Shared Memory Parallel Programming with OpenMP

Overview and Introduction

Heiko Herrmann

October 27, 2010

Are Choice of Hardware and Choice of Programming Languages connected?

Yes!

This is e.g. due to different storing philosophy of matrices.

- FORTRAN stores matrices in column-major format,
- most other languages use row-major.

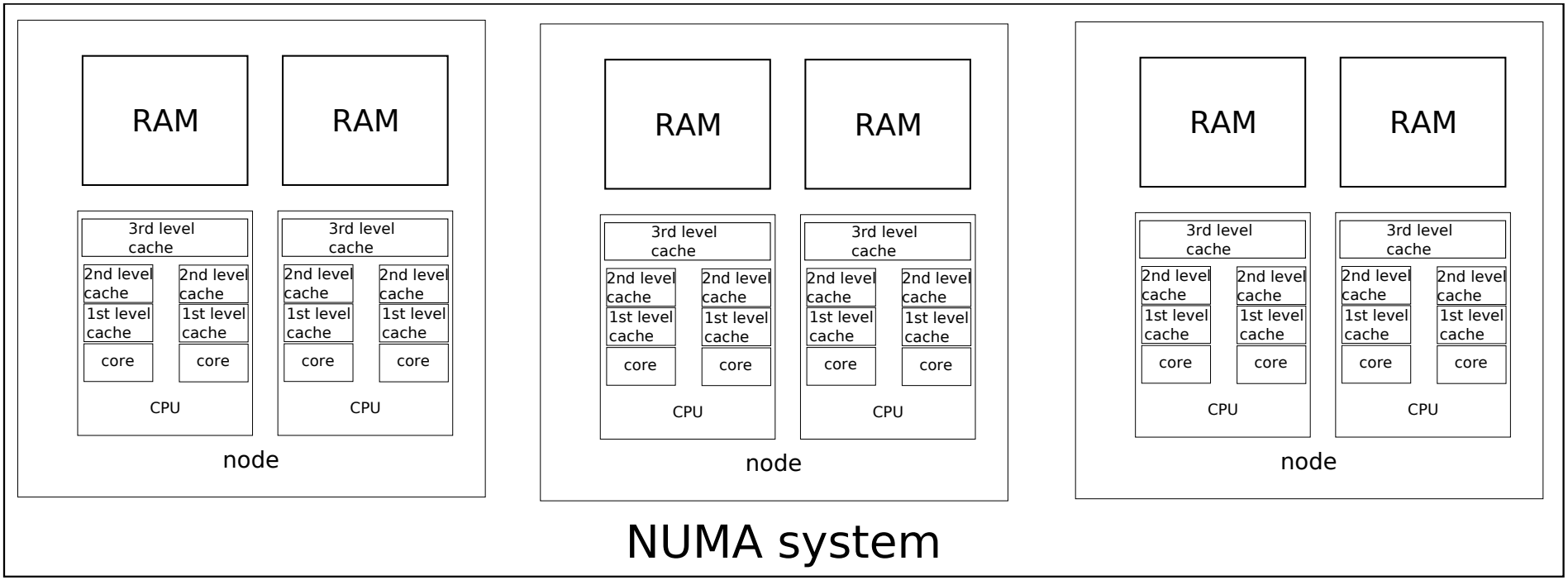
The column-major format is especially suited for vectorizing code and matrix-vector multiplications on vector-processors.

Other factors include

- available libraries
- extensions (OpenCL, OpenMP, MPI) are only available for some languages
  notably
  - OpenMP for C/C++ and FORTRAN (there is no python version!)
  - OpenCL only for C
  - the C++ bindings will be dropped from the next MPI standard (only ISO C and FORTRAN 90 remain in MPI-3, C++ bindings are already deprecated in MPI-2.2)
- not all languages/libraries exist for all hardware platforms

SMP: ccNUMA

NUMA: Non Uniform Memory Access



It is important to keep data in caches coherent

$\longrightarrow$ ccNUMA architecture

ccNUMA requires special interconnect between nodes (low latency, high bandwidth)

does not scale as good as distributed memory parallelism (e.g. with MPI) due to necessary cache coherence

Work/Data Distribution

- Work decomposition
  (based on loop decomposition)

  do i=1,100 ->
  - do i=1,25
  - do i=26,50
  - do i=51,75
  - do i=76,100

- Data decomposition
  (all work for the local portion of data is done by local processor)

  A(1:40,1:100) ->
  - A(1:20,1:50)
  - A(1:20,51:100)
  - A(21:40,1:50)
  - A(21:40,51:100)

- Domain decomposition
  (decomposition of work and data is done in a higher model)

Parallel Programming Models

Hardware independent (more or less):

- OpenMP
  - Shared Memory Directives
  - to define the work decomposition
  - no data decomposition
  - synchronization is implicit (can be also user-defined)
- OpenCL (Open Compute Language)
- HPF (High Performance Fortran)
  - Data Parallelism
  - User specifies data decomposition with directives
  - Communication (and synchronization) is implicit
- MPI (Message Passing Interface), PVM
  - User specifies how work & data is distributed
  - User specifies how and when communication has to be done
  - by calling MPI communication library-routines

Hardware dependent:

- IBM Cell SDK
- nVidia CUDA
- SSE/MMX/. . . for x86 processors

Distribution Methods

decomposition	easiest programming interface
work	OpenMP
data	HPF
domain	MPI

Program Optimization

Different optimization goals and strategies:

- speed (Flops)
- memory usage
- portability
- maintainability
- programming effort/time

NB! Most of these goals are not compatible with each other!

OpenMP

- most OpenMP constructs are compiler directives of pragmas
- focus is to parallelize loops
- an incremental approach to parallelism is offered

Serial Program:

```
void main()
{
  double foo[1000];
  int i;

  for(i=0;i<1000;i++){
    do_huge_calc(foo[i]);
  }
}
```

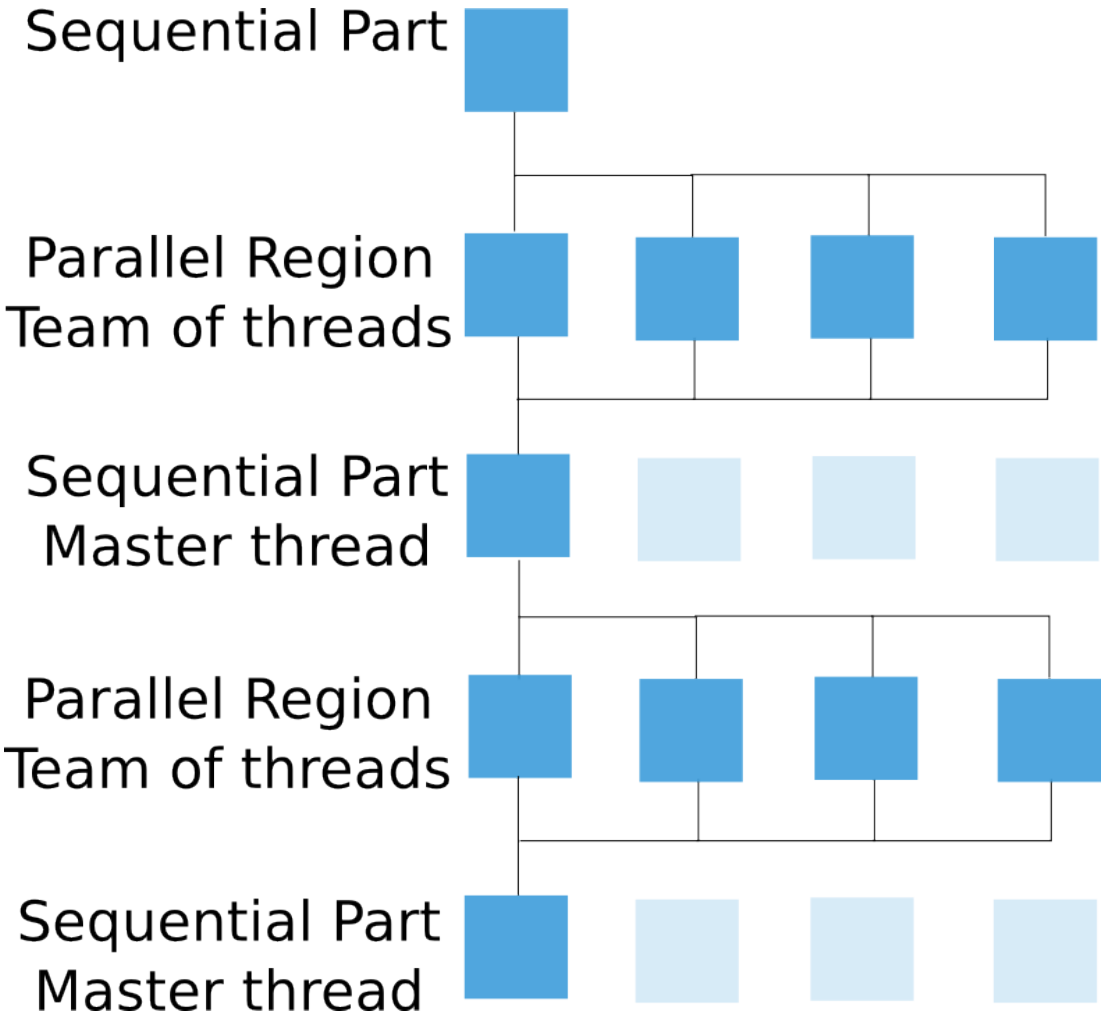
Parallel Program:

```
void main()
{
  double foo[1000];
  int i;
#pragma omp parallel for
  for(i=0;i<1000;i++){
    do_huge_calc(foo[i]);
  }
}
```

OpenMP Programming Model

- OpenMP is a shared memory model (SMP)
- workload is distributed between threads
- enabled via a compiler switch (`-fopenmp` for GCC)
- variables can be
  - shared among threads
  - duplicated for each thread
- unintended sharing of data leads to race conditions
  race condition: when the output of the program changes as the threads are scheduled differently
- control race conditions:
  use synchronization to prevent data conflicts

OpenMP Execution Model



OpenMP Parallel Region

Fortran:

```
!$OMP PARALLEL
  code
!$OMP END PARALLEL
```

C/C++:

```
#pragma omp parallel
  structured code block
/* omp end parallel */
```

OpenMP Directive Format: C/C++

- **`#pragma`** directives – case sensitive
- Format:
  `#pragma` `omp` directive_name [clause [[,]clause] . . .] newline
- Conditional compilation

```
#ifdef _OPENMP
   printf("number of processors: %d\n",get_num_procs());
#endif
```

- include library functions

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

- constant `_OPENMP` is defined when `gcc -fopenmp` is used

OpenMP Data Scope Clauses

- `private` (list)
- `shared` (list)
- if not specified defaults to `shared`, but
  - stack (local) variables in called sub-programs are PRIVATE
  - automatic variables within a block are PRIVATE
  - loop control variables of parallel OMP DO/for are PRIVATE

Recommendation: Avoid private variables, use local variables instead (in C/C++).

OpenMP Environment Variables

- `OMP_NUM_THREADS`
  - sets number of threads
  - if dynamic adjustment is enabled: maximum number of threads
  - `setenv OMP_NUM_THREADS 4` [csh,tcsh]
  - `export OMP_NUM_THREADS=4` [sh, ksh, bash]
- `OMP_SCHEDULE`
- applies only to `DO/for` directives that have schedule type `RUNTIME`
- sets schedule type and chunk size
- `setenv OMP_SCHEDULE "GUIDED,4"` [csh,tcsh]
- `export OMP_SCHEDULE="GUIDED,4"` [sh, ksh, bash]

OpenMP Runtime Library

- C/C++:
  `#include <omp.h>`
  Fortran:
  `!$ INCLUDE 'omp_lib.h'` or `!$ USE omp_lib`
  (availability implementation dependent)
- `int omp_get_num_threads(void);`
  returns number of threads currently in the team of the parallel region
- `int omp_get_thread_num(void);`
  returns the number of the thread in the team. Master thread is 0.

OpenMP Work Sharing Constructs

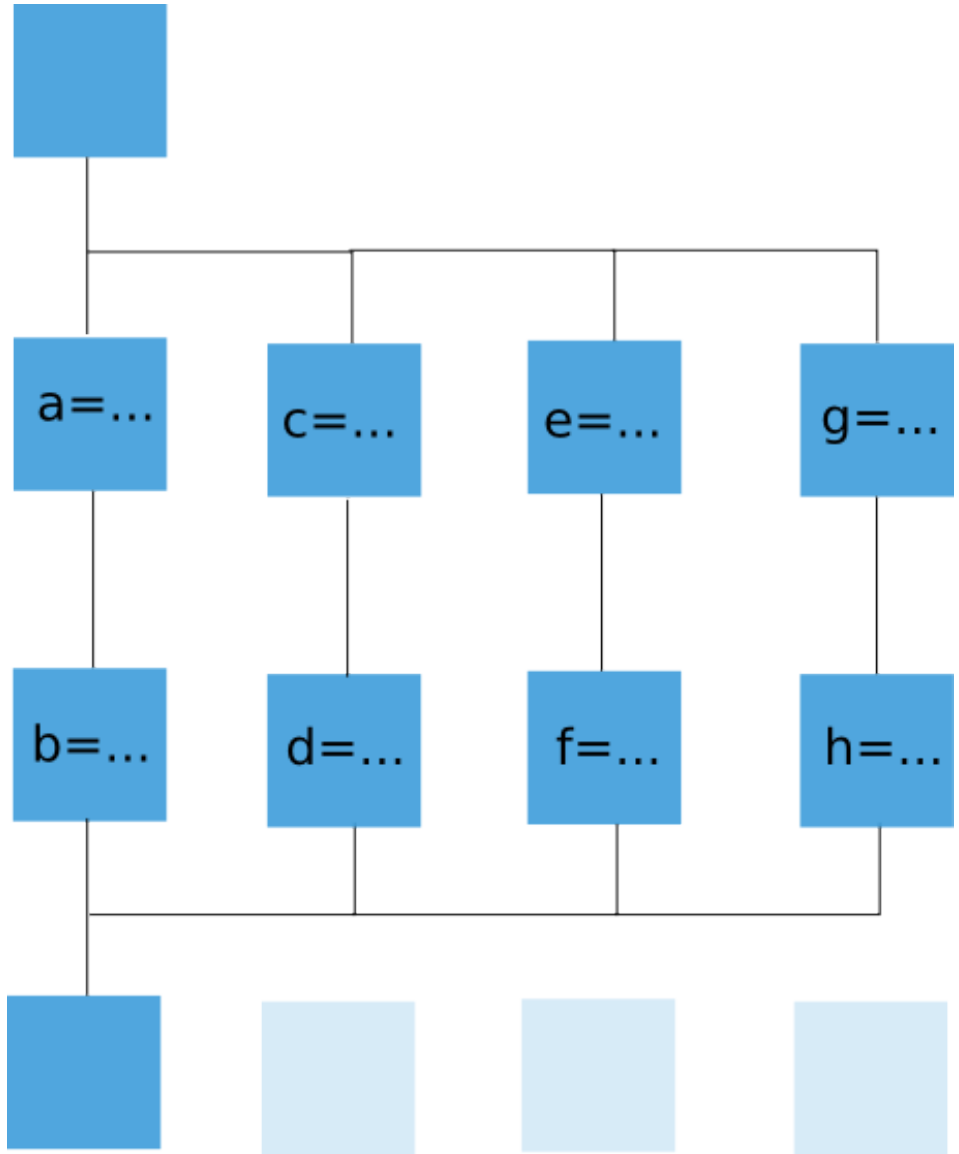
- `sections`
- `for` (C/C++)
- `do` (Fortran)
- `workshare` (Fortran)
- `single`

divide execution of enclosed code among the team
must be within parallel region
they do not launch new threads
no barrier on entry

OpenMP `sections` Directive

C/C++:

```
#pragma omp parallel
{
#pragma omp sections
 {{ a=...;
    b=...;}
#pragma omp section
   { c=...;
     d=...;}
#pragma omp section
   { e=...;
     f=...;}
#pragma omp section
   { g=...;
     h=...;}
 } /* end omp sections */
} /* end omp parallel */
```



OpenMP `do/for` Directive

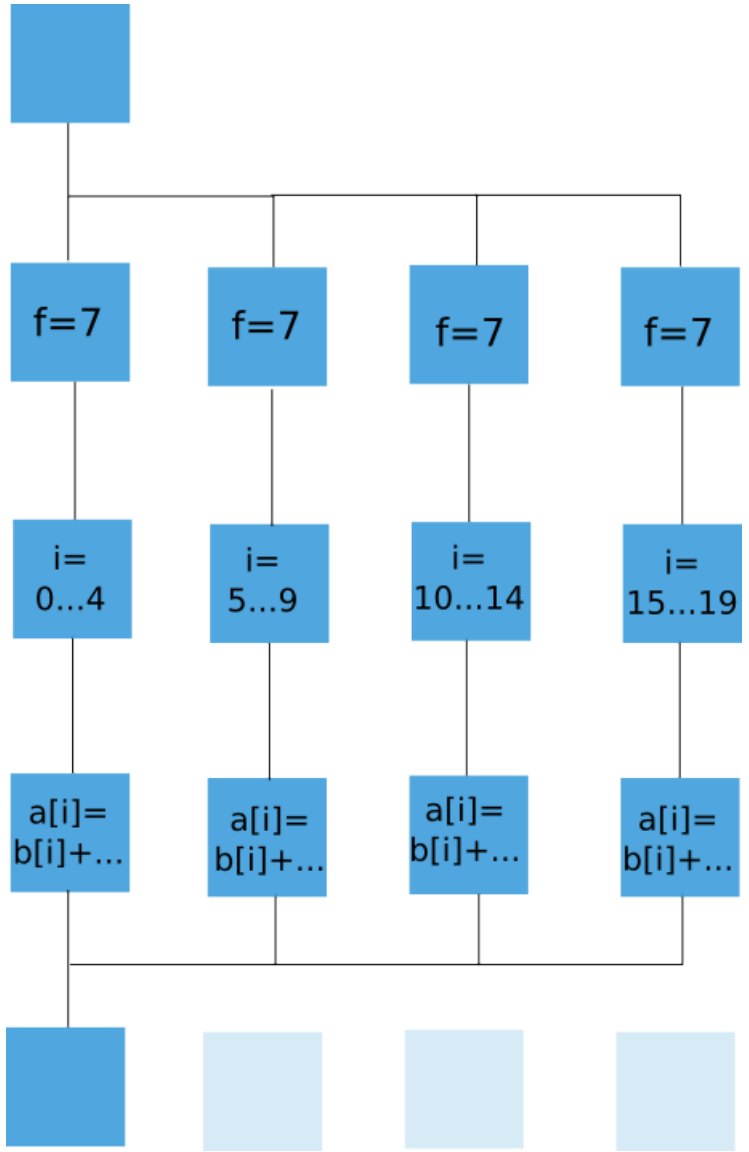
C/C++:

```
#pragma omp parallel private(f)
{
  f=7;

#pragma omp for

  for(i=0;i<20;i++)
    a[i]=b[i]+f*(i+1);


} /* omp end parallel */
```



OpenMP `reduction` Clause

`reduction (operator:list)`
performs a reduction on the variables that appear in `list`, with the operator `operator`
operator: one of

- C/C++: +,*,-,&,^,|,&&,||
- Fortran: +,*,-, .and., .or., ,eqv., .neqv., max, min, iand, ior, ieor

variables must be `shared`
at the end of the `reduction` the shared variable is updated with the result.

OpenMP `reduction` Example

C/C++:

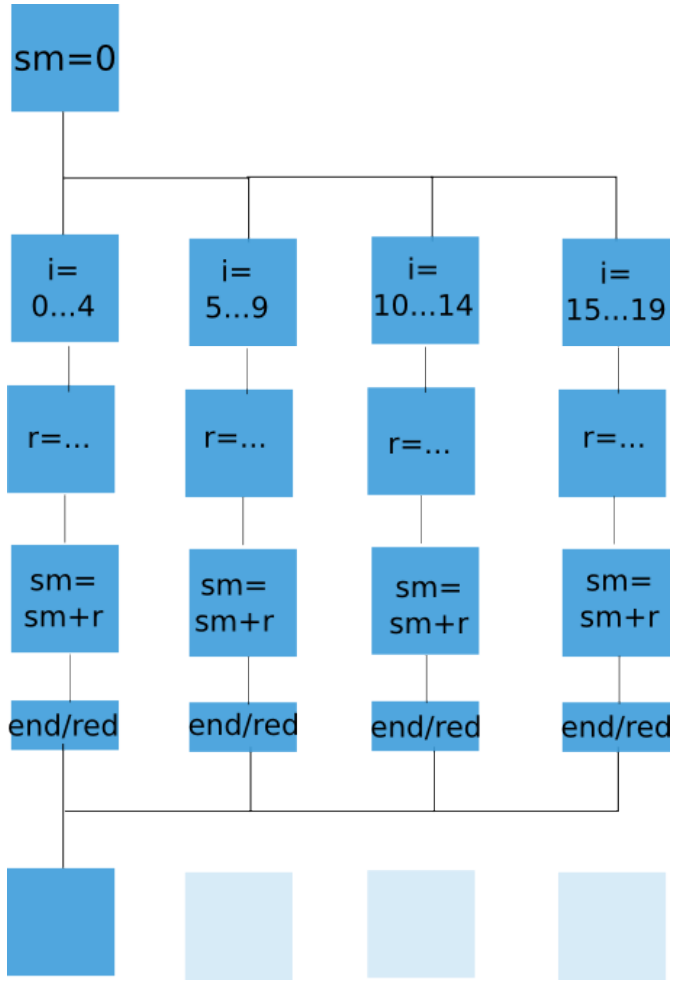
```
sm=0;
#pragma omp parallel for reduction(+:sm)
for(i=0;i<20;i++)
{
  double r;

  r= work(i);

  sm=sm+r;

} /* omp end parallel for */
```

[There is a combined `parallel do/for` directive for parallel regions that contain a single do/for.]



OpenMP Errors

Two types of SMP errors:

- Race Conditions:
  - Def.: Two threads access the same shared variable and at least one thread modifies the variable and the sequence of the access is undefined, i.e. unsynchronized
  - the outcome of a program depends on the detailed timing of the threads in the team
  - often caused by unintended share of data
- Deadlock
  - threads lock up waiting on a locked resource that will never become free

OpenMP Race Conditions – Example 1

```
#pragma omp parallel
{
#pragma omp sections
 a=b+c;
#pragma omp section
 b=a+c;
#pragma omp section
 c=b+a;
} /* omp end parallel sections */
```

OpenMP Race Conditions – Example 2

```
#pragma omp parallel shared(x), private(tmp)
{
 id=omp_get_thread_num();
#pragma omp for reduction(+:x) nowait
 for(i=1;i<100;i++)
   {
   tmp=work1(i);
   x=x+tmp;
   } /* omp end for reduction */
 y(id)=work2(x,id);
/* omp end parallel */
```

OpenMP Summary (1)

- standardized compiler directives for shared memory programming
- fork-join model based on threads
- support from (relevant) hardware vendors
- incremental approach for parallelism
- allows to keep single source for parallel and sequential execution
- race conditions and deadlock possible
- use tools to check for race conditions

NB! Single Source

Don't fork the code for parallelization, keep a single source tree!

Thread Checker

Goals:

- detect race conditions
- other parallelizations errors, like missing firstprivate

NB!

OpenMP parallelizations should never be used in production without verification with race-condition checking tools!

Parallel Debugging

- Intel Thread Checker (needs Intel C/FORTRAN Compiler)
  Linux and Windows; needs Intel compiler; commercial:
  - C++ Compiler prof. Linux 499,- + VAT (download), 539,- + VAT (CD)
  - Compiler suite prof. Linux (C++ and FORTRAN) 1099,- + VAT (CD)
  - Thread Checker Linux 409,- + VAT (download), 449,- + VAT (CD)
  - academic pricing on request
- Portland Group pgdbg and compiler
  commercial (akad.: C/C++/FORTRAN 699,-, C 299,-, FORTRAN 499,-)
- Sun/Oracle Solaris Studio Thread Analyzer
  Linux, Solaris; free (but closed source)
- maybe PathScale Eko with pathdb (or path64)
- Helgrind (Valgrind suite)
  Linux (Posix) only; free open-source
- DRD (Valgrind suite)
  Linux (Posix) only; free open-source
  (gcc needs to be compiled with `--disable-linux-futex`)
- ThreadSanitizer
  Linux (based on Valgrind), Windows (based on PIN); free open-source

Race Checker: Method

- Compile your OpenMP program with thread checker/debugging info
- start and execute with race checker
  - executed on 1 thread
  - verifying all memory accesses
  - $\sim$ 300 times slower than normal execution (use small but *relevant* data set)
- invoke analysis tool
  - error report
  - with references to your source code
- try to find the parallelization bugs in your code
- try to correct these (without modifying the serial semantics of the program
- compile and execute again
- repeat until all errors are resolved

Intel Thread Checker: Processing

TCI-mode (requires ICC):

compile	`icc -tcheck -openmp -g -o myprog myprog.c`
run	`tcheck_cl -w 90 -o myprog.txt ./myprog`
text output	`tcheck_cl -f txt -w 130 threadchecker.thr`
csv output	`tcheck_cl -f csv threadchecker.thr`

good tool

Sun/Oracle Thread Analyzer

Used with Sun/Oracle Solaris Studio

compile	`cc -xinstrument=datarace source.c`
run	`collect -r [ race | deadlock ] a.out`
display	`er_print [-race | -deadlock] tha.1.er`
display (GUI)	`tha tha.1.er`

untested

Valgrind-DRD

Requires gcc > 4.2 compiled with `--disable-linux-futex` (e.g. use `valgrind-3.6.0~svn11254/drd/scripts/download-and-build-gcc`) and recent Valgrind (SVN or debian/squeeze should be sufficient).

compile `gcc -g -fopenmp source.c`

set gcc `export CC=/path/to/gcc`

set libs `export LD_LIBRARY_PAT=/path/to/libs`

run
```
valgrind --tool=drd --check-stack-var=yes
    --read-var-info=yes --first-race-only=yes a.out
```

take care to use the futex-less gcc and libs! (most distributions have futex enabled)

Didn't find all the data-races in my tests.

Valgrind-Helgrind

Requires gcc > 4.2 compiled with `--disable-linux-futex` (e.g. use `valgrind-3.6.0~svn11254/drd/scripts/download-and-build-gcc`) and recent Valgrind (SVN or debian/squeeze should be sufficient).

compile `gcc -g -fopenmp source.c`

set gcc `export CC=/path/to/gcc`

set libs `export LD_LIBRARY_PAT=/path/to/libs`

run
```
valgrind --tool=helgrind
    --check-stack-var=yes a.out
```

take care to use the futex-less gcc and libs! (most distributions have futex enabled)

Didn't find all the data-races in my tests.

Thread Checker – Summary

- Intel Thread Checker finds the locations of race conditions, but the programmer must find the reason
- pgdbg, pathdb and Thread Analyser: untested
- Valgrind DRD/Helgrind didn't find all races
- Source code instrumentation returns an important error report (executed with 1 thread)
- programmer has to eliminate all errors or must be sure that the reported error is a "false positive" (note in source code with sign. of programmer)

It is absolutely necessary to use a tool to check for race-conditions!

And . . . I think I will stop here

Acknowledgements:

- Thanks to Eero Vainikko for inviting me!