Chapter 19 OpenMP

Speaker: Lung-Sheng Chien

Reference: [1] OpenMP C and C++ Application Program Interface v2.0
[2] OpenMP C and C++ Application Program Interface v3.0
[3] OpenMP forum, http://www.openmp.org/forum/
[4] OpenMP tutorial: https://computing.llnl.gov/tutorials/openMP/
[5] Getting Started with OpenMP:
http://rac.uits.iu.edu/hpc/openmp_tutorial/C/

OutLine

- OpenMP introduction
  - shared memory architecture
  - multi-thread

- Example 1: hello world
- Example 2: vector addition
- enable openmp in vc2005
- Example 3: vector addition + Qtime
- Example 4: matrix multiplication
- Example 5: matrix multiplication (block version)

What is OpenMP

http://en.wikipedia.org/wiki/OpenMP

- The **OpenMP** (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

- OpenMP is a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

- An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI).
  OpenMP: shared memory
  MPI: distributed memory

History of OpenMP

- The OpenMP Architecture Review Board (ARB) published its first API specifications, OpenMP for Fortran 1.0, in October 1997. October the following year they released the C/C++ standard.

- 2000 saw version 2.0 of the Fortran specifications with version 2.0 of the C/C++ specifications being released in 2002.

- Version 2.5 is a combined C/C++/Fortran specification that was released in 2005.

- Version 3.0, released in May, 2008, is the current version of the API specifications. Included in the new features in 3.0 is the concept of *tasks* and the **task** construct. These new features are summarized in Appendix F of the OpenMP 3.0 specifications.

Goals of OpenMP

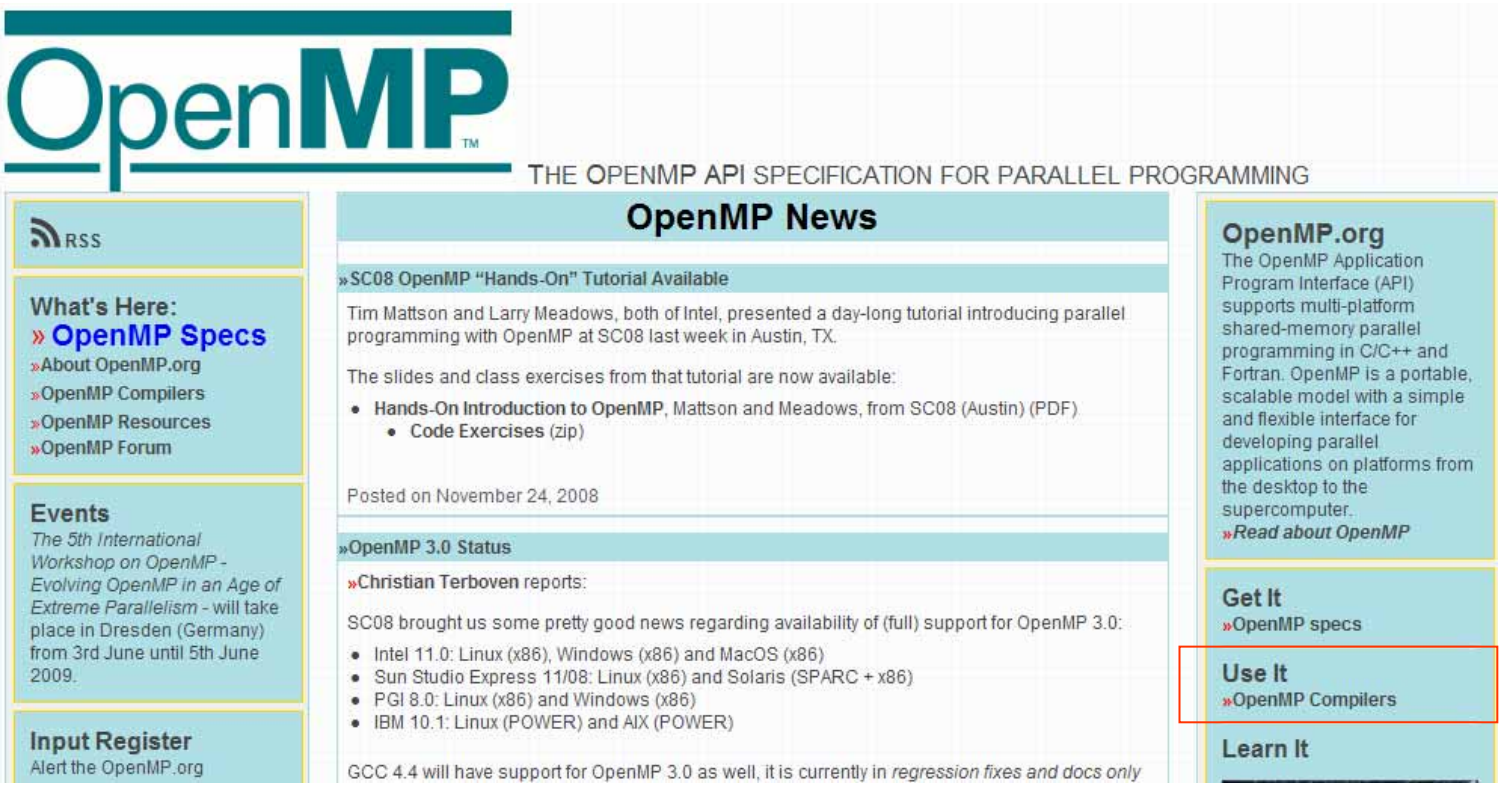
- **Standardization:**
  Provide a standard among a variety of shared memory architectures/platforms.

- **Lean and Mean:**
  establish a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.

- **Ease of Use:**
  -Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
  -Provide the capability to implement both coarse-grain and fine-grain parallelism

- **Portability:**
  -Supports Fortran (77, 90, and 95), C, and C++
  -Public forum for API and membership

Website: http://openmp.org/wp/



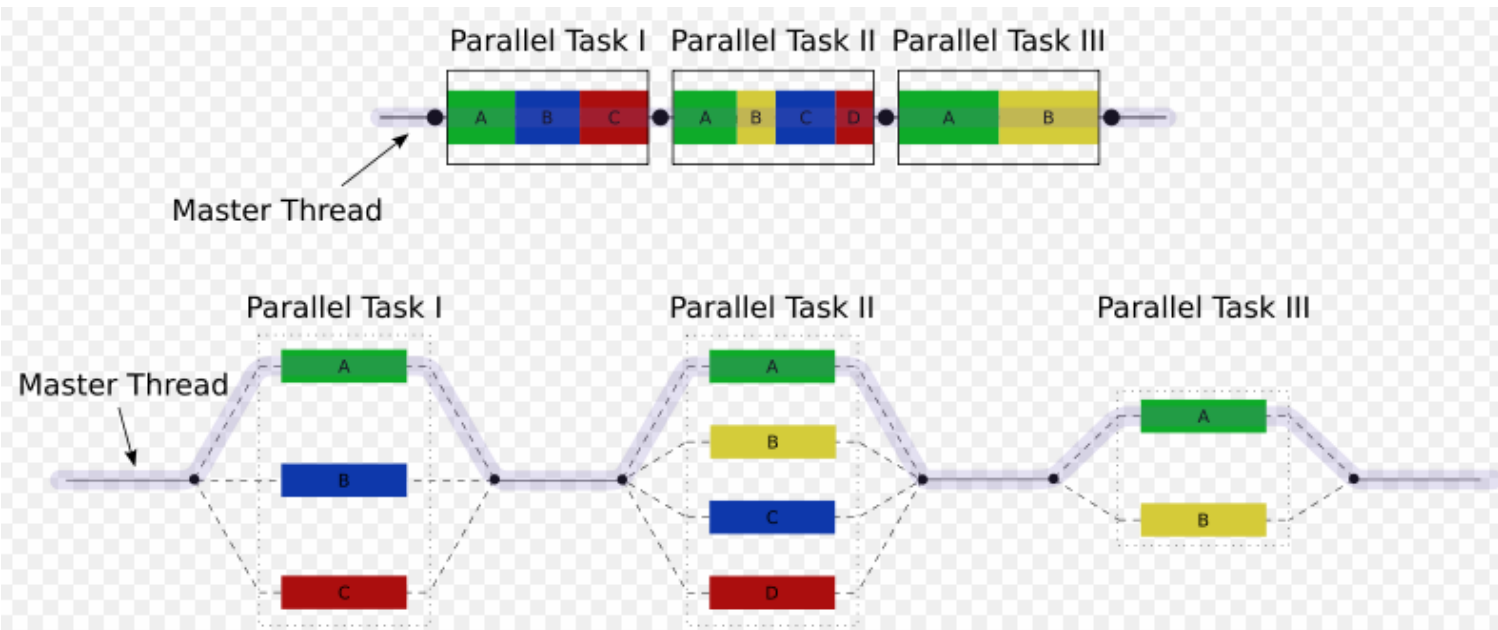
Vendor/Source	Compiler	Information
»GNU	gcc (4.3.2)	Free and open source - Linux, Solaris, AIX, MacOSX, Windows Compile with `-fopenmp` »More information
»Intel	C/C++ / Fortran (10.1)	Windows, Linux, and MacOSX. Compile with `-Qopenmp` on Windows, or just `-openmp` on Linux or Mac OSX »More information

OpenMP forum: http://www.openmp.org/forum/

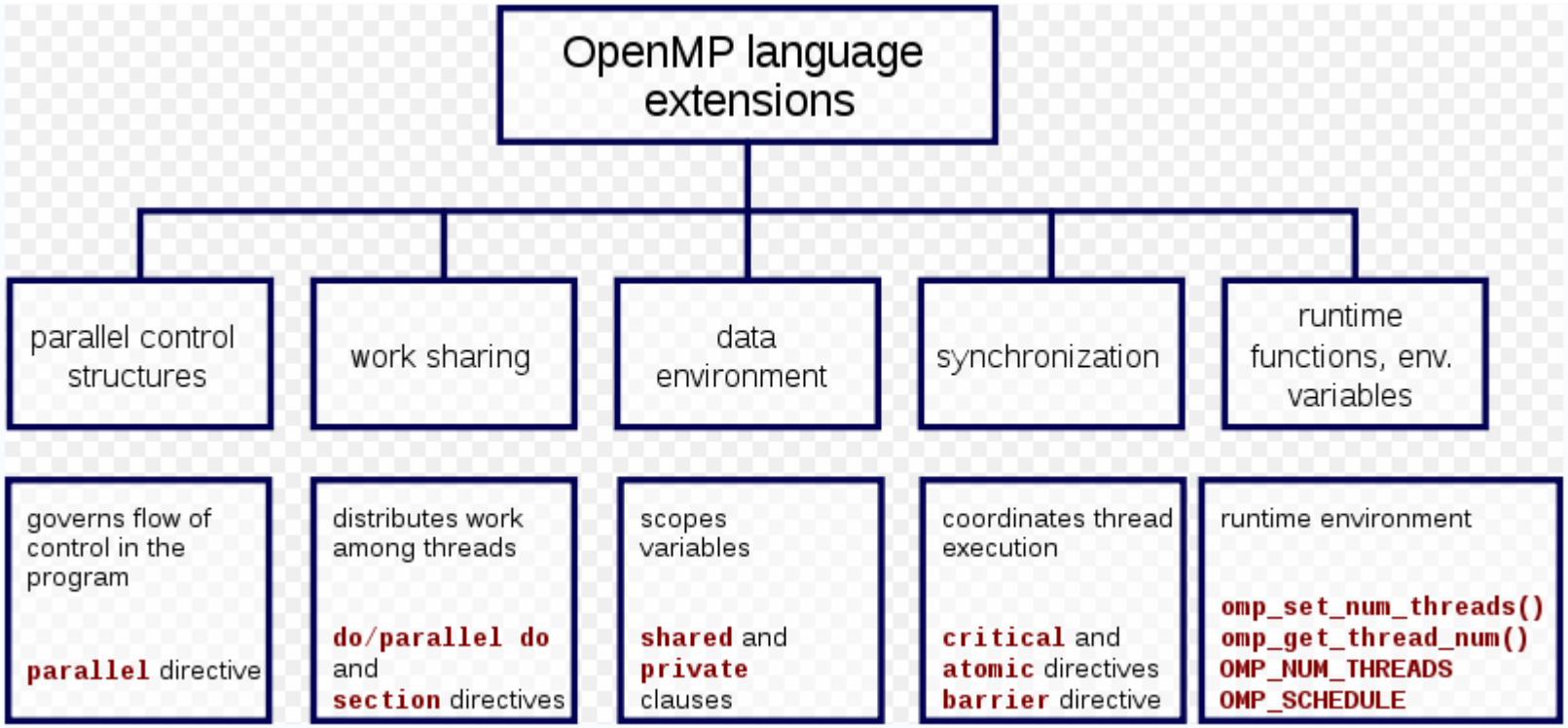
Please register in this forum and browse articles in “General” item

Multithread (多執行緒)

- OpenMP is an implementation of multithreading, a method of parallelization whereby the master "thread" (a series of instructions executed consecutively) "forks" a specified number of slave "threads" and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.
- The runtime environment allocates threads to processors depending on usage, machine load and other factors. The number of threads can be assigned by the runtime environment based on environment variables or in code using functions. The OpenMP functions are included in a header file labelled "omp.h" in C/C++



Core elements



A compiler directive in C/C++ is called a *pragma* (pragmatic information). It is a preprocessor directive, thus it is declared with a hash (#). Compiler directives specific to OpenMP in C/C++ are written in codes as follows:

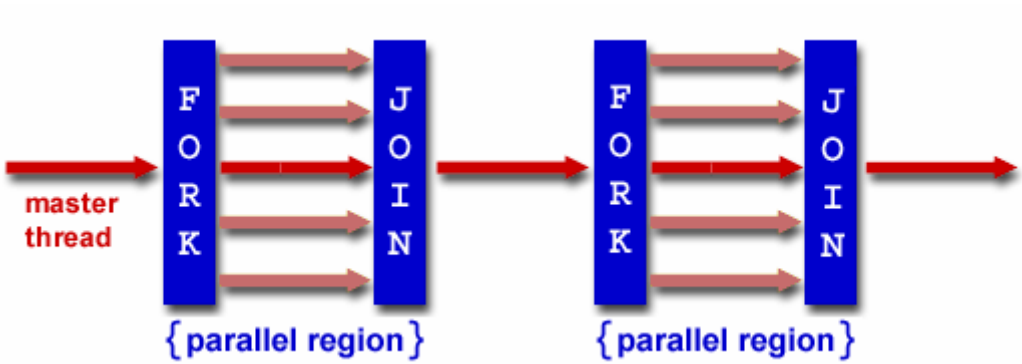
```
#pragma omp <rest of pragma>
```

OpenMP programming model [1]

- **Shared Memory, Thread Based Parallelism:**
  OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. A shared memory process consists of multiple threads.

- **Explicit Parallelism:**
  OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

- **Fork - Join Model:**
  - OpenMP uses the fork-join model of parallel execution
  - All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered
  - ***FORK*****:** the master thread then creates a ***team*** **of parallel threads**
  - The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads
  - ***JOIN*****:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread



OpenMP programming model [2]

- **Compiler Directive Based:**
  OpenMP parallelism is specified through the use of compiler directives.

- **Nested Parallelism Support:**
  - The API provides for the placement of parallel constructs inside of other parallel constructs
  - Implementations may or may not support this feature.

- **Dynamic Threads:**
  -The API provides for dynamically altering the number of threads which may used to execute different parallel regions
  - Implementations may or may not support this feature.

- **I/O:**
  -OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
  -If every thread conducts I/O to a different file, the issues are not as significant.
  -It is entirely up to the programmer to insure that I/O is conducted correctly within the context of a multi-threaded program.

- **FLUSH Often?:**
  -OpenMP provides a "relaxed-consistency" and "temporary" view of thread memory (in their words). In other words, threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time.
  -When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is FLUSHed by all threads as needed.

OutLine

- OpenMP introduction

- Example 1: hello world
  - parallel construct

- Example 2: vector addition
- enable openmp in vc2005
- Example 3: vector addition + Qtime
- Example 4: matrix multiplication
- Example 5: matrix multiplication (block version)

Example 1: hello world [1]

hello.c

```
1
2  #include <omp.h>
3  #include <stdio.h>
4
5  int main (int argc, char *argv[])
6  {
7    #pragma omp parallel
8    {
9      printf("Hello World\n");
10   }
11   return 0;
12 }
13
```

Makefile

```
1
2 hello: hello.c
3    icpc -openmp -mp -c hello.c
4    icpc -openmp -o hello hello.o
5
```

header file “omp.h” is necessary for OpenMP programming

MSDN library 2005

The **#pragma** directives offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages. Pragmas are machine- or operating system-specific by definition, and are usually different for every compiler.

