Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 6:

- Shared-Memory Programming with OpenMP
  - Introduction to OpenMP
  - The Fork-Join Model
  - Loop Parallelization

OpenMP

OpenMP is a shared-memory API that provides a portable, user-friendly and efficient approach to shared-memory parallel programming.

However, OpenMP is not a new programming language, but it provides notion that can be added to existing sequential code written in C/C++ and Fortran to

- describe how the work is to be divided among threads that will run on different cores
- synchronize accesses to shared data as needed

The OpenMP environment is comprised of the following three components:

- a set of compiler directives, commonly known as OpenMP pragmas
- a library of support functions
- a runtime environment

OpenMP

- The file *omp.h* must be included as a header file.
- Different compilers need different option flags.
  - gcc, llvm/clang: -fopenmp
  - intel : -openmp

OpenMP: Incremental Parallelization

OpenMP is designed to allow an incremental approach to parallelizing existing sequential code:

- Portions of an existing sequential program are parallelized in successive steps.
- This is in contrast to the all-or-nothing conversion of an entire program in a single step.
  - This is typically the approach other parallel programming environments such as Pthreads and MPI adopt.
- Therefore, OpenMP provides us with the ability to add parallelism to existing sequential code without the need to significantly rewrite it.

OpenMP also enables the programmer to work with a single source code:

- If a single set of source files contains both the sequential and parallel versions of a program, then maintenance effort is substantially reduced.
- OpenMP requires compiler support: Compiling an OpenMP program with a non-OpenMP-enabled compiler will default to the sequential version.

OpenMP: Incremental Parallelization

The ability of OpenMP to support incremental parallelization is one of its greatest advantages over the other parallel programming environments since it allows the programmer to:

- profile the execution of a sequential program
- sort the program blocks according to how much time they consume
- consider each block in turn beginning with the most-time consuming parallelizable one
- stop when the effort required to achieve further performance improvements is not warranted.

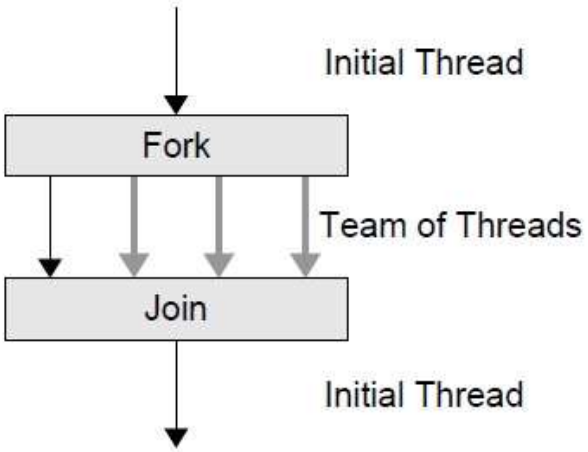
OpenMP: The Fork-Join Model

OpenMP attempts to provide a structured approach to multithreading programming in the form of the Fork-Join programming model.

Under this multithreading programming model, a program starts as a single thread of execution, just like a sequential program:

- The thread that executes this code is referred to as the initial thread.
- Whenever an OpenMP parallel construct is encountered by a thread, the OpenMP runtime creates or awakens a team of threads (this is called a fork) and becomes the master thread of the team.
- The master thread and the other team members will then cooperatively execute the code within the parallel construct.
- At the end of the parallel construct, the master thread continues whereas the other team members are terminated or suspended (this is called a join).
- The code enclosed by a parallel construct is called a parallel region.

OpenMP: The Fork-Join Model



The Fork-Join Programming Model of OpenMP under which a program starts as a single thread, the initial thread, which forks a team of threads when it encounters a parallel region and joins with the other team members at the end of the parallel region.

You can think of a sequential program as a special case of parallel execution with only the initial thread and no fork-joins in it.

OpenMP: Overview

OpenMP provides a set of compiler directives.

- An OpenMP directive is a pragma that applies to code that immediately follows it.

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

- An OpenMP directive generally affects only those threads that encounter it.
- Many of the OpenMP directives are applied to a structured block – a sequence of statements with a single entry at the top and a single exit at the bottom.
  - In other words, the program is not allowed to branch in and out of the associated block of code.
  - In C/C++, only the start of a block is marked by a pragma since the end of the block is explicit in C/C++.
  - In Fortran, both the start and the end need to be explicitly marked.

OpenMP: Overview

OpenMP provides means for users to:

- create teams of threads for parallel execution
- specify how to share work among the members of a team
- declare both shared and private variables
- synchronize threads and allow them to perform certain operations exclusively, i.e., without interference by other threads

OpenMP: Parallel Construct

To create a team of threads upon entering a parallel region, the programmer can simply specify the parallel region by inserting a $parallel$ directive immediately before the start of the parallel region.

- Each thread in a team is assigned a unique thread number.
- The OpenMP library function $omp\_get\_thread\_num()$ can be used to obtain the thread number.

At the end of a parallel region is an implicit barrier:

- Thus, no thread can progress past the barrier until all the threads in the team have reached that point in the program.
- Only the initial thread continues execution after the end of the parallel region.

Note: If a team of threads executing a parallel region encounter another $parallel$ directive, each thread in the current team creates a new team of threads and becomes its master.

- Nesting enables realization of multilevel parallel programs.

OpenMP: Parallel Construct

The $parallel$ directive plays a crucial role in OpenMP: a program without a parallel construct will be executed sequentially:

Caveats:

- Although the parallel construct ensures that computations are performed in parallel, it does not distribute the work among the threads in the team so the work will be replicated.
- An OpenMP program that branches into and out of a parallel region is non-conforming and its behavior is undefined.
- In C++, a throw inside a parallel region must cause execution to resume within the same parallel region and it must be caught by the same thread that threw the exception.

OpenMP: Parallel Construct

```
#pragma omp parallel
{
    std::cout << "The parallel region is executed by thread " << omp_get_thread_num() << std::endl;

    if(omp_get_thread_num()== 2)
    {
        std::cout << "Thread " << omp_get_thread_num() << " does things differently." << std::endl;
    }

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

The snippet illustrates an example of a parallel region where all threads execute the first *cout* statement, but only the thread with thread number 2 executes the second one.

OpenMP: Parallel Construct

```
The parallel region is executed by thread 0
The parallel region is executed by thread 3
The parallel region is executed by thread 2
  Thread 2 does things differently
The parallel region is executed by thread 1
```

The figure shows the output of the OpenMP code from the previous slide, where 4 threads are used in this execution.

OpenMP: Parallel Construct

if *(scalar-expression)*	(C/C++)
if *(scalar-logical-expression)*	(Fortran)
num_threads *(integer-expression)*	(C/C++)
num_threads *(scalar-integer-expression)*	(Fortran)
private *(list)*	
firstprivate *(list)*	
shared *(list)*	
default(none | shared)	(C/C++)
default(none | shared | private)	(Fortran)
copyin *(list)*	
reduction *(operator:list)*	(C/C++)
reduction *({operator | intrinsic_procedure_name}:list)*	(Fortran)

Clauses supported by the parallel construct – note that the *default(private)* clause is not available in C/C++.

OpenMP: Parallel Construct

Caveats:

- An OpenMP program must not depend on any ordering of the evaluations of the clauses of the parallel directive or any side effects of the evaluations of the clauses.
- At most one *if* clause can appear on the directive.
- At most *num_threads* clause can appear on the directive.

OpenMP: Parallel Construct

- You can specify the number of threads to be created upon entry to a parallel construct with the *num_threads* clause.

- You can deactivate a parallel construct using the *if* clause to ensure that it contains enough work for the parallelization of the parallel region to be worthwhile.

  - In such a circumstance, we say that the parallel region is inactive.

OpenMP: Work Sharing Constructs

OpenMP's work-sharing constructs are used to distribute work among the threads in a team and are also used to specify the manner in which the work in the region is to be distributed:

- A work-sharing construct must bind to an active parallel construct.
- If a work-sharing construct is encountered within an inactive parallel construct, it is simply ignored.

The two main rules regarding work-sharing constructs are as follows:

- Each work sharing region is encountered by all threads in a team or by none at all.
- The sequence of work sharing regions and barrier regions must be the same for all threads in a team.

By default, threads waits at an implicit barrier at the end of a work-sharing construct until the last thread has completed its share of the work:

- You can suppress the implicit barrier using the *nowait* clause.

OpenMP: Loop Construct

The loop construct causes the parallel region immediately follows it to be executed in parallel:

- At runtime, the loop iterations are distributed across the threads in the team.

```
#pragma omp for [clause[[,] clause]...]
    for-loop
```

- It is limited to the kinds of loops where the number of iterations can be counted:
  - The loop must have an integer counter variable whose value is incremented (or decremented), by a fixed amount at each iteration until some specified upper bound (or lower bound) is reached.

```
for ( init-expr ; var relop b ; incr-expr )
```

OpenMP: Loop Construct

private *(list)*	
firstprivate *(list)*	
lastprivate *(list)*	
reduction *(operator:list)*	(C/C++)
reduction *({operator | intrinsic_procedure_name}:list)*	(Fortran)
ordered	
schedule *(kind[,chunk_size)]*	
nowait	

Clauses supported by the loop construct

OpenMP: Loop Construct

```
int i,n = 1024;

#pragma omp parallel shared(n), private(i)
{
    #pragma omp for
    for(i=0;i<n;i++)
    {
        std::cout << "Thread " << omp_get_thread_num() << " executes loop iteration " << i << std::endl;
    }/*---End of parallel for ---*/

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

The snippet illustrates an example of a work-sharing loop where each thread executes a subset of the iteration space $i = 0, \dots, n - 1$.

OpenMP: Loop Construct

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

The figure shows the output of the example where a work-sharing loop is implemented from the previous slide - The example was executed for *n=9* and used *4* threads.

OpenMP: Loop Construct

```
#pragma omp parlllel shared(n,a,b), private(i)
{
    #pragma omp for
    for(i=0 ; i<n ; i++)
    {
        a[i] = static_cast<double>(i);
    }/*---Implicit barrier---*/

    #pragma omp for
    for(i=0 ; i<n ; i++)
    {
        b[i] = a[i]*static_cast<double>(i);
    }/*---Implicit barrier---*/
}/*---End of parallel region---*/
```

Two work-sharing loops in one parallel region – one cannot assume that the distribution of the iterations to threads is identical for both loops but the implicit barrier ensures that the results are available when needed.

OpenMP: Loop Construct

private *(list)*	
firstprivate *(list)*	
lastprivate *(list)*	
reduction *(operator:list)*	(C/C++)
reduction *({operator | intrinsic_procedure_name}:list)*	(Fortran)
ordered	
schedule *(kind[,chunk_size)]*	
nowait	

Clauses supported by the loop construct

OpenMP: Loop Construct

Schedule Clause:

The schedule clause describes how iterations of the given loop are divided among the threads in the team:

- The syntax is *schedule(kind,[chunk_size])*.
- The default schedule is implementation-dependent.
- The granularity of this workload is a chunk optionally specified by the parameter *chunk_size*.

OpenMP: Loop Construct

STATIC:

- Loops iterations are divided into pieces of size *chunk_size* and then statically assigned to threads in a round robin manner, in the order of the thread number.

- If *chunk_size* is not specified, the iterations are evenly and contiguously divided among the threads.

- It has the smallest overhead and is the default on many OpenMP-enabled compiler.

OpenMP: Loop Construct

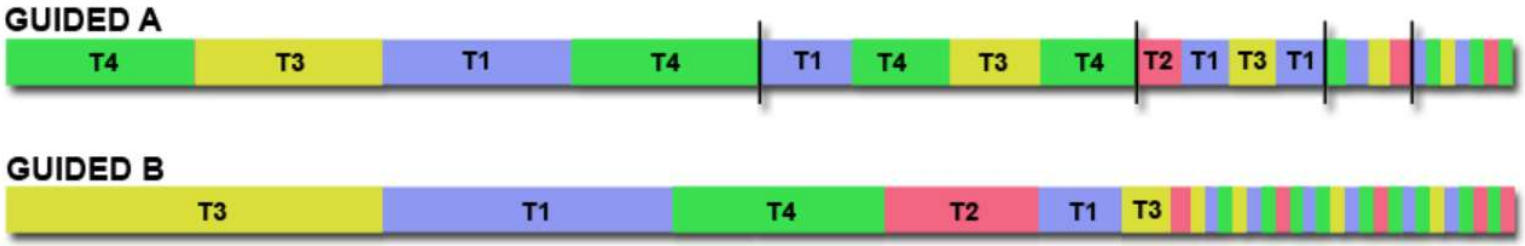
DYNAMIC:

- Loops iterations are divided into pieces of size *chunk_size* and then dynamically assigned to threads as the threads request them.
- When a thread is finished with one chunk, it is dynamically assigned another chunk until there is no more chunk to work on.
- The default *chunk_size* is one.
- Dynamic scheduling is suitable when the different iterations in a loop take different amounts of time to execute.

OpenMP: Loop Construct

GUIDED:

- The behavior is similar to *DYNAMIC* except that the block size decreases each time a parcel of work is given to a thread.
- When *chunk_size=1*, the size of each chunk is proportional to $\frac{the\ number\ of\ unassigned\ iterations}{the\ number\ of\ threads}$, decreasing over time down to *1*.
- *When chunk_size=k>1*, the size each chunk is determined the same way, with the restriction that the chunks do not contain fewer than *k* iterations, with a possible exception that the last chunk may have fewer than *k* iterations.

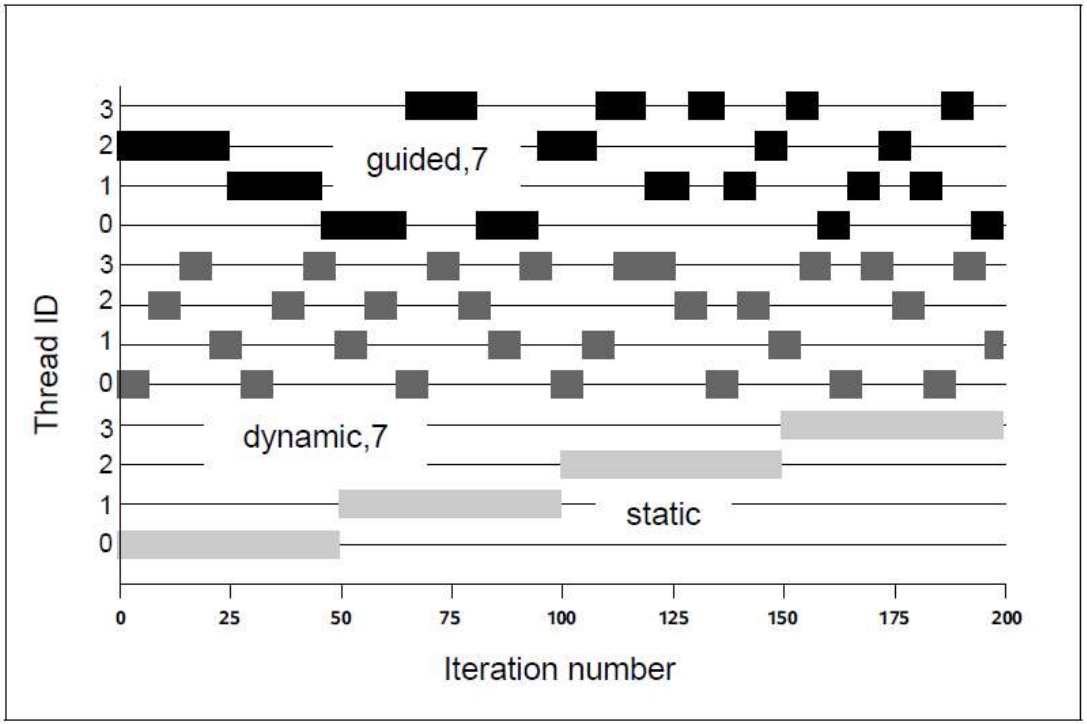


OpenMP: Loop Construct

RUNTIME:

- The decision is made at runtime through the use of the *OMP_SCHEDULE* environment variable

OpenMP: Loop Construct



OpenMP: Shared Clause

Shared Clause:

- The syntax is *shared(list)*.

- The shared clause is used to specify which variables will be shared among the threads executing the parallel region it is associated with.

- That is, there is only one unique instance of these variables.

- One crucial issue with shared variables is that multiple threads might attempt to simultaneously update the same memory location or that one thread might try to read from a location that another thread is updating.
  - Synchronization constructs must be used to prevent data races.

OpenMP: Shared Clause

```
#pragma omp parallel for shared(a)
for(int i=0;i<N;i++)
{
  a[i] += static_cast<double>(i);
}/*--- End of parallel for ---*/
```

- The code snippet illustrates the use of the shared clause.
- The array variable *a* is declared to be shared.
- Thus, all the threads are able to read and modify elements of *a*.
- Within the parallel loop, each thread will access the pre-existing values of those elements *a[i]*.
- After the parallel region, all the new values for elements of *a* will be available in main memory, where the master thread can access them.

OpenMP: Private Clause

Private Clause:

- The syntax is *private(list)*.
- When a variable is declared *private*, OpenMP replicates this variable and assigns its local copy to each thread in the team.
- The values of private variable are undefined on loop entry and exit.

OpenMP: Private Clause

```
int tid = 0;
//The value of tid is 0

#pragma omp parallel private(tid)
{
    //The value of tid is undefined
    tid = omp_get_thread_num();
    //The value of tid is defined
    std::cout << "Thread " << tid << std::endl;

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

//The value of tid is undefined
```

- The code snippet illustrates the use of the private clause.
- The variable *tid* is declared to be private.
- Thus, all the threads are assigned their local copies of *tid*.
- Within the parallel loop, each thread will assign the thread number to its local copy.
- After the parallel region, the value of *tid* is undefined.

OpenMP: Private Clause

```
#pragma omp parallel
{
    //The variable tid is declared here
    int tid = omp_get_thread_num();

    std::cout << "Thread " << tid << std::endl;

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

The code snippet illustrates how to avoid listing private variables by declaring them inside the parallel region.

OpenMP: First-Private Clause

First-Private Clause:

- The syntax is *firstprivate(list)*.
- Recall that private data is undefined on entry to the parallel construct where it is specified as private and this could pose a problem if we need to pre-initialize private variables with values that are available prior to the parallel construct.
- The initialization is performed by the initial thread prior to the execution of the parallel construct.
- When a variable is declared *firstprivate*, OpenMP initializes the local copy of each thread to the value of the master thread's copy.

OpenMP: First-Private Clause

```
int n = 255;
int vlen = 1024;
int a[vlen];

for(int i=0;i<vlen;i++) a[i] = -i-1;

int indx = 4;

#pragma omp parallel default(none) firstprivate(indx) shared(n,a)
{
   int tid = omp_get_thread_num();
   indx += n*tid;

   for(int i=indx ; i<indx+n ; i++)
       a[i] = tid + i;
}/*---End of parallel region---*/

std::cout << "After the parallel region:" << std::endl;

for(int i=0;i<vlen;i++)
    std::cout << "a["<<i<<"] = " << a[i] << std::endl;
```

In this code snippet, the local copies of the private variable *indx* are initialized to the value of *4* in all the threads in the team.

OpenMP: Last-Private Clause

Last-Private Clause:

- The syntax is *lastprivate(list)*.
- What if we need the value of a private variable after the parallel region?
- The *lastprivate* clause ensures that the last value of a data object is accessible after the corresponding construct has been completed.
- In a parallel program, however, we need to define what "*last*" means.
- In the case of its use with a work-shared loop, the data object will have the value from the iteration of the loop that would be last in a sequential execution.

OpenMP: Last-Private Clause

```
int n = 20;
int a;

#pragma omp parallel for shared(n) lastprivate(a)
for(int i=0;i<n;i++)
{
  a = i+1;

}/*---End of parallel for---*/

std::cout << "After the parallel region, a = " << a << std::endl;
```

- Without the OpenMP option flag, the code snippet will print out *a=20*.
- With the OpenMP option flag, the code snippet will also print out *a=20*, which is the same as the value of the last loop iteration of the sequential version.

OpenMP: Last-Private Clause

The use of the *lastprivate* clause can add overhead to OpenMP programs as the OpenMP runtime needs to keep track of which thread executes the last loop iteration:

- For a static workload distribution scheme, this is relatively lightweight.
- For a dynamic workload distribution scheme, this is more expensive.

OpenMP: Default Clause

Default Clause:

- The *default* clause is used to give a default data-sharing attribute.

- The syntax is as follows:
  - With *default(shared)*, all variables referenced in the construct are assigned the shared attribute unless explicitly specified otherwise.
  - With *default(none)*, the programmer is forced to specify a data-sharing attribute for each variable that appears in the construct.

- You are strongly encouraged to use *default(none)* to force yourself to think carefully and improve readability.

OpenMP: No-Wait Clause

No-Wait Clause:

- The *nowait* clause allows the programmer to fine-tune the performance of an OpenMP program.
- There is an implicit barrier at the end of every work-sharing construct.
- The *nowait* clause overrides this feature and the implicit barrier will be suppressed.
  - Thus, when threads reach the end of the construct, they will immediately proceed to perform other work.
  - Note that the implicit barrier at the end of a parallel region cannot be suppressed.

OpenMP: Reduction Clause

Reduction Clause:

- The syntax is *reduction(op:list)*.

- Reductions are so common that OpenMP allows the programmer to add a reduction clause to a parallel for.

- All we have to do is specify the reduction operation and the reduction variable, and OpenMP will take care of the details, such as storing partial sums in private variables and then adding the partial sums to the shared variable after the parallel for loop.

- A reduction variable is a shared variable although OpenMP will create a local copy of the original variable for each thread in the team and appropriately assign each of these local variables some initial value, which depends the type of the reduction operation (See the table on the next slide).

OpenMP: Reduction Clause

Operator	Initialization value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

OpenMP reduction operators for C and C++

OpenMP

The difference between *parallel*, *parallel for* and for is as follows:

- At the beginning, The program consists of a single thread a.k.a. the initial thread:
  - The *parallel* pragma tells the initial thread to fork a new team of threads for the associated parallel region, after which the current team joins back into one.
- The *for* pragma divides the work of the for-loop among the threads of the currently executing team:
  - It does not create threads.
  - it only divides the work amongst the threads of the currently executing team
- The *parallel for* pragma is a shorthand for two directives at once: *parallel* and *for*:
  - *The parallel* pragma creates a new team.
  - The *for* pragma distributes different portions of the loop to the threads in the team.
- If your program does not contain a parallel construct, there is no more than one thread.

OpenMP: Critical Section Construct

In OpenMP, a critical section can be denoted by the critical section construct which provides a means to ensure that multiple threads do not attempt to update shared data simultaneously.

```
#pragma omp critical [(name)]
structured block
```

- An optional name can be given to a critical construct and this name is global.
- When a thread encounters a critical section, it waits until no thread is executing inside the critical section with the same name.

OpenMP: Critical Section Construct

```
#pragma omp parallel default(none) shared(totalHits) private(seedVal)
{
    seedVal = omp_get_thread_num();//Each thread seeds with its own thread id.

    #pragma omp for
    for(int i=0 ; i<NUMITERATIONS ; i++)
    {
        double  x = (double) rand_r(&seedVal) / (double) RAND_MAX;
        double  y = (double) rand_r(&seedVal) / (double) RAND_MAX;

        double result = std::sqrt((x*x) + (y*y));

        if(result<1.0){
            #pragma omp critical(update_total_hits)
            {//critical section
                totalHits += 1.0;//check if the generated value is inside a unit circle.
            }
        }

    }
}
```

In this snippet, the program calculates the value of π using Monte Carlo simulation.

- Updates to the shared variable *totalHits* are protected by the critical pragma *update_total_hits*.
- However, speedup is poor with this approach.
- It is more efficient to use the reduction clause.

References

[1] *Barbara Chapman, Gabriele Jost, and Ruud van der Pas. 2007. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press.*

[2] *Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. 2001. Parallel programming in OpenMP. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.*