How to: Write a parallel_for Loop

Visual Studio 2015

This example demonstrates how to use concurrency::parallel_for to compute the product of two matrices.

Example

The following example shows the **matrix_multiply** function, which computes the product of two square matrices.

C++

```
// Computes the product of two square matrices.
void matrix_multiply(double** m1, double** m2, double** result, size_t size)
{
   for (size_t i = 0; i < size; i++)
   {
      for (size_t j = 0; j < size; j++)
      {
         double temp = 0;
         for (int k = 0; k < size; k++)
         {
            temp += m1[i][k] * m2[k][j];
         }
         result[i][j] = temp;
      }
   }
}
```

Example

The following example shows the **parallel_matrix_multiply** function, which uses the **parallel_for** algorithm to perform the outer loop in parallel.

C++

```
// Computes the product of two square matrices in parallel.
void parallel_matrix_multiply(double** m1, double** m2, double** result, size_t size)
{
   parallel_for (size_t(0), size, [&](size_t i)
   {
      for (size_t j = 0; j < size; j++)
      {
         double temp = 0;
         for (int k = 0; k < size; k++)
         {
            temp += m1[i][k] * m2[k][j];
         }
         result[i][j] = temp;
      }
   });
}
```

This example parallelizes the outer loop only because it performs enough work to benefit from the overhead for parallel processing. If you parallelize the inner loop, you will not receive a gain in performance because the small amount of work that the inner loop performs does not overcome the overhead for parallel processing. Therefore, parallelizing the outer loop only is the best way to maximize the benefits of concurrency on most systems.

Example

The following more complete example compares the performance of the **matrix_multiply** function versus the **parallel_matrix_multiply** function.

C++

```
// parallel-matrix-multiply.cpp
// compile with: /EHsc
#include <windows.h>
#include <ppl.h>
#include <iostream>
#include <random>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
   __int64 begin = GetTickCount();
   f();
   return GetTickCount() - begin;
}

// Creates a square matrix with the given number of rows and columns.
double** create_matrix(size_t size);

// Frees the memory that was allocated for the given square matrix.
void destroy_matrix(double** m, size_t size);

// Initializes the given square matrix with values that are generated
// by the given generator function.
template <class Generator>
double** initialize_matrix(double** m, size_t size, Generator& gen);

// Computes the product of two square matrices.
void matrix_multiply(double** m1, double** m2, double** result, size_t size)
{
   for (size_t i = 0; i < size; i++)
   {
      for (size_t j = 0; j < size; j++)
      {
         double temp = 0;
         for (int k = 0; k < size; k++)
         {
            temp += m1[i][k] * m2[k][j];
         }
         result[i][j] = temp;
      }
   }
}

// Computes the product of two square matrices in parallel.
void parallel_matrix_multiply(double** m1, double** m2, double** result, size_t size)
{
   parallel_for (size_t(0), size, [&](size_t i)
   {
      for (size_t j = 0; j < size; j++)
      {
         double temp = 0;
         for (int k = 0; k < size; k++)
         {
            temp += m1[i][k] * m2[k][j];
         }
         result[i][j] = temp;
      }
   });
}

int wmain()
{
   // The number of rows and columns in each matrix.
   // TODO: Change this value to experiment with serial
   // versus parallel performance.
   const size_t size = 750;

   // Create a random number generator.
   mt19937 gen(42);

   // Create and initialize the input matrices and the matrix that
   // holds the result.
```

```
      double** m1 = initialize_matrix(create_matrix(size), size, gen);
      double** m2 = initialize_matrix(create_matrix(size), size, gen);
      double** result = create_matrix(size);

      // Print to the console the time it takes to multiply the
      // matrices serially.
      wcout << L"serial: " << time_call([&] {
         matrix_multiply(m1, m2, result, size);
      }) << endl;

      // Print to the console the time it takes to multiply the
      // matrices in parallel.
      wcout << L"parallel: " << time_call([&] {
         parallel_matrix_multiply(m1, m2, result, size);
      }) << endl;

      // Free the memory that was allocated for the matrices.
      destroy_matrix(m1, size);
      destroy_matrix(m2, size);
      destroy_matrix(result, size);
   }

   // Creates a square matrix with the given number of rows and columns.
   double** create_matrix(size_t size)
   {
      double** m = new double*[size];
      for (size_t i = 0; i < size; ++i)
      {
         m[i] = new double[size];
      }
      return m;
   }

   // Frees the memory that was allocated for the given square matrix.
   void destroy_matrix(double** m, size_t size)
   {
      for (size_t i = 0; i < size; ++i)
      {
         delete[] m[i];
      }
      delete m;
   }

   // Initializes the given square matrix with values that are generated
   // by the given generator function.
   template <class Generator>
   double** initialize_matrix(double** m, size_t size, Generator& gen)
   {
      for (size_t i = 0; i < size; ++i)
      {
         for (size_t j = 0; j < size; ++j)
         {
            m[i][j] = static_cast<double>(gen());
         }
      }
      return m;
   }
```

The following sample output is for a computer that has four processors.

```
serial: 3853
parallel: 1311
```

Compiling the Code

To compile the code, copy it and then paste it in a Visual Studio project, or paste it in a file that is named **parallel-matrix-multiply.cpp** and then run the following command in a Visual Studio Command Prompt window.

cl.exe /EHsc parallel-matrix-multiply.cpp

See Also

Parallel Algorithms
parallel_for Function