Karlsruhe Institute of Technology

Introduction to OpenMP

Holger Obermaier

Steinbuch Centre for Computing (SCC)

- Introduction
- Programming Model
- Basic Usage
- Synchronisation
- Variable Scope
- For Loop
- Task
- SIMD
- Additional OpenMP Features
- References

Basic OpenMP Facts

- Application Program Interface (API)
- API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Portable and versatile:
  - Multiple platforms and compilers
  - Supports C/C++ and Fortran
- Standardised, see www.openmp.org
- Simple and limited set of directives
- Allows for partial parallelisation of a program

Conclusion
Easy way to convert a serial program into a parallel program

Basic OpenMP Facts

- Application Program Interface (API)
- API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Portable and versatile:
  - Multiple platforms and compilers
  - Supports C/C++ and Fortran
- Standardised, see www.openmp.org
- Simple and limited set of directives
- Allows for partial parallelisation of a program

Conclusion
Easy way to convert a serial program into a parallel program

Basic OpenMP Facts

- Application Program Interface (API)
- API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Portable and versatile:
  - Multiple platforms and compilers
  - Supports C/C++ and Fortran
- Standardised, see www.openmp.org
- Simple and limited set of directives
- Allows for partial parallelisation of a program

Conclusion
Easy way to convert a serial program into a parallel program

Basic OpenMP Facts

- Application Program Interface (API)
- API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Portable and versatile:
  - Multiple platforms and compilers
  - Supports C/C++ and Fortran
- Standardised, see www.openmp.org
- Simple and limited set of directives
- Allows for partial parallelisation of a program

Conclusion
Easy way to convert a serial program into a parallel program

Basic OpenMP Facts

- Application Program Interface (API)
- API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Portable and versatile:
  - Multiple platforms and compilers
  - Supports C/C++ and Fortran
- Standardised, see www.openmp.org
- Simple and limited set of directives
- Allows for partial parallelisation of a program

Conclusion
Easy way to convert a serial program into a parallel program

Basic OpenMP Facts

- Application Program Interface (API)
- API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Portable and versatile:
  - Multiple platforms and compilers
  - Supports C/C++ and Fortran
- Standardised, see www.openmp.org
- Simple and limited set of directives
- Allows for partial parallelisation of a program

Conclusion
Easy way to convert a serial program into a parallel program

Basic OpenMP Facts

- Application Program Interface (API)
- API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Portable and versatile:
  - Multiple platforms and compilers
  - Supports C/C++ and Fortran
- Standardised, see www.openmp.org
- Simple and limited set of directives
- Allows for partial parallelisation of a program

Conclusion

Easy way to convert a serial program into a parallel program

OpenMP Programming Model (1)

- Multi-threaded shared-memory parallelism
- Explicit parallelism; no auto-parallelism
- Based on compiler directives (pragmas)
- Support for nested parallelism (parallel constructs within parallel constructs)
- Number of threads can be changed during execution

OpenMP Programming Model (1)

- Multi-threaded shared-memory parallelism
- Explicit parallelism; no auto-parallelism
- Based on compiler directives (pragmas)
- Support for nested parallelism (parallel constructs within parallel constructs)
- Number of threads can be changed during execution

OpenMP Programming Model (1)

- Multi-threaded shared-memory parallelism
- Explicit parallelism; no auto-parallelism
- Based on compiler directives (pragmas)
- Support for nested parallelism (parallel constructs within parallel constructs)
- Number of threads can be changed during execution

OpenMP Programming Model (1)

Karlsruhe Institute of Technology

- Multi-threaded shared-memory parallelism
- Explicit parallelism; no auto-parallelism
- Based on compiler directives (pragmas)
- Support for nested parallelism (parallel constructs within parallel constructs)
- Number of threads can be changed during execution

OpenMP Programming Model (1)

Karlsruhe Institute of Technology

- Multi-threaded shared-memory parallelism
- Explicit parallelism; no auto-parallelism
- Based on compiler directives (pragmas)
- Support for nested parallelism (parallel constructs within parallel constructs)
- Number of threads can be changed during execution

OpenMP Programming Model (2)

Karlsruhe Institute of Technology

Fork-join model used:

- Execution begins with the single master thread
- At the beginning of a parallel region master thread forks
- Team of threads works in parallel
- When work is finished team of threads joins master thread

OpenMP Programming Model (2)

Fork-join model used:

- Execution begins with the single master thread
- At the beginning of a parallel region master thread forks
- Team of threads works in parallel
- When work is finished team of threads joins master thread

