Multicore and Multiprocessor Systems: Part III

Open Multi-Processing (OpenMP)

This is OpenMP: The Mission

Mission

"The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer." [7]

[7] The Mission statement from `http://www.openmp.org/about/about-us/`

Open Multi-Processing (OpenMP)

This is OpenMP: The Mission

Mission

> *"The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer."* [7]

The OpenMP Architecture Review Board (ARB)

The ARB is a non-profit enterprise owning the OpenMP brand and responsible for overseeing, producing and approving the OpenMP standards.

[7] The Mission statement from `http://www.openmp.org/about/about-us/`

Open Multi-Processing (OpenMP)

This is OpenMP: The Contributors

Permanent Members of the ARB: **(status: April 23, 2017)**[8]

- AMD (Greg Stoner)
- ARM (Chris Adeniyi-Jones)
- Cray (Luiz DeRose)
- Fujitsu (Eiji Yamanaka)
- HP (Sujoy Saraswati)
- IBM (Kelvin Li)
- Intel (Xinmin Tian)
- Micron (Kirby Collins)
- NEC (Kazuhiro Kusano)
- NVIDIA (Jeff Larkin)
- Oracle Corporation (-TBD-)
- redhat (Torvald Riegel)
- Texas Instruments (Eric Stotzer)

[8] http://www.openmp.org/about/members/

Open Multi-Processing (OpenMP)

This is OpenMP: The API standard

History

Oct. 1997	OpenMP 1.0 for Fortran,
Oct. 1998	OpenMP 1.0 for C/C++,
Nov. 2000	OpenMP 2.0 for Fortran,
March 2002	OpenMP 2.0 for C/C++,
May 2005	OpenMP 2.5 (first joint Fortran/C/C++ version),
May 2008	OpenMP 3.0,
Sept. 2011	OpenMP 3.1,
July 2013	OpenMP 4.0,
Nov. 2015	OpenMP 4.5 (current standard).

Open Multi-Processing (OpenMP)

What OpenMP can do for us

- **Easy shared memory parallel adaption of existing sequential codes**

Open Multi-Processing (OpenMP)

What OpenMP can do for us

- **Easy shared memory parallel adaption of existing sequential codes**
- **Easy preservation of sequential implementations**

Open Multi-Processing (OpenMP)

What OpenMP can do for us

- **Easy shared memory parallel adaption of existing sequential codes**
- **Easy preservation of sequential implementations**
- **Easy porting to different platforms and compilers**

Open Multi-Processing (OpenMP)

What OpenMP can do for us

- **Easy shared memory parallel adaption of existing sequential codes**
- **Easy preservation of sequential implementations**
- **Easy porting to different platforms and compilers**
- **Parallel implementation of only fragments**

Open Multi-Processing (OpenMP)

What OpenMP can do for us

- **Easy shared memory parallel adaption of existing sequential codes**
- **Easy preservation of sequential implementations**
- **Easy porting to different platforms and compilers**
- **Parallel implementation of only fragments**
- **No extra runtime environment**

Open Multi-Processing (OpenMP)

What OpenMP can do for us

- Easy shared memory parallel adaption of existing sequential codes
- Easy preservation of sequential implementations
- Easy porting to different platforms and compilers
- Parallel implementation of only fragments
- No extra runtime environment
- Easy to learn and apply

Open Multi-Processing (OpenMP)

What OpenMP is NOT for!

- **Distributed memory parallel systems (by itself)**

Open Multi-Processing (OpenMP)

What OpenMP is NOT for!

- **Distributed memory parallel systems (by itself)**
- **Most efficient use of shared memory systems**

Open Multi-Processing (OpenMP)

What OpenMP is NOT for!

