OpenMP Reference Sheet for C/C++

Constructs

<parallelize a for loop by breaking apart iterations into chunks>
#pragma omp parallel for [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr), ordered, schedule(type[,chunkSize])]
<A,B,C such that total iterations known at start of loop>
for(A=C;A<B;A++) {
<your code here>

<force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>
#pragma omp ordered {
<your code here>
}
}

<parallelized sections of code with each section operating in one thread>
#pragma omp parallel sections [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr)] {
#pragma omp section {
<your code here>
}
#pragma omp section {
<your code here>
}
....
}

<grand parallelization region with optional work-sharing constructs defining more specific splitting of work and variables amongst threads. You may use work-sharing constructs without a grand parallelization region, but it will have no effect (sometimes useful if you are making OpenMP'able functions but want to leave the creation of threads to the user of those functions)>
#pragma omp parallel [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(private|shared|none), reduction(op:vars), copyin(vars), if(expr)] {
<the work-sharing constructs below can appear in any order, are optional, and can be used multiple times. Note that no new threads will be created by the constructs. They reuse the ones created by the above parallel construct.>

<your code here (will be executed by all threads)>

<parallelize a for loop by breaking apart iterations into chunks>
#pragma omp for [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), ordered, schedule(type[,chunkSize]), nowait]
<A,B,C such that total iterations known at start of loop>
for(A=C;A<B;A++) {
<your code here>

<force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>
#pragma omp ordered {
<your code here>
}
}

<parallelized sections of code with each section operating in one thread>
#pragma omp sections [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), nowait] {
#pragma omp section {
<your code here>
}
#pragma omp section {
<your code here>
}
....
}

<only one thread will execute the following. NOT always by the master thread>
#pragma omp single {
<your code here (only executed once)>
}
}

Directives

shared(vars) *<share the same variables between all the threads>*
private(vars) *<each thread gets a private copy of variables. Note that other than the master thread, which uses the original, these variables are not initialized to anything.>*
firstprivate(vars) *<like private, but the variables do get copies of their master thread values>*
lastprivate(vars) *<copy back the last iteration (in a for loop) or the last section (in a sections) variables to the master thread copy (so it will persist even after the parallelization ends)>*
default(private|shared|none) *<set the default behavior of variables in the parallelization construct. shared is the default setting, so only the private and none setting have effects. none forces the user to specify the behavior of variables. Note that even with shared, the iterator variable in for loops still is private by necessity >*
reduction(op:vars) *<vars are treated as private and the specified operation(op, which can be +,*,-,&,|,&,&&,||) is performed using the private copies in each thread. The master thread copy (which will persist) is updated with the final value.>*

copyin(vars) *<used to perform the copying of threadprivate vars to the other threads. Similar to firstprivate for private vars.>*
if(expr) *<parallelization will only occur if expr evaluates to true.>*
schedule(type [,chunkSize]) *<thread scheduling model>*

type	*chunkSize*
static	*number of iterations per thread pre-assigned at beginning of loop (typical default is number of processors)*
dynamic	*number of iterations to allocate to a thread when available (typical default is 1)*
guided	*highly dependent on specific implementation of OpenMP*

nowait *<remove the implicit barrier which forces all threads to finish before continuation in the construct>*

Synchronization/Locking Constructs *<May be used almost anywhere, but will only have effects within parallelization constructs.>*

<only the master thread will execute the following. Sometimes useful for special handling of variables which will persist after the parallelization.>

```
#pragma omp master {
    <your code here (only executed once and by the master thread).
}
```

<mutex lock the region. name allows the creation of unique mutex locks.>

```
#pragma omp critical [(name)] {
    <your code here (only one thread allowed in at a time)>
}
```

<force all threads to complete their operations before continuing>
#pragma omp barrier

<like critical, but only works for simple operations and structures contained in one line of code>
#pragma omp atomic
<simple code operation, ex. a += 3; Typical supported operations are ++,--,+,,-,/,&,^,<<,>>,| on primitive data types>*

<force a register flush of the variables so all threads see the same memory>
#pragma omp flush[(vars)]

<applies the private clause to the vars of any future parallelize constructs encountered (a convenience routine)>
#pragma omp threadprivate(vars)

Function Based Locking *< nest versions allow recursive locking>*

void **omp_init_[nest_]lock**(omp_lock_t*) *<make a generic mutex lock>*
void **omp_destroy_[nest_]lock**(omp_lock_t*) *<destroy a generic mutex lock>*
void **omp_set_[nest_]lock**(omp_lock_t*) *<block until mutex lock obtained>*
void **omp_unset_[nest_]lock**(omp_lock_t*) *<unlock the mutex lock>*
int **omp_test_[nest_]lock**(omp_lock_t*) *<is lock currently locked by somebody>*

Settings and Control

int **omp_get_num_threads**() *<returns the number of threads used for the parallel region in which the function was called>*
int **omp_get_thread_num**() *<get the unique thread number used to handle this iteration/section of a parallel construct. You may break up algorithms into parts based on this number.>*
int **omp_in_parallel**() *<are you in a parallel construct>*
int **omp_get_max_threads**() *<get number of threads OpenMP can make>*
int **omp_get_num_procs**() *<get number of processors on this system>*
int **omp_get_dynamic**() *<is dynamic scheduling allowed>*
int **omp_get_nested**() *<is nested parallelism allowed>*
double **omp_get_wtime**() *<returns time (in seconds) of the system clock>*
double **omp_get_wtick**() *<number of seconds between ticks on the system clock>*
void **omp_set_num_threads**(int) *<set number of threads OpenMP can make>*
void **omp_set_dynamic**(int) *<allow dynamic scheduling (note this does not make dynamic scheduling the default)>*
void **omp_set_nested**(int) *<allow nested parallelism; Parallel constructs within other parallel constructs can make new threads (note this tends to be unimplemented in many OpenMP implementations)>*

<env vars- implementation dependent, but here are some common ones>
OMP_NUM_THREADS "number" *<maximum number of threads to use>*
OMP_SCHEDULE "type,chunkSize" *<default #pragma omp schedule settings>*

Legend

vars is a comma separated list of variables
[optional parameters and directives]
<descriptions, comments, suggestions>
.... above directive can be used multiple times
For mistakes, suggestions, and comments please email e_berta@plutospin.com

Visual Studio for parallel programming - Agenda

- Overview and Project Management
- Using OpenMP
- Debugging OpenMP programs

Paralel processing support VS2010 - Overview

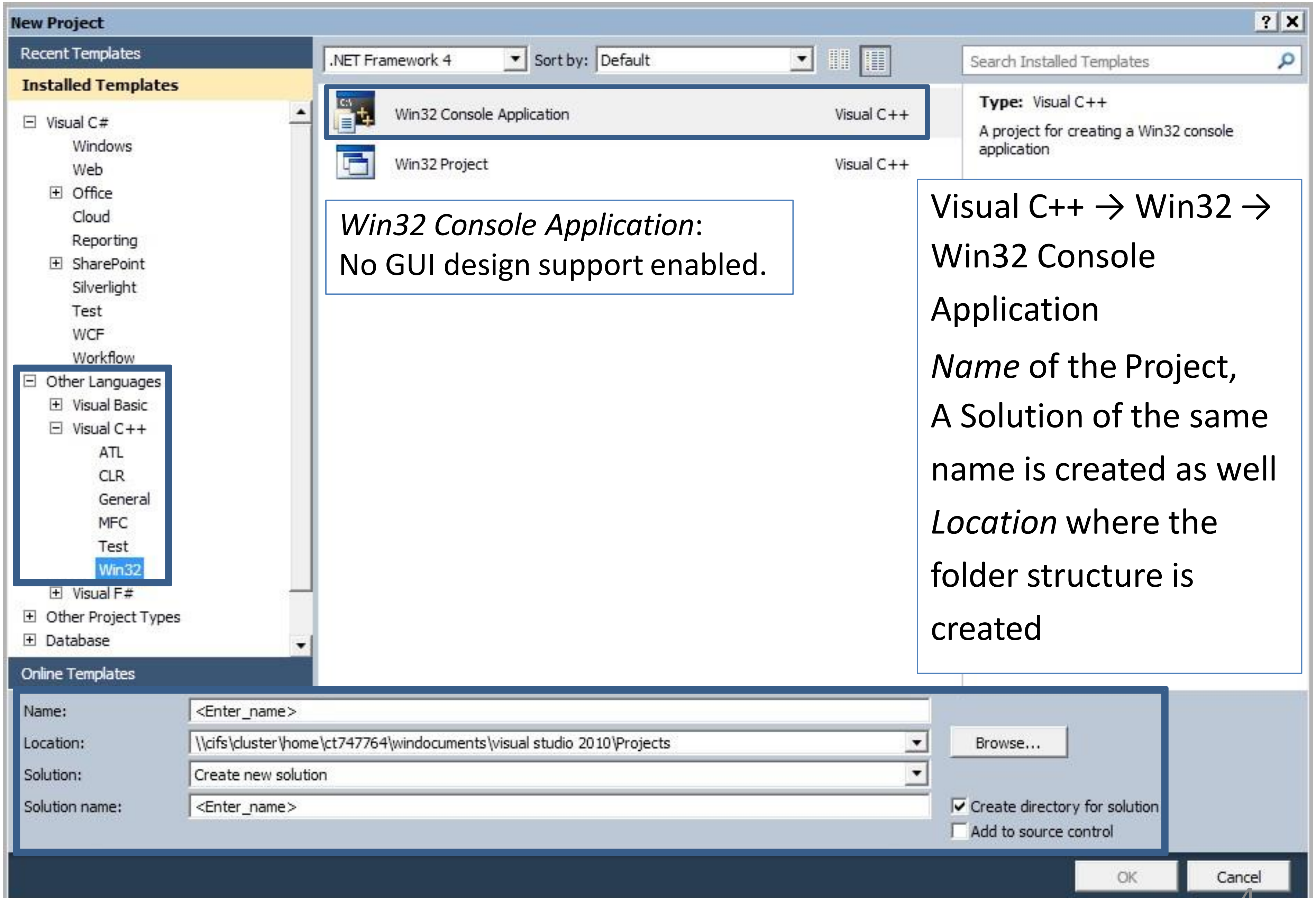
- VS2010 provides good support for Parallel Programming:
  - Support for OpenMP for Shared-Memory parallel compilation
  - Debugging of parallel programs: OpenMP and MPI
  - Architecture-specific compiler optimizations

