Shared Memory Programming in OpenMP and Intel TBB

Kenjiro Taura

University of Tokyo
Today’s topics

1. What is shared memory programming?

2. OpenMP
   - OpenMP overview
   - `parallel` and `for` pragma
   - Data sharing and scheduling
   - example

3. Intel Threading Building Block
   - Overview
   - Parallel loops
   - Reductions
   - Task parallelism
Shared memory programming model

- parallel programming model in which data are *shared*
- if somebody does `a[10] = 100`; everybody will see it (sooner or later)

you wrote:

```c
double a[n];
```

concurrent activities
(often called *threads*)

```
[...]
```

a[0], a[1], a[2], ... , a[n-1]

data (shared)
Distributed memory programming model

- it’s in contrast to distributed memory programming model, in which data space are segregated among concurrent activities
  - I have my x, you have your own
  - passing pointer to my x (&x) to you does not let you read it
  - want to communicate? → send/recv messages

**concurrent activities**
*(often called processes)*

you wrote:

```c
double a[n/p];
```

![Diagram of distributed memory programming model](image)
it is a natural abstraction of shared memory *machines*, in which the CPU provides shared memory *by hardware*

- CPU 0 stores 39 to (physical) address 0x12345000, then other CPUs will see it (sooner or later)

in principle it’s possible to build the former on top of distributed memory machines (*by software*)

- we’ll see some examples in this class (UPC and Chapel)

but in today’s *widely adopted* systems,

shared memory programming model

≈ programming model on shared memory machines
1 What is shared memory programming?

2 OpenMP
   - OpenMP overview
   - parallel and for pragma
   - Data sharing and scheduling
   - example

3 Intel Threading Building Block
   - Overview
   - Parallel loops
   - Reductions
   - Task parallelism
OpenMP

- *de facto* standard model for programming shared memory machines
- C/C++/Fortran + parallel directives + APIs
  - by `#pragma` in C/C++
  - by comments in Fortran
- many free/vendor compilers, including gcc
- this tutorial uses C language and gcc compiler for it
Reference

GCC and OpenMP

- gcc 4.2 → OpenMP spec 2.5
- gcc 4.4 → OpenMP spec 3.0
- gcc 4.7 → OpenMP spec 3.1
Compiling and running OpenMP programs with gcc

- compile with `-fopenmp`

```bash
$ gcc -Wall -fopenmp program.c
```

- run the executable specifying the number of threads with `OMP_NUM_THREADS` environment variable

```bash
$ OMP_NUM_THREADS=1 ./a.out # use 1 thread
$ OMP_NUM_THREADS=4 ./a.out # use 4 threads
```

- see 2.4 for other ways to control the number of threads
Two pragmas you must know first

- `#pragma omp parallel` to launch a team of threads (2.4)
- `#pragma omp for` to distribute work to threads (2.5.1)

Note: all OpenMP pragmas have the common format: `#pragma omp ...`
#pragma parallel

- **basic syntax:**
  ```
  ...  
  #pragma omp parallel
  S
  ...
  ```

- **basic semantics:**
  - make a team of `OMP_NUM_THREADS` threads
  - the current thread becomes the *master* of the team
  - *S will be executed by each member of the team*
  - the master thread waits for all to finish and continue
#pragma omp for (work-sharing for)

- **basic syntax:**
  ```
  #pragma omp for
  for(i=...; i...; i+=...){
    S
  }
  ```

- **basic semantics:**
  The threads in the team divide the iterations among them.
#include <stdio.h>
int main() {
    #pragma omp parallel
        printf("hello\n");
        return 0;
}
Remarks (1)

the **parallel** pragma applies to only one statement that immediately follows

yet it may execute arbitrary number of statements:
- it may be a compound statement (\{ \ldots \})
- it may call arbitrary functions
Remark (2)

- `#pragma omp parallel` creates threads, *all executing the same statement*
- contrary to its name, it is *not* a means to parallelize work
- it is a combination of *parallel* and *for* that does it
not arbitrary for statement is allowed after a `for` pragma

strong syntactic restrictions apply, so that *the iteration space is easily identified at the beginning* of the loop

roughly, it must be of the form:

```plaintext
#pragma omp for
for (i = init; i < limit; i += incr)
  \( S \)
```

except `< (+=) may be other operators

*init, limit, and incr* must be loop invariant
in simple cases, \texttt{#pragma omp for} immediately follows \texttt{#pragma omp parallel}, which can be written in a single \texttt{#pragma omp parallel for}

\begin{verbatim}
#pragma omp parallel for
for(i = init; i < limit; i += incr)
  S
\end{verbatim}

≡

\begin{verbatim}
#pragma omp parallel
#pragma omp for
for(i = init; i < limit; i += incr)
  S
\end{verbatim}
Getting slightly deeper

- data sharing in `parallel` pragma (2.9)
  - reduction
- APIs to query number of threads etc. (3.2.2 - 3.2.5)
- scheduling in `for` pragma (2.5.1.1)
- work-sharing loop nests (2.5.1)
Data sharing among threads (2.4)

- **ground rule**: data are on shared memory, so everything is basically shared, *including local variables/arrays*
- local variables declared inside the parallel region are private to each thread
- you can overturn the default by optional clauses in parallel pragma (*private*, *firstprivate*, *shared*, *copyin*, *reduction*)
int main() {
    int S; /* shared */
    int P; /* made private below */
#pragma omp parallel private(P)
    {
        int L; /* automatically private */
        printf("S at %p, P at %p, L at %p\n",
                &S, &P, &L);
    }
    return 0;
}

$ OMP_NUM_THREADS=2 ./a.out
S at 0x..777f494, P at 0x..80d0e28, L at 0x..80d0e2c
S at 0x..777f494, P at 0x..777f468, L at 0x..777f46c
Reduction

- you almost always need to somehow “combine” (reduce) partial results produced by many threads in parallel
- if done poorly, it drags your speedup
- always pay attention to how reductions are supported in the language you chose
Simple reduction in OpenMP (2.5.1)

- simple reduction on scalar values done by data sharing clause
- syntax:

```c
#pragma omp parallel reduction(op:var,var,...)
```

- it is as if
  - listed variables (var,var,...) are declared as private
  - after S finished, values of listed variables from all threads are combined by the specified reduction operator op
  - op is one of +, *, -, &, ^, |, &&, and ||
  - (OpenMP 3.1) op can also be min or max
Complex reductions?

- e.g.
  - what if we reduce all elements of an array?
  - what if operation is a complex one (e.g., merging two sets)?
- you are on your own; finish a for loop and reduce by yourself
- things easily become ugly, as we’ll see later
APIs to get the number/id of threads

- `omp_get_num_threads()` (3.2.2): the number of threads in the current team
- `omp_get_max_threads()` (3.2.3): the number of threads available if the current thread executes `parallel` pragma
- `omp_get_thread_num()` (3.2.4): the current thread’s id (0, 1, ...) in the team
Scheduling (2.5.1)

- `schedule` clause in work-sharing for loop determines how iterations are divided among threads
- There are three alternatives (static, dynamic, and guided)
static, dynamic, and guided

- **schedule**(*static[,chunk]*):
  predictable round-robin

- **schedule**(*dynamic[,chunk]*):
  each thread repeats fetching *chunk* iterations

- **schedule**(*guided[,chunk]*):
  threads grab many iterations in early stages; gradually reduce iterations to fetch at a time

```c
#pragma omp for schedule(static)
#pragma omp for schedule(static,3)
0 1 2 3
#pragma omp for schedule(dynamic)
#pragma omp for schedule(dynamic,2)
#pragma omp for schedule(guided)
#pragma omp for schedule(guided,2)
```
Other scheduling options and notes

- `schedule(runtime)` determines the schedule by `OMP_SCHEDULE` environment variable. e.g.,

  ```
  $ OMP_SCHEDULE=dynamic,2 ./a.out
  ```

- `schedule(auto)` or no `schedule` clause choose an implementation dependent default (it seems `schedule(static)` in gcc implementation)

  **caution: is this a gcc bug?**

  ```
  $ OMP_SCHEDULE=static ./a.out
  ```

  appears to mean `schedule(static,1)`, not `schedule(static)`
Parallelizing loop nests by collapse

- collapse($l$) can be used to partition nested loops. e.g.,

```c
#pragma omp for collapse(2)
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        S
```

will partition $n^2$ iterations of the doubly-nested loop

- schedule clause applies to nested loops as if the nested loop is an equivalent flat loop

- restriction: the loop must be “perfectly nested” (the iteration space must be a rectangular and no intervening statement between different levels of the nest)
General nested parallelism in OpenMP

- beyond perfectly nested loops, OpenMP lacks a support of arbitrarily nested parallelism
- e.g.,
  - what if you encounter another parallel pragma during the execution of a parallelized loop?
  - parallel recursions?
- OpenMP gives you a way to control the number of threads allocated to each parallel pragma, but that’s basically it
- task parallelism (task and taskwait pragma) is a partial rescue for it, but AFAIK, performance of gcc implementation has been disappointing
- → we cover task parallelism with TBB
Problem:

1. you are given an array `double a[n]`
2. version A: compute

\[
\sum_{0 \leq i < j < n} (a[i] - a[j])^2
\]

3. version B: instead compute

\[
\min_{0 \leq i < j < n} (a[i] - a[j])^2
\]
A serial code

double sum_dist(double * a, int n) {
    double S = 0.0;
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = i + 1; i < n; i++) {
            double dx = a[i] - a[j];
            S += dx * dx;
        }
    }
    return S;
}

What we should do to parallelize?

- parallelize loops
- reduce partial sum into S → we can use OpenMP’s reduction clause, as S is a scalar and operation is a mere ’+’
double sum_dist(double * a, int n) {
    double S = 0.0;
    int i, j;
    #pragma omp parallel reduce(+: S)
    #pragma omp parallel for
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            double dx = a[i] - a[j];
            S += dx * dx;
        }
    }
    return S;
}

Quiz:

- can you see it’s trapped in a pitfall?
- which schedule clause will be appropriate?
equally simple in OpenMP \( \geq 3.1 \) (\texttt{min}) is also supported as a builtin reduction operator

let’s study how to do it without builtin reductions
**Version B : alternatives**

1. share $S$ and everybody atomically increments it ($\rightarrow$ no worry that it might scale)

2. do what the compiler is doing for builtin: duplicate $S$ for each thread and reduce them after the loop finished
   - make $S$ an array of as many elements as the number of threads
   - OK to do this final reduction sequentially for large data/thread
   - may need to do recursively when data/thread is small
double sum_dist(double * a, int n) {
    int i, j;
    // duplicate S for each thread
    int P = omp_get_max_threads();
    double * S = (double *) malloc(sizeof(double) * P);
    #pragma omp parallel
    {
        int p = omp_get_thread_num();
        S[p] = A LARGE NUMBER;
        #pragma omp for
        for (i = 0; i < n; i++) {
            for (j = i + 1; i < n; i++) {
                double dx = a[i] - a[j];
                if (dx * dx < S[p]) S[p] = dx * dx;
            }
        }
    }
}
\{ // hand-made reduction
    double s = A LARGE NUMBER;
    for (p = 0; p < P; p++)
        if (S[p] < s) s = S[p];
    return s;
\}
Intel Threading Building Blocks (TBB)

- C++ library for parallel programming
- Debian has a package libtbb-dev
- many functions
  - parallel loops
  - task parallelism
  - concurrent data structures
  - concurrent memory management
References and HOWTO’s

- visit http://threadingbuildingblocks.org/documentation.php
  - Section numbers below refer to those in this document
Compile TBB programs

- Ideally link the library and that’s it

```
g++ -Wall program.cc -ltbb
```

- You need `-std=c++0x` or `-std=gnu++0x` when using lambda expressions (supported by gcc ≥ 4.5)

```
g++ -Wall -std=c++0x program.cc -ltbb
```

- You may need `-I`, `-L`, and `-Wl,-R` when it is not installed in the system directory (e.g., under your home directory)

```
g++ -Wall -I/home/you/local/include -L/home/you/local/lib -Wl,-R/home/you/local/lib -std=c++0x program.cc
```
Run TBB programs

- just run it
  
  $ ./a.out

- it doesn’t let you specify the processor counts as easily as OpenMP (idealistic?)
Specifying number of processors used

You can do so from within your program

```
#include <tbb/task_scheduler_init.h>
int main() {
  new tbb::task_scheduler_init(n);
  ...
}
```
you almost always `#include <tbb/tbb.h>`
besides, you may need one `#include <tbb/something.h>` for each function/class you use. e.g.
  `#include <tbb/parallel_for.h>` to use `parallel_for`
do not forget to prefix all names with `tbb::` or put `using namespace tbb;` in your file
yet compilation errors around templates are undecipherable; take a deep breath and check if `you didn’t you forget any const or &?`
**parallel for** is what you must learn first

the simplest syntax for iterating over an integer region:

```c
parallel_for(a, b, f);
```

semantics: performs the following in parallel

```c
for (i = a; i < b; i++)
    f(i);
```
**parallel_for** accepts various int-taking functions

1. **a usual function**

```cpp
void f(int i) { ... }
```

2. **any object defining void operator()(int i)**

```cpp
struct F {
    void operator()(int i) const { ... }
};
```

useful to put additional variables (other than **int i**) into objects

3. **a lambda expression or a closure** (as in C++0x standard)

```cpp
[=] (int i) { ... }
```

ditto. and you avoid tedious one-time class definitions
#include <stdio.h>
#include <tbb/tbb.h>
#include <tbb/parallel_for.h>

void f(int i) { printf("f(%d)\n", i); }

struct F {
    void operator()(int i) const { printf("F::operator()(%d)\n", i); }
};

int main() {
    // regular function
    tbb::parallel_for(0, 10, &f);
    F fo;
    // function-like object
    tbb::parallel_for(0, 10, fo);
}
// closure (lambda)
tbb::parallel_for(0, 10,
    [=] (int i) {
        printf("lambda (%d)\n", i); 
    });
Lambda expression and closures

- *a closure* is a function that encapsulates values defined outside it
- *a lambda expression* is an expression that creates an anonymous function
- many modern programming languages that support first-class functions support them. e.g., in python:

```python
def make_adder(x):
    return (lambda y: x + y)
```
C++ lambda expression

- **syntax:**

  
  ```
  [data-sharing-spec] { statements }
  ```

- **data-sharing-spec** specifies which variables are copied from or shared with the parent context:
  - ```[=] { ... } ```: copy all variables mentioned in the closure
  - ```[&] { ... } ```: share all variables mentioned in the closure
  - ```[&x,&y,=] { ... } ```: share x and y and copy others
An alternative \texttt{parallel\_for} on ranges

- syntax:
  \begin{verbatim}
  parallel\_for(r, f);
  \end{verbatim}

where
- \( r \) is a “range object” specifying the iteration space, and
- \( f \) a function taking a range and works on that range (sequentially)

- pre-defined range classes (rectangular iteration spaces)
  - \texttt{blocked\_range}
  - \texttt{blocked\_range2d}
  - \texttt{blocked\_range3d}
/ include <stdio.h>
/ include <tbb/tbb.h>
/ include <tbb/parallel_for.h>
/ include <tbb/blocked_range.h>
using namespace tbb; // to reduce width

void fr(const blocked_range<int>& r) {
    for (int i = r.begin(); i != r.end(); i++) { printf("fr(%d)\n", i); }
}

struct FR {
    void operator() (const blocked_range<int>& r) const {
        for (int i = r.begin(); i != r.end(); i++) {
            printf("fr(%d)\n", i);
        }
    }
};
int main() {
    parallel_for(blocked_range<int>(0, 10), &fr);
    FR fro;
    parallel_for(blocked_range<int>(0, 10), fro);
    parallel_for(blocked_range<int>(0, 10),
                 [=] (const blocked_range<int>& r) {
                    for (int i = r.begin(); i != r.end();
                         i++) {
                        printf("lambda (%d)\n", i);
                    }
                });
}
parallel_for (and many other functions in TBB) are extensible

A range object you pass to parallel_for is an instance of any class $R$ that implements the following interfaces:

- `bool empty()`: returns true if it’s empty
- `is_divisible()`: returns true if it’s divisible
- `R(R& r, split)`: splits it into two

You may define your own range class that fits your purpose (e.g., non-rectangular region)
Execution model of \texttt{parallel\_for}

- the runtime system splits the range until it becomes not divisible
- and dispatches indivisible ranges to workers
Reduction in TBB

- It can be done a `parallel_reduce` template function.
- It is similar to `parallel_for` working over a range.
- In addition to a function that works on a range, it also takes a function that specifies how to combine two partial results into one (i.e., add two partial sums of an array).
- Mathematically, it is a tool to compute the following:

\[ e \oplus f(x_0) \oplus f(x_1) \oplus f(x_2) \oplus \cdots \quad (\forall x_i \in R) \]
parallel_reduce (4.5)

- syntax:

```c
parallel_reduce(R, e, F, C);
```

- semantics: altogether, it computes

\[ F(e, r_0) \oplus F(e, r_1) \oplus \cdots \]

where \( \{r_i\}_{i=0,1,...} \) is a disjoint partition of \( R \) and \( x \oplus y \equiv C(x, y) \)

- it may contract some terms, like

\[ F(F(e, r_0), r_1) \oplus F(F(F(e, r_2), r_3), r_4) \cdots \]
in other words, you are asked to provide:

- $F(v, r)$: a function that computes
  $$v \oplus f(x_0) \oplus f(x_1) \oplus \cdots \quad (\forall x_i \in r)$$

and

- $C(x, y)$: a function that computes
  $$x \oplus y$$

- $e$: a value that satisfies
  $$e \oplus x = x$$
Task parallelism

- *task parallelism* vaguely refers to parallelism dynamically created at an arbitrary point of execution.
- Two basic primitives are:
  - create task
  - wait for some tasks to finish
- I contrast it to *loop parallelism*, which extracts parallelism only by partitioning (often only perfectly nested) loop.
- It’s often contrasted to *data parallelism*, which refers to parallelism extract from lots of data, but it’s misleading, IMO (source of parallelism is irrelevant).
Task parallelism in TBB

- provided by **task_group** class
  - there are lower-level APIs, but forget about them

**syntax (example):**

```cpp
task_group tg; // create a group
tg.run(f); // create a task that belongs to it
tg.run(g); // as many times as you want
...
tg.wait(); // wait for all tasks that belong to it
```

where \( f \) and \( g \) are functions taking no parameters

- as always, they can be regular functions, objects defining `operator()()`, or lambda expressions.
task_group and parallel recursion

- task_group is a perfect vehicle to express \textit{parallel} recursions

```c
// quicksort \([a...b)\)
qs(double * a, double * b) {
    if (b - a > 1) {
        task_group tg;
        double * c = split(a, b);
        //\texttt{aka\{tg.run([=] \{ qs(a, c);
                     
        \}}
        qs(c+1, b);
        tg.wait();
    }
}
```
parallel recursion and parallel loops

- The execution model of `parallel_for` and `parallel_reduce` already suggest they are parallel recursions too.
- Many things look more straightforward when you use recursions rather than syntactically ugly template functions.

```cpp
void parallel_for(range& r, func& f) {
    if (!is_divisible(r)) {
        f(r); // leaf -> call f
    } else {
        // split r into h and r
        range h(r, split);
        task_group tg;
        tg.run([=] {
            parallel_for(h, f); });
        parallel_for(r, f);
        tg.wait();
    }
}
```
Next week ... 

- an interesting way to parallelize our running example using recursion
- what you must know to improve/understand/analyze *performance* of shared memory programs?