If the compiler finds a pragma it does not recognize, it issues a warning, but compilation continues.

man icpc

```
-openmp
        Enable the parallelizer to generate multi-threaded code based on
        the OpenMP* directives. The code can be executed in parallel  on
        both uniprocessor and multiprocessor systems. The -openmp option
        works with both -O0 (no optimization) and any optimization level
        of -O1, -O2 (default) and -O3. Specifying -O0 with -openmp helps
        to debug OpenMP applications.
```

Example 1: hello world [2]

```
[macrold@quartet2 hello_wordl]$ ls
Makefile  hello.c
[macrold@quartet2 hello_wordl]$ make hello
icpc -openmp -mp -c hello.c
hello.c(7): (col. 3) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
icpc -openmp -o hello hello.o
[macrold@quartet2 hello_wordl]$ ls
Makefile  hello  hello.c  hello.o
[macrold@quartet2 hello_wordl]$ ./hello
Hello World
Hello World
Hello World
Hello World
[macrold@quartet2 hello_wordl]$
```

hello.c

```
1
2  #include <omp.h>
3  #include <stdio.h>
4
5  int main (int argc, char *argv[])
6  {
7    #pragma omp parallel
8    {
9      printf("Hello World\n");
10   }
11   return 0;
12 }
13
```

Machine quartet2 has 4 cores

```
top - 11:17:15 up 14 days, 22:54,  2 users,  load average: 0.00, 0.00, 0.00
Tasks: 138 total,   1 running, 137 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8201628k total,  3524316k used,  4677312k free,   219744k buffers
Swap:  8193140k total,        0k used,  8193140k free,  2849960k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
14400 macrold   20   0 18936 1208  900 R    0  0.0   0:00.01 top
    1 root      20   0 10328  688  580 S    0  0.0   0:05.36 init
    2 root      15  -5     0    0    0 S    0  0.0   0:00.00 kthreadd
```

```
[macrold@quartet2 hello_wordl]$ cat /proc/cpuinfo
```

```
model name      : Intel(R) Core(TM)2 Quad CPU    Q6600  @ 2.40GHz
stepping        : 11
cpu MHz         : 1596.000
cache size      : 4096 KB
```

Example 1: hello world [3]

octet1

```
[macrold@octet1 hello_wordl]$ ls
Makefile  hello  hello.c  hello.o
[macrold@octet1 hello_wordl]$ ./hello
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Machine octet1 has 8 cores (two quad-core)

```
top - 09:58:07 up 80 days, 18:39,  1 user,  load average: 0.00, 0.00, 0.00
Tasks: 194 total,   1 running, 193 sleeping,   0 stopped,   0 zombie
Cpu0  :  1.5%us,  0.1%sy,  0.0%ni, 98.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  1.1%us,  0.1%sy,  0.0%ni, 98.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.9%us,  0.1%sy,  0.0%ni, 98.8%id,  0.1%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.5%us,  0.0%sy,  0.0%ni, 99.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  :  1.5%us,  0.1%sy,  0.0%ni, 98.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu5  :  0.8%us,  0.0%sy,  0.0%ni, 99.1%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  :  1.1%us,  0.2%sy,  0.0%ni, 98.6%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  :  0.8%us,  0.0%sy,  0.0%ni, 99.2%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  65342468k total, 11726988k used, 53615480k free,   416284k buffers
Swap: 67103496k total,    30464k used, 67073032k free, 10182848k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0 10328  280  252 S    0  0.0   0:13.96 init
    2 root      15  -5     0    0    0 S    0  0.0   0:00.03 kthreadd
```

Question 1: How to impose number of threads in code?

environment variable *OMP_NUM_THREADS*

```
[macrold@quartet2 hello_wordl]$
[macrold@quartet2 hello_wordl]$ set | grep OMP_NUM
OMP_NUM_THREADS=4
[macrold@quartet2 hello_wordl]$
```

```
[macrold@octet1 hello_wordl]$
[macrold@octet1 hello_wordl]$ set | grep OMP_NUM
OMP_NUM_THREADS=8
[macrold@octet1 hello_wordl]$
```

hello.c

```
1
2   #include <omp.h>
3   #include <stdio.h>
4
5   int main (int argc, char *argv[])
6   {
7     #pragma omp parallel
8     {
9       printf("Hello World\n");
10    }
11    return 0;
12  }
13
```

Example 1: hello world [4]

Question 2: How can we run the same code in sequential mode?

hello.c

```
1
2  #include <omp.h>
3  #include <stdio.h>
4
5  int main (int argc, char *argv[])
6  {
7    #pragma omp parallel
8    {
9      printf("Hello World\n");
10   }
11   return 0;
12 }
13
```

Makefile

```
1
2 hello: hello.c
3   icpc -openmp -mp -c hello.c
4   icpc -openmp -o hello hello.o
5
6
7 hello_seq: hello.c
8   icpc -mp -c hello.c
9   icpc -o hello_seq  hello.o
10
```

sequential version

quartet2

```
[macrold@quartet2 hello_wordl]$
[macrold@quartet2 hello_wordl]$ make hello_seq
icpc -mp -c hello.c
hello.c(7): warning #161: unrecognized #pragma
    #pragma omp parallel
            ^

icpc -o hello_seq  hello.o
[macrold@quartet2 hello_wordl]$ ls
Makefile  hello  hello.c  hello.o  hello_seq
[macrold@quartet2 hello_wordl]$ ./hello_seq
Hello World
[macrold@quartet2 hello_wordl]$
```

octet1

```
[macrold@octet1 hello_wordl]$
[macrold@octet1 hello_wordl]$ ./hello_seq
Hello World
[macrold@octet1 hello_wordl]$
```

only one core executes

Example 1: hello world [5]

Question 3: How can we issue number of threads explicitly in code?

hello.c

```
1
2  #include <omp.h>
3  #include <stdio.h>
4
5  int main (int argc, char *argv[])
6  {
7    int th_id, nthreads;
8
9    #pragma omp parallel private(th_id) num_threads(5)
10   {
11     th_id = omp_get_thread_num();
12     printf("Hello World from thread %d\n", th_id);
13
14     #pragma omp barrier
15
16     if ( th_id == 0 ) {
17       nthreads = omp_get_num_threads();
18       printf("There are %d threads\n",nthreads);
19     }
20   }
21   return 0;
22 }
```

every thread has its own copy

use 5 threads (explicit) to execute concurrently

synchronization

wait until all 5 threads execute "printf" statement.

The **barrier** directive synchronizes all the threads in a team. When encountered, each thread in the team waits until all of the others have reached this point. The syntax of the **barrier** directive is as follows:

```
#pragma omp barrier new-line
```

After all threads in the team have encountered the barrier, each thread in the team begins executing the statements after the barrier directive in parallel.

Example 1: hello world [6]

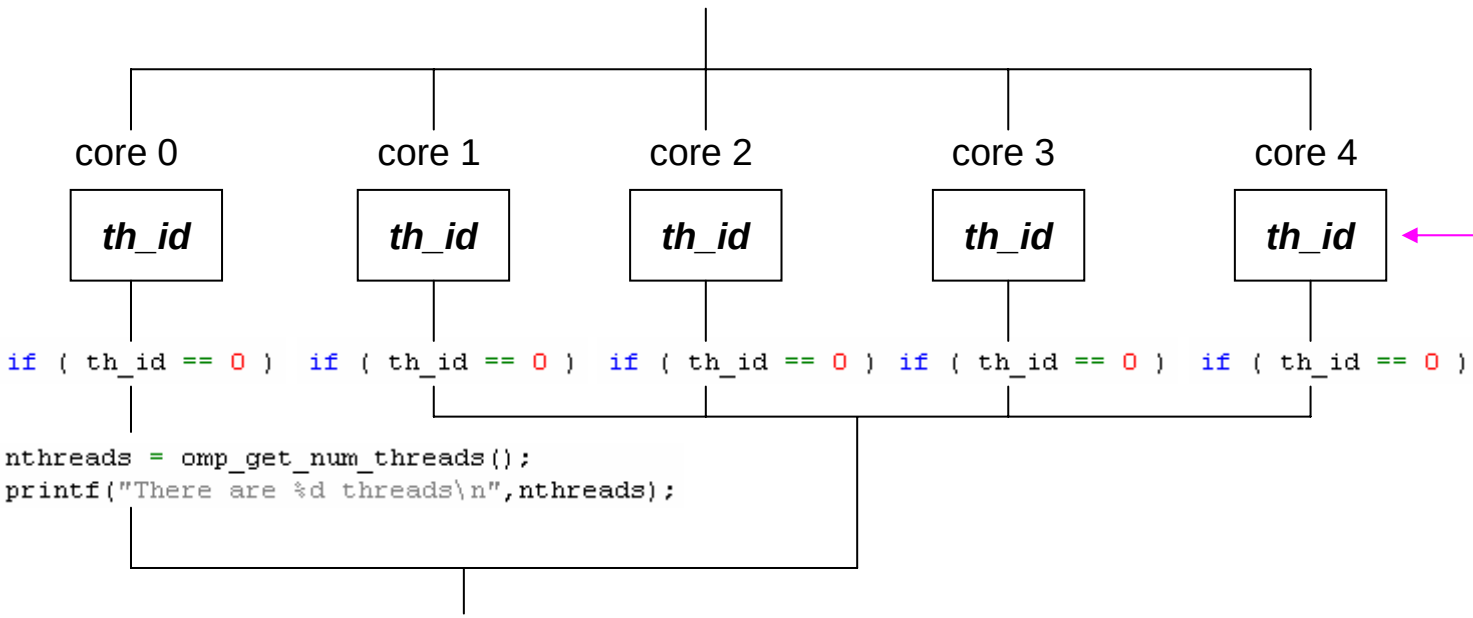
quartet2

```
[macrold@quartet2 hello_word2]$
[macrold@quartet2 hello_word2]$ make hello
icpc -openmp -mp -c hello.c
hello.c(8): (col. 3) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
icpc -openmp -o hello hello.o
[macrold@quartet2 hello_word2]$ ls
Makefile  hello  hello.c  hello.o
[macrold@quartet2 hello_word2]$ ./hello
Hello World from thread 1
Hello World from thread 2
Hello World from thread 3
Hello World from thread 0
Hello World from thread 4
There are 5 threads
[macrold@quartet2 hello_word2]$
```

octet1

```
[macrold@octet1 hello_word2]$
[macrold@octet1 hello_word2]$ ./hello
Hello World from thread 3
Hello World from thread 0
Hello World from thread 4
Hello World from thread 2
Hello World from thread 1
There are 5 threads
[macrold@octet1 hello_word2]$
```

```
th_id = omp_get_thread_num();
printf("Hello World from thread %d\n", th_id);
```

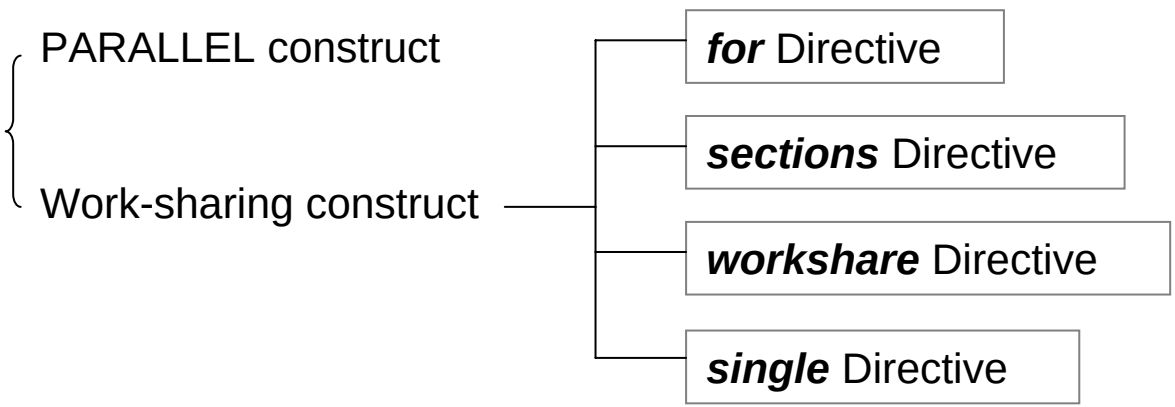


Directive Format

The syntax of an OpenMP directive is formally specified by the grammar

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

Each directive starts with **#pragma omp**, to reduce the potential for conflict with other (non-OpenMP or vendor extensions to OpenMP) *pragma* directives with the same names. White space can be used before and after the #, and sometimes white space must be used to separate the words in a directive. Preprocessing tokens following the **#pragma omp** are subject to macro replacement.



Conditional compilation

```
#ifdef _OPENMP
iam = omp_get_thread_num() + index;
#endif
```

Parallel construct

```
#pragma omp parallel [clause ...]  newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)

   structured_block
```

```
#pragma omp parallel private(th_id) num_threads(5)
{
  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);
  }
}
```

- The number of physical processors hosting the threads is implementation-defined. Once created, the number of threads in the team remains constant for the duration of that parallel region.
- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel region. Only the master thread of the team continues execution at the end of a parallel region.

How many threads

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - evaluation of the *IF* clause
  - setting of the *NUM_THREADS* clause
  - use of the *omp_set_num_threads()* library function
  - setting of the *OMP_NUM_THREADS* environment variable
  - implementation default - usually the number of CPUs on a node, though it could be dynamic.

- Threads are numbered from 0 (master thread) to N-1.

- Master thread is numbered as 0.

Question 4: How to write parallel code such that it is independent of number of cores of host machine?

Question 5: What happens if number of threads is larger than number of cores of host machine?

Private clause

The PRIVATE clause declares variables in its list to be private to each thread.

"private variable" means each thread has its own copy and cannot interchange information.

```
#pragma omp parallel private(th_id) num_threads(5)
{
  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);
  }
}
```

- PRIVATE variables behave as follows:
  - a new object of the same type is declared once for each thread in the team
  - all references to the original object are replaced with references to the new object
  - variables declared PRIVATE are uninitialized for each thread

Exercise 1: modify code of hello.c to show "every thread has its own private variable ***th_id***", that is, shows th_id has 5 copies.

Exercise 2: modify code of hello.c, remove clause "private (th_id)" in #pragma directive, what happens? Can you explain?

OutLine

- OpenMP introduction
- Example 1: hello world

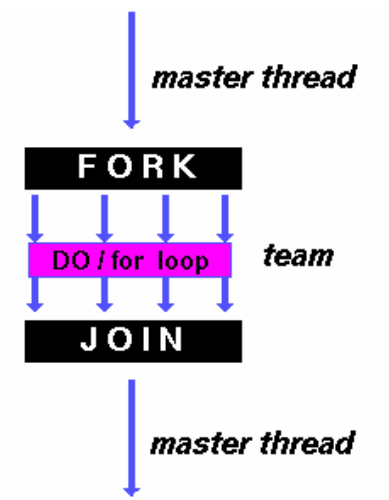
- Example 2: vector addition
  - work-sharing construct: for Directive

- enable openmp in vc2005
- Example 3: vector addition + Qtime
- Example 4: matrix multiplication
- Example 5: matrix multiplication (block version)

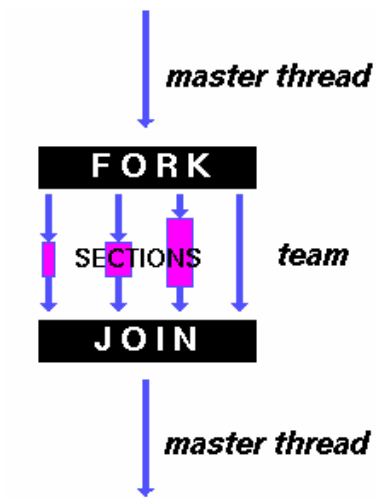
Work-sharing construct

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it
- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct

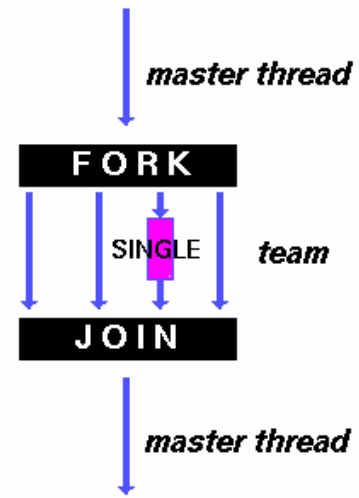
for: shares iterations of a loop across the team.
A type of data parallelism



sections: breaks work into separate, discrete sections. Each section is executed by a thread.
A type of functional parallelism



single: serializes a section of code.



Example 2: vector addition [1]

vecadd.c

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

double walltime( double *t0 ) ;
void randomInit( float* data, int size) ;

int main(int argc, char *argv[] )
{
  long int N = 200000000 ;
  int  thread_num = 4 ;
  long int i;
  float *a, *b, *c;
  double startTime, elapsedTime; /* for timing */
  double clockZero = 0.0;

  a = (float*) malloc( sizeof(float)*N ) ; assert(a) ;
  b = (float*) malloc( sizeof(float)*N ) ; assert(b) ;
  c = (float*) malloc( sizeof(float)*N ) ; assert(c) ;

  startTime = walltime( &clockZero );
  randomInit(a, N);
  randomInit(b, N);
  elapsedTime = walltime( &startTime );
  printf("Time to randomize a, b = %6.4f (s)\n", elapsedTime);
```

parameter

walltime.c

```c
#include <sys/time.h>
// return current_time - t0 in seconds
double walltime( double *t0 )
{
  double mic, time;
  double mega = 0.000001;
  struct timeval tp;
  struct timezone tzp;
  static long base_sec = 0;
  static long base_usec = 0;

  (void) gettimeofday(&tp,&tzp);
  if (base_sec == 0)
    {
      base_sec = tp.tv_sec;
      base_usec = tp.tv_usec;
    }

  time = (double) (tp.tv_sec - base_sec);
  mic = (double) (tp.tv_usec - base_usec);
  time = (time + mic * mega) - *t0;
  return(time);
}
```