OpenMP Programming Model (2)

Fork-join model used:

- Execution begins with the single master thread
- At the beginning of a parallel region master thread forks
- Team of threads works in parallel
- When work is finished team of threads joins master thread

OpenMP Programming Model (2)

Fork-join model used:

- Execution begins with the single master thread
- At the beginning of a parallel region master thread forks
- Team of threads works in parallel
- When work is finished team of threads joins master thread

Basic OpenMP Program

```
#ifdef _OPENMP
    #include <omp.h>
#endif

int main(int argc, char *argv[]) {
    #ifdef _OPENMP
      // your code when OpenMP is present
    #else
      // your code when no OpenMP is present
    #endif

    return 0;
}
```

Runtime Library Routines

`omp_get_thread_num` Thread position within the team

`omp_get_num_threads` Get total number of team threads

`omp_set_num_threads` Set total number of team threads

`omp_get_max_threads` maximum number of threads

`omp_get_num_procs` number of processors

Runtime Library Routines

`omp_get_thread_num` Thread position within the team

`omp_get_num_threads` Get total number of team threads

`omp_set_num_threads` Set total number of team threads

`omp_get_max_threads` maximum number of threads

`omp_get_num_procs` number of processors

Runtime Library Routines

`omp_get_thread_num` Thread position within the team

`omp_get_num_threads` Get total number of team threads

`omp_set_num_threads` Set total number of team threads

`omp_get_max_threads` maximum number of threads

`omp_get_num_procs` number of processors

Runtime Library Routines

`omp_get_thread_num` Thread position within the team

`omp_get_num_threads` Get total number of team threads

`omp_set_num_threads` Set total number of team threads

`omp_get_max_threads` maximum number of threads

`omp_get_num_procs` number of processors

Runtime Library Routines

`omp_get_thread_num` Thread position within the team

`omp_get_num_threads` Get total number of team threads

`omp_set_num_threads` Set total number of team threads

`omp_get_max_threads` maximum number of threads

`omp_get_num_procs` number of processors

Runtime Library Routines (2)

`omp_in_parallel` Determine if within a parallel region

`omp_get_nested` Determine if nested parallelism is enabled

`omp_set_nested` Enable or disable nested parallelism

`omp_get_dynamic` Determine if number of team threads can be adjusted dynamically

`omp_set_dynamic` Enable or disable dynamic adjustment of number of team threads

`omp_get_wtime` Get wall clock time

`omp_get_wtick` Get resolution of wall clock time

Runtime Library Routines (2)

`omp_in_parallel` Determine if within a parallel region

`omp_get_nested` Determine if nested parallelism is enabled

`omp_set_nested` Enable or disable nested parallelism

`omp_get_dynamic` Determine if number of team threads can be adjusted dynamically

`omp_set_dynamic` Enable or disable dynamic adjustment of number of team threads

`omp_get_wtime` Get wall clock time

`omp_get_wtick` Get resolution of wall clock time

Runtime Library Routines (2)

`omp_in_parallel` Determine if within a parallel region

`omp_get_nested` Determine if nested parallelism is enabled

`omp_set_nested` Enable or disable nested parallelism

`omp_get_dynamic` Determine if number of team threads can be adjusted dynamically

`omp_set_dynamic` Enable or disable dynamic adjustment of number of team threads

`omp_get_wtime` Get wall clock time

`omp_get_wtick` Get resolution of wall clock time

Runtime Library Routines (2)

`omp_in_parallel` Determine if within a parallel region

`omp_get_nested` Determine if nested parallelism is enabled

`omp_set_nested` Enable or disable nested parallelism

`omp_get_dynamic` Determine if number of team threads can be adjusted dynamically

`omp_set_dynamic` Enable or disable dynamic adjustment of number of team threads

`omp_get_wtime` Get wall clock time

`omp_get_wtick` Get resolution of wall clock time

Runtime Library Routines (2)

`omp_in_parallel` Determine if within a parallel region

`omp_get_nested` Determine if nested parallelism is enabled

`omp_set_nested` Enable or disable nested parallelism

`omp_get_dynamic` Determine if number of team threads can be adjusted dynamically

`omp_set_dynamic` Enable or disable dynamic adjustment of number of team threads

`omp_get_wtime` Get wall clock time

`omp_get_wtick` Get resolution of wall clock time

Runtime Library Routines (2)