Visual Studio: Project Management (1/5)

- Everything that you do in Visual Studio will take place within the context of a *Solution*.
  - A Solution is a higher-level container for other items, for example a *Project*. Any other kind of file type can also be added to a Solution, for example documentation items.
  - A Solution can not contain another Solution.
  - Solutions group and apply properties across projects.

- A *Project* maps one to one with a compiler target.
  - A Project organizes the code.

- To start your work, a new Project has to be created with *File → New → Project…*



Win32 Console Application:
No GUI design support enabled.

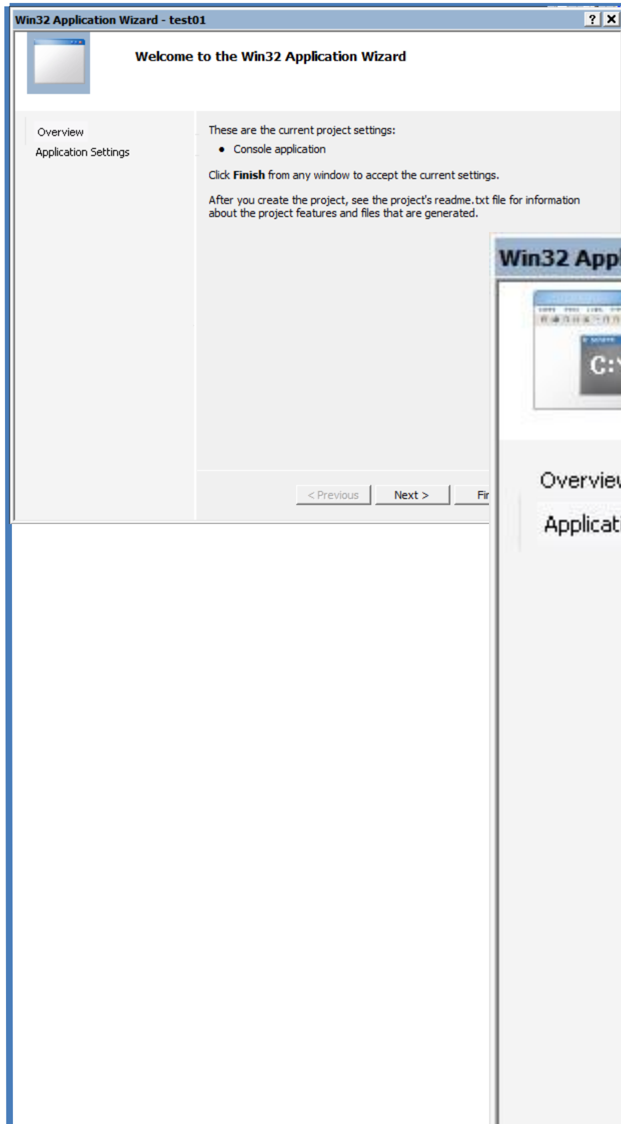
Visual C++ → Win32 → Win32 Console Application

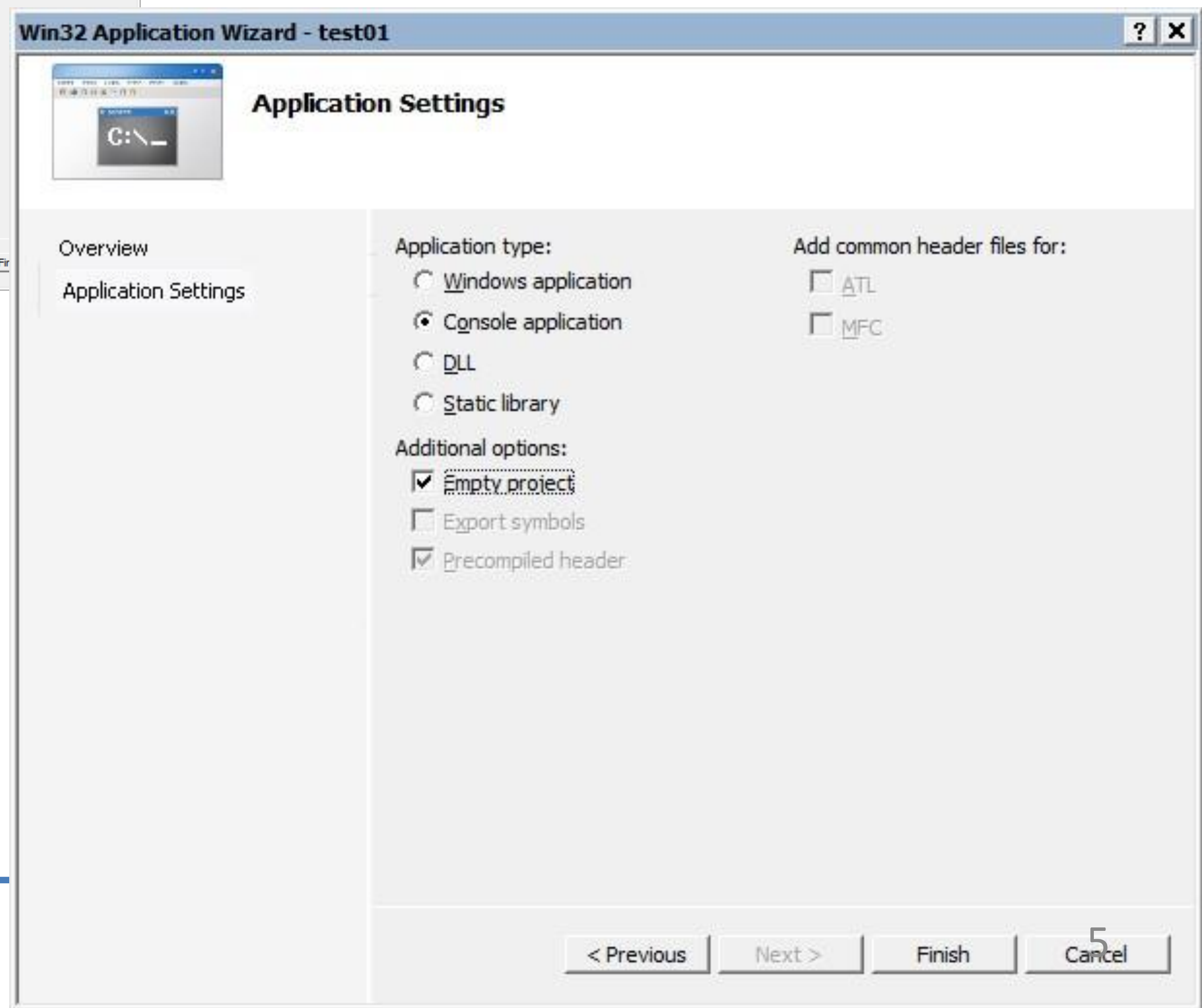
Name of the Project,
A Solution of the same name is created as well

Location where the folder structure is created

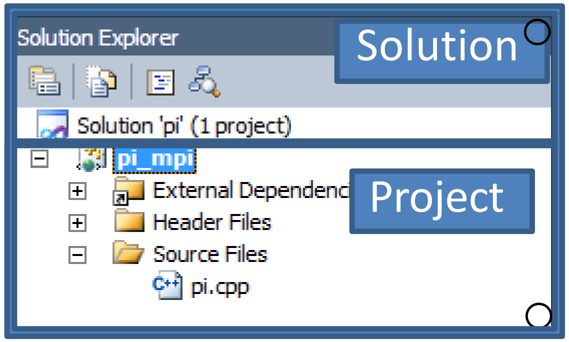
Visual Studio: Project Management (3/5)

Choose *Empty project* if you already have source files.





