OpenMP Usage

Orna Agmon Ben-Yehuda

What is this talk about?

- Dilemmas I encountered when transforming legacy code using openMP
- Tricks I found to make my life easier
- The things I bothered to remember about openMP, because I found them useful.
- Presenting option for openMP usage

This talk is not:

- Inclusive - this is why we did the tutorial first! (and there is more in the spec)
- Conclusive - this is work in progress.

Code Preparation For thread Parallelization

Thread safe programming before going to actually use the omp:

- Identify and/or eliminate static variables in your own code
- Identify and/or eliminate non-thread safe function calls.
- Protect global work area by making them threadprivate or get rid of them.
- Define variables with as small a scope as possible.
- Add const wherever possible.
- Add /*not const*/ when impossible.
- Add /*could have been const*/ when c limitations prevent adding the const, but otherwise it could have been.

Thread safe functions

A function is thread-safe when it does not use static data. There is a big overlap of reentrant and threadsafe code. For example, man 3 rand:

DESCRIPTION...

" The function rand() is not reentrant or thread-safe, since it uses hidden state that is modified on each call. This might just be the seed value to be used by the next call, or it might be something more elaborate. In order to get reproducible behaviour in a threaded application, this state must be made explicit. The function rand_r() is supplied with a pointer to an unsigned int, to be used as state. This is a very small amount of state, so this function will be a weak pseudo-random generator. Try drand48_r(3) instead."

Thought preparation for thread parallelization

- Chart a flow diagram of who calls who - top to bottom.
- Consider automatic tools like gprof (it will anyhow focus on main time consuming functions) or callgraph (valgrind 3.3.1).
- Decide on the parallelization granularity for now - Start with large granularity and refine? How much does opening a thread cost? Always profile.
- Go along the flow, parallelize from top, then if needed to balance loads, parallelize the bottomer parts.
- Add /*pll inside*/ before functions which are pll inside, but their surrounding is also pll, to keep the parallelization in the same level.

Combining Compilers

- Compilers which are not from the same house may support the syntax with a different internal implementation. Code compiled using another compiler must be treated as non-thread-safe (a critical code).
- Features left for implementation are really different and cannot be depended upon: gomp does not support nested for loops - it just gets stuck. pgcc does not get stuck, although an old PG book I have says the feature is not supported.

configure.in

```
C_OMP="-no-open-mp-support!!!"
F_OMP="-no-open-mp-support!!!"
SAME_HOUSE=0
case "$FC" in
        pgf90)FFLAGS="-r8 -i4"
                case "$CC" in
                    pgcc)SAME_HOUSE=1;;
                    *gcc);;
                esac
              F_OMP="-mp";;
        g77)  FFLAGS="-r8 -i4";;
        gfortran) FFLAGS="-fdefault-real-8 -i4"
                case "$CC" in
                    pgcc);;
```

```
                    *gcc)SAME_HOUSE=1;;
                esac
                F_OMP="-fopenmp";;
        ifort) FFLAGS="-align all -r8 -i4 -nofor_main"
        F_OMP="-openmp -openmp-lib compat";;
esac

AC_SUBST(FFLAGS)
case "$SAME_HOUSE" in
        1) CPPFLAGS="-DSAME_HOUSE $CPPFLAGS" ;;
esac
```

makefile.in

```
omp:
        $(MAKE) OUTPUT_LETTER="m" C_EXTRA_FLAGS="$(OPTIMIZE)
$(C_OMP)" F_EXTRA_FLAGS="$(OPTIMIZE) $(F_OMP)"
GPP_EXTRA_FLAGS="$(OPTIMIZE) $(C_OMP)"
EXTRA_LIBS="$(EXTRA_LIBS_M)" LIBZ_LIB=-lz_p -o
$(MACHINE_TYPE)/$(OUTPUT)m_o exe
```

Large Programs in Fortran

- setenv GOMP_STACKSIZE 10000000, needing patch for gcc 4.2 (beginning of 2008).

omp_support.h

```
#ifndef OMP_SUPPORT_H
#define OMP_SUPPORT_H

#ifdef _OPENMP
/*might not exist on some systems*/
#include <omp.h>
/**create #pragma omp blah, usage: OMP("omp for")
        Thanks to Oleg Goldshmidt*/
#define OMP(C) _Pragma(#C)
#define PRINT_THREAD(I) printf(\
"thread %d on iteration %d",omp_get_thread_num(),I)
```

omp_support.h (cont)

```
#else
#define OMP(C)
#define PRINT_THREAD(I)
#endif

/**call omp_get_num_threads even without omp*/
int protected_omp_get_num_threads(void);
#endif
```

omp_support.c

```
#include <stdio.h>
#include "omp_support.h"
/**for fortran*/ int protectedompgetnumthreads_(void);
int protectedompgetnumthreads_(void){
    return protected_omp_get_num_threads();}
int protected_omp_get_num_threads(void){
#ifdef _OPENMP
    int i=omp_in_parallel();
#else
    int i=0;
#endif
    printf("%d active threads",i);
    return i;}
```

One parallel loop

For correct and fast (programmer time) results, begin with combined:

```
int i;/*not uint!!*/
OMP(omp parallel for)
    for (i=0; i < JMAX ; ++i){}
```

Then profile and replace by longer parallel parts, with single and for instructions.

Protecting the Setting of a shared variable

- It can be protected using a critical section.
- It can be protected using an atomic pragma, which is more efficient (may be hardware supported) but supports only a single statement with specific syntax (for example referring to only one shared variable).

Can I Ignore protection of a shared variable?

When the variable is only as long as the machine's word, so there is support for an atomic operation of that size (on 64 bit - up to a double). In addition, one of the following applies:

- The previous value or value of other variables are not considered, so there is only one machine operation
- or the exact outcome does not matter (adding when only the general amount matters, setting to a non-zero value)

```
if (a[i]==-1){
        printf("a[%d] is negative",ONE(i));
        *there_is_a_negative_value=1;
        continue;
}
```

Breaking out

- Breaking out of pll loops is not allowed and will not compile in gcc.
- A continue can replace many breaks, pending on the probabilities known to programmer.
- An exit using a wrapper function will not be identified at compile time, but results will be undefined.
- If exit data is important, better to collect status in a variable and exit at the end of the loop.
- Need to treat the possibility that more than one thread reaches a certain error state.

max

Sadly, there is no reduction in openMP for min, max operations.

A maxself operation on a shared variable must be protected. The following is not safe:

```
#define maxself(a,b) a=(a>b)?a:b;
            maxself (amax,part[i].a);
```

Protecting max

- By code protection (openMP way):

```
        OMP(omp critical (a))
{
            maxself (a_max,part[i].a);
}
```

- No data locks, so we lock as little as possible, success oriented:

```
            if (a_max<part[i].a)
                OMP(omp critical (a_max))
                   maxself (a_max,part[i].a);
```

Less locks - Success oriented

The lock is taken when there is a suspicion for a need, and then, when nobody can change its value, the estimation is done again.

- This behavior fits a max out of randomly ordered data.
- Worst case: 2N checks+N locks - achieved on increasing data
- Best case: N checks +1 lock - achieved on decreasing data

GBL - Great Big Lock

- GBL Take the global (unnamed) lock at first, whenever there is danger.
- Ensure correctness.
- What about performance?
- How much work will it be to split the lock later on?
- How much more chances for mistakes, to split the lock later, given the programmer is currently most familiar with the specific part of the code, and the work is performed in stages anyway?

Parallelization level - inner loops

Parallelization can take place in different levels (inner/outer loop, extending the parallel command into dynamic context - over functions). Parallelizing Inner loops:

- Enables more evenly distributed load
- Calls to threads are more often (overhead - results may even get worse).
- Harder to trace the parallelization flow (how many time is the function called? From what scopes?)

Parallelization level -Alternatives:

- Start from the top down, and first make sure the upper loops are parallel. Then it can be refined to inner loops.
- Use dynamic scheduling policies - overhead of calculating the policy, but in bigger chunks.
- Mix master and work-sharing constructs in a larger parallel scope

Cannot exit from a loop

Legacy code to exit from a most inner loop:

```
if (retval<0)
   free_data1_structures(data1);/*internal treatment*/
goto FREE_INTERNALS;/*address out of the loops*/
```

New exit from loop:

```
int problem=0;
OMP(omp parallel for)
   for (i=0; i < JMAX ; ++i){
       if (unlikely(problem)) continue;
.../*error occured*/
    if (retval<0)
      free_data1_structures(data1);/*internal treatment*/
    problem=1;/*no goto*/
```

Using reduction for error catching

```
OMP(omp parallel for reduction(||:bad))
for (i = 0 ; i < (int)JMAX ; ++i){
   if (unlikely(bad)) continue;
   bad=func(i);
   if (bad) printf("Problem with %d",i);
   else if (func2(i)) bad=1;
}
if (bad) safely_end_run();
```

Not Using Reduction

- When the lock seldom needs to be taken
- When code change is to be avoided - we want the code to be very similar to previous code.

```
OMP(omp parallel for)
    if (part->r>-1)
     OMP(omp atomic)
     ++r_parts;
```

top(1)

- Shift i (I) toggles Irix mode.
- On (default): %CPU is out of the current cpu the process uses, which is pretty useless for smp (usually 100%).
- Off: %CPU is out of the total cpu power.
- A common mistake is to consider cpu usage as signifying success -

```
PID USER PR NI RES  SHR S %CPU %MEM TIME+   COMMAND
30  orna 15  0 7m   188  S 71.0  2.0 6:22.02 mypllm
  1 root 16  0 660  556 S  0.0  0.0 0:02.10 init
```

  These top results (Irix mode off) only means the application does run in pll, consuming 142% CPU (out of 200%). It does not mean it is more efficiency nor faster than the scalar application.

time(1)

- The basic profiling tool is time(1), which provides a bottom line.
- On small workloads, the overhead means that the more parallelization, the less you achieve:

```
> time /orna/mypll_more > & batch
2.478u 5.008s 0:13.06 57.1%     0+0k 0+0io 0pf+0w
> time /orna/mypll_less > & batch
1.984u 3.595s 0:11.41 48.8%     0+0k 0+0io 0pf+0w
```

Uniting Parallel Parts more Efficiently

```
OMP(omp parallel)
{
     OMP(omp for nowait)
     for (i=0; i < JMAX ; ++i){
        /*something that has nothing to do with the next loop*/
     }
    /*Indepent of previous task, no barrier needed*/
    OMP(omp for)
    for (i=0; i < JMAX ; ++i){
        /*this is an individual task, no barrier needed*/
    }
    /*here we already need threads to cync*/
}
```