Multicore programming: Low-level libraries

Jukka Julku

19.2.2009

Outline

Contents

1 Processes and threads 1
2 TBB 2
3 MPI 3
4 UPC 4
5 OpenMP 5
  5.1 Model 5
  5.2 Directives 7
  5.3 Considerations 13
6 Examples 13
7 14

Disclaimer

• There are several low-level libraries, languages and directive based approaches
  • But no silver bullets
• This presentation only covers some examples of them
• OpenMP is today’s main subject!

1 Processes and threads

Lowest of the low

• Processes
• Threads
  • POSIX threads, etc.
• Basically “The assembly language of parallel computing”
But there are also advanced threading libraries like Boost C++ threading library. Programming processes and threads directly is usually tedious and error prone task. For example, data races and deadlocks are common problems. But on the other hand the programmer has full control over the parallelism. Scalability might be a problem as it is not trivial to remove (memory) dependencies and balance work loads between the threads. Development time is certain to increase from that of single-threaded code.

If POSIX threads are considered as the assembly language of multicore programming, then the Boost threading library can be compared to C. The Boost C++ threading library encapsulates threads inside objects. The library adds lots of functionality compared to normal low-level threading libraries. For example, thread groups, lock types, like shared and upgradable locks, and barriers are included.

2 TBB

Intel's Threading Building Blocks (TBB)

- C++ Library
- Abstracts low-level threading detail
  - Concept of tasks
- Uses C++ templates
  - Resembles STL
- Designed for portability and scalability
- Includes
  - Task manager to schedule tasks
  - Concurrent data structures
  - Concurrent algorithms
  - Synchronization primitives and memory allocators

The concept of logical tasks allows to abstract from low-level threading details. A task is an C++ object. This makes tasks lighter execution entities than threads. The programmer specifies tasks that he or she wants to perform concurrently by deriving objects from TBB's task base class. TBB runtime system maps these task to processor cores automatically.

When a program is initialized task manager analyzes the program and execution environment, chooses optimal numbers of threads that form a thread pool. TBB also performs dynamic load-balancing between the threads. This is implemented as work stealing between the threads: Each thread has its own task queue and when the queue runs empty threads start to fetch tasks from other thread’s queues randomly. Because of TBB’s own scheduler, the scheduling may trade fairness for efficiency (compared to OS level scheduling).

TBB contains concurrent containers. These are thread-safe implementations of container classes. They are based on fine-grain blocking or lockless algorithms to maximize concurrency. The concurrent algorithms of TBB, like parallel for, while and pipeline, are also highly tuned for parallelism. There are both free and commercial version of TBB available.

The following example illustrates how elements of an array are squared by using TBB’s parallel_for:
```cpp
#include "tbb/blocked_range.h"

class SqChunk {
    float *const local_a;

public:
    void operator()(const blocked_range<size_t>& x) const {
        float *a = local_a;
        for(size_t i=x.begin(); i!=x.end(); ++i)
            a[i] = a[i];
    }

    SqChunk(float a[]) :
        local_a(a) {}
};

void Square(float a[], size_t n) {
    parallel_for(blocked_range<size_t>(0, n), 1000, SqChunk(a));
}

3 MPI

Message-passing interface (MPI)

• The de-facto standard in distributed memory computing
  - Huge number of existing parallel applications already run over MPI!
• Processes, point-to-point and collective communications, topologies, etc.
• Multicores?
  - Depends on the library implementation
  - Hybrid model

MPI is for distributed memory like what threads are for shared-memory: very low-level programming interface. MPI is a standard and there are several implementation of it, like MPICH and OpenMPI. Currently MPI is especially used in cluster computing and to program worlds largest super computers.

MPI program uses single-program-multiple-data (SPDM) paradigm and is divided into several processes that all have their own address space. SPDM means that each process executes the same program, but may branch to different parts of it freely.

In addition to the features mentioned above the standard includes data marshalling methods, different communication modes, like asynchronous, synchronous and buffered modes, communicators to combine logically related communication, one-sided communications etc. Scalability and portability are high in general. Data races do not exist because of separate address spaces, but deadlocks are possible. Development time is likely to increase because of the low-level approach.

MPI processes can execute on single core, multicore or distributed system. Different question is how efficient it is to run separate processes that communicate via message-passing inside shared-memory system. Especially the data copies between process address spaces may become a problem. Shared-memory doesn't certainly hurt MPI program and many applications originally designed for distributed memory run quite well on multicore [5]. It's not obvious that shared-memory programming tool like OpenMP always outperforms MPI [6]. If the difference in performance is small, then using MPI might be a better option as it allows for distributed-memory at the same time. Some MPI implementations optimize for
```
shared-memory communication, but in other cases communications may be expensive. There seems to be some on-going research to make MPI efficient on multicore.