Tool for measuring time

only valid in Linux system

vecadd.c

```c
// Allocates a matrix with random float entries.
void randomInit(float* data, int size)
{
    for (int i = 0; i < size; ++i){
        data[i] = rand() / (float)RAND_MAX;
    }
}
```

Example 2: vector addition [2]

vecadd.c

```
27
28    startTime = walltime( &clockZero );
29
30  #pragma omp parallel default(none) num_threads(thread_num)  \
31       shared(a,b,c,N) private(i)
32    {
33      #pragma omp for schedule( static ) nowait
34      for (i=0; i < N; i++){
35        c[i] = a[i] + b[i];
36      }
37
38    } /* end of parallel section */
39    elapsedTime = walltime( &startTime );
40    double size = ((double)N)*sizeof(float)/1.E6 ;
41    printf("size = %6.2f (MB) \n", size );
42    printf("thread_num = %d, time for vecadd = %6.4f (s)\n",
43        thread_num,  elapsedTime) ;
44
45    free(a) ; free(b) ; free(c) ;
46    return 0 ;
47  }
```

Makefile

```
1
2 vecadd: vecadd.c walltime.c
3    icpc -openmp -mp -O0  -c vecadd.c
4    icpc -c walltime.c
5    icpc -openmp -o vecadd walltime.o vecadd.o
6
7 clean:
8    rm -f *.o
```

"**O0**" means no optimization

```
[macrold@quartet2 vecadd]$ ls
Makefile  vecadd.c  walltime.c
[macrold@quartet2 vecadd]$ make vecadd
icpc -openmp -mp -O0  -c vecadd.c
vecadd.c(33): (col. 5) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
vecadd.c(30): (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
icpc -c walltime.c
icpc -openmp -o vecadd walltime.o vecadd.o
[macrold@quartet2 vecadd]$ ./vecadd
Time to randomize a, b = 6.4568 (s)
size = 800.00 (MB)
thread_num = 4, time for vecadd = 0.6257 (s)
[macrold@quartet2 vecadd]$
```

shared clause and default clause

The SHARED clause declares variables in its list to be shared among all threads in the team

```
#pragma omp parallel default(none) num_threads(thread_num)   \
     shared(a,b,c,N) private(i)
  {
    #pragma omp for schedule( static ) nowait
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
    }

  } /* end of parallel section */
```

- A shared variable exists in only one memory location and all threads can read or write to that address (every thread can "**see**" the shared variable)
- It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

Question 6: Why index ***i*** must be private variable and ***a,b,c,N*** can be shared variable? What happens if we change ***i*** to shared variable? What happens if we change ***a,b,c,N*** to private variable?

The DEFAULT clause allows the user to specify a default PRIVATE, SHARED, or NONE scope for all variables in the lexical extent of any parallel region.

```
default (shared | none)
```

Work-Sharing construct: for Directive

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                nowait

  for_loop
```

```
#pragma omp parallel default(none) num_threads(thread_num)  \
     shared(a,b,c,N) private(i)
  {
    #pragma omp for schedule( static ) nowait
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
    }

  } /* end of parallel section */
```

- **SCHEDULE**: Describes how iterations of the loop are divided among the threads in the team
  - *static*: loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads
  - *dynamic*: loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another.
    The default chunk size is 1.
- **nowait**: If specified, then threads do not synchronize at the end of the parallel loop.

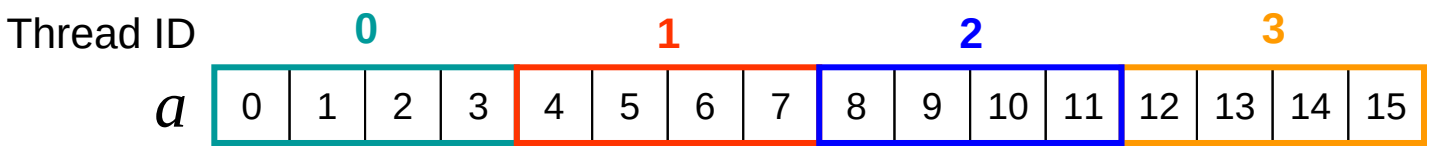
Example of static schedule

Assume we have 16 array elements, say a[16], b[16] and c[16] and use 4 threads

1 no chunk is specified, compiler would divide 16 elements into 4 threads

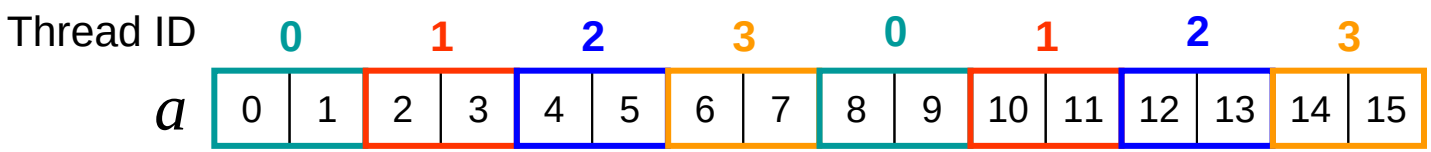
```
#pragma omp parallel default(none) num_threads(thread_num)   \
     shared(a,b,c,N) private(i)
  {
    #pragma omp for schedule( static ) nowait
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
    }

  } /* end of parallel section */
```



2 chunk = 2

```
#pragma omp parallel default(none) num_threads(thread_num)   \
     shared(a,b,c,N) private(i)
  {
    #pragma omp for schedule( static, 2 ) nowait
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
    }
  } /* end of parallel section */
```



Results of example 2

$N = 2 \times 10^8$

compiler: Intel C compiler icpc 10.0

Compiler option: -O0

```
#pragma omp parallel default(none) num_threads(thread_num)   \
     shared(a,b,c,N) private(i)
  {
    #pragma omp for schedule( static ) nowait
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
    }

  } /* end of parallel section */
```

Octet1: $\frac{T(\text{single})}{T(8-core)} = \frac{1.5451}{0.483} = 3.199$

quartet2: $\frac{T(\text{single})}{T(4-core)} = \frac{1.6571}{0.5433} = 3.05$

Question 7: the limitation of performance improvement is 3, why? Can you use different configuration of schedule clause to improve this number?

Number of thread	quartet2	Octet1
1	1.6571 (s)	1.5451 (s)
2	0.9064 (s)	0.9007 (s)
4	0.5433 (s)	0.5165 (s)
8	0.6908 (s)	0.4830 (s)
16	0.7694 (s)	0.5957 (s)
32	0.9263 (s)	0.7098 (s)
64	0.9625 (s)	0.7836 (s)

OutLine

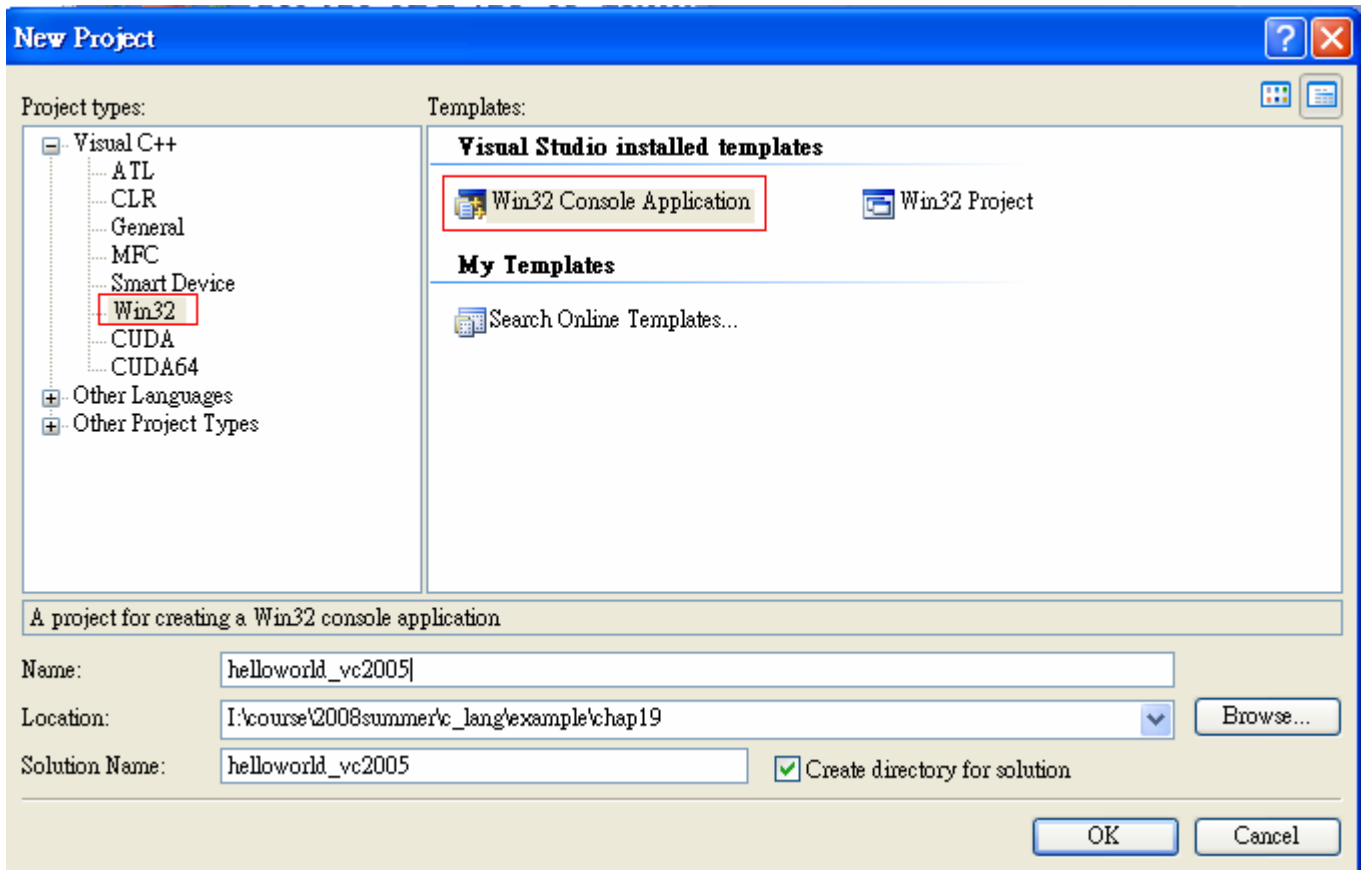
- OpenMP introduction
- Example 1: hello world
- Example 2: vector addition

- enable openmp in vc2005
  - vc2005 supports OpenMP 2.0
  - vc 6.0 does not support OpenMP

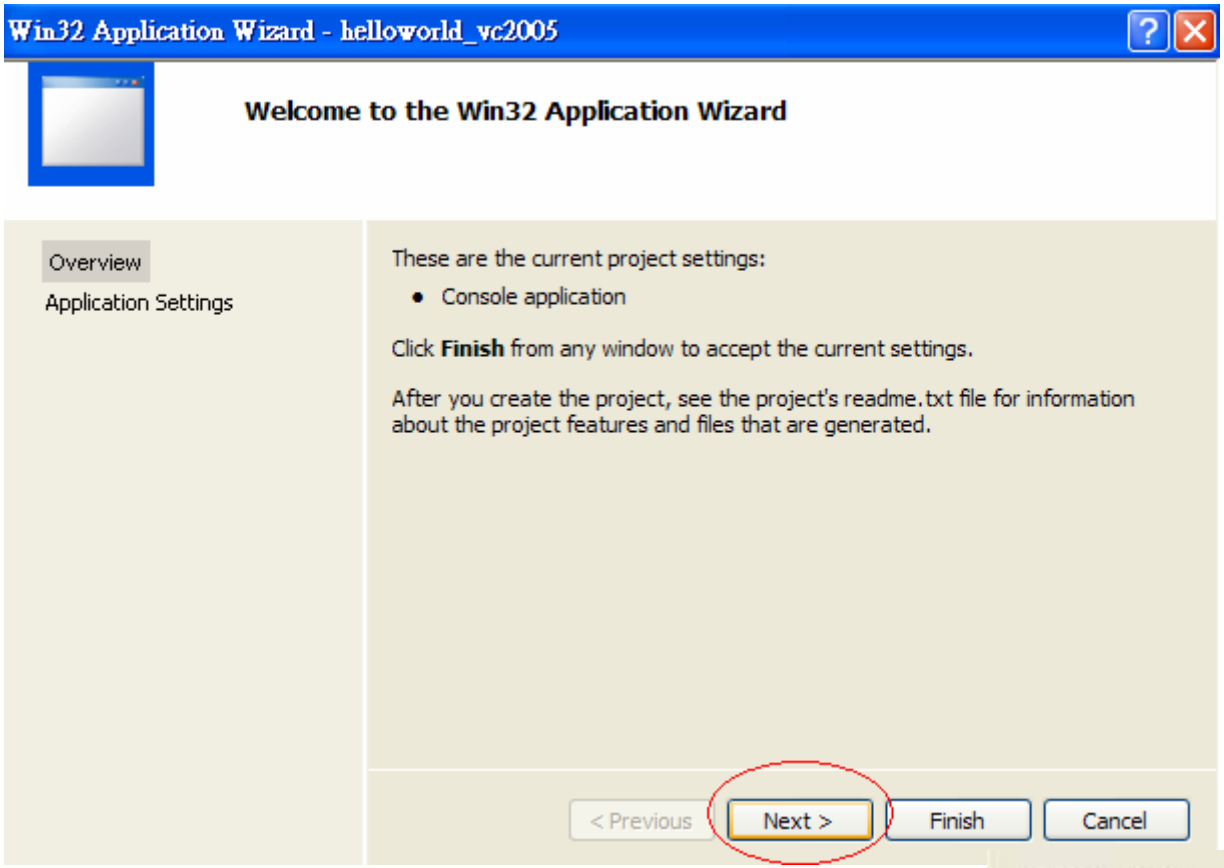
- Example 3: vector addition + Qtime
- Example 4: matrix multiplication
- Example 5: matrix multiplication (block version)

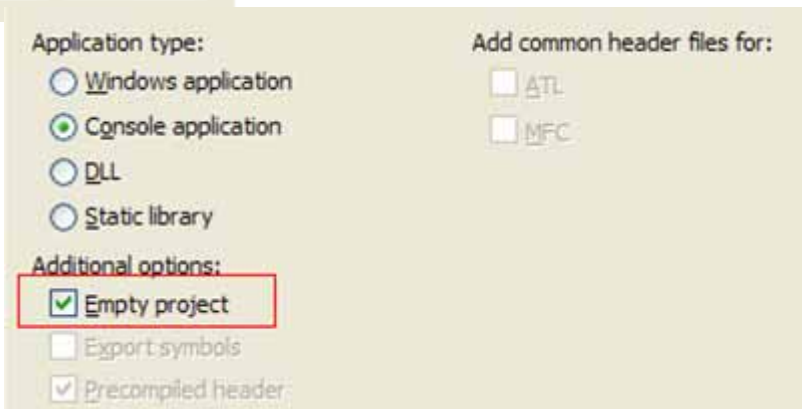
Example 1 (hello world) in vc2005 [1]

Step 1: create a empty consol application

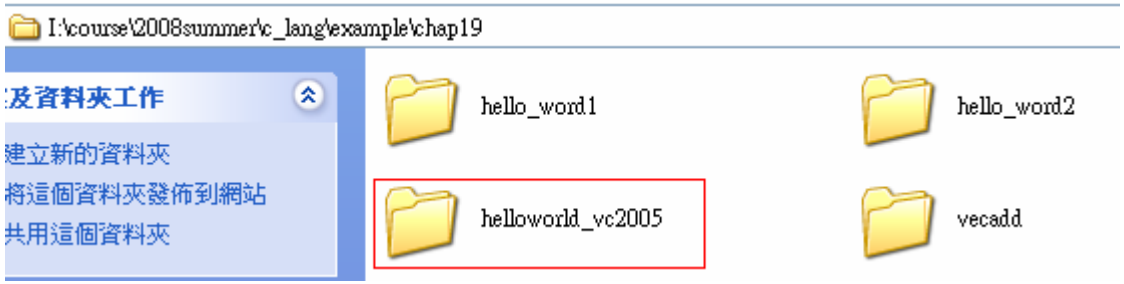


Example 1 (hello world) in vc2005 [2]

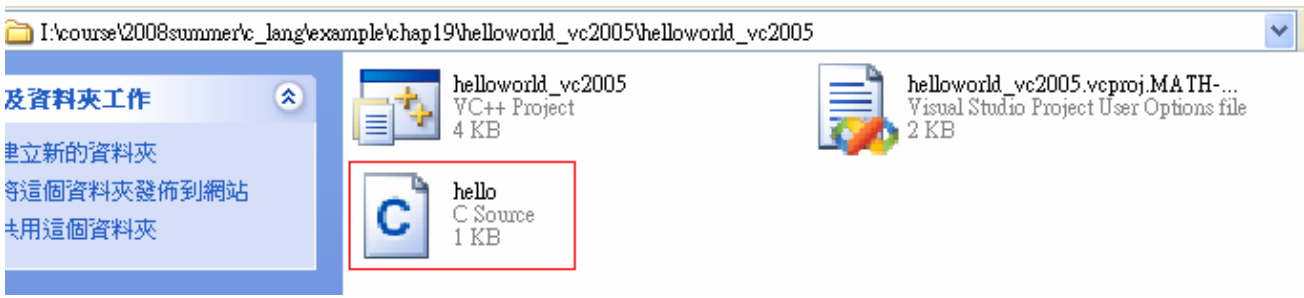


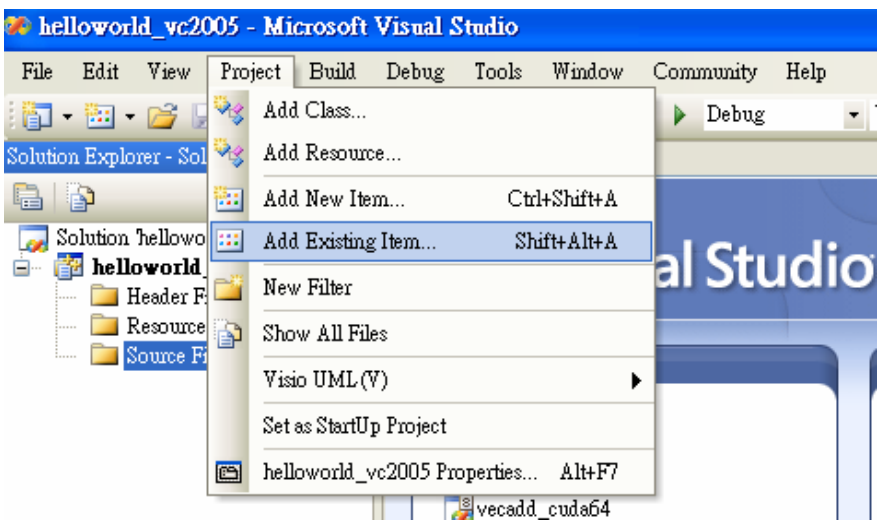


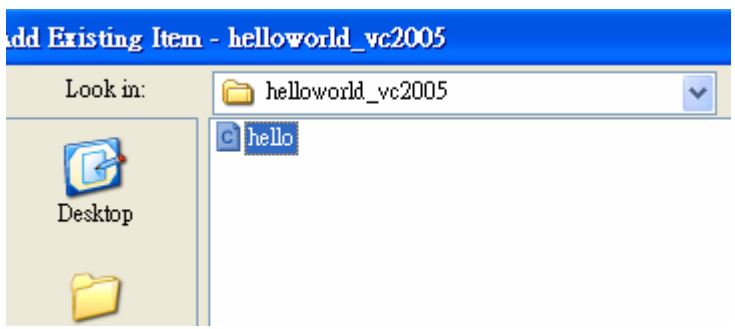
Example 1 (hello world) in vc2005 [3]



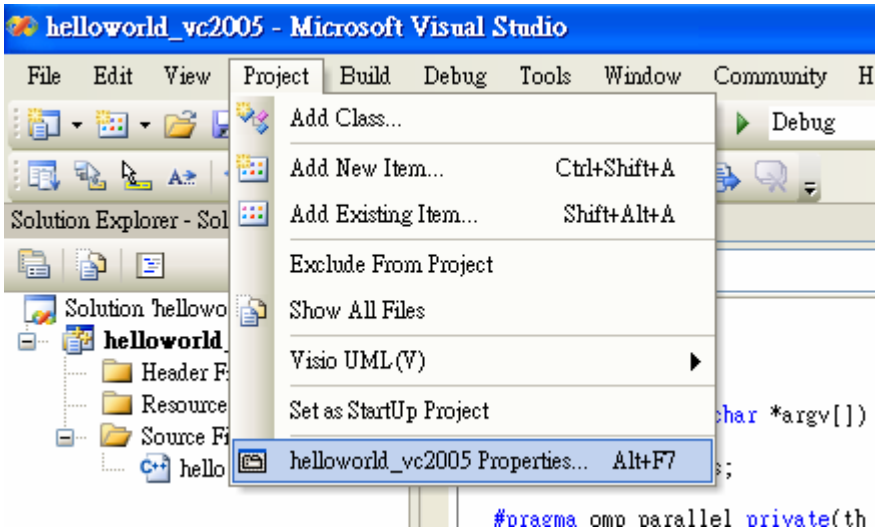
Step 2: copy hello.c to this project and add hello.c to project manager



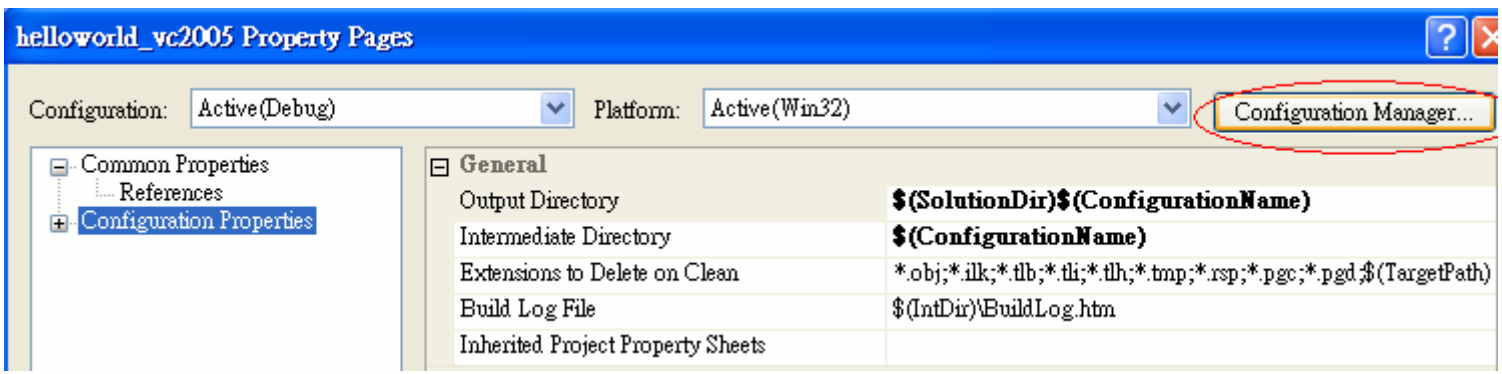




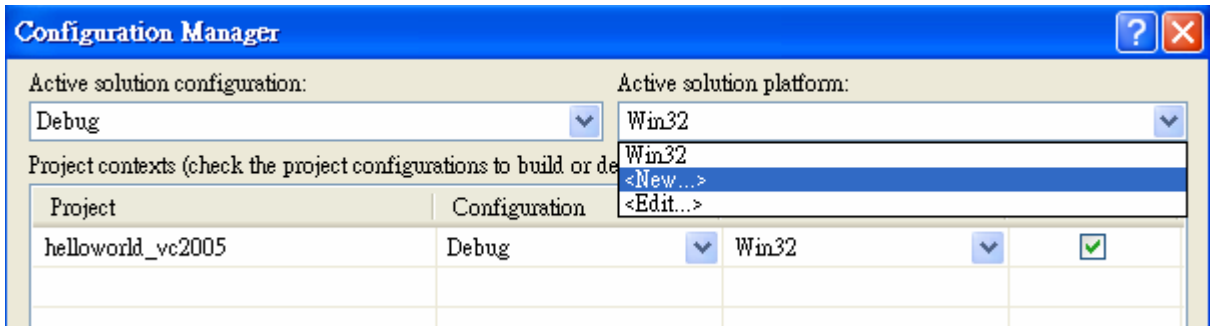
Example 1 (hello world) in vc2005 [4]

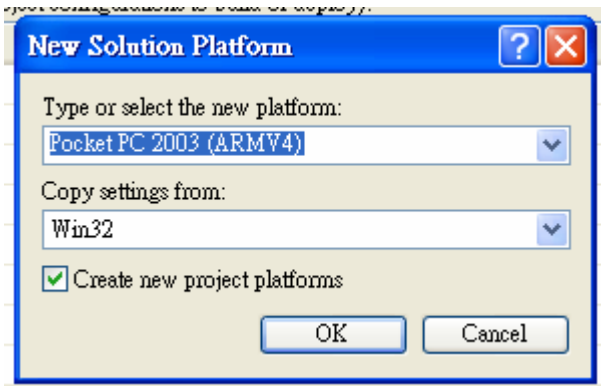


Step 3: change platform to x64

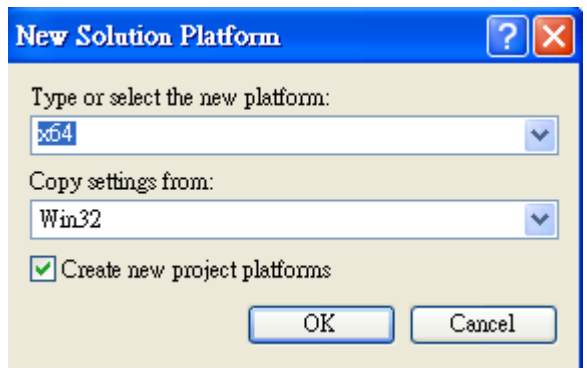


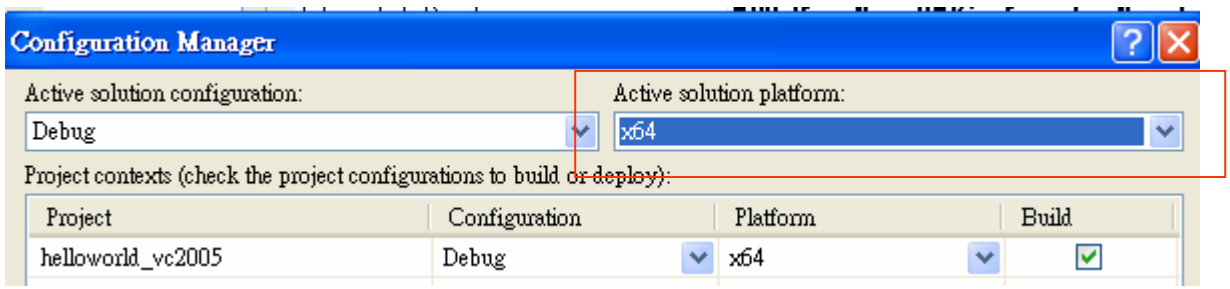
Example 1 (hello world) in vc2005 [5]





choose option “x64”

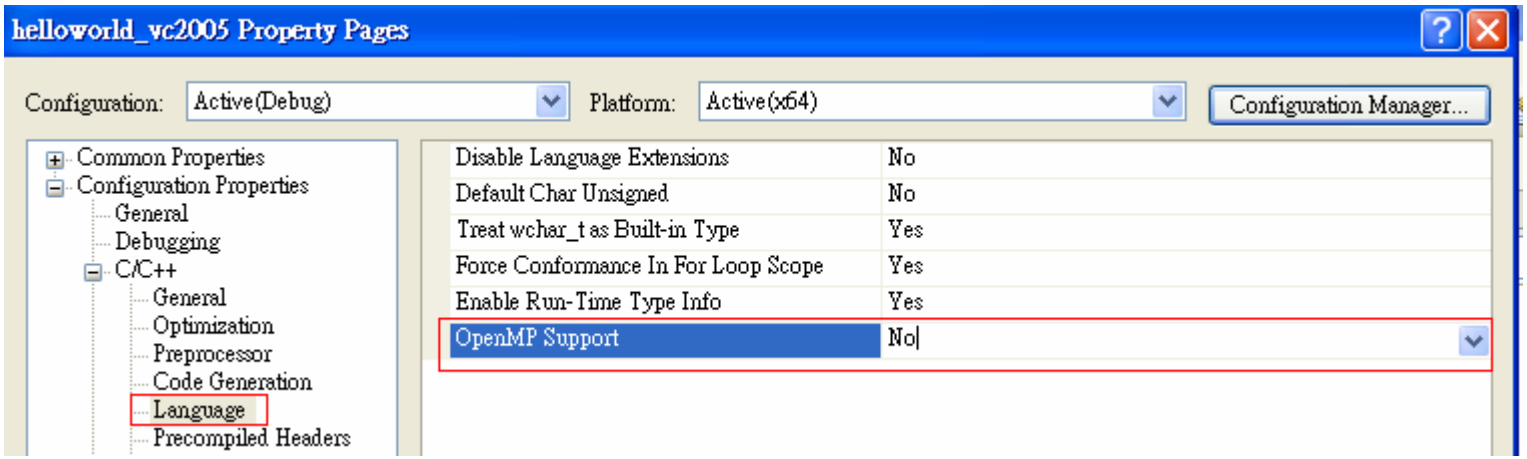




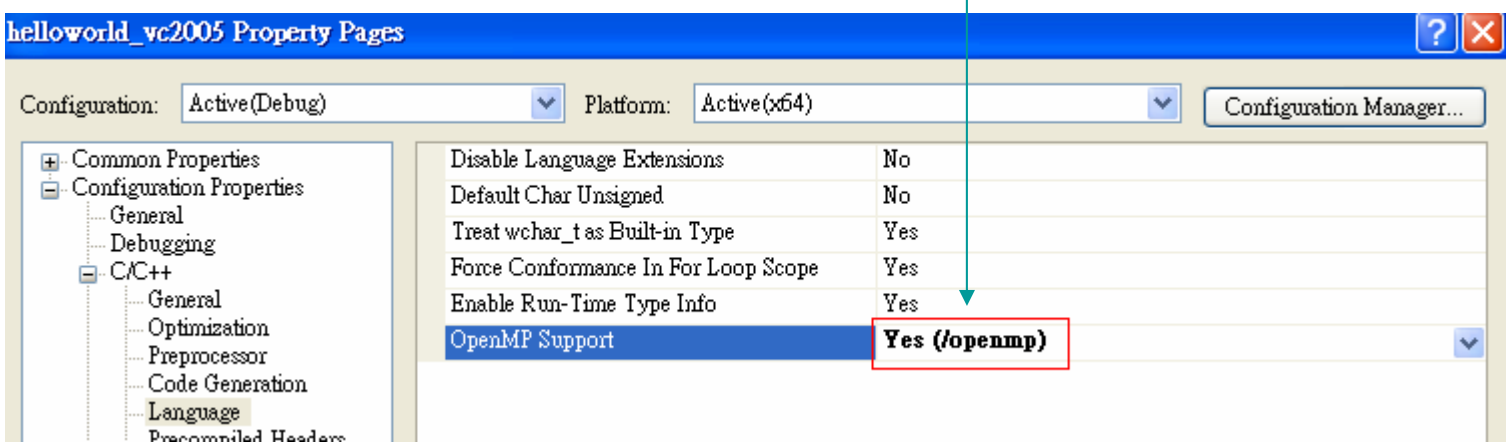
update platform as “x64”

Example 1 (hello world) in vc2005 [6]

Step 4: enable “openmp” support

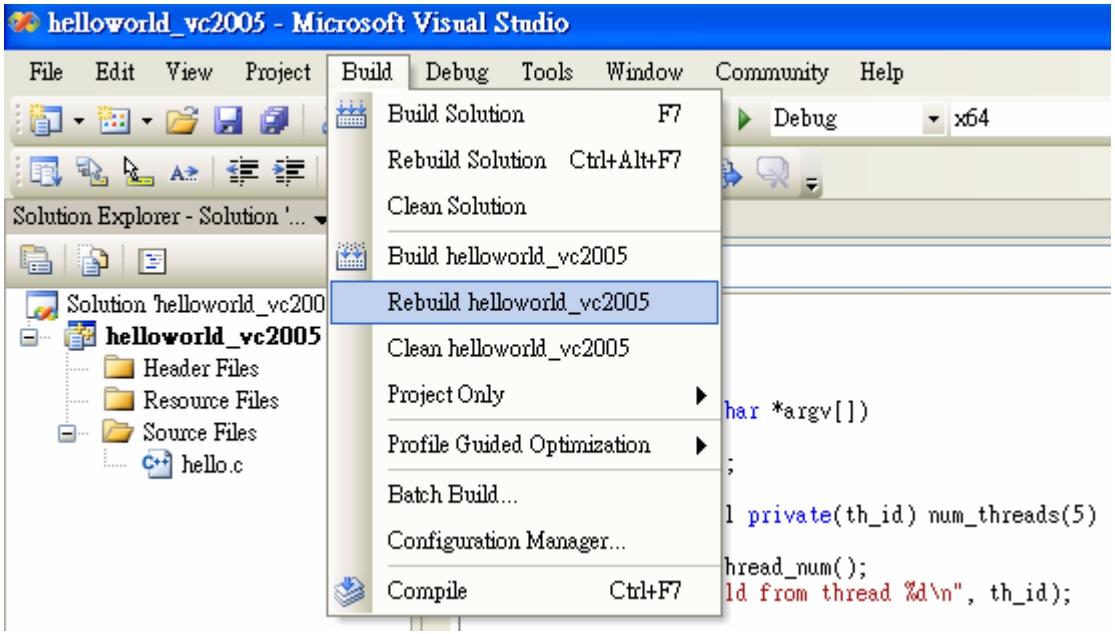


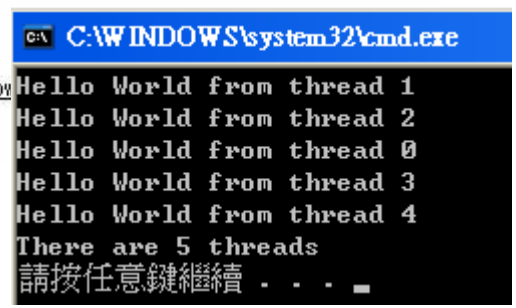
vc 2005 support OpenMP 2.0



Example 1 (hello world) in vc2005 [7]

Step 5: compile and execute

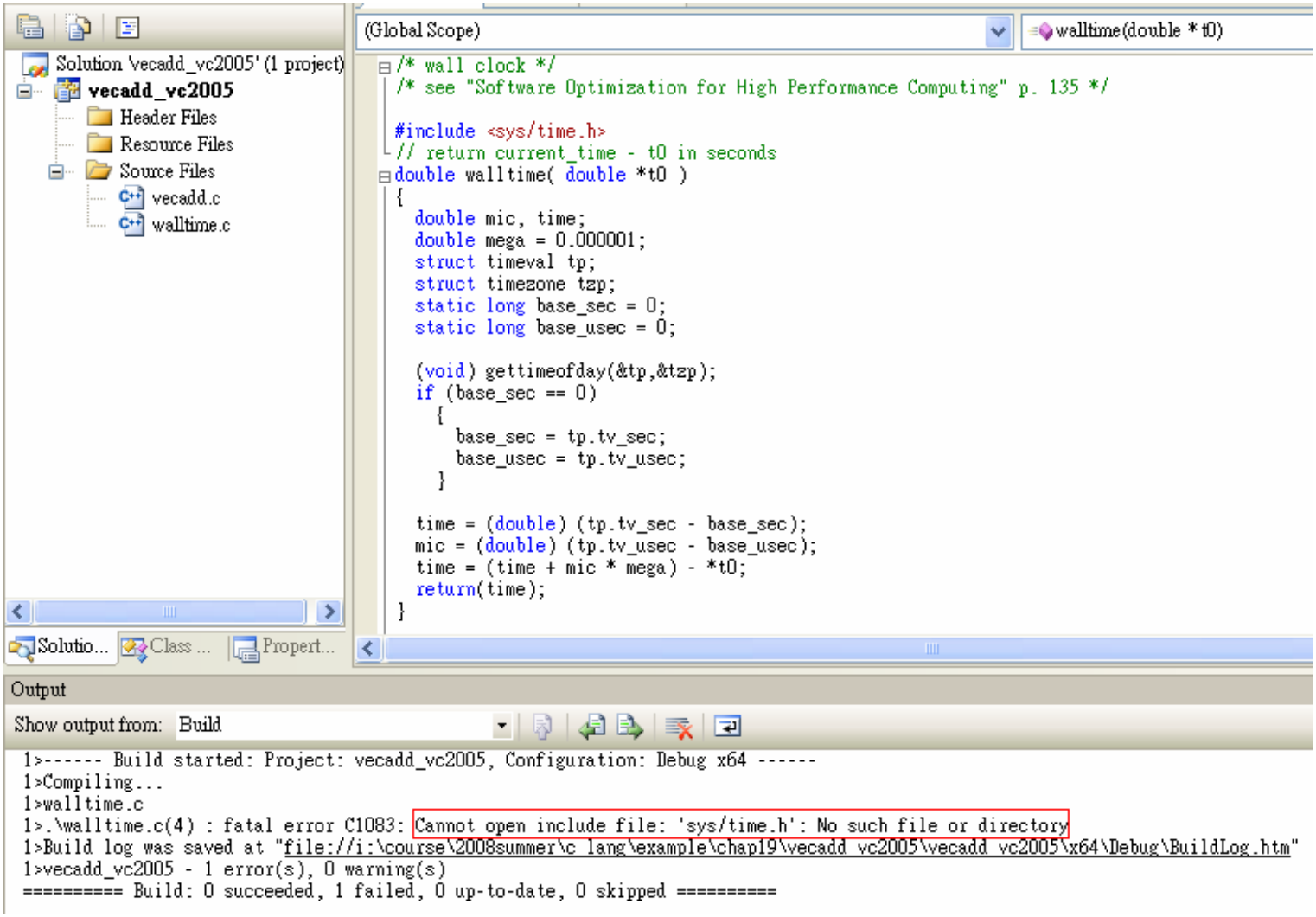




Example 2 (vector addition) in vc2005 [1]

walltime.c only works in Linux machine since no “sys/time.h” in windows

In time.h of ANCI C, no function “gettimeofday”, hence we give up walltime.c



Example 2 (vector addition) in vc2005 [2]

time_t **time**(*time_t* *tp)

returns the current calendar time or -1 if the time is not available. If tp is not NULL, the return value is also assigned to *tp.

double **difftime**(*time_t* time_2, *time_t* time_1)

returns time_2 – time_1 expressed in seconds

vecadd.cpp

```
16  //  double startTime, elapsedTime; /* for timing */
17    time_t startTime, endTime ;
18    double elapsedTime;

24  //  startTime = walltime( &clockZero );
25    time( &startTime ) ;
26    randomInit(a, N);
27    randomInit(b, N);
28  //  elapsedTime = walltime( &startTime );
29    time( &endTime ) ;
30    elapsedTime = difftime(endTime, startTime) ;
31    printf("Time to randomize a, b = %6.4f (s)\n", elapsedTime);

35  //  startTime = walltime( &clockZero );
36    time( &startTime ) ;
37  #pragma omp parallel default(none) num_threads(thread_num)  \
38    shared(a,b,c,N) private(i)
39    {
40      #pragma omp for schedule( static ) nowait
41      for (i=0; i < N; i++){
42        c[i] = a[i] + b[i];
43      }
44
45    } /* end of parallel section */
46  //  elapsedTime = walltime( &startTime );
47    time( &endTime ) ;
48    elapsedTime = difftime(endTime, startTime) ;
```

OutLine

- OpenMP introduction
- Example 1: hello world
- Example 2: vector addition
- enable openmp in vc2005

- Example 3: vector addition + Qtime

- Example 4: matrix multiplication
- Example 5: matrix multiplication (block version)

Example 3: vector addition (Qtime) [1]

vecadd.cpp

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <assert.h>
5
6  #include <qdatetime.h>
7
8  void randomInit( float* data, int size) ;
9
10 int main(int argc, char *argv[] )
11 {
12   long int N = 200*1024*1024 ;
13   int  thread_num = 4 ;
14   long int i;
15   float *a, *b, *c ;
16
17   QTime t; // QT timer
18
19 #ifdef _OPENMP
20   printf("OpenMP-compliant implementation\n");
21 #endif
22
23   a = (float*) malloc( sizeof(float)*N ) ; assert(a) ;
24   b = (float*) malloc( sizeof(float)*N ) ; assert(b) ;
25   c = (float*) malloc( sizeof(float)*N ) ; assert(c) ;
```

constructs the time 0 hours, minutes, seconds and milliseconds, i.e. 00:00:00.000 (midnight).

This is a valid time.

- A QTime object contains a clock time, i.e. the number of hours, minutes, seconds, and milliseconds since midnight
- QTime uses the 24-hour clock format; it has no concept of AM/PM. It operates in local time; it knows nothing about time zones or daylight savings time.
- QTime can be used to measure a span of elapsed time using the start(), restart(), and elapsed() functions

Example 3: vector addition (Qtime) [2]

vecadd.cpp

```
26
27      t.start() ;
28
29    randomInit(a, N);
30    randomInit(b, N);
31    printf("Time to randomize a, b = %d (ms)\n", t.elapsed());
32
33      t.start() ;
34
35  #pragma omp parallel default(none) num_threads(thread_num)   \
36       shared(a,b,c,N) private(i)
37    {
38      #pragma omp for schedule( static ) nowait
39      for (i=0; i < N; i++){
40        c[i] = a[i] + b[i];
41      }
42    } /* end of parallel section */
43
44    printf("thread_num = %d, time for vecadd = %d (ms)\n",
45        thread_num,  t.elapsed()) ;
46
```

void QTime::start ()

Sets this time to the current time. This is practical for timing:

int QTime::elapsed () const

Returns the number of milliseconds that have elapsed since the last time start() or restart() was called.

Note that the counter wraps to zero 24 hours after the last call to start() or restart.

Note that the accuracy depends on the accuracy of the underlying operating system; not all systems provide 1-millisecond accuracy.

Example 3: vector addition (Qtime) [3]

```
[macrold@octet1 vecadd_qt]$
[macrold@octet1 vecadd_qt]$ ls
Makefile  vecadd.cpp
[macrold@octet1 vecadd_qt]$ qmake -project
[macrold@octet1 vecadd_qt]$ ls
Makefile  vecadd.cpp  vecadd_qt.pro
[macrold@octet1 vecadd_qt]$ qmake -spec linux-icc-openmp vecadd_qt.pro
[macrold@octet1 vecadd_qt]$ ls
Makefile  vecadd.cpp  vecadd_qt.pro
[macrold@octet1 vecadd_qt]$ make
icpc -c -w -O2 -openmp -mp  -DQT_NO_DEBUG -DQT_SHARED -DQT_THREAD_SUPPORT -I/opt/qt/mkspecs/li
nux-icc-openmp -I. -I. -I/opt/qt/include -o vecadd.o vecadd.cpp
vecadd.cpp(36): (col. 5) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
vecadd.cpp(33): (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
vecadd.cpp(37): (col. 5) remark: LOOP WAS VECTORIZED.
icpc -openmp  -Qoption,ld,-rpath,/opt/qt/lib -o vecadd_qt vecadd.o   -L/opt/qt/lib -L/usr/X11R
6/lib -lqt-mt -lXext -lX11 -lm
[macrold@octet1 vecadd_qt]$ ls
Makefile  vecadd.cpp  vecadd.o  vecadd_qt  vecadd_qt.pro
[macrold@octet1 vecadd_qt]$ ./vecadd_qt
OpenMP-compliant implementation
Time to randomize a, b = 5515 (ms)
thread_num = 4, time for vecadd = 522 (ms)
size = 800.00 (MB)
[macrold@octet1 vecadd_qt]$
```

generate project file ***vecadd_qt.pro***

generate Makefile

Makefile

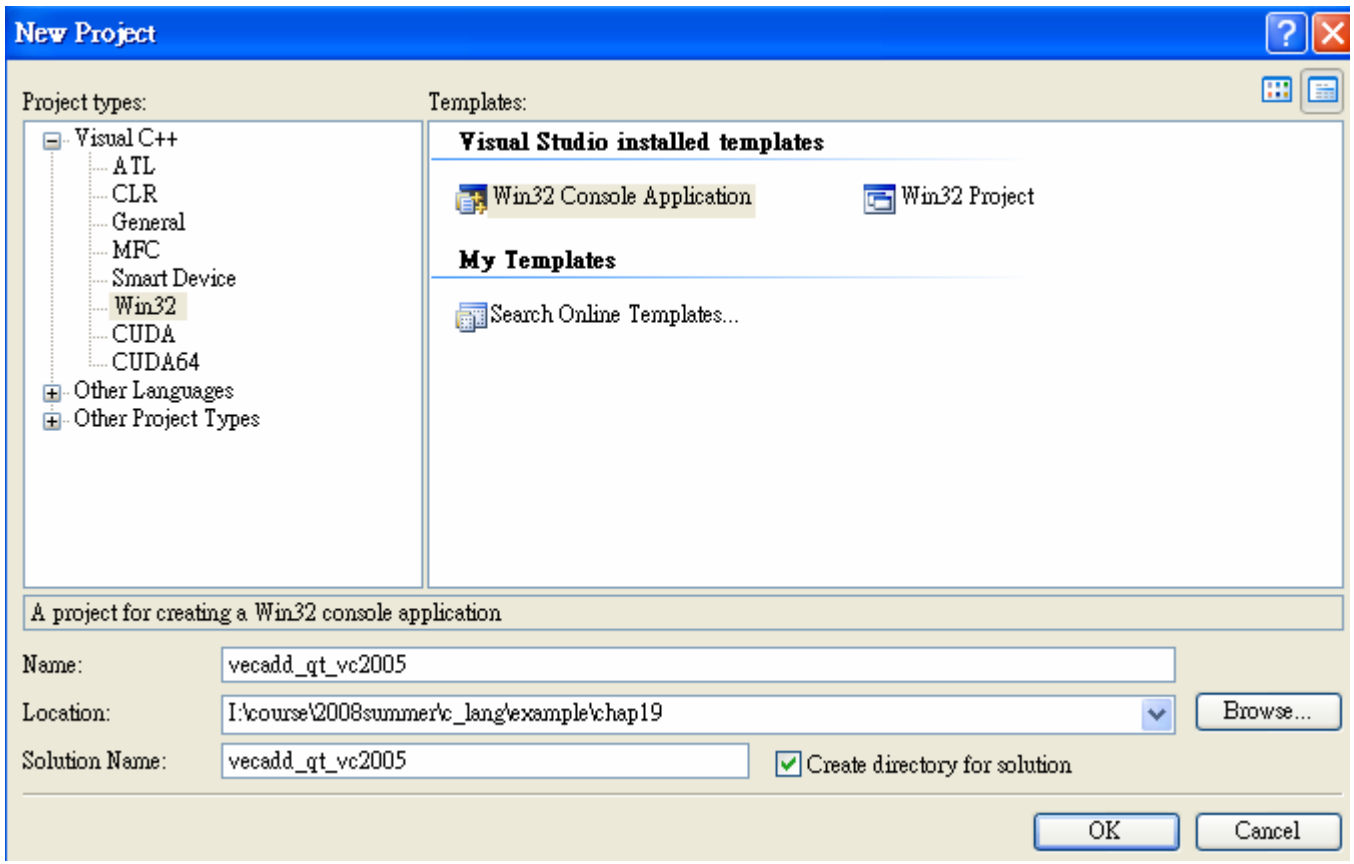
```
####### Compiler, tools and options

CC       = icc
CXX      = icpc
LEX      = flex
YACC     = yacc
CFLAGS   = -w -O2 -openmp -mp  -DQT_NO_DEBUG -DQT_SHARED -DQT_THREAD_SUPPORT
CXXFLAGS = -w -O2 -openmp -mp  -DQT_NO_DEBUG -DQT_SHARED -DQT_THREAD_SUPPORT
LEXFLAGS =
YACCFLAGS= -d
INCPATH  = -I/opt/qt/mkspecs/linux-icc-openmp -I. -I. -I$(QTDIR)/include
LINK     = icpc
LFLAGS   = -openmp -Qoption,ld,-rpath,$(QTDIR)/lib
LIBS     = $(SUBLIBS) -L$(QTDIR)/lib -L/usr/X11R6/lib -lqt-mt -lXext -lX11 -lm
```