Visual Studio: Project Management (5/5)



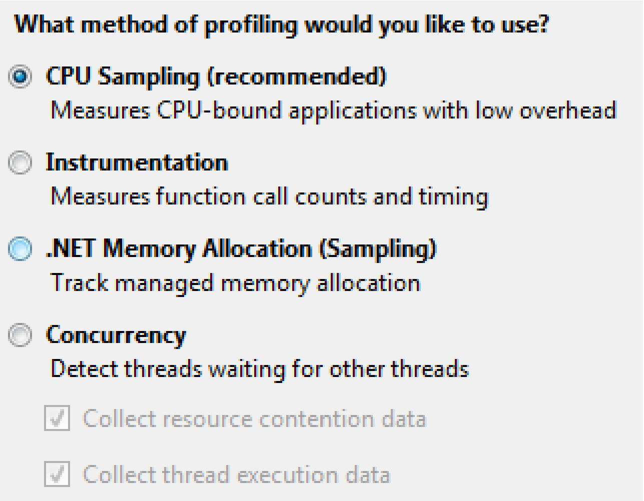
In many cases, the shortest way to a desired operation can be found by right-clicking on a GUI element and using the context menu.

- Adding existing source code items (files) to a project: right-click on the Project (not the Solution !) and *Add → Existing Item…*
- Adding new items: right-click on the Project and *Add → New Item…*

- The folders (e.g. *Source Files*) do not have any other meaning than aiding you in structuring the files in a project. They do not map to physical folders. Creating your own folders may help to organize large projects.

Visual Studio 2010: Performance Analyzer

- VS2010 comes with new tools to analyze your (parallel) application‘s performance: *Analyze* → Profiler → *New Performance Session*, then *Analyze* → *Launch Performance Wizard*



All this can be done on the local Workstation, or in the Cluster!

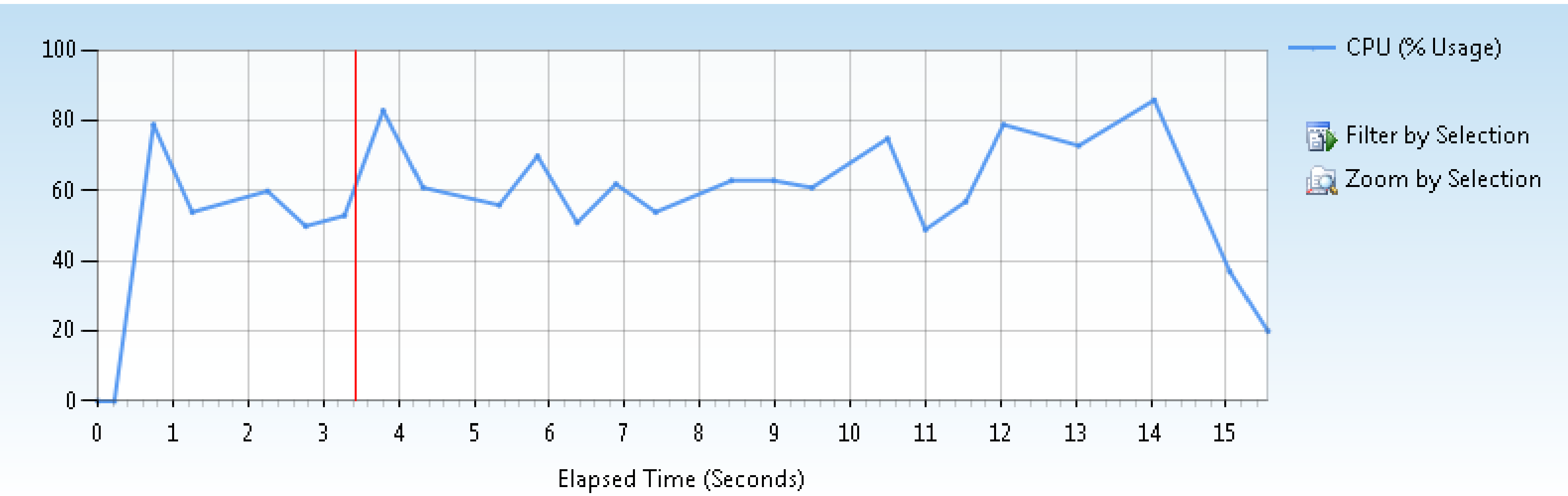
- CPU Sampling:
  - Will run the application under the control of a sampling performance analyzer (snapshots of the program‘s call tree are taken at regular intervals → non-intrusive, low overhead)

Performance Analyzer: CPU Sampling (1/2)

- Summary View highlights the program's Critical Path:

Sample Profiling Report

2,702 total samples collected



Hot Path

The most expensive call path based on sample counts

Name	Inclusive %	Exclusive %
jacobi_aut.exe	100,00	0,00
_mainCRTStartup	100,00	0,00
__tmainCRTStartup	100,00	0,00
_main	100,00	0,00
_Jacobi	**98,70**	**98,48**

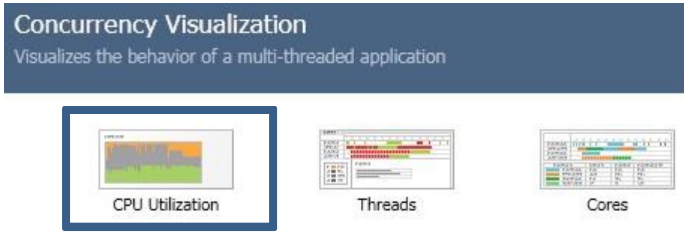
Related Views: Call Tree Functions

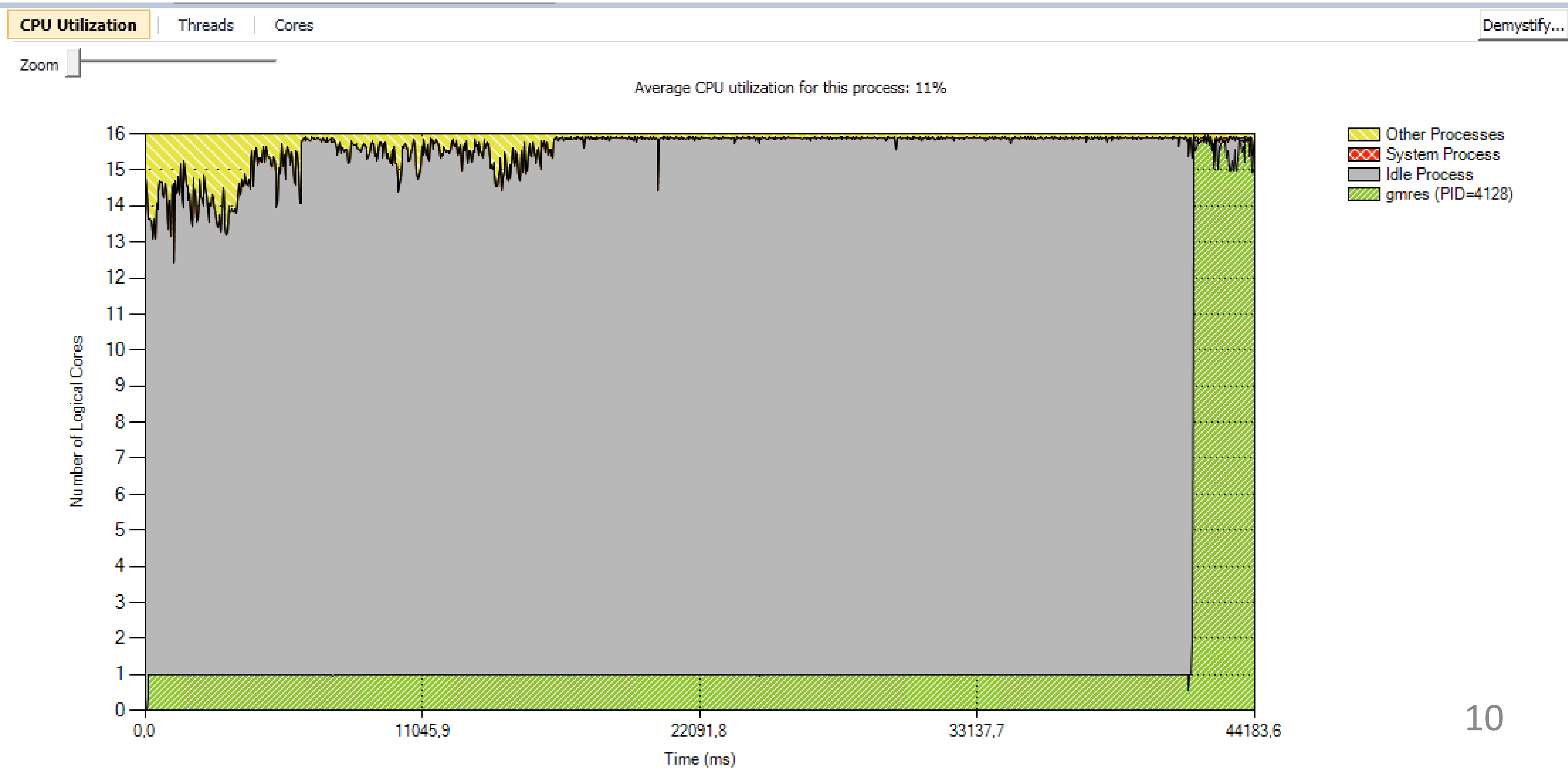
Performance Analyzer: CPU Sampling (2/2)