- **Distributed memory parallel systems (by itself)**
- **Most efficient use of shared memory systems**
- **Automatic checking for data dependencies, data conflicts, race conditions, or deadlocks**

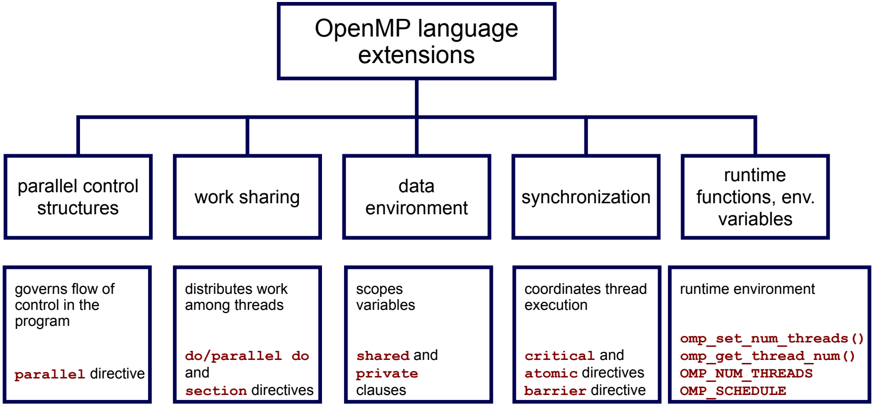
Open Multi-Processing (OpenMP)

What OpenMP is NOT for!

- **Distributed memory parallel systems (by itself)**
- **Most efficient use of shared memory systems**
- **Automatic checking for data dependencies, data conflicts, race conditions, or deadlocks**
- **Automatic synchronization of input and output**

Open Multi-Processing (OpenMP)

The Structure of the Standard



OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

Figure: Classification of the OpenMP extensions by tasks of the elements (Image Source: `https://commons.wikimedia.org/wiki/File:OpenMP_language_extensions.svg`).

Open Multi-Processing (OpenMP)

The Structure of the Standard

The standard divides the extensions into four classes:

1. **Directives:**
   Basic control structures that initialize/end the parallel environments

Open Multi-Processing (OpenMP)

The Structure of the Standard

The standard divides the extensions into four classes:

1. **Directives:**

   Basic control structures that initialize/end the parallel environments

2. **Clauses:**

   Fine tuning parameters added to the directives.

Open Multi-Processing (OpenMP)

The Structure of the Standard

The standard divides the extensions into four classes:

1. **Directives:**
   Basic control structures that initialize/end the parallel environments
2. **Clauses:**
   Fine tuning parameters added to the directives.
3. **Environment Variables:**
   Variables in the calling shell used to control the parallel environment without recompilation.

Open Multi-Processing (OpenMP)

The Structure of the Standard

The standard divides the extensions into four classes:

1. **Directives:**
   Basic control structures that initialize/end the parallel environments
2. **Clauses:**
   Fine tuning parameters added to the directives.
3. **Environment Variables:**
   Variables in the calling shell used to control the parallel environment without recompilation.
4. **Runtime Library Routines:**
   Runtime usable functions for the determination and modification of parameters of the parallel environment.

Open Multi-Processing (OpenMP)

OpenMP directives

The #pragma directive was introduced in C89 as the universal method for extending the space of directives. It was further standardized in C99, where especially the token STDC was reserved for standard C extensions.

Example (standard C #pragma usage)

In part 1 of the Scientific Computing lecture we have seen the floating point environment for, e.g., checking the exception flags in IEEE arithmetic:

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
/* starting here the compiler needs to assume we are accessing the
floating point status and mode registers*/
```

Open Multi-Processing (OpenMP)

OpenMP directives

OpenMP is an extension in the sense of C89 and enabled by the

```
#pragma omp
```

preprocessor directive. It applies to the succeeding structural code block.

Open Multi-Processing (OpenMP)

OpenMP directives

OpenMP is an extension in the sense of C89 and enabled by the

```
#pragma omp
```

preprocessor directive. It applies to the succeeding structural code block.

Compilers that do not know the `omp` pragma simply ignore it. The following switches enable OpenMP support for your code:

GNU GCC	`-fopenmp`
Intel ICC	`-qopenmp`
LLVM CLANG	`-fopenmp`
IBM XLC	`-qsmp`
PGI	`-mp`

Otherwise the `omp` pragmas are ignored and the sequential code version is compiled.

Open Multi-Processing (OpenMP)

OpenMP directives

OpenMP is an extension in the sense of C89 and enabled by the

```
#pragma omp
```

preprocessor directive. It applies to the succeeding structural code block.

Compilers that do not know the `omp` pragma simply ignore it. The following switches enable OpenMP support for your code:

GNU GCC	`-fopenmp`
Intel ICC	`-qopenmp`
LLVM CLANG	`-fopenmp`
IBM XLC	`-qsmp`
PGI	`-mp`

Otherwise the `omp` pragmas are ignored and the sequential code version is compiled.

A list of compilers supporting OpenMP can be found at
`http://www.openmp.org/resources/openmp-compilers/`

Open Multi-Processing (OpenMP)

OpenMP directives: Parallel

The `parallel` construct initializes a group of threads and starts parallel execution:

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

The clauses can be used to influence the behavior of the parallel execution. They will be explained later.

Available clauses for `parallel`:

- `if(scalar expression)`
- `num_threads(integer expression)`
- `default(shared| none)`
- `private(list)`
- `firstprivate(list)`
- `shared(list)`
- `copyin(list)`
- `reduction(operation:list)`

Open Multi-Processing (OpenMP)

OpenMP directives

Example (A minimal OpenMP parallel “hello world” program)

```
#include <stdio.h>

int main(void)
{
#pragma omp parallel
    printf("Hello,␣world.\n");
  return 0;
}
```

The example automatically lets OpenMP tune the number of threads used to the number of available processors. Afterward the parallel execution environment is started and all threads execute the `printf` statement.

Open Multi-Processing (OpenMP)

OpenMP directives: Loop

The `loop` construct specifies that the iterations of the loop should be distributed among the active threads.

```
#pragma omp for [clause[[,]clause]...]
  for loops
```

The `for`-loop construct needs to be used inside a structured code block of `parallel` construct.

Available clauses for `for`:

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `reduction(operator:list)`
- `schedule(kind[,chunk_size])`
- `collapse(n)`
- `ordered`
- `nowait`

Open Multi-Processing (OpenMP)

OpenMP directives: Parallel Loop

Since often the `parallel` environment is used to introduce a `for`-loop construction only, a shortcut `parallel for` exists for this special task

```
#pragma omp parallel for [clause[[,] clause]...]
```

With the exception of the `nowait` clause all clauses accepted by `parallel` and `for` can be used with `parallel for` with the identically same behaviors and restrictions.

Open Multi-Processing (OpenMP)

OpenMP directives: Parallel Loop

Example (OpenMP parallel vector triad)

```
double triad(double *a, double *b, double *c, double *d, int length){
  int i,j;
  const int repeat=100;
  double start, end;

  get_walltime(&start);
  for (j=0; j<repeat; j++){
#pragma omp parallel for
    for (i=0 ; i<length; i++){
      a[i]=b[i] + c[i] * d[i];
    } /*end of parallel section*/
  }
  get_walltime(&end);
  return repeat*length*2.0 / ((end-start) * 1.0e6); /* return MFLOPS */
}
```

Open Multi-Processing (OpenMP)

OpenMP directives: Parallel Loop

Example (OpenMP parallel vector triad)

```
double triad(double *a, double *b, double *c, double *d, int length){
  int i,j;
  const int repeat=100;
  double start, end;

  get_walltime(&start);
  for (j=0; j<repeat; j++){
#pragma omp parallel for
    for (i=0 ; i<length; i++){
      a[i]=b[i] + c[i] * d[i];
    } /*end of parallel section*/
  }
  get_walltime(&end);
  return repeat*length*2.0 / ((end-start) * 1.0e6); /* return MFLOPS */
}
```

Note that loop counters are protected automatically.

OpenMP directives: Sections

When different tasks are to be distributed among the encountering team of threads the `sections` construct can be used

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

Available clauses for `sections`:

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `reduction(operator:list)`
- `nowait`

Open Multi-Processing (OpenMP)

OpenMP directives: Parallel Sections

Analogous to the `for` construct, also `sections` can be used only inside a `parallel` construct. The `parallel sections` construct merges them for easier use

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

Available clauses are those available for `parallel` and `sections` with the exception of `nowait`, as in the case of `for`.

Open Multi-Processing (OpenMP) I

OpenMP directives: Parallel Sections

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N     50

int main (int argc, char *argv[]) {
  int i, nthrd, tid;
  float a[N], b[N], c[N], d[N];

  /* Some initializations */
  for (i=0; i<N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 42.0;
    c[i] = d[i] = 0.0;
  }
  /* Start 2 threads */
#pragma omp parallel shared(a,b,c,d,nthrd) private(i,tid) num_threads(2)
{
  tid = omp_get_thread_num();
  if (tid == 0) {
    nthrd = omp_get_num_threads();
    printf("Number␣of␣threads␣=␣%d\n", nthrd);
  }
  printf("Thread␣%d␣starting...\n",tid);
```

Open Multi-Processing (OpenMP) II

OpenMP directives: Parallel Sections

```
#pragma omp sections
    {
#pragma omp section
      {
        printf("Thread␣%d␣doing␣section␣1\n",tid);
        for (i=0; i<N; i++) {
          c[i] = a[i] + b[i];
        }
        sleep(tid+2);  /* Delay the thread for a few seconds */
      } /* End of first section */

#pragma omp section
      {
        printf("Thread␣%d␣doing␣section␣2\n",tid);
        for (i=0; i<N; i++) {
          d[i] = a[i] * b[i];
        }
        sleep(tid+2);   /* Delay the thread for a few seconds */
      } /* End of second section */
    }  /* end of sections */
    printf("Thread␣%d␣done.\n",tid);
  }  /* end of omp parallel */

/* Print the results */
  printf("c:␣␣");
```

Open Multi-Processing (OpenMP) III

OpenMP directives: Parallel Sections

```
  for (i=0; i<N; i++) {
    printf("%.2f␣", c[i]);
  }
  printf("\n\nd:␣␣");
  for (i=0; i<N; i++) {
    printf("%.2f␣", d[i]);
  }
  printf("\n");
  exit(0);
}
```

Open Multi-Processing (OpenMP)

OpenMP directives: Single

A construct that makes sure that a structured code block is executed by only one thread in a team of threads is given by the `single` directive.

```
#pragma omp single [clause[[,] clause]...]
```

Available clauses for the `single` construct are:

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `nowait`

Open Multi-Processing (OpenMP)

OpenMP directives: Single

Example (OpenMP 4.5 Example 1.11 – single1.c)

```
#include <stdio.h>

void work1() {}
void work2() {}
void main()
{
#pragma omp parallel
  {
  #pragma omp single
      printf("Beginning␣work1.\n");
  work1();
  #pragma omp single
      printf("Finishing␣work1.\n");
  #pragma omp single nowait
      printf("Finished␣work1␣and␣beginning␣work2.\n");
  work2();
  }
}
```

Open Multi-Processing (OpenMP)

OpenMP directives: Master

The `master` construct specifies a structured block that is executed by the master thread of the team.

```
#pragma omp master
```

The following structured block is only executed by the master thread of the parallel team. There is no synchronization on entry or on exit with the other threads.

Open Multi-Processing (OpenMP)

OpenMP directives: Master

Example (OpenMP 4.5 Example 1.13 – master1.c)

```
void master_example( float* x, float* xold, int n, float tol ){
  int c = 0, i, toobig;   loat error, y;
  #pragma omp parallel
  {
    do{
      #pragma omp for private(i)
      for( i = 1; i < n-1; ++i ){ xold[i] = x[i]; }
      #pragma omp single
      toobig = 0;
      #pragma omp for private(i,y,error) reduction(+:toobig)
      for( i = 1; i < n-1; ++i ){
        y = x[i];
        x[i] = average( xold[i-1], x[i], xold[i+1] );
        error = y - x[i];
        if( error > tol || error < -tol ) ++toobig;
      }
      #pragma omp master
      { printf( "iteration␣%d,␣toobig=%d\n", ++c, toobig ); }
    }while( toobig > 0 );
  }
}
```

Open Multi-Processing (OpenMP)

OpenMP directives: Tasks

The OpenMP task construct (introduced with OpenMP 3.0) allows to parallelize irregular algorithms. The task construct inserts a piece of work into a thread pool running in the background.

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

Available clauses for `task` (not complete):

- `if(scalar-expression)`
- `final(scalar-expression)`
- `default(shared | none)`
- `private(list)`
- `firstprivate(list)`
- `shared(list)`
- `priority(priority-value)`

Open Multi-Processing (OpenMP)

OpenMP directives: Tasks

Using the `taskwait` directive:

```
#pragma omp taskwait
```

one can wait for the completion of all previously created tasks at any position inside a parallel region in order to synchronize the parallel execution.

Due to the fact that tasks are running in the background they are mostly emitted by a single thread or a sequential code block. Therefore, mostly the `single` and `master` directives are used.

Tasks have to be defined inside an OpenMP `parallel` region. The end of the parallel region, unless it is used with the `nowait` clause, is and implicit synchronization point and the program waits until all tasks created inside the parallel region are finished.

Open Multi-Processing (OpenMP)

OpenMP directives: Task Dependencies

The support to describe data dependencies between tasks is one of the most beneficial features of the **OpenMP 4** standard. For scientific computing this means that algorithms relying on dependency-graphs can be parallelized without using other third-party code or libraries.

> *"Although we expect to see DAG-based models widely adopted, changes in other parts of the software ecosystem will inevitably affect the way that that model is implemented. The appearance of DAG scheduling constructs in the OpenMP 4.0 standard offers a particularly important example of this point. [...] However, the inclusion of DAG scheduling constructs in the OpenMP standard, along with the rapid implementation of support for them (with excellent multithreading performance) in the GNU compiler suite, throws open the doors to widespread adoption of this model in academic and commercial applications for shared memory."* [9]

[9] Jack Dongarra et. al., Numerical Algorithms and Libraries at Exascale

Open Multi-Processing (OpenMP)

OpenMP directives: Task Dependencies

The data dependencies are defined using the `depend` clause during the task creation:

```
#pragma omp task depend(direction:list) [depend(direction:list)] [clauses...]
structured code block
```

Each `depend` clause consists of a data-flow direction and a list of identifiers. Possible directions are:

- `in` – The identifiers are **input** dependencies.
- `out` – The identifiers are **output** dependencies.
- `inout` – The identifiers are **input** and **output** dependencies.

The list of identifiers is a comma separated list of variables from which a pointer can be created.

Tasks with a common `inout` or `output` dependencies are executed in the order as they are created.

Open Multi-Processing (OpenMP)

OpenMP directives: Task Dependencies

Example (OpenMP 4.5 Example 3.3.4 – task_dep4.c)

```
#include <stdio.h>
int main() {
  int x = 1;
  #pragma omp parallel
    #pragma omp single
  {
    #pragma omp task shared(x) depend(out: x)
    x = 2;
    #pragma omp task shared(x) depend(in: x)
    printf("x␣+␣1␣=␣%d.␣", x+1);
    #pragma omp task shared(x) depend(in: x)
    printf("x␣+␣2␣=␣%d\n", x+2);
  }
  return 0;
}
```

Array elements, e.g. `y[i]` or `A[i+ldA*j]`, are also valid identifiers in the `depend` clause. Intervals on arrays, like `A[i:j]`, are also allowed.

Example (OpenMP 4.5 Example 3.3.5 – task_dep5.c)

```c
// Assume BS divides N perfectly
void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N] )
{
  int i, j, k, ii, jj, kk;
  for (i = 0; i < N; i+=BS) {
    for (j = 0; j < N; j+=BS) {
      for (k = 0; k < N; k+=BS) {
        #pragma omp task private(ii, jj, kk) firstprivate(i,j,k) \
          depend ( in: A[i][k], B[k][j] ) \
          depend ( inout: C[i][j] )
        for (ii = i; ii < i+BS; ii++ )
          for (jj = j; jj < j+BS; jj++ )
            for (kk = k; kk < k+BS; kk++ )
              C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
      }
    }
  }
}
```

Open Multi-Processing (OpenMP)

OpenMP directives: Barrier

A synchronization construct that makes the threads wait until all threads in the team have reached this point and only then continues execution.

```
#pragma omp barrier
```

Note that all constructs that allow the `nowait` clause have an implicit barrier at their end. Still sometimes explicit synchronization is desirable.

Open Multi-Processing (OpenMP)

OpenMP clauses: Classification

The OpenMP clauses we have seen above can be divided into two classes

1. **attribute clauses related to data sharing**
2. **clauses controlling data copying**

The OpenMP clauses we have seen above can be divided into two classes

1. **attribute clauses related to data sharing**
2. **clauses controlling data copying**

- clauses usually take a list of arguments
- lists are comma separated and enclosed by `()`.
- all list items must be visible to the clause

Open Multi-Processing (OpenMP)

OpenMP clauses: Data Sharing

Data sharing attributes of a variable in a `parallel` or `task` construct can be one of

- **predetermined**, e.g., loop counters in `for` or `parallel for` constructs are always `private`, `const` qualified variables are `shared`, more can be found in Section 2.9.1 of the OpenMP standard

Open Multi-Processing (OpenMP)

OpenMP clauses: Data Sharing

Data sharing attributes of a variable in a `parallel` or `task` construct can be one of

- **predetermined**, e.g., loop counters in `for` or `parallel for` constructs are always `private`, `const` qualified variables are `shared`, more can be found in Section 2.9.1 of the OpenMP standard
- **explicitly determined** are those attributes where variables are referenced in a clause setting the attributes

Open Multi-Processing (OpenMP)

OpenMP clauses: Data Sharing

Data sharing attributes of a variable in a `parallel` or `task` construct can be one of

- **predetermined**, e.g., loop counters in `for` or `parallel for` constructs are always `private`, `const` qualified variables are `shared`, more can be found in Section 2.9.1 of the OpenMP standard
- **explicitly determined** are those attributes where variables are referenced in a clause setting the attributes
- **implicitly determined**, are the attributes of variables referenced in a given construct but are neither predetermined nor explicitly specified

Open Multi-Processing (OpenMP)

OpenMP clauses: Data Sharing

`default(shared|none)`

- determines the default attributes of variables in the context of a `task` or `parallel` construct.
- defaults to `shared` when not explicitly given in a `parallel` construct
- all other (except `task`) constructs inherit the default from the enclosing construct if no `default` clause is given explicitly.

Open Multi-Processing (OpenMP)

OpenMP clauses: Data Sharing

`default(shared|none)`

- determines the default attributes of variables in the context of a `task` or `parallel` construct.
- defaults to `shared` when not explicitly given in a `parallel` construct
- all other (except `task`) constructs inherit the default from the enclosing construct if no `default` clause is given explicitly.

`shared(list)`

Sets the data sharing attributes of all variables in `list` to be of shared type. That means the variable is considered to be in the shared memory of the team of threads.

Open Multi-Processing (OpenMP)

OpenMP clauses: Data Sharing

`private(list)`

Each variable of the `list` is declared to be a private copy of the thread and not accessible from other threads in the team. It can not be applied to variables that are part of other variables*(elements in arrays or members of a structure)*.

Open Multi-Processing (OpenMP)

OpenMP clauses: Data Sharing

`private(list)`

Each variable of the `list` is declared to be a private copy of the thread and not accessible from other threads in the team. It can not be applied to variables that are part of other variables*(elements in arrays or members of a structure)*.

`firstprivate(list)`

As above but additionally the value of the item in the list is initialized from the corresponding original item when the construct is encountered. The clause has a few more restrictions found in the standard.

Open Multi-Processing (OpenMP)

OpenMP clauses: Data Sharing

`private(list)`

Each variable of the `list` is declared to be a private copy of the thread and not accessible from other threads in the team. It can not be applied to variables that are part of other variables*(elements in arrays or members of a structure)*.

`firstprivate(list)`

As above but additionally the value of the item in the list is initialized from the corresponding original item when the construct is encountered. The clause has a few more restrictions found in the standard.

`lastprivate(list)`

As `private` but causes the original item to be updated after the end of the region from the last iterate of the enclosed loop or the lexically last `section` in a `sections` region.

Open Multi-Processing (OpenMP)

OpenMP clauses: Data Sharing

`reduction(operator:list)`

Accumulates all items of the list into a private copy according to the given `operator` and then combines it with the original instance.

+	(0)	|	(0)
*	(1)	^	(0)
-	(0)	&&	(1)
&	(~0)	||	(0)
max	(Least number in reduction list item type)		
min	(Largest number in reduction list item type)		

Table: Operators for `reduction` with initialization values in ()

Open Multi-Processing (OpenMP)

OpenMP clauses: Data Sharing

Example (OpenMP reduction minimal example)

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

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

/* Some initializations */
n = 100;
for (i=0; i < n; i++)
  a[i] = b[i] = i * 1.0;
sum = 0.0;

#pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);
printf("   Sum = %f\n",sum);
}
```

Open Multi-Processing (OpenMP)

OpenMP clauses: Data Copying

The following are cited from OpenMP 3.1 API C/C++ Syntax Quick Reference Card:
“These clauses support the copying of data values from private or threadprivate variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.”

`copyin(list)`

“Copies the value of the master thread’s threadprivate variable to the threadprivate variable of each other member of the team executing the `parallel` region.”

`copyprivate(list)`

“Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the `parallel` region.”

Open Multi-Processing (OpenMP)

OpenMP Environment Variables

Environment variables can be used to influence the behavior of an OpenMP process, without recompiling the binary, at runtime.

Open Multi-Processing (OpenMP)

OpenMP Environment Variables

Environment variables can be used to influence the behavior of an OpenMP process, without recompiling the binary, at runtime.

`OMP_SCHEDULE`

Specifies the runtime schedule type. Available values are `static`, `dynamic`, `guided`, or `auto` together with an optional `chunk` size.

Open Multi-Processing (OpenMP)

OpenMP Environment Variables

Environment variables can be used to influence the behavior of an OpenMP process, without recompiling the binary, at runtime.

OMP_SCHEDULE

Specifies the runtime schedule type. Available values are `static`, `dynamic`, `guided`, or `auto` together with an optional `chunk` size.

OMP_NUM_THREADS

Must be set to a list of positive integers determining the numbers of threads at the corresponding nested level.

OpenMP Environment Variables

Environment variables can be used to influence the behavior of an OpenMP process, without recompiling the binary, at runtime.

OMP_SCHEDULE

Specifies the runtime schedule type. Available values are `static`, `dynamic`, `guided`, or `auto` together with an optional `chunk` size.

OMP_NUM_THREADS

Must be set to a list of positive integers determining the numbers of threads at the corresponding nested level.

OMP_PROC_BIND

The value of this variable must be `true` or `false`. It determines whether threads may be moved between processors at runtime.

Open Multi-Processing (OpenMP)

OpenMP Environment Variables

Environment variables can be used to influence the behavior of an OpenMP process, without recompiling the binary, at runtime.

OMP_SCHEDULE

Specifies the runtime schedule type. Available values are `static`, `dynamic`, `guided`, or `auto` together with an optional `chunk` size.

OMP_NUM_THREADS

Must be set to a list of positive integers determining the numbers of threads at the corresponding nested level.

OMP_PROC_BIND

The value of this variable must be `true` or `false`. It determines whether threads may be moved between processors at runtime.

More environment variables can be found in Section 4 of the OpenMP standard.

Open Multi-Processing (OpenMP)
OpenMP runtime library functions

We only treat thread and processor number related functions

```
void omp_set_num_threads(int num_threads)
```

Determines the number of threads in subsequent parallel regions that do not specify a `num_threads` clause.

```
int omp_get_num_threads(void)
```

Returns the number of threads in the current team.

Open Multi-Processing (OpenMP)

OpenMP runtime library functions

```
int omp_get_max_threads(void)
```

Provides the maximum number of threads that could be used in a subsequent `parallel` construct.

```
int omp_get_thread_num(void)
```

Returns the thread ID of the current thread. IDs are integers from zero (the master thread) to the number of threads in the team minus one.

```
int omp_get_num_procs(void)
```

returns the number of processors available to the program.

More runtime library functions and detailed descriptions can be found in Section 3 of the OpenMP standard.

Open Multi-Processing (OpenMP)

OpenMP runtime library functions

Example (Hello World revisited)

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

int main (int argc, char *argv[]) {
  int th_id, nthreads;
#pragma omp parallel private(th_id)
  {
     th_id = omp_get_thread_num();

     printf("Hello␣World␣from␣thread␣%d\n", th_id);
     #pragma omp barrier
     if ( th_id == 0 ) {
       nthreads = omp_get_num_threads();
       printf("There␣are␣%d␣threads\n",nthreads);
     }
  }
  return EXIT_SUCCESS;
}
```

Open Multi-Processing (OpenMP)

OpenMP runtime library functions

Two important rules of thumb:

In case of nested loops it is usually best to apply the parallelization to the outermost possible loop.

Open Multi-Processing (OpenMP)

OpenMP runtime library functions

Two important rules of thumb:

In case of nested loops it is usually best to apply the parallelization to the outermost possible loop.

It is in general a good idea to first optimize the sequential code and only then add parallelism to further increase the speed of execution.