



Chapter 7 - Troubleshooting

Using OpenMP: Portable Shared Memory Parallel Programming

by Barbara Chapman, Gabriele Jost and Ruud van der Pas

The MIT Press © 2008 *Citation*

Recommend this title? yes no

◀ Previous

Next ▶

7.2 Common Misunderstandings and Frequent Errors

In this section we describe concepts, constructs, and programming techniques that are most likely to introduce (subtle) errors. We also provide recommendations on how these errors can be avoided.

7.2.1 Data Race Conditions

One of the biggest drawbacks of shared-memory parallel programming is that it might lead to the introduction of a certain type of bug that manifests itself through silent data corruption. To make matters worse, the runtime behavior of code with this kind of error is also not reproducible: if one executes the same erroneous program a second time, the problem might not show up.

OpenMP has several safety nets to help avoid this kind of bug. But **OpenMP** cannot prevent its introduction, since it is typically a result of faulty use of one of the directives. For example it may arise from the incorrect parallelization of a loop or an unprotected update of shared data. In this section we elaborate on this type of error, commonly known as a *data race condition*. This is sometimes also referred to simply as *data race* or *race condition*.

A data race condition exists when two threads may concurrently access the same shared variable between synchronization points, without holding any common locks and with at least one thread modifying the variable. The order of these accesses is nondeterministic. The thread reading the value might get the old value or the updated one, or some other erroneous value if the update requires more than one store operation. This usually leads to indeterministic behavior, with the program producing different results from run to run.

Consider the following for-loop:

```
for (i=0; i<n-1; i++)
    a[i] = a[i] + b[i];
```

The iterations of this loop are independent. The order in which they are executed does not affect the result. If, for example, we were to run the loop backwards, the results would be the same. What about the following loop, then?

```
for (i=0; i<n-1; i++)
    a[i] = a[i+1] + b[i];
```

The only change is that we now use $a[i+1]$ to update $a[i]$. This is called a *loop-carried dependence*: the loop iterations are dependent on each other. This minor change in the indexing destroys the parallelism. As written, the above loop can no longer be executed in parallel. The explanation is the following. When we update $a[i]$, we read the *old* value of $a[i+1]$. In the next iteration, $a[i+1]$ is then updated. Of course, this is a small change to the loop, but it is not a minor change in the algorithm at all.

If we do go ahead and execute the second loop in parallel, different threads will simultaneously execute the statement $a[i] = a[i+1] + b[i]$ for different values of i . Thus there arises the distinct possibility that for some value of i , the thread responsible for executing iteration $i+1$ does so *before* iteration i is executed. At this point we have an error in our program. When the statement is executed for iteration i , the *new* value of $a[i+1]$ is read, leading to an incorrect result. Unfortunately, there is no easy way to detect that this has occurred. This is an example of a data race condition introduced by the inappropriate parallelization of a loop.

Generally speaking, whether a data race condition affects a program's numerical results depends on various factors:

- Load on the system. The relative timing between the threads matters.
- Input data set. This might, for instance, lead to some load imbalance that affects the speed with which individual threads reach the incorrectly parallelized code.
- Number of threads used. In some cases the problem shows up only for a specific number of threads.

Therefore, a bug caused by a data race condition leads to nondeterministic behavior.^[1] A bug clearly exists in the parallel application, but one might not notice it during the test phase or even in production mode, for example because the number of threads used has been such that the problem was not exposed. Changing the number of threads or some other aspect of its execution could cause the bug to (re)surface.

To demonstrate the typical behavior of a data race condition, we have executed the second loop above in parallel, and observe the impact of its deliberate bug. We use integer values for vectors a and b . Our measure of correctness is the checksum of the relevant values of a , defined as

$$\text{checksum} \equiv \sum_{i=0}^{n-2} a_i.$$

The correct result is printed together with the actual value, computed after vector a is updated in the parallel loop. The program has been executed using 1, 2, 4, 32, and 48 threads. These runs are performed four times each to see whether the error surfaces. We have set n to a value of 64. The results are listed in [Figure 7.1](#). The second column gives the number of threads used. Column four contains the value of the checksum that is computed after the parallel loop. The last column has the correct value of the checksum. Note that we used the *same* binary program for all these runs. To change the number of threads, we modified the value for the `OMP_NUM_THREADS` environment variable only.

```
threads: 1 checksum 1953 correct value 1953
threads: 1 checksum 1953 correct value 1953
threads: 1 checksum 1953 correct value 1953
threads: 1 checksum 1953 correct value 1953

threads: 2 checksum 1953 correct value 1953
threads: 2 checksum 1953 correct value 1953
threads: 2 checksum 1953 correct value 1953
```

```

threads:  2 checksum  1953 correct value  1953

threads:  4 checksum  1905 correct value  1953
threads:  4 checksum  1905 correct value  1953
threads:  4 checksum  1953 correct value  1953
threads:  4 checksum  1937 correct value  1953

threads:  32 checksum  1525 correct value  1953
threads:  32 checksum  1473 correct value  1953
threads:  32 checksum  1489 correct value  1953
threads:  32 checksum  1513 correct value  1953

threads:  48 checksum  936 correct value   1953
threads:  48 checksum 1007 correct value   1953
threads:  48 checksum 887 correct value   1953
threads:  48 checksum 822 correct value   1953

```

Figure 7.1: Output from a loop with a data race condition—On a single thread the results are always correct, as is to be expected. Even on two threads the results are correct. Using four threads or more, the results are wrong, except in the third run. This demonstrates the non-deterministic behavior of this kind of code

As to be expected, the single thread results are correct. Surprisingly, the results on two threads are all correct, too. On four threads, the results are wrong in three out of the four cases. Two of the incorrect results are the same. For 32 and 48 threads none of the results are correct, and they are always different. This unpredictable runtime behavior is typical for a data race condition.

At the **OpenMP** level, data race conditions could also be introduced as the result of missing **private** clauses, missing **critical** regions, or incorrectly applied **nowait** clauses. Because of the lack of a barrier, the **master** construct can also introduce a data race if not used carefully. Other potential sources of data race conditions are the **SAVE** and **DATA** statements in Fortran and **static** or **extern** in C. Throughout this chapter, examples of these kinds of error are given.

7.2.2 Default Data-Sharing Attributes

We think it is good practice (although arguably a matter of personal preference and style) to explicitly specify the data-sharing attributes of variables and not rely on the default data-sharing attribute. Doing so reduces the possibility of errors. Moreover, for good performance, it is often best to minimize sharing of variables (see also [Sections 5.5.2](#) and [5.5.3](#)).

Probably the most important rule to watch out for is that, in many cases, variables are shared by default. This is illustrated in [Figure 7.2](#). The variable **Xshared** is shared by default. If multiple threads execute the parallel region, they all try to simultaneously write a different value into the same variable **Xshared**. This is an example of a data race.

```

#pragma omp parallel
{
    int Xlocal = omp_get_thread_num();
    Xshared = omp_get_thread_num(); /*-- Data race --*/

    printf("Xlocal = %d Xshared = %d\n", Xlocal, Xshared);
} /*-- End of parallel region --*/

```

Figure 7.2: Example of implied sharing—By default, variable **Xshared** is shared. The assignment to **Xshared** causes a data race: if multiple threads are used, they simultaneously store a different value in the same variable

Errors like this can easily sneak in. The code example in [Figure 7.3](#) shows a data race condition introduced by the fact that variable **x** is shared by default, rather than having been explicitly declared **private**.

```

void compute(int n)
{
    int i;
    double h, x, sum;

    h = 1.0/(double) n;
    sum = 0.0;
    #pragma omp for reduction(+:sum) shared(h)
    for (i=1; i <= n; i++) {
        x = h * ((double)i - 0.5);
        sum += (1.0 / (1.0 + x*x));
    }
    pi = h * sum;
}

```

Figure 7.3: Data Race Condition due to missing private declaration—The variables **i** and **x** are not explicitly declared as private. Variable **i** is implicitly declared to be private according to the **OpenMP** default data-sharing rules. Variable **x** is shared by default, it is written to by multiple threads, leading to a race condition

In the example in [Figure 7.4](#), multiple threads update elements of the shared data structure **material**. In order to ensure deterministic results, the update loop over **i** in the parallel region should be enclosed by a critical region.

```

integer ind (1:numt)
....
allocate (material(1:numt), istat)
do i = 1, numt
    material(i) = 0.
    material(i)%x = vall
end do
...
!$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(material,plq,ind, numt)
do n = 1, numt
    mid = ind(i)

```

```

    qm = material(mid)%x * plq(n)
    do i = 1, 4
        material(mid)%x = material(mid)%x + qm * px(i)
    end do
end do
!$OMP END PARALLEL DO

```

Figure 7.4: Data race condition due to missing critical region—The value of the private variable `mid` could be the same for different threads, causing the shared variable `material(mid)%x` to be updated by multiple threads at the same time. In order to guarantee predictable output, the `i`-loop in the parallel region must be enclosed by a `critical` region

A subtle distinction exists between Fortran and C/C++ regarding the default data-sharing rules. In Fortran, loop index variables are private by default. In C, the index variables of the parallel for-loop are private by default, but this does not extend to the index variables of loops at a deeper nesting level. An example is shown in [Figure 7.5](#). Loop variable `j` is shared by default, resulting in undefined runtime behavior. It is another example of a data race.

```

int i, j;
#pragma omp parallel for
for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
        a[i][j] = compute(i,j);
    }

```

Figure 7.5: Example of a loop variable that is implicitly shared— Loop variable `i` is private by default, but this is not the case for `j`: it is shared by default. This results in undefined runtime behavior

The error is easy to avoid through an explicit `private` clause for the loop variables, or by using a local variable instead. For example loop variable `x` is private by default if used as follows: `for (int k=0; ...)`.

Our recommendation to be specific on the data-sharing attributes is also language dependent. Fortran has more need of this approach than does C/C++. The main reason is that in Fortran, variables cannot be declared locally in a code block, such as a loop.

7.2.3 Values of Private Variables

One of the most important decisions to be made when developing a shared memory parallel program is what data should be shared between threads and what should be local to a thread.

Whenever each thread requires its own "local" copy of a variable in a calculation, this variable needs to be listed in a private clause.

One can avoid errors that result from not adding variables to the private clause. The key is to use the `default (none)` clause, thereby forcing all data sharing attributes to be specified explicitly rather than by relying on defaults. Additionally, one should keep in mind two points:

- The value of the private copy is uninitialized on entry to the parallel region.
- The value of the original variable is undefined on exit from the parallel region.^[2]

An example of using an uninitialized private variable is given in [Figure 7.6](#). The programmer uses variable `b` without realizing that it does not have an initial value within the parallel loop, despite the fact it has a value prior to the loop. As a result, the variable is undefined and can take any value. If the intent is to initialize `b` with the value it had before the parallel region, then the `firstprivate` clause achieves exactly this. Alternatively, it can be made `shared`, since it is not modified in the parallel loop. There is also a problem with variables `a` and `b`, both of which are undefined after the parallel loop. The `lastprivate` clause is a convenient feature to make the last value of a private list item available after the parallel region terminates, so this problem can easily be avoided as well. As explained in [Section 4.5.3](#) the interpretation of "last" depends on the construct. There is also a (modest) performance penalty when using this construct. A correct version of this code is shown in [Figure 7.7](#).

```

void main ()
{
    .....
    #pragma omp parallel for private(i,a,b)
    for (i=0; i<n; i++)
    {
        b++;
        a = b+i;
    } /*-- End of parallel for --*/
    c = a + b;
    .....
}

```

Figure 7.6: Incorrect use of the private clause—This code has two problems. First, variable `b` is used but not initialized within the parallel loop. Second, variables `a` and `b` should not be used after the parallel loop. The values after the parallel loop are undefined and therefore implementation dependent

```

void main ()
{
    .....
    #pragma omp parallel for private(i), firstprivate(b) \
        lastprivate(a,b)
    for (i=0; i<n; i++)
    {
        b++;
        a = b+i;
    } /*-- End of parallel for --*/
    c = a + b;
    .....
}

```

Figure 7.7: Corrected version using firstprivate and lastprivate variables— This is the correct version of the code in [Figure 7.6](#)

7.2.4 Problems with the Master Construct

Whenever a piece of work within a parallel region needs to be performed by only one thread, either the `single` or the `master` construct can be used. For many tasks involving reading, writing or general control, the master construct is the natural choice. Unfortunately the `master` construct does not have an implied barrier. [Figure 7.8](#) shows a simple, but erroneous, example of its use. Not only is there no synchronization, but there is also no guaranteed flushing of any modified data upon completion of this construct (see also [Section 7.3.1](#)). As a result, a potential problem arises if data is read in, initialized, or updated in this construct and subsequently used by other threads. For correct results, a barrier *must* be inserted before any accesses to the variables modified in the master construct. In many cases, the simplest solution is to use the `single` construct, because it implies a `barrier` at the end of the construct.

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

void main()
{
    int Xinit, Xlocal;

    #pragma omp parallel shared(Xinit) private(Xlocal)
    {
        #pragma omp master
        {Xinit = 10;}

        Xlocal = Xinit; /*-- Xinit might not be available yet --*/

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

Figure 7.8: Incorrect use of the master construct—This code fragment implicitly assumes that variable `xinit` is available to the threads after initialization. This is incorrect. The master thread might not have executed the assignment when another thread reaches it, or the variable might not have been flushed to memory

7.2.5 Assumptions about Work Scheduling

Earlier we saw (for example, in [Section 5.4.1](#)) that the `nowait` clause can help to increase performance by removing unnecessary barriers at the end of work-sharing constructs for example. In such a case, however, care must be taken not to rely on assumptions about which thread executes which loop iterations. An example of such an incorrect assumption is shown in [Figure 7.9](#). If the loop bound `n` is not a multiple of the number of threads, then, according to the [OpenMP 2.5](#) specifications, there are several compliant algorithms for distributing the remaining iterations. As of [OpenMP 2.5](#) there is no requirement that the same algorithm has to be used in different loops. A compiler may choose to employ different strategies for dealing with remainder iterations in order to take advantage of memory alignment. Therefore, the second loop in [Fig. 7.9](#) might read values of array `b` that have not yet been written to in the first loop. This action, however, results in a data race condition.

```
#pragma omp parallel
{
    #pragma omp for schedule(static) nowait
    for (i=0; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
    #pragma omp for schedule(static) nowait
    for (i=0; i<n; i++)
        z[i] = sqrt(b[i]);
}
```

Figure 7.9: Example of incorrect assumptions about work scheduling in the OpenMP 2.5 specifications—The `nowait` clause might potentially introduce a data race condition, even with static work scheduling, if `n` is not a multiple of the number of threads

7.2.6 Invalid Nesting of Directives

In many programs it may seem natural to further subdivide the work that has been handed out to the threads in a team. For example, if a number of different `sections` have been defined, one or more of them might contain enough work for multiple threads. Indeed, the computation can be further distributed in this way. Before doing so, however, the programmer must take care to create a new team of threads to carry out this work. The way to do so is to introduce a new parallel region. A common cause of error is to nest work-sharing directives in a program, without providing a new parallel region. An example of such incorrect use is given in [Figure 7.10](#).^[3]

```
#pragma omp parallel shared(n,a,b)
{
    #pragma omp for
    for (int i=0; i<n; i++)
    {
        a[i] = i + 1;
        #pragma omp for // WRONG - Needs a new parallel region
        for (int j=0; j<n; j++)
            b[i][j] = a[i];
    }
} /*-- End of parallel region --*/
```

Figure 7.10: Example of incorrectly nested directives—Nested parallelism is implemented at the level of parallel regions, not work-sharing constructs, as erroneously attempted in this code fragment

However, nested work-sharing constructs without nested parallel regions cannot work. The threads in the current team have already been assigned their portion of the work by the existing work-sharing directive, and no idle threads are waiting for more work. The new parallel region must supply the new threads. The correct code is shown in [Figure 7.11](#).

```
#pragma omp parallel shared(n,a,b)
{
    #pragma omp for
    for (int i=0; i<n; i++)
    {
```

```

        a[i] = i + 1;
        #pragma omp parallel for // Okay - This is a parallel region
        for (int j=0; j<n; j++)
            b[i][j] = a[i];
    }
} /*-- End of parallel region --*/

```

Figure 7.11: Example of correctly nested directives — This is correct use of nested parallelism. This code fragment has two nested parallel regions

This kind of error may also inadvertently occur if orphan directives are used.

There are other ways in which invalid nesting of directives can lead to unexpected program behavior. The following are examples of the erroneous use of directives:

- A **barrier** is in a work-sharing sharing construct, a **critical** section, or a master construct.
- A **master** construct is within a work-sharing construct.
- An **ordered** directive is within a **critical** section.
- The **barrier** is not executed by all threads in the team.

Another example of an error in the use of a **barrier** is illustrated in the code fragment in [Figure 7.12](#). According to the [OpenMP 2.5](#) specifications, one of the restrictions on the **barrier** is as follows: "Each barrier region must be encountered by all threads in a team, or none at all" (Section 2.7.3 in [2]). This rule is violated in the code fragment shown here.

```

#pragma omp parallel // Incorrect use of the barrier
{
    if (omp_get_thread_num() == 0)
    {
        .....
        #pragma omp barrier
    }
    else
    {
        .....
        #pragma omp barrier
    }
} /*-- End of parallel region --*/

```

Figure 7.12: Illegal use of the barrier — The **barrier** is not encountered by all threads in the team, and therefore this is an illegal [OpenMP](#) program. The runtime behavior is undefined

The example in [Figure 7.12](#) could have an interesting side effect. The pragma translates to a function call that implements the barrier functionality. An optimizing compiler might potentially detect that in this case the barrier function can be called unconditionally, effectively executing the following code fragment.

```

if (omp_get_thread_num() == 0)
{ ..... }
else
{ ..... }
#pragma omp barrier

```

This use of the barrier no longer violates the specifications. The question is, of course, whether a compiler is able to perform this transformation. This would require it to be able to analyze and transform parallel code, an area of active research.

Another illegal use of a barrier in a work-sharing construct is demonstrated in the example in [Figure 7.22](#) in [Section 7.3.5](#) on [page 269](#). This is erroneous because all threads in the team must encounter the barrier.

7.2.7 Subtle Errors in the Use of Directives

Errors in the directives can have subtle undesired effects. [Section 3.1](#) describes the [OpenMP](#) directive syntax and cautions that not all errors in the directives are detected at compile time. In the example in [Figure 7.13](#), the continuation line with the private declaration of variables `i` and `c1` contains an extra exclamation mark. As a result, the compiler no longer recognizes this as an [OpenMP](#) directives, and the **private** clause is ignored. Following the default data-sharing rules in [OpenMP](#), loop variable `i` is private. This is exactly what was intended, but by virtue of these rules variable `c1` is shared, thereby introducing a data race. This error is caught by the compiler if the **default (none)** clause is used.

```

subroutine dot(n, a, b, c)
implicit none

integer(kind=4):: n
real      (kind=8):: a(1:n), b(1:n), c, c1
integer   :: i
!$OMP PARALLEL SHARED(n,a,b,c)
!!$OMP& PRIVATE(i,c1)
!$OMP DO
do i = 1, n
    c1 = c1 + b(i)*a(i)
end do
!$OMP END DO
!$OMP CRITICAL
c = c + c1
!$OMP END CRITICAL
!$OMP END PARALLEL
return
end

```

Figure 7.13: Example of an error in the OpenMP directive — The continuation contains an extra exclamation mark. As a result the compiler ignores the **private** clause. Loop variable `i` is private by default, as intended, but variable `c1` is shared. This introduces a data race. If the

default (none) clause is used, the compiler catches this kind of error

In C, curly brackets are used to define a parallel region that spans more than a single statement. If these brackets are not placed correctly or are left out entirely, unexpected runtime behavior may occur, ranging from a reduced speedup to an incorrect result. The code fragment in [Figure 7.14](#) illustrates such a situation. In the first parallel region, both functions `work1` and `work2` are executed in parallel, but in the second parallel region, only function `work1` is.

```
main()
{
#pragma omp parallel
{
    work1(); /*-- Executed in parallel --*/
    work2(); /*-- Executed in parallel --*/
}

#pragma omp parallel
{
    work1(); /*-- Executed in parallel --*/
    work2(); /*-- Executed sequentially --*/
}
}
```

Figure 7.14: Example of the impact of curly brackets on parallel execution— It is very likely an error was made in the definition of the second parallel region: function `work2` is executed by the master thread only

7.2.8 Hidden Side Effects, or the Need for Thread Safety

Using libraries can potentially introduce side effects if they are not *thread-safe*. The terminology *thread-safe* refers to the situation that, in a multithreaded program, the same functions and the same resources may be accessed concurrently by multiple flows of control. The use of global data is not thread-safe. For example, library routines for multithreaded programs that make use of global data must be written such that shared data is protected from concurrent writes.

The code in [Figure 7.15](#) shows a global variable being incremented every time a library routine is executed. If the library is called from within a parallel region, multiple threads may try to access variable `icount` concurrently. Because the increment `++` is not an atomic operation, it can be interrupted before completion. This constitutes a data race condition and might yield indeterministic results, as discussed in [Section 7.2.1](#). In order to make the routine thread-safe, access to variable `icount` has to be protected by a lock, an atomic construct or a critical section.

```
int icount;

void lib_func()
{
    icount++;
    do_lib_work();
}

main ()
{
    #pragma omp parallel
    {
        lib_func();
    } /*-- End of parallel region -- */
}
```

Figure 7.15: Example of a function call that is not thread-safe— The library keeps track of how often its routines are called by incrementing a global counter. If executed by multiple threads within a parallel region, all threads read and modify the shared counter variable, leading to a race condition

Library routines written in Fortran should be built such that all local data is allocated on the stack for thread-safety. This could be a problem in cases where the `SAVE` statement is used. Originally introduced for sequential processing, the `SAVE` statement has an unpleasant side effect in a parallel context.

According to Section 2.8.1.2 in the [OpenMP 2.5 specifications](#) [2], a local variable that is used in a `SAVE` statement changes from private to shared. If multiple threads update such a variable, the risk of a data race arises.

An example of this kind of use can be found in the linear algebra library package LAPACK [13].^[4] A number of its auxiliary library routines contain `SAVE` statements on some or all of the variables. An example is routine `dlamch`, which is called to determine double precision machine parameters. A number of variables are listed in the `SAVE` statement, for example to indicate the first usage of the routine to perform certain initializations. A code snippet is provided in [Figure 7.16](#).

```
DOUBLE PRECISION FUNCTION DLAMCH(CMACH)
* .. Local Scalars ..
LOGICAL FIRST
* .. Save statements ..
SAVE FIRST
* .. Data statements ..
DATA FIRST / .TRUE. /
* ..
...
IF(FIRST) THEN
    FIRST = .FALSE.
    CALL DLAMC2(BETA, IT, LRND, EPS, IMIN, RMIN, IMAX, RMAX)
    ...
ENDIF
```

Figure 7.16: Example of a Fortran library call that is not thread-safe—The library routine performs certain initializations the first time it is called. When it is called from within a `parallel` region, access to variable `FIRST` has to be protected to avoid data race conditions

One possibility to make such a routine thread-safe is to serialize access to the shared data. Most vendors provide thread-safe implementations of important libraries, such as LAPACK. It is good practice, however, to check the documentation when in doubt.

Another source of hidden side effects is shared class objects and methods in C++. When class objects with methods defined on them are used as shared variables within [OpenMP](#) parallel regions, race conditions can result. An example is shown in [Figure 7.17](#). In order to make the code thread-safe, the invocation of the method should be enclosed in a critical region, or the update of the shared variable within the method should be enclosed by a critical region.

```
class anInt {
public:
    int x;
    anInt(int i = 0){ x = i; };
    void addInt (int y){ x = x + y; }
};
main()
{
    anInt a(10);
    #pragma omp parallel
    {
        a.addInt(5);
    }
}
```

Figure 7.17: Example of unsafe use of a shared C++ object—When executed on 2 threads, the expected result is 20. However, data race conditions may yield indeterministic results


[1]A data race also implies that false sharing occurs, possibly degrading performance. See also [Section 5.5.2 on page 153](#).

[2]This will probably not be true in [OpenMP 3.0](#).

[3]Compilers will probably flag this kind of incorrect use and issue a warning.

[4]The Fortran source code can be downloaded at [14].

 [Previous](#)

[Next](#) 

Use of content on this site is expressly subject to the restrictions set forth in the [Membership Agreement](#).
 Books24x7 and Referenceware are registered trademarks of Books24x7, Inc.
 Copyright © 1999-2008 Books24x7, Inc. - [Feedback](#) | [Privacy Policy \(updated 03/2005\)](#)
[RSS feed](#)