`omp_in_parallel` Determine if within a parallel region

`omp_get_nested` Determine if nested parallelism is enabled

`omp_set_nested` Enable or disable nested parallelism

`omp_get_dynamic` Determine if number of team threads can be adjusted dynamically

`omp_set_dynamic` Enable or disable dynamic adjustment of number of team threads

`omp_get_wtime` Get wall clock time

`omp_get_wtick` Get resolution of wall clock time

Runtime Library Routines (2)

`omp_in_parallel` Determine if within a parallel region

`omp_get_nested` Determine if nested parallelism is enabled

`omp_set_nested` Enable or disable nested parallelism

`omp_get_dynamic` Determine if number of team threads can be adjusted dynamically

`omp_set_dynamic` Enable or disable dynamic adjustment of number of team threads

`omp_get_wtime` Get wall clock time

`omp_get_wtick` Get resolution of wall clock time

Environment Variables

`OMP_SCHEDULE` Determines how iterations of loops are scheduled on processors

`OMP_NUM_THREADS` Set maximum number of threads

`OMP_PROC_BIND` Set whether threads can be moved

`OMP_PLACES` Set where thread can be executed

`OMP_NESTED` Enable or disable nested parallelism

`OMP_DYNAMIC` Enable or disable dynamic adjustment of number of team threads

Examples

- Compile and execute OpenMP programm with GNU compiler
- Compile and execute OpenMP programm with Intel compiler

Environment Variables

OMP_SCHEDULE Determines how iterations of loops are scheduled on processors

OMP_NUM_THREADS Set maximum number of threads

OMP_PROC_BIND Set whether threads can be moved

OMP_PLACES Set where thread can be executed

OMP_NESTED Enable or disable nested parallelism

OMP_DYNAMIC Enable or disable dynamic adjustment of number of team threads

Examples

- Compile and execute OpenMP programm with GNU compiler
- Compile and execute OpenMP programm with Intel compiler

Environment Variables

Karlsruhe Institute of Technology

`OMP_SCHEDULE` Determines how iterations of loops are scheduled on processors

`OMP_NUM_THREADS` Set maximum number of threads

`OMP_PROC_BIND` Set whether threads can be moved

`OMP_PLACES` Set where thread can be executed

`OMP_NESTED` Enable or disable nested parallelism

`OMP_DYNAMIC` Enable or disable dynamic adjustment of number of team threads

Examples

- Compile and execute OpenMP programm with GNU compiler
- Compile and execute OpenMP programm with Intel compiler

Environment Variables

OMP_SCHEDULE Determines how iterations of loops are scheduled on processors

OMP_NUM_THREADS Set maximum number of threads

OMP_PROC_BIND Set whether threads can be moved

OMP_PLACES Set where thread can be executed

OMP_NESTED Enable or disable nested parallelism

OMP_DYNAMIC Enable or disable dynamic adjustment of number of team threads

Examples

- Compile and execute OpenMP programm with GNU compiler
- Compile and execute OpenMP programm with Intel compiler

Environment Variables

Karlsruhe Institute of Technology

`OMP_SCHEDULE` Determines how iterations of loops are scheduled on processors

`OMP_NUM_THREADS` Set maximum number of threads

`OMP_PROC_BIND` Set whether threads can be moved

`OMP_PLACES` Set where thread can be executed

`OMP_NESTED` Enable or disable nested parallelism

`OMP_DYNAMIC` Enable or disable dynamic adjustment of number of team threads

Examples

- Compile and execute OpenMP programm with GNU compiler
- Compile and execute OpenMP programm with Intel compiler

Environment Variables

OMP_SCHEDULE Determines how iterations of loops are scheduled on processors

OMP_NUM_THREADS Set maximum number of threads

OMP_PROC_BIND Set whether threads can be moved

OMP_PLACES Set where thread can be executed

OMP_NESTED Enable or disable nested parallelism

OMP_DYNAMIC Enable or disable dynamic adjustment of number of team threads

Examples

- Compile and execute OpenMP programm with GNU compiler
- Compile and execute OpenMP programm with Intel compiler

Compiler Directive: parallel

```
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
      // block of code executed in parallel
    }

    return 0;
}
```

Examples

- openmp.c
- openmp_wtime.c

Compiler Directive: `single`, `master`

```
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
      #pragma omp single
      // serial code executed by one thread

      #pragma omp master
      // serial code executed by master thread
    }

    return 0;
}
```

Example

`openmp_single_master.c`

Compiler Directive: `parallel sections`

```
int main(int argc, char *argv[]) {
    #pragma omp parallel sections
    {
      #pragma omp section
      // code block 1 executed in parallel

      #pragma omp section
      // code block 2 executed in parallel
    }

    return 0;
}
```

Example

openmp_section.c

Synchronisation

- Compiler directive `critical`: code region must be executed by only one thread at a time. Multiple critical code regions can be distinguished by names
- Compiler directive `atomic`: memory location must be updated by only one thread at a time
- Compiler directive `barrier`: thread waits at the barrier until all other threads have reached it
- Runtime library locking methods
  - `omp_init_lock`, `omp_destroy_lock`
  - `omp_set_lock`, `omp_unset_lock`
  - `omp_test_lock`: attempt to set a lock. Do not block if the lock is unavailable

Synchronisation

- Compiler directive `critical`: code region must be executed by only one thread at a time. Multiple critical code regions can be distinguished by names
- Compiler directive `atomic`: memory location must be updated by only one thread at a time
- Compiler directive `barrier`: thread waits at the barrier until all other threads have reached it
- Runtime library locking methods
  - `omp_init_lock`, `omp_destroy_lock`
  - `omp_set_lock`, `omp_unset_lock`
  - `omp_test_lock`: attempt to set a lock. Do not block if the lock is unavailable

Synchronisation

- Compiler directive `critical`: code region must be executed by only one thread at a time. Multiple critical code regions can be distinguished by names
- Compiler directive `atomic`: memory location must be updated by only one thread at a time
- Compiler directive `barrier`: thread waits at the barrier until all other threads have reached it
- Runtime library locking methods
  - `omp_init_lock`, `omp_destroy_lock`
  - `omp_set_lock`, `omp_unset_lock`
  - `omp_test_lock`: attempt to set a lock. Do not block if the lock is unavailable

Synchronisation

- Compiler directive `critical`: code region must be executed by only one thread at a time. Multiple critical code regions can be distinguished by names
- Compiler directive `atomic`: memory location must be updated by only one thread at a time
- Compiler directive `barrier`: thread waits at the barrier until all other threads have reached it
- Runtime library locking methods
  - `omp_init_lock`, `omp_destroy_lock`
  - `omp_set_lock`, `omp_unset_lock`
  - `omp_test_lock`: attempt to set a lock. Do not block if the lock is unavailable

OpenMP Variable Scope in Parallel Regions

Karlsruhe Institute of Technology

- How are variables transferred from serial to parallel regions?
- Which variables are visible to all threads?
- Which variables are private to a thread?
- Variables with file scope, static variables: Always global
- Loop index variables, stack variables in subroutines called from parallel regions: Always local

OpenMP Variable Scope in Parallel Regions

- How are variables transferred from serial to parallel regions?
- Which variables are visible to all threads?
- Which variables are private to a thread?
- Variables with file scope, static variables: Always global
- Loop index variables, stack variables in subroutines called from parallel regions: Always local

OpenMP Variable Scope in Parallel Regions

Karlsruhe Institute of Technology

- How are variables transferred from serial to parallel regions?
- Which variables are visible to all threads?
- Which variables are private to a thread?
- Variables with file scope, static variables: Always global
- Loop index variables, stack variables in subroutines called from parallel regions: Always local

OpenMP Variable Scope in Parallel Regions

- How are variables transferred from serial to parallel regions?
- Which variables are visible to all threads?
- Which variables are private to a thread?
- Variables with file scope, static variables: Always global
- Loop index variables, stack variables in subroutines called from parallel regions: Always local

OpenMP Variable Scope in Parallel Regions

Karlsruhe Institute of Technology

- How are variables transferred from serial to parallel regions?
- Which variables are visible to all threads?
- Which variables are private to a thread?
- Variables with file scope, static variables: Always global
- Loop index variables, stack variables in subroutines called from parallel regions: Always local

OpenMP Variable Scope Attributes

Karlsruhe Institute of Technology

`private` Variable is local to each thread

- New variable is declared for each thread
- Access to the original variable is replaced by access to the new variable
- Private variable is uninitialised for each thread

`firstprivate` Like private but local variable is initialised with the current global value before parallel region

`lastprivate` Like private but global variable is assigned value of its local pendant in last (sequential) iteration or section after parallel region

OpenMP Variable Scope Attributes

`private` Variable is local to each thread

- New variable is declared for each thread
- Access to the original variable is replaced by access to the new variable
- Private variable is uninitialised for each thread

`firstprivate` Like private but local variable is initialised with the current global value before parallel region

`lastprivate` Like private but global variable is assigned value of its local pendant in last (sequential) iteration or section after parallel region

OpenMP Variable Scope Attributes

Karlsruhe Institute of Technology

`private` Variable is local to each thread

- New variable is declared for each thread
- Access to the original variable is replaced by access to the new variable
- Private variable is uninitialised for each thread

`firstprivate` Like private but local variable is initialised with the current global value before parallel region

`lastprivate` Like private but global variable is assigned value of its local pendant in last (sequential) iteration or section after parallel region

OpenMP Variable Scope Attributes (2)

`threadprivate` File scope variable is local to each thread and persistent over multiple parallel regions

`copyin` Initialise all instances of a `threadprivate` variable from serial region

`shared` Variable is shared among all threads

- All threads can read and write to the same variable
- Coordination of the threads for correct concurrent accesses is necessary

`default` Specify default variable scope

`reduction` Perform global reduction (e.g. sum, product) on the variables

Example

`openmp_scope.c`

OpenMP Variable Scope Attributes (2)

`threadprivate` File scope variable is local to each thread and persistent over multiple parallel regions

`copyin` Initialise all instances of a `threadprivate` variable from serial region

`shared` Variable is shared among all threads

- All threads can read and write to the same variable
- Coordination of the threads for correct concurrent accesses is necessary

`default` Specify default variable scope

`reduction` Perform global reduction (e.g. sum, product) on the variables

Example

`openmp_scope.c`

OpenMP Variable Scope Attributes (2)

`threadprivate` File scope variable is local to each thread and persistent over multiple parallel regions

`copyin` Initialise all instances of a `threadprivate` variable from serial region

`shared` Variable is shared among all threads

- All threads can read and write to the same variable
- Coordination of the threads for correct concurrent accesses is necessary

`default` Specify default variable scope

`reduction` Perform global reduction (e.g. sum, product) on the variables

Example

`openmp_scope.c`

OpenMP Variable Scope Attributes (2)

`threadprivate` File scope variable is local to each thread and persistent over multiple parallel regions

`copyin` Initialise all instances of a `threadprivate` variable from serial region

`shared` Variable is shared among all threads

- All threads can read and write to the same variable
- Coordination of the threads for correct concurrent accesses is necessary

`default` Specify default variable scope

`reduction` Perform global reduction (e.g. sum, product) on the variables

Example

`openmp_scope.c`

OpenMP Variable Scope Attributes (2)

Karlsruhe Institute of Technology

`threadprivate` File scope variable is local to each thread and persistent over multiple parallel regions

`copyin` Initialise all instances of a `threadprivate` variable from serial region

`shared` Variable is shared among all threads

- All threads can read and write to the same variable
- Coordination of the threads for correct concurrent accesses is necessary

`default` Specify default variable scope

`reduction` Perform global reduction (e.g. sum, product) on the variables

Example

`openmp_scope.c`

Compiler Directive: `parallel for`

- Iterations of the following loop are executed in parallel
- Parameter `schedule` specifies how iterations are divided among threads:

`static`	Iterations are evenly divided among threads. Chunk size can be specified
`dynamic`	Iterations are divided into chunks. Each thread gets a chunk. When a thread finishes chunk, it gets another. Chunk size can be specified
`guided`	Chunk size is proportional to the number of unassigned iterations divided by the number of threads
`runtime`	Use environment variable `OMP_SCHEDULE`

Compiler Directive: `parallel for`

- Iterations of the following loop are executed in parallel
- Parameter `schedule` specifies how iterations are divided among threads:

`static`	Iterations are evenly divided among threads. Chunk size can be specified
`dynamic`	Iterations are divided into chunks. Each thread gets a chunk. When a thread finishes chunk, it gets another. Chunk size can be specified
`guided`	Chunk size is proportional to the number of unassigned iterations divided by the number of threads
`runtime`	Use environment variable `OMP_SCHEDULE`

Compiler Directive: `parallel for`

- Iterations of the following loop are executed in parallel
- Parameter `schedule` specifies how iterations are divided among threads:

`static`	Iterations are evenly divided among threads. Chunk size can be specified
`dynamic`	Iterations are divided into chunks. Each thread gets a chunk. When a thread finishes chunk, it gets another. Chunk size can be specified
`guided`	Chunk size is proportional to the number of unassigned iterations divided by the number of threads
`runtime`	Use environment variable `OMP_SCHEDULE`