The following example shows how MPI SPMD model works. Process with rank 0 sends all other processes “hello” text using point-to-point communications and the other processes print the text.

```c
#include <mpi.h>
#include <stdio.h>

#define BUF_SIZE 8
#define TAG 0

int main(int argc, char *argv[]) {
    MPI_Status status;
    int my_rank;
    int comm_size;
    char buf[BUF_SIZE];
    int i;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    if (my_rank == 0) {
        buf = strcpy(buf, "Hello!\n");
        printf("Process %d: %sn", my_rank, buf);
        for (i = 0; i < comm_size; i++) {
            MPI_Send(buf, BUF_SIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(buf, BUF_SIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &status);
        printf("Process %d: %sn", my_rank, buf);
    }
    MPI_Finalize();
    return 0;
}
```

4 UPC

Unified parallel C

- C extension
  - Full language
- SPMD
- Shared or distributed memory
- Single logically partitioned address space
  - Threads have shared and private data
- Work distribution
  - upc_forall()
• Synchronization
  - barriers, locks

UPC is just one example of low-level languages. There are several others, usually C or Fortran extension. For example Cilk and High-performance Fortran.

As UPC is a language, compiler support is needed in addition to a runtime library. UPC partitions the program into several logical partitions. Single thread executes each partition. UPC uses single-program-multiple-data paradigm like MPI. The language adds some new keywords to the ISO-C99 standard. The keywords can be used to share execution of for-loops between threads, form barriers and share data. The programmer can distribute data to the partitions and for example arrays can be spread over the partitions.

The following example calculates Celsius to Fahrenheit conversions for temperatures of 0, 10, ..., 40 degrees, sharing the loop iterations between the threads.

```c
#include <upc.h>
#include <stdio.h>

#define TABLE_SIZE 5

int main(int argc, char *argv[]) {
    static shared int fahrenheit[TABLE_SIZE];
    static shared int step = 10;
    int celsius, i;

    /* Iterations are shared between the threads */
    for (i = MY_THREAD; i < TABLE_SIZE; i += THREADS) {
        celsius = step * i;
        fahrenheit[i] = celsius * (9.0 / 5.0) + 32;
    }
    upc_barrier;

    if (MY_THREAD == 0) { /* Only done by one thread */
        for (i = MY_THREAD; i < TABLE_SIZE; i += THREADS) {
            celsius = step * i;
            printf("%d\t%f\n", fahrenheit[i], celsius);
        }
    }

    return 0;
}

5 OpenMP

OpenMP

• Portable programming interface for shared-memory parallel computers

• Compiler assisted parallel programming
  - Compiler directives, library routines and environmental variables

• Provides means to
  - Create teams of threads
  - Share work between the threads
  - Declare private or shared data
  - Synchronization and exclusive access
OpenMP’s main idea arises from the fact that compilers cannot do parallelization well in most cases because they lack knowledge about the dependencies in the program. Because of this compilers have to make safe assumptions. OpenMP tries to help the compiler in this by providing description of the parallelism and tell how the parallelism can be exploited.

OpenMP requires compiler support. Current GCC supports OpenMP version 2.5, GCC versions from 4.4 on will support OpenMP standard 3.0.

5.1 Model

The basic idea

- Write (sequential) code
- Identify parallel regions
  - The main effort of parallelization
- Use OpenMP directives to express parallelism
- Compile and run

In OpenMP the programmer provides high-level description of parallelism by marking certain blocks of code with OpenMP directives. The OpenMP implementation (compiler and runtime system) take care of the actual parallelization. The standard gives much freedom for the compiler to extract best possible performance from the code. The compiler can for example rearrange the code quite freely.

In most cases OpenMP allows the programmer to preserve the sequential code beside the directives. An OpenMP program can be compiled to normal single-threaded program by simply ignoring the directives. For example, re-writing of algorithms to parallel environment is not always needed. Minimal changes to the code and possibility to compile the code to sequential program simplify debugging.

Motivation

- Simple idea
- Easy to learn syntax
- Parallelize (old) sequential code
- Standard technique
  - Portable
- Incremental parallelization
- C/C++ and Fortran support

Execution model

- Fork and join model
  - Based on blocks of parallel computation
  - Thread team is formed at entry
  - Barrier at exit
  - Nested blocks
OpenMP is primarily designed to exploit parallelism in loop nest. The number of threads used to execute a block can be specified before or during the execution by using environmental variables. Threads may have different execution paths inside a block. The thread that encounters the block, becomes the master thread of the block.

**Memory model**

- Shared-memory
  - Threads also have private memory
  - Data transfer is controlled by directives
- Relaxed memory consistency
  - Threads may have their own view of the world between certain synchronization points
    - The memory is automatically synchronized at barriers, entry and exit to critical sections and lock routines
  - Flush-operation for the programmer
  - Motivation: Flexibility of implementation

The programmer can decide which parts of the memory are private and which are not by granularity of variable. Data transfers to threads private memory are generated from the directives, but are otherwise transparent to the programmer. By default all data is shared, with some exceptions, like loop variables of parallel for loops.

**Core elements**

Control variables are used to obtain information from the program at runtime and influence the program execution. Environmental variables govern the behavior program in general and affect for example things like default number of threads used (OMP_NUM_THREADS) in parallel section, loop scheduling (OMP_SCHEDULE) declared with clause “runtime” and so forth. Each value can be queried or set at runtime by OpenMP runtime library routines. Other routines include for example omp_get_thread_num(), to query number of the executing thread. More throughout description of these can be found in OpenMP specification [2].
5.2 Directives

Directives: C-language

```
#pragma omp directive -name [clause [,] clause]...
```

- Clauses control how the directives are compiled and executed

Evaluation order of the clauses is arbitrary. There are several clauses to control the directives, but all directives don’t support every clause. Not all the clauses are presented here. Refer [2] for the rest of them.

Parallel constructs

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

- The most important construct
- Defines a parallel section
  - Everything not inside a parallel construct is sequential
  - Best practice: Maximize parallel regions
- Spawns a team of threads
- Does not do any work sharing!
- Implicit barrier at the end

There are some restrictions for the parallel blocks: No branching in or out of the section is allowed and the outcome must not depend on the evaluation order, i.e. in what order the threads actually execute. Nested parallel regions are allowed, but according to the standard an OpenMP implementation does not have to support this. Using many nested parallel structures may lead to very heavy memory use at runtime as each threads needs its own stack area. Using parallel constructs in inner loops is not advisable, as entry and exit to parallel sections form performance overhead.

If-clause can be used to make the parallel execution conditional. If the clause evaluates to false, the section is executed sequentially instead. num_threads - clause can be used to dynamically define the number of threads to use in the parallel block.
Example

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

int main()
{
#pragma omp parallel
{
    printf("Thread \%d\n", omp_get_thread_num());
}
    return 0;
}
```

The above example prints the text once per thread.

Work-sharing constructs

- Specify how to share work between the thread team

- Restrictions
  - Each thread at the team must encounter or non at all
  - Sequence of work-sharing constructs and barriers must be the same

- Barriers at exit by default
  - Can be removed with nowait -clause

- Orphaning

  The concept of orphaning means directives don't need to be in lexical scope of a parallel block, but they may be for example inserted inside functions. If function of this sort is called from within parallel block, the work-sharing is performed. In case the function is called from sequential code, the directives are simply ignored.

  The programmer can affect the performance of the program much by carefully considering the work-sharing. Minimizing number of barriers and avoiding load-imbalance and false-sharing (cache) can increase the parallelism in the program.

Loop construct

```c
#pragma omp for [clause [; clause] ...]
```

- The most common work-sharing construct

- Distributes loop iterations among the threads

- Limited to loops whose iterations can be counted
  - i.e. for (init-expr; var relop b; incr-expr)

  The loop variables must be of type integer and they are private to each thread by default. The iterations must be independent and order of execution must not affect the outcome. It is responsibility of the programmer to take care of this.

  Schedule clause can be used to define how the iterations are shared between the threads, otherwise the compiler decides this. Scheduling can have several different types: static, dynamic, guided or runtime. In static scheduling the iterations are divided into chunks and shared to threads in round-robin fashion. In dynamic scheduling threads request for more chunks as soon as they finish their earlier work. Guided scheduling is close to dynamic scheduling, but the chunk size varies over time. Type runtime specifies that the scheduling is decided at runtime according to environmental variable.
Example

```c
#pragma omp parallel shared(A, B, C, n) private(i)
{
    #pragma omp for schedule(dynamic, 100)
    for (i=0; i<n; i++) {
        A[i] = B[i] + C[i];
    }
}
```

Sections and single constructs

```c
#pragma omp sections [clause [, clause] ...]
{[#pragma omp section structured block]
    [#pragma omp section structured block]
    ...
}
```

- Distribute independent units of work
- Each section is executed exactly once

```c
#pragma omp single [clause [, clause] ...]
structured block
```

- Block is executed by only one thread
- For example: Initialization tasks between two loops

Sections construct can be used to assign task that do not depend on each other to threads in the team. The number of sections don’t have to be the same as number of threads. In case there are less threads than sections, some threads execute more than one section. Also, execution order of the sections is arbitrary. For performance reasons sections should have balanced loads.

The thread that executes the block of single construct is not specified and can be any thread. Other threads wait at the implicit barriers until the execution stops.

Example

```c
#pragma omp parallel private(i, j)
{
    #pragma omp sections
    {
        #pragma omp section
        {
            for (i=0; i<n; i++) {
                C[i] = A[i] + B[i];
            }
        }
        #pragma omp section
        { // m != n
            for (j=0; j<m; j++) {
                F[j] = D[j] * E[j];
            }
        }
    }
}
```
**Task construct**

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

- New in OpenMP 3.0
- When the directive is encountered a new task is created from the block
- The task can be executed by any thread of the team
- Taskwait construct specifies a wait on the completion of child tasks generated since the beginning of the current task
- Where to use: While-loops and recursive code

**Example**

```c
void quicksort(int *array, int left, int right) {
    int i;
    if (left >= right) {
        return;
    }
    i = partition(array, left, right);
    #pragma omp task
    quicksort(array, left, i - 1);
    quicksort(array, index + 1, right);
    #pragma omp taskwait
}
```

OpenMP also provides shortcuts for combinations of parallel construct and a work-sharing construct:

```
#pragma omp parallel for/section/...
```

This mainly provides better readability, but in some cases the compiler can perform also better as it can assume more. This of course applies only to single work-sharing constructs. If multiple constructs are used, using single parallel block for multiple constructs decreases parallelization overhead as less threads are created and some barriers can be avoided.

**Some memory clauses**

- **shared(list)**
  - Shared by all threads
- **private(list)**
  - Private copy for each thread
- **firstprivate(list)**
  - Private, initialized with variable of the same name
- **lastprivate(list)**
- Private, last value is accessible after the section
  - Last: last iteration in sequential order, last section
  - default(none|shared|private)
    - Used to state default sharing attribute

Memory clauses are used to control memory access and define which variables are allocated from each thread’s private stack area. Private variables are not defined at entry or after exit from the parallel section. Private variables do not automatically get values of the variables with the same name than private variable that are declared before the parallel section. In addition, value of these variables are not defined after the parallel section in OpenMP 2.5 either. In OpenMP 3.0 the value is preserved.

There are some other memory clauses mainly for broadcasting values of one thread to the other threads at entry to a block. For example, copyin clause copies master thread’s values into variables of all other threads.

Synchronization 1/2

- Implicit barrier is not always enough
- OpenMP defines set of other synchronization mechanisms
  - Directives for different kinds of critical sections
  - Lock routines

OpenMP defines the following lock routines:

```c
void omp_init_lock (omp_lock_t *lck)
void omp_destroy_lock (omp_lock_t *lck)
void omp_set_lock (omp_lock_t *lck)
void omp_unset_lock (omp_lock_t *lck)
void omp_test_lock (omp_lock_t *lck)
```

Locks work like mutexes in any threading library. There are also another version of locks, nested locks, that can be locked several times by the same thread (omp_XXX_next_lock). Entry and exit from lock routine are memory synchronization points. Using directives is easier from the programming point of view than directly using locks as the programmer has full responsibility, just like with normal threading libraries, when using locks.

Synchronization 2/2

```c
#pragma omp barrier
#pragma omp ordered
structured block
#pragma omp critical [(name)]
structured block
#pragma omp atomic statement
#pragma omp master
structured block
```

The ordered construct allows a block inside parallel loop to be performed in the sequential order. It should be avoided as it limits the concurrency and so forms an obvious performance penalty. Ordered structures must be predeclared in the parallel directive with “ordered” clause.
Atomic construct is limited to certain binary operators. In principle only the assignment is atomic, so if for example function call is made as part of the atomic statement, the call is not atomic. In general, the programmer must ensure atomicity by declaring all uses of the same variable atomic.

The master directive is used to declare blocks that are executed only by the master thread. There is no implicit barrier at the end of the block.

Flush

```
#pragma omp flush [(list)]
```

- Relaxed memory consistency
  - Synchronized at synchronization points
    * Barriers, entry and exit to critical section and lock routines
- Threads may need to have consistent view between the synchronization points
- Flush can be used to synchronize memory in these situations
  - Granularity: Variables

Flush operation works as memory fence. The compiler cannot move reads and writes of the flushed memory over the flush operation. However, flushes may be reordered and this may cause memory consistency problems if the programmer doesn’t notice it.

Example

```
int x, y;
#pragma omp parallel shared(x) private(y)
{
  #pragma omp master
  { x = 1; }
  y = x;
}
```

- Its not guaranteed that x is 1 as there’s no barrier!
- Flush(x) is needed before the assignment

5.3 Considerations

Simple, but...

- Cannot prevent:
  - Data races
  - Thread-safety problems
  - Deadlocks
- Memory consistency problems
  - Programmer must ensure that the values are the same between the synchronization points
  - Compiler can rearrange code, even flush!
    * This may lead to hard to find problems
- But still, OpenMP removes lot of work and in most cases is quite simple approach!
Performance

• Just parallelizing the code with OpenMP directives doesn’t necessarily bring the expected benefits
  – Especially with large number of threads fine-tuning may be required

• Parallelization overhead
  – Thread management
  – Waiting at barriers
  – Etc.

6 Examples

Code examples

• If there is still time left, some more examples

7

The end

Any questions?

Sources

1 Chapman, B., Jost, G. and Van Der Pas, R: Using OpenMP - Portable shared memory parallel programming. 2008.
  – Very good introduction to OpenMP and parallel programming in general

2 OpenMP specification 3.0. www.openmp.org

3 TBB: http://www.threadingbuildingblocks.org/

4 MPI standard 2.0

5 Leiserson, C. Mirman, I.: How to survive the multicore software revolution.

6 Krawezik, G. and Capello, F.: Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors.