インテル® C++ コンパイラーと OpenMP* 4.5 ライブラリーを使用した効率良い並列 3 分岐基数クイックソート

この記事は、インテル® デベロッパー・ゾーンに公開されている「An Efficient Parallel Three-Way Quicksort Using Intel C++ Compiler And OpenMP 4.5 Library」の日本語参考訳です。

はじめに

この記事では、有名なヒープソートやマージソート・アルゴリズムよりも、漸近的に高速で効率良い並列 3 分岐基数クイックソートを実装する C++11 コードを紹介します。また、qsort(…) (ANSI C) や std::sort(…) (ISO/IEC 14882(E)) などの既存の高速なクイックソート実装と比較して、優れたパフォーマンスをもたらす並列コードについても説明します。

具体的には、3 分岐基数クイックソート・アルゴリズムとその複雑性について調べて、インテル® C++ コンパイラーと OpenMP* 4.5 ライブラリーを使用して、ソートを並列に実行する現代的なコードの記述方法を詳しく述べます。OpenMP* の並行タスクを使用して再帰的サブソートを並列化し、アルゴリズム全体のパフォーマンスを大幅に向上する方法を説明します。

そして、インテル® Core™ i9 プロセッサー・ファミリーやインテル® Xeon® プロセッサー E5 ファミリーなど、インテル製の対称型マルチコア CPU を搭載したマシンで並列ソートを行うサンプルプログラムを実行して、C++ 標準ライブラリーの std::sort(…) 実装のパフォーマンスと比較して評価します。

3 分岐基数クイックソート・アルゴリズム

3 分岐基数クイックソートは、配列全体をピボット要素の値より小さい、等しい、または大きい要素で構成される 3 つのパーティションに分割します。そして、左右のパーティションを同じように 3 分割することで再帰的にソートします。3 分岐基数クイックソートは、膨大な要素数の配列ソートを可能にするだけでなく、ベストケースのソートの複雑性を準線形 O(n log n) から線形 O(n) に軽減します。以下のアルゴリズムは、ヒープソートやマージソート・アルゴリズムよりも、O((n log n) / n) = O(log n) 倍高速です。さらに、3 分岐基数・クイックソートは、キャッシュ・コヒーレントであり、CPU のキャッシュ・パイプラインに大きな影響を与えるため、一般にソート全体のパフォーマンスにも影響します。

3 分岐基数・クイックソート・アルゴリズムは、配列内の各要素を左から右へ一度だけパススルーします。配列内の各要素は、以下の 3 分岐比較によりピボット値と比較されます。3 分岐比較は、要素がピボット値よりも小さいか、大きいか、または等しいかの 3 つのケースを処理します。要素がピボット値よりも小さい場合は左のパーティションと交換し、要素がピボット値よりも大きい場合は右のパーティションと交換します。要素がピボット値と等しい場合、交換は行われません。この処理では、left、right、および i の 3 つのインデックスを使用します。インデックス i は配列内の各要素へのアクセスに使用され、インデックス left と right は左右のパーティションとの要素の交換に使用されます。

3 分岐基数・クイックソート・アルゴリズムは、次のように定義できます。

1. 要素の配列は arr[0..N] とし、最初と最後の要素のインデックスは low と high とします。
2. 最初の要素の値をピボットにします (pivot = arr[low])。
3. left 変数を最初の要素のインデックスに初期化します (left = low)。
4. right 変数を最後の要素のインデックスに初期化します (right = high)。
5. 変数 i を第 2 要素のインデックスに初期化します (i = low + 1)。
6. 配列の各 i 番目の要素 arr[i] (i <= high) に対して、次の処理を行います。

i 番目の要素 arr[i] とピボット値を比較します。

1. i 番目の要素 arr[i] がピボット値よりも小さい場合、left インデックスの要素と交換して、left インデックスと i インデックスを 1 つインクリメントします。
2. i 番目の要素 arr[i] がピボット値よりも大きい場合、right インデックスの値を交換して、right インデックスを 1 つデクリメントします。
3. i 番目の要素 arr[i] がピボット値と等しい場合、交換は行わず、インデックス i を 1 つインクリメントします。
4. 配列の左のパーティション arr[low..left - 1] を再帰的にソートします。
5. 配列の右のパーティション arr[right + 1..high] を再帰的にソートします。

3 分割を実行後、以下のアルゴリズムは左右のパーティションのソートを個別に再実行します。この処理は、再帰的なソートを並列に実行する 2 つの並行タスクをスポーンすることで行えます。

効率良い並列ソート

以下の現代的な C++11 コードは、3 分岐基数・クイックソート・アルゴリズムを実装します。

```
namespace internal
{
    std::size_t g_depth = 0L;
    const std::size_t cutoff = 1000000L;

    template<class RanIt, class _Pred>
    void qsort3w(RanIt _First, RanIt _Last, _Pred compare)
    {
        if (_First >= _Last) return;

        std::size_t _Size = 0L; g_depth++;
        if ((_Size = std::distance(_First, _Last)) > 0)
        {
            RanIt _LeftIt = _First, _RightIt = _Last;
            bool is_swapped_left = false, is_swapped_right = false;
            typename std::iterator_traits<RanIt>::value_type _Pivot = *_First;

            RanIt _FwdIt = _First + 1;
            while (_FwdIt <= _RightIt)
            {
                if (compare(*_FwdIt, _Pivot))
                {
                    is_swapped_left = true;
                    std::iter_swap(_LeftIt, _FwdIt);
                    _LeftIt++; _FwdIt++;
                }

                else if (compare(_Pivot, *_FwdIt)) {
                    is_swapped_right = true;
                    std::iter_swap(_RightIt, _FwdIt);
                    _RightIt--;
                }

                else _FwdIt++;
```

```
            }

            if (_Size >= internal::cutoff)
            {
                #pragma omp taskgroup
                {
                    #pragma omp task untied mergeable
                    if ((std::distance(_First, _LeftIt) > 0) && (is_swapped_left))
                        qsort3w(_First, _LeftIt - 1, compare);

                    #pragma omp task untied mergeable
                    if ((std::distance(_RightIt, _Last) > 0) && (is_swapped_right))
                        qsort3w(_RightIt + 1, _Last, compare);
                }
            }

            else
            {
                #pragma omp task untied mergeable
                {
                    if ((std::distance(_First, _LeftIt) > 0) && is_swapped_left)
                        qsort3w(_First, _LeftIt - 1, compare);

                    if ((std::distance(_RightIt, _Last) > 0) && is_swapped_right)
                        qsort3w(_RightIt + 1, _Last, compare);
                }
            }
        }
    }

    template<class BidirIt, class _Pred >
    void parallel_sort(BidirIt _First, BidirIt _Last, _Pred compare)
    {
        std::size_t pos = 0L; g_depth = 0L;
        if (!misc::sorted(_First, _Last, pos, compare))
        {
            #pragma omp parallel num_threads(12)
            #pragma omp master
                internal::qsort3w(_First, _Last - 1, compare);
        }
    }
}
```

このコードは、通常、データフローの依存関係により並列に実行できないため、3 分割をシーケンシャルに実行します。また、このケースでは、3 分割の複雑性は常に O(n) であり、十分なパフォーマンスの向上が得られることから、並列処理を最適化する必要はありません。

そのため、再帰的なソートを行うコードを簡単に並列実行して、ソート全体のパフォーマンスを大幅に向上できます。再帰的なソートを並列に実行するため、#pragma omp taskgroup {} ディレクティブを使用して、実行時に 2 つの OpenMP* 並行タスクを生成するコードを実装します。これらのタスクは両方とも OpenMP* の #pragma omp task untied mergeable {} ディレクティブによりスケジュールおよび起動され、個別のスレッドで再帰的なソートを実行します。ここでは、以下のタスクが複数のスレッドで実行されるように untied 節を使用しています。同様に、タスクが生成するコードと同じデータ・コンテキストを使用するように mergeable 節を指定しています。

```
        if (_Size >= internal::cutoff)
        {
            #pragma omp taskgroup
            {
                #pragma omp task untied mergeable
                if ((std::distance(_First, _LeftIt) > 0) && (is_swapped_left))
                    qsort3w(_First, _LeftIt - 1, compare);

                #pragma omp task untied mergeable
                if ((std::distance(_RightIt, _Last) > 0) && (is_swapped_right))
                    qsort3w(_RightIt + 1, _Last, compare);
            }
        }

        else
        {
            #pragma omp task untied mergeable
            {
                if ((std::distance(_First, _LeftIt) > 0) && is_swapped_left)
                    qsort3w(_First, _LeftIt - 1, compare);

                if ((std::distance(_RightIt, _Last) > 0) && is_swapped_right)
                    qsort3w(_RightIt + 1, _Last, compare);
            }
        }
```

スケジュールされる最初のタスクが左のパーティションのソートを実行し、2 つ目のタスクが右のパーティションのソートを実行します。

上記のコードは、条件付きタスクを実行します。タスクを生成する前に、空のパーティションをソートしないように、パーティションをソートする必要があるかどうかをチェックします。ソートの必要がある場合、ソートを実行するため、qsort3w(…) 関数を再帰的に呼び出すタスクをスポーンします。

並列 3 分岐基数・クイックソートに効果的な最適化方法があります。重複する要素が多い配列をソートする場合、ソートの再帰の深さが増加し、コードが膨大な数の並列タスクを生成することで同期とタスク・スケジュールのオーバーヘッドが大きくなることがあります。このような問題が発生しないように、最初にソートする配列のサイズがしきい値を超えていないかチェックするコードを実装します。しきい値を超えていなければ、通常、前述のように 2 つの並行タスクを生成します。しきい値を超える場合は、qsort3w(…) 関数の再帰呼び出しをマージして 1 つのスレッドで実行します。これにより、スケジュールされる並列再帰タスクの数を減らします。

ソートの全プロセスは、qsort3w(…) 関数を呼び出すことから始まります。

```
template<class BidirIt, class _Pred >
void parallel_sort(BidirIt _First, BidirIt _Last, _Pred compare)
{
    std::size_t pos = 0L; g_depth = 0L;
    if (!misc::sorted(_First, _Last, pos, compare))
    {
        #pragma omp parallel num_threads(12)
        #pragma omp master
            internal::qsort3w(_First, _Last - 1, compare);
    }
}
```

OpenMP* の task ディレクティブではなく、並列領域のマスタースレッドで qsort3w(…) を呼び出すことを推奨します。

C++11 サンプルプログラム

以下のサンプルプログラムは、通常の std::sort と並列コードのソートの実行時間を評価して、並列コードのパフォーマンスと効率を検証します。

```
namespace parallel_sort_impl
{
#if defined( _WIN32 )
    static HANDLE hStdout = ::GetStdHandle(STD_OUTPUT_HANDLE);
    const WORD wdRed = FOREGROUND_RED | FOREGROUND_INTENSITY;
    const WORD wdWhite = FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE;
#endif

    void stress_test(void)
    {
        while (true)
        {
            std::size_t count = 0L;
            std::vector<std::int64_t> array, array_copy;
            misc::init(array, std::make_pair(std::pow(10, misc::minval_radix), \
                std::pow(10, misc::maxval_radix)), count);

            array_copy.resize(array.size());
            std::copy(array.begin(), array.end(), array_copy.begin());

            std::chrono::system_clock::time_point \
                time_s = std::chrono::system_clock::now();

            std::cout << "sorting an array...\n";

            std::sort(array.begin(), array.end(),
                [](std::int64_t first, std::int64_t end) { return first < end; });

            std::chrono::system_clock::time_point \
                time_f = std::chrono::system_clock::now();

            std::chrono::system_clock::duration \
                std_sort_time_elapsed = time_f - time_s;

            std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(4)
                << "array size = " << count << " execution time (std::sort): "
                << std_sort_time_elapsed.count() << " ms ";

            time_s = std::chrono::system_clock::now();

            internal::parallel_sort(array_copy.begin(), array_copy.end(),
                [](std::int64_t first, std::int64_t end) { return first < end; });

            time_f = std::chrono::system_clock::now();

            std::size_t position = 0L;
            std::chrono::system_clock::duration \
                qsort_time_elapsed = time_f - time_s;

            bool is_sorted = misc::sorted(array_copy.begin(), array_copy.end(), position,
                [](std::int64_t first, std::int64_t end) { return first < end; });

            std::double_t time_diff = \
                std_sort_time_elapsed.count() - qsort_time_elapsed.count();

            #if defined( _WIN32 )
            ::SetConsoleTextAttribute(hStdout, \
                (is_sorted == true) ? wdWhite : wdRed);
            #else
            if (is_sorted == false)
                std::cout << "\033[1;31m";
            #endif

            std::cout << "<--> (internal::parallel_sort): " << qsort_time_elapsed.count()
                    << " ms " << "\n";

            std::cout << "verification: ";

            if (is_sorted == false) {
                std::cout << "failed at pos: " << position << "\n";
                std::cin.get();
                misc::print_out(array_copy.begin() +
                        position, array_copy.end() + position + 10);
            }

            else {
                std::double_t ratio = qsort_time_elapsed.count() / \
                    (std::double_t)std_sort_time_elapsed.count();
                std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(2)
                    << "passed... [ time_diff: " << std::fabs(time_diff)
```

```
                    << " ms (" << "ratio: " << (ratio - 1.0) * 100
                    << "% (" << (1.0f / ratio) << "x faster)) depth = "
                    << internal::g_depth << " ]" << "\n";
            }

            std::cout << "\n";

            #if !defined ( _WIN32 )
            if (is_sorted == false)
                std::cout << "\033[0m";
            #endif
        }
    }

    void parallel_sort_demo(void)
    {
        std::size_t count = 0L;
        std::cout << "Enter the number of data items N = "; std::cin >> count;

        std::vector<std::int64_t> array;
        misc::init(array, std::make_pair(std::pow(10, misc::minval_radix), \
            std::pow(10, misc::maxval_radix)), count);

        std::chrono::system_clock::time_point \
            time_s = std::chrono::system_clock::now();

        internal::parallel_sort(array.begin(), array.end(),
            [](std::int64_t first, std::int64_t end) { return first < end; });

        std::chrono::system_clock::time_point \
            time_f = std::chrono::system_clock::now();

        std::size_t position = 0L;
        std::chrono::system_clock::duration \
            qsort_time_elapsed = time_f - time_s;

        std::cout << "Execution Time: " << qsort_time_elapsed.count()
                  << " ms " << "depth = " << internal::g_depth << " ";

        bool is_sorted = misc::sorted(array.begin(), array.end(), position,
            [](std::int64_t first, std::int64_t end) { return first < end; });

        std::cout << "(verification: ";

        if (is_sorted == false) {
            std::cout << "failed at pos: " << position << "\n";
            std::cin.get();
            misc::print_out(array.begin() + position, array.end() + position + 10);
        }

        else {
            std::cout << "passed...)" << "\n";
        }

        char option = &apos;\0&apos;;
        std::cout << "Do you want to output the array [Y/N]?"; std::cin >> option;

        if (option == &apos;y&apos; || option == &apos;Y&apos;)
            misc::print_out(array.begin(), array.end());
    }
}

int main()
{
    std::string logo = "Parallel Sort v.1.00 by Arthur V. Ratz";
    std::cout << logo << "\n\n";

    char option = &apos;\0&apos;;
    std::cout << "Do you want to run stress test first [Y/N]?"; std::cin >> option;
    std::cout << "\n";

    if (option == &apos;y&apos; || option == &apos;Y&apos;)
        parallel_sort_impl::stress_test();
    if (option == &apos;n&apos; || option == &apos;N&apos;)
        parallel_sort_impl::parallel_sort_demo();

    return 0;
}

#endif // PARALLEL_STABLE_SORT_STL_CPP
```

サンプルプログラムを実行すると、ここで紹介した並列ソートは、通常の qsort または std::sort 関数よりもかなり高速 (最大 2 ~ 6 倍) であることが分かります。

添付ファイル	サイズ
parallel-sort-w64.zip	8.5KB
parallel-sort-w64-exe.zip	52.1KB

製品とパフォーマンス情報

[1] インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。

注意事項の改訂 #20110804