Example 3: vector addition (Qtime) [4]

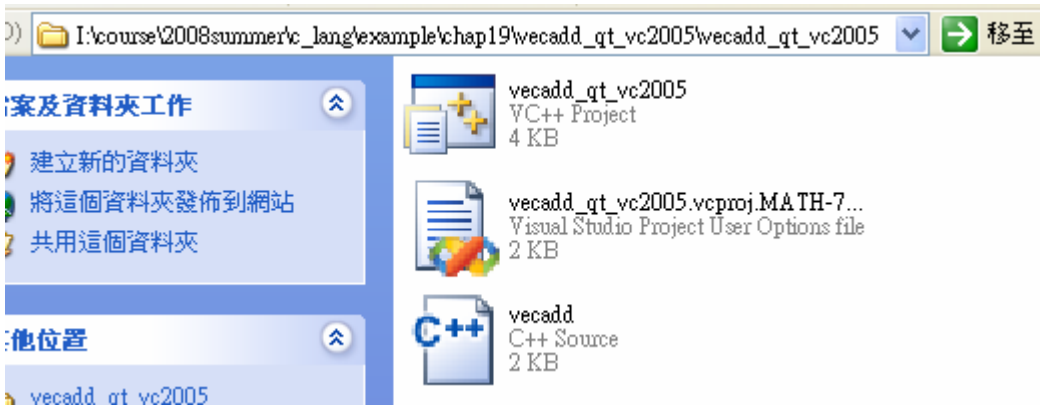
Embed Qt 3.2.1 non-comercial version into vc 2005

Step 1: setup an empty project

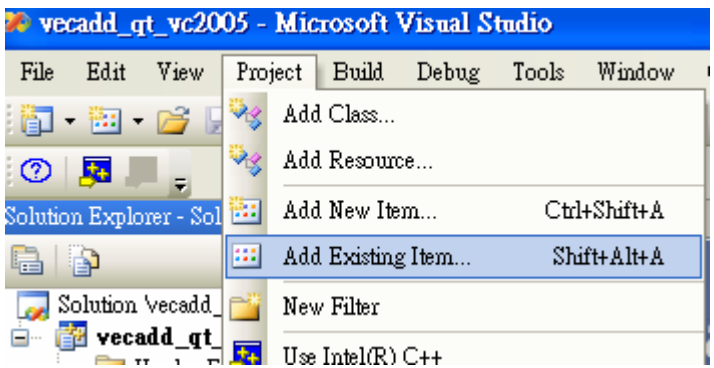


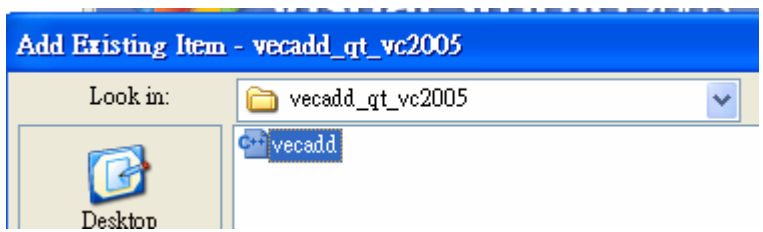
Example 3: vector addition (Qtime) [5]

Step 2: copy vecadd.cpp into this project



Step 3: add item “vecadd.cpp” in project manager

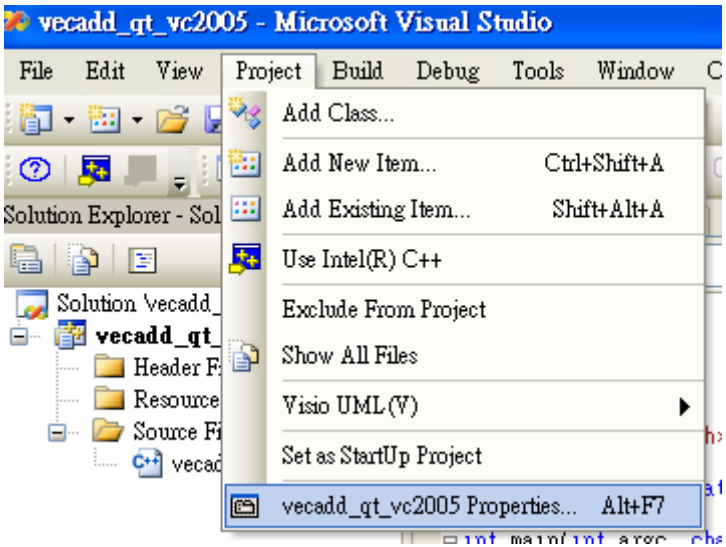




Example 3: vector addition (Qtime) [6]

Step 4: project → properties → C/C++ → General → Additional include Directories

.;$(QTDIR)\include;C:\Qt\3.2.1NonCommercial\mkspecs\win32-msvc

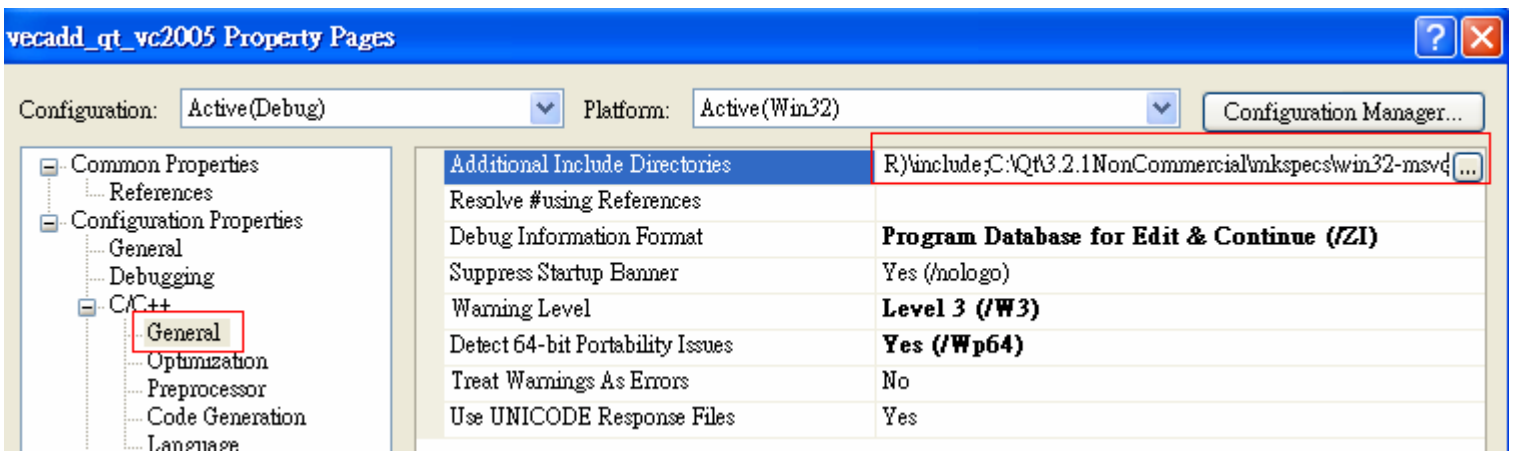



```
C:\Documents and Settings\LungShengChien>set
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\LungShengChien\Application Data
CC=cl
CLIENTNAME=Console
```

⋮

```
QMAKESPEC=win32-msvc
QTDIR=C:\Qt\3.2.1NonCommercial
```

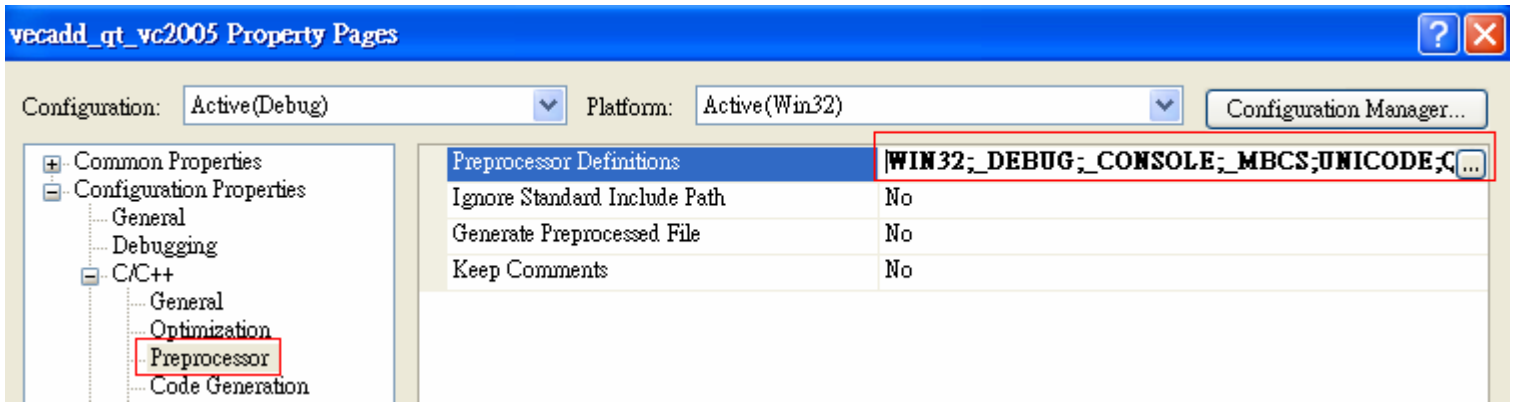
⋮



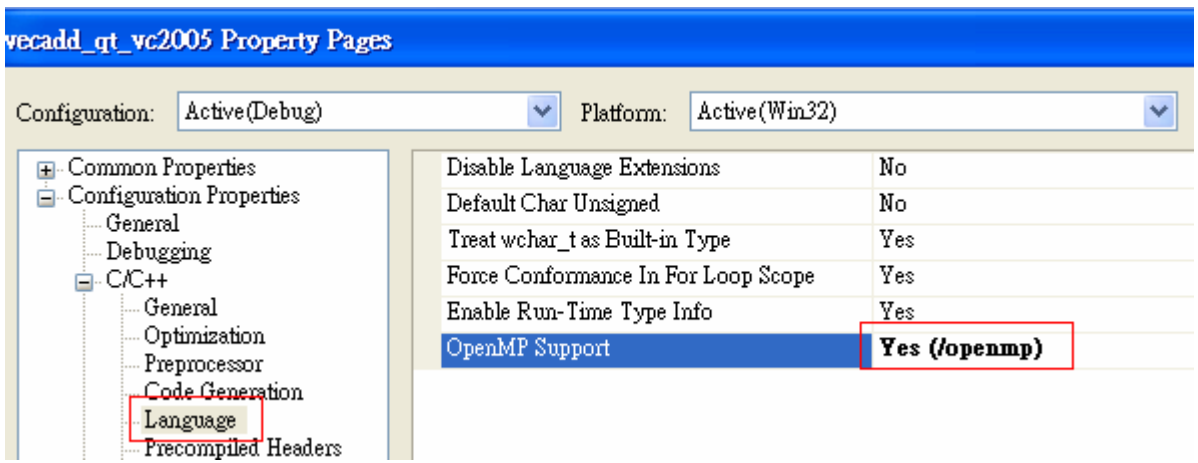
Example 3: vector addition (Qtime) [7]

Step 5: project → properties → C/C++ → Preprocessor → Preprocessor Definitions

WIN32;_DEBUG;_CONSOLE;_MBCS;UNICODE;QT_DLL;QT_THREAD_SUPPORT



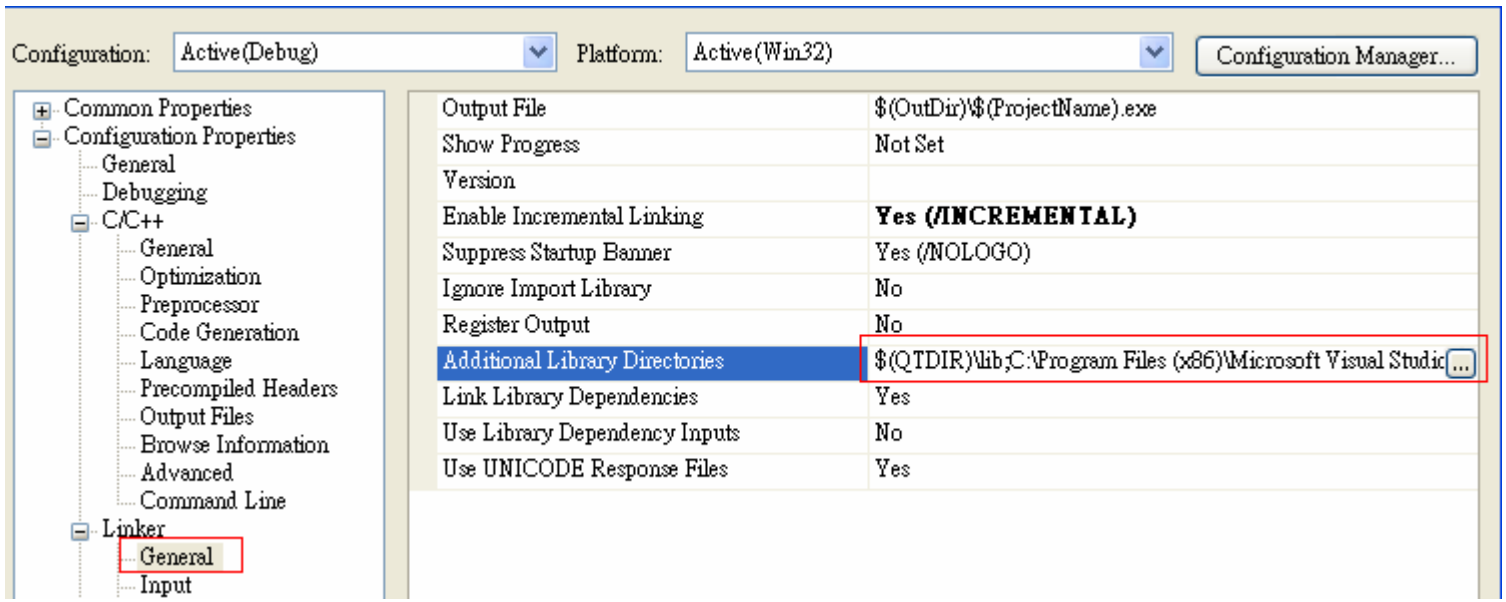
Step 6: project → properties → C/C++ → Language → OpenMP Support



Example 3: vector addition (Qtime) [8]

Step 7: project → properties → Linker → General → Additional Library Directories

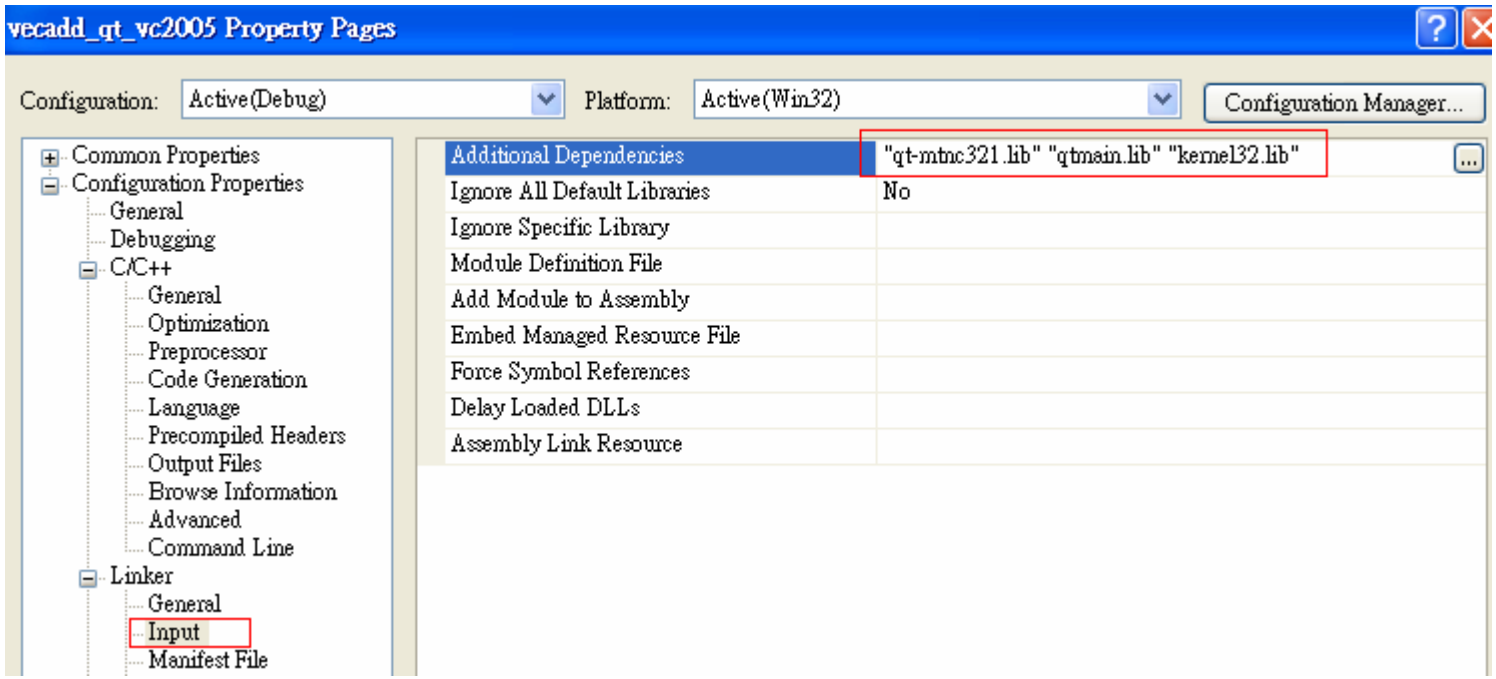
$(QTDIR)\lib;C:\Program Files (x86)\Microsoft Visual Studio 8\VC\lib



Example 3: vector addition (Qtime) [9]

Step 8: project → properties → Linker → Input → Additional Dependence

"qt-mtnc321.lib" "qtmain.lib" "kernel32.lib"



Step 9: compile and execute

Restriction: QT3 in windows only support 32-bit application, we must choose platform as “Win32”, we will solve this problem after installing QT4

OutLine

- OpenMP introduction
- Example 1: hello world
- Example 2: vector addition
- enable openmp in vc2005
- Example 3: vector addition + Qtime

- Example 4: matrix multiplication

- Example 5: matrix multiplication (block version)

Example 4: matrix multiplication [1]

matrixMul.h

```
#ifndef _MATRIXMUL_H_
#define _MATRIXMUL_H_

// Thread block size
#define BLOCK_SIZE 16

// Matrix dimensions
#define WA (25 * BLOCK_SIZE) // Matrix A width
#define HA (25 * BLOCK_SIZE) // Matrix A height
#define WB (25 * BLOCK_SIZE) // Matrix B width
#define HB WA  // Matrix B height
#define WC WB  // Matrix C width
#define HC HA  // Matrix C height

#endif // _MATRIXMUL_H_
```

matrixMul.cpp

```
#include "matrixMul.h"

void matrixMul_seq(float* C, const float* A, const float* B,
  unsigned int hA, unsigned int wA, unsigned int wB ) ;

void matrixMul_parallel(float* C, const float* A, const float* B,
  unsigned int hA, unsigned int wA, unsigned int wB, int nthreads ) ;

// c = a * B
void matrixMul_seq(float* C, const float* A, const float* B,
  unsigned int hA, unsigned int wA, unsigned int wB )
{
    double sum ;
    unsigned int i, j, k ;
    double a, b ;
    for ( i = 0; i < hA; ++i){
        for (j = 0; j < wB; ++j) {
            sum = 0;
            for (k = 0; k < wA; ++k) {
                a = A[i * wA + k];
                b = B[k * wB + j];
                sum += a * b;
            }// for k
            C[i * wB + j] = (float)sum;
        }// for j
    }// for i
}
```

sequential version

$$c_{ij} = \sum_{k=1}^{wA} a_{ik} b_{kj}$$

row-major index

$$a_{ik} = A\left[i \cdot wA + k\right]$$

$$b_{kj} = A\left[k \cdot wB + j\right]$$

$$c_{ij} = A\left[i \cdot wC + j\right]$$

Example 4: matrix multiplication [2]

matrixMul.cpp

```
30  void matrixMul_parallel(float* C, const float* A, const float* B,
31    unsigned int hA, unsigned int wA, unsigned int wB, int nthreads )
32  {
33      double sum ;
34      int i, j, k ;
35      double a, b ;
36  #pragma omp parallel default(none) num_threads(nthreads)  \
37       shared(A,B,C, hA, wB, wA) private(i,j,k,sum,a,b)
38      {
39      #pragma omp for schedule( static)  nowait
40          for (i = 0; i < hA; ++i){
41             for (j = 0; j < wB; ++j) {
42                sum = 0;
43                for (k = 0; k < wA; ++k) {
44                    a = A[i * wA + k];
45                    b = B[k * wB + j];
46                    sum += a * b;
47                }// for k
48                C[i * wB + j] = (float)sum;
49              }// for j
50          }// for i
51       }// end of parallel section
52  }
53
```

parallel version

Question 8: we have three for-loop, one is for "***i***", one is for "***j***" and last one is for "***k***", which one is parallelized by OpenMP directive?

Question 9: explain why variable ***i, j, k, sum, a, b*** are declared as *private*? Can we move some of them to *shared* clause?

Example 4: matrix multiplication [3]

main.cpp

```
 2  #include <stdlib.h>
 3  #include <stdio.h>
 4  #include <string.h>
 5  #include <math.h>
 6  #include <assert.h>
 7  #include <iostream>
 8
 9  #include <qdatetime.h>
10
11  #include "matrixMul.h"
12
13  void runTest(int argc, char** argv) ;
14  void randomInit(float*, int);
15
16  void matrixMul_seq(float* C, const float* A, const float* B,
17    unsigned int hA, unsigned int wA, unsigned int wB ) ;
18
19  void matrixMul_parallel(float* C, const float* A, const float* B,
20    unsigned int hA, unsigned int wA, unsigned int wB, int nthreads ) ;
21
22  int main(int argc, char** argv)
23  {
24      runTest(argc, argv);
25      return 0 ;
26  }

28  void runTest(int argc, char** argv)
29  {
30      int  nthreads = 2 ;
31      unsigned int total_size = 0 ;
32      QTime t; // QT timer
33
34      // set seed for rand()
35      srand(2006);
```

use QT timer

Example 4: matrix multiplication [4]

main.cpp

```
37      // allocate host memory for matrices A and B
38      unsigned int size_A = WA * HA;
39      unsigned int mem_size_A = sizeof(float) * size_A;
40      float* h_A = (float*) malloc(mem_size_A);
41      assert( h_A ) ;
42
43      total_size += mem_size_A ;
44
45      unsigned int size_B = WB * HB;
46      unsigned int mem_size_B = sizeof(float) * size_B;
47      float* h_B = (float*) malloc(mem_size_B);
48      assert( h_B ) ;
49      total_size += mem_size_B ;
50
51      // initialize host memory
52      randomInit(h_A, size_A);
53      randomInit(h_B, size_B);
54
55      unsigned int size_C = WC * HC;
56      unsigned int mem_size_C = sizeof(float) * size_C;
57      float* h_C = (float*) malloc(mem_size_C);
58      assert( h_C ) ;
59      total_size += mem_size_C ;
61      t.start() ;
62      if ( 1 == nthreads ){
63         matrixMul_seq( h_C, h_A, h_B, HA, WA, WB ) ;
64      }else{
65         matrixMul_parallel( h_C, h_A, h_B,
66               HA, WA, WB, nthreads ) ;
67      }
68      printf("threads = %d, matrixMul cost = %d (ms)\n",
69         nthreads,  t.elapsed()) ;
70      printf( "size(A) = (%d,%d)\n", HA, WA );
71      printf( "size(B) = (%d,%d)\n", HB, WB );
72      printf("total memory size = %6.4f (MB)\n",
73              total_size/1048576.0 );
74      // clean up memory
75      free(h_A); free(h_B); free(h_C);
76  }
```

use qmake to generate Makefile

```
[macrold@quartet2 matrixMul]$ ls
main.cpp  matrixMul.cpp  matrixMul.h
[macrold@quartet2 matrixMul]$ qmake -project
[macrold@quartet2 matrixMul]$ ls
main.cpp  matrixMul.cpp  matrixMul.h  matrixMul.pro
[macrold@quartet2 matrixMul]$ qmake -spec linux-icc-openmp matrixMul.pro
[macrold@quartet2 matrixMul]$ ls
Makefile  main.cpp  matrixMul.cpp  matrixMul.h  matrixMul.pro
[macrold@quartet2 matrixMul]$ make
icpc -c -w -O2 -openmp -mp  -DQT_NO_DEBUG -DQT_SHARED -DQT_THREAD_SUPPORT
/qt/mkspecs/linux-icc-openmp -I. -I. -I/opt/qt/include -o main.o main.cpp
icpc -c -w -O2 -openmp -mp  -DQT_NO_DEBUG -DQT_SHARED -DQT_THREAD_SUPPORT
/qt/mkspecs/linux-icc-openmp -I. -I. -I/opt/qt/include -o matrixMul.o matr
cpp
matrixMul.cpp(39): (col. 5) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
matrixMul.cpp(36): (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED
icpc -openmp  -Qoption,ld,-rpath,/opt/qt/lib -o matrixMul main.o matrixMul
L/opt/qt/lib -L/usr/X11R6/lib -lqt-mt -lXext -lX11 -lm
[macrold@quartet2 matrixMul]$ ls
Makefile  main.o     matrixMul.cpp  matrixMul.o
main.cpp  matrixMul  matrixMul.h    matrixMul.pro
[macrold@quartet2 matrixMul]$
```

```
[macrold@quartet2 matrixMul]$ ./matrixMul
threads = 2, matrixMul cost = 125 (ms)
size(A) = (400,400)
size(B) = (400,400)
total memory size = 1.8311 (MB)
[macrold@quartet2 matrixMul]$
```

Example 4: matrix multiplication [5]

Let BLOCK_SIZE = 16 and $size(A) = size(B) = size(C) = (N \cdot BLOCK\_SIZE)^2$

total memory usage $= size(A) + size(B) + size(C)$ float

Platform: oectet1, with compiler icpc 10.0, -O2

N	Total size	Thread 1	Thread 2	Thread 4	Thread 8
16	0.75 MB	53 ms	31 ms	21 ms	24ms
32	3 MB	434 ms	237 ms	121 ms	90 ms
64	12 MB	17,448 ms	8,964 ms	6,057 ms	2,997 ms
128	48 MB	421,854 ms	312,983 ms	184,695 ms	92,862 ms
256	192 MB	4,203,536 ms	2,040,448 ms	1,158,156 ms	784,623 ms

Large performance gap amogn *N* = 32, *N* = 64 and *N* = 128, so this algorithm is NOT good. Besides improvement of multi-thread is not significant.

Example 4: matrix multiplication [6]

```
[macrold@octetl matrixMul]$
[macrold@octetl matrixMul]$ ./matrixMul
```
running

Use command “top” to see resource usage

```
[macrold@octetl ~]$
[macrold@octetl ~]$ top
```

```
top - 12:57:59 up 84 days, 21:39,  2 users,  load average: 1.38, 0.30, 0.10
Tasks: 198 total,   4 running, 194 sleeping,   0 stopped,   0 zombie
Cpu0  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu5  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  65342468k total, 11876088k used, 53466380k free,   416924k buffers
Swap: 67103496k total,    30464k used, 67073032k free, 10193384k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
12290 macrold   20   0  291m 135m 4988 R  799  0.2   1:16.45 matrixMul
12287 macrold   20   0 14668 1144  812 R    0  0.0   0:00.05 top
13120 gdm       20   0 98.8m 3428 2740 S    0  0.0  34:43.42 at-spi-registry
    1 root      20   0 10328  280  252 S    0  0.0   0:14.56 init
```

CPU usage is 800 %, 8 cores are busy

Exercise 3: verify subroutine ***matrixMul_parallel***

matrixMul.cpp

```
30  void matrixMul_parallel(float* C, const float* A, const float* B,
31    unsigned int hA, unsigned int wA, unsigned int wB, int nthreads )
32  {
33      double sum ;
34      int i, j, k ;
35      double a, b ;
36  #pragma omp parallel default(none) num_threads(nthreads)  \
37       shared(A,B,C, hA, wB, wA) private(i,j,k,sum,a,b)
38      {
39      #pragma omp for schedule( static)  nowait
40          for (i = 0; i < hA; ++i){
41             for (j = 0; j < wB; ++j) {
42                sum = 0;
43                for (k = 0; k < wA; ++k) {
44                    a = A[i * wA + k];
45                    b = B[k * wB + j];
46                    sum += a * b;
47                }// for k
48                C[i * wB + j] = (float)sum;
49             }// for j
50          }// for i
51       }// end of parallel section
52  }
53
```

Combine Parallel Work-sharing constructs

`parallel for` Construct

The **`parallel for`** directive is a shortcut for a **`parallel`** region that contains only a single **`for`** directive. The syntax of the **`parallel for`** directive is as follows:

```
#pragma omp parallel for [clause[[,] clause] ...] new-line
    for-loop
```

This directive allows all the clauses of the **`parallel`** directive and the **`for`** directive, except the **`nowait`** clause, with identical meanings and restrictions. The semantics are identical to explicitly specifying a **`parallel`** directive immediately followed by a **`for`** directive.

matrixMul.cpp

```
30  void matrixMul_parallel(float* C, const float* A, const float* B,
31    unsigned int hA, unsigned int wA, unsigned int wB, int nthreads )
32  {
33      double sum ;
34      int i, j, k ;
35      double a, b ;
36  #pragma omp parallel for default(none) num_threads(nthreads)  \
37       shared(A,B,C, hA, wB, wA) private(i,j,k,sum,a,b)  \
38       schedule( static)
39          for (i = 0; i < hA; ++i){
40             for (j = 0; j < wB; ++j) {
41                sum = 0;
42                for (k = 0; k < wA; ++k) {
43                    a = A[i * wA + k];
44                    b = B[k * wB + j];
45                    sum += a * b;
46                }// for k
47                C[i * wB + j] = (float)sum;
48             }// for j
49          }// for i
50  }
```

Exercise 4: verify following subroutine *matrix_parallel*, which parallelizes loop-***j*** , not loop-***i.***

1. Performance between loop-***i*** and loop-***j***
2. why do we declare index ***i*** as shared variable? What happens if we declare index ***i*** as private variable?

matrixMul.cpp

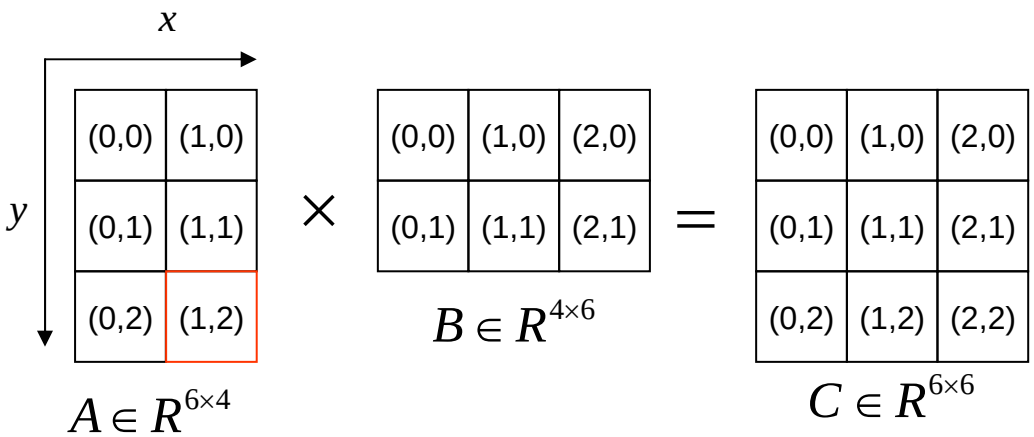
```
30  void matrixMul_parallel(float* C, const float* A, const float* B,
31    unsigned int hA, unsigned int wA, unsigned int wB, int nthreads )
32  {
33      double sum ;
34      int i, j, k ;
35      double a, b ;
36
37          for (i = 0; i < hA; ++i){
38  #pragma omp parallel for default(none) num_threads(nthreads)  \
39       shared(A,B,C, hA, wB, wA, i) private(j,k,sum,a,b)  \
40       schedule( static)
41              for (j = 0; j < wB; ++j) {
42                  sum = 0;
43                  for (k = 0; k < wA; ++k) {
44                      a = A[i * wA + k];
45                      b = B[k * wB + j];
46                      sum += a * b;
47                  }// for k
48                  C[i * wB + j] = (float)sum;
49              }// for j
50          }// for i
51  }
```

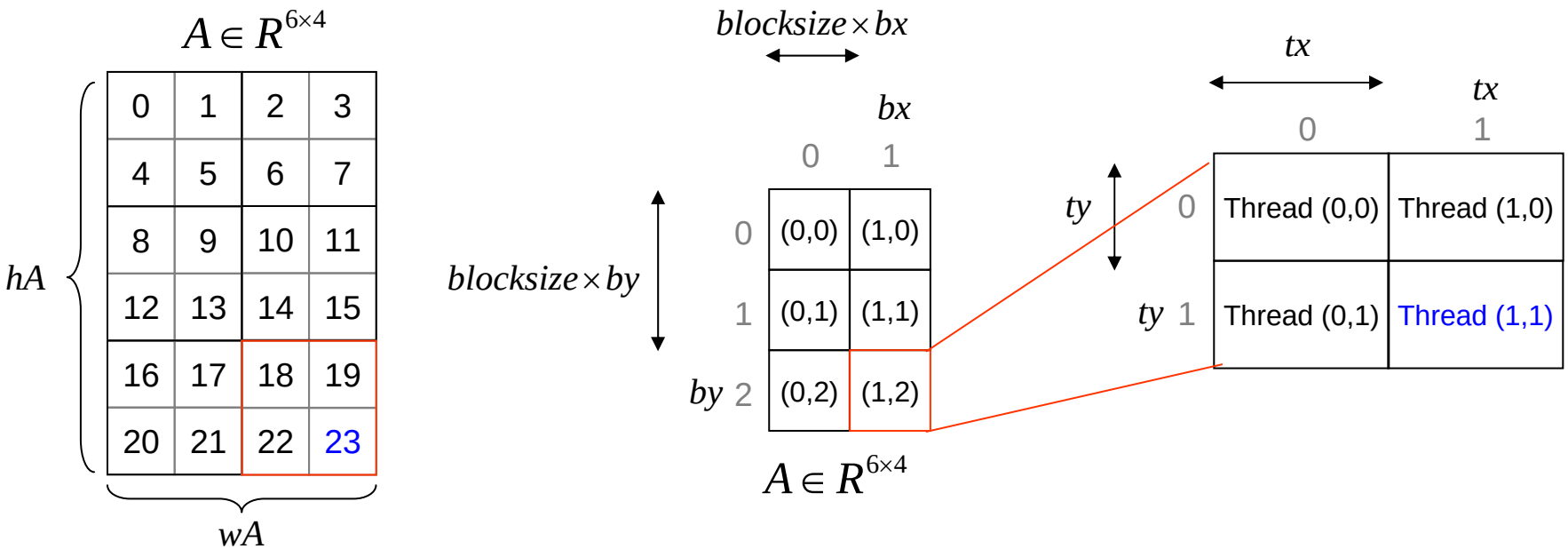
OutLine

- OpenMP introduction
- Example 1: hello world
- Example 2: vector addition
- enable openmp in vc2005
- Example 3: vector addition + Qtime
- Example 4: matrix multiplication

- Example 5: matrix multiplication (block version)

Example 5: matrix multiplication (block version) [1]





global index

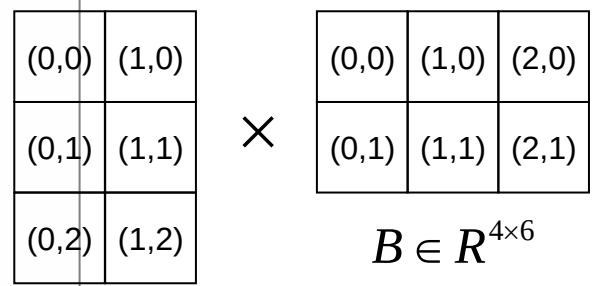
$\big((bx, by), (tx, ty)\big) \longrightarrow \big(blocksize \times bx + tx,\ blocksize \times by + ty\big) \longrightarrow$ row-major

Example 5: matrix multiplication (block version) [2]

matrixMul_block.cpp

```
 7  // store the sub-matrix of A
 8  doublereal As[BLOCK_SIZE][BLOCK_SIZE];
 9  // store the sub-matrix of B
10  doublereal Bs[BLOCK_SIZE][BLOCK_SIZE];
11
12  // Matrix multiplication on the device: C = A * B
13  void matrixMul_block_seq( doublereal* C, doublereal* A, doublereal* B,
14       const int hA, const int wA, const int wB,
15       const int hA_grid, const int wA_grid, const int wB_grid )
16  {
17      int bx, by ; // Block index
18      int tx, ty ;
19      int aBegin, aEnd, aStep, bBegin, bStep ;
20      int a, b, c, k ;
21  #ifdef  HIGH_PRECISION_PACKAGE
22      doublereal Asub, Bsub, Csub ;
23  #else
24      double Asub, Bsub, Csub ;
25  #endif
26
27      unsigned long int i ;
28      unsigned long int size = hA*wB ; // size of matrix C
29  #ifdef  HIGH_PRECISION_PACKAGE
30      doublereal* C_ptr = C ;
31      for (i = 0 ; i < size ; i++ ){
32        *C_ptr++ = 0.0 ;
33      }
34  #else
35      memset( C, 0, size*sizeof(doublereal) ) ;
36  #endif
```

Shared memory in GPU



$A \in R^{6\times 4}$ $B \in R^{4\times 6}$

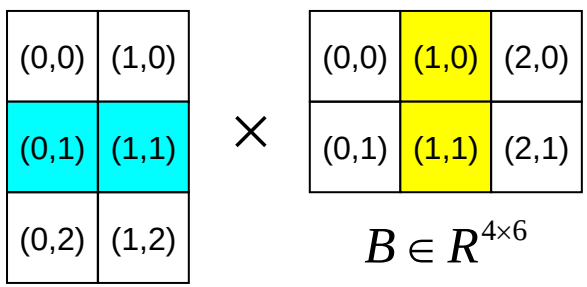
$hA\_grid = 3$ $wA\_grid = 2$ $wB\_grid = 3$

Example 5: matrix multiplication (block version) [3]

matrixMul_block.cpp

```
38      for ( by = 0 ; by < hA_grid ; by++ ){
39        for ( bx = 0 ; bx < wB_grid ; bx++ ){
40          // Index of the first sub-matrix of A processed by the block
41          aBegin = wA * BLOCK_SIZE * by;
42          // Index of the last sub-matrix of A processed by the block
43          aEnd   = aBegin + wA - 1;
44          // Step size used to iterate through the sub-matrices of A
45          aStep  = BLOCK_SIZE;
46          // Index of the first sub-matrix of B processed by the block
47          bBegin = BLOCK_SIZE * bx;
48          // Step size used to iterate through the sub-matrices of B
49          bStep  = BLOCK_SIZE * wB;
50
51          // Loop over all the sub-matrices of A and B
52          // required to compute the block sub-matrix
53          for (a = aBegin, b = bBegin;
54               a <= aEnd;
55               a += aStep, b += bStep) {
56          // Load the matrices from main memory
57            for ( ty = 0 ; ty < BLOCK_SIZE ; ty++ ){
58              for ( tx = 0 ; tx < BLOCK_SIZE ; tx++ ){
59                As[ty][tx] = A[a + wA * ty + tx];
60                Bs[ty][tx] = B[b + wB * ty + tx];
61              }// for tx ;
62            }// for ty
```

copy global data to small block, why?



$A \in R^{6\times4}$

$B \in R^{4\times6}$

$aBegin =$ physical index of first entry in block A (0,1)

$bBegin =$ physical index of first entry in block B (1,0)

Example 5: matrix multiplication (block version) [4]

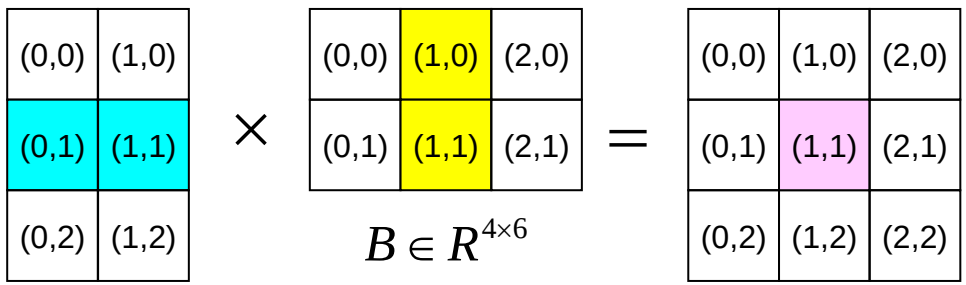
matrixMul_block.cpp

```
64          // Multiply the two matrices together
65            for ( ty = 0 ; ty < BLOCK_SIZE ; ty++ ){
66              for ( tx = 0 ; tx < BLOCK_SIZE ; tx++ ){
67                Csub = 0.0 ;
68                for (k = 0; k < BLOCK_SIZE; ++k ){
69                   Asub = As[ty][k ] ;
70                   Bsub = Bs[k ][tx] ;
71                   Csub += Asub * Bsub ;
72                }
73                c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
74                C[c + wB * ty + tx] += (doublereal) Csub;
75              }// for tx ;
76            }// for ty
77          }// for each submatrix A and B
78
79        }// for bx
80      }// for by
81  }
```

Compute submatrix of C sequentially

$$C(i,j) = \sum_{k=1}^{wA} A(i,k)\,B(k,j)$$

for all $(i,j) \in block(1,1)$



$A \in R^{6\times4}$ $B \in R^{4\times6}$ $C \in R^{6\times6}$

or equivalently $A\,(0,1)\;B\,(1,0) + A\,(1,1)\;B\,(1,1) = C\,(1,1)$

Example 5: matrix multiplication (block version) [5]

Parallel version

```
124         // Loop over all the sub-matrices of A and B
125         // required to compute the block sub-matrix
126         for (a = aBegin, b = bBegin;
127              a <= aEnd;
128              a += aStep, b += bStep) {
129         // Load the matrices from main memory
130  #pragma omp parallel for default(none) num_threads(nthreads)  \
131       shared(A,B,As,Bs,a,b,wA,wB) private(ty,tx)  \
132       schedule( static)
133           for ( ty = 0 ; ty < BLOCK_SIZE ; ty++ ){
134             for ( tx = 0 ; tx < BLOCK_SIZE ; tx++ ){
135               As[ty][tx] = A[a + wA * ty + tx];
136               Bs[ty][tx] = B[b + wB * ty + tx];
137             }// for tx ;
138           }// for ty
139
140         // Multiply the two matrices together
141  #pragma omp parallel for default(none) num_threads(nthreads)  \
142       shared(As,Bs,C,bx,by,wB) private(ty,tx,k,c,Asub,Bsub,Csub)  \
143       schedule( static)
144           for ( ty = 0 ; ty < BLOCK_SIZE ; ty++ ){
145             for ( tx = 0 ; tx < BLOCK_SIZE ; tx++ ){
146               Csub = 0.0 ;
147               for (k = 0; k < BLOCK_SIZE; ++k ){
148                  Asub = As[ty][k ] ;
149                  Bsub = Bs[k ][tx] ;
150                  Csub += Asub * Bsub ;
151               }
152               c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
153               C[c + wB * ty + tx] += (doublereal) Csub;
154             }// for tx ;
155           }// for ty
156         }// for each submatrix A and B
157
158       }// for bx
159     }// for by
160  }
```

GPU code

```
for (int a = aBegin, b = bBegin;
          a <= aEnd;
          a += aStep, b += bStep) {

    // Declaration of the shared memory array As
    // used to store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Declaration of the shared memory array Bs
    // used to store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load the matrices from device memory
    // to shared memory; each thread loads
    // one element of each matrix
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];

    // Synchronize to make sure the matrices are
    __syncthreads();

    // Multiply the two matrices together;
    // each thread computes one element
    // of the block sub-matrix
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);

    // Synchronize to make sure that the precedi
    // computation is done before loading two ne
    // sub-matrices of A and B in the next itera
    __syncthreads();
}

// Write the block sub-matrix to device memory;
// each thread writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

Example 5: matrix multiplication (block version) [6]

Let BLOCK_SIZE = 16 and $size(A) = size(B) = size(C) = (N \cdot BLOCK\_SIZE)^2$

total memory usage $= size(A) + size(B) + size(C)$ float

Platform: oectet1, with compiler icpc 10.0, -O2

N	Total size	Thread 1	Thread 2	Thread 4	Thread 8
16	0.75 MB	40 ms	34 ms	34 ms	44 ms
32	3 MB	301 ms	309 ms	240 ms	219 ms
64	12 MB	2,702 ms	2,310 ms	1,830 ms	1,712 ms
128	48 MB	24,548 ms	19,019 ms	15,296 ms	13,920 ms
256	192 MB	198,362 ms	151,760 ms	129,754 ms	110,540 ms

Non-block version

N	Total size	Thread 1	Thread 2	Thread 4	Thread 8
16	0.75 MB	53 ms	31 ms	21 ms	24 ms
32	3 MB	434 ms	237 ms	121 ms	90 ms
64	12 MB	17,448 ms	8,964 ms	6,057 ms	2,997 ms
128	48 MB	421,854 ms	312,983 ms	184,695 ms	92,862 ms
256	192 MB	4,203,536 ms	2,040,448 ms	1,158,156 ms	784,623 ms

Question 10: non-block version is much slower than block version, why?

Example 5: matrix multiplication (block version) [7]

Block version, BLOCK_SIZE = 512

N	Total size	Thread 1	Thread 2	Thread 4	Thread 8
2	12 MB	3,584 ms	1,843 ms	961 ms	453 ms
4	48 MB	27,582 ms	14,092 ms	7,040 ms	3,533 ms
8	192 MB	222,501 ms	110,975 ms	55,894 ms	28,232 ms

Block version, BLOCK_SIZE = 16

N	Total size	Thread 1	Thread 2	Thread 4	Thread 8
64	12 MB	2,702 ms	2,310 ms	1,830 ms	1,712 ms
128	48 MB	24,548 ms	19,019 ms	15,296 ms	13,920 ms
256	192 MB	198,362 ms	151,760 ms	129,754 ms	110,540 ms

Question 11: larger BLOCK_SIZE implies better performance when using multi-thread, why?

Question 12: small BLOCK_SIZE is better in single thread, why?

Question 13: matrix-matrix multiplication is of complexity O(N^3), which algorithm is "good" to achieve this property?

Example 5: matrix multiplication (block version) [8]

```
[macrold@octet1 matrixMul_block2]$
[macrold@octet1 matrixMul_block2]$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Xeon(R) CPU           X5365  @ 3.00GHz
stepping        : 11
cpu MHz         : 2000.000
cache size      : 4096 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx l
m constant_tsc arch_perfmon pebs bts rep_good pni monitor ds_cpl vmx est tm
2 ssse3 cx16 xtpr dca lahf_lm
bogomips        : 5987.36
clflush size    : 64
cache_alignment : 64
address sizes   : 38 bits physical, 48 bits virtual
power management:
```

Cache has 4 MB, we can have large BLOCK_SIZE

cache line is 64 byte (16 float)

In CPU

BLOCK_SIZE = 512 $\longrightarrow$ $size(Bs) = size(As) = 512^2\ float = 1024^2\ Byte = 1MB$

In GPU

BLOCK_SIZE = 16 $\longrightarrow$ $size(Bs) = size(As) = 16^2\ float = 1kB$

Exercise 5: verify subroutine ***matrixMul_block_seq*** with non-block version, you can use high precision package.

Non-block version

```
12  // C = A * B
13  void matrixMul_seq(doublereal* C, const doublereal* A, const doublereal* B,
14    unsigned int hA, unsigned int wA, unsigned int wB )
15  {
16      unsigned int i, j, k ;
17  #ifdef  HIGH_PRECISION_PACKAGE
18      doublereal sum ;
19      doublereal a, b ;
20  #else
21      double sum ;
22      double a, b ;
23  #endif
24      for ( i = 0; i < hA; ++i){
25          for (j = 0; j < wB; ++j) {
26              sum = 0.0 ;
27              for (k = 0; k < wA; ++k) {
28                  a = A[i * wA + k];
29                  b = B[k * wB + j];
30                  sum += a * b;
31              }// for k
32              C[i * wB + j] = (doublereal)sum;
33          }// for j
34      }// for i
35  }
```

Exercise 6: if we use “double”, how to choose value of BLOCK_SIZE, show your experimental result.

Exercise 7: Can you modify subroutine ***matrixMul_block_parallel*** to improve its performance?

Exercise 8: compare parallel computation between CPU and GPU in your host machine