- Performance information can be display on source level:

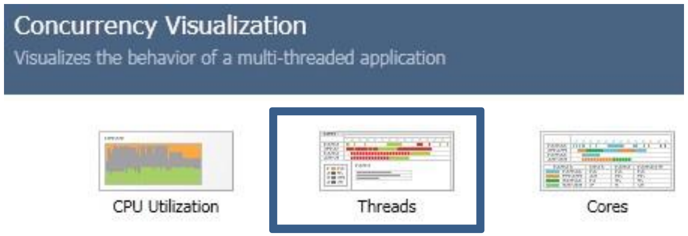
```
            /* compute stencil, residual and update */
            for (j = data->iRowFirst + 1; j <= data->iRowLast - 1; j++)
            {
  1.9 %         for (i = 1; i <= data->iCols - 2; i++)
                {
                    fLRes = ( ax * (UOLD(j, i-1) + UOLD(j, i+1))
                            + ay * (UOLD(j-1, i) + UOLD(j+1, i))
 66.5 %                     +  b * UOLD(j, i) - F(j, i)) / b;

                    /* update solution */
 11.4 %             U(j,i) = UOLD(j,i) - data->fRelax * fLRes;

  1.0 %
  0.7 %
< 0.1 %
```

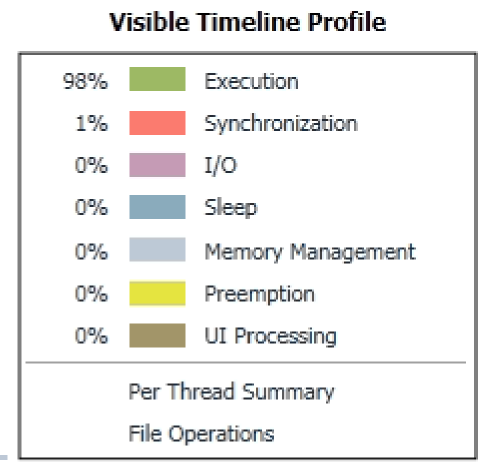
Concurrency (1/3)

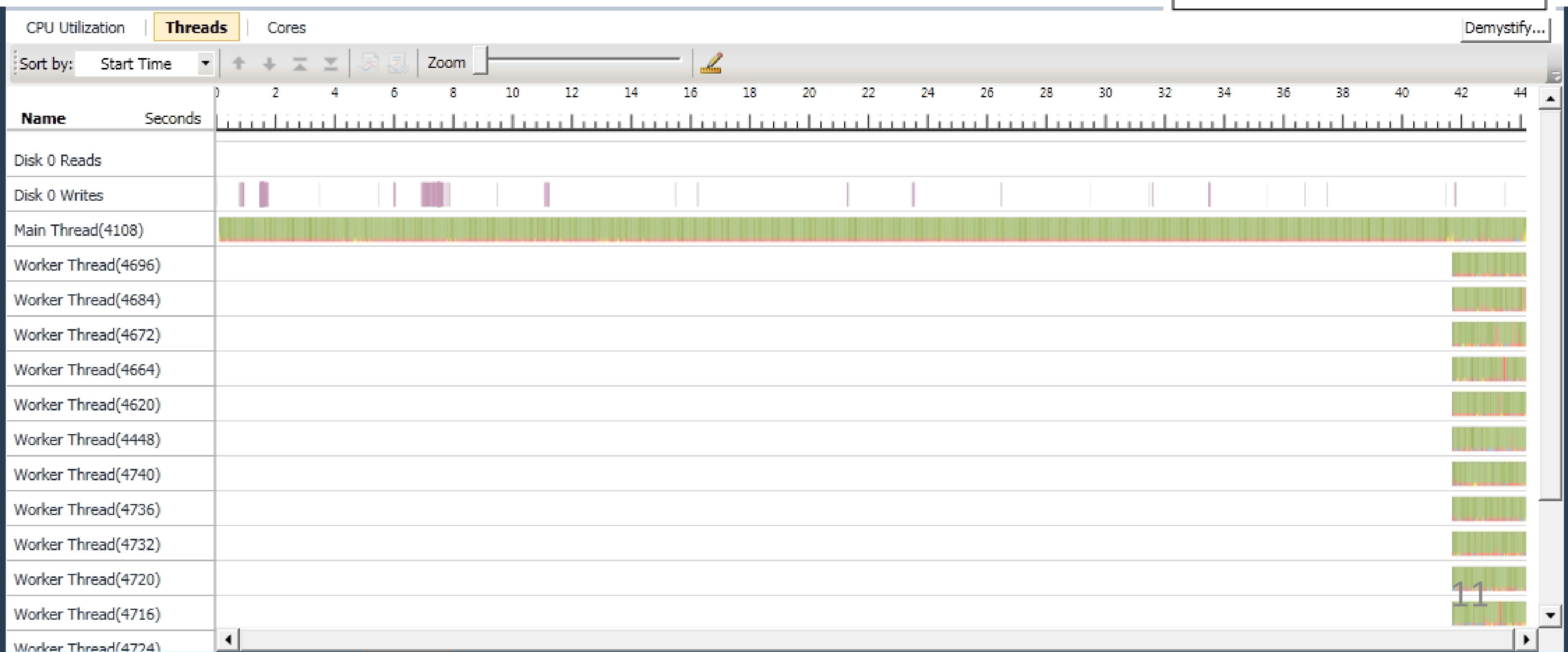




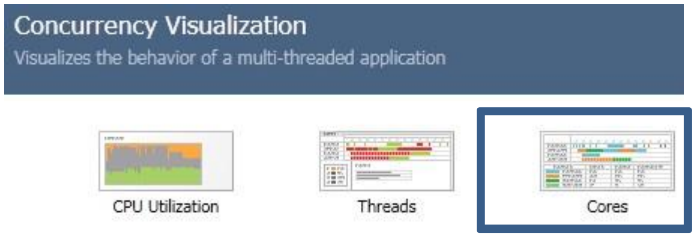
Concurrency (2/3)

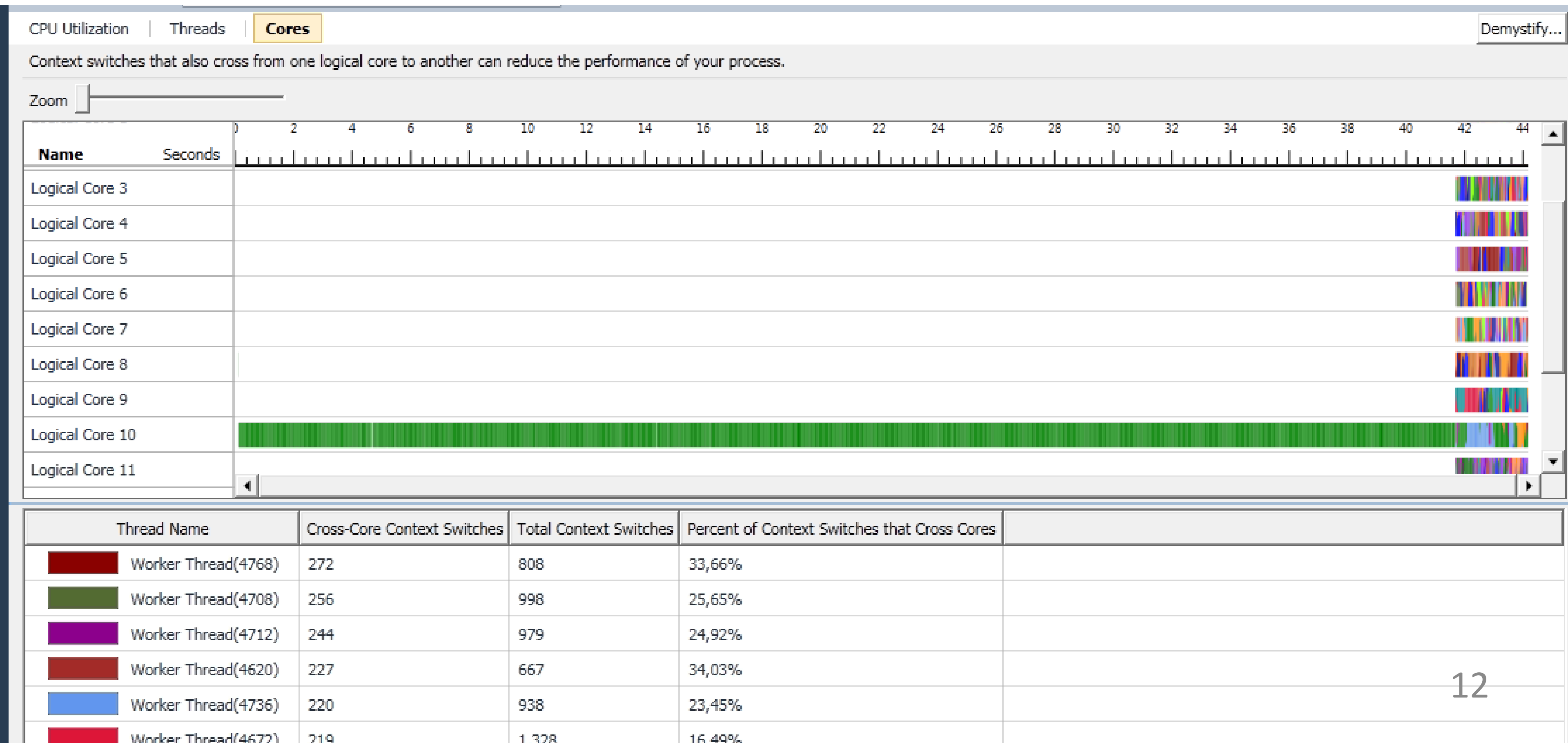






Concurrency (3/3)





Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
Worker Thread(4768)	272	808	33,66%
Worker Thread(4708)	256	998	25,65%
Worker Thread(4712)	244	979	24,92%
Worker Thread(4620)	227	667	34,03%
Worker Thread(4736)	220	938	23,45%
Worker Thread(4672)	219	1 328	16,49%

Performance Analyzer: Thread Contention (1/2)

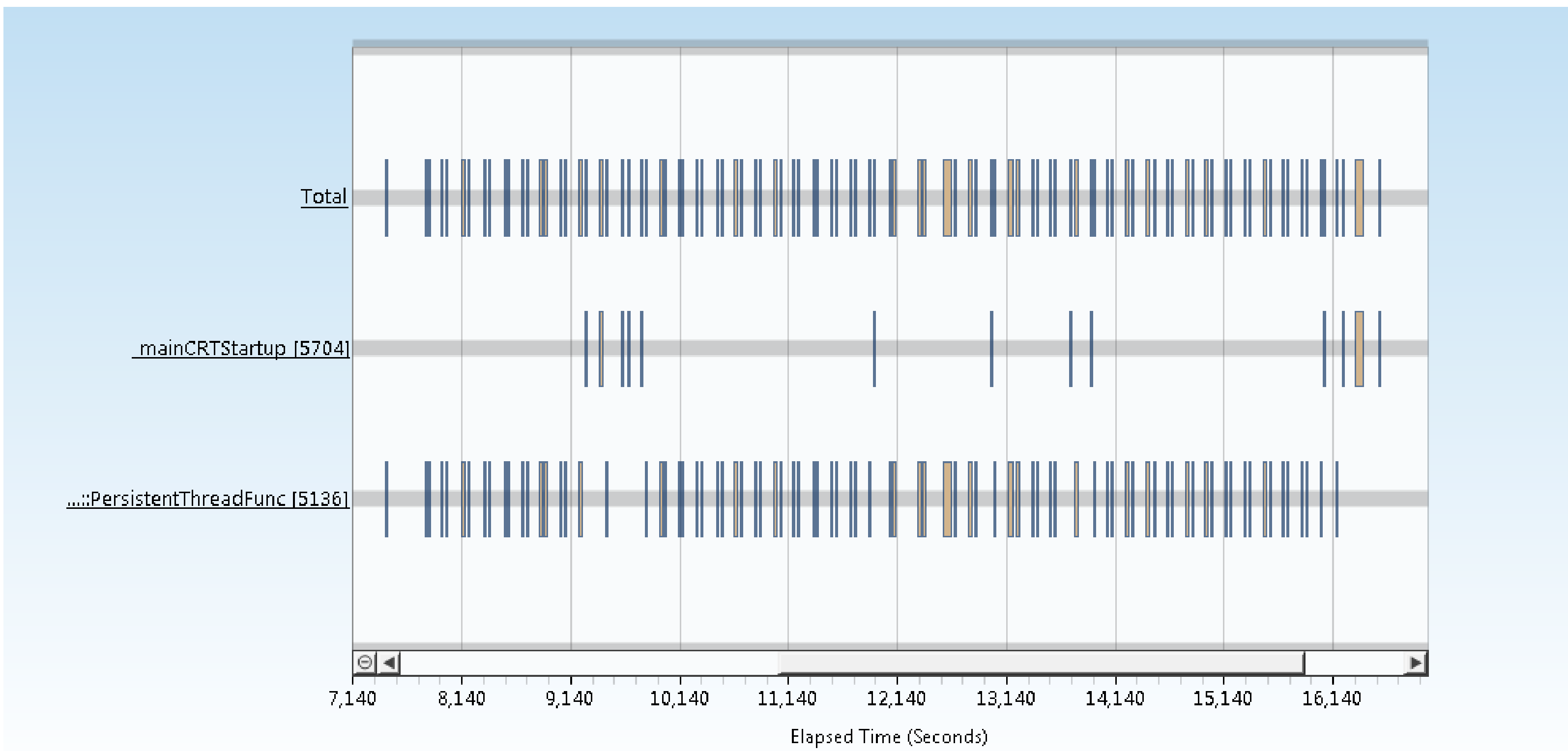
- If threads compete for resources, they can get stalled:

Most Contended Resources

Resources with the highest number of total contentions

Name	Contentions %	Contentions
Handle 1	100,00	97

Contentions of "Handle 1"



Performance Analyzer: Thread Contention (2/2)

- Reason for this contention: OpenMP Barrier

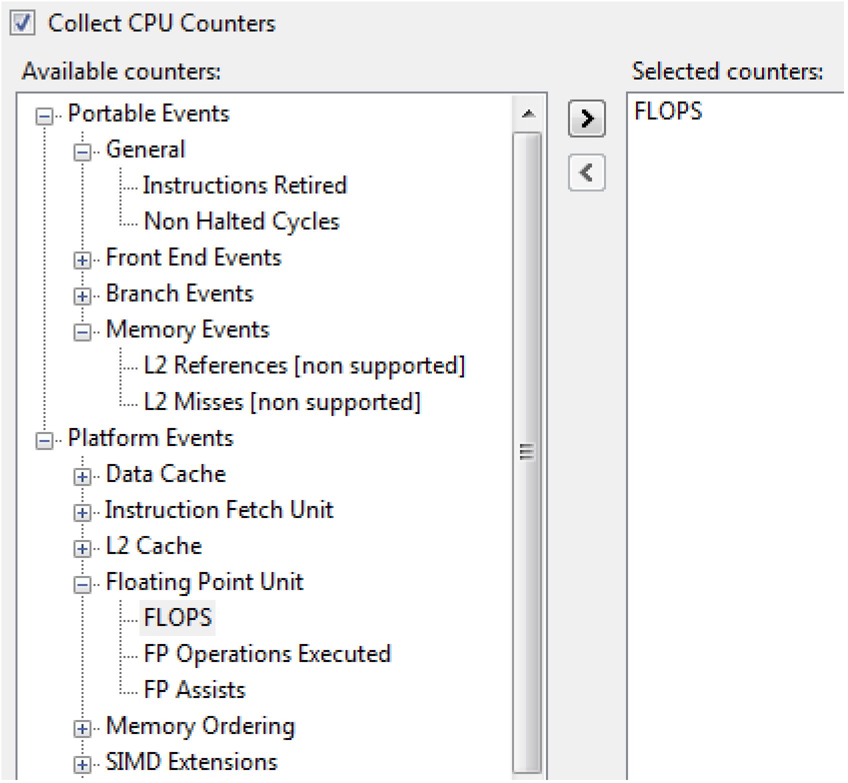
Function Name	Module Name	Inclusive Contentions	Exclusive Contentions	Inclusive Blocked Time	Exclusive Blocked Time
jacobi_aut.exe		97	0	1.518	0
Jacobiomp1	jacobi_aut.exe	84	0	1.346	0
__vcomp_barrier	VCOMP90D.DLL	84	84	1.346	1.346
_mainCRTStartup	jacobi_aut.exe	13	0	172	0
Unknown Frame(s)	UNKNOWN	13	0	172	0
Jacobiomp1	jacobi_aut.exe	13	0	172	0
__vcomp_barrier	VCOMP90D.DLL	13	13	172	172

  – Support for OpenMP constructs is not yet optimal

- This analysis is crucial if you do your own synchronization!

Performance Analyzer: More future features...

- Performance tuning can be a never ending story, so you need metrics to decide where to work / when to stop: Hardware Counter Information.



L2 information can be used to measure the memory bandwidth consumed by the application → is your scalability limited by the system architecture?

The FLOPS rate is good to estimate how efficient the code runs!

...

Visual Studio Configurations (1/3)

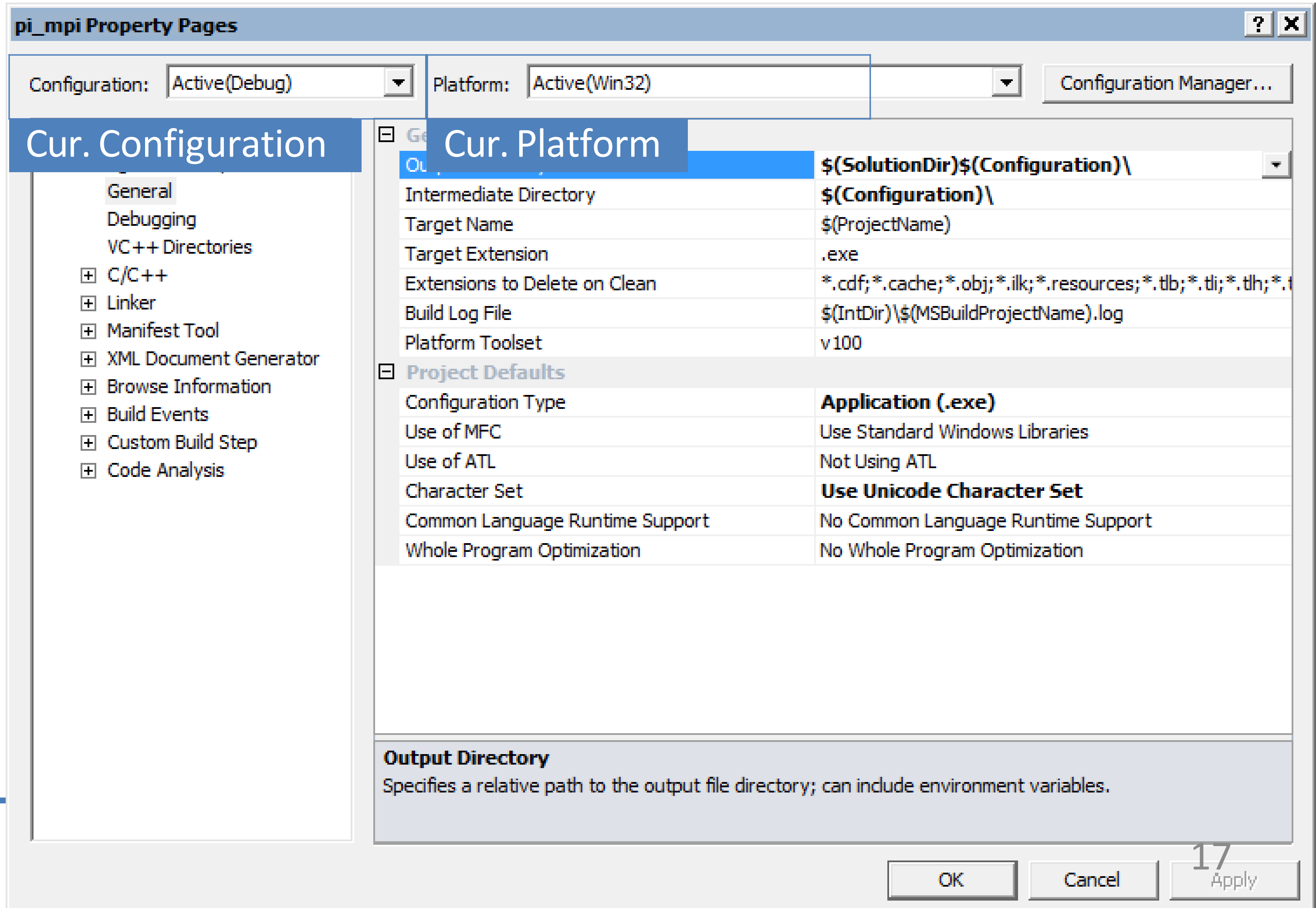
- The set of compiler options is managed in a *Configuration*.
- There are two configurations pre-defined: *Debug* and *Release*.
  - Debug: typical options for debugging, no optimization.
  - Release: debugging still possible, some optimization options.

- The compile process can be triggered by right-clicking on the project and choosing *Build*. Or from the menu: *Build → Build <projectname>*.
- *Build → Build Solution* builds all projects in the solution.

- During and after the compile process compiler output (informational messages, warnings, errors) is displayed in the tool windows *Output* or *Error List*.
- By double-clicking on such a message, the cursor jumps to the corresponding place in the code.

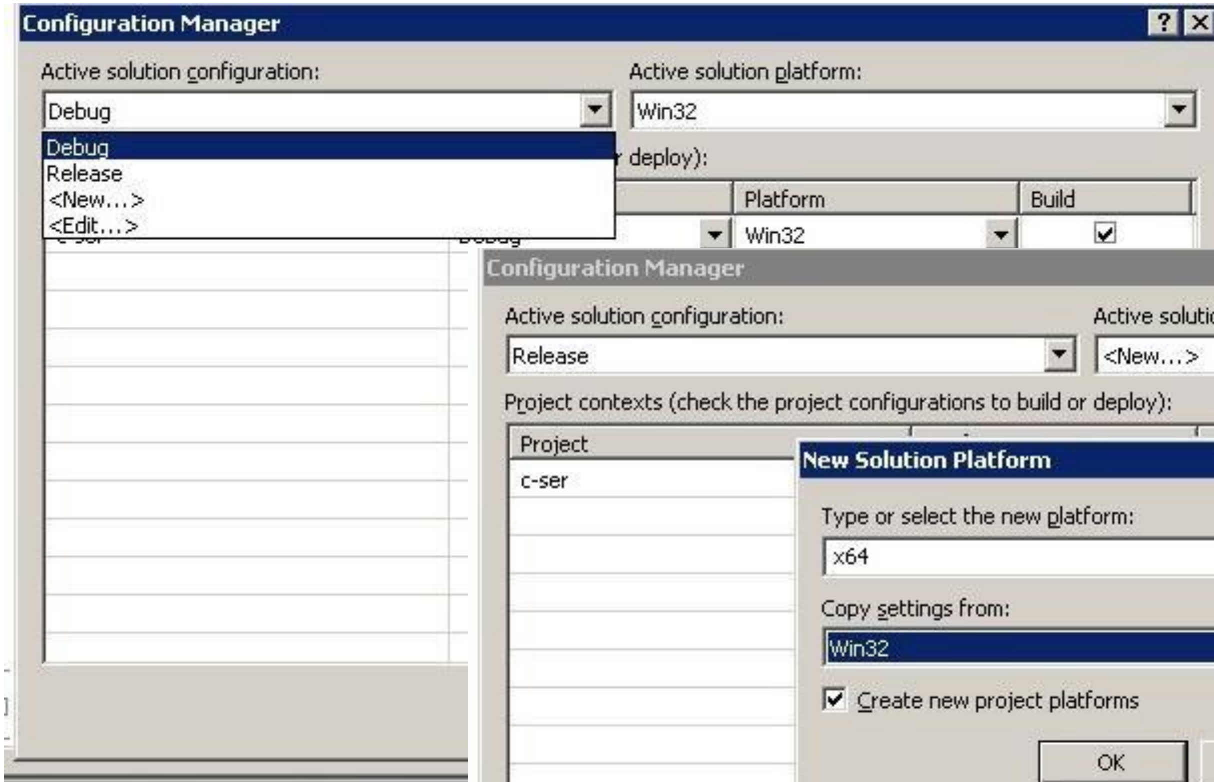
Visual Studio Configurations (2/3)

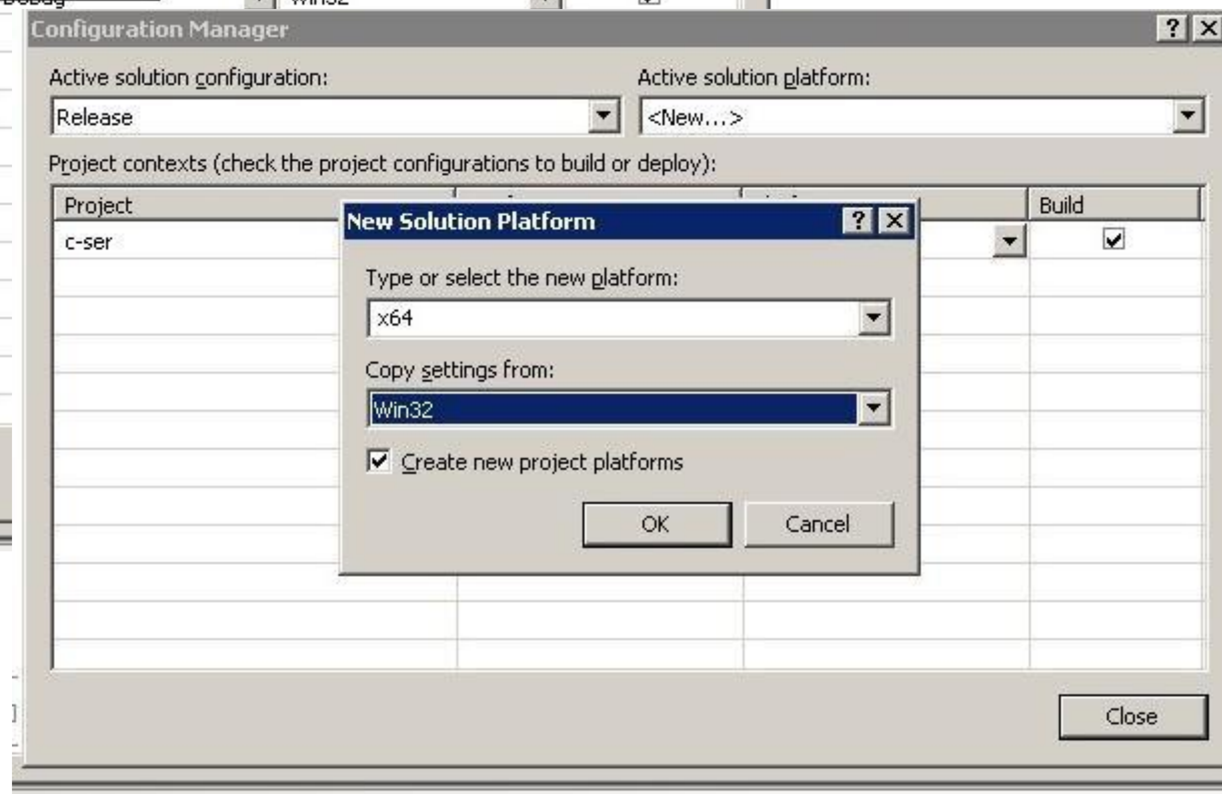
- Right-clicking on a project and choosing Properties leads to the project configuration dialog.



Visual Studio Configurations (3/3)

Build → Configuration Manager:

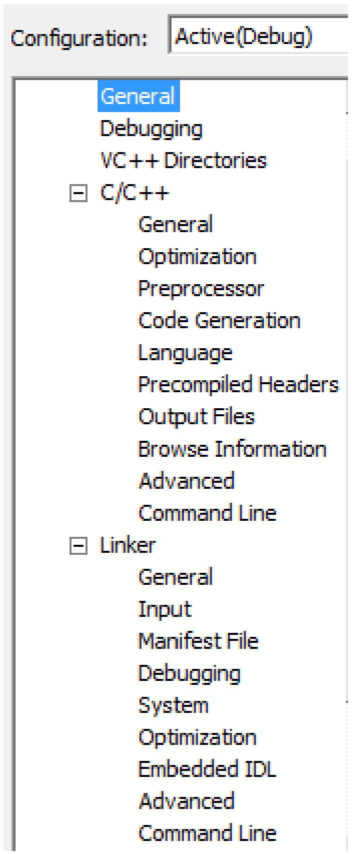




Only Win32 and x64 are supported on our Cluster, not Itanium.

You can create your own configurations.

Microsoft C/C++-specific settings



- Important General Settings:
  - C/C++ → General
    - Addition Include Directories: Include Path
  - Linker → General
    - Additional Library Directories: Library Path
  - Linker → Input
    - Additional Dependencies: Libraries to be used
- Important Optimization Settings:
  - C/C++ → Optimization
    - Optimization: General Optimization Level
    - Inline Function Expansion: Inlining
  - C/C++ → Code Generation
    - Enable Enhanced Instruction Set: Vectorization

Portable Time Measurement (1/3)

- Porting applications from Unix to Windows (or the other way around) can be quite hard … but it was not for most user codes (HPC) we tried on Windows.
  (1) The most common problem was time measurement as `gettimeofday()` is not available on Windows,
  (2) followed by directory management issues where ‚/' instead of ‚\' had been used before.

- In most cases we attacked (2) using `#ifdefs`.
- Handling (1) depends on the programming language:
  – C++: We have written a version of `double realtime()` for Windows and Unix.

Portable Time Measurement (2/3)

```
#ifdef WIN32
   #include <Windows.h>
   #define Li2Double(x) ((double)((x).HighPart) * 4.294967296E9 + \
       (double)((x).LowPart))
#else
  #include <sys/time.h>
  #include <time.h>
#endif

double realtime (void) {
#ifdef WIN32
   LARGE_INTEGER time, freq;
   double dtime, dfreq;
   if (QueryPerformanceCounter(&time) == 0) { … error … }
   if (QueryPerformanceFrequency(&freq) == 0) { … error … }
   return Li2Double(time) / Li2Double(freq);
#else
   struct timeval tv;
   gettimeofday(&tv, (struct timezone*)0);
   return ((double)tv.tv_sec + (double)tv.tv_usec / 1000000.0 );
}
```

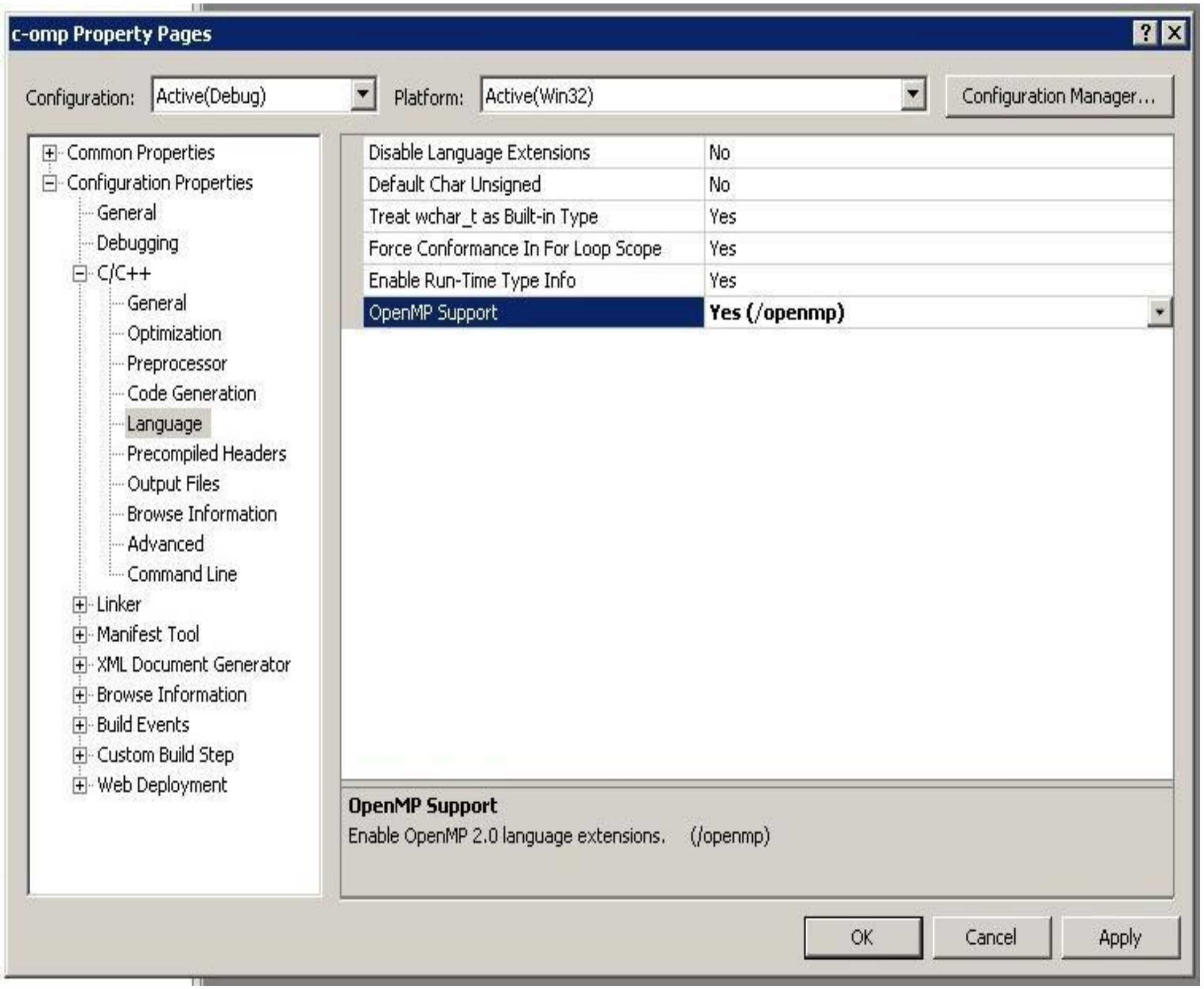
Portable Time Measurement (3/3)

- Taking time the OpenMP way:

```
#include <omp.h>
   ...
   double t1, t2 elapsed_seconds;
   t1 = omp_get_wtime();
   ...
   t2 = omp_get_wtime();
   elapsed_seconds = t2 - t1;
```

Enabling OpenMP (1/3)

- OpenMP support has to be enabled in a configuration:



OpenMP 2.0 / 2.5:

- VS2005 C/C++
- VS2008 C/C++
- VS2010 C/C++

OpenMP 3.0:

- Intel C/C++
- Intel FORTRAN

Enabling OpenMP (2/3)

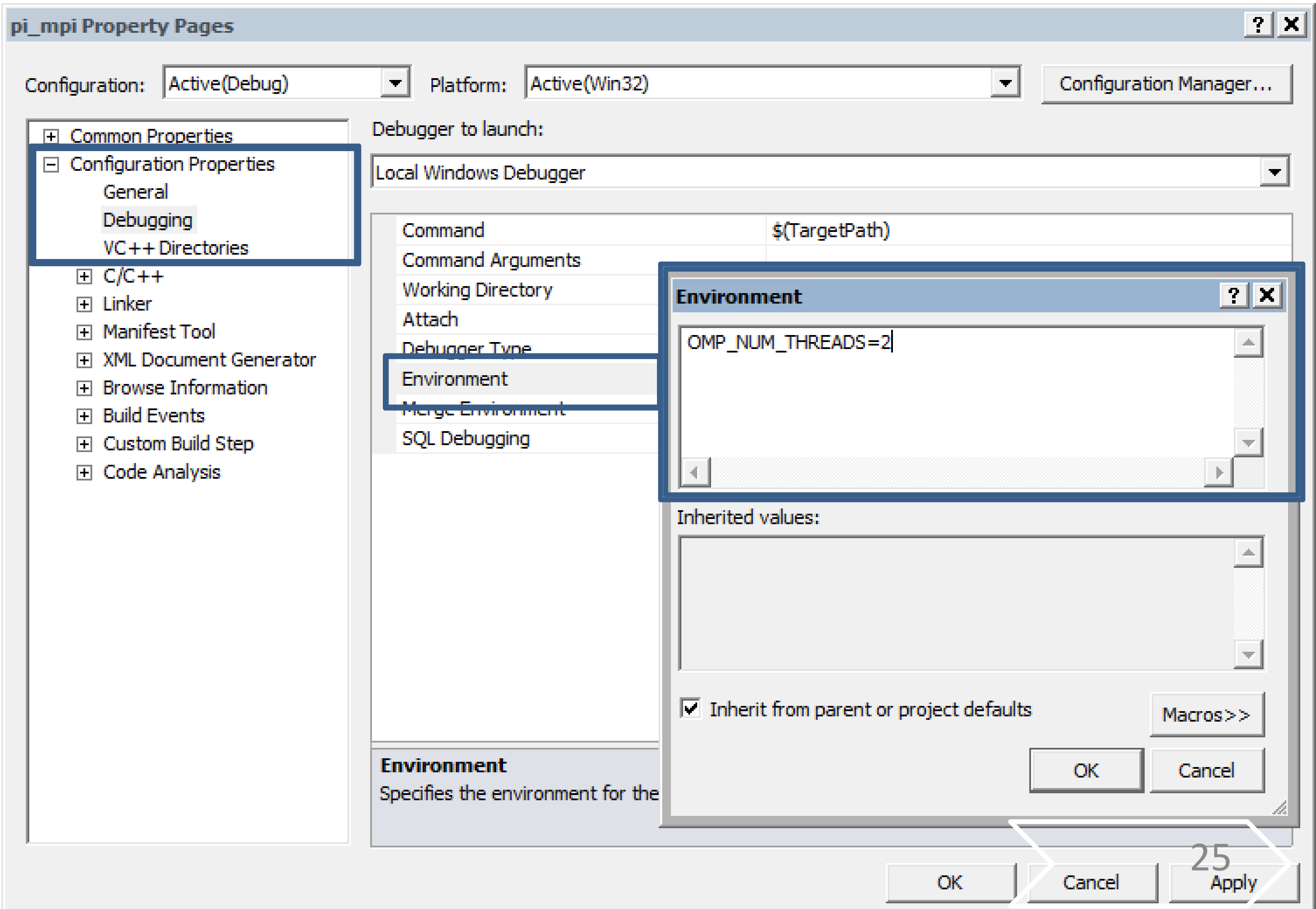
- Known problem with Visual Studio and OpenMP:

- The message appears if an OpenMP program has been compiled with OpenMP support enabled, but `omp.h` had not been included.

- Solution: include `omp.h` in at least one file per project.

Enabling OpenMP (3/3)

- Setting the number of threads for debugging of OpenMP programs: set environment variable `OMP_NUM_THREADS`.



Debugging OpenMP Programs (1/4)

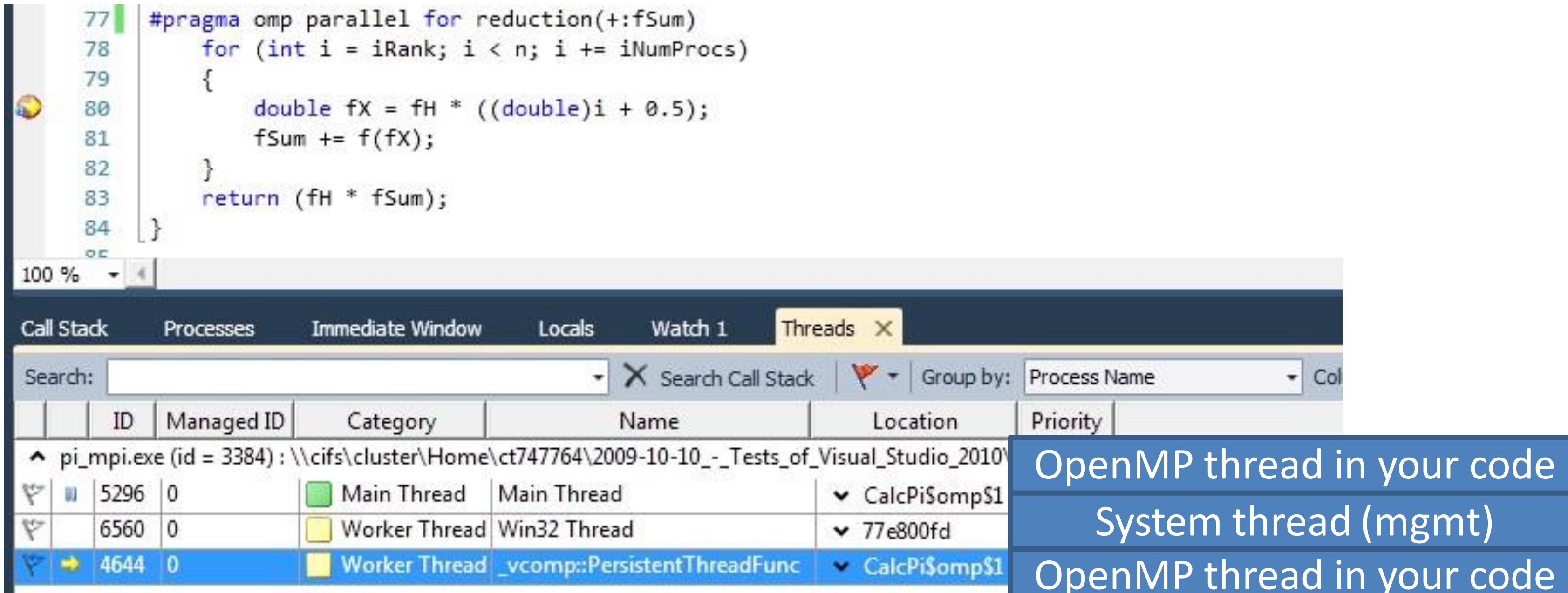
- Debugging of OpenMP applications in Visual Studio works with all compilers: the Microsoft C/C++ compiler, the Intel C/C++ compiler and the Intel Fortran compiler.
- Note: If you use one of the Intel compilers or VS21010 and start a program with `n` threads, you will see `n+1` threads (one management thread).
- We advise you to compile without any optimization for debugging, that means use the pre-configured *Debug* configuration and just enable OpenMP.
- Control debugging:

» Start / Continue, Break, Stop, Restart

» Show next statement, Step Into, Step Over, Step Out

Debugging OpenMP Programs (2/4)

- All threads stop at a breakpoint (first thread encounters it).
  - You can open the *Threads* register from the menu via *Debug* → *Windows* → *Threads*. Double-clicking a thread will select it.



  - If you want a thread to stand still you *Freeze* it (context menu).
  - If you want it to continue *Thaw* it.

Debugging OpenMP Programs (3/4)

- For all threads, you can view the *local* and *shared* variables.
  - The *Locals* register contains all variables of the current scope.
  - The *Autos* register contains a set of interesting variables guessed by the compiler – remarkably good.
- Some limitations when using the Intel Fortran compiler:
  - The *Autos* register is empty, *Locals* is working fine.
  - One can not (at least sometimes) identify the management thread by the name – it is the one you get an error message of no source code being available if you select it ;-)
  - Sometimes expressions have to be updated because of (possible) compiler optimization.

Debugging OpenMP Programs (4/4)

- The Parallel Stacks window is a new feature of VS2010 and in the menu under *Debug* → *Windows* → *Parallel Stacks*: