DPC++ 言語を使用した MPI プログラムのコンパイルと実行

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「Compile and Run MPI Programs Using DPC++ Language」の日本語参考訳です。

はじめに

メッセージ・パッシング・インターフェース (MPI) は、分散コンピューティング環境でマルチプロセッサー・プログラムを実行できるプログラミング・モデルです。インテル® oneAPI データ並列 C++ (DPC++) (英語) の登場により、開発者は、CPU、GPU、FPGA を含む多様なプラットフォームで実行可能な、単一のソースコードを記述できるようになりました。MPI と DPC++ 言語を組み合わせることで、開発者は、分散コンピューティング環境でアプリケーションを実行しながら、多様なプラットフォーム間でスケーリングを行うことができます。この記事では、開発者に、この組み合わせの例、DPC++ コンパイラーを使用して MPI アプリケーションをコンパイルする方法、および Linux* オペレーティング・システムで作成した MPI アプリケーションを実行する方法を示します。

MPI と DPC++ の統合

コードサンプルは、MPI コードと DPC++ コードを組み合わせた例です。アプリケーションは、ワークをすべての MPI プロセス (ランク) に均等に分割して円周率 (π) を計算する MPI プログラムです。円周率は、その積分表示を適用して計算できます。

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

各 MPI ランクは、上記の式に従って円周率の部分結果を計算します。計算の最後に、MPI マスターランクはほかのランクの部分結果をすべて加算して、その結果を出力します。ソースコードは、インテル® oneAPI HPC ツールキットのコードサンプルの一部で、GitHub* (英語) からダウンロードできます (MIT ライセンス (英語) 条件でリリースされています)。
円周率の部分結果を計算するため、各 MPI ランクは `main` 関数内で `CalculatePiParallel` 関数を呼び出します。

```
int main(int argc, char* argv[]) {
  int i, id, num_procs;
  float total_pi;
  MPI_Status stat;

  // MPI を開始。
  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) {
    cout << "Failed to initialize MPI\n";
    exit(-1);
```

```
  }

  // コミュニケーターを作成してプロセスの数を取得。
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

  // プロセスのランクを決定。
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  int num_step_per_rank = kTotalNumStep / num_procs;
  float* results_per_rank = new float[num_step_per_rank];

  for (size_t i = 0; i < num_step_per_rank; i++) results_per_rank[i] = 0.0;

  // 円周率の一部を並列で計算。
  CalculatePiParallel(results_per_rank, id, num_procs);

  float sum = 0.0;

  for (size_t i = 0; i < num_step_per_rank; i++) sum += results_per_rank[i];

  delete[] results_per_rank;

  MPI_Reduce(&sum, &total_pi, 1, MPI_FLOAT, MPI_SUM, kMaster, MPI_COMM_WORLD);

  if (id == kMaster) cout << "---> pi= " << total_pi << "\n";

  MPI_Finalize();

  return 0;
}
```

下記は DPC++ で記述された `CalculatePiParallel` 関数です。この関数には、`results` (各計算ユニットの結果の配列のポインター)、`rank_num` (MPI ランクの番号) および `num_procs` (MPI プロセスの数) の 3 つの引数があります。
関数は、各ランクが処理する x 座標の配列 `x_pos_per_rank` を定義します。ホスト上の各 MPI ランクにより実行される各 DPC++ ホストコードは、SYCL アプリケーションを起動します。ホストコードは、データの移動を調整して、デバイスへのオフロードを計算します。

```
void CalculatePiParallel(float* results, int rank_num, int num_procs) {
  char machine_name[MPI_MAX_PROCESSOR_NAME];
  int name_len;
  int num_step = kTotalNumStep / num_procs;
  float* x_pos_per_rank = new float[num_step];
  float dx, dx_2;

  // マシン名を取得。
  MPI_Get_processor_name(machine_name, &name_len);

  dx = 1.0f / (float)kTotalNumStep;
  dx_2 = dx / 2.0f;

  for (size_t i = 0; i < num_step; i++)
    x_pos_per_rank[i] = ((float)rank_num / (float)num_procs) + i * dx + dx_2;
```

try ブロックで、デバイスキューを作成すると DPC++ の機能が開始されます。ホストコードは `q` キューを使用して実行するデバイスにデバイスコードを送ります。この例ではデフォルトセレクターを使用しているため、SYCL ランタイムがシステムで最適なデバイスを選択し、適切なワークを開始します。

```
  default_selector device_selector;
  // 例外ハンドラー
  //
  // exception_list パラメーターは std::exception_ptr オブジェクトの反復可能な
  // リストです。これらのポインターは常に直接読み取り可能であるとは限りません。
  // そのため、ポインターを再スローしてキャッチすると、例外自体が発生します。
  // 注： 操作によっては、いくつかの例外が発生することがあります。
  //
  auto exception_handler = [&](exception_list exceptionList) {
```

```
    for (std::exception_ptr const& e : exceptionList) {
      try {
        std::rethrow_exception(e);
      } catch (cl::sycl::exception const& e) {
        std::cout << "Failure" << std::endl;
        std::terminate();
      }
    }
  };

  try {
    // DPC++ クラスキューを使用してデバイスキューを作成。
    queue q(device_selector, exception_handler);

    cout << "Rank " << rank_num << " of " << num_procs
         << " runs on: " << machine_name
         << ", uses device: " << q.get_device().get_info<info::device::name>()
```

次のステップは、ホストとデバイス間のデータを連動するバッファー・オブジェクトを作成することです。バッファーはメモリーの抽象的な表現を提供します。x_pos_per_rank、results、rank_num および num_procs のデータを格納する x_pos_per_rank_buf、results_buf、ranknum_buf、numprocs_buf を作成します。

```
    // バッファーに与えられるメモリーの量のサイズ。
    range<1> num_items{kTotalNumStep / size_t(num_procs)};

    // ホストとデバイス間で共有するデータをバッファーを使用して SYCL に通知。
    buffer<float, 1> x_pos_per_rank_buf(
        x_pos_per_rank, range<1>(kTotalNumStep / size_t(num_procs)));
    buffer<float, 1> results_buf(results,
                                 range<1>(kTotalNumStep / size_t(num_procs)));
    buffer<int, 1> ranknum_buf(&rank_num, 1);
    buffer<int, 1> numprocs_buf(&num_procs, 1);
```

コマンドグループを上記で作成したキューに送ります。コマンドグループは、カーネルを実行するための要件をすべて含む h ハンドラーを処理します。

バッファーはホストとデバイスにより直接アクセスされないため、バッファーを読み書きするアクセサーを作成します。x_pos_per_rank_accessor、ranknum_accesor、numprocs_accessor アクセサーを作成して、デバイスが results_buf、ranknum_buf、numprocs_buf バッファーを読み取れるようにします。また、results_accessor accessor を作成して、デバイスが results_buf バッファーに書き込めるようにします。

```
    // 実行時に作成されるコマンド・グループ・ハンドラーで渡されるラムダを処理。
    q.submit([&](handler& h) {
      // アクセサーを使用してバッファーが所有しているメモリーにアクセス。
      auto x_pos_per_rank_accessor =
          x_pos_per_rank_buf.get_access<access::mode::read>(h);
      auto results_accessor = results_buf.get_access<access::mode::write>(h);
      auto ranknum_accessor =
          ranknum_buf.template get_access<access::mode::read>(h);
      auto numprocs_accessor =
          numprocs_buf.template get_access<access::mode::read>(h);
```

parallel_for 関数は、デバイスで並列に実行するインスタンスの数を作成する基本的なカーネルを示します。関数には、起動するアイテムの数を指定する num_items と各インデックスで実行するカーネル関数の 2 つの引数があります。各インスタンスは結果の 1 つの値を計算して results_buf バッファーに書き込みます。SYCL ランタイムは、results_buf バッファーが範囲を終了すると結果をホストの results 配列にコピーします。これで各 MPI ランクでの円周率の部分結果の計算は完了です。
最後に、マスター MPI ランクですべての MPI ランクのすべての部分結果を合計します。

```
      // parallel_for を使用して円周率の一部を並列で計算。
      // カーネル関数のインスタンスの数を作成。
      h.parallel_for(num_items, [=](id<1> k) {
        float x, dx;

        dx = 1.0f / (float)kTotalNumStep;
        x = x_pos_per_rank_accessor[k];
        results_accessor[k] = (4.0f * dx) / (1.0f + x * x);
      });
    });
  } catch (...) {
    std::cout << "Failure" << std::endl;
  }

  // クリーンアップ。
  delete[] x_pos_per_rank;
}
```

Linuix* で MPI/DPC++ プログラムをコンパイルして実行する

プログラムは、Linux* および Windows* でコンパイルして実行できます。このセクションでは、Linux* でプログラムをコンパイルして実行する方法を説明します。プログラムのテストには 2 つのインテル® NUC システムを使用しました。これらのシステムにはインテル® Iris® Plus グラフィックスを内蔵したインテル® Core™ i7 プロセッサーが搭載されており、オペレーティング・システムは Ubuntu* 18.04 を使用しました。インテル® Iris® Plus グラフィックスは、インテル® oneAPI ツールキットでサポートされているインテル® プロセッサー・グラフィックスの Gen9 バージョンです。これらのシステムのホスト名は NUC1 と SSEC-HDC02 で、IP アドレスはそれぞれ 10.54.72.150 と 10.23.3.154 です。
次のステップでは、DPC++ で記述された MPI プログラムをコンパイルおよび実行する方法を示します。

1. インテル® C++ コンパイラーとインテル® MPI ライブラリーを含むインテル® oneAPI ベース・ツールキット (英語) およびインテル® oneAPI HPC ツールキット (英語) をインストールします。この記事を記述したときのテストには、インテル® oneAPI Beta06 を使用しました。このコードサンプルは Beta06 以降のバージョンで動作します。
   この例では、両方のマシンでインテル® oneAPI をデフォルトのパス `/opt/intel/` にインストールしました。
2. MPI プログラムを開始するマシンのファイアウォールを無効にします。

   ```
   $ sudo ufw disable
   Firewall stopped and disabled on system startup
   $ sudo ufw status
   Status: inactive
   ```

3. これらの 2 つのマシンでパスワードなしの SSH ログインをセットアップします。
4. oneAPI 環境変数をセットアップします。
   コードサンプルから実行ファイルを生成するには、ユーザーがプログラムを実行するホストで source コマンドを使用して oneAPI スクリプトを実行する必要があります。

   ```
   $ source /opt/intel/inteloneapi/setvars.sh
   ```

5. MPI プログラムをコンパイルします。
   環境変数を設定した後、`mpiicpc` スクリプトを使用して C++ で記述された MPI プログラムをコンパイルおよびリンクします。オプションと MPI プログラムに必要な特別なライブラリーが提供されます。このスクリプトは `icpc` コンパイラー (インテル® oneAPI HPC ツールキットに含まれる **mpicc**) を使用しています。`mpiicpc` は、インテル® C++ コンパイラー向けのインテル® MPI ライブラリー・コンパ

イラー・コマンドです。-show オプションは、使用する C++ コンパイラーと実行時に必要なオプションを示します。

```
$ mpiicpc -show
icpc -I/opt/intel/inteloneapi/mpi/2021.1-beta06/include -
L/opt/intel/inteloneapi/mpi/2021.1-beta06/lib/release -
L/opt/intel/inteloneapi/mpi/2021.1-beta06/lib -Xlinker --enable-new-dtags -Xlinker
-rpath -Xlinker /opt/intel/inteloneapi/mpi/2021.1-beta06/lib/release -Xlinker -
rpath -Xlinker /opt/intel/inteloneapi/mpi/2021.1-beta06/lib -lmpicxx -lmpifort -
lmpi -ldl -lrt -lpthread
```

上記のコマンドは、C++ プログラム向けの mpiicpc スクリプトを起動した場合のコマンドラインを表示します。C++ プログラムのコンパイルとリンクには icpc コンパイラーを使用します。
一方、DPC++ プログラムのコンパイルには、インテル® oneAPI DPC++ コンパイラー dpcpp を使用する必要があります。DPC++ は、SYCL と拡張を含む C++ で構成されます。DPC++ で記述された MPI プログラムをコンパイルしてリンクするには、上記のコマンドの **icpc** を **dpcpp** に置換します。

```
$ dpcpp -I/opt/intel/inteloneapi/mpi/2021.1-beta06/include -
L/opt/intel/inteloneapi/mpi/2021.1-beta06/lib/release -
L/opt/intel/inteloneapi/mpi/2021.1-beta06/lib -Xlinker --enable-new-dtags -Xlinker
-rpath -Xlinker /opt/intel/inteloneapi/mpi/2021.1-beta06/lib/release -Xlinker -
rpath -Xlinker /opt/intel/inteloneapi/mpi/2021.1-beta06/lib -lmpicxx -lmpifort -
lmpi -ldl -lrt -lpthread mpi_dpcpp.cpp -o mpi_dpcpp
```

または、次のコマンドを使用して MPI プログラムをコンパイルします。

```
$ export I_MPI_CXX=dpcpp
$ mpiicpc -lOpenCL -fsycl mpi_dpcpp.cpp -o mpi_dpcpp
```

上記のコマンドは、mpi_dpcpp.cpp を呼び出す DPC++ で記述された MPI プログラムのコンパイル方法を示しています。-o オプションでは、実行ファイルの名前 mpi_dpcpp を指定します。

6. 実行ファイルを別のマシンに転送します。
ホスト (この場合は NUC1) でコマンドを実行するときに、実行ファイルを別のホスト (SSEC-HDC02、IP アドレスは 10.23.3.154) に転送する必要があります。

```
$ scp mpi_dpcpp user@10.23.3.154:~/
```

7. mpirun コマンドを使用して 2 ノードクラスターで MPI 実行ファイルを実行します。
これで、両方のホスト上で実行ファイルを実行できます。-n オプションは、ノードごとの MPI ランクの数を指定します。-host オプションは、MPI ランクを実行するホストを指定します。セミコロン ":" で 2 つのノード (NUC1 および SSEC-HDC02) を区切ります。下記のコマンドは、最初のホスト NUC1 で 1 つの MPI ランクを実行し、2 つ目のホスト SSEC-HDC02 で 1 つの MPI ランクを実行します。各 MPI ランクは、データ並列 C++ を使用して円周率の部分結果を計算します。これにより、SYCL ランタイムはカーネルにオフロードする最適なデバイスを選択できます。このケースでは、SYCL ランタイムは、並列でカーネルを実行するため、両方のシステムで利用可能な GPU を選択します。

```
$ scp mpi_dpcpp user@10.23.3.154:~/
$ mpirun -n 1 -host localhost ./mpi_dpcpp : -n 1 -host 10.23.3.154 ./mpi_dpcpp
Rank 1 of 2 runs on: SSEC-HDC02, uses device: Intel(R) Gen9 HD Graphics NEO
Rank 0 of 2 runs on: NUC1, uses device: Intel(R) Gen9 HD Graphics NEO
Elapsed time from rank 0: 264.635(msec)
Elapsed time from rank 1: 220.89(msec)
---> pi= 3.14166
```

MPI 環境変数 (英語) を設定して実行ファイルを実行できます。例えば、デバッグ情報を出力するには、I_MPI_DEBUG=<level> 環境変数を設定します。

```
$ I_MPI_DEBUG=5 mpirun -n 1 -host localhost ./mpi_dpcpp : -n 1 -host
10.23.3.154 ./mpi_dpcpp
[0] MPI startup(): libfabric version: 1.9.0a1-impi
[0] MPI startup(): libfabric provider: tcp;ofi_rxm
[0] MPI startup(): Rank    Pid      Node name      Pin cpu
[0] MPI startup(): 0       5541     NUC1  {0,1,2,3,4,5,6,7}
[0] MPI startup(): 1       5783     SSEC-HDC02     {0,1,2,3,4,5,6,7}
[0] MPI startup(): I_MPI_ROOT=/opt/intel/inteloneapi/mpi/2021.1-beta06
[0] MPI startup(): I_MPI_MPIRUN=mpirun
[0] MPI startup(): I_MPI_HYDRA_TOPOLIB=hwloc
[0] MPI startup(): I_MPI_INTERNAL_MEM_POLICY=default
[0] MPI startup(): I_MPI_DEBUG=5
Rank 1 of 2 runs on: SSEC-HDC02, uses device: Intel(R) Gen9 HD Graphics NEO
Rank 0 of 2 runs on: NUC1, uses device: Intel(R) Gen9 HD Graphics NEO
Elapsed time from rank 0: 265.085(msec)
Elapsed time from rank 1: 222.215(msec)
---> pi= 3.14166
```

まとめ

DPC++ を使用すると、開発者は、CPU、GPU、FPGA など、複数のハードウェア・ターゲットでコードを再利用することができます。この機能を活用するには、MPI プログラムと DPC++ を組み合わせます。この記事では、インテル® DPC++ コンパイラーを使用して MPI/DPC++ プログラムをコンパイルして実行する方法を説明しました。

関連情報

- インテル® MPI ライブラリー
- インテル® oneAPI データ並列 C++ (英語)
- MPI 環境変数リファレンス (英語)
- インテル® oneAPI ベース・ツールキット (英語)
- インテル® oneAPI HPC ツールキット (英語)