Compiler Directive: `parallel for`

Karlsruhe Institute of Technology

- Iterations of the following loop are executed in parallel
- Parameter `schedule` specifies how iterations are divided among threads:

`static`	Iterations are evenly divided among threads. Chunk size can be specified
`dynamic`	Iterations are divided into chunks. Each thread gets a chunk. When a thread finishes chunk, it gets another. Chunk size can be specified
`guided`	Chunk size is proportional to the number of unassigned iterations divided by the number of threads
`runtime`	Use environment variable `OMP_SCHEDULE`

Compiler Directive: `parallel for`

Karlsruhe Institute of Technology

- Iterations of the following loop are executed in parallel
- Parameter `schedule` specifies how iterations are divided among threads:

`static`	Iterations are evenly divided among threads. Chunk size can be specified
`dynamic`	Iterations are divided into chunks. Each thread gets a chunk. When a thread finishes chunk, it gets another. Chunk size can be specified
`guided`	Chunk size is proportional to the number of unassigned iterations divided by the number of threads
`runtime`	Use environment variable `OMP_SCHEDULE`

Compiler Directive: `parallel for` (2)

- Loop iteration variable must be an integer
- Loop control parameters must be the same for all threads
- Loop iterations must be independent of each other

Compiler Directive: `parallel for` (2)

- Loop iteration variable must be an integer
- Loop control parameters must be the same for all threads
- Loop iterations must be independent of each other

Compiler Directive: `parallel for` (2)

- Loop iteration variable must be an integer
- Loop control parameters must be the same for all threads
- Loop iterations must be independent of each other

Compiler Directive: `parallel for` (3)

```
int main(int argc, char *argv[]) {
    #pragma omp parallel for
    for (int i = 0; i < 100; i++) {
      // code for i-th iteration
    }
    return 0;
}
```

Example

- `openmp_for_schedule.c`
- `openmp_for_reduce.c`

Compiler Directive: `task`

- Directive `task` creates a task (unit of work)
  - Task can be executed immediately
  - Execution of the task can be defered
  - Task can be executed by any thread in the team
- Similar to `parallel sections`
- Avoids to many nested parallel regions
- Allows to parallelize irregular problems (e.g. recursive algorithms)

Example

- `openmp_task.c`
- `openmp_task_fibonacci.c`

SIMD (1)

- Current systems offer parallelism on various levels
  - Multi cores can be exploited by multi threading
  - Wider processor register can be exploited by vectorization
  - *AVX:* 2 Double, *AVX2:* 4 Double, *AVX512:* 8 Double
- SIMD: Single Instruction, Multiple Data
- Before OpenMP: Proprietary vectorization directives
- Compiler directive: `simd`
  - Cut loop iterations into chuncks that fit into vector registers

$$A = (\underbrace{a_0, a_1, a_2, a_3}_{v_1}, \underbrace{a_4, a_5, a_6, a_7}_{v_2}, \underbrace{a_8, a_9, a_{10}, a_{11}}_{v_3}, a_{12}, a_{13})$$

  - Overrides all dependencies and cost-benefit analysis
  - No multi threading parallelization

SIMD(2)

- Parameters for directive: `simd`

  `safelen` maximum number of elements which can be processed concurrently in one vector operation without breaking dependencies

  `simdlen` suggested number of vector elements in a vector operation

  `aligned` vector memory alignment

- Compiler directive: `for simd`
  - Loop iterations are distributed among threads
  - Each thread vectorizes its chunck

Example

`openmp_simd.c`

Additional OpenMP Features

New OpenMP directives

- to support accelerators (e.g. Intel Xeon Phi, GPGPU)
- for custom data structures in reduction operations
- to cancel parallel execution

References (1)

- Blaise Barney; OpenMP Tutorial
  https://computing.llnl.gov/tutorials/openMP/
- Matthias Müller, Rainer Keller, Isabel Loebich, Rolf Rabenseifner; Introduction to OpenMP
  https://fs.hlrs.de/projects/par/par_prog_ws/2006F/07_openmp-intro12.pdf
- Michael Klemm, Christian Terboven; Die wichtigsten Neuerungen von OpenMP 4.0
  https://www.heise.de/-1915844
- Michael Klemm, Christian Terboven; OpenMP 4.5: Eine kompakte Übersicht zu den Neuerungen
  https://www.heise.de/-3020235

References (2)

- OpenMP Application Program Interface, Version 4.5
  http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
  http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf