
And some slides from
OpenMP Usage / Orna Agmon Ben-Yehuda
www.haifux.org/lectures/209/openmp.pdf
OpenMP

- Shared memory parallel applications API for C, C++, and Fortran
- High-level directives for threads manipulation
- OpenMP directives provide support for:
  - Concurrency
  - Synchronization
  - Data handling
OpenMP

Based on the `#pragma` compiler directive

- Usually not recommended (not portable between compilers)
- But, OpenMP is considered a standard

Hides the dirty details of multithreaded programming

- Lets the compiler to the dirty work for you
- Injects its code to your program using the `#pragma` directive
Reminder – Quicksort

- Choose one of the elements to be the *pivot*.
- Rearrange list so that all smaller elements are before the pivot, and all larger ones are after it.
- After rearrangement, *pivot* is in its final location!
- Recursively rearrange smaller and larger elements.
- Divide and conquer algorithm – suitable for parallel execution (why?).
Serial Quicksort code

```c
void quicksort(int* values, int from, int to) {
    if (from < to) {
        int p = values[from];
        int b = from+1, t=to;
        while(b <= t) {
            while(t >= b && values[t] >= p) t--;
            while(b <= t && values[b] <= p) b++;
            if (b <= t)
                swap(values[b++], values[t--]);
        }
        swap(values[from], values[t]);
        quicksort(values, from, t);
        quicksort(values, t + 1, to);
    }
}
```

(a) \[3 2 1 5 8 4 3 7\]

(b) \[1 2 3 5 8 4 3 7\]

(c) \[1 2 3 3 4 5 8 7\]

(d) \[1 2 3 3 4 5 7 8\]

(e) \[1 2 3 3 4 5 7 8\]

**Pivot**

**Final position**
void quicksort(int* values, int from, int to) {
    if (from < to) {
        // ... choose pivot and do swaps ...

        #pragma omp parallel sections
        {
            #pragma omp section
            quicksort(values, from, t);
            quicksort(values, t + 1, to);
        }
    }
}

Sections are performed in parallel

Without omp_set_nested(1) no threads will be created recursively

Is this efficient?
Naïve parallel Quicksort – Try 2

```c
void quicksort(int* values, int from, int to) {
    ...
    bool create_thread;
    static int max_threads = omp_get_max_threads();
    static int threads_num = 1;
    if (from < to) {
        // ... choose pivot and do swaps ...
        #pragma omp atomic
        create_thread = (++threads_num <= max_threads);
        if (create_thread) {
            #pragma omp parallel sections
            {
                #pragma omp section
                quicksort(values, from, t);
                #pragma omp section
                quicksort(values, t + 1, to);
            }
            #pragma omp atomic
            threads_num--;
        } else {
            #pragma omp atomic
            threads_num--;
            quicksort(values, from, t);
            quicksort(values, t + 1, to);
        }
    }
}
```

Does this compile? What’s the atomic action?
void quicksort(int* values, int from, int to) {
    ...
    if (from < to) {
        // ... choose pivot and do swaps ...
        bool create_thread = change_threads_num(1);
        if (create_thread) {
            #pragma omp parallel sections
            {
                #pragma omp section
                quicksort(values, from, t);
                #pragma omp section
                quicksort(values, t + 1, to);
            }
            change_threads_num(-1);
        } else {
            quicksort(values, from, t);
            quicksort(values, t + 1, to);
        }
    }
}
Naïve parallel Quicksort – Try 3

```c
bool change_threads_num(int change) {
    static int threads_num = 1;
    static int max_threads = omp_get_max_threads();
    bool do_change;

    #pragma omp critical
    {
        threads_num += change;
        if (threads_num <= max_threads) return true;
        else {
            threads_num -= change;
            return false;
        }
    }
}
```
Breaking Out

• Breaking out of parallel loops (and any other structured block) is not allowed and will not compile.
  • A continue can sometimes be used instead of break.

```c
#pragma omp parallel
#pragma omp for
{
  for (i=0; i < 256; i++) {
    bool b = some_work();
    if (b) break;
  }
}
```

```c
bool b = false;
#pragma omp parallel
#pragma omp for
{
  for (i=0; i < 256; i++) {
    if (b) continue;
    if (some_work()) b = true;
  }
}
```
Breaking Out

- Careful when changing flow prevents reaching barriers.

```cpp
#pragma omp parallel
#pragma omp for
{
    for (i=0; i < 256; i++) {
        bool b = some_work();
        if (b) break;
        if (i % 16 == 0)
            #pragma omp barrier
    }
}
```

```cpp
bool b = false;
#pragma omp parallel
#pragma omp for
{
    for (i=0; i < 256; i++) {
        if (b) continue;
        if (some_work()) b = true;
        if (i % 16 == 0)
            #pragma omp barrier
    }
}
```
Breaking Out

- Calling `exit()` or throwing an exception from nested function will lead to undefined behavior.
- If exit data is important, collect status in a variable.
- What if a few threads reach error state?
Naïve parallel Quicksort – Try 3.1

```c
bool change_threads_num(int change) {
    static int threads_num = 1;
    static int max_threads = omp_get_max_threads();
    bool do_change;

    #pragma omp critical
    {
        do_change = (threads_num + change) <= max_threads;
        if (do_change) threads_num += change;
    }

    return do_change;
}
```

Is this always better than non-atomic/syncronized actions?

Should thread_num be volatile?
bool change_threads_num(int change) {
    static int threads_num = 1;
    static int max_threads = omp_get_max_threads();

    bool do_change = change < 0 || (threads_num + change) <= max_threads;

    if (do_change) {
        #pragma omp atomic
        threads_num += change;
    }

    return do_change;
}
Moving data between threads

- `#pragma omp flush [(list)]`
- OpenMP `flush (list) directive can be used for specific variables:
  1. Write variable on thread A
  2. Flush variable on thread A
  3. Flush variable on thread B
  4. Read variable on thread B
- Parameters are flushed by the order of the list.
- More about OpenMP memory model later on this course.
Naïve parallel Quicksort – Try 3.3

```c
bool change_threads_num(int change) {
    static int threads_num = 1;
    static int max_threads = omp_get_max_threads();

    #pragma omp flush (threads_num)
    bool do_change = change < 0 || (threads_num + change) <= max_threads;

    if (do_change) {
        #pragma omp atomic
        threads_num += change;
    }

    return do_change;
}
```
Lessons from naïve Quicksort

- Control the number of threads when creating them recursively.
- No compare-and-swap operations in OpenMP 😞
- Critical sections are relatively expensive.
- Implicit flush occurs after barriers, critical sections and atomic operations (only to the accessed address).
- Implicit barrier takes place at the entry and exit of work-sharing construct – parallel, critical, ordered and the exit of single construct.
Efficient Quicksort

- Naïve implementation performs partitioning serially
- We would like to parallelize partitioning
- In each step
  - Pivot selection
  - Local rearrangement
  - Global rearrangement
Efficient Quicksort

First Step

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>13</td>
<td>18</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>20</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>3</td>
<td>16</td>
<td>19</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>5</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>12</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

pivot selection

pivot=7

after local rearrangement

after global rearrangement

Second Step

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>18</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>14</td>
<td>20</td>
<td>6</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>4</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>8</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>

pivot selection

pivot=5

pivot=17

after local rearrangement

after global rearrangement

Third Step

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>14</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>9</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>19</td>
<td>20</td>
<td>19</td>
</tr>
</tbody>
</table>

pivot selection

pivot=11

after local rearrangement

after global rearrangement

Fourth Step

<table>
<thead>
<tr>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>

after local rearrangement

Solution
Efficient global rearrangement

- Without efficient global rearrangement the partitioning is still serial.
- Each processor will keep track of smaller \((S_i)\) and larger \((L_i)\) numbers than pivot.
- Then we will sum prefixes to find the location for each value.
- Prefix sum can be done in parallel (PPC)
Lessons from efficient Quicksort

- Sometimes we need to rethink parallel algorithms, and make them different from serial ones.
- Algorithms should be chosen by actual performance and not only complexity.
- Pivot selection has higher importance in the efficient Quicksort
  - If we choose largest element as pivot the problem size is reduced by 1 and we have only $p-1$ processors for it!
Backup
void p_quicksort(int* A, int n) {
    struct proc *d;
    int i, p, ss, p_loc, done = 0;
    int *temp, *next, *cur = A;
    next = temp = malloc(sizeof(int) * n);
    #pragma omp parallel shared(next, done, d, p) private(i, s, ss, p_loc)
    {
        int c = omp_get_thread_num();
        p = omp_get_num_threads();
        #pragma omp single
d = malloc(sizeof(struct proc) * p);
        d[c].start = c * n / p;
        d[c].end = (c + 1) * n / p - 1;
        d[c].sec = d[c].done = 0;
        do {
            s = d[c].start;
            if (c == d[c].sec && !d[c].done) {
                int p_loc = quick_select(cur+s, d[c].end-s+1);
                swap(cur[s], cur[s+p_loc]); /* Set pivot at start */
            }
            ss = d[d[c].sec].start; /* pivot location */
            #pragma omp barrier
            /* Local Rearrangement */
            for (i = d[c].start; i <= d[c].end; i++)
                if (cur[i] <= cur[ss]) {
                    swap(cur[s], cur[i]);
                    s++;
                }
    }
}
More code for the brave...

```c
/* Sum prefix */
d[c].s_i = max(s - d[c].start, 0);
d[c].l_i = max(d[c].end - s + 1, 0);
#pragma omp barrier
if (c == d[c].sec)
    d[c].p_loc = s_prefix_sum(d, d[c].sec, p) + d[c].start - 1;
/* Global rearrangement */
#pragma omp barrier
p_loc = d[c].p_loc = d[d[c].sec].p_loc;
memcpy(next + ss + d[c].s_i, cur + d[c].start, sizeof(int)*(s - d[c].start));
memcpy(next + p_loc + d[c].l_i + 1, cur + s, sizeof(int)*(d[c].end - s + 1));
#pragma omp barrier
/* Put pivot in place */
if (d[c].sec == c && !d[c].done) {
    swap(next[ss], next[p_loc]);
    cur[p_loc] = next[p_loc]; /* the pivot in its final current */
}
#pragma omp barrier
#pragma omp single
{
    done = p_rebalance(d, p);
    swap(cur, next);
}
} while (!done);
/* One more iteration sequentially */
seq_quicksort(cur+d[c].start, 0, d[c].end - d[c].start + 1);
#pragma omp barrier
if (A != cur)
    memcpy(A + n*c/p, cur + n*c/p, sizeof(int)*(n*(c+1)/p - n*c/p));
}
free(d); free(temp);
```
Efficient Quicksort – last bits

int p_rebalance(struct proc* d, int p) {
    int c, new_sec, sec, done = 1;
    new_sec = 1;
    for (c = 0; c < p; c++) {
        d[c].sec = sec = new_sec ? c : max(sec, d[c].sec);
        new_sec = 0;
        if (c+1 < p && d[c].sec == d[c+1].sec && d[c].p_loc >= d[c].start && d[c].p_loc <= d[c].end) {
            d[c+1].start = d[c].p_loc + 1;
            d[c].end = d[c].p_loc - 1;
            new_sec = 1;
        }
        if (c+2 < p && d[c+2].sec == d[c].sec && d[c].p_loc >= d[c+1].start && d[c].p_loc <= d[c+1].end &&
            d[c+2].end - d[c].p_loc > d[c].p_loc - d[c].start) {
            d[c+1].start = d[c].p_loc + 1;
            d[c].end = d[c].p_loc - 1;
            new_sec = 1;
        }
        if (d[c].end < d[c].start) {
            d[c].sec = c;
            new_sec = 1;
            d[c].done = 1;
        } else
            if (d[c].end > d[sec].start && d[c].sec != c)
                done = 0;
    }
    return done;
}

This is the serial implementation. Could also be done in parallel.
Moving data between threads

- Barriers may be overkill (overhead...).
- OpenMP `flush (list) directive can be used for specific variables:
  1. Write variable on thread A
  2. Flush variable on thread A
  3. Flush variable on thread B
  4. Read variable on thread B
- Parameters are flushed by the order of the list.
- More about OpenMP memory model later on this course.