementation performance

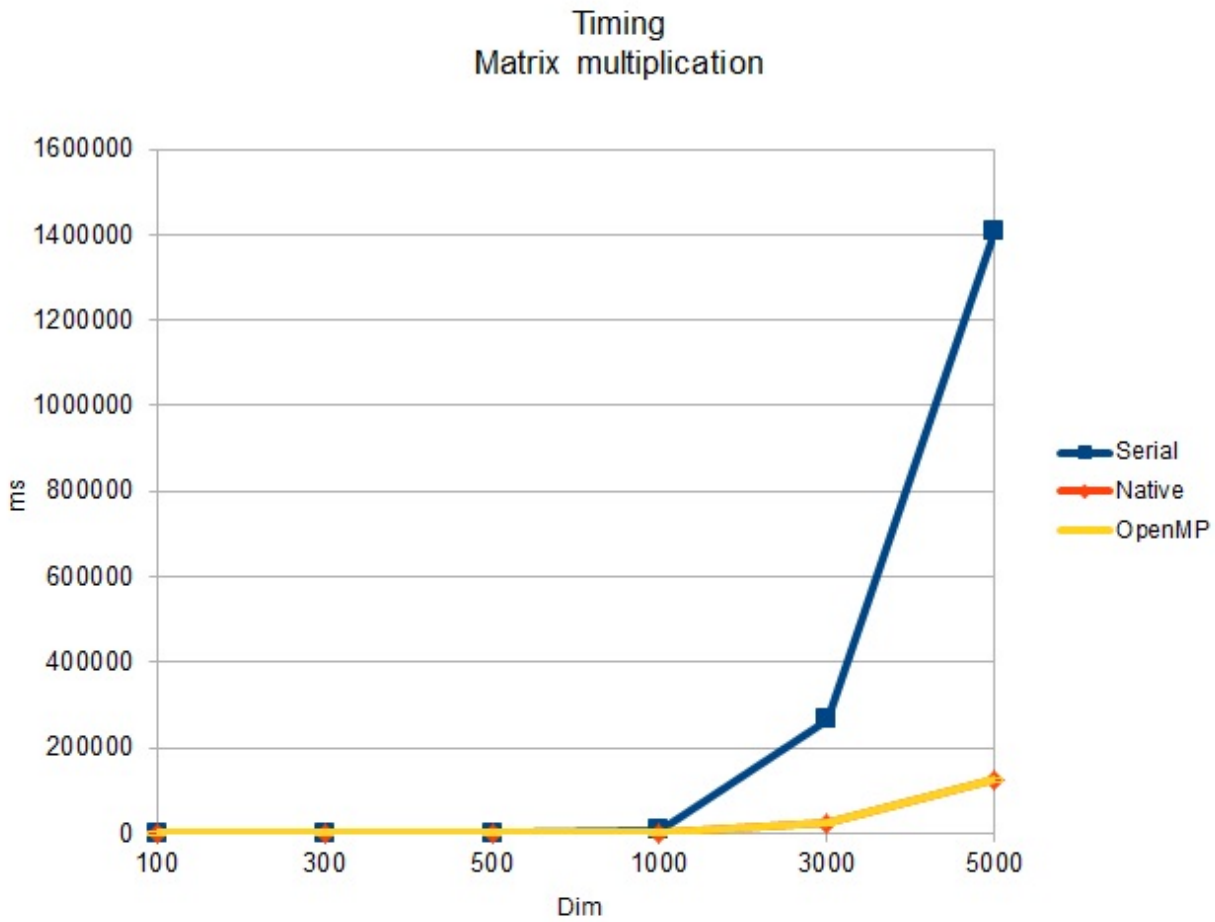
Benchmarking is based on multiplying 100,300,500,1000,3000 and 5000-large square matrices.
OpenMP-based code gives the following results

Dim	Time - serial	Time - native threads	Time - OpenMP
100	1	1	1
300	36	8	8
500	195	45	43
1000	8617	696	656
3000	267337	23179	23101
5000	1409010	126169	125913

Table: OpenMP-based performance.

Combined performance graph

Figure: Timing of matrix multiplication code



Advanced loop parallelisation - nested loops

- Problem - nested loops, can they be parallelised?
- This code **DOES** not work!

```
#pragma omp parallel for
for(int i=0;i<10;i++)
#pragma omp parallel for
for(int j=0;j<10;j++)
...
```

- OpenMP 3.0 loop nesting works

```
#pragma omp parallel for collapse(2)
for(int i=0;i<10;i++)
for(int j=0;j<10;j++)
...
```

- Another alternative: enable nesting by calling omp_set_nested(1);

Conclusions

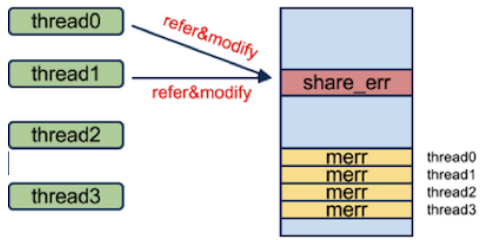
+ Threads **may** increase performance proportional to the threads team size
+ Native threads implementation requires significant rework of the code
+ OpenMP threads usage can be implemented with minimal code changes
+ OpenMP threads usage is cross-platform and is easy to maintain across various operating systems and compilers
- Threads can lead to a variety of conditional problems
- Multithreaded applications are harder to debug


1
Honey Bees are assigned jobs based on their age:
For instance, for the worker bee:
1-2 days old: spend their time cleaning cells, starting with the or
they were born in, and keeping the brood warm
3-5 days old: they feed older larvae
6-11 days old: they feed the youngest larvae

Figure: Bees

Locking issues - race conditions

- Multithreaded code suffers from the bugs related to multiple readers/writers of the same object
- If not protected, an object access may suffer from **race condition** where multiple threads may try and change/retrieve status of the same object at the same time
- Serialization of access is required in order to protect against race condition
- Locking mechanisms differ from one OS to another



Locking issues - race conditions

- Race conditions are difficult to detect and debug
- Mathematical proof of software correction is not necessarily enough to ensure the lack of race conditions
- Example: Therac-25 disaster, 6 died
- North American blackout of 2003

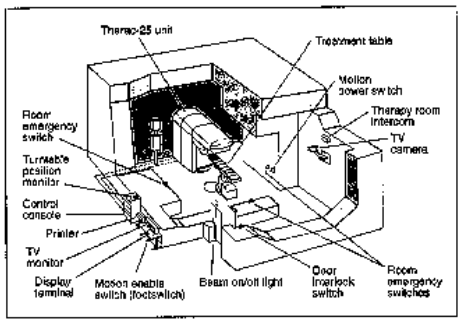
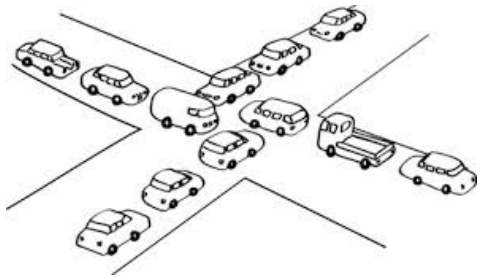


Figure 1. Typical Therac-25 facility.

Locking issues - deadlocks

- A **deadlock** occurs when at least two tasks wait for each other and each cannot resume until the other task proceeds
- Often happens when code block requires locking of multiple mutexes at once
- Usually the order of mutexes to be locked must be preserved among threads in order to avoid deadlocks
- No matter how much time is allowed to pass, this situation will never resolve itself



Locking issues - OpenMP locking API

- OpenMP provides rich, cross-platform API for locking support
- OpenMP locks are wrappers around the platform-specific implementations of mutex operations
- OpenMP runtime library provides a lock type, omp_lock_t in its include file omp.h

OpenMP API

- omp_init_lock initializes the lock
- omp_destroy_lock destroys the lock
- omp_set_lock attempts to set the lock. If the lock is already set by another thread, it will wait until the lock is no longer set, and then sets it
- omp_unset_lock unsets the lock
- omp_test_lock attempts to set the lock. If the lock is already set by another thread, it returns 0; if it managed to set the lock, it returns 1.

Locking issues - OpenMP locking API example

- omp_destroy_lock can be called only for unlocked objects
- Kernel transition inside the lock call adds up to the overhead

```
omp_lock_t writelock;

omp_init_lock(&writelock);
#pragma omp parallel for
for( i = 0; i < x; i++ )
{
    omp_set_lock(&writelock);
    // one thread at a time
    a=b*c;
    omp_unset_lock(&writelock);
}
omp_destroy_lock(&writelock);
```

Scoped locks vs. CRITICAL vs. ATOMIC

- It is allowed to leave the locked region with jumps (e.g. break, continue, return), this is forbidden in regions protected by the critical-directive
- Scoped locking is exception safe, critical is not
- All criticals wait for each other, with guard objects you can have as many different locks as you like - named critical sections help a bit, but name must be given at compile-time instead of at run-time like for scoped locking
- The most important difference between critical and atomic is that atomic can protect only a single assignment and you can use it with specific operators
- Addition with critical section is 200 times more expensive than simple addition, atomic addition is 25 times more expensive then simple addition

Environment variables

OpenMP uses a set of environment variables that can be modified to ensure the best performance for the application.

- **OMP_NUM_THREADS**: number, set the desired number of threads in a team
- **OMP_DYNAMIC**: true or false, forces dynamic schedule type to be used
- **OMP_STACKSIZE**: number optionally followed by unit specification B, K, M or G, specifies the size of the stack for threads created by OpenMP. If unit is not specified, kilobytes (K) is assumed

Loadleveler and OpenMP

- Loadleveler provides special considerations for OpenMP applications
- #parallel_threads=N reserves the required number of cores
- OMP_NUM_THREADS variable is automatically set by Loadleveler according to parallel_threads value

Loadleveler job description example

```
#@account_no=uoa
#@class=default
#@group=nesi
#@resources=ConsumableMemory(100mb) ConsumableVirtualMemory(100mb)
#@wall_clock_limit=10:00
#@job_type=serial
#@parallel_threads=4
#@output = $(job_name).$(jobid).out
#@error = $(job_name).$(jobid).err
#@queue

./run_my_omp_app
```

Questions